ISSN No:-2456-2165

Attacks on Elliptic Curve Cryptography Implementations in Sage Math

Pagilla Manohar Reddy
Amrita Vishwa Vidhyapeetham

Abstract:- This paper presents a summary of the various attacks that have been discovered on the implementation of Elliptic Curve Cryptography (ECC) in SageMath, a widely used open-source mathematics software. ECC is a popular method for providing secure communication and is used in a variety of applications, such as secure key exchange, digital signatures, and more. However, the implementation of ECC in real life has been found to have several vulnerabilities that could potentially be exploited by attackers. This paper will look at the various attacks that have been found and how they affect the safety of ECC-based systems.

Keywords:- *Discrete Log Problem, Elliptic Curve, Cryptography, Pohlig Hellman, Singular Curve, Smart Attack, P-Adic, Hensel Lifting, Weil Pairing, Tate Pairing, Frey-Rück Attack, Supersingular Curves, MOV Attack, ECDLP, Lattice, Nonce, Digital Signatures.*

I. INTRODUCTION TO ELLIPTIC CURVES

A Weierstrass equation is a homogeneous equation, $F(X,Y,Z) = Y^2Z + a_1XYZ + a_3YZ^2 - X^3 - a_2X^2Z - a_6Z^3$, where a_1, a_2, a_3, a_4, a_6 are constants which belong to its algebraic closure. An elliptic curve E is defined to be a set of solutions to $F(X,Y,Z)$=0 in the projection of a plane.

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 Fp and Fq , 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 curves which can be simplified when p = 2 or 3; they will have different properties but we will not be dealing with these cases here.

II. GROUP LAWS & POINT ADDITION

It is well known that an abelian group is formed when certain additions are made to the points on an elliptic curve. Give the Weierstrass equation as the base of the elliptic curve E.The addition rules are given below:

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.

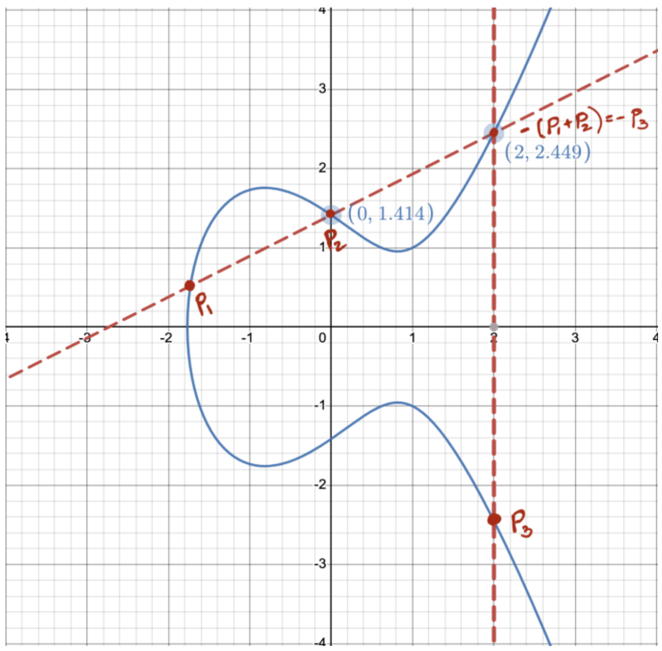


Fig 1 Shows Addition of Two Points on an Elliptic Curve of the form $\underline{E: y^2 = x^3 + Ax + B}$

Let l be the line passing through P_1, P_2 where $P_1, P_2 \in E$, then $P_3 = P_1 + P_2$.

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)

III. POINT MULTIPLICATION

Point multiplication in elliptic curve cryptography (ECC) is the process of multiplying a point on an elliptic curve by a scalar value. This is typically done using the double and add algorithm, which involves repeatedly doubling a point and then adding it to itself, using the scalar value to determine the number of times the point should be doubled and added.

Given an elliptic curve and a point P on that curve, the algorithm repeatedly doubles the point P and adds it to itself, using a specific scalar value (k) to determine the number of times the point should be doubled and added.

- *Mathematically, the Double and Add Algorithm can be Represented as follows:*

$$P = P + O,$$

$$2P = P + P,$$

$$4P = 2P + 2P,$$

$$8P = 4P + 4P,$$

In general we can represent as,

$$kP = 2^{k-1}P + 2^{k-2}P + 2^{k-3}P + \ldots\ldots + 2P + P.$$

The double and add algorithm is efficient because it uses a small number of point doublings and a minimal number of point additions. This is because the elliptic curve group operation is distributive over addition. The running time of this algorithm is proportional to the number of bits in the scalar value (k) being used.

- *The mathematical algorithm for the "double and add" method in Elliptic Curve Cryptography (ECC) is as follows:*

Given an elliptic curve defined by the equation $\underline{y^2 = x^3 + Ax + B}$ and a point P(x_1, y_1) on that curve, the double and add algorithm can be used to calculate kP for any scalar value k:

- *Initialize a variable "result" to the point at infinity (also known as the "neutral element" of the group)*
- *Convert the scalar value k to binary form, and for each bit in the binary representation of k:*

  - ✓ *If the current bit is 0, do nothing*
  - ✓ *If the current bit is 1, add the point P to the result*
  - ✓ *Double the point P*

- *Return the Result as kP*

Note that the point addition in step 2.b is done using the group operation defined by the elliptic curve, which is typically a different operation than standard coordinate addition.

It is also important to note that in practice, the scalar value k is often chosen to be a large randomly generated number, and the point P is also chosen to be a generator point of the elliptic curve group.

IV. SAGE IMPLEMENTATION OF ELLIPTIC CURVES

- *Construct an Elliptic Curve.*

```
sage: p = 1307
sage: a = 13
sage: b = 7
sage: E = EllipticCurve(GF(p), [a,b])
Elliptic Curve defined by y^2 = x^3 + 13*x + 7
over Finite Field of size 1307
```

- *Point Multiplication And Addition*

```
sage: G = E.gens()[0] #G is a point on the curve
(generator)
sage: G
(314 : 462 : 1)
sage: P = 3*G
sage: Q = G+G+G
sage: P == Q
True
```

➢ *General Usage*

```
sage: E.lift_x(13) #Return one or all points with
given x-coordinate.
(13 : 829 : 1)

sage: E.order() #Return the number of points on this
elliptic curve.
1344
sage: E.gens() #Return points which generate the
abelian group of points on EC
((314 : 462 : 1),)
sage: E.lift_x(13) #Return one or all points with
given x-coordinate.
(13 : 829 : 1)
```

V. DISCRETE LOGARITHM PROBLEM

The Discrete Logarithm Problem (DLP) is a mathematical problem that is central to many cryptographic systems, including those based on Elliptic Curve Cryptography (ECC) and the Diffie-Hellman key exchange.

The DLP is defined as follows: given a prime number p, a generator g, and an element h in the group $\mathbb{Z}_p^*$ (the group of integers modulo p), find an integer x such that $g^x \equiv h \bmod p$. In other words, the DLP is the problem of finding the exponent x when the base g and the result h are known.

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 $g^x \bmod p$. They then exchange the results ($g^x \bmod p$) 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.

VI. ATTACKS ON DISCRETE LOGARITHM PROBLEM

➢ *There are Several Known Attacks on the Discrete Logarithm Problem (DLP) that can be used to Solve it in Certain Cases. The Most Common Attacks Are:*

- Brute force attack: This is the most straightforward attack, where an attacker simply tries every possible value of x until the correct one is found. However, this is infeasible for large values of x because the number of possible values is exponential in the size of x.
- Baby-step giant-step attack: This is a more efficient attack that reduces the search space for x by dividing it into two parts: a *"baby step"* part and a *"giant step"* part. The attacker first computes a table of all possible values of g^{i+j} for small values of i and j, and then uses this table to search for a match with h.
- Pollard's rho attack: This is a more sophisticated attack that uses mathematical properties of the group to reduce the search space for x. It is based on the observation that if $g^x \equiv h \bmod p$, then $g^{x+y} \equiv g^x * g^y \bmod p$ for any integers x and y. The attack uses this property to find a collision between two different values of x and y that result in the same value of h.
- Index calculus: This is a class of algorithms that allow to find discrete logarithms in a group whose order is a large composite number. The general idea is to find a set of relations between logarithms of group elements and then solving a system of linear equations in order to find the discrete logarithm.
- Pohlig-Hellman: This is a special case of index calculus, which is applied when the order of the group is a power of a prime. The algorithm is based on solving a series of discrete logarithm problems, each modulo a prime power, and then combining the solutions.

It's important to note that these attacks are not always applicable, and the security of DLP depends on the group it is defined on and the parameters used. Also, many cryptographic systems use large prime numbers and generator values to make the DLP infeasible to solve, even with the above attacks.

➢ *Brute Force Attack*

A brute force attack on the Discrete Logarithm Problem (DLP) in Elliptic Curve Cryptography (ECC) involves trying every possible value of the secret exponent (x) until the correct one is found. The secret exponent is used to calculate the public key by point multiplication of a generator point with the secret exponent.

- *The Algorithm Works as follows:*

✓ *Define an elliptic curve over a finite field with a prime order (p).*
✓ *Choose a generator point G on the curve*
✓ *Choose a target point Q, which is the result of point multiplication of G by a secret scalar k*
✓ *Initialize a variable to store the result (x)*
✓ *Loop through all possible exponents from 1 to p-1*

- *For Each Value of the Exponent:*

- ✓ *Calculate Q'=x*G*
- ✓ *Compare Q' with Q*
- ✓ *If Q'=Q, store the value of x and break the loop*
- ✓ *If a solution is found return x, otherwise return "No solution found"*

➢ *Sage Implementation for Brute-Force Attacks as follows:*

```
a = 13; b = 7; p = 1307;
E = EllipticCurve(GF(p), [a,b])
# Define the generator point & the target point
P = E.gens()[0]; Q = 137*P
# Loop through all possible exponents(brute force)
x = next((x for x in range(1,p) if x*P == Q), None)
if x:
    print("The DL of Q with respect to P is:", x)
else: print("No discrete logarithm found")
```

But, It's important to note that this method is very slow and infeasible for large values of p and k, the time complexity of a brute force attack is $O(p)$ and for a 256-bit prime modulus it would take on the order of 2^{256} operations. Furthermore, it's not practical to use this method for real-world scenarios. For example, for a 256-bit prime modulus it would take longer than the age of the universe to perform a brute force attack.

In practice, more efficient algorithms such as the baby-step giant-step attack, Pollard's rho attack, index calculus, and Pohlig-Hellman should be used to solve the DLP.

➢ *Baby-Step Giant-Step*

The baby-step giant-step attack is a more efficient method for solving the Discrete Logarithm Problem (DLP) in Elliptic Curve Cryptography (ECC). It reduces the search space for the secret exponent (x) by dividing it into two parts: a "baby step" part and a "giant step" part.

- *The Algorithm Works as follows:*

- ✓ *Define an elliptic curve over a finite field with a prime order p.*
- ✓ *Choose a generator point G on the curve*
- ✓ *Choose a target point Q, which is the result of point multiplication of G by a secret scalar k*
- ✓ *Calculate $m = \sqrt{p}$ and initialize empty hash tables H and T*
- ✓ *For i in 0 to $m-1$:*

  - ▪ *Compute $T[i] = i * G$ & $H[T[i]] = i$*

- ✓ *For j in 0 to $m-1$:*

  - ▪ *Compute $Q' = j * m * G$*
  - ▪ *if Q' exists in H table, compute $x = j * m - H[Q']$ & check$G^x = Q \bmod p$*

- ✓ *If a Solution is Found Return x, Otherwise Return "No Solution Found"*

```
sage: from sage.groups.generic import bsgs
sage: F.<a> = GF(1307)
sage: E = EllipticCurve(F, [13,7])
sage: P = E.lift_x(a); P
(1 : 844 : 1)
sage:
bsgs(P,P.parent()(0),Hasse_bounds(F.order()),operati
on='+')
1344
```

It's important to note that the attack requires $O(\sqrt{p})$ storage space to store the hash table, and the time complexity of the attack is dependent on the hash function used, and the collisions. Furthermore, the attack can be improved by using a more efficient hash function, and by using more advanced data structures such as a binary search tree.

It's also important to note that this attack, as well as other known attacks on DLP, relies on the difficulty of solving the DLP in the group in which the problem is defined. For example, for a group defined over a prime order field, the DLP is considered hard, but for a group defined over a composite order field, it can be broken by solving multiple DLP instances over prime order subgroups.

➢ *Pollard's Rho Attack*

Pollard's rho attack algorithm for the Discrete Logarithm Problem (DLP) is a probabilistic algorithm that can be applied to different groups, not only elliptic curves. The basic idea of the attack is to find a collision between two different values of x and y that result in the same value of h.

- *The Algorithm Works as follows:*

- ✓ *Choose a generator g and a target element h in the group G*
- ✓ *Choose two random elements x_1 and x_2 in the group G*
- ✓ *Set $y_1 = g^{x1}$ and $y_2 = g^{x2}$*
- ✓ *Initialize two variables i and j to 0*

- *Repeat the following steps until a collision is found:*

- ✓ i = i + 1, $y_1 = y_1 * g^{(f(i))}$
- ✓ j = j + 1, $y_2 = y_2 * g^{(f(j))}$
- ✓ if $y_2 = y_1$, calculate
- ✓ $x = i - j/((f(j)) - (f(j)))$ and check if $g^x = h$
- ✓ *If a solution is found return x, otherwise return "No solution found"*

```
def Pollard_rho_EC(P,Q):
    order=P.order()
    i_1=[P,1,0]
    i_2=[P,1,0]
    matr=[]
    def prw(n):
        if ZZ(n[0][1])%3==0:
            return [P+n[0],(n[1]+1)%order,n[2]]
        elif ZZ(n[0][1])%3==1:
            return
[2*n[0],(2*n[1])%order,(2*n[2])%order]
        else:
            return [Q+n[0],n[1],(n[2]+1)%order]
    jo=True
    while jo:
        i_1=prw(i_1)
        i_2=prw(prw(i_2))
        matr.append([i_1,i_2])
        if i_1[0]==i_2[0]:
            jo=False
    x1=(i_1[1]-i_2[1])%order
    y1=(i_2[2]-i_1[2])%order
    g,n,s=xgcd(order,y1)
    sol=[ZZ((s*x1+k*order)/g) for k in [0..g-1]]
    sol1=[k for k in sol if k*P==Q]
    return sol1[0]

E=EllipticCurve(GF(743),[11,101])
P=E.gens()[0]
Q=222*P
Pollard_rho_EC(P,Q)
```

In summary, while Pollard's rho method is a more efficient algorithm than a brute force attack, it still has some disadvantages and it's not always the best choice for solving the DLP. It's important to evaluate the specific requirements of the application and the group in question, to choose the best algorithm to solve the DLP.

➢ *Index Calculus*

Index calculus is a method for solving the Discrete Logarithm Problem (DLP) in general, it can be applied to different groups, not only elliptic curves. The basic idea of the algorithm is to find relations between logarithms of different elements in the group by using algebraic techniques.

- *The Algorithm can be Divided into Two Main Steps:*

✓ Factorization of the group order: The first step of the algorithm is to factorize the group order, which is the number of elements in the group. This is done by finding the prime factors of the group order.
✓ Finding relations between logarithms: Once the group order has been factored, the algorithm proceeds by finding relations between logarithms of different elements in the group. This is done by using a technique called "index calculus", which involves solving a system of polynomial equations.

The idea behind index calculus is to find relations between logarithms of elements in the group by considering the group operation in the form of a polynomial equation.

This means that if $g^x \equiv h \bmod p$, then $g^{x+y} \equiv g^x * g^y \bmod p$ for any integers x and y. Therefore, the algorithm uses this property to find a collision between two different values of x and y that result in the same value of h. After relations are found, the algorithm uses the Chinese remainder theorem to combine them and find the solution for DLP.

```
def IndexCalculus(x,b,p,B):
    s=Set([0..p-2])
    primeDiv=prime_divisors(p-1)
    T=True
    M,V=[]
    dimM=0
    while T:
        xi=s.random_element()
        s=s.difference(Set([xi]))
        q=[valuation((x^xi)%p,k) for k in B]
        if prod([k^valuation((x^xi)%p,k) for k in
B])==(x^xi)%p:
            M0=M+[q]
            if all(matrix(GF(x),M0).rank()==dimM+1
for x in primeDiv):
                M.append(q)
                V.append(xi)
                dimM=dimM+1
        if dimM==len(B):
            T=False
    lgs=matrix(Integers(p-1),M).inverse()*vector(V)
    s=Set([0..p-2])
    T=True
    while T:
        xi=s.random_element()
        s=s.difference(Set([xi]))
        q=[valuation((x^xi*b)%p,k) for k in B]
        if prod([k^valuation((x^xi*b)%p,k) for k in
B])==(x^xi*b)%p:
            DL=(sum(q[k]*lgs[k] for k in
[0..len(q)-1])-xi)%(p-1)
            return DL
```

It's important to note that the index calculus algorithm is not efficient for curves defined over a prime field and it's considered to be less efficient than the Pollard's rho algorithm for solving DLP in elliptic curve groups.

➢ *Pohlig-Hellman*

The Pohlig-Hellman algorithm is a method for solving the Discrete Logarithm Problem (DLP) on an elliptic curve. It is based on the Chinese Remainder Theorem (CRT) and the idea of reducing the problem to smaller subproblems.

The basic idea of the algorithm is to find the discrete logarithm of a point P with respect to a generator point G, by solving a series of subproblems, each one corresponding to a prime factor of the order of the subgroup generated by G.

- *Here are the Details of each step:*

✓ *Factorize the order of the subgroup generated by G: Let n be the order of the subgroup generated by G, we factorize n into prime factors $n = {p_1}^{e_1} * {p_2}^{e_2} * \ldots * {p_k}^{e_k}$*

✓ *For each prime factor ${p_i}^{e_i}$, find the discrete logarithm of P with respect to G in the subgroup of order ${p_i}^{e_i}$.*

▪ *Define a new point $P' = P^{(n/{p_i}^{e_i})}$*
▪ *Define a new generator point $G' = G^{(n/{p_i}^{e_i})}$*
▪ *Compute x_i = discrete log of P' with respect to G' mod ${p_i}^{e_i}$*

- *Use the Chinese Remainder Theorem (CRT) to combine the solutions of the subproblems:*

✓ *for each i from 1 to k, we can find the unique solution x mod n by using the CRT as following*

$$x = \sum_{i=1}^{k} x_i y_i y_i' \bmod n$$

Where,

$$y_i = \frac{n}{{p_i}^{e_i}} \;\&\; y_i' = {y_i}^{-1} mod\, {p_i}^{e_i}$$

- *The Pohlig-Hellman algorithm has several advantages when applied to the Elliptic Curve Discrete Logarithm Problem (ECDLP):*

✓ Reduced Complexity: The Pohlig-Hellman algorithm reduces the ECDLP to a series of smaller subproblems, each one corresponding to a prime factor of the group order. This reduces the overall complexity of the problem, making the algorithm more efficient than a brute force attack.

✓ Parallelizability: Because the algorithm solves a series of subproblems independently, it can be parallelized, allowing for faster computations.

✓ Flexibility: The Pohlig-Hellman algorithm can be used with any method that can solve the DLP in subgroups of prime order, such as Baby-Step Giant-Step or Pollard's rho. This allows for flexibility in choosing the most efficient method for a given set of parameters.

✓ Provable security: Pohlig-Hellman algorithm is based on the number theory, and the security proof is based on the difficulty of factoring the order of the subgroup.

✓ Practicality: Pohlig-Hellman algorithm is widely used in practice, it's implemented in many libraries and it's considered to be one of the most efficient classical algorithms for solving the ECDLP.

```
def pohling_hellman_dlog(G,Q):
    n=G.order()
    fac=list(factor(n))
    m=[];r=[]
    for i,j in fac:
        mod=i**j
        g=G*(n//mod);q=Q*(n//mod)
        dl=g.discrete_log(q)
        m.append(mod);r.append(dl)
    dlog=CRT_list(r,m)
    return dlog

def pohling_hellman(G,Q):
    n=G.order()
    fac=list(factor(n))
    m=[];r=[]
    for i,j in fac:
        mod=i**j
        g=G*(n//mod);q=Q*(n//mod)
        dl=discrete_log(q,g,operation="+")
        m.append(mod);r.append(dl)
    dlog=CRT_list(r,m)
    return dlog
```

Limited group order: The Pohlig-Hellman algorithm requires the group order to be factorizable into small prime factors, which may not always be the case. The algorithm is less efficient when the group order is a large composite number, or when it is a prime number, in which case the algorithm is not efficient.

Priv key should be less than the group order: The Pohlig-Hellman algorithm is based on the Chinese Remainder Theorem (CRT), which requires the discrete logarithm d to be less than the group order n.

Large k: The Pohlig-Hellman algorithm requires solving a subproblem for each prime factor of the group order. This means that if the group order has a large number of prime factors, the algorithm can become computationally expensive. In such cases we can avoid or neglect the large factors which are automatically taken care of by CRT.

VII. SMART'S ATTACK

For an elliptic curve E over a field F_p, 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).

A. *Hensel's Lifting*

Hensel's lifting is a concept in number theory that allows the lift of a solution of an equation modulo a prime power to a solution modulo a higher power of the same prime. The basic idea is that if you have an approximate solution of an equation modulo a prime power, it is possible to transform it into an exact solution modulo a higher power of the same prime. $f(x) \equiv 0 (mod\ p)\ and\ for\ x'\ such\ that\ f(x') \equiv 0 (mod\ p^2)$

& $x' \equiv x\ (mod\ p)$, this can be achieved by hensel's lemma.

B. *P-adic Numbers & Curve reduction*

P-adic numbers are a generalization of the concept of real numbers. They are a non-archimedean extension of the rational numbers, and are used in many areas of mathematics, including algebraic number theory and algebraic geometry.

In general x p-adic number is represented $x = a_n * p^n + a_{n-1} * p^{n-1} + \ldots + a_1 * p + a_0$ where p is a prime number and a_i are integers such that $0 \leq a_i < p$. This absolute value is different from the usual absolute value of real numbers, and it satisfies the ultrametric inequality, $|x + y|_p \leq max(|x|_p, |y|_p)$ for any two p-adic numbers x,y.

Curve reduction is a technique used in elliptic curve cryptography (ECC) to reduce the size of an elliptic curve modulo a prime number P. The process is used to reduce the size of the coefficients of the elliptic curve equation, making the calculations more efficient.

The basic idea of curve reduction is to take an elliptic curve defined over a finite field with a large characteristic and reduce it modulo a prime number P. This is done by taking each coefficient of the elliptic curve equation and reducing it modulo P.

➢ *Here is an Example of how Curve Reduction Works:*

Consider the elliptic curve $\underline{E: y^2 = x^3 + Ax + B}$defined over a finite field F_p^m where p^m is a large prime number. To reduce the curve modulo P, we take each coefficient a and b and reduce them modulo P. The resulting curve is $\underline{E' : y^2 = x^3 + (A\ mod\ p)x + (B\ mod\ p)}$

The new curve E' is defined over the finite field F_p, which is much smaller than F_p^m , making the calculations more efficient.

C. *The Attack*

The P-adic elliptic logarithm is an extension of the discrete logarithm problem (DLP) in elliptic curve cryptography (ECC) to the realm of P-adic numbers.

The P-adic elliptic logarithm is defined as the unique integer d such that P = dG, where P is a point on an elliptic curve, G is a generator point of the curve and d is an integer.

```
def Hlift(E, P, gf):
    x, y = map(ZZ, P.xy())
    for p in E.lift_x(x, all=True):
        xx, yy = map(gf, p.xy())
        if y == yy:
            return p

def attack(G, P):
    E = G.curve()
    gf = E.base_ring()
    p = gf.order()
assert E.order() == p

Eqq = E.change_ring(QQ)
Eqp = Eqq.change_ring(Qp(p))
G = p*Hlift(Eqp, G, gf)
P = p*Hlift(Eqp, P, gf)
G_x, G_y = G.xy()
P_x, P_y = P.xy()
return int(gf((P_x / P_y) / (G_x / G_y)))
```

➢ *The Attack is Performed as follows:*

- *The function takes in two points G and P on an elliptic curve E, and the base field gf of the curve.*
- *The x and y coordinates of the point P are mapped to integers using the ZZ function.*
- *The function Hlift(hensel lifting) is called to find the point P on the elliptic curve E in the ring of p-adic numbers.*
- *The order of the curve and the order of the base field are checked to make sure they are equal.*
- *The curve E is changed to a rational field and then to a field of p-adic numbers, and the points G and P are scaled by a factor of p.*
- *The x and y coordinates of the points G and P are obtained, and the ratio of the x and y coordinates of P and G is calculated.*
- *The result of the calculation is cast to an integer and returned as the solution to the ECDLP.*

This attack relies on the specific properties of curves that have an order equal to the base field, as well as the trace of Frobenius or the cardinality of the curve equal to the base field. The attack is based on the idea of reducing the problem to solving a system of polynomial equations, and these specific properties of the curve allow for an efficient reduction.

VIII. SINGULAR CURVE ATTACKS

A. *Singular Curve*

A singular curve in elliptic curve cryptography (ECC) is a curve that does not have a unique group structure, meaning that it does not have a unique set of points that can be used for encryption and decryption.

A curve is considered singular if its discriminant (the value of the discriminant of the equation that defines the curve) is equal to zero. This means that the curve has a double root, which can lead to non-unique solutions when solving the elliptic curve discrete logarithm problem (ECDLP).

When an elliptic curve is said to be singular then its discriminant is “zero”, which indirectly implies isomorphic to the multiplicative group, which enables it to solve the Discrete Logarithm Problem faster than usual.

$$\triangle = -16(4A^3 + 27B^2) \; mod \; p \; \equiv \; 0$$

In singular curves it must satisfy the above property. There are different methods to check for singularity of a curve, one of them is to check the discriminant, and another one is to check the height of the curve, which is a measure of the complexity of the curve. If the height of the curve is zero, it means that it is a singular curve.

B. *The Attack*

An elliptic curve ***E*** with $\triangle = 0$, might possibly have double or triple roots x_0 then the point $(x_0, 0)$ is mentioned as a singular point.

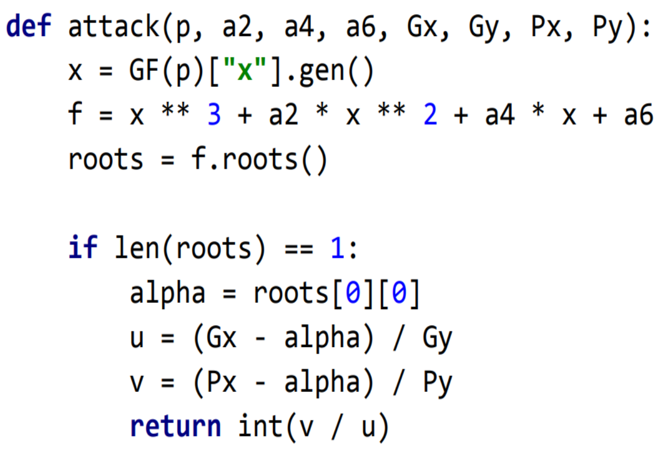
➢ *Case 1:*

$$\text{Cusp } (y^2 = x^3)$$

A singular curve in an elliptic curve can also have a cusp, which is a point on the curve where the slope of the curve becomes infinite. Cusps are also known as "tangent points" or "points of inflection".

A cusp on a curve can be caused by a double root in the equation defining the curve. This can lead to non-unique solutions when solving the elliptic curve discrete logarithm problem (ECDLP) as the cusp point can be mistaken for another point on the curve.

Cusps can be detected by checking the y-coordinate of the point; if it's zero, it means that it's a cusp point. The point is considered to be on a cusp if the x-coordinate is a rational number and y-coordinate is zero, otherwise it's considered to be at infinity.



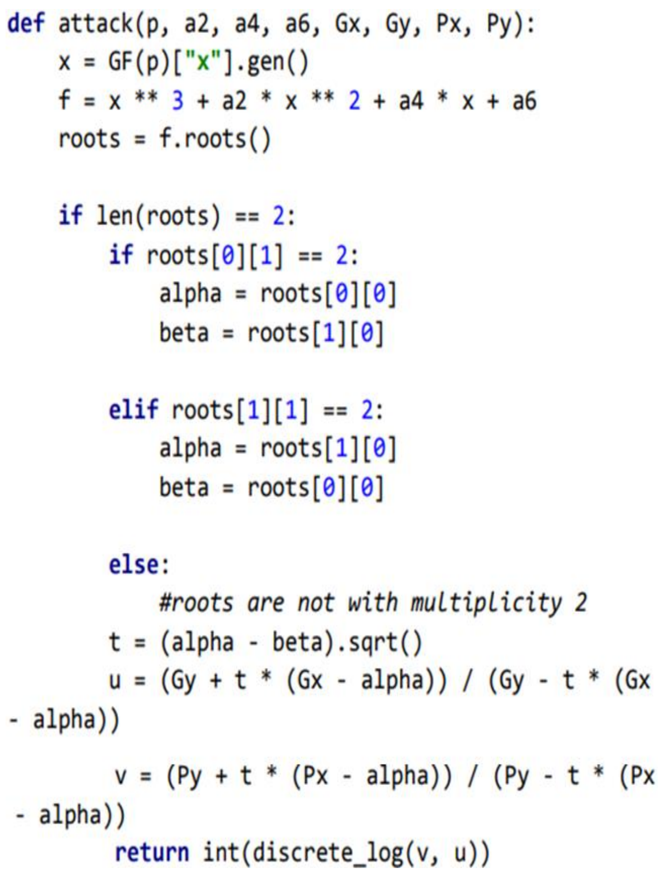
```
def attack(p, a2, a4, a6, Gx, Gy, Px, Py):
    x = GF(p)["x"].gen()
    f = x ** 3 + a2 * x ** 2 + a4 * x + a6
    roots = f.roots()

    if len(roots) == 1:
        alpha = roots[0][0]
        u = (Gx - alpha) / Gy
        v = (Px - alpha) / Py
        return int(v / u)
```

➢ *Case 2:*

$$\text{Node } (y^2 = x^2 * (x - 1))$$

A singular curve in an elliptic curve can also have a node, which is a point on the curve where the curve intersects itself. Nodes are also known as "double points" or "self-intersections"

Nodes can be detected by checking the x-coordinate of the point; if it's equal to the x-coordinate of another point on the curve, it means that it's a node point.



```
def attack(p, a2, a4, a6, Gx, Gy, Px, Py):
    x = GF(p)["x"].gen()
    f = x ** 3 + a2 * x ** 2 + a4 * x + a6
    roots = f.roots()

    if len(roots) == 2:
        if roots[0][1] == 2:
            alpha = roots[0][0]
            beta = roots[1][0]

        elif roots[1][1] == 2:
            alpha = roots[1][0]
            beta = roots[0][0]

        else:
            #roots are not with multiplicity 2
        t = (alpha - beta).sqrt()
        u = (Gy + t * (Gx - alpha)) / (Gy - t * (Gx
- alpha))

        v = (Py + t * (Px - alpha)) / (Py - t * (Px
- alpha))
        return int(discrete_log(v, u))
```

IX. MOV ATTACK

The MOV attack is an acronym that stands for "Miyaji, Ohgishi, and Veselov", the three researchers who first proposed this attack in a paper they published in 2001.

The problem of finding the integer k such that $G * k = P$, where G is a known point on the elliptic curve called the generator or base point, P is an arbitrary point on the curve and k is the private key. The goal of the MOV attack is to find this integer k efficiently by exploiting algebraic properties of the elliptic curve.

A. *Weil Pairing*

Weil pairing is a mathematical function to create a one-way function. It takes two points on an elliptic curve and maps them to a value in a finite field. It's based on the idea of taking the dot product of two points on the curve. It's mainly used to achieve various cryptographic primitives such as Identity-based Encryption, Short Signature and Authentication. It's only defined on a special type of elliptic curve called "pairing-friendly"curves.

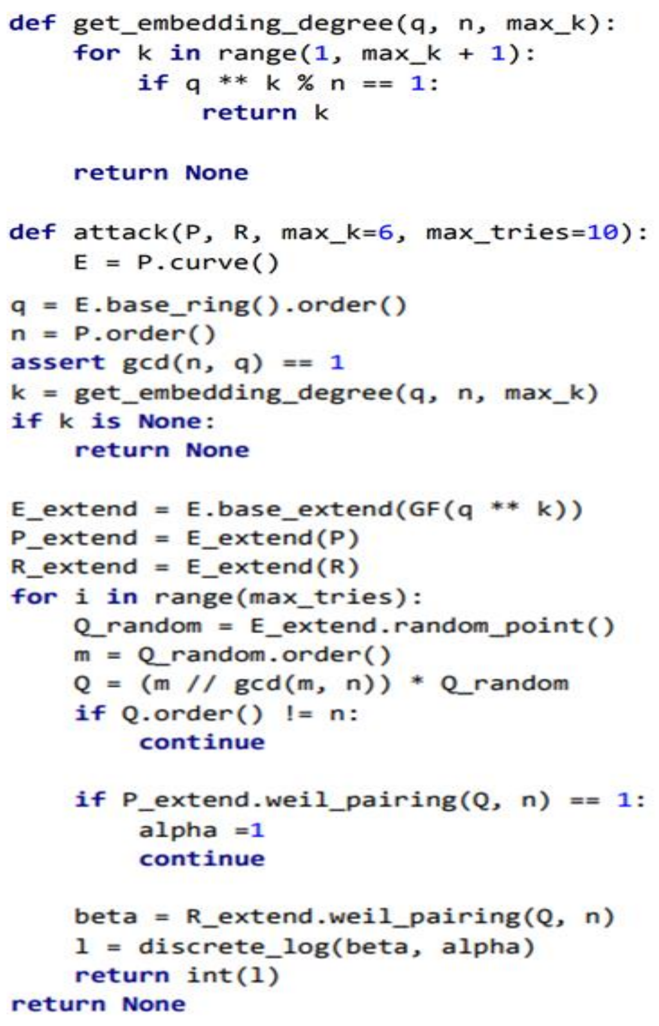
Let elliptic curve E is defined over the field K and n be an integer where K is coprime to n such that $E[n] \subseteq E[K]$. Then we can tell that, the Weil pairing is the mapping $e_n: E[n]xE[n] \rightarrow \mu_n$.

Given that $T \in E[n]$, there exist a function f such that $div(f) = n[T] - n[\infty]$. Then choose $T \in E[n^2]$ with $nT = T$, there exist g such that $div(g) = \sum_{R\in E[n]} ([T + R] - [R])$. For $S \in E[n], P \in E[K□]$, then $g(P+S)^n = f[n(P+S)]$=$f(nP) = g(P)^n$.Thus $\frac{g(P+S)}{g(P)} \in \mu_n$and $\frac{g(P+S)}{g(P)}$ does not depend on P.

Hence, the Weil Pairing is $e_n(S,T) = \frac{g(P+S)}{g(P)}$.

B. *The Attack*

Suppose the points P, G where $P = k * G$. Let $w()$be the Weil Pairing. Let O be the order of G and P, G are linearly independent. The $w(G,P)$ and $w(kG,P)$=$w(G,P)^k$is calculated in the field K. Since we know that G, P are linearly independent and $w(G,P) \neq 1$ does not hold many points by Weil Pairing. We can reduce k by the discrete problem on the finite elliptic curve.


```
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, R, max_k=6, max_tries=10):
    E = P.curve()
q = E.base_ring().order()
n = P.order()
assert gcd(n, q) == 1
k = get_embedding_degree(q, n, max_k)
if k is None:
    return None

E_extend = E.base_extend(GF(q ** k))
P_extend = E_extend(P)
R_extend = E_extend(R)
for i in range(max_tries):
    Q_random = E_extend.random_point()
    m = Q_random.order()
    Q = (m // gcd(m, n)) * Q_random
    if Q.order() != n:
        continue

    if P_extend.weil_pairing(Q, n) == 1:
        alpha =1
        continue

    beta = R_extend.weil_pairing(Q, n)
    l = discrete_log(beta, alpha)
    return int(l)
return None
```

X. FR REDUCTION

The technique is based on the work of Gerhard Frey and Ernst Rück, and it is a type of algebraic attack.The Frey-Rück attack can also be applied to the Tate pairing which is a specific type of Weil pairing. The Tate pairing is a pairing function that is defined on a specific set of elliptic curves called supersingular elliptic curves. The basic idea of the Frey-Rück attack on Tate pairing is to find a rational point on the curve, and then use it to construct an algebraic equation that relates the coordinates of a point on the curve to the discrete logarithm of the point. By solving this equation, the attacker can obtain the discrete logarithm of the point.

D. *Tate Pairing*

Suppose we choose a p of prime order, and an elliptic curve E in the field F_p has m points in it, let q be the order of the elliptic curve q where q^2not divisible by m. In other words we can say that for the subgroup of P we have s security multiplier α, for some integer alpha>0, if the order of p in F_q^* is α.

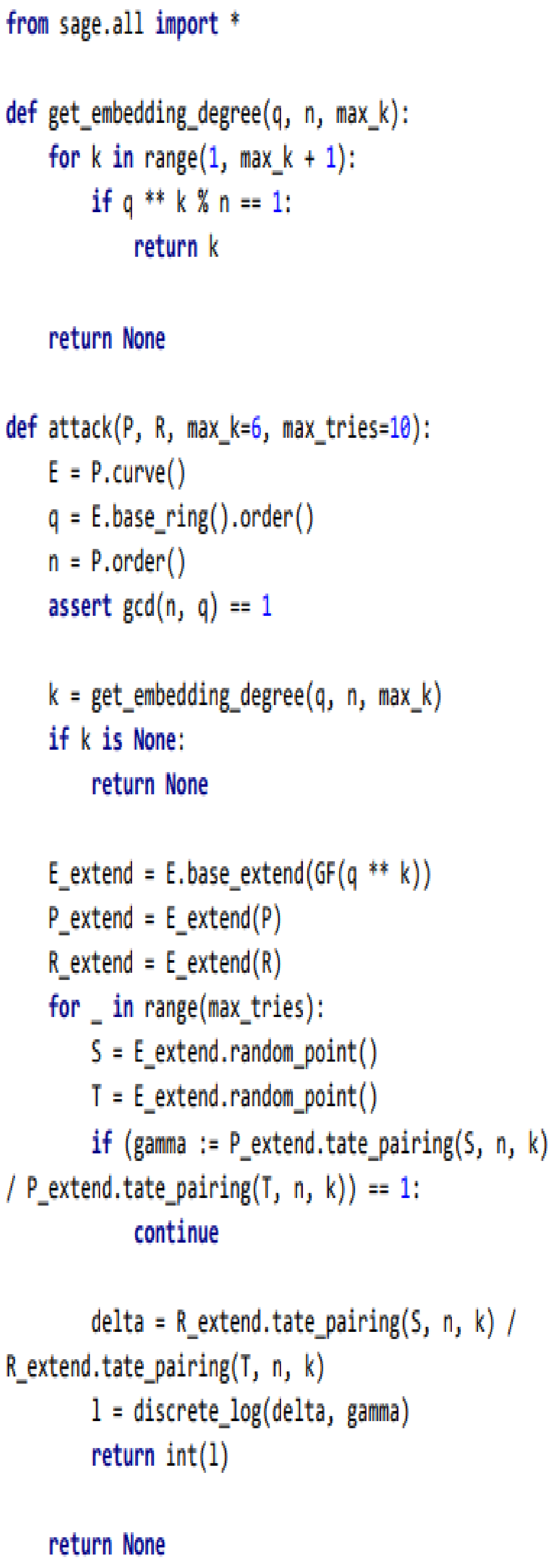
$p^\alpha - 1 \, mod \, q \; \equiv 0 \; and \; p^k - 1 \, mod \, q \; \neq \; 0$ for all k=1, 2, 3, …, α-1 this security multiplier is the security multiplier of the largest prime order subgroup of $E(F_p)$.

Let an elliptic curve E defined over the prime field P. Let n be an integer so that $n|(q-1)$ where q is multiplicative order of extensive field E. The points of E on the field P of n denoted by $E(F_p)[n]$ is dividing order, and let $\mu = \{x \in F_p | \, x^n = 1\}$. Assume $E(F_p)$ contains an element of order n. Then, there exists a non degenerate bilinear mapping.

So $<.,.>_n : E(F_p)[n] \times E(F_p)/nE(F_p) \rightarrow F_p^\times/(F_p^\times)^n$ is called the Tate-Lichtenbaum pairing. And $\tau_n: E(F_p)[n] \times E(F_p)/nE(F_p)$ where τ_n is the modified Tate-Lichtenbaum pairing.

E. *The Attack*

Similar to the MOV-attack, the FR-attack is based on FR-reduction and tate pairing. It works in reducing the discrete logarithm problem on the elliptic curve E over a prime field p to the multiplicative order of extensive field q^k of embedded degree k. This reduction is due to Tate pairing instead of Weil pairing unlike MOV-Attack.



```
from sage.all import *

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, R, max_k=6, max_tries=10):
    E = P.curve()
    q = E.base_ring().order()
    n = P.order()
    assert gcd(n, q) == 1

    k = get_embedding_degree(q, n, max_k)
    if k is None:
        return None

    E_extend = E.base_extend(GF(q ** k))
    P_extend = E_extend(P)
    R_extend = E_extend(R)
    for _ in range(max_tries):
        S = E_extend.random_point()
        T = E_extend.random_point()
        if (gamma := P_extend.tate_pairing(S, n, k)
/ P_extend.tate_pairing(T, n, k)) == 1:
            continue

        delta = R_extend.tate_pairing(S, n, k) /
R_extend.tate_pairing(T, n, k)
        l = discrete_log(delta, gamma)
        return int(l)

    return None
```

XI. LATTICE BASED SIGNING ATTACK

F. *Signing*

Signing in elliptic curve cryptography is a process of generating a digital signature for a message or data. The main idea behind elliptic curve signing is to use the properties of the elliptic curve to produce a signature that is secure and difficult to forge.

➤ *The Process of Signing Involves the following Steps:*

- *The message to be signed is hashed to produce a unique representation of the message.*
- *The signer generates a private key which is used to produce a digital signature for the message.*
- *The private key is used in combination with the hash of the message to produce a digital signature.*
- *The signature is verified using the public key of the signer and the original message.*

The security of the signature depends on the security of the underlying elliptic curve, as well as the security of the private key.

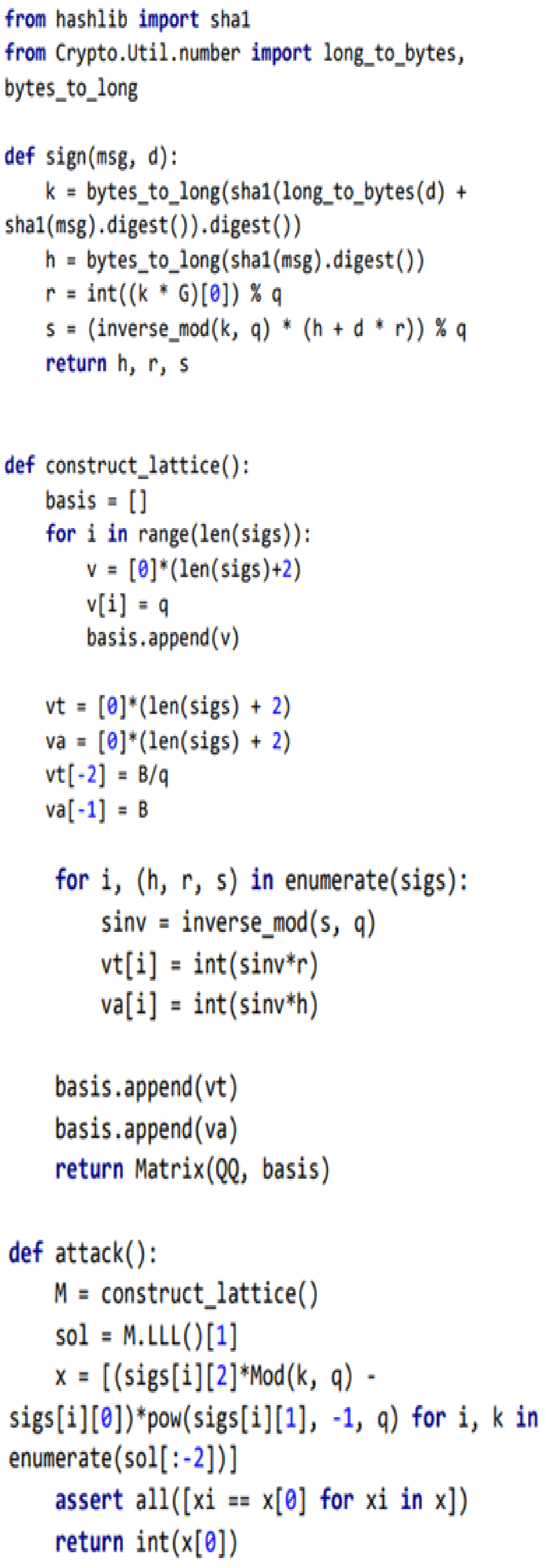
G. *Biased Nonce Attack*

If the attacker can predict the value of the nonce, they can compute a valid signature for any message of their choice, which is known as a "forgery." This type of attack is particularly dangerous in systems where the same nonce is reused multiple times, as it allows the attacker to produce multiple valid signatures. To avoid this attack, it is important to use a truly random nonce and to ensure that it is never reused.

Usually if the nonce is small then we can use a hidden number problem(HNP) and attack such kind of signing messages. Hidden Number Problem (HNP) attack on an elliptic curve signature scheme. In an HNP attack, the attacker tries to find the private key "d" used in signing messages by having access to multiple signature pairs (msg, sig) created using the same private key "d".

$$\begin{bmatrix} p & & & & & \\ & p & & & & \\ & & \cdot & & & \\ & & & p & & \\ t_1 & t_2 & \dots & t_m & B/p & \\ a_1 & a_2 & \dots & a_m & & B \end{bmatrix}$$

The attack uses constructing a lattice of basis vectors from the multiple signature pairs available to the attacker. The lattice which is generated is used for attack, which uses the LLL algorithm to find a non-zero solution. The solution found is used to compute the private key "d".



```
from hashlib import sha1
from Crypto.Util.number import long_to_bytes,
bytes_to_long

def sign(msg, d):
    k = bytes_to_long(sha1(long_to_bytes(d) +
sha1(msg).digest()).digest())
    h = bytes_to_long(sha1(msg).digest())
    r = int((k * G)[0]) % q
    s = (inverse_mod(k, q) * (h + d * r)) % q
    return h, r, s


def construct_lattice():
    basis = []
    for i in range(len(sigs)):
        v = [0]*(len(sigs)+2)
        v[i] = q
        basis.append(v)

    vt = [0]*(len(sigs) + 2)
    va = [0]*(len(sigs) + 2)
    vt[-2] = B/q
    va[-1] = B

    for i, (h, r, s) in enumerate(sigs):
        sinv = inverse_mod(s, q)
        vt[i] = int(sinv*r)
        va[i] = int(sinv*h)

    basis.append(vt)
    basis.append(va)
    return Matrix(QQ, basis)

def attack():
    M = construct_lattice()
    sol = M.LLL()[1]
    x = [(sigs[i][2]*Mod(k, q) -
sigs[i][0])*pow(sigs[i][1], -1, q) for i, k in
enumerate(sol[:-2])]
    assert all([xi == x[0] for xi in x])
    return int(x[0])
```

The conditions that attack work is when multiple messages signed with same private key and when the length of the hashes B is several bits smaller than the curve order q.

ACKNOWLEDGEMENT

I want to sincerely thank everyone who has helped me with my elliptic curve research. I want to thank one of my mentors, Abhishek, for his invaluable guidance and assistance throughout the entire study process. I want to thank my friend Connor M for supporting me as I discovered new things along the way. Their understanding of and contributions to elliptic curve research have been substantial.

I'm also appreciative of the Team bi0s community for providing me with the motivation and inspiration I needed to complete my research. I want to thank my peers for their encouragement and for creating a stimulating environment for my intellectual growth.

I would also like to express my gratitude to the authors and academics who have contributed to the understanding of elliptic curves. Their works have continuously inspired me and served as the foundation for my research.

I sincerely appreciate everyone's assistance in making this research a success.

REFERENCES

[1]. Pairing-Based Cryptography by Martijn Maas https://www.win.tue.nl/~bdeweger/downloads/MT%20Martijn%20Maas.pdf
[2]. Pohlig-Hellman Attack in ECDLP https://l0z1k.com/pohlig_hellman_attack
[3]. The Pohlig-Hellman Method by Rong-Jaye Chen https://people.cs.nctu.edu.tw/~rjchen/ECC2009/18_Pohlig-Hellman.pdf
[4]. Weak Curves In Elliptic Curve Cryptography by Peter Novotney, March 2010 https://wstein.org/edu/2010/414/projects/novotney.pdf
[5]. Hensel Lifting, Instructor: Piyush P Kurur, Scribe: Ramprasad Saptharishi https://www.cmi.ac.in/~ramprasad/lecturenotes/comp_numb_theory/lecture26.pdf
[6]. The Weil Pairing on Elliptic Curves and Its Cryptographic Applications by Alex Edward Aftuck https://digitalcommons.unf.edu/cgi/viewcontent.cgi?article=1138&context=etd
[7]. Elliptic curve cryptography and the Weil pairing by Dias da Cruz Steve https://studylib.net doc/18759762/elliptic-curve-cryptography-and-the-weil-pairing
[8]. Supersingular Curves in Cryptography by Steven D. Galbraith https://www.iacr.org/archive/asiacrypt2001/22480497.pdf

[9]. Biased Nonce Sense: Lattice Attacks against Weak ECDSA Signatures in Cryptocurrencies by Joachim Breitner and Nadia Heninger https://eprint.iacr.org/2019/023.pdf

[10]. Mukundan, P. M., Manayankath, S., Srinivasan, C., & Sethumadhavan, M. (2016, April 14). *IET Digital Library: Hash-One: a lightweight cryptographic hash function*. IET Digital Library: Hash-One: A Lightweight Cryptographic Hash Function. http://digitallibrary.theiet.org/content/journals/10.1049/iet-ifs.2015.0385

[11]. Praveen, K., Sethumadhavan, M., & Krishnan, R. (2017, February 17). Visual cryptographic schemes using combined Boolean operations. *Journal of Discrete Mathematical Sciences and Cryptography*, *20*(2), 413–437. https://doi.org/10.1080/09720529.2015.1086067

[12]. Praveen, I., & Sethumadhavan, M. (2012, August 17). A more efficient and faster pairing computation with cryptographic security. *Proceedings of the First International Conference on Security of Internet of Things*. https://doi.org/10.1145/2490428.2490448

[13]. Praveen, K., & Sethumadhavan, M. (2014). A Probabilistic Essential Visual Cryptographic Scheme for Plural Secret Images. *Advanced Computing, Networking and Informatics- Volume 2*, 225–231. https://doi.org/10.1007/978-3-319-07350-7_25