Cryptographic Engineering: Assignment 5 Prelab

In this assignment, you implement another important cryptography algorithms which is used inside Transport Layer Security (TLS) protocol in real world applications. Similar to the previous algorithm, this algorithm computes a shared secret key between two parties using a set of public parameters, so they can communicate securely using the generated shared secret key.

Point addition and Point doubling in Elliptic Curves over $\mathbb{F}_p$

Elliptic Curve E is an algebraic structure studied by mathematicians. There are many forms of Elliptic Curves. One of the most commonly used form is short Weierstrass curve $y^2 = x^3 + ax + b$. Points with coordinates (x, y) that satisfy the curve equation are on the curve.

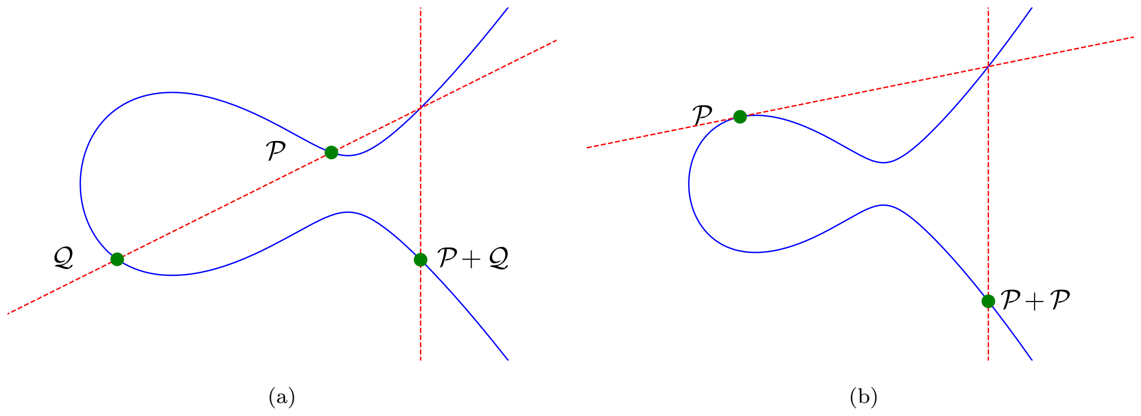

Figure 1: (a) Point Addition and (b) Point Doubling

Point addition of two points P and Q is a third point $R = P + Q$ which is constructed by first finding the third intersection point of the Elliptic curve and the line containing these two point and then drawing a vertical line from that point and intersecting it with the elliptic Curve. This is shown in Figure 1a. If the two points P and Q form a vertical line (same x-coordinate and opposite y-coordinate), then the line never intersects the Elliptic curve at a third point. We will say that they intersect at a third point called point at infinity $\mathcal{O}$. In this case, $Q = -P$ and $P + Q = \mathcal{O}$. $\mathcal{O}$is defined similar to how 0 is defined in regular arithmetic operations; $P + \mathcal{O} = P$. Note that $\mathcal{O}$is on the Elliptic curve but does not have any valid (x, y) coordinates like other points on the curve.

If the two points being added are the same, then point addition is defined as point doubling. **Point doubling** of a point P is a point $R = P + P = [2]P$ which is constructed by using the tangent to the point to get another intersection point with the Elliptic Curve and then drawing a vertical line from that point and intersecting it with the elliptic Curve. This is shown in Figure 1b. If the y-coordinate of a point P is 0, then the tangent line will be a vertical line which means it will never intersect the Elliptic curve at another point. Performing point doubling on such point will give point at infinity $\mathcal{O}$.

For purposes of this assignment, we will not handle special cases where point addition is point doubling and where point at infinity $\mathcal{O}$ is involved. In real world implementations, you might need to handle point doubling case and usually **projective coordinates** (not covered in this course) handles cases that involve point at infinity $\mathcal{O}$. Algorithm 1 shows how to compute the coordinates of point addition for short Weierstrass curve and Algorithm 2shows how to compute coordinates of point doubling for short Weierstrass curve.

Algorithm 1: Point Addition

```
Input  : $P(x_P, y_P)$, $Q(x_Q, y_Q)$, $E : y^2 = x^3 + ax + b$
Output: $R = P + Q$ where $R(x_R, y_R)$
1 $m \leftarrow (y_Q - y_P)/(x_Q - x_P)$
2 $x_R \leftarrow m^2 - x_P - x_Q$
3 $y_R \leftarrow m(x_P - x_R) - y_P$
```

Algorithm 2: Point Doubling

```
Input  : $P(x_P, y_P)$, $E : y^2 = x^3 + ax + b$
Output: $R = P + P$ where $R(x_R, y_R)$
1 $m \leftarrow (3x_P^2 + a)/(2y_P)$
2 $x_R \leftarrow m^2 - 2x_P$
3 $y_R \leftarrow m(x_P - x_R) - y_P$
```

Elliptic Curves over $\mathbb{F}_p$

Figures 1a and 1b show curves defined over Rational field (all rational numbers). There are infinite number of points on the curve defined over Rational field which requires infinite storage. To limit the number of points, the Elliptic curve E is defined over finite field $\mathbb{F}_p$. This means that the coordinates of all points except point at infinity $\mathcal{O}$ are in the field $\mathbb{F}_p$. In other words, for any point (x, y) in E other than point at $\mathcal{O}$, x and y are integers between 0 (inclusive) and p (exclusive) and satisfy the curve equation. To perform arithmetic operations, the addition becomes modular addition, subtraction becomes modular subtraction, multiplication becomes modular multiplication, and squaring becomes modular squaring. As for division, it is modular multiplication of the numerator with the modular inverse of the denominator. All modular arithmetic functions has been done in Assignment 3 and provided for you in this assignment. Therefore, you only need to implement point addition and point doubling using these modular arithmetic operations. Once point addition and point doubling are implemented, you have a cyclic group in the Elliptic curve. Basically if you start from a generator point G of order k then $\{[0]G = \mathcal{O}, G, [2]G, [3]G, [4]G, ..., [k-1]G\}$ similar to the $\{g^1 = 1, g^1, g^2, g^3, g^4, ...g^{k-1}\}$ in Assignment 4. Therefore, Diffie-Hellman algorithm can be applied here.

Point Multiplication

Elliptic curve scalar multiplication is the successive addition of a point to itself in the elliptic curve. It is commonly named **point multiplication** but it is actually a multiplication between a **scalar (integer) and a point** not multiplication between 2 points. Point multiplication is generated using the double-and-add algorithm using point addition and point doubling as shown in Algorithm 3. This algorithm is exactly like the exponentiation algorithm you implemented in Assignment 4 but modular squaring becomes point doubling and modular multiplication becomes point addition. Furthermore, the select function is now select point which is 2 integers (the coordinates) instead of 1 integer.

Elliptic Curve Diffie-Hellman (ECDH)

The algorithm you are going to implement in this assignment is called Elliptic Curve Diffie-Hellman key-exchange algorithm. The concept is very similar to the regular Diffie-Hellman, but the computation is now performed on an Elliptic curve defined over $\mathbb{F}_p$. A generator point G is chosen such that it generates a large number of points in the elliptic curve. In this assignment, we will use the elliptic curve and generator point defined in Draft NIST Special Publication 800-186. The prime used is still the same prime from Assignments 3 and 4. $p = 2^{255} - 19$

Algorithm 3: Point Multiplication using double-and-add algorithm.

```
Input  : $P$, $E$, $s = (s_{k-1} \dots s_1 s_0)$ with $s_{k-1} = 1$
Output: $R = [s]P$
1 $R \leftarrow P$
2 for $i \leftarrow k-2$ downto $0$ do
3     $R \leftarrow (R+R)$
4     if $s_i = 1$ then
5         $R \leftarrow (R+P)$
6 return $R$
```

Key Generation

In key generation, the secret key is generated exactly like how it is generated in Diffie-Hellman in Assignment 4. The public key pk is generated from the generator point and secret key by performing $M = [sk]G$ and taking the x-coordinate of M as the public key.

Shared Secret Generation

In shared secret generation, the first step is to regenerate the y-coordinate of point $M(x_M, y_M)$ whose x-coordinate is the public key. Since the original point is on the curve $E : y^2 = x^3 + ax + b$, the first step is to compute $s = y_M^2 = (x_M^3 + ax_M + b) \mod p$. Now, y_M has two possible values $s^{1/2}$ or $-s^{1/2}$. Choosing either one will have no impact on the generated shared secret, so only one of them needs to be computed.

To compute modular square root for $p = 2^{255} - 19$, follow Algorithm 4 (it applies to any prime that gives remainder 5 when divided by 8 which our p does). The algorithm is straight forward as you apply the modular arithmetic operations constructed in Assignment 3. For Line 2, $(p-5)/2$ is a constant integer and can be computed in SageMath. Its value is 1 followed by 249 1's followed by a 0 followed by 1. You use the same approach for modular inversion in assignment 3 as the exponent is also constant here.

Once y_M is recovered, the shared secret is generated from M and the secret key by performing $N = [sk]M$ and taking the x-coordinate of N as the shared secret.

Algorithm 4: Modular square-root for primes $\mod 8 = 5$

```
Input  : $s$, $p$
Output: $y = s^{1/2} \mod p$
1 $u \leftarrow 2 \cdot s \mod p$
2 $v \leftarrow u^{(p-5)/8} \mod p$
3 $w \leftarrow u \cdot v^2 \mod p$
4 $y \leftarrow s \cdot v \cdot (w-1) \mod p$
5 return $y$
```

Elliptic Curve Diffie-Hellman Algorithm

Two parties Alice and Bob want to generate a shared secret using ECDH. The algorithm works as follows:

- Alice generates her pair of secret key $sk_A = a$ and public key $pk_A = [a]G$ using the key generation function. Similarly, Bob generates his pair of secret key $sk_B = b$ and public key $pk_B = [b]G$

using the key generation function. This can be done ahead of time as no party is required to communicate with the other party.

- Alice and Bob exchange their public keys pk_A and pk_B through a public medium (Example: Internet).
- Alice generates her shared secret ss_A using her secret key $sk_A = a$ and Bob's pubic key $pk_B = [b]G$ using the shared secret generation function. Similarly, Bob generates his shared secret ss_B using his secret key $sk_B = b$ and Alice's pubic key $pk_A = [a]G$ using the shared secret generation function.

The two shared secrets are guaranteed to be equal.

$$ss_A = [a][b]G = [ab]G = [ba]G = [b][a]G = ss_B$$

SageMath

The following SageMath code will help you in performing Elliptic curve computations and verifying your results. SageMath code used to generate Figures 1a and 1b is provided here. You can play around with it to get a good idea how Elliptic curves and point arithmetic are constructed. Furthermore, you can define the curve over a finite field with the code provided below and see what you get.

```
# Finite field $F_p$
U = GF(p)
# Create an Elliptic Curve over $F_p$ with equation: $y^2 = x^3 + ax + b$
E = EllipticCurve(GF(p),[a,b])
# Point at infinity.
# Note when printing point at infinity, sage will show (0:1:0)
O = E(0)
# Point with coordinates $(x, y)$.
# Note when printing any point other than point at infinity, sage will show (x:y:1)
P = E(x,y)
# Get points that have coordinate x in E.
P1 = E.lift_x(x) # First point
P2 = E.lift_x(x, all=True)[1] # second point
# The point coordinates $x$ and $y$ are stored in an array. To access them,
x = P[0]
y = P[1]
# Point Addition $R = P + Q$: Adding two points P and Q
R = P + Q
# Point Doubling $R = P + P = [2]P$: Doubling a point P
R = 2*P
# Point Multiplication $R = [s]P$: Multiplying P by scalar s
R = s*P
# Point Inversion $R = -P$: $P + R$ = point at infinity
# Note if $P = (x, y)$, then this is just $R = (x, -y)$
R = -P
# Order of point P :  $[k]P$ = point at infinity
k = P.order()
```