Elliptic Curve Cryptography

CS6025 Data Encoding

Yizong Cheng

3-12-15

Elliptic Curves over Z_p

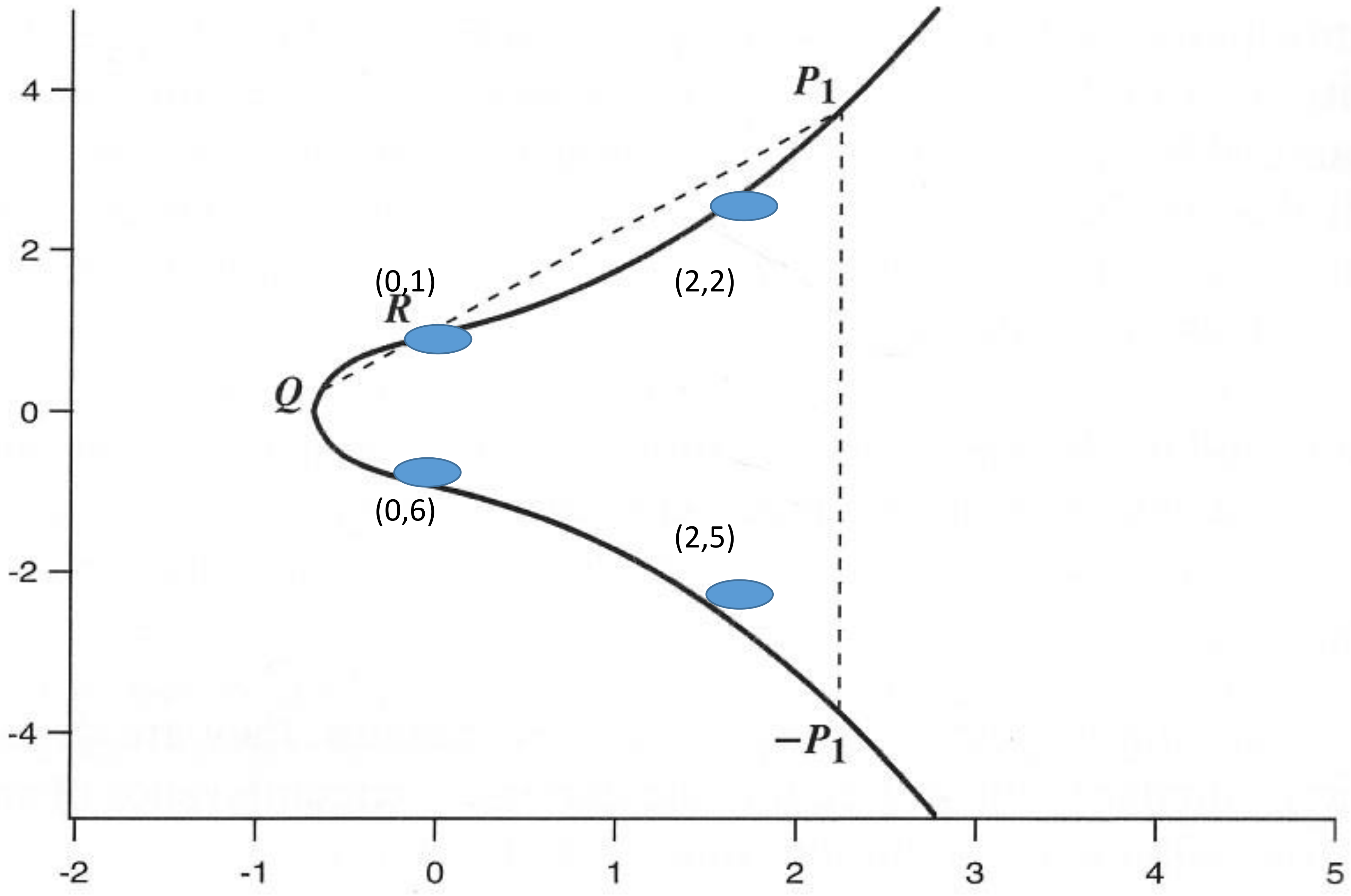
- The elliptic curve $E_p(a, b)$ is the set of points (x, y) from Z_p, satisfying the equation $y^2 = x^3 + ax + b \bmod p$, together with the point at infinity, O.
- Requirement: $4a^3 + 27b^2 \bmod p \neq 0$.
- Example: $E_7(1, 1)$, for each of x= 0,1,…,6, compute $x^3 + x + 1 \bmod 7$.
- If quadratic residue (square), find roots as y.

Z_7 Exponentiation Table

- a a^2 a^3 a^4 a^5 a^6
- 2 4 1 2 4 1
- 3 2 6 4 5 1
- 4 2 1 4 2 1
- 5 4 6 2 3 1
- 6 1 6 1 6 1
- Quadratic residues: 1, 2, 4.

$f(x) = x^3 + x + 1 \bmod 7$

- $x=0$, $f(x) = 1$, $y = 1, 6$ Points $(0, 1)$, $(0, 6)$
- $x=1$, $f(x) = 3$
- $x=2$, $f(x) = 4$, $y = 2, 5$ Points $(2, 2)$, $(2, 5)$
- $x=3$, $f(x) = 3$
- $x=4$, $f(x) = 6$
- $x=5$, $f(x) = 5$
- $x=6$, $f(x) = 6$
- $E_7(1, 1)$ has five elements: O, $(0, 1)$, $(0, 6)$, $(2, 2)$, $(2, 5)$.



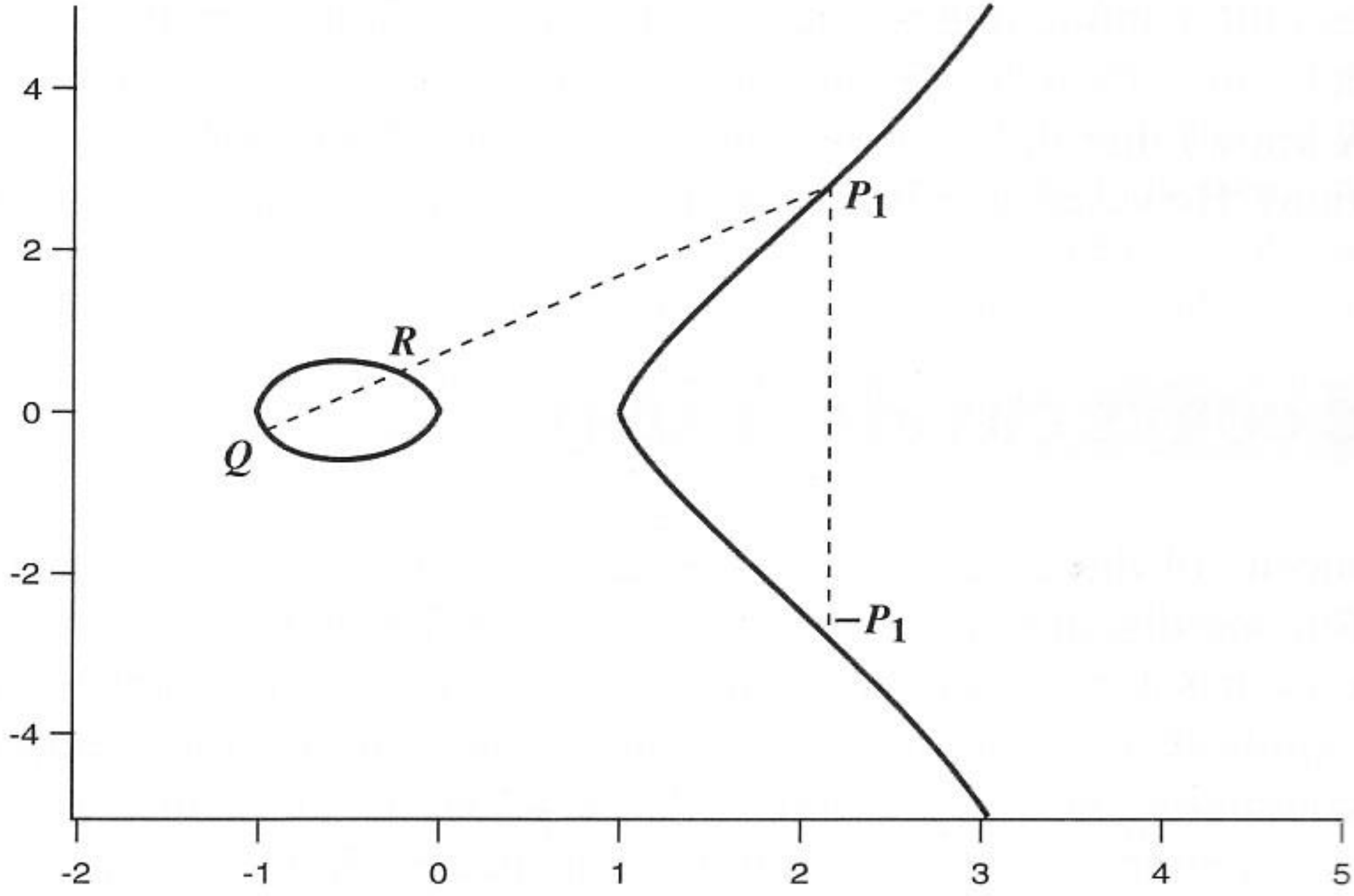
4
2
0
-2
-4
-2
-1
0
1
2
3
4
5
P_1
$-P_1$
Q
R
(0,1)
(2,2)
(0,6)
(2,5)

Class Point for (x, y) and a Unique O

```
class Point{
  public BigInteger x;
  public BigInteger y;
  static Point O = new Point(null, null);
  public Point(BigInteger xx, BigInteger yy){ x = xx; y = yy; }
  public String toString(){
    return this.equals(O) ? "O" :
    "(" + x.toString(16) + ",\n" + y.toString(16) + ")";
  }
}
```

Elliptic Curves Addition

- If P, Q, R are 3 points on the curve and also on a line, then P + Q + R = O.
  - When the line is a tangent to the curve, count the point on both twice: P + P + R = O.
- A vertical line meets O at infinity.
- O is the additive identity.
- (x, y) +(x, -y) = O
  - (0, 1) + (0, 6) = O, (2, 2) + (2, 5) = O
- -(x, y) = (x, -y)
- P + Q = -R.



(a) $y^2 = x^3 - x$

Slope, Line, and Intersection

- The slope of the line going through (x_1, y_1) and (x_2, y_2) is
- $\lambda = (y_2 - y_1)(x_2 - x_1)^{-1}$.
- The equation of the line is $y = \lambda x + \nu$.
- The intersections of the line and the elliptic curve are roots of
- $(\lambda x + \nu)^2 = x^3 + ax + b$, or those of
- $x^3 - \lambda^2 x^2 + (a - 2\lambda\nu)x + b - \nu^2 = 0$

The Roots of the Cubic Equation

- We know that x_1 and x_2 are roots of the above equation and are looking for the third root, x_3.
- The equation itself can be written as $(x-x_1)(x-x_2)(x-x_3) = 0$, or,
- $x^3-(x_1+x_2+x_3)x^2+(x_1x_2+x_1x_3+x_2x_3)x - x_1x_2x_3 = 0$. Compare coefficients,
- $\lambda^2 = x_1 + x_2 + x_3$, or $x_3 = \lambda^2 - x_1 - x_2$.

$(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$

- $x_3 = \lambda^2 - x_1 - x_2$.
- $(x_3, -y_3)$ and (x_1, y_1) have slope λ:
- $y_3 = \lambda(x_1 - x_3) - y_1$.
- Note: a and b are not used in computing slope or addition.

```
Point add(Point P1, Point P2){
   if (P1.equals(Point.O) return P2;
   if (P2.equals(Point.O) return P1;
   if (P1.x.equals(P2.x)) if (P1.y.equals(P2.y)) return selfAdd(P1);
                          else return Point.O;
   BigInteger t1 = P1.x.subtract(P2.x).mod(p);
   BigInteger t2 = P1.y.subtract(P2.y).mod(p);
   BigInteger k = t2.multiply(t1.modInverse(p)).mod(p); // slope
   t1 = k.multiply(k).subtract(P1.x).subtract(P2.x).mod(p); // x3
   t2 = P1.x.subtract(t1).multiply(k).subtract(P1.y).mod(p); // y3
   return new Point(t1,t2);
}
```

$(x_1, y_1) + (x_1, y_1) = (x_3, y_3)$

- Slope is that of the tangent at (x_1, y_1).
- Differentiating $y^2 = x^3 + ax + b$
- $2ydy = (3x^2 + a)dx$
- $\lambda = dy/dx = (3x_1{}^2 + a)(2y_1)^{-1}$
- $x_3 = \lambda^2 - 2x_1$.
- $y_3 = \lambda(x_1 - x_3) - y_1$.
- Note: b is not used in slope computation.

```
Point selfAdd(Point P){
     if (P.equals(Point.O)) return Point.O; // O+O=O
     BigInteger t1 = P.y.add(P.y).mod(p);  // 2y
     BigInteger t2 = P.x.multiply(P.x).mod(p).multiply(three).add(a).mod(p);
          // 3xx+a
     BigInteger k = t2.multiply(t1.modInverse(p)).mod(p); // slope or tangent
     t1 = k.multiply(k).subtract(P.x).subtract(P.x).mod(p); // x3 = kk-x-x
     t2 = P.x.subtract(t1).multiply(k).subtract(P.y).mod(p); // y3 = k(x-x3)-y
     return new Point(t1,t2);
}
```

Addition on $E_7(1,1)$

- $(0,1)+(0,1)$: $\lambda=(0+1)2^{-1}=4$, $\lambda^2=2$
- $x_3=\lambda^2-0-0=2$, $y_3=\lambda(x_1-x_3)-y_1=5$, $(2,5)$
- $(0,1)+(2,2)$: $\lambda=(2-1)(2-0)^{-1}=4$
- $x_3=2-0-2=0$, $y_3=4(0-0)-1=6$, $(0,6)$
- $(0,1)+(2,5)$: $\lambda=(5-1)2^{-1}=2$, $\lambda^2=4$
- $x_3=4-0-2=2$, $y_3=2(0-2)-1=2$, $(2,2)$
- $(0,6)+(2,2)$: $\lambda=(2-6)2^{-1}=5$, $\lambda^2=4$
- $x_3=4-0-2=2$, $y_3=5(0-2)-6=5$, $(2,5)$

Addition on $E_7(1,1)$

- $(0,6)+(0,6)$: $\lambda=(0+1)(2x6)^{-1}=3$, $\lambda^2=2$
- $x_3=2-0-0=2$, $y_3=3(0-2)-6=2$, $(2,2)$
- $(0,6)+(2,5)$: $\lambda=(5-6)(2-0)^{-1}=3$
- $x_3=2-0-2=0$, $y_3=3(0-0)-6=1$, $(0,1)$
- $(2,2)+(2,2)$: $\lambda=(3(2)^2+1)4^{-1}=5$, $\lambda^2=4$
- $x_3=4-2-2=0$, $y_3=5(2-0)-2=1$, $(0,1)$
- $(2,5)+(2,5)$: $\lambda=(3(2)^2+1)3^{-1}=2$
- $x_3=4-2-2=0$, $y_3=2(2-0)-5=6$, $(0,6)$

Addition Table for $E_7(1,1)$

- O (0,1) (0,6) (2,2) (2,5)
- (0,1) (2,5) O (0,6) (2,2)
- (0,6) O (2,2) (2,5) (0,1)
- (2,2) (0,6) (2,5) (0,1) O
- (2,5) (2,2) (0,1) O (0,6)

- (0,1) + (0,1) + (2,2) = O = (0,6) + (0,6) + (2,5)
- (2,2) + (2,2) + (0,6) = O = (2,5) + (2,5) + (0,1)

Multiplication or the Power Table in $E_7(1,1)$

P	2P	3P	4P	5P
(0,1)	(2,5)	(2,2)	(0,6)	O
(0,6)	(2,2)	(2,5)	(0,1)	O
(2,2)	(0,1)	(0,6)	(2,5)	O
(2,5)	(0,6)	(0,1)	(2,2)	O

- Multiplication is defined as repeated addition.
- This is a cyclic group with all nonzero elements primitive (with order 5).

Russian Peasant's Algorithm

- How to multiply 22 (10110_2) to a point P?
- Or 22 of P added together?
- The same as P selfAdded 4 times (multiplied by 16) plus P selfAdded 2 times (multiplied by 4) plus P selfAdded once (multiplied by 2).
- ((P x 2 x 2 + P) x 2 + P) x 2
- = P x 2 x 2 x 2 x 2 + P x 2 x 2 + P x 2
- = P x 16 + P x 4 + P x 2
- = P x (10000_2 + 100_2 + 10_2) = P x 10110_2 = P x 22

Efficient Multiplication/Power Computation

```
Point multiply(Point P, BigInteger n){
   if(n.equals(BigInteger.ZERO)) return Point.O;
   int len = n.bitLength();  // position preceding the most significant bit 1
   Point product = P;
   for(int i = len - 2; i >= 0; i--){
      product = selfAdd(product);
      if (n.testBit(i)) product = add(product, P);
   }
   return product;
}
```

Elliptic Curve Logarithm

- For a elliptic curve group of a large size, and a primitive element G,
- computing power kG is easy
- but computing k, knowing G and kG, is difficult.
- This is called elliptic curve logarithm.
- used in public-key encryption/authentication.

Elliptic Curve Key Exchange

- Alice has private key n_A and computes n_AG as the public key.
- Bob has private key n_B and public key n_BG.
- Alice and Bob exchange the public keys.
- Each computes $n_A(n_BG)=n_B(n_AG) = (n_An_B)G$ as the shared secret.

Elliptic Curve Standard P-256

- NIST standard
- p =
  1157920892103562487626974469494075735300861434
  15290314195533631308867097853951
- a = -3, b is not needed in implementation
- G =
  (6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0
  f4a13945d898c296,
  4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececb
  b6406837bf51f5)
- n = order of G = size of the cyclic group =
  1157920892103562487626974469494075735299969552
  24135760342422259061068512044369

ECP256.txt Contains p, n, a, b, Gx, Gy

```
ffffffff00000001000000000000000000000000ffffffffffffffffffffffff
ffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551
-3
5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b
6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296
4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5
```

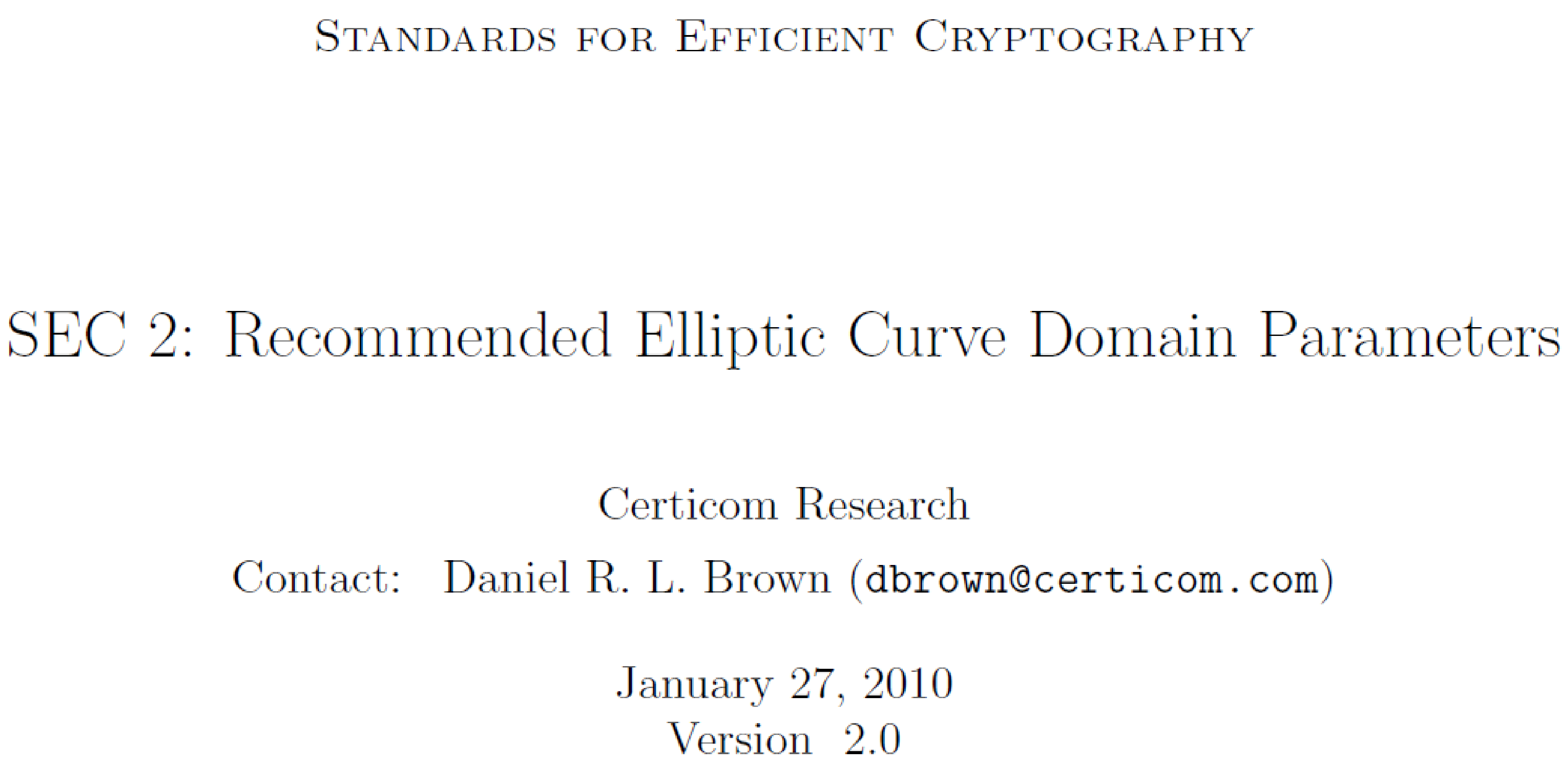
Read Curve Parameters

```
void readCurveSpecs(String filename){
    Scanner in = null;
    try {
     in = new Scanner(new File(filename));
    } catch (FileNotFoundException e){
      System.err.println(filename + " not found");
      System.exit(1);
    }
    p = new BigInteger(in.nextLine(), 16);
    n = new BigInteger(in.nextLine(), 16);
    a = new BigInteger(in.nextLine(), 16);
    b = new BigInteger(in.nextLine(), 16 );
    G = new Point(new BigInteger(in.nextLine(), 16),
                  new BigInteger(in.nextLine(), 16));
    in.close();
}
```

G is on $y^2 = x^3 + ax + b$ and $nG = O$

```
void checkParameters(){
   BigInteger lhs = G.y.multiply(G.y).mod(p);
   BigInteger rhs = G.x.modPow(three, p).add(G.x.multiply(a).mod(p)).add(b).mod(p);
   System.out.println(lhs.toString(16));
   System.out.println(rhs.toString(16));  // these two lines should be the same
   Point power = multiply(G, n);
   System.out.println(power);  // this should be O
 }
```

Certicom Standards


Standards for Efficient Cryptography

SEC 2: Recommended Elliptic Curve Domain Parameters

Certicom Research

Contact: Daniel R. L. Brown (dbrown@certicom.com)

January 27, 2010
Version 2.0

secp256k1 Used by Bitcoin Wallet

2.4.1 Recommended Parameters secp256k1

The elliptic curve domain parameters over $\mathbb{F}_p$ associated with a Koblitz curve `secp256k1` are specified by the sextuple $T = (p, a, b, G, n, h)$ where the finite field $\mathbb{F}_p$ is defined by:

$$\begin{aligned} p &= \mathtt{FFFFFFFF\ FFFFFFFF\ FFFFFFFF\ FFFFFFFF\ FFFFFFFF\ FFFFFFFF\ FFFFFFFE} \\ &\quad\ \mathtt{FFFFFC2F} \\ &= 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1 \end{aligned}$$

The curve E: $y^2 = x^3 + ax + b$ over $\mathbb{F}_p$ is defined by:

$$\begin{aligned} a &= \mathtt{00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000} \\ &\quad\ \mathtt{00000000} \\ b &= \mathtt{00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000} \\ &\quad\ \mathtt{00000007} \end{aligned}$$

secp256k1 G and Order

The base point G in compressed form is:

$$G = \texttt{02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798}$$

and in uncompressed form is:

$$G = \texttt{04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8}$$

Finally the order n of G and the cofactor are:

$$n = \texttt{FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141}$$

$$h = \texttt{01}$$

secp256k1.txt with p, n, a, b, Gx, Gy

```
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141
0
7
79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798
483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8
```

H16.java

```
public class H16{

static BigInteger three = new BigInteger("3");
BigInteger p;
Point G;
BigInteger a;
BigInteger b;
BigInteger n;

public static void main(String[] args){
   H16 h16 = new H16();
   h16.readCurveSpecs(args[0]);
   h16.checkParameters();
}
```

Analog of Diffie-Hellman Key Exchange

```
void generateKeys(){
   privateKeyA = new BigInteger(privateKeySize, random);
   publicKeyA = multiply(G, privateKeyA);
   privateKeyB = new BigInteger(privateKeySize, random);
   publicKeyB = multiply(G, privateKeyB);
}

void sharedSecret(){
   Point KA = multiply(publicKeyB, privateKeyA);  // secret computed by A
   Point KB = ? // secret computed by B
   System.out.println(KA);
   System.out.println(KB);
}
```

Elliptic Curve Encryption

- Sender has message m, a number smaller than n, the order of the EC group.
- It generates a per-message random number k, also less than n.
- Compute kG and kY = (u,v), where G is the generator and Y is receiver’s public key.
- Send (kG, mu mod p).
- Receiver computes kY = x(kG) with its private key x and then decodes $m = muu^{-1} \bmod p$.

```
void encryptionForB(){
   BigInteger message = new BigInteger(privateKeySize, random);
   System.out.println(message.toString(16));
   BigInteger k = new BigInteger(privateKeySize, random);
   Point kG; // k times G
   Point kY; // k times publicKeyB
   BigInteger mu;  // message times kY.x mod p
   System.out.println(kG); System.out.println(mu.toString(16));
   // (kG, mu) is the encrypted message
   Point kY2;  // B computes kY as privateKeyB times kG
   BigInteger decodedMessage; // kY2.x modinverse times mu mod p
   System.out.println(decodedMessage.toString(16));
 }
```

Elliptic Curve Digital Signature Algorithm

- B generates a message m, smaller than n.
- B generates a per-message random number k, also less than n.
- B computes $kG = (x,y)$ and $r = x \bmod n$
- B computes $s = k^{-1}(m + privateKeyB * r) \bmod n$
- (r, s) is the signature for message m
- A computes $w = s^{-1} \bmod n$
- A computes $u1 = mw$ and $u2 = rw$
- A computes $X = u1\ G + u2\ PublicKeyB$
- A validates the signature if $X.x \bmod n$ is the same as r

```
void ECDSA(){
   BigInteger message = new BigInteger(160, random);
   System.out.println(message.toString(16));
   BigInteger k = new BigInteger(privateKeySize, random);
   Point kG; // k times G
   BigInteger r; // r = kG.x mod n
   BigInteger s; // s = k's modInverse times (message plus privateKeyB times r) mod n
   // if r or s == 0 redo k
   // (r,s) is the signature for message (digest)
   BigInteger w; // w is s's multiplicative inverse mod n
   BigInteger u1; // u1 = message times w
   BigInteger u2; // u2 = r times w
   Point X;  // X is u1 times G + u2 times publicKeyB
   if (X.equals(Point.O)) // reject the signature
   else if (X.x.mod(n).equals(r)) // accept the signature
   else // reject the signature
 }
```

Homework 16: due 3-25-15

- Complete H16.java and apply it to either ECP256.txt or secp256k1.txt and display the results.