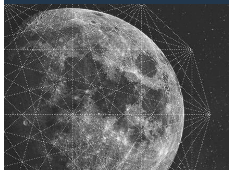
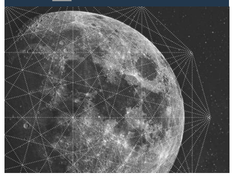
ECC & SM2

Long Wen
longwen6@gmail.com
20250712 @ Qingdao

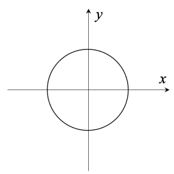
目
录
Contents



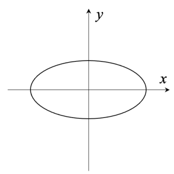
目
录
Contents



1.1 ECC basics

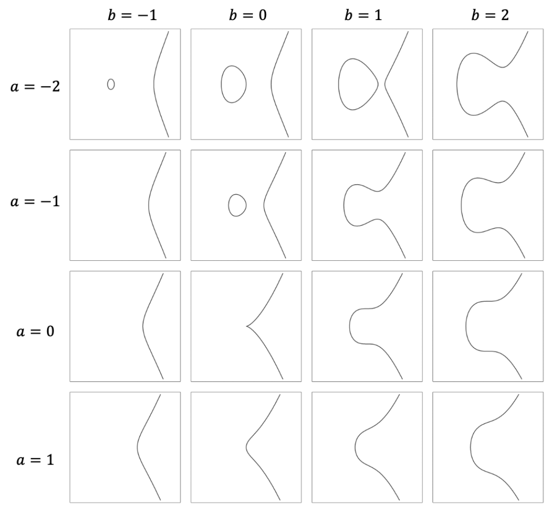


$x^2 + y^2 = r^2$



$ax^2 + by^2 = c$

Pay attention to the symmetry property:
Allows us to compress the point representation

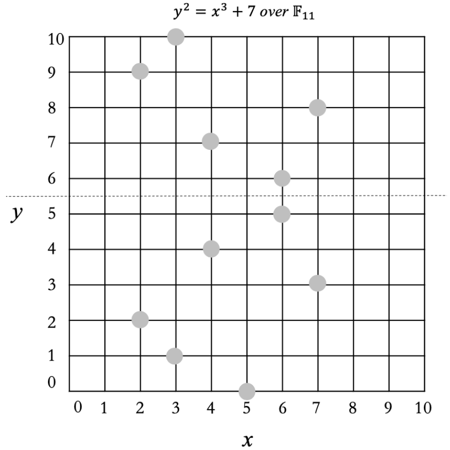


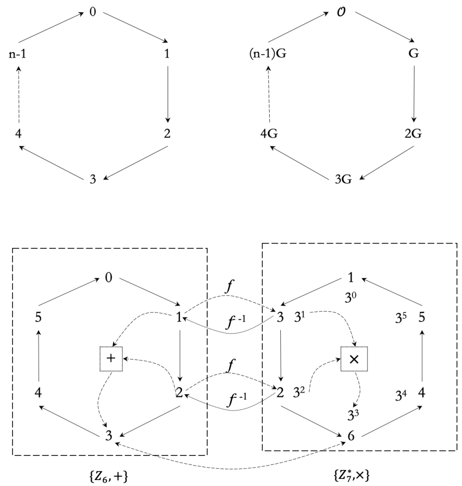
$y^2 = x^3 + ax + b$

1.1 ECC basics – finite field

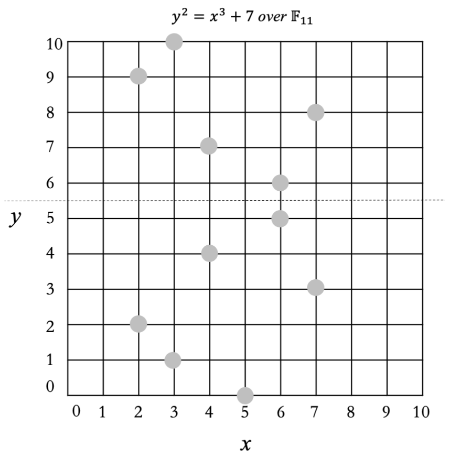
- **Finite field $\mathbb{F}_q$:**
  - A finite set which is a field, this means that multiplication, addition, subtraction and division (excluding division by zero) are defined and satisfy the rules of arithmetic
- **Finite field order:**
  - The number of elements of a finite field is called its order $|F_q| = q$
- **Characteristic:**
  - A finite field of order q exists if and only if q is a prime power p^m (where p is a prime number and m is a positive integer.) Namely $q = p^m$, We call the characteristic of the field is p
    - If $m = 1$, the finite field is prime field
    - If $m \geq 2$, the finite field is extension field, $m = 2$, we call binary field
- **Cofactor:**
  - Defined as the ratio between the order of a group and that of the subgroup
- **Cofactor && order && cyclic subgroup:**
  - Let elliptic curve E is defined on a finite field $\mathbb{F}_q$ where $q = p^m$, p is prime. Let $N = \#(E(\mathbb{F}_q))$ to be the number of elements of elliptic curve group, namely order of the elliptic curve group. So how to find a cyclic subgroup which reduce to find the generator of the cyclic subgroup.
    - Let $r \mid N$ and r is the biggest prime which can divide N
    - Randomly choose a point $P \in E(\mathbb{F}_q)$ on the elliptic curve group, then the order of point P can divide N, namely $ord(P) \mid N$
    - $ord(P) \cdot P = O, ord(P) \mid N \Rightarrow N \cdot P = O$
    - Let cofactor $h = N/r$, $N \cdot P = O \Rightarrow r \cdot h \cdot P = O \Rightarrow r(h \cdot P) = O$
    - Let subgroup generator $G = h \cdot P$, its' order is prime r. Finally, we create a subgroup $< G >$with order r . Since the subgroup is isomorphism with $\mathbb{Z}_r$, thus the subgroup is a cyclic group

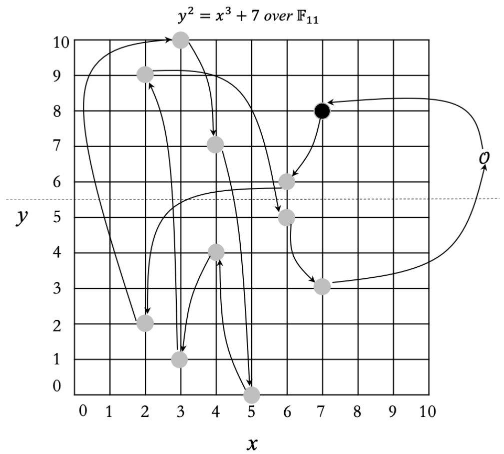
1.1 ECC basics





1.1 ECC basics





1.2 SM2 Parameters

- SM2 system parameters:
  - $\mathbb{F}_q$: finite field where $|\mathbb{F}_q| = q$
  - a, b: elliptic curve equation parameters
  - $G = (x_G, y_G)$: base point
  - n: order
  - h: cofactor where $h = |E(F_q)|/n$
- SM2 system parameters: prime field
  - Elliptic curve equation: $y^2 = x^3 + ax + b$ over $\mathbb{F}_q$-256
  - Prime q: 8542D69E 4C044F18 E8B92435 BF6FF7DE 45728391 5C45517D 722EDB8B 08F1DFC3
  - a: 787968B4 FA32C3FD 2417842E 73BBFEFF 2F3C848B 6831D7E0 EC65228B 3937E498
  - b: 63E4C6D3 B23B0C84 9CF84241 484BFE48 F61D59A5 B16BA06E 6E12D1DA 27C5249A
  - $G = (x_G, y_G)$, $ord(G) = n$
  - x_G: 421DEBD6 1B62EAB6 746434EB C3CC315E 32220B3B ADD50BDC 4C4E6C14 7FEDD43D
  - y_G: 0680512B CBB42C07 D47349D2 153B70C4 E5D7FDFC BFA36EA1 A85841B9 E46E09A2
  - n : 8542D69E 4C044F18 E8B92435 BF6FF7DD 29772063 0485628D 5AE74EE7 C32E79B7

1.2 SM2 Signature Algorithm

- Precompute:
  - compute $Z_A = H_{256}(ENTL_A||ID_A||a||b||x_G||y_G||x_A||y_A)$
    - identifier ID_A length is $entlen_A$
    - $ENTL_A$ is encoded from $entlen_A$ and takes two bytes
    - H_{256}: hash function SM3
- KeyGen:
  - $P_A = d_A \cdot G = (x_A, y_A)$
- Sign(M):
  - $Sign_{d_A}(M, Z_A) \to (r, s)$
  - Set $\overline{M} = Z_A||M$
  - Compute $e = H_v(\overline{M})$, where the output of H_v is v
  - Generate random number $k \in [1, n-1]$
  - Compute $kG = (x_1, y_1)$
  - Compute $r = (e + x_1) \ mod \ n,$
    - if $r = 0 \ or \ r + k = n$, generate random number k again
  - Compute $s = ((1 + d_A)^{-1} \cdot (k - r \cdot d_A))mod \ n$
    - if $s = 0$, generate random number k again

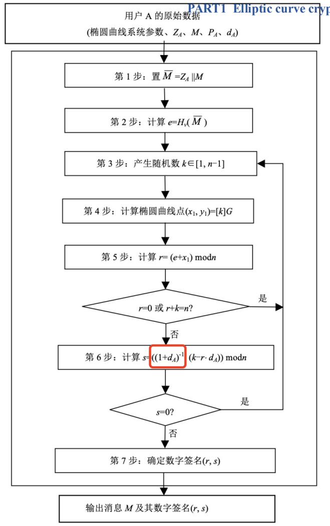


图 1 数字签名生成算法流程

1.2 SM2 Signature Algorithm

- Verify signature
  - $Verify_{P_A}(M', r', s') \to 0/1$
  - Compute $Z_A = H_{256}(ENTL_A||ID_A||a||b||x_G||y_G||x_A||y_A)$
  - Check $r' \in [1, n-1]$
  - Check $s' \in [1, n-1]$
  - Set $\overline{M'} = Z_A||M'$
  - Compute $e' = H_v(\overline{M'})$
  - Compute $t = (r' + s') \ mod \ n$
  - Compute $(x_1', y_1') = s'G + tP_A$
  - Compute $R = (e' + x_1') \ mod \ n$, check $R == r'$
- Correction check
  - $s'G + tP_A = (s' + (r' + s')d_A)G$

    $= s'(1 + d_A)G + r'd_AG$
  - $s = ((1 + d_A)^{-1} \cdot (k - r \cdot d_A))mod \ n \Rightarrow k = s(1 + d_A) + rd_A$

    $kG = s(1 + d_A)G + rd_AG$

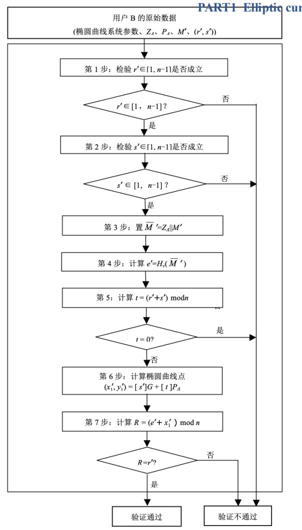


图 2 数字签名验证算法流程

1.3 SM2 Encryption

- Encryption: $Enc(M, P_B)$
  - Generate random number $k \in [1, n-1]$
  - Compute $C_1 = kG = (x_1, y_1)$
  - Compute $S = hP_B$
    - If S is O, revert error (check pubkey)
  - Compute $kP_B = (x_2, y_2)$
  - Compute $t = KDF(x_2||y_2, klen)$
    - If $t == 0$, generate random number again
  - Compute $C_2 = M \oplus t$
  - Compute $C_3 = Hash(x_2||M||y_2)$
  - Output ciphertext $C = C1||C_2||C_3$

- Key derivation function：$KDF(Z, klen)$
  - Init 32 bit counter $ct = 0x00000001$
  - for $i \in \ [1, \lceil klen/v \rceil]$
    - Compute $H_{a_i} = H_v(Z||ct)$
    - $ct++$
    - If $klen/v$ is integer，then set $H_{a}!_{\lceil klen/v \rceil} = H_{a_{\lceil klen/v \rceil}}$，
    - else $H_{a}!_{\lceil klen/v \rceil}$ is $H_{a_{\lceil klen/v \rceil}}$ the left most $(klen - (v * \lfloor klen/v \rfloor))$ bits
  - Compute $K = H_{a_1}||H_{a_2} \ || \ ... \ || \ H_{a_{\lceil klen/v \rceil}}|| \ H_{a}!_{\lceil klen/v \rceil}$

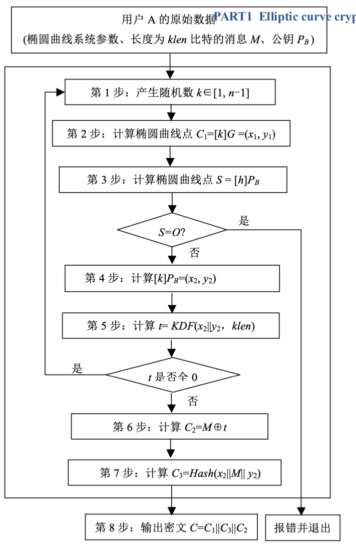


图 1 加密算法流程

SM2 Decryption

- Decryption: $Dec(C, d_B)$
  - Check C_1 satisfies the elliptic curve equation
  - Compute $S = hC_1$
    - If S is O, revert error
  - Compute $d_BC_1 = (x_2, y_2)$
  - Compute $t = KDF(x_2||y_2, klen)$
    - If $t == 0$, generate random number again
  - Compute $M' = C_2 \oplus t$
  - Compute $u = \ Hash(x_2||M'||y_2)$，check $u = C_3$
  - Output plaintext M'

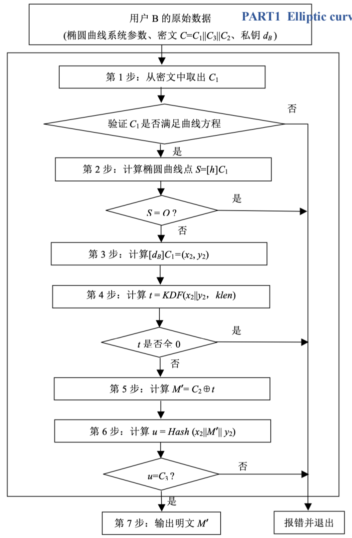
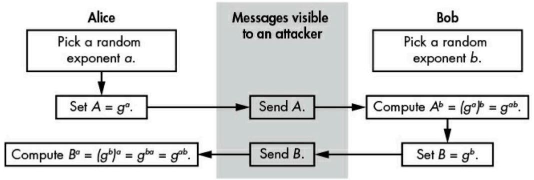
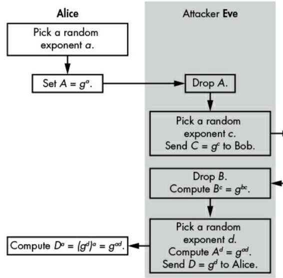


图 2 解密算法流程

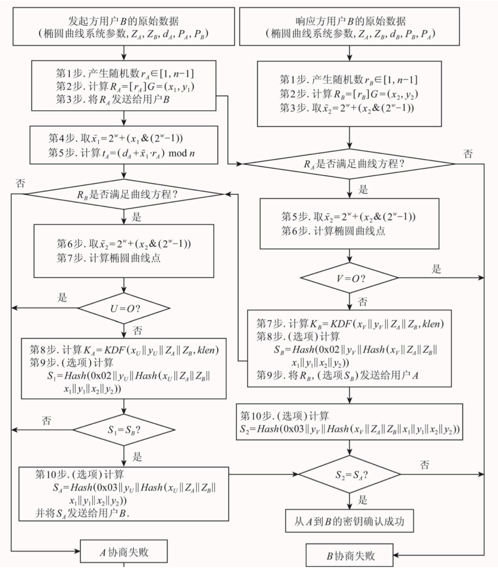
Diffie-Hellman and MITM





SM2 key exchange

SM2 密钥交换协议中，用户 A 的密钥对包括其私钥 d_A 和公钥 $P_A=[d_A]G=(x_A,y_A)$，用户 B 的密钥对包括其私钥 d_B 和公钥 $P_B=[d_B]G=(x_B,y_B)$。用户 A 具有位长为 $entlen_A$ 的可辨别标识 ID_A，记 $ENTL_A$ 是由整数 $entlen_A$ 转换而成的 2B 数据；用户 B 具有位长为 $entlen_B$ 的可辨别标识 ID_B，记 $ENTL_B$ 是由整数 $entlen_B$ 转换而成的 2B 数据。A，B 双方都需要用密码杂凑算法求得用户 A 的杂凑值 $Z_A=H_{256}(ENTL_A\|ID_A\|a\|b\|x_G\|y_G\|x_A\|y_A)$ 和用户 B 的杂凑值 $Z_B=H_{256}(ENTL_B\|ID_B\|a\|b\|x_G\|y_G\|x_B\|y_B)$。



SM2 key exchange

- Key exchange protocol: user A and user B Cooperatively generate a key of length $klen$

(1) Generate random number $r_A \in [1, n-1]$, and compute $R_A = r_A G = (x_1, y_1)$

R_A

(1) Generate random number $r_B \in [1, n-1]$, and compute $R_B = r_B G = (x_2, y_2)$

(2) Compute $\overline{x_2} = 2^\omega + (x_2 \& (2^\omega - 1)), \omega = \lceil(\lceil \log_2 n \rceil/2)\rceil - 1$

(3) Compute $t_B = (d_B + \overline{x_2} \cdot r_B) \bmod n$

(4) Verify R_A satisfies elliptic curve equation, and compute $\overline{x_1} = 2^\omega + (x_1 \& (2^\omega - 1)), \omega = \lceil(\lceil \log_2 n \rceil/2)\rceil - 1$

(5) Compute $V = (h \cdot t_B)(P_A + \overline{x_1} R_A) = (x_V, y_V)$

(6) Compute $K_B = KDF(x_V || y_V || Z_A || Z_B, klen)$

(7) Compute $S_B = Hash(0x02 || y_V || Hash(x_V || Z_A || Z_B || x_1 || y_1 || x_2 || y_2))$

R_B, S_B

(2) Compute $\overline{x_1} = 2^\omega + (x_1 \& (2^\omega - 1)), \omega = \lceil(\lceil \log_2 n \rceil/2)\rceil - 1$

(3) Compute $t_A = (d_A + \overline{x_1} \cdot r_A) \bmod n$

(4) Verify R_B satisfies elliptic curve equation, and compute $\overline{x_2} = 2^\omega + (x_2 \& (2^\omega - 1)), \omega = \lceil(\lceil \log_2 n \rceil/2)\rceil - 1$

(5) Compute $U = (h \cdot t_A)(P_B + \overline{x_2} R_B) = (x_U, y_U)$

(6) Compute $K_A = KDF(x_U || y_U || Z_A || Z_B, klen)$

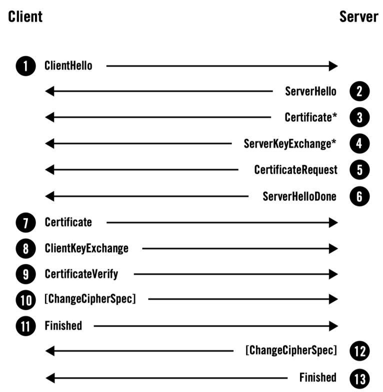
(7) Compute $S_1 = Hash(0x02 || y_U || Hash(x_U || Z_A || Z_B || x_1 || y_1 || x_2 || y_2))$, check $S_1 = S_B$

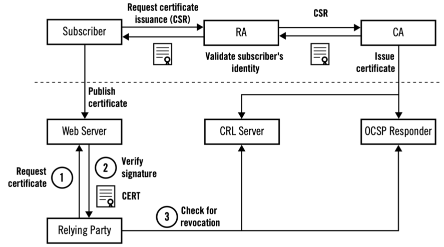
(8) Compute $S_A = Hash(0x03 || y_U || Hash(x_U || Z_A || Z_B || x_1 || y_1 || x_2 || y_2))$

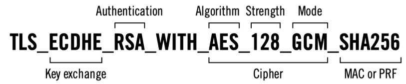
S_A

(8) Compute $S_2 = Hash(0x03 || y_V || Hash(x_V || Z_A || Z_B || x_1 || y_1 || x_2 || y_2))$ check $S_2 = S_A$

Secure Communication



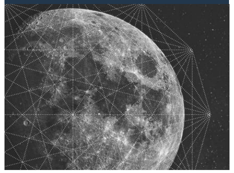




目

录

Contents



PART2 SM2 Implementation

- Scalar multiplication
- Inverse
- Public key format
- Deduce public key from signature

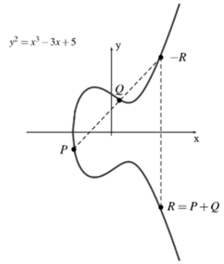
PART3 SM2 Application

2.1 Scalar multiplication Q = kP – point add and point double

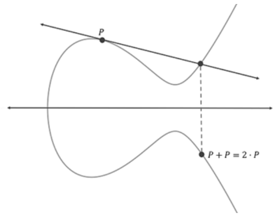
```python
    @classmethod
    def add(cls, point1: EcpPoint, point2: EcpPoint) -> EcpPoint:
        assert point1.isOnCurve() and point2.isOnCurve()
        if point1.isInf():
            return point2
        if point2.isInf():
            return point1
        if point1.x == point2.x:
            if point1.y != point2.y:
                return cls.genInf()  # Point at infinity
            else:  # P.x == Q.x && P.y == Q.y
                return cls.double(point1)
        else:  # point1.x != point2.x
            lamb = (point2.y - point1.y) * inverse_mod_prime((point2.x - point1.x) % cls.p(), cls.p()) % cls.p()

            x3 = (lamb ** 2 - point1.x - point2.x) % cls.p()
            y3 = (lamb * (point1.x - x3) - point1.y) % cls.p()

            return cls(x3, y3)
```




```python
    @classmethod
    def double(cls, point: EcpPoint) -> EcpPoint:
        assert point.isOnCurve()
        if point.isInf():
            return cls.genInf()
        lmbd = (3 * (point.x ** 2) + cls.a()) * inverse_mod_prime((2 * point.y) % cls.p(), cls.p()) % cls.p()
        x3 = (lmbd ** 2 - 2 * point.x) % cls.p()
        y3 = (lmbd * (point.x - x3) - point.y) % cls.p()
        return cls(x3, y3)
```



2.1 Scalar multiplication Q = kP – double and add

```python
    @classmethod
    def mulByScalar(cls, point: EcpPoint, scalar: int) -> EcpPoint:
        assert point.isOnCurve()

        flag = 1 << 255
        accumulator = cls.genInf()
        for i in range(255):
            if 0 != scalar & flag:
                accumulator = cls.add(accumulator, point)
            accumulator = cls.double(accumulator)
            flag >>= 1
        if 0 != scalar & flag:
            accumulator = cls.add(accumulator, point)
        return accumulator
```

Input: $\mathbf{P}$, $k = (k_{N-1} \ldots k_1 k_0)_2$ with $k_{N-1} = 1$

Output: $\mathbf{Q} = k\mathbf{P}$

$\mathbf{Q} \leftarrow \mathbf{P}$;

For $i \leftarrow N-2$ downto 0 do

Begin

  $\mathbf{Q} \leftarrow 2\mathbf{Q}$;

  If $k_i \neq 0$ then $\mathbf{Q} \leftarrow \mathbf{Q} + \mathbf{P}$;

End

2.2 Inversion

- Module inversion operation for finite field
  - Extended Euclidean algorithm
  - Fermat Little theorem
  - Constant-time extended Euclidean algorithm*

```python
from __future__ import annotations


def inverse_mod_prime(a: int, primeMod: int) -> int:
    """
    use Fermat little theorm, a^(p-2) == a^(-1) mod p
    """
    assert 0 < a < primeMod
    return pow(a, primeMod-2, primeMod)


if __name__ == '__main__':
    _x = inverse_mod_prime(5, 13)
    print(_x)
```

- Input: (r_0, r_1) *and* $r_0 > r_1$
- Output: $gcd(r_0, r_1)$ *and* s, t *s.t.* $gcd(r_0, r_1) = s \cdot r_0 + t \cdot r_1$
  - Init
    - $s_0 = 1, t_0 = 0;$
    - $s_1 = 0, t_1 = 1$
  - Init $i = 1$,
  - Do while $r_i \neq 0$:
    - $i = i + 1$
    - $r_i = r_{i-2}\ mod\ (r_{i-1})$
    - $q_{i-1} = (r_{i-2} - r_i)/r_{i-1}$
    - $s_i = s_{i-2} - q_{i-1} \cdot s_{i-1}$
    - $t_i = t_{i-2} - q_{i-1} \cdot t_{i-1}$
  - Return
    - $gcd(r_0, r_1) = r_{i-1}$
    - $s = s_{i-1}$
    - $t = t_{i-1}$

2.3 Public key format

- Public key format:
  - A point $P=(x_P, y_P)$ on elliptic curve $E: y^2=x^3+ax+b$, where x_P, y_P is 256 bits
  - Format: prefix||x||y, let $\overline{y_P}$ is the rightmost bit of y_P
    - Uncompress public key: prefix is $04||x||y$
    - Compress public key: prefix is 02 or 03
      - if y is even: $02||x$
      - if y is odd: $03||x$
  - Recover point P with x_P and $\overline{y_P}$ for E on F_p
    - Compute $\alpha=(x_P^3+ax_P+b) \; mod \; p$
    - Compute $\alpha \; mod \; p$ square root β
    - If the rightmost bit of β is $\overline{y_P}$ then set $y_P=\beta$, else set $y_P=p-\beta$

x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A
y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB

K =04F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A
07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB

2.4 Deduce public key from signature

- Recover public key from signature
  - Send tx without attach public key, improve blockchain system
- Assume precompute info Z_A is not corresponding with public key

- $s=((1+d_A)^{-1}\cdot(k-r\cdot d_A)) mod \; n$
- $s\cdot(1+d_A)=(k-r\cdot d_A) \; mod \; n$
- $(s+r)d_A=(k-s) \; mod \; n$
- $(s+r)d_AG=(k-s)G \; mod \; n$
- $d_A\cdot G=P_A=(s+r)^{-1}(kG-sG)$
- How to compute kG
  - $(kG)_x=x_1=(r-e) \; mod \; n$, then compute y_1
  - $e=Hash(Z_A||M) \; where \; Z_A$ is not related public key*

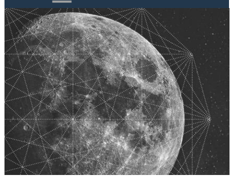
- Precompute:
  - compute $Z_A=H_{256}(ENTL_A||ID_A||a||b||x_G||y_G||x_A||y_A)$
- KeyGen:
  - $P_A=d_A\cdot G$
- Sign(M):
  - $Sign_{d_A}(M,Z_A)\to(r,s)$
  - Set $\overline{M}=Z_A||M$
  - Compute $e=H_v(\overline{M})$, where the output of H_v is v
  - Generate random number $k\in[1,n-1]$
  - Compute $kG=(x_1,y_1)$
  - Compute $r=(e+x_1) \; mod \; n$,
    - if $r=0 \; or \; r+k=n$, generate random number k again
  - Compute $s=((1+d_A)^{-1}\cdot(k-r\cdot d_A)) mod \; n$
    - if $s=0$, generate random number k again

*Project: report on the application of this deduce technique in Ethereum with ECDSA

目

录

Contents



PART3 SM2 Application

- Signature pitfalls
- SM2 signature pitfalls
- UTXO Commitment: Elliptic curve MultiSet Hash
- Private key protection via two party sign
- SM2 two party decrypt
- Google's password leaking detection
- PSI

3.1 SM2 signature: leaking k

- Precompute：
  - $Z_A=H_{256}(ENTL_A||ID_A||a||b||x_G||y_G||x_A||y_A)$
- Key Generation： $P_A=d_A\cdot G$， order is n
- Sign(Z_A, M): $Sign_{d_A}(M,Z_A)\to(r,s)$
  - Set $\overline{M}=Z_A||M$，
  - $e=H_v(\overline{M})$
  - $k\leftarrow Z_n^*$， $kG=(x_1,y_1)$
  - $r=(e+x_1) \; mod \; n$，
  - $s=((1+d_A)^{-1}\cdot(k-r\cdot d_A)) mod \; n$
  - Signature is (r,s)
- Verify (r,s) of M with P_A
  - $Z_A=H_{256}(ENTL_A||ID_A||a||b||x_G||y_G||x_A||y_A)$
  - Set $\overline{M}=Z_A||M$， $e=H_v(\overline{M})$
  - $t=(r+s) \; mod \; n$
  - $(x_1,y_1)=sG+tP_A$
  - $R=(e+x_1) \; mod \; n, \; Verify \; R=r$

- Compute d_A with $\sigma=(r,s)$ and k：
  - $s=((1+d_A)^{-1}\cdot(k-r\cdot d_A)) mod \; n$
  - $s(1+d_A)=(k-r\cdot d_A) \; mod \; n$
  - $d_A=(s+r)^{-1}\cdot(k-s) \; mod \; n$

3.1 SM2 signature: reusing k

- Precompute：
  - $Z_A = H_{256}(ENTL_A||ID_A||a||b||x_G||y_G||x_A||y_A)$
- Key Generation： $P_A = d_A \cdot G$， order is n
- Sign(Z_A, M): $Sign_{d_A}(M, Z_A) \to (r, s)$
  - Set $\overline{M} = Z_A||M$，
  - $e = H_v(\overline{M})$
  - $k \leftarrow Z_n^*$， $kG = (x_1, y_1)$
  - $r = (e + x_1) \bmod n$，
  - $s = ((1 + d_A)^{-1} \cdot (k - r \cdot d_A)) \bmod n$
  - Signature is (r, s)
- Verify (r, s) of M with P_A
  - $Z_A = H_{256}(ENTL_A||ID_A||a||b||x_G||y_G||x_A||y_A)$
  - Set $\overline{M} = Z_A||M$， $e = H_v(\overline{M})$
  - $t = (r + s) \bmod n$
  - $(x_1, y_1) = sG + tP_A$
  - $R = (e + x_1) \bmod n$, $Verify\ R = r$

- Signing message M_1 with d_A
  - Randomly select $k \in [1, n-1]$， $kG = (x, y)$
  - $r_1 = (Hash(Z_A||M_1) + x) \bmod n$
  - $s_1 = ((1 + d_A)^{-1} \cdot (k - r_1 \cdot d_A)) \bmod n$
- Signing message M_2 with d_A
  - Reuse the same k, $kG = (x, y)$
  - $r_2 = (Hash(Z_A||M_2) + x) \bmod n$
  - $s_2 = ((1 + d_A)^{-1} \cdot (k - r_2 \cdot d_A)) \bmod n$
- Recovering d_A with 2 signatures $(r_1, s_1), (r_2, s_2)$
  - $s_1(1 + d_A) = (k - r_1 \cdot d_A) \bmod n$
  - $s_2(1 + d_A) = (k - r_2 \cdot d_A) \bmod n$
  - $d_A = \frac{s_2 - s_1}{s_1 - s_2 + r_1 - r_2} \bmod n$

3.1 SM2 signature: reusing k by different users

- Precompute：
  - $Z_A = H_{256}(ENTL_A||ID_A||a||b||x_G||y_G||x_A||y_A)$
- Key Generation： $P_A = d_A \cdot G$， order is n
- Sign(Z_A, M): $Sign_{d_A}(M, Z_A) \to (r, s)$
  - Set $\overline{M} = Z_A||M$，
  - $e = H_v(\overline{M})$
  - $k \leftarrow Z_n^*$， $kG = (x_1, y_1)$
  - $r = (e + x_1) \bmod n$，
  - $s = ((1 + d_A)^{-1} \cdot (k - r \cdot d_A)) \bmod n$
  - Signature is (r, s)
- Verify (r, s) of M with P_A
  - $Z_A = H_{256}(ENTL_A||ID_A||a||b||x_G||y_G||x_A||y_A)$
  - Set $\overline{M} = Z_A||M$， $e = H_v(\overline{M})$
  - $t = (r + s) \bmod n$
  - $(x_1, y_1) = sG + tP_A$
  - $R = (e + x_1) \bmod n$, $Verify\ R = r$

- Alice signed message M_1 with $d_A, \sigma_A = (r_1, s_1)$
  - Randomly select $k \in [1, n-1]$， $kG = (x, y)$
  - $r_1 = (Hash(Z_A||M_1) + x) \bmod n$
  - $s_1 = ((1 + d_A)^{-1} \cdot (k - r_1 \cdot d_A)) \bmod n$
- Bob signed message M_2 with $d_B, \sigma_B = (r_2, s_2)$
  - Reuse the same k, $kG = (x, y)$
  - $r_2 = (Hash(Z_B||M_2) + x) \bmod n$
  - $s_2 = ((1 + d_B)^{-1} \cdot (k - r_2 \cdot d_B)) \bmod n$
- Alice can deduce Bob secret key
  - $d_B = \frac{k - s_2}{s_2 + r_2} \bmod n$
- Bob can deduce Alice secret key
  - $d_A = \frac{k - s_1}{s_1 + r_1} \bmod n$

Project 5+: impl sm2 with RFC6979

3.1 Signatures – ECDSA, SM2, Schnorr

ECDSA

- Key Gen: $P = dG$, n is order
- Sign(m)
  - $k \leftarrow Z_n^*, R = kG$
  - $r = R_x \bmod n, r \neq 0$
  - $e = hash(m)$
  - $s = k^{-1}(e + dr) \bmod n$
  - Signature is (r, s)
- Verify (r, s) of m with P
  - $e = hash(m)$
  - $w = s^{-1} \bmod n$
  - $(r', s') = e \cdot wG + r \cdot wP$
  - Check if $r' == r$
  - Holds for correct sig since
  - $es^{-1}G + rs^{-1}P = s^{-1}(eG + rP) =$
  - $k(e + dr)^{-1}(e + dr)G = kG = R$

SM2

- Precompute：
  - $Z_A = H_{256}(ENTL_A||ID_A||a||b||x_G||y_G||x_A||y_A)$
- Key Generation： $P_A = d_A \cdot G$， order is n
- Sign(Z_A, M): $Sign_{d_A}(M, Z_A) \to (r, s)$
  - Set $\overline{M} = Z_A||M$，
  - $e = H_v(\overline{M})$
  - $k \leftarrow Z_n^*$， $kG = (x_1, y_1)$
  - $r = (e + x_1) \bmod n$，
  - $s = ((1 + d_A)^{-1} \cdot (k - r \cdot d_A)) \bmod n$
  - Signature is (r, s)
- Verify (r, s) of M with P_A
  - $Z_A = H_{256}(ENTL_A||ID_A||a||b||x_G||y_G||x_A||y_A)$
  - Set $\overline{M} = Z_A||M$， $e = H_v(\overline{M})$
  - $t = (r + s) \bmod n$
  - $(x_1, y_1) = sG + tP_A$
  - $R = (e + x_1) \bmod n$, $Verify\ R = r$

Schnorr two variants

- Key Generation
  - $P = dG$
- Sign on given message M
  - randomly k, let $R = kG$
  - $e = hash(R||M)$
  - $s = k + ed \bmod n$
  - Signature is : (R, s)
- Verify (R, s) of M with P
  - Check sG vs $R + eP$
  - $sG = (k + ed)G = kG + edG = R + eP$

- Key Generation
  - $P = dG$
- Sign on given message M
  - randomly k, let $R = kG$
  - $e = hash(R||M)$
  - $s = r - ex \bmod n$
  - Signature is : (e, s)
- Verify (e, s) of M with P
  - Compute $R' = sG + eP$
  - Check $hash(R'||M)$ vs e
  - $R' = (r - ed)G + eP = rG$

3.2 SM2 signature: same d and k with ECDSA

- ECDSA signing with private key d
  - Randomly select k, $R = kG = (x, y)$
  - $e_1 = hash(m)$
  - $r_1 = x \bmod n, s_1 = (e_1 + r_1 d)k^{-1} \bmod n$
  - Signature (r_1, s_1)
- SM2 signing with private key d
  - Reuse the same k as ECDSA, $(x, y) = kG$
  - $e_2 = h(Z_A||m)$
  - $r_2 = (e_2 + x) \bmod n$
  - $s_2 = (1 + d)^{-1} \cdot (k - r_2 d) \bmod n$
  - Signature (r_2, s_2)
- With the two sigs, private key d can be recovered:
  - $d \cdot r_1 = ks_1 - e_1 \bmod n$
  - $d \cdot (s_2 + r_2) = k - s_2 \bmod n$
  - $d = \frac{s_1 s_2 - e_1}{(r_1 - s_1 s_2 - s_1 r_2)} \bmod n$

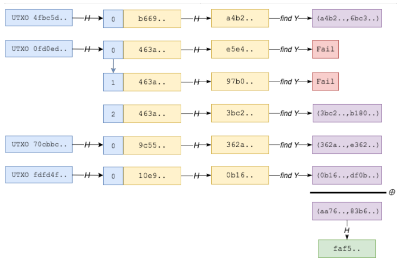
3.1 Signatures pitfalls summary

pitfalls	ECDSA	Schnorr	SM2-sig
Leaking k leads to leaking of d	✓	✓	✓
Reusing k leads to leaking of d	✓	✓	✓
Two users, using k leads to leaking of d, that is they can deduce each other's d	✓ RFC 6979	✓ RFC 6979	✓
Malleability, e.g. (r,s) and $(r,-s)$ are both valid signatures, lead to blockchain network split	✓	✓	$r=(e+x_1)\ mod\ n$ $e=Hash(Z_A\|\|M)$
Ambiguity of DER encode could lead to blockchain network split	✓	✓	----
One can forge signature if the verification does not check m	✓	✓	✓
Same d and k with ECDSA, leads to leaking of d	✓	✓	✓

Project 5: verify the above pitfalls with proof-of-concept code with SM2 (ECDSA & Schnorr are optional)

3.3 UTXO Commitment: Elliptic curve MultiSet Hash

- Homomorphic, or incremental, multiset hash function
  - hash({a}) + hash({b}) = hash({a,b})
- Basic idea: hash each element to an EC point (try and increment)
  - An empty set maps to the infinity point of EC
- Combine/add/remove elements → Points Add of corr. EC Point
- The order of the elements in the multiset does not matter
- Duplicate elements are possible, {a} and {a, a} have different digest
- To update the digest of a multiset, only needs to compute the difference
- Can be constructed on any elliptic curve
- Collision resistant relies on hardness of ECDLP
  - The same security assumption as SM2/ECDSA sign/verify
  - Need more eyes to investigate the security proof of ECMH (maybe worry too much)
- Gains: fast node synchronization, no need to start from the beginning
- Still: you cannot prove to others that you own some Bitcoin efficiently



*Project: Implement the above ECMH scheme

3.5 SM2 two-party sign

- Public key: $P=[(d_1d_2)^{-1}-1]G$
- Private key: $d=(d_1d_2)^{-1}-1$

- Signature
  - $(k_1k_3+k_2)G=(x_1,y_1)$
  - $r=(x_1+e)\ mod\ n$
  - $s=(1+d)^{-1}\cdot\left((k_1k_3+k_2)-r\cdot d\right)\ mod\ n$

(1) Generate sub private key $d_1\in[1,n-1]$,
compute $P_1=d_1^{-1}\cdot G$

$\xrightarrow{P_1}$

(1) Generate sub private key $d_2\in[1,n-1]$,
(2) Generate shared public key: compute $P=d_2^{-1}\cdot P_1-G$,
publish public key P

(3) Set Z to be identifier for both parties, message is M
- Compute $M'=Z\|M$, $e=Hash(M')$
- Randomly generate $k_1\in[1,n-1]$, compute $Q_1=k_1G$

$\xrightarrow{Q_1,e}$

(4) Generate partial signature r:
- Randomly generate $k_2\in[1,n-1]$, compute $Q_2=k_2G$
- Randomly generate $k_3\in[1,n-1]$, compute $k_3Q_1+Q_2=(x_1,y_1)$
- Compute $r=x_1+e\ mod\ n\quad(r\neq 0)$
- Compute $s_2=d_2\cdot k_3\ mod\ n$,
- Compute $s_3=d_2(r+k_2)mod\ n$

$\xleftarrow{r,s_2,s_3}$

(5) Generate signature $\sigma=(r,s)$
- Compute $s=(d_1*k_1)*s_2+d_1*s_3-r\ mod\ n$
- If $s\neq 0$ or $s\neq n-r$, output signature $\sigma=(r,s)$

Project 5: implement sm2 2P sign with real network communication

3.6 SM2 two-party decrypt

- Public key: $P=[(d_1d_2)^{-1}-1]G$
- Private key: $d=(d_1d_2)^{-1}-1$

- Encrypt:
  - $C_1=kG=(x_1,y_1)\ where\ k\in[1,n-1]$
  - $kP=(x_2,y_2)$
  - $t=KDF(x_2\|y_2,klen)$
  - $C_2=M\oplus t$
  - $C_3=H(x_2\|M\|y_2)$

(1) Generate sub private key $d_1\in[1,n-1]$,

(1) Generate sub private key $d_2\in[1,n-1]$

(2) get ciphertext $C=C_1\|C_2\|C_3$
- Check $C_1\neq 0$
- Compute $T_1=d_1^{-1}\cdot C_1$

$\xrightarrow{T_1}$

(3) compute $T_2=d_2^{-1}\cdot T_1$,

$\xleftarrow{T_2}$

(4) Recover plaintext M'
- Compute $T_2-C_1=(x_2,y_2)=[(d_1d_2)^{-1}-1]\cdot C_1=kP$
- Compute $t=KDF(x_2\|y_2,klen)$
- Compute $M''=C_2\oplus t$
- Compute $u=Hash(x_2\|M''\|y_2)$
- If $u=C_3$, output M''

*Project: implement sm2 2P decrypt with real network communication

3.7 Google Password Checkup

- Username and password detection

*Project: PoC impl of the scheme, or do implement analysis by Google

client

Google server $sk = b$

(1) Process data info

- Data records: $(userName, password) \rightarrow (u_i, p_i)$
- Create key-value table (1TB): (k_i, v_i)
  - compute $h_i = Argon2(u_i, p_i)$
  - k_i is the first two bytes of h_i, namely $k_i = h_i[:2]$
  - $v_i = (h_i)^b$
- Divide the table into 2^{16} sets according to the key k_i (2 bytes)

(2) User input name and password: (u, p)

- Client generate ephemeral secret key: $sk_c = a$
- Client compute key-value: (k, v)
  - compute $h = Argon2(u, p)$
  - $k = h[:2]$
  - $v = h^a$

(k, v) →

(3) Find the data set

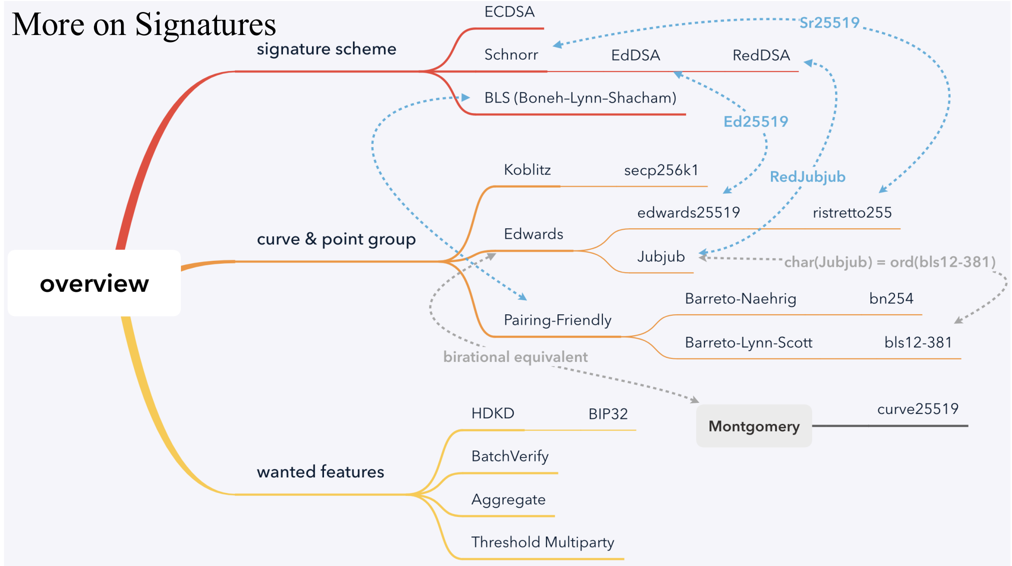
- compute h^{ab}
- Find set S according to key k

← h^{ab}, data set S

(4) Username and password detection

- Compute $(h^{ab})^{a^{-1}} = h^b$
- Check whether h^b exists in S

Conclusion: The client knows whether its userName and password are leaked, but cannot obtain any other information about the set S returned by the server



THANKS
QUESTIONS TIME