CSIT6910 Project Report

Ring Signature Research in Privacy-preserving Blockchain Application

Niu Yutong (20176192)

yniuaa@connect.ust.hk

Table of Contents

Abstract

This project reviews and implements the ring signature research works and the application of them in modern privacy-preserving blockchain systems. Source code for the project can be found in this GitHub repo.

The first phase of the project researches the implementation of Bitcoin. The focus is on its ECC signature scheme and serialization for communication in p2p network. The technical design for Bitcoin is described in detail in the Book Programming Bitcoin and an implementation from scrach can be found in this GitHub repo. Two major takes-aways from the Bitcoin book are the implementation of ECC signature scheme and the serialization of Bitcoin data in p2p network. Since the book uses Python and builds everything from scratch, it becomes a good starting point for exploring the different ring signature scheme and implementing a Monero-like cryptocurrency in the following phases, especially the implementation of ECC signature library. Actually only the ECC library and some auxiliary helper functions are re-used in codes of phase 2 and 3.

The second phase explores different ring signature schemes. 6 papers are selected and implemented based on the ECC signature library from phase 1 only. Here is the list of 6 papers. And in the rest of this report, these papers will be only referenced by the code.

Code	Year	Title
RST01	2001	How to leak a secret
AOS02	2002	1-out-of-n signatures from a variery of keys
Borromean	2016	Borromean Ring Signatures
LWW04	2004	Linkable spontaneous anonymous group signature for ad hoc groups
Bac15	2015	Adam Back.Ring signature efficiency
MLSAG	2016	Ring Confidential Transactions

Each paper above describes a ring signature scheme. And the 6 schemes are all implemented and tested. The codes can be found in the paper sub-directory.

The third phase implements a Monero-like cryptocurrency. All the three main technologies of Monero are implemented, including stealth address, ring signature and RingCT. The system is dockerized for deployment and the docker image can be found on DockerHub as yniuaa/ring. The docker image is built on an Apple Silicon with ARM64 architecture. So to deploy it on any other architectures, like x86 requires an docker re-build. The detailed deployment guide can be found in Part IV of the report.

Part I: Take-aways from Bitcoin Implementation

Bitcoin and Monero have a lot of things shared in common. They both use Proof of Work for mining and Elliptic Curves for signature scheme. Altough bicoin chose the curve secp256k1, while Monero chose the curve edwards25519, the underlying ECC signature nature is the same. In the implementation of the Monero-like cryptocurrency in Phase 3, the same curve as Bitcoin, secp256k1 is chosen, simply because the pure Python library of ECC is available from the Programming Bitcoin book.

ECC Signature

Secp256k1 curve

The ECC library implements the ECC signature scheme on the curve secp256k1 from scratch. It does not depend on any other third-party packages. The curve has the following parameters:

```
# curve: y ^ 2 = x ^ 3 + 7
# y ^ 2 = x ^ 3 + A * x + B
# A = 0
A = 0
# B = 7
B = 7
# Finite Field Fp
# P = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F
P = 2**256 - 2**32 - 977
# Generator Point
G = S256Point(
    0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798,
    0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8)
# Order of G
# N * G = Point(Infinity)
N = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141
```

Keys

The library starts from the Finite Field class to Point the ECC curve. It represents the secret key ***sk*** as an integer and the verification key ***vk*** as an S256 Point, where $vk = sk \times G$.

The verification key (public key) is represented as a `S256Point`. The point is on the curve secp256k1 ($y^2 = x^3 + 7$). And x,y-coordinates are both Finite Field element F_p with $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$.

```python
class S256Point(Point):

    def __init__(self, x, y, a=None, b=None):
        # a = 0, b = 7
        # curve: y^2 = x^3 + 7
        a, b = S256Field(A), S256Field(B)
        if type(x) == int:
            super().__init__(x=S256Field(x), y=S256Field(y), a=a, b=b)
        else:
            super().__init__(x=x, y=y, a=a, b=b)

    def __repr__(self):
        if self.x is None:
            # Inifity point = N * G
            return 'S256Point(infinity)'
        else:
            return 'S256Point({}, {})'.format(self.x, self.y)

    # P(x1, y1) - P(x2, y2) = P(x1, y1) + P(x2, -y2)
    def __sub__(self, other):
        return self + self.__class__(other.x,  S256Field(0)-other.y)

    def __rmul__(self, coefficient):
        coef = coefficient % N
        return super().__rmul__(coef)

    def __hash__(self):
        return int(hashlib.sha1(self.sec()).hexdigest(), 16)
```

The `S256Point` class supports addition, subtraction and multiplication with integer for S256Point, which is sufficient for ECC signature scheme.

Signature

The sk and vk key pair is defined in class `PrivateKey`. The class also has a `sign` method to sign the message with ECDSA. The signing procdedure is described below: ($e \leftarrow sk, P \leftarrow vk$)

$$P = e \times G \tag{1}$$

k is a random 256-bit number

$$R = k \times G = u \times G + v \times P \tag{2}$$

$$P = \frac{k-u}{v} \times G = e \times G \implies u + v \times e = k \tag{3}$$

z is the signature hash and r is the x-coordinate of R

$$u = \frac{z}{s} \ and \ v = \frac{r}{s} \implies s = \frac{(z + r \times e)}{k} \tag{4}$$

The `sign` method has the prototype of $Sign(z) = (r, s)$

```
class PrivateKey:

    def __init__(self, secret):
        self.secret = secret
        self.point = secret * G

    def hex(self):
        return '{:x}'.format(self.secret).zfill(64)

    def sign(self, z):
        k = self.deterministic_k(z)
        # r is the x coordinate of the resulting point k*G
        r = (k * G).x.num
        # remember 1/k = pow(k, N-2, N)
        k_inv = pow(k, N - 2, N)
        # s = (z+r*secret) / k
        s = (z + r * self.secret) * k_inv % N
        if s > N / 2:
            s = N - s
        # return an instance of Signature:
        # Signature(r, s)
        return Signature(r, s)
```

Verification

The `verify` method is straightfoward. The signature, signature hash is taken as input. The protoype is like $Verify(z, (r, s)) = Bool$. The value u and v can be calculated from r and s, since

$$u = \frac{z}{s} \ and \ v = \frac{r}{s} \tag{5}$$

Then R can be calculated as

$$R = u \times G + v \times P \tag{6}$$

To verify, just to check if the x-coordinate of R equals to r. Below is the code for ECC signature verification.

```
class S256Point(Point):
    ...
    def verify(self, z, sig):
        # By Fermat's Little Theorem, 1/s = pow(s, N-2, N)
        s_inv = pow(sig.s, N - 2, N)
        # u = z / s
        u = z * s_inv % N
        # v = r / s
        v = sig.r * s_inv % N
        # u*G + v*P should have as the x coordinate, r
        total = u * G + v * self
        return total.x.num == sig.r
```

Although the ECDSA scheme is never used in the ring signature algorithms below, the method itself lays the foudation for the ring signature schemes. And the class `S256Point` is repeatedly imported in the ring signature and Monerno address modules.

Serialization

Another important take-away from Bitcoin implementation is the serialization of Python classes for transmission purpose on the p2p network or to the disk.

SEC Format for ECC Public Key

Serialization of ECC Public Key (class `S256Point`) is used a lot in the following implementation. Here the compressed SEC(Standards for Efficient Cryptography) format is chosen for serialization. The compressed version of SEC represents each `S256Point` object by 33 bytes. One byte for prefix and 32 bytes for x-coordinate for the `S256Point`.

```
0349fc4e631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278a

            - 02 if y is even, 03 if odd - Marker
            - x coordinate - 32 bytes
```

The implementation for SEC format is like below:

```
class S256Point(Point):
    ...
    def sec(self, compressed=True):
        '''returns the binary version of the SEC format'''
        # if compressed, starts with b'\x02' if self.y.num is even, b'\x03' if self.y
```

```
is odd
        # then self.x.num
        # remember, you have to convert self.x.num/self.y.num to binary
(some_integer.to_bytes(32, 'big'))
        if compressed:
            if self.y.num % 2 == 0:
                return b'\x02' + self.x.num.to_bytes(32, 'big')
            else:
                return b'\x03' + self.x.num.to_bytes(32, 'big')
        else:
            # if non-compressed, starts with b'\x04' followod by self.x and then self.y
            return b'\x04' + self.x.num.to_bytes(32, 'big') + \
                self.y.num.to_bytes(32, 'big')
```

For every serialization in the project, there is a parse function related to it. The parse function does exactly the opposite of serialization. It takes a series a bytes and return a corresponding object instead. Here is the parse function for `S256Point`:

```
class S256Point(Point):
    ...
    @classmethod
    def parse(self, sec_bin):
        '''returns a Point object from a SEC binary (not hex)'''
        if sec_bin[0] == 4:
            x = int.from_bytes(sec_bin[1:33], 'big')
            y = int.from_bytes(sec_bin[33:65], 'big')
            return S256Point(x=x, y=y)
        is_even = sec_bin[0] == 2
        x = S256Field(int.from_bytes(sec_bin[1:], 'big'))
        # right side of the equation y^2 = x^3 + 7
        alpha = x**3 + S256Field(B)
        # solve for left side
        beta = alpha.sqrt()
        if beta.num % 2 == 0:
            even_beta = beta
            odd_beta = S256Field(P - beta.num)
        else:
            even_beta = S256Field(P - beta.num)
            odd_beta = beta
        if is_even:
            return S256Point(x, even_beta)
        else:
            return S256Point(x, odd_beta)
```

Note that in both serialization and parse function, the SEC format has two forms, compressed form and uncompressed form. The uncompressed form includes both x-coordination and y-coordination of the Point and occupies double space as the compressed form. Since the point has to be on secp256k1 curve, the coordinates for both x and y axis are redundant. Hence, the compressed form is always preferred.

Little Endian Redux

Although the Big Endian is used for SEC format above, the rest of the implementation uses the Little Endian Redux. Here provides the function to convert between integers and bytes object with Little Endian.

```python
def little_endian_to_int(b):
    '''little_endian_to_int takes byte sequence as a little-endian number.
    Returns an integer'''
    return int.from_bytes(b, 'little')


def int_to_little_endian(n, length):
    '''endian_to_little_endian takes an integer and returns the little-endian
    byte sequence of length'''
    return n.to_bytes(length, 'little')
```

Variable Integer

Sometimes, it is hard to tell how many bytes an integer requires. Here the variable integer is used for serialization. It encodes integer to bytes that range from 0 to $2^{64} - 1$. It saves space if the input is small and would not take as many as 8 bytes. Here is the implementation.

```python
def read_varint(s):
    '''read_varint reads a variable integer from a stream'''
    i = s.read(1)[0]
    if i == 0xfd:
        # 0xfd means the next two bytes are the number
        return little_endian_to_int(s.read(2))
    elif i == 0xfe:
        # 0xfe means the next four bytes are the number
        return little_endian_to_int(s.read(4))
    elif i == 0xff:
        # 0xff means the next eight bytes are the number
        return little_endian_to_int(s.read(8))
    else:
        # anything else is just the integer
        return i


def encode_varint(i):
    '''encodes an integer as a varint'''
    if i < 0xfd:
        return bytes([i])
    elif i < 0x10000:
        return b'\xfd' + int_to_little_endian(i, 2)
    elif i < 0x100000000:
        return b'\xfe' + int_to_little_endian(i, 4)
    elif i < 0x10000000000000000:
        return b'\xff' + int_to_little_endian(i, 8)
    else:
        raise ValueError('integer too large: {}'.format(i))
```

Part II: Ring Signature Algorithm Implementation

This section discusses the implementation of different ring signature schemes. The term of ring signature originated from paper **RST01**. It defines ring signature as a way to leak a secret. It depends on the one-way trap door function for public key encryption scheme like RSA. Since RSA does not scale as well as ECC, the paper **AOS02** further develops the work from RST01 to DL-type keys. Based on that, **Borromean** ring signature generalizes the construction to handle conjunctive statements from the OR-only disjunctive statement. The three other papers explores the linkability of the ring signature. The linkability is defined as, the signature produced by a signer can be linked together. Although the identity of the sign remains unknown, it can be proved that several signatures are produced by the same signer. **LWW04** provides the first implementation of ring signature with linkability property, Linkable Spontaneous Anonymous Group (LSAG). **Bak15** continues the result from LSAG to propose a way to reduce the size of the ring signature. Finally, **MLSAG** is proposed by RingCT. Like Borromean ring, in MLSAG, the keys form key vectors. The signer proves s/he knows the corresponding private key for every public key in an unknown key vector, which makes the idea of amount hidden transaction possible.

The code implementation for different ring signatures can be find in papers sub-directory. Each algorithm also comes with a unittest case, which tests the signature and its verification. To run any test case, just *change directory* to the target tests folder, and run

```
python -m unittest test_ring
```

The command may differ for different platform, please check the unittest module documentation.

RST01

This paper is a classic since it defines the term ring signature. It firstly proposes ring signature as a scheme to leak a secret, to authorize the source of a secret leakage without leaking which exact person is the whistleblower. It relies on the trap-door function for signature scheme such as RSA.

Since each ring member is a RSA public key, each key has a different domain size. Before forming the ring, the trap-door permutations of all the ring members need to be extended to the same domain size. The method is described below, where $f_i(x) = x^{e_i} \ (mod\ n_i)$ is the one-way permutation for RSA public key $P_i = (n_i, e_i)$.

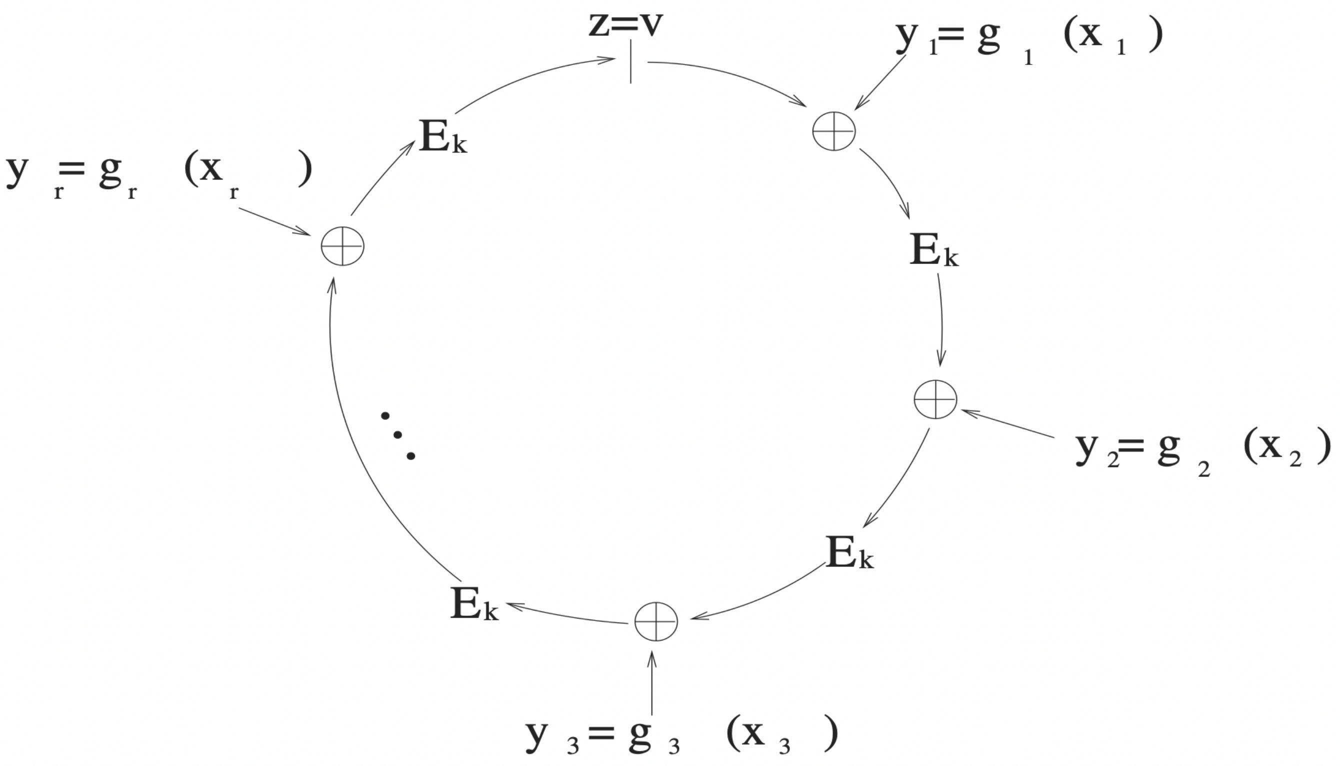
$$g_i(m) = q_i \times n_i + f_i(r_i)\ if (q_i + 1)n_i \leq 2^b$$

$$g_i(m) = m\ else$$

The algorithm also requires a keyed hash function. For simplity, the keyed hash function defined as

$$E_k(x) = H(k|x)$$

where k is the hash result of the input message to be signed.



The implementation of **RST01** can be found below.

```
class Ring:
    """RSA implementation."""
    def __init__(self, k, L = 1024):
        self.k = k
        self.l = L
        self.n = len(k)
        self.q = 1 << (L - 1)

    def sign_message(self, m, z):
        self._permut(m)
        s = [None] * self.n
        u = random.randint(0, self.q)
        c = v = self._E(u)

        first_range = list(range(z + 1, self.n))
        second_range = list(range(z))
        whole_range = first_range + second_range

        for i in whole_range:
            s[i] = random.randint(0, self.q)
            e = self._g(s[i], self.k[i].e, self.k[i].n)
            v = self._E(v ^ e)
            if (i + 1) % self.n == 0:
                c = v

        s[z] = self._g(v ^ u, self.k[z].d, self.k[z].n)
        return [c] + s

    def verify_message(self, m: str, X):
        self._permut(m)

        def _f(i):
            return self._g(X[i + 1], self.k[i].e, self.k[i].n)

        y = map(_f, range(len(X) - 1))
        y = list(y)

        def _g(x, i):
            return self._E(x ^ y[i])
        r = functools.reduce(_g, range(self.n), X[0])
        return r == X[0]

    def _permut(self, m):
        msg = m.encode("utf-8")
        self.p = int(hashlib.sha1(msg).hexdigest(), 16)

    def _E(self, x):
        msg = f"{x}{self.p}".encode("utf-8")
        return int(hashlib.sha1(msg).hexdigest(), 16)

    def _g(self, x, e ,n):
        q, r = divmod(x, n)
        if((q + 1) * n) <= ((1 << self.l) - 1):
            result = q * n + pow(r, e, n)
        else:
            result = x
        return result
```

The `Ring` class takes a list of RSA public keys as input for initialization. The one-way trap-door permutation is defined as `Ring._g`, it extends all the RSA keys to the same domain. The algorithm is very efficient and scales well. The signing process takes in the message to be signed. The signing starts from the real signer and initializes the signer a random number u, u is defined as the input for function E_k right after the signer and also the XOR result between y_z and the result of the last E_k, v_{z-1}. And it generates each ring member a random x_i. Looping through the ring with the previously defined E_k and XOR functions. The ring comes back to the signer with the result of the last E_k, v_{z-1}. Since

$$u = v_{z-1} \oplus y_z = v_{z-1} \oplus g_z(x_z)$$

y_z can be calculated from the result $u \oplus v_{z-1}$. From the one-way permutation function g_z, the value of x_z can be revealed.

The verification is straighforward. Start from the first ring member and go through the ring. If after the loop the value remains the same, then the verification suceeds.

AOS02

This paper generalizes the ring signature from the one-way trapdoor scheme like RSA. DL-type keys like ECDSA signature scheme is also supported. It even supports different kinds of keys mixed together to produce a signature. The methodology behind this is to define three steps for both one-way type(like RSA) and three-move type(like ECDSA). The three steps are initialization, forward sequence, and forming the ring. The implementation supports both RSA and ECC keys, but here only ECC keys are described. For the full implementation, please refer to the source code.

Refer to the ECDSA sign/vefify steps described in phase 3. The signing algorithm can be divided into three steps.

$$sk : x$$
$$vk : P = x \times G$$

$S_{sk}^{sig}(m) =$

1. $a \leftarrow A(r) = r \times G$ (r is randomly selected)
2. $c = H(m, a)$ (H is a hash function)
3. $s = Z(x, r, c) = c \times x + r$
4. return the signature $\sigma = (s, c)$

The verifying algorithm algorithm is described as: $V_{vk}^{sig}(m, \sigma) =$

1. $\sigma = (s, c)$
2. $z = V(s, c, P) = s \times G - c \times P = (c \times x + r) \times G - cx \times G = r \times G$
3. $e = H(m, z)$ (H is the same hash function)
4. return 1 if $c = e$. otherwise 0

The support functions: A, H, Z, V are implemented below:

```
class AOSRing:

    def __init__(self, k, L = 1024):
        self.k = k
        self.vk_serialize()
        self.q = 1 << (L - 1)
        self.l = L
        self.n = len(k)

    def vk_serialize(self):
        self.L = b''
        for key in self.k:
            if isinstance(key, Crypto.PublicKey.RSA.RsaKey):
                self.L += key.publickey().export_key('DER')
            elif isinstance(key, EccKey):
                self.L += key.point.sec()
            else:
                raise TypeError('Only RSA or ECC key is allowed')

    @staticmethod
    def F(s, k):
        if not isinstance(k, Crypto.PublicKey.RSA.RsaKey):
            raise TypeError('Only RSA key is allowed')
        return pow(s, k.e, k.n)

    @staticmethod
    def I(c, k):
        if not isinstance(k, Crypto.PublicKey.RSA.RsaKey):
            raise TypeError('Only RSA key is allowed')
        return pow(c, k.d, k.n)

    @staticmethod
    def A():
        return random.randint(0, EccOrder)

    @staticmethod
    def Z(r, c, k):
        if not isinstance(k, EccKey):
            raise TypeError('Only ECC key is allowed')
        return (r + c * k.secret) % EccOrder

    @staticmethod
    def V(s, c, k):
        if not isinstance(k, EccKey):
            raise TypeError('Only ECC key is allowed')
        return s * EccGenerator - c * k.point

    def H(self, m, e):
        h = hashlib.sha1(self.L)
        h.update(m.encode('utf-8'))
        h.update(e.to_bytes(self.l, 'big'))
        return int(h.hexdigest(),16)
```

The ring signature generation algorithm also has three steps:

1. Initialization:
   $e_k = A(\alpha)$
   $c_{k+1} = H_{k+1}(L, m, e_k)$
2. Forward Sequence:
   For $i = k+1, .., n-1, 0, .., k-1$, compute
   $e_i = V_i(s_i, c_i, vk_i)$
   $c_{i+1} = H_{i+1}(L, m, e_i)$
   (s_i is randomly selected)

3. Forming the ring:
   $s_k = Z_k(sk_k, \alpha, c_k)$
4. return signature $\sigma = (c_0, s_0, s_1, ..., s_{n-1})$

```
class AOSRing:
    ...
    def sign(self, m, z):
        e = [None] * self.n
        c = [None] * self.n
        s = [None] * self.n

        _alpha = None

        ## Initialization
        _alpha = self.A()
        e[z] = int.from_bytes((_alpha * EccGenerator).sec(), 'big')
        c[(z+1) % self.n] = self.H(m, e[z])

        ## Forward sequence
        first_range = list(range(z + 1, self.n))
        second_range = list(range(z))
        whole_range = first_range + second_range

        for i in whole_range:
            s[i] = random.randint(0, self.q)
            e[i] = int.from_bytes(self.V(s[i], c[i], self.k[i]).sec(), 'big')
            c[(i+1) % self.n] = self.H(m, e[i])

        ## Forming the ring
        s[z] = self.Z(_alpha, c[z], self.k[z])

        return [c[0]] + s
```

The signature verification process is described below:
For $i = 0, ..., n-1$, compute

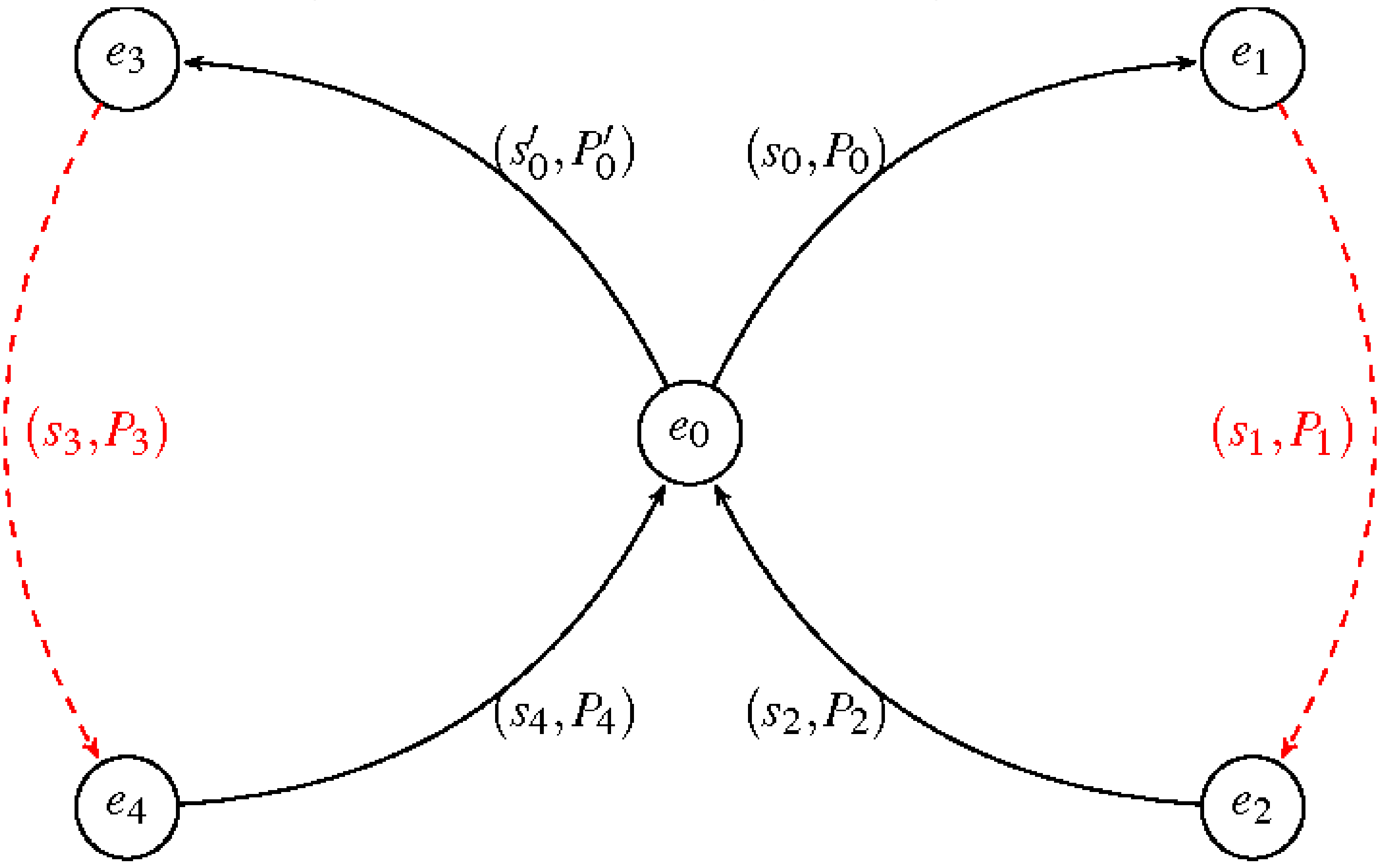
$$e_i = V_i(s_i, c_i, vk_i)$$
$$c_{i+1} = H_{i+1}(L, m, e_i)\ if\ i \neq n-1$$

Accept if $c_0 = H_0(L, m, e_{n-1})$, reject otherwise

```
class AOSRing:
    ...
    def verify(self, m, sig):
        e = [None] * self.n
        c = [None] * self.n
        c[0] = sig[0]
        s = sig[1:]
        for i in range(self.n):
            if isinstance(self.k[i], Crypto.PublicKey.RSA.RsaKey):
                e[i] = c[i] + self.F(s[i], self.k[i])
            elif isinstance(self.k[i], EccKey):
                e[i] = int.from_bytes(self.V(s[i], c[i], self.k[i]).sec(), 'big')
            else:
                raise TypeError('Only RSA or ECC key is allowed')
            if i != self.n - 1:
                c[i + 1] = self.H(m, e[i])
        return c[0] == self.H(m, e[self.n - 1])
```

Borromean

This paper further generalizes the ring signature to describe signature signed with arbitrary functions of the signing keys. The implementation is about one concrete example. There are n rings and the signer proves his/her knowledge on at least one secret for the key on each ring.



For the example in the figure above, the knowledge to be proved is $(P_0|P_1|P_2)\&(P_0'|P_3|P_4)$.

Signing Procedure:

1. $M = H(m, \{P_{i,j}\}_{i=0}^{n-1})$
2. For each $0 \leq i \leq n-1$
   i. k_i is chosen randomly
   ii. $e_{i,j_i^*+1} = H(M\|k_iG\|i\|j_i^*)$
   iii. For j such that $j_i^* \leq j < m_{i-1}$: $e_{i,j+1} = H(M\|s_{i,j}G - e_{i,j}P_{i,j}\|i\|j)$
3. Choose s_{i,n_j} for each i at randome and set.
   $e_0 = H(s_{i,m_j}G - e_{i,m_j}P_{i,j}\|...\|s_{n,m_j}G - e_{n,m_j}P_{i,j})$

e_0 commits to several s values, one from each ring.

4. For each $0 \leq i \leq n-1$
   i. For j such that $0 \leq j < j_i^* - 1$: $e_{i,j+1} = H(M \| s_{i,j}G - e_{i,j}P_{i,j} \| i \| j)$
   ii. Set $s_{i,j_i^*} = k_i + x_i e_{i,j_{i-1}^*}$
5. Return signature $\sigma = \{e_0, s_{i,j} : 0 \leq i \leq n, 0 \leq j \leq m_i\}$

```
class BorromeanRing:

    def __init__(self, rings):
        self.rings = rings
        self.n = len(rings)
        self.vk_serialize()

    def vk_serialize(self):
        self.L = b''
        for ring in self.rings:
            for key in ring:
                if not isinstance(key, EccKey):
                    raise TypeError('Only ECC key is allowed')
                self.L += key.point.sec()
    ...
    def sign(self, m, i_sk):
        if len(i_sk) != self.n:
            raise ValueError('No. of sk not correct')

        M = self.H([m, self.L])
        k = [None] * self.n
        e = [[None] * len(self.rings[i]) for i in range(self.n)]
        s = [[None] * len(self.rings[i]) for i in range(self.n)]

        hashin = []

        for i in range(self.n):
            k[i] = random.randint(0, EccOrder)
            m_i = len(self.rings[i])
            j_i = i_sk[i]
            if j_i == m_i - 1:
                hashin.append(k[i] * EccGenerator)
                continue
            else:
                e[i][j_i+1] = self.H([M, k[i] * EccGenerator, i, j_i])
            for j in range(j_i+1, m_i-1):
                s[i][j] = random.randint(0, EccOrder)
                e[i][j+1] = self.H([
                    M,
                    s[i][j] * EccGenerator - e[i][j] * self.rings[i][j].point,
                    i,
                    j,
                ])
            s[i][m_i - 1] = random.randint(0, EccOrder)

            hashin.append(s[i][m_i - 1] * EccGenerator - e[i][m_i - 1] * self.rings[i]
[m_i - 1].point)
        e0 = self.H(hashin)
        for i in range(self.n):
            j_i = i_sk[i]
            e[i][0] = e0
            for j in range(j_i):
                s[i][j] = random.randint(0, EccOrder)
                e[i][j+1] = self.H([
                    M,
                    s[i][j] * EccGenerator - e[i][j] * self.rings[i][j].point,
                    i,
                    j,
                ])
            s[i][j_i] = k[i] + self.rings[i][j_i].secret * e[i][j_i]

        return [e0, s]
```

Verification Procedure:

1. $M = H(m, \{P_{i,j}\}_{i=0}^{n-1})$
2. For each $0 \leq i \leq n-1$, for each $0 \leq j \leq m_j - 1$
    i. $R_{i,j+1} = s_{i,j}G + e_{i,j}P_{i,j}$
    ii. $e_{i,j+1} = H(M \| R_{i,j+1} \| i \| j+1)$
3. Compute: $e'_0 = H(R_{0,m_0} \| ... \| R_{n,m_n})$
4. Accept if $e_0 = e'_0$. Reject otherwise

```
class BorromeanRing:
    ...
    def verify(self, m, sig):
        M = self.H([m, self.L])
        e = [[None] * (len(self.rings[i])+1) for i in range(self.n)]
        R = [[None] * (len(self.rings[i])+1) for i in range(self.n)]
        e0 = sig[0]
        s = sig[1]
        for i in range(self.n):
            e[i][0] = e0

        for i in range(self.n):
            for j in range(len(self.rings[i])):
                R[i][j + 1] = s[i][j] * EccGenerator - e[i][j] * self.rings[i][j].point
                e[i][j + 1] = self.H([M, R[i][j + 1], i, j])

        hashin = []
        for i in range(self.n):
            m_i = len(self.rings[i])
            hashin.append(R[i][m_i])
        calculated_e0 = self.H(hashin)

        return calculated_e0 == e0
```

LWW04

This paper introduces linkability to ring signature. Linkability is defined as two signatures with the same public key list L are linked if they are generated using the same private key. The algorithm is quite straightforward. Let $L = P_1, ..., P_n, where P_n = x_n G$. The signing private key is x_π.

The signing procedure is described below:

1. $H = H(L) * G$ and $\tilde{y} = x_\pi H$
2. Pick u randomely and compute $c_{\pi+1} = H(L, \tilde{y}, m, uG, uH)$
3. For $i = \pi + 1, ..., n, 1, ...\pi - 1$, select s_i randomly and compute $c_{i+1} = H(L, \tilde{y}, m, s_i G + c_i P_i, s_i H + c_i \tilde{y})$
4. Compute $s_\pi = u - x_\pi c_\pi$
5. Return signature $\sigma_L(m) = (c_1, s_1, ..., s_n, \tilde{y})$

> $\tilde{y}$ is the key image, if $\tilde{y}$ is the same for two signatures, these signatures are produced by the same private key.

```python
class LSAG:

    def __init__(self, k):
        self.k = k
        self.n = len(k)
        self.vk_serialize()
        self.H2()

    def vk_serialize(self):
        self.L = b''
        for key in self.k:
            if not isinstance(key, EccKey):
                raise TypeError('Only ECC key is allowed')
            self.L += key.point.sec()

    def H2(self):
        hashed_L = hashlib.sha1(self.L)
        self.h = (int(hashed_L.hexdigest(), 16) % EccOrder) * EccGenerator
    ...
    def sign(self, m, z):
        c = [None] * self.n
        s = [None] * self.n
        y = self.k[z].secret * self.h
        u = random.randint(0, EccOrder)

        c[(z+1) % self.n] = self.H([
            self.L,
            y,
            m,
            u * EccGenerator,
            u * self.h
        ])

        first_range = list(range(z + 1, self.n))
        second_range = list(range(z))
        whole_range = first_range + second_range
        for i in whole_range:
            s[i] = random.randint(0, EccOrder)
            c[(i + 1) % self.n] = self.H([
                self.L,
                y,
                m,
                s[i] * EccGenerator + c[i] * self.k[i].point,
                s[i] * self.h + c[i] * y,
            ])
        s[z] = (u - self.k[z].secret*c[z]) % EccOrder

        return [c[0]] + s + [y]
```

Verification procedure:

1. For $1 \leq i \leq n$, $c_{i+1} = H(L, \tilde{y}, m, s_i G + c_i P_i, s_i H + c_i \tilde{y})$
2. Accept if $c_1 = H(L, \tilde{y}, m, s_n G + c_n P_n, s_n H + c_n \tilde{y})$. Reject otherwise

```
class LSAG:
    ...
    def verify(self, m, sig):
        c0 = sig[0]
        s = sig[1:-1]
        c = [None] * (self.n + 1)
        c[0] = c0
        y = sig[-1]

        for i in range(self.n):
            c[i+1] = self.H([
                self.L,
                y,
                m,
                s[i] * EccGenerator + c[i] * self.k[i].point,
                s[i] * self.h + c[i] * y,
            ])
        return c0 == c[self.n]
```

Bac15

This paper modifies **LSAG** algorithm to provide a more efficient linkable ring signature while adding the feature that no secret key is allowed to sign twice on the blockchain. Noted the key image in **LSAG**, the private key and the whole public key set together generate the key image. It provides the linkability only if the public key group remain the same. The modified version does not have this requirement. All the signature signed by the same secret key can be linked, ignoring the ring members selected. This is an important feature for applying ring signature to cryptocurrency to prevent double-spending problem. The algorithm is described below.

n members on the ring: $P_i, i = 0, 1, ..., n-1$
secret index j such that $xG = P_j$
key image: $I = xH(P_j)G$
message to be signed: m

The signing procedure:

1. randomly select α and $s_i, i \neq j, 1 \leq i \leq n$
2. Compute:
   $L_j = \alpha G$
   $R_j = \alpha H(P_j)G$
   $c_{j+1} = H(m, L_j, R_j)$
3. For $i = j+1, ..., n, 1, ..., j-1$, compute
   $L_i = s_i G + c_i P_i$
   $R_i = s_i H(P_i)G + c_i I$
   $c_{i+1} = H(m, L_i, R_i)$
4. Let $s_j = \alpha - c_j x_j$
5. return signature $\sigma = (I, c_1, s_1, ..., s_n)$

```
class Bac_LSAG:

    def __init__(self, k):
        self.k = k
        self.n = len(k)
        self.vk_serialize()

    def vk_serialize(self):
        self.L = b''
        for key in self.k:
            if not isinstance(key, EccKey):
                raise TypeError('Only ECC key is allowed')
            self.L += key.point.sec()

    @staticmethod
    def H_p(point):
        hashed_p = hashlib.sha1(point.sec())
        return (int(hashed_p.hexdigest(), 16) % EccOrder) * EccGenerator
    ...
    def sign(self, m, z):
        I = self.k[z].secret * self.H_p(self.k[z].point)
        L = [None] * self.n
        R = [None] * self.n
        c = [None] * self.n
        s = [random.randint(0, EccOrder) if i !=z else None for i in range(self.n)]
        _alpha = random.randint(0, EccOrder)

        L[z] = _alpha * EccGenerator
        R[z] = _alpha * self.H_p(self.k[z].point)
        c[(z+1) % self.n] = self.H([m, L[z], R[z]])

        first_range = list(range(z + 1, self.n))
        second_range = list(range(z))
        whole_range = first_range + second_range

        for i in whole_range:
            L[i] = s[i] * EccGenerator + c[i] * self.k[i].point
            R[i] = s[i] * self.H_p(self.k[i].point) + c[i] * I
            c[(i+1) % self.n] = self.H([m, L[i], R[i]])

        s[z] = (_alpha - c[z] * self.k[z].secret) % EccOrder

        return [I] + [c[0]]+ s
```

Verification procedure:

1. For $1 \leq i \leq n$, compute
   $L_i = s_i G + c_i P_i$
   $R_i = s_i H(P_i) G + c_i I$
   $c_{i+1} = H(m, L_i, R_i)
2. Accept if $c_1 = c_{n+1}$. Reject otherwise

```
class Bac_LSAG:
    ...
    def verify(self, m, sig):
        I = sig[0]
        c0 = sig[1]
        s = sig[2:]
        c = [None] * (self.n + 1)
        L = [None] * self.n
        R = [None] * self.n

        c[0] = c0

        for i in range(self.n):
            L[i] = s[i] * EccGenerator + c[i] * self.k[i].point
            R[i] = s[i] * self.H_p(self.k[i].point) + c[i] *I
            c[i+1] = self.H([m, L[i], R[i]])

        return c[0] == c[self.n]
```

MLSAG

This paper introduces the ring signature scheme used in Monero. MLSAG is similar to Borromean ring signature. Instead of dealing with one ring, it deals with a list of key-vectors. Each key-vector contains exactly m keys and the ring contains n key vectors. The signature proves that the signer has the knowledge of the secret keys to the entire key vector. And the signature also provides the linkability property just as Bac's LSAG scheme. Each two signatures signed by the same private key are linked together, so that the latter one is discarded to prevent the double-spending problem in cryptocurrency. The algorithm is described below.

n key vectors, each have m keys: $\{P_i^j\}_{j=1,...,m}^{i=1,...,n}$
secret index: π
secret keys: for $j = 1, ..., m, x_j$
key images: for $j = 1, ..., m, I_j = x_j H(P_\pi^j)$

The signing procedure:

1. For $j = 1, ..., m, i = 1, ..., \tilde{\pi}, ...n$: randomly select s_i^j
2. For $j = 1, ..., m$ randomly select α_j
3. $L_\pi^j = \alpha_j G$
   $R_\pi^j = \alpha_j H(P_\pi^j)$ $c_{\pi+1} = H(m, L_\pi^1, R_\pi^1, ..., L_\pi^m, R_\pi^m)$
4. For $i = \pi + 1, ..., n, 1, ..., \pi - 1$, compute
   $L_i^j = s_i^j G + c_i P_i^j$
   $R_i^j = s_i^j H(P_i^j) + c_i I_j$
   $c_{i+1} = H(m, L_i^1, R_i^1, ..., L_i^m, R_i^m)$
5. For $j = 1, ..., m$, $s_\pi^j = \alpha_j - c_\pi x_j$
6. Return signature $\sigma = (I_1, ..., I_m, c_1, s_1^1, ..., s_1^m, ..., s_n^1, ..., s_n^m)$

```
class MLSAG:

    def __init__(self, k):
        self.k = k
        self.n = len(k)
        self.m = len(k[0])

    @staticmethod
    def H_p(point):
        hashed_p = hashlib.sha1(point.sec())
        return (int(hashed_p.hexdigest(), 16) % EccOrder) * EccGenerator
    ...
    def sign(self, m, z):
        I = [None] * self.m
        for j in range(self.m):
            I[j] = self.k[z][j].secret * self.H_p(self.k[z][j].point)
        s = [ [ None for j in range(self.m) ] for i in range(self.n) ]
        L = [ [ None for j in range(self.m) ] for i in range(self.n) ]
        R = [ [ None for j in range(self.m) ] for i in range(self.n) ]
        c = [None] * self.n
        _alpha = [random.randint(0, EccOrder) for j in range(self.m)]
        for i in range(self.n):
            for j in range(self.m):
                if i == z:
                    continue
                s[i][j] = random.randint(0, EccOrder)

        for j in range(self.m):
            L[z][j] = _alpha[j] * EccGenerator
            R[z][j] = _alpha[j] * self.H_p(self.k[z][j].point)

        hashin = [m]
        for j in range(self.m):
            hashin.append(L[z][j])
            hashin.append(R[z][j])
        c[(z+1) % self.n] = self.H(hashin)

        first_range = list(range(z + 1, self.n))
        second_range = list(range(z))
        whole_range = first_range + second_range

        for i in whole_range:
            hashin = [m]
            for j in range(self.m):
                L[i][j] = s[i][j] * EccGenerator + c[i] * self.k[i][j].point
                R[i][j] = s[i][j] * self.H_p(self.k[i][j].point) + c[i] * I[j]
                hashin.append(L[i][j])
                hashin.append(R[i][j])
            c[(i+1) % self.n] = self.H(hashin)

        for j in range(self.m):
            s[z][j] = (_alpha[j] - c[z] * self.k[z][j].secret) % EccOrder

        return I + [c[0]] + s
```

Verification procedure:

1. For $i = 1, ..., n$
   $L_i^j = s_i^j G + c_i P_i^j$
   $R_i^j = s_i^j H(P_i^j) + c_i I_j$
   $c_{i+1} = H(m, L_i^1, R_i^1, ..., L_i^m, R_i^m)$
2. Accept if $c_1 = c_{n+1}$. Reject otherwise

```
class MLSAG:
    ...
    def verify(self, m, sig):
        I = sig[0:self.m]
        c0 = sig[self.m]
        s = sig[self.m+1:]
        c = [None] * (self.n + 1)
        L = [ [ None for j in range(self.m) ] for i in range(self.n) ]
        R = [ [ None for j in range(self.m) ] for i in range(self.n) ]

        c[0] = c0

        for i in range(self.n):
            hashin = [m]
            for j in range(self.m):
                L[i][j] = s[i][j] * EccGenerator + c[i] * self.k[i][j].point
                R[i][j] = s[i][j] * self.H_p(self.k[i][j].point) + c[i] * I[j]
                hashin.append(L[i][j])
                hashin.append(R[i][j])
            c[i+1] = self.H(hashin)

        return c[0] == c[self.n]
```

Part III: Monero Implementation

This section develops a Monero-like cryptocurrency blockchain system. It implements all the three major technologies used in Monero for privacy protection, the stealth address, ring signature, and RingCT. This section can be divided into two parts. The first part discusses the three major components. The second part documents the technical design in the Monero-like cryptocurrency. It covers all the major components, including address, transaction, block, chain, wallet, node. All the code can be found in the monero subdirectory.

Technical Discussion

Monero is a very privacy-preserving cryptocurrency. It privacy-preserving nature is supported by its three main technologies: stealth address, ring signature, and RingCT. The stealth address ensures every transaction output is targeted to a different receiver address. Any address cannot be re-used to receive amount, so that the receivers of transaction are not traceable. The ring signature ensures the sender of any transaction is mixed into a group of other addresses, so that it is nearly impossible to analyze the behavior of any user. Finally, the RingCT hides the amount in the transaction. In this way, the transaction reveals basically nothing about the sender, receiver, amount.

Stealth Address

Each user in Monero has two sets of private/public keys, (k^v, K^v) and (k^s, K^s). So, the address of a user is the pair of public keys (K^v, K^s). And his/her private keys is (k^v, k^s). k^v is known as the view key since it can determine if the user owns an output. The spend key k^s allows the user to spend the outputs s/he owns, or figure out if an output has already been spent.

In every transaction, the sender generates a one-time address given the receiver's address(public key par). The one-time address generation procedure is as the following:

Alice wants to make a payment to Bob. Bob has private/public keys (k_B^v, k_B^s) and (K_B^v, K_B^s).

1. Alice generates a random number r, one-time address is calculated as
   $K^O = H_n(rK_B^v)G + K_B^s$
2. Alice make the payment to K^O and add the value rG to the transaction as the *transaction public key*.
3. Bob receives rG and K^O. He calculates.
   $k_B^v rG = rK_B^v$
   ${K'}_B^s = K^O - H_n(rK_B^v)G$
   if ${K'}_B^s = K_B^s$: Bob knows the output is addressed to him
4. One-time keys for the output
   $K^O = H_n(rK_B^v)G + K_B^s = (H_n(rK_B^v) + k_B^s)G$
   $k^O = H_n(rK_B^v) + k_B^s$

With the view key k^v, Bob can view which output belongs to him. With the spend key k^s, Bob can spend the output.

If the transaction has multiple output, the sender usually generates only one random number r The *transaction public key* rG is also published alongside other transaction data in the blockchain. By appending the transaction output index before hashing it, all one-time addresses are different even the same addresses is used twice. The one-time address for multi-output transaction is like this:
$K_t^O = H_n(rK_t^v, t)G + K_t^s = (H_n(rK_t^v, t) + k_t^s)G$
$k_t^O = H_n(rK_t^v, t) + k_t^s$

Users can also generate subaddresses from his/her main address. Payments sent to a subaddress can be viewed and spent using its main address's view and spend keys. The subaddress function is also implemented in the address module. But the support for subaddress is not fully tested with the other modules. So the subaddress generation procedure is not discussed here.

Ring Signature

The technical details about ring signature have been discussed thoroughly in the last section. Monero uses MLSAG scheme discussed earlier. One topic to be discussed here is how Monero handles the double-spending problem.

An MLSAG signature contains key images $\tilde{K_j}$ of private key $k_{\pi,j}$. The blockchain network needs to verify that each key images included in MLSAG signatures($\tilde{K_j^O} = k_{\pi,j}^O H_p(K_{\pi,j}^O)$) have not appeared before in other transactions. If the key image have appeared before, it can be assured as an double-spending attempt.

RingCT

The most important feature in RingCT is to hide the transaction amount from everyone except senders and receivers. To achieve this, **Pedersen commitments** is used, which is *additively homomorphic*. For the amount b, the commitment is defined as $C(y, b) = yG + bH$, where y is the blinding factor and H is another generator($H = \gamma G$). There is no way for attacker to figure out a without knowing the random mask y.

The amount commitment is computed like this:

$$y_t = H_n("commitment_mask", H_n(rK_B^v, t))$$
$$amount_t = b_t \oplus_8 H_n("amount", H_n(rK_B^v, t))$$

The receiver will be able to calculate the blinding facter y_t and the amount b_t using the transaction public key rG and his view key k_B^v. b_t and amount are restricted to 8 bytes.

Since the commitments are additive. It can proved the inputs equal outputs to observers if:

$$\sum_j C_{j,in} - \sum_t C_{t,out} = 0$$

The sender can create new commitments to the same amounts but using different blinding factors.

$$C_j^a = x_j G + a_j H \implies C'^a_j = x'_j G + a_j H$$

The difference between the two commitments:

$$C_j^a - C'^a_j = (x_j - x'_j)G$$

The sender can make a signature with the private key $(x_j - x'_j) = z_j$ and prove there is no H component to the sum. And C'^a_j is known as the *pseudo output commitment*.

To prove input amounts equal output amounts:

$$(\sum_j C'^a_j - \sum_t C_t^b) = 0$$

The blinding factors for pseudo and output commitments are selected such that

$$\sum_j x'_j - \sum_t y_t = 0$$

To achieve this, all blinding factors are random except for the m^{th} pseudo out commitment, so that

$$x'_m = \sum_t y_t - \sum_{j=1}^{m-1} x'_j$$

Typically transaction outputs are lower in total than transaction outputs. The difference is the transaction fee. Transaction fee amounts f are stored in clear text in the transaction data and credited to the miner on mining. The transaction fee is calculated as

$$f = \sum_{j=1}^{m} a_j - \sum_{t=0}^{p-1} b_t$$

And to prove the input amounts equal output amounts:

$$(\sum_j C'^a_j - \sum_t C^b_t) - fH = 0$$

To produce a ring signature for a transaction input, the sender selects m sets of size v, of additional unrelated one-time address and their commitments from the blockchain. To sign input j, the sender stirs a set of size v into a *ring* with his/her own j^{th} unspent one-time address:

$$\begin{gathered} R_j = \{\{K^O_{1,j}, (C_{1,j} - C'^a_{\pi,j})\} \\ \ldots \\ \{K^O_{\pi,j}, (C^a_{\pi,j} - C'^a_{\pi,j})\} \\ \ldots \\ \{K^O_{v+1,j}, (C_{v+1,j} - C'^a_{\pi,j})\}\} \end{gathered}$$

Since the sender knows the private key $k^O_{\pi,j}$ for $K^O_{\pi,j}$, and z_j for the commitment to zero $(C^a_{\pi,j} - C'^a_{\pi,j})$, s/he can provide a valid MLSAG signature as long as the key image never appeared in the blockchain network before. The message to be signed is the hash of the whole transaction data except for the MLSAG signatures.

Implementation Design

To create a Monero-like cryptocurrency, there are an amount of technical decisions to be made. The technical designs are explained and documented below. Each following parts represent a module. The source code can be found in the monero subdirectory. Each module comes with one or more test cases. To run the unittest, *change directory* to the **tests** directory and run

```
python -m unittest <test filename>
```

To get prepared for all the test, please ensure the following directories exist:

- ${repo_home}/monero/data/ring
- ${repo_home}/monero/secret

> The tests might fail if the directories above are missing

To run all the test cases, run

```
python -m unittest test_*
```

Some test cases may take some time to finish. It takes more than 15 minutes to finish all the test cases on a *Intel i7* machine.

The test command may differ from different platform. Please check with Python unittest module for the detail.

Address

The Address module implements the stealth address. It only depends on the ECC library from Part I. The technical details about the stealth address have been discussed before. And here shows some implementation details. The sub-address of the stealth address is also implemented, but it will not be fully explained here.

```python
class UserKeys:
    """
    Users have two sets of private/public keys
    (k_v, K_v) and (k_s, K_s)
    view key and spend key
    view key: determine if their address owns an output
    spend key: spend that output / check if spent
    """

    def __init__(self, secret1, secret2):
        # view key
        self.view = EccKeyPair(secret1)
        # spend key
        self.spend = EccKeyPair(secret2)
```

The user account is abstracted to a Python class called `UserKeys`. Each `UserKeys` contains two `EccKeyPair`: the view key pair and the spend key pair.

```python
class UserKeys:
    ...
    @classmethod
    def generate(cls):
        # generate a random user address
        secret1 = random.randint(1, EccOrder)
        secret2 = random.randint(1, EccOrder)
        return cls(secret1, secret2)
```

To generate a new random user account, there is a `classmethod` for `UserKeys`.

```python
class UserKeys:
    def getPubKey(self):
        # pub key getter
        return (self.view.point, self.spend.point)

    def serialize(self):
        (K_v, K_s) = self.getPubKey()
        return K_v.sec().hex() + K_s.sec().hex()
```

Each user address can be represented by two public keys: (K^v, K^s). And the address can be represented by combining the SEC format of the two public keys together. The compressed SEC format is by default in all the following modules. Reminded that SEC of one `S256Point` occupies 33 bytes. So the user address occupies 66 bytes in total.

```python
class UserKeys:
    ...
    @classmethod
    def generateOneTimeAddr(cls, pubKeyPair, r=None):
        # generate one time address
        # taking pubkey pair as input
        # output: tx pubkey and one-time address
        (K_v, K_s) = pubKeyPair

        # generate random number
        if r is None:
            r = random.randint(1, EccOrder)
        # K = H(r * K_v) * G + K_s
        K = cls.H_n(r * K_v) * EccGenerator + K_s

        # (tx pubkey, one-time address)
        if sub:
            return (r * K_s, K)
        else:
            return (r * EccGenerator, K)
```

The `UserKeys` class also contains a `classmethod` to generate a one-time address for the user. Since usually the sender generates the one-time address, so only the public key pair is taken as input. Since the module also supports sub-address and multi-output transaction. So the actual implementation may be a bit more complex. But the detail will not be discussed here.

```python
class UserKeys:
    ...
    def ownsOneTimeAddr(self, oneTimeAddr):
        # txPubKey = r * G
        (txPubKey, oneTimePubKey) = oneTimeAddr
        # k_v * r * G = r * K_v
        hashin = self.view.secret * txPubKey
        # K_s' = K - H(r*K_v)G
        K_s = oneTimePubKey - self.H_n(hashin) * EccGenerator
        # check if K_s' == K_s or K_s in subsSpendKeys
        return K_s == self.spend.point or K_s in subSpendKeys
```

The user account needs to be able to justify if it owns specific one-time address. So there is a method named `ownsOneTimeAddr` to achieve this. The name `oneTimeAddr` is actually a tuple of (txPubKey, oneTimeAddr). It may be a bit confusing. But since rG is also needed to tell if the user owns the one-time address, it may be better to take two inputs for the method. It is the same case for `oneTimeAddr` in the `generateOneTimeSecret method right below.

```
class UserKeys:
    ...
    def generateOneTimeSecret(self, oneTimeAddr):
        # generate one-time secret from one-time addr
        if not self.ownsOneTimeAddr(oneTimeAddr):
            raise RuntimeError("OneTimeAddress not owned by this addr")
        # txPubKey = r * G
        (txPubKey, oneTimePubKey) = oneTimeAddr
        # k_v * r * G = r * K_v
        hashin = self.view.secret * txPubKey
        # K_s' = K - H(r*K_v, t)G
        K_s = oneTimePubKey - self.H_n(hashin) * EccGenerator
        if K_s == self.spend.point:
            # k = H(r * K_v) + k_s
            return (self.H_n(hashin) + self.spend.secret) % EccOrder
```

The user should be able to generate the one-time secret corresponding to the one-time address to successfully sign the output when spending it. And here is the method to accomplish this. It requires the spend key k^s.

Transaction

The transaction module implements the RingCT. It can be divided into 4 parts: commitments, transaction input, transaction output, and the whole trasaction.

Commit

The class `Commit` inherits the class `EccPoint`, since by nature a commitment is actually an ECC point on the curve.

Recall that $C(y, b) = yG + bH$

```
H = 8 * EccKey(H_n([EccGenerator])).point
```

The choice of the second generator of commitment H follows the design of the Monero: $H = 8 \times H_p(G)$

```
class Commit(EccPoint):
    def __init__(self, y, b):
        p = y * EccGenerator + b * H
        return super().__init__(p.x, p.y)

    @classmethod
    def generate(cls, K_v, b, r, t=0):
        # b: actual amount
        # r: random number for txPubKey
        y = H_n(["commitment_mask", H_n([r * K_v, t])])
        amount = first_eight_bytes(H_n(["amount", H_n([r * K_v, t])])) ^
first_eight_bytes(b)

        return (cls(y, b), amount)
```

Since commitments are generated in transaction out. It takes the public view key K^v of the receiver, the random number r generated for the transaction public key and the output index as input, and generate a commitment from these. The commitment itself is a `EccPoint` by nature, so it also uses the SEC format for serialization.

```
class Commit(EccPoint):
    ...
    @staticmethod
    def resolve(txPubKey, amount, k_v, t=0):
        return first_eight_bytes(amount) ^ \
            first_eight_bytes(H_n(["amount", H_n([k_v * txPubKey, t])]))
```

On receiving the commitment, the receiver should be able to decrypt the actual transaction amount from it. It only requires the private view key k^v from the receiver. The staticmethod `resolve` implements this.

```
class Commit(EccPoint):
    ...
    def newCommit(self, txPubKey, amount, k_v, t, new_y=None):
        b = Commit.resolve(txPubKey, amount, k_v, t)
        y = H_n(["commitment_mask", H_n([k_v * txPubKey, t])])
        if new_y is None:
            new_y = random.randint(1, EccOrder)

        return self.__class__(new_y, b)
```

When the receiver would like to spend the previous transaction output, s/he needs to construct a new commitment. It requires a new mask and the old actual amount. The new mask is usually generated at random, except for the last one in order to prove the knowledge of zero.

TxOut

The class `TxOut` represents each transaction output.

```python
class TxOut:
    """
    Transaction output
    4 field for each output:
    1. oneTimeAddr
    2. txPubKey (r * G)
    3. amount (used to calculate actual amount b)
    4. commit (amount hidden by commit mask)
    """

    def __init__(self, oneTimeAddr, txPubKey, amount, commit):
        self.oneTimeAddr = oneTimeAddr
        self.txPubKey = txPubKey
        self.amount = amount
        self.commit = commit
```

It contains the one-time address of the receiver, transaction public key, encrypted amount generated during calculating the commitment and the commitment itself.

```python
class TxOut:
    ...
    @classmethod
    def generate(cls, b, pubKeyPair, t=0, r=None):
        if r is None:
            r = random.randint(1, EccOrder)
        if t == 0:
            (txPubKey, oneTimeAddr) = UserKeys.generateOneTimeAddr(
                pubKeyPair=pubKeyPair,
                r = r,
            )
        else:
            (txPubKey, oneTimeAddr) = UserKeys.generateOneTimeAddrMultiOut(
                pubKeyPair=pubKeyPair,
                t = t,
                r = r,
            )
        (commit, amount) = Commit.generate(
            K_v = pubKeyPair[0],
            b = b,
            r = r,
            t = t,
        )
        return cls(oneTimeAddr, txPubKey, amount, commit)
```

The `generate` method combines the `UserKeys` module and the previous `Commit` module. It is quite straightforward. Given the receiver's address(public key pair) and the actual amount for payment b, it generates a `TxOut` object. It utilizes the class method `UserKeys.generateOneTimeAddrMultiOut`, which is not talked about. Basically it does the same thing with `UserKeys.generateOneTimeAddr`, but for the transaction with multiple outputs. Here, it is assumed that, regardless how many outputs a transaction has, it is using the `UserKeys.generateOneTimeAddr` method for its first output and `UserKeys.generateOneTimeAddrMultiOut` method for the rest of transaction outputs

```
class TxOut:
    ...
    @classmethod
    def parse(cls, s):
        # Takes a byte stream and parses the tx_output
        # and return a TxOut Object
        oneTimeAddress = EccPoint.parse(s.read(33))
        txPubKey = EccPoint.parse(s.read(33))
        commit = EccPoint.parse(s.read(33))
        amount = little_endian_to_int(s.read(8))

        return cls(oneTimeAddress, txPubKey, amount, commit)

    def serialize(self):
        # returns the byte serialization of the transaction output
        # the result will be static 107 bytes
        # 33 + 33 + 33 + 8
        result = self.oneTimeAddr.sec()
        result += self.txPubKey.sec()
        result += self.commit.sec()
        result += int_to_little_endian(self.amount, 8)
        return result
```

The transaction output is easy for serialization and parsing, since its four components are all of fixed sizes. `TxOut.oneTimeAddress`, `TxOut.txPubKey` and `TxOut.commit` are all `EccPoint`. They can all be represented in SEC format, which occupies 33 bytes. `TxOut.amount` has a fixed length of 8 bytes. So a `TxOut` object always occupies 107 bytes in total.

TxIn

The transaction input is represented by Python class `TxIn`.

```
class TxIn:
    """
    Transaction Input

    1. ring (list of one-time address from previous output)
    2. pseudo output commitment (sum of which equals to sum of output commitments)
    3. key image (part of ring sig)
    4. signature (ring signature)
    """

    def __init__(self, ring, pseudoOut, keyImage, sig):
        # ring has the size 6 * 2
        self.ring = ring
        self.pseudoOut = pseudoOut
        self.keyImage = keyImage
        # (I + c0 + s)
        # I has size 2
        # c0 is a hash256 int
        # s has size 6 * 2
        self.sig = sig
```

Each `TxIn` has four components. `TxIn.ring` represents the ring members selected for the ring signature. The ring size is fixed to **6**, which means that apart from the real previous transaction output, 5 mixins are randomly selected to form the ring. `TxIn.pseudoOut` is the pseudo output commitment randomly generated or calculated to prove the knowledge of zero. `TxIn.keyImage` is the key image of the ring signature to prevent double-spending. `TxIn.sig` is the actual ring signature.

```
class TxIn:
    ...
    @classmethod
    def parse_unsigned(cls, s):
        ring = [[None for i in range(2)] for j in range(RING_SIZE)]
        for i in range(RING_SIZE):
            for j in range(2):
                ring[i][j] = EccPoint.parse(s.read(33))
        pseudoOut = EccPoint.parse(s.read(33))
        keyImage = EccPoint.parse(s.read(33))

        return cls(ring=ring, pseudoOut=pseudoOut, keyImage=keyImage, sig=None)


    @classmethod
    def parse(cls, stream):
        unsigned = cls.parse_unsigned(BytesIO(stream.read(462)))
        I = [None] * 2
        for i in range(2):
            I[i] = EccPoint.parse(stream.read(33))
        c0 = little_endian_to_int(stream.read(32))
        s = [[None for i in range(2)] for j in range(RING_SIZE)]
        for i in range(RING_SIZE):
            for j in range(2):
                s[i][j] = little_endian_to_int(stream.read(32))
        unsigned.sig = I + [c0] + s

        return unsigned


    def serialize_unsigned(self):
        # returns serialization without sig
        # includes only ring, pseuodoOut, keyImage
        # ring has size 6 * 2; each is a one-time address(EccPoint)
        #   size: 6 * 2 * 33bytes
        # pseudoOut is a EccPoint
        #   size: 33 bytes
        # keyImage is a EccPoint
        #   size: 33 bytes
        # In total: 33 bytes * 14 = 462 bytes
        result = b''
        for i in range(RING_SIZE):
            for j in range(2):
                result += self.ring[i][j].sec()
        result += self.pseudoOut.sec()
        result += self.keyImage.sec()

        return result

    def serialize(self):
        # serialized unsigned: 462 bytes
        # sig:
        #   (I + c0 + s)
        #   I has size 2 * 33 bytes
        #   c0 is a hash256 int: 256 / 8 = 32 bytes
        #   s has size 6 * 2, each has 32 bytes
        #   sig total = 2 * 33 + 32 + 12 * 32 = 482 bytes
        # Total: 462 bytes + 482 bytes = 944 bytes
        result = self.serialize_unsigned()
        I = self.sig[0:2]
        c0 = self.sig[2]
        s = self.sig[3:]
        for i in range(2):
            result += I[i].sec()
        result += int_to_little_endian(c0, 32)
        for i in range(RING_SIZE):
            for j in range(2):
                result += int_to_little_endian(s[i][j], 32)

        return result
```

The serialization for `TxIn` object is a bit complex. Since transaction iput needs to be generated first without the signature, and then signed with the whole transaction data as input. So both unsigned and signed version of `TxIn` object needs a pair of serialization/parsing method. For unsigned `TxIn`, the ring is composed of 12 EccPoint, which occupies $(6 \times 2 \times 33)$ bytes. `TxIn.pseudoOut` and `TxIn.keyImage` are actually `EccPoint`s, each occupying 33 bytes. The unsigned `TxIn` occupies 462 bytes in total. The signed `TxIn` has an additional signature. The signature is composed of $(I + c_0 + s)$, I is actually a pair of `EccPoint`s, occupying (2×33) bytes. c_0 is actually a hash256 integer, occupying 32 bytes. So are elements in s. There are 12 hash256 integer in s, occupying (12 \times 32) bytes in total. So the signed `TxIn` occupies 944 bytes in total.

```
class TxIn:
    ...
    @classmethod
    def generateUnsigned(cls, oneTimeAddr, user, t=0, pseudoMask=None):
        prevOut = searchOneTimeAddr(oneTimeAddr)
        if not user.ownsOneTimeAddr((prevOut.txPubKey, prevOut.oneTimeAddr, t)):
            raise RuntimeError("user does NOT own prevOut")

        # calculate pseudo out commit
        if pseudoMask is None:
            pseudoMask = random.randint(1, EccOrder)
        if prevOut.commit == Commit(1, prevOut.amount):
            # miner tx output
            b = prevOut.amount
            t = 0
        else:
            # normal tx output
            b = Commit.resolve(
                txPubKey = prevOut.txPubKey,
                amount = prevOut.amount,
                k_v = user.view.secret,
                t = t,
            )
        pseudoOut = pseudoMask * EccGenerator + b * H

        # construct ring
        ring = [None] * RING_SIZE
        _pi = random.randint(0, RING_SIZE - 1)
        ring[_pi] = [oneTimeAddr, prevOut.commit - pseudoOut]
        for i in range(len(ring)):
            while ring[i] is None:
                randomOneTimeAddr = selectOneTimeAddr()
                if randomOneTimeAddr not in [_[0] for _ in ring if _ is not None]:
                    randomOut = searchOneTimeAddr(randomOneTimeAddr)
                    ring[i] = [randomOneTimeAddr, randomOut.commit - pseudoOut]

        # skip sig since the input is unsigned
        sig = None

        # add key Image
        keyImage = user.generateOneTimeSecret((prevOut.txPubKey, oneTimeAddr, t)) * \
            MLSAG.H_p(oneTimeAddr)

        return cls(ring, pseudoOut, keyImage, sig)
```

The generation of unsigned `TxIn` is straightforward. 3 steps are needed: calculate the pseudo commitment, form the ring, add the key image. All the methods were introduced before, so no more discussion here.

```
class TxIn:
    ...
    def sign(self, oneTimeAddr, user, m, pseudoMask, t=0):
        if self.sig is not None:
            raise RuntimeError("cannot re-sign TxIn")
        prevOut = searchOneTimeAddr(oneTimeAddr)
        if not user.ownsOneTimeAddr((prevOut.txPubKey, prevOut.oneTimeAddr, t)):
            raise RuntimeError("user does NOT own prevOut")
        if prevOut.commit == Commit(1, prevOut.amount):
            # miner tx output
            prevMask = 1
            t = 0
        else:
            prevMask = H_n(["commitment_mask", H_n([user.view.secret *
prevOut.txPubKey, t])])
        secrets = [user.generateOneTimeSecret((prevOut.txPubKey, oneTimeAddr, t)),
(prevMask - pseudoMask) % EccOrder]

        _pi = [_[0] for _ in self.ring].index(oneTimeAddr)
        self.sig = MLSAG(self.ring).sign(m, _pi, secrets)

    def verify(self, m):
        if self.sig is None:
            raise RuntimeError("cannot verify unsigned TxIn")
        return MLSAG(self.ring).verify(m, self.sig)
```

After hashing the whole transaction data, the sender can produce signature for each of the inputs. Signing and verification procedure rely on the **MLSAG** implementation in Part II. Noted that miner transaction (coinbase transaction in Bitcoin) has a different signing secret calculation method, which will be discussed later in the `Tx` class.

Tx

The transaction object is represented in `Tx` class:

```
class Tx:
    """
    Transaction class:
    4 fields
    1. type: 0 (miner transaction), 1 (normal transaction)
    2. tx_ins: a list of TxIn object
    3. tx_outs: a list of TxOut object
    4. fee: clear text
    """
    def __init__(self, type, tx_ins, tx_outs, fee):
        if type == 0 and len(tx_ins) != 0:
            raise ValueError("miner transaction cannot have tx_ins")
        self.type = type
        self.tx_ins = tx_ins
        self.tx_outs = tx_outs
        self.fee = fee

    @classmethod
    def generateMiner(cls, pubKeyPair, fee):
        reward = MINER_REWARD
        if reward < 0 or reward > 0xffffffffffffffff:
            raise ValueError("Invalid miner reward value")
        if fee < 0 or fee > 0xffffffffffffffff:
            raise ValueError("Invalid fee reward value")
        amount = reward + fee
        if amount < 0 or amount > 0xffffffffffffffff:
            raise ValueError("Invalid amount value")

        tx_out = TxOut.generate(
            pubKeyPair=pubKeyPair,
            b = 0,
        )
        tx_out.amount = amount
        tx_out.commit = Commit(1, amount)
        tx_outs = [tx_out]
        tx_ins = []
        type = 0
        fee = 0

        return cls(type, tx_ins, tx_outs, fee)
```

The transaction objects have two types, the miner transaction and the normal transaction. The miner transaction is like the coinbase transaction in Bitcoin, it rewards the miners for successfully mining a new block. To simplify the implementation, it is assumed that in each miner transaction, only **1** one-time address can be specified as the transaction output. And each successful mining rewards the miner a reward of **100**. And each transaction also includes a fee to encourage the miner to include the transaction into the new block. The miner transaction always has 0 fee. Apart from these the transaction has two lists: list of `TxIn` objects and list of `TxOut` objects.

```
class Tx:
    ...
    def serialize(self):
        result = b''
        # 1 byte for type (0 or 1)
        result += int_to_little_endian(self.type, 1)
        # 8 bytes for tx fee
        result += int_to_little_endian(self.fee, 8)
        # 1 byte for input len
        in_len = len(self.tx_ins)
        if in_len < 0 or in_len > 0xff:
            raise ValueError("Invalid input len")
        result += int_to_little_endian(in_len, 1)
        for i in range(in_len):
            # each input has len 944 bytes
            result += self.tx_ins[i].serialize()
        out_len = len(self.tx_outs)
        if out_len < 0 or out_len > 0xff:
            raise ValueError("Invalid output len")
        result += int_to_little_endian(out_len, 1)
        for i in range(out_len):
            # each output has len 107 bytes
            result += self.tx_outs[i].serialize()

        total_len = 1 + 8 + 1 + in_len * 944 + 1 + out_len * 107
        if len(result) != total_len:
            raise ValueError("error when serializing tx")

        return result

    @classmethod
    def parse(cls, s):
        # 1 byte for type (0 or 1)
        type = little_endian_to_int(s.read(1))
        if type != 0 and type != 1:
            raise ValueError("Tx type can only be 0 or 1")
        # 8 bytes for tx fee
        fee = little_endian_to_int(s.read(8))
        in_len = little_endian_to_int(s.read(1))
        tx_ins = []
        if type == 0 and in_len != 0:
            raise ValueError("Miner Tx cannot have non-zero inputs")
        for i in range(in_len):
            tx_ins.append(TxIn.parse(BytesIO(s.read(944))))

        out_len = little_endian_to_int(s.read(1))
        tx_outs = []
        if type == 0 and out_len != 1:
            raise ValueError("Miner Tx can only have 1 output")
        for i in range(out_len):
            tx_outs.append(TxOut.parse(BytesIO(s.read(107))))

        return cls(
            type = type,
            tx_ins = tx_ins,
            tx_outs = tx_outs,
            fee = fee,
        )
```

The serialization of `Tx` objects assumes the input length and output length is within 1 bytes, which means a transaction can have at most 255 inputs and 255 outputs. The rest of serialization just depends on the serialization of `TxIn` and `TxOut`. Each `TxIn` occupies 944 bytes and each `TxOut` occupies 107 bytes. The total length of a transaction is variable.

```
class Tx:
    @classmethod
    def generate(cls, user, oneTimeAddresses, outs):
        ...
    def verify(self):
```

Apart from the methods above, the `Tx` class also has two useful methods. The method `generate` generates a normal transaction for a user. It receives a list of receiver and amounts and a list of one-time addresses owned by the user. The transaction transfers all the amounts in inputs to the outputs and transfers the change back to the user. It then produces the ring signature and return the `Tx` object. The implementation is tedious in logic, but all the critical components have been explained above. So the code is not shown here. Another method is the `verify` method. It verifies the transaction by checking ring signatures, commitments and key images. It does a thorough verification on the validity of both normal and miner transaction.

Block

After the miner collecting the transactions, s/he tries to build a block from them. Monerno uses the same POW as the Bitcoin. In the implementation, the target of mining is deliberately set to be very easy. As long as the block hash starts with '0000', the block is treated as valid. Each block contains a miner transaction, through which miners collect the rewards and fees from normal transactions.

```python
class Block:

    def __init__(self, prev, txs=[], miner=None,
timestamp=int(datetime.timestamp(datetime.now())), nonce=None):
        # previous block hash (should start with '0000')
        # hash256 (32 bytes)
        # hexdigest, as string
        self.prev_block = prev
        # 4 bytes int
        self.timestamp = timestamp
        # 4 bytes int
        self.nonce = nonce
        # miner tx (118 bytes)
        self.miner = miner
        # list of tx object
        self.txs = txs
```

The components in `Block` are all easy to understand. `Block.prev_block` is what links the blocks together to form a Blockchain. `Block.timestamp` records the time the block is blocked. `Block.nonce` is a 4-bytes integer, which is tuned to achieve the mining target. `Block.miner` is the miner transaction, which takes no input and exactly one output to reward the miner. `Block.txs` is a list of `Tx` objects.

```python
class Block:
    ...
    @classmethod
    def parse(cls, s):
        # prev_block as bytes (32 bytes)
        # convert it to int
        prev_block = little_endian_to_int(s.read(32))
        prev_block = format(prev_block, 'x')
        prev_block = '0' * (64 - len(prev_block)) + prev_block

        timestamp = little_endian_to_int(s.read(4))
        nonce = little_endian_to_int(s.read(4))
        miner = Tx.parse(BytesIO(s.read(118)))

        txs = []
        len_tx = read_varint(s)
        for i in range(len_tx):
            tx_len = read_varint(s)
            tx = Tx.parse(BytesIO(s.read(tx_len)))
            txs.append(tx)

        return cls(prev=prev_block, timestamp=timestamp, nonce=nonce, miner=miner,
txs=txs)


    def serialize(self):
        result = b''
        result += int_to_little_endian(int(self.prev_block, 16), 32)
        result += int_to_little_endian(self.timestamp, 4)
        result += int_to_little_endian(self.nonce, 4)
        result += Tx.serialize(self.miner)
        len_tx = len(self.txs)
        result += encode_varint(len_tx)
        for i in range(len_tx):
            tx_serialized = self.txs[i].serialize()
            tx_len = len(tx_serialized)
            result += encode_varint(tx_len)
            result += tx_serialized
        return result
```

The serialization of `Block` objects make use of the variable integer introduced in Part I. The variable integer is used to record the transaction length. Apart from transactions, all the other components in the `Block` object are of fixed length. `Block.prev_block` is a hash256 integer, occupying 32 bytes. `Block.timestamp` and `Block.nonce` both occupy 4 bytes. Even the miner transaction is of fixed length. Since only one output is included in the miner transaction.

```
class Block:
    ...
    def hash(self):
        return hashlib.sha256(self.serialize()).hexdigest()

    def pow(self):
        for n in range(0, 0xffffffff + 1):
            self.nonce = n
            if self.hash().startswith(TARGET):
                break
        else:
            raise RuntimeError("Failed to mine the block")
```

The block hash is calculated by applying sha256 hash to the serialized bytes array. And the mining method is represented by the `Block.pow` method.

```
class Block:
    ...
    def verify(self):
        if not self.prev_block.startswith(TARGET):
            return False
        if not self.hash().startswith(TARGET):
            return False
        if not self.miner.verify():
            return False
        minerAmount = self.miner.tx_outs[0].amount
        for tx in self.txs:
            if not tx.verify():
                return False
            minerAmount -= tx.fee
        if minerAmount != MINER_REWARD:
            return False
        images = self.getKeyImages()
        if len(images) != len(set(images)):
            return False
        return True
```

After the block is mined, it needs to be verified and will be verified by all the other nodes accepting it. It checks the previous block hash, the miner transaction, all the normal transactions and the key images.

Chain

The Blockchain is represented by the Python `Chain` class:

```
class Chain:

    def __init__(self, name, blocks=[], txs=[]):
        self.name = name
        self.blocks = blocks
        self.txs = txs
```

In the following implementation, the name for the blockchain is simply called **ring**. The blockchain contains two types of data: `Chain.blocks` is a list of blocks and `Chain.txs` is a list of transactions to be added to future blocks by miners.

```
class Chain:
    ...
    def serializeBlocks(self):
        ...
    def serializeTxs(self):
        ...
    @classmethod
    def parseBlocks(cls, s):
        ...
    @classmethod
    def parseTxs(cls, s):
        ...
    def getBlockDataFile(self):
        return os.path.join(currentdir, 'data', self.name, 'chain.dat')
    def getTxDataFile(self):
        return os.path.join(currentdir, 'data', self.name, 'tx.dat')
    def dumpBlockData(self):
        with open(self.getBlockDataFile(), 'wb+') as f:
            f.write(self.serializeBlocks())
    def dumpTxData(self):
        with open(self.getTxDataFile(), 'wb+') as f:
            f.write(self.serializeTxs())
    def loadBlockData(self):
        with open(self.getBlockDataFile(), 'rb') as f:
            self.blocks = self.parseBlocks(BytesIO(f.read()))
    def loadTxData(self):
        with open(self.getTxDataFile(), 'rb') as f:
            self.txs = self.parseTxs(BytesIO(f.read()))
```

A series of supporting methods listed above make it possible to store the block data and transaction data from transient memory to permanent disk. In this implementation, the chain name is 'ring'. The block data is stored in the sub-directory data/ring/chain.dat and the transaction data is stored in data/ring/tx.dat. The two files are binary files and not readable.

```python
class Chain:
    ...
    def mine(self, pubKeyPair):
        if not self.verifyBlocks():
            raise RuntimeError("Cannot mine on invalid chain")
        txs = []
        # at most 5 tx per block
        while len(txs) < 5 and len(self.txs) > 0:
            to_be_added = self.txs.pop()
            if not to_be_added.verify():
                raise RuntimeError("Cannot mine with invalid tx")
            txs.append(to_be_added)

        # mined block
        block = Block(prev=self.blocks[-1].hash(), txs = txs)
        block.createMiner(pubKeyPair=pubKeyPair)
        block.pow()
        if not block.verify():
            raise RuntimeError("Mined block does not pass verification")

        # double spending check
        images = self.getKeyImages()
        for i in block.getKeyImages():
            if i in images:
                raise RuntimeError("Discovered double spending")

        self.blocks.append(block)

        self.dumpBlockData()
        self.dumpTxData()
```

The `Chain.mine` method is like an interface provided to the miner. It looks into the transactoin to be mined data and construct the block from these transactions. At most 5 transactions (excluding miner transaction) can be added to one block. And after the block is mined, it dumps the updated chain and transaction data to the disk.

```python
class Chain:
    ...
    def replace(self, blocks):
        longer = self.__class__(name = self.name, blocks = blocks)
        if not longer.verifyBlocks():
            raise RuntimeError("Invalid chain received")
        if (len(blocks) > len(self.blocks)) or (not self.verifyBlocks()):
            self.blocks = blocks

            # cleanup tx
            images = self.getKeyImages()
            for tx in self.txs:
                if not tx.verify():
                    self.txs.remove(tx)
                for i in tx.getKeyImages():
                    if i in images:
                        self.txs.remove(tx)
                        break
            self.dumpBlockData()
            self.dumpTxData()
```

The `Chain.replace` method provides an interface for peer node to update the blockchain if the peer has a longer chain. After the longer chain is updated, it searches the transactions data and remove all the transactions already included in the new chain.

```
class Chain:
    ...
    def add_tx(self, tx):
        if not tx.verify():
            raise RuntimeError("Invalid tx to be added")
        images = self.getKeyImages()
        for i in tx.getKeyImages():
            if i in images:
                raise RuntimeError("Discovered double spending")
        self.txs.append(tx)
        self.dumpTxData()
```

The `Chain.add_tx` method provides an interface for user to publish new transactions to transaction to-be-mined data. It verifies if the inputs of the newly added transaction are not double spended and adds the new transaction if it passes the verification.

```
class Chain:
    ...
    def replace_tx(self, txs):
        images = self.getKeyImages()
        for tx in txs:
            try:
                if not tx.verify():
                    raise RuntimeError("Invalid tx to be added")
                for i in tx.getKeyImages():
                    if i in images:
                        raise RuntimeError("Discovered double spending")
                if tx not in self.txs:
                    self.txs.append(tx)
            except:
                continue
        self.dumpTxData()
```

The `Chain.replace_tx` method provides an interface for peer node to update the transaction to-be-mined data. It receives a list of transactions and checks the possible transactions that can be added its own transaction data.

Wallet

The `Wallet` Python class represents a single account on the Blockchain network:

```python
SECRET_PATH = os.path.join(currentdir, 'secret')
VIEW_SECRET_PATH = os.path.join(SECRET_PATH, 'view')
SPEND_SECRET_PATH = os.path.join(SECRET_PATH, 'spend')

class Wallet:
    def __init__(self, key):
        self.key = key

    @classmethod
    def me(cls):
        # check if user secret already exists
        if os.path.exists(VIEW_SECRET_PATH) and \
            os.path.exists(SPEND_SECRET_PATH):
                with open(VIEW_SECRET_PATH, 'r') as f:
                    k_v = f.read()
                k_v = int(k_v, 16)
                with open(SPEND_SECRET_PATH, 'r') as f:
                    k_s = f.read()
                k_s = int(k_s, 16)
                return cls(UserKeys(k_v, k_s))
        else:
        # create secret if not exist
            os.makedirs(os.path.dirname(VIEW_SECRET_PATH), exist_ok=True)
            key = UserKeys.generate()
            with open(VIEW_SECRET_PATH, 'w+') as f:
                f.write(format(key.view.secret, 'x'))
            with open(SPEND_SECRET_PATH, 'w+') as f:
                f.write(format(key.spend.secret, 'x'))
            return cls(key)
```

The secrets for the account is stored in sub-directory secret/view and secret/spend. The `Wallet.me` method reads the secret from the secret files or generates a new random account if secret files do not exist.

```python
class Wallet:
    ...
    def scan_chain(self, chain):
        # return list of unspent
        # (oneTimeAddr, amount, keyImage)
    def amount(self, chain, unspent=None):
        if unspent is None:
            unspent = self.scan_chain(chain)
        return sum([u[1] for u in unspent])
```

The `Wallet.scan_chain` method scans the chain to get all the unspent transaction outpus owned by the wallet. Then the `Wallet.amount` method calculates the balance in the wallet.

```
class Wallet:
    ...
    def send(self, addr, chain, amount, fee):
        unspent = self.scan_chain(chain)
        balance = self.amount(chain, unspent=unspent)
        if amount < 0 or amount > 0xffffffffffffffff:
            raise ValueError("Invalid tx amount")
        if balance < amount + fee:
            raise ValueError("Balance not enough")
        pubKey = self.parseAddress(addr)
        ins = []
        out_amount = 0
        for i in unspent:
            ins.append(i[0])
            out_amount += i[1]
            if out_amount >= amount + fee:
                break
        change = out_amount - amount - fee
        t = Tx.generate(
            user = self.key,
            oneTimeAddresses=ins,
            outs = [
                (pubKey, amount),
                (self.key.getPubKey(), change)
            ]
        )
        if t.verify():
            chain.add_tx(t)
        else:
            raise RuntimeError("Tx verification failed during creation")
```

The `Wallet.send` method searches the one-time addresses owned by the user and generate a list of `TxIn` objects automatically based on the rule that the oldest payment is spent the first.

Node

In this implementation, all the nodes on the network are assumed to be both a wallet and a miner. All the nodes store the whole copy of the blockchain data. The class `Network` represents such network node. Each `Network` object contains three types of data: a `Wallet` object, a `Chain` object and a list of nodes (representing all the peer nodes recognized on the p2p network. All the nodes should be in the format of valid IPv4 string. The `Network` class also provides all the interfaces needed for the node to broadcast to the p2p network. So, it exposes all the necessary interfaces to the peer for nodes, wallet and chain.

```
class Network:
    ...
    """
    methods managing node
    """
    def loadNodes(self):
        ...
    # automated in register and discard
    def dumpNodes(self):
        ...
    # internal helper function
    def registerNode(self, ip):
        ...
    # Called by /registerNode
    def registerNodes(self, ip):
        ...
    # automated in /connect
    def unregisterNode(self, ip):
        ...
    # Called by /node
    def node(self):
        return list(self.nodes)
```

The node data is stored in the same sub-directory with chain data and transaction data: data/ring/node.dat. The above is a list of interfaces concerning nodes

```
class Network:
    ...
    """
    methods managing wallet
    """
    # Called by /address
    def address(self):
        return self.wallet.address()

    # Called by /balance
    def balance(self):
        return self.wallet.amount(self.chain)

    # Called by /mine
    def mine(self):
        self.wallet.mine(self.chain)

    # Called by /transfer
    def transfer(self, address, amount, fee):
        self.wallet.send(
            addr = address,
            chain = self.chain,
            amount = amount,
            fee = fee
        )
```

All the wallet interfaces have been discussed earlier.

```
class Network:
    ...
    """
    methods managing chain
    """
    # Called by /chain
    def getChainInBytes(self):
        return self.chain.serializeBlocks()

    # Called by /tx
    def getTxInBytes(self):
        return self.chain.serializeTxs()

    # automated in /connect
    # s <- Bytes stream
    def replaceChainInBytes(self, s):
        blocks = Chain.parseBlocks(s)
        self.chain.replace(blocks)

    # automated in /connect
    # s <- Bytes stream
    def replaceTxInBytes(self, s):
        txs = Chain.parseTxs(s)
        self.chain.replace_tx(txs)
```

All the chain interfaces have been discussed earlier.

Web

With all the interfaces available, the next step is to decentralize the app. A peer-to-peer network design is hard to implement. So an HTTP web server solution is proposed here with the **Flask** framework in Python. Port 6707 is used for exposing the endpoints.

All the endpoints and methods are listed below:

endpoint	method	description	data
/height	GET	blockchain height	
/queue	GET	number of transactions to be mined	
/nodes	GET	recognized peer nodes	
/register	POST	register a new node	ip=*newNodeIP*
/address	GET	wallet address	
/balance	GET	wallet balance	
/mine	GET	mine a block	

/transfer	POST	make a payment	address=*receiverAddr*
			amount=*transferAmount*
			fee=*feeAmount*

/chain	GET	chain data in octet array	

endpoint	method	description	data
/tx	GET	transaction data in octet array	
/sync	GET	sync all the data with at most 5 random peer nodes	

Part IV: Test and Deployment Guide

Unit tests

The project contains a number of unittest file. All the test file names start with 'test_'. All the unit tests are pure Python unittest module. To pass all the unittest, please make sure the following subdirectories exist under the project repo home directory:

- ${repo_home}/monero/data/ring
- ${repo_home}/monero/secret

To run the unittest, run the command:

```
python -m unittest <test filename>
```

To run all the unittests in the same directory, run the command:

```
python -m unittest test_*
```

The commands above may differ from different platform. The detail please refer to Python unittest module.

The testcase takes some time to finish. All the testcases together takes more than 15 minutes to finish on a *Intel i7* machine.

Deployment guide

The project is ready for deployment as a Docker image. A Dockerfile is provided in the monero subdirectory.

To build the docker image, *change directory* to the monero subdirectory and run:

```
docker build -t <tag-name> .
```

The docker image is published as yniuaa/ring. The docker image is built on an Apple Silicon with ARM64 architecture. So to deploy it on any other architectures, like x86 requires an docker re-build.

Potential Improvement

There are several possible potential improvements in the future:

Range Proof

One missing part not implemented from RingCT is the range proof. Since RingCT hides all the amount, it is possible for the users to trick the network with invalid value (for example negative value) to gain profit. Every transaction contains a number of range proofs to ensure all the amounts in the transaction are valid.

Merkle Tree

The block is big in size now since it contains all the data inside the transaction. Since every transaction input takes 944 bytes and transaction output takes 107 bytes. The transaction size can grow very big. The merkle tree can exclude transaction data from the block data, only include the transaction hash.

P2P Network

The P2P network is not implemented in the project. Although the serialization for disk storage can also be applied for network communication. A p2p network design is definitely preferred than the current network design in the project.

Reference

Abe, M., Ohkubo, M., & Suzuki, K. (2002). 1-out-of-N signatures from a variety of keys. Lecture Notes in Computer Science, 415–432. https://doi.org/10.1007/3-540-36178-2_26

Back, A. (2015, March 1). Ring signature efficiency [web log]. Retrieved August 15, 2022, from https://bitcointalk.org/index.php?topic=972541.msg10619684#msg10619684.

KOE, Alonso, K. M., & Noether, S. (2020). Zero to Monero. getmonero. Retrieved from https://www.getmonero.org/library/Zero-to-Monero-2-0-0.pdf.

Liu, J. K., Wei, V. K., & Wong, D. S. (2004). Linkable spontaneous anonymous group signature for ad hoc groups. Information Security and Privacy, 325–335. https://doi.org/10.1007/978-3-540-27800-9_28

Maxwell, G., & Poelstra, A. (2015). Borromean Ring Signatures ∗.

Noether, S., Mackenzie, A., & Research Lab, T. M. (2016). Ring Confidential Transactions. Ledger, 1, 1–18. https://doi.org/10.5195/ledger.2016.34

Rivest, R. L., Shamir, A., & Tauman, Y. (2001). How to leak a secret. Advances in Cryptology — ASIACRYPT 2001, 552–565. https://doi.org/10.1007/3-540-45682-1_32

Ring signatures. (n.d.). Retrieved August 15, 2022, from https://asecuritysite.com/encryption/ring_sig

Song, J. (2019). Programming Bitcoin. oreilly. O'REILLY. Retrieved August 15, 2022, from https://learning.oreilly.com/library/view/programming-bitcoin/9781492031482/.

Appendix

Meeting Minutes

1. Jun 30 (30 mins): set objectives, timeline for the project
2. July 15 (1 hr): discussed ring signature algorithm and algorithm implementation
3. July 29 (1 hr): discussed Monero technologies and its implementation details
4. Aug 12 (30 mins): reviewed the source code and report