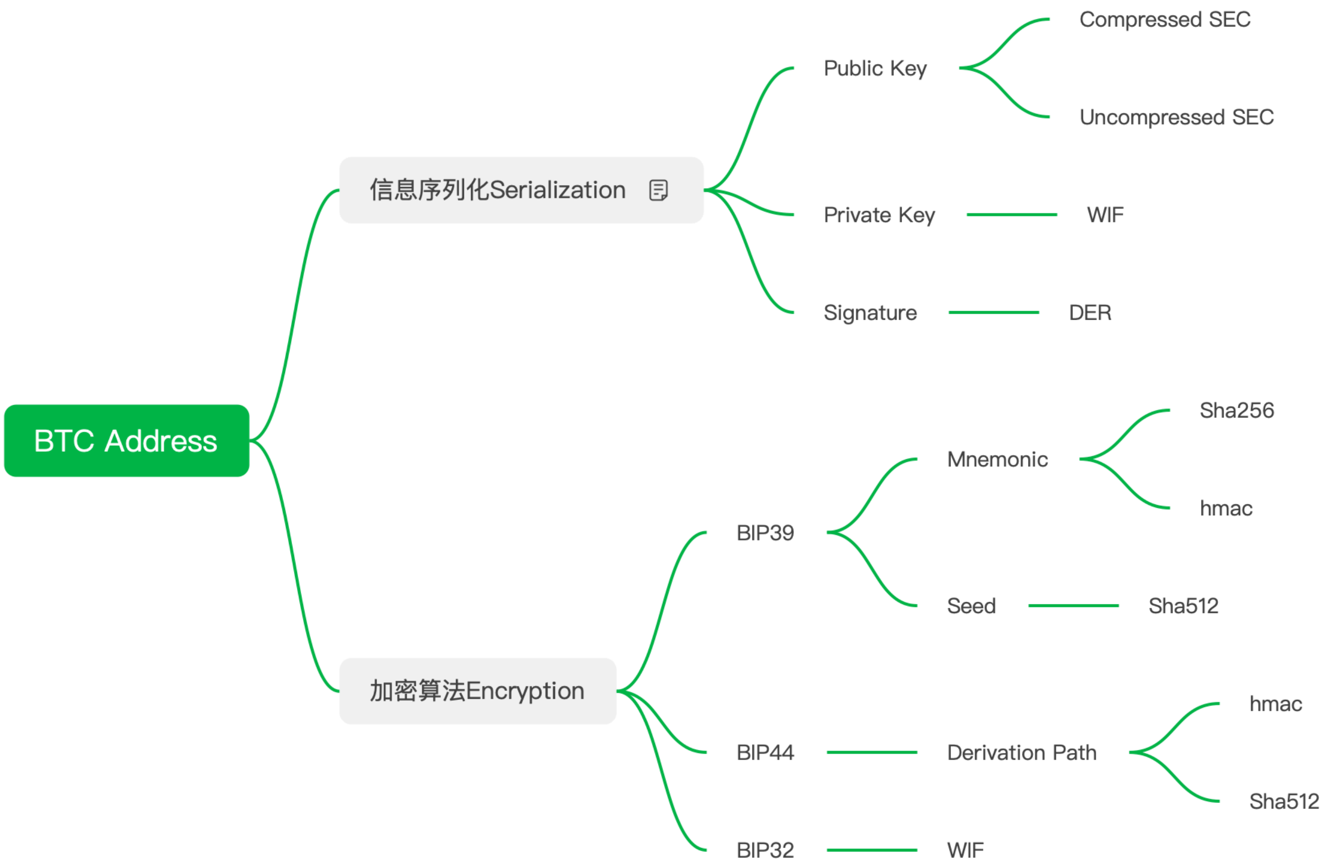
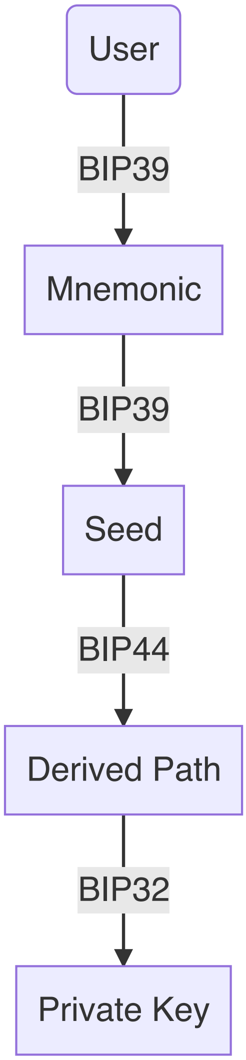
From Mnemonic to Private Key

Author: Alex

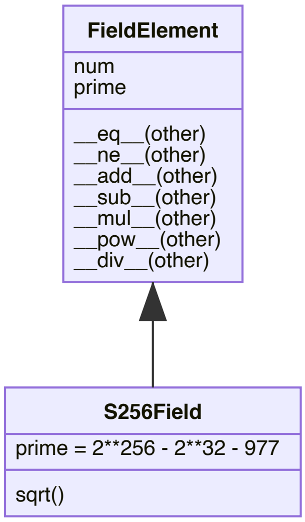


Procedure


User
BIP39
Mnemonic
BIP39
Seed
BIP44
Derived Path
BIP32
Private Key

Finite Field Element & Elliptic Curve Point

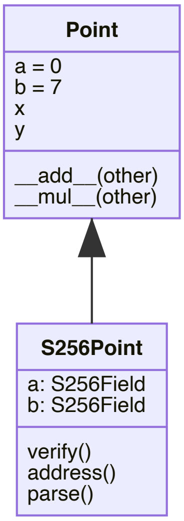
Finite Field Element




```python
class FieldElement:

    def __init__(self, num, prime):
        if num >= prime or num < 0:
            error = 'Num {} not in field range 0 to {}'.format(
                num, prime - 1)
            raise ValueError(error)
        self.num = num
        self.prime = prime

    def __repr__(self):
        return 'FieldElement_{}({})'.format(self.prime, self.num)

    def __eq__(self, other):
        if other is None:
            return False
        return self.num == other.num and self.prime == other.prime

    def __ne__(self, other):
        # this should be the inverse of the == operator
        return not (self == other)

    def __add__(self, other):
        if self.prime != other.prime:
            raise TypeError('Cannot add two numbers in different Fields')
        # self.num and other.num are the actual values
        # self.prime is what we need to mod against
        num = (self.num + other.num) % self.prime
        # We return an element of the same class
```

```
        return self.__class__(num, self.prime)

    def __sub__(self, other):
        if self.prime != other.prime:
            raise TypeError('Cannot subtract two numbers in different Fields')
        # self.num and other.num are the actual values
        # self.prime is what we need to mod against
        num = (self.num - other.num) % self.prime
        # We return an element of the same class
        return self.__class__(num, self.prime)

    def __mul__(self, other):
        if self.prime != other.prime:
            raise TypeError('Cannot multiply two numbers in different Fields')
        # self.num and other.num are the actual values
        # self.prime is what we need to mod against
        num = (self.num * other.num) % self.prime
        # We return an element of the same class
        return self.__class__(num, self.prime)

    def __pow__(self, exponent):
        n = exponent % (self.prime - 1)
        num = pow(self.num, n, self.prime)
        return self.__class__(num, self.prime)

    def __truediv__(self, other):
        if self.prime != other.prime:
            raise TypeError('Cannot divide two numbers in different Fields')
        # self.num and other.num are the actual values
        # self.prime is what we need to mod against
        # use fermat's little theorem:
        # self.num**(p-1) % p == 1
        # this means:
        # 1/n == pow(n, p-2, p)
        num = (self.num * pow(other.num, self.prime - 2, self.prime)) % self.prime
        # We return an element of the same class
        return self.__class__(num, self.prime)

    def __rmul__(self, coefficient):
        num = (self.num * coefficient) % self.prime
        return self.__class__(num=num, prime=self.prime)

class S256Field(FieldElement):

    def __init__(self, num, prime=None):
        super().__init__(num=num, prime=P)

    def __repr__(self):
        return '{:x}'.format(self.num).zfill(64)

    # tag::source2[]
    def sqrt(self):
        return self**((P + 1) // 4)
```

Elliptic Curve




```
class Point:

    def __init__(self, x, y, a, b):
        self.a = a
        self.b = b
        self.x = x
        self.y = y
        # x being None and y being None represents the point at infinity
        # Check for that here since the equation below won't make sense
        # with None values for both.
        if self.x is None and self.y is None:
            return
        # make sure that the elliptic curve equation is satisfied
        # y**2 == x**3 + a*x + b
        if self.y**2 != self.x**3 + a * x + b:
            # if not, throw a ValueError
            raise ValueError('({}, {}) is not on the curve'.format(x, y))

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y \
            and self.a == other.a and self.b == other.b

    def __ne__(self, other):
        # this should be the inverse of the == operator
        return not (self == other)

    def __repr__(self):
        if self.x is None:
            return 'Point(infinity)'
        elif isinstance(self.x, FieldElement):
            return 'Point({},{})_{}_{} FieldElement({})'.format(
                self.x.num, self.y.num, self.a.num, self.b.num, self.x.prime)
        else:
            return 'Point({},{})_{}_{}'.format(self.x, self.y, self.a, self.b)

    def __add__(self, other):
```

```
        if self.a != other.a or self.b != other.b:
            raise TypeError('Points {}, {} are not on the same curve'.format(self, other))
        # Case 0.0: self is the point at infinity, return other
        if self.x is None:
            return other
        # Case 0.1: other is the point at infinity, return self
        if other.x is None:
            return self

        # Case 1: self.x == other.x, self.y != other.y
        # Result is point at infinity
        if self.x == other.x and self.y != other.y:
            return self.__class__(None, None, self.a, self.b)

        # Case 2: self.x ≠ other.x
        # Formula (x3,y3)==(x1,y1)+(x2,y2)
        # s=(y2-y1)/(x2-x1)
        # x3=s**2-x1-x2
        # y3=s*(x1-x3)-y1
        if self.x != other.x:
            s = (other.y - self.y) / (other.x - self.x)
            x = s**2 - self.x - other.x
            y = s * (self.x - x) - self.y
            return self.__class__(x, y, self.a, self.b)

        # Case 4: if we are tangent to the vertical line,
        # we return the point at infinity
        # note instead of figuring out what 0 is for each type
        # we just use 0 * self.x
        if self == other and self.y == 0 * self.x:
            return self.__class__(None, None, self.a, self.b)

        # Case 3: self == other
        # Formula (x3,y3)=(x1,y1)+(x1,y1)
        # s=(3*x1**2+a)/(2*y1)
        # x3=s**2-2*x1
        # y3=s*(x1-x3)-y1
        if self == other:
            s = (3 * self.x**2 + self.a) / (2 * self.y)
            x = s**2 - 2 * self.x
            y = s * (self.x - x) - self.y
            return self.__class__(x, y, self.a, self.b)

    def __rmul__(self, coefficient):
        coef = coefficient
        current = self
        result = self.__class__(None, None, self.a, self.b)
        while coef:
            if coef & 1:
                result += current
            current += current
            coef >>= 1
        return result

class S256Point(Point):

    def __init__(self, x, y, a=None, b=None):
        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)
```

```python
    def __repr__(self):
        if self.x is None:
            return 'S256Point(infinity)'
        else:
            return 'S256Point({}, {})'.format(self.x, self.y)

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

    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
```

Private Key

```python
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)

    def deterministic_k(self, z):
        k = b'\x00' * 32
        v = b'\x01' * 32
        if z > N:
            z -= N
        z_bytes = z.to_bytes(32, 'big')
        secret_bytes = self.secret.to_bytes(32, 'big')
        s256 = hashlib.sha256
        k = hmac.new(k, v + b'\x00' + secret_bytes + z_bytes, s256).digest()
        v = hmac.new(k, v, s256).digest()
        k = hmac.new(k, v + b'\x01' + secret_bytes + z_bytes, s256).digest()
        v = hmac.new(k, v, s256).digest()
        while True:
            v = hmac.new(k, v, s256).digest()
```

```python
            candidate = int.from_bytes(v, 'big')
            if candidate >= 1 and candidate < N:
                return candidate
            k = hmac.new(k, v + b'\x00', s256).digest()
            v = hmac.new(k, v, s256).digest()

    # tag::source6[]
    def wif(self, compressed=True, testnet=False):
        secret_bytes = self.secret.to_bytes(32, 'big')
        if testnet:
            prefix = b'\xef'
        else:
            prefix = b'\x80'
        if compressed:
            suffix = b'\x01'
        else:
            suffix = b''
        return encode_base58_checksum(prefix + secret_bytes + suffix)
```

Serialization

After clarifying a lot of terms, including PrivateKey, PublicKey, and Signature, we want to communicate or store them in an efficient way. Steps by steps, I would explain how they are encrpyted in the real apps.

Uncompressed SEC

For public keys, there's already a standard for serializing ECDSA (Elliptic Curve Digital Signature Algorithm), which is called *Standards for Efficient Cryptography (SEC)*. There are two forms of SEC: compressed and umcompressed.

The uncompressed SEC format for a given point $P = (x, y)$ is generated by:

1. Start with the prefix byte, which is 0x04
2. Append the x coordinate in 32 bytes as a big-endian integer.
3. Append the y coordinate in 32 bytes as a big-endian integer.

```
class S256Point(Point):
  def sec(self):
      return b'\x04' + self.x.num.to_bytes(32, 'big') + \
             self.y.num.to_bytes(32, 'big')
```

Compressed SEC

Recall for any x coordinate, there are at most corresponding y coordinates in the elliptic curve $y^2 = x^3 + ax + b$. Through a preodained odevity, we can shorten the uncompressed SEC format by providing the x coordinate and the evenness of the y coordinate.

The compressed SEC format for a given point $P = (x, y)$ is generated by:

1. Start with the prefix byte. If y is even, it's 0x02; otherwise, it's 0x03.

2. Append the x coordinate in 32 bytes as a big-endian integer.

```
class S256Point(Point):
  def sec(self, compressed=True):
        '''returns the binary version of the SEC format'''
        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:
            return b'\x04' + self.x.num.to_bytes(32, 'big') + \
                self.y.num.to_bytes(32, 'big')
```

Parse SEC pubkey

How to calculate a square root in a finite field? Assume w and v are both Finite Field Elements, we can do by following:

$$w^2 = v$$
$$w = w^{(p+1)/2} = (w^2)^{(p+1)/4} = v^{(p+1)/4}$$
$$w = v^{(p+1)/4}$$

Thus, we can determine evenness and return the correct point.

```
def parse(sec_bin):
        '''returns a Point object from a SEC binary (not hex)'''
        if sec_bin[0] == 4:  # <1>
            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  # <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()  # <3>
        if beta.num % 2 == 0:  # <4>
            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)
```

Base58

BTC address needs to encode numbers in Base58. All numbers, uppercase letters, and lowercase letters are utilized, except for 0/O and l/I.

```
BASE58_ALPHABET = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz'

def encode_base58(s):
    count = 0
    for c in s:  # <1>
        if c == 0:
            count += 1
        else:
            break
    num = int.from_bytes(s, 'big')
    prefix = '1' * count
    result = ''
    while num > 0:  # <2>
        num, mod = divmod(num, 58)
        result = BASE58_ALPHABET[mod] + result
    return prefix + result  # <3>
```

Address Format

By not using the SEC format directly, we can go from 33 bytes to 20 bytes, shortening the address significantly.

Here is how a Bitcoin address is created:

1. For mainnet addresses, start with the prefix 0x00, for testnet 0x6f.
2. Take the SEC format(compressed or uncompressed) and do a sha256 operation followed by the ripemd160 hash operation, the combination of which is called a hash160 operation.
3. Combine the prefix from #1 and resulting hash from #2.
4. Do a hash256 of the result from #3 and get the first 4 bytes.
5. Take the combination of #3 and #4 and encode it in Base58.

```
def hash160(s):
    '''sha256 followed by ripemd160'''
    return hashlib.new('ripemd160', hashlib.sha256(s).digest()).digest()  # <1>

def encode_base58_checksum(b):
    return encode_base58(b + hash256(b)[:4])
```

WIF Format

The private key till now is tstill a 256-bit number. Wallet Import Format (WIF) is a serialization of the private key widely used in wallet apps.

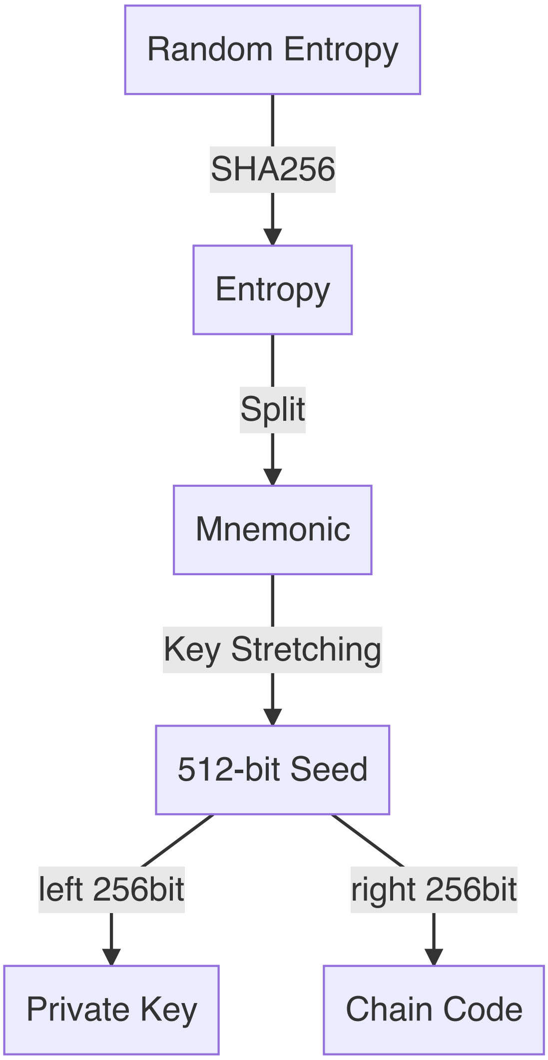
Here is how the WIF format is created:

1. For mainnet private keys, start with the prefix 0x80, for testnet 0xef.
2. Encode the secret in 32-byte big-endian.
3. If the SEC format used for the public key address was compressed, add a suffix of 0x01.
4. Combine the prefix from #1, serialized secret from #2, and suffix from #3.
5. Do a hash256 of the result from #4 and get the first 4 bytes.
6. Take the combination of #4 and #5 and encode it in Base58.

```
class PrivateKey:
 def wif(self, compressed=True, testnet=False):
        secret_bytes = self.secret.to_bytes(32, 'big')
        if testnet:
            prefix = b'\xef'
        else:
            prefix = b'\x80'
        if compressed:
            suffix = b'\x01'
        else:
            suffix = b''
        return encode_base58_checksum(prefix + secret_bytes + suffix)
```

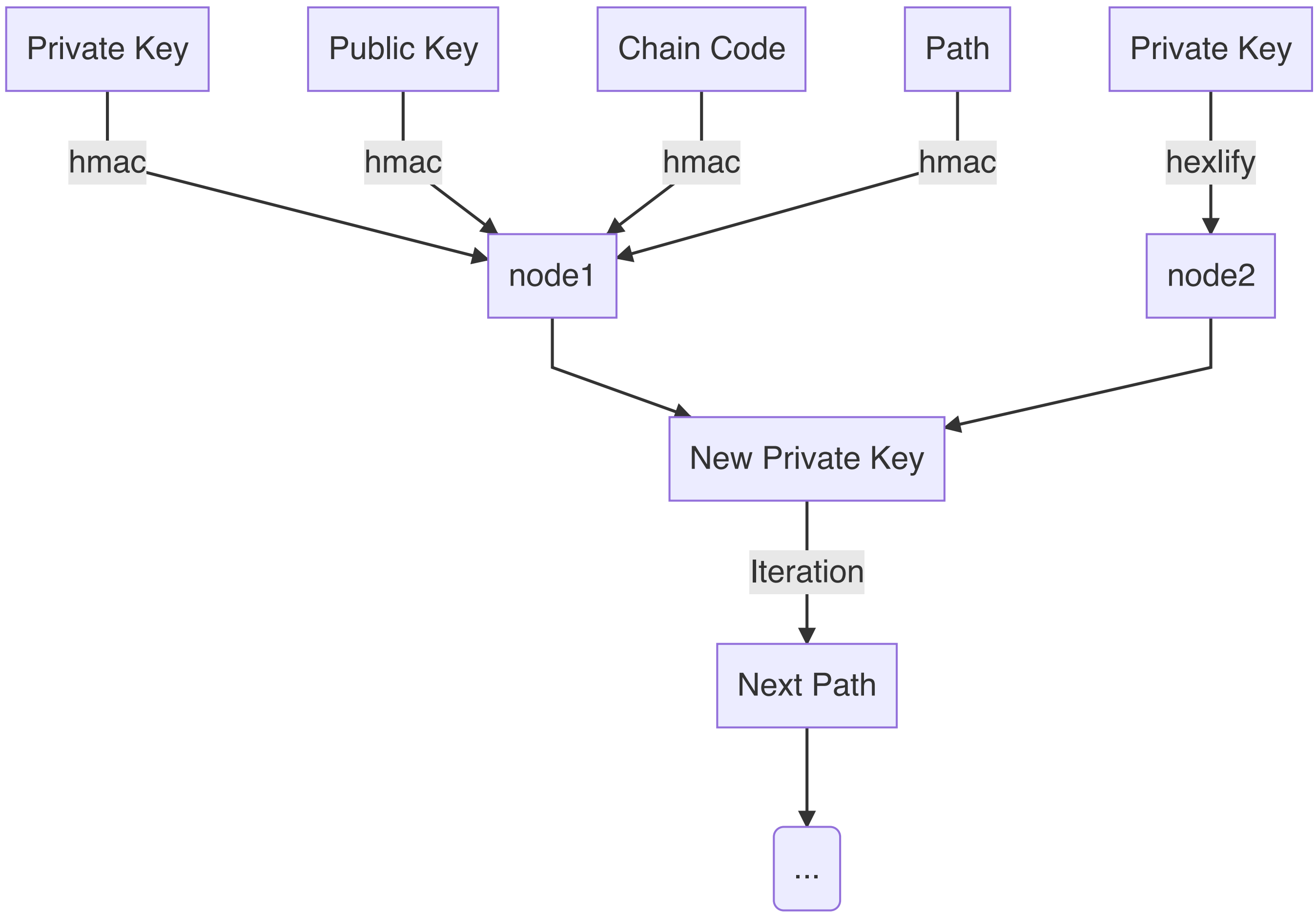
BIPs

BIP39




```python
import hmac
import hashlib
passphrase = 'play crush alone level street fox hockey impose develop waste kind fluid'
seed = bip39.bip39_seed_from_mnemonic(passphrase) # Key Stretching

seed = hmac.new(b"Bitcoin seed", seed, hashlib.sha512).digest()
chain_code = seed[32:]
seed = seed[:32]
```

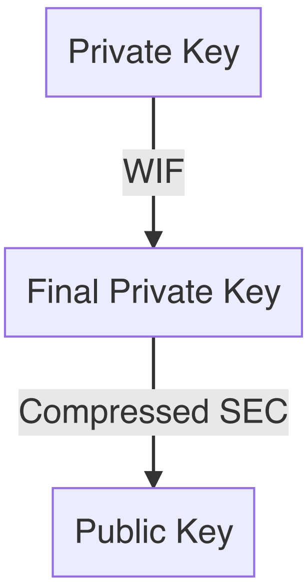
BIP44




```
import struct
from binascii import unhexlify,hexlify
import hmac, hashlib

CURVE_GEN = ecdsa.ecdsa.generator_secp256k1
CURVE_ORDER: int = CURVE_GEN.order()
BIP32KEY_HARDEN: int = 0x80000000

key = seed
secret = seed
PK = PrivateKey(int.from_bytes(seed,'big'))
print(PK.wif(compressed=True,testnet=False))

path = "m/44'/0'/0'/0/0"
for raw_index in path.lstrip("m/").split("/"):
    if "'" in raw_index:
        index = int(raw_index[:-1]) + BIP32KEY_HARDEN
        i_str = struct.pack(">L", index)

        if index & BIP32KEY_HARDEN:
            data = b"\0" + key + i_str #这里的key是private key 格式为byte
        else:
            data = unhexlify(public_key) + i_str #这里的public key 格式为hex
```

```
        i = hmac.new(chain_code, data, hashlib.sha512).digest()
        il, ir = i[:32], i[32:]

        il_int = int(hexlify(il),16)
        pvt_int = int(hexlify(secret),16)
        k_int = (il_int + pvt_int) % CURVE_ORDER

        secret = (b"\0" * 32 + int(k_int).to_bytes(32,'big'))[-32:]
        PK = PrivateKey(int.from_bytes(secret,'big'))
        public_key = hexlify(PK.point.sec()).decode()
        key = secret
        chain_code = ir
    else:
        index = int(raw_index)
        i_str = struct.pack(">L", index)

        if index & BIP32KEY_HARDEN:
            data = b"\0" + key + i_str #这里的key是private key 格式为byte
        else:
            data = unhexlify(public_key) + i_str #这里的public key 格式为hex

        i = hmac.new(chain_code, data, hashlib.sha512).digest()
        il, ir = i[:32], i[32:]

        il_int = int(hexlify(il),16)
        pvt_int = int(hexlify(secret),16)
        k_int = (il_int + pvt_int) % CURVE_ORDER

        secret = (b"\0" * 32 + int(k_int).to_bytes(32,'big'))[-32:]
        PK = PrivateKey(int.from_bytes(secret,'big'))
        public_key = hexlify(PK.point.sec()).decode()
        key = secret
        chain_code = ir
```

BIP32



```python
print(PK.wif(compressed=True,testnet=False))
print(PK.point.address(compressed=True,testnet=False))
```