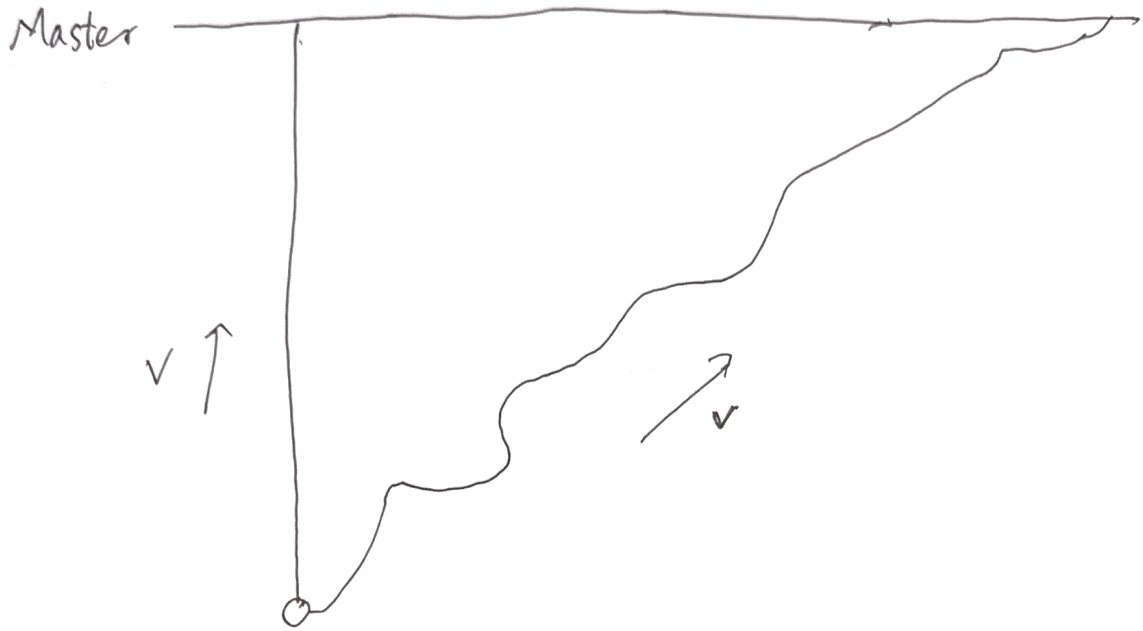
Author: Alex

1 Learning Curve



2 Reference

- Hoffstein, Jeffrey, et al. An introduction to mathematical cryptography. Vol. 1. New York: Springer, 2008.
- Silverman, Joseph H. The arithmetic of elliptic curves. Vol. 106. New York: Springer, 2009.
- Bitansky, Nir, et al. "From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again." Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. 2012.
- Parno, Bryan, et al. "Pinocchio: Nearly practical verifiable computation." 2013 IEEE Symposium on Security and Privacy. IEEE, 2013.
- Petkus, Maksym. "Why and How zk-SNARK Works: Definitive Explanation."
- Dutta, Ratna, Rana Barua, and Palash Sarkar. "Pairing-Based Cryptographic Protocols: A Survey." IACR Cryptol. ePrint Arch. 2004 (2004): 64.
- Menezes, Alfred. "An introduction to pairing-based cryptography." Recent trends in cryptography 477 (2009): 47-65.

3 Elliptic Curve

3.1 Secp256k1

Elliptic curves have a form like this:

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

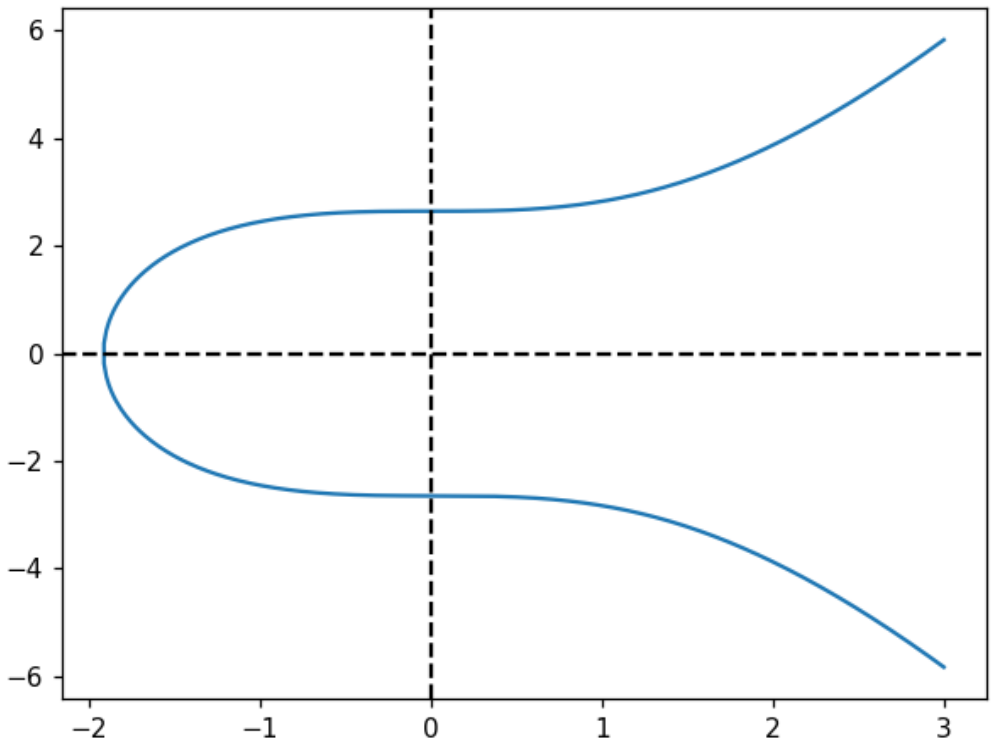
Secp256k1 has the following parameter

$$A = 0$$
$$B = 7$$

In [1]:

```
%matplotlib notebook
import numpy as np
import matplotlib.pyplot as plt

A = 0
B = 7
x = np.linspace(-4,3,500)
ySquare = x**3 + A*x + B
x = x[ySquare>=0]
y = np.sqrt(ySquare[ySquare>=0])
y = np.append(y[::-1],-y)
x = np.append(x[::-1],x)
plt.plot(x,y)
plt.axhline(0, color='black', linestyle="--")
plt.axvline(0, color='black', linestyle="--")
plt.show()
```

```
<IPython.core.display.Javascript object>
```

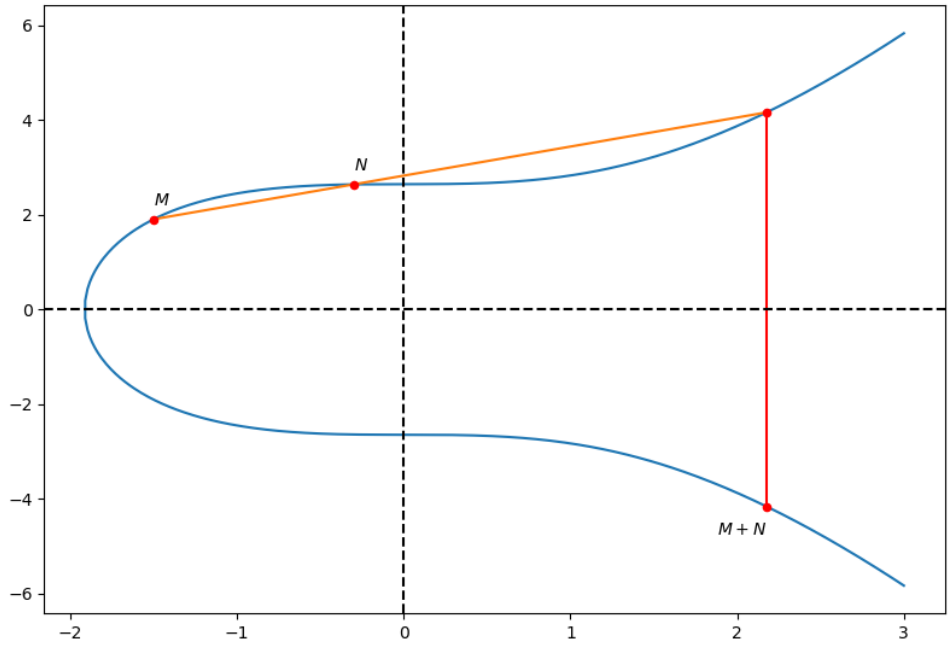


3.2 Addition on Real Set of Numbers

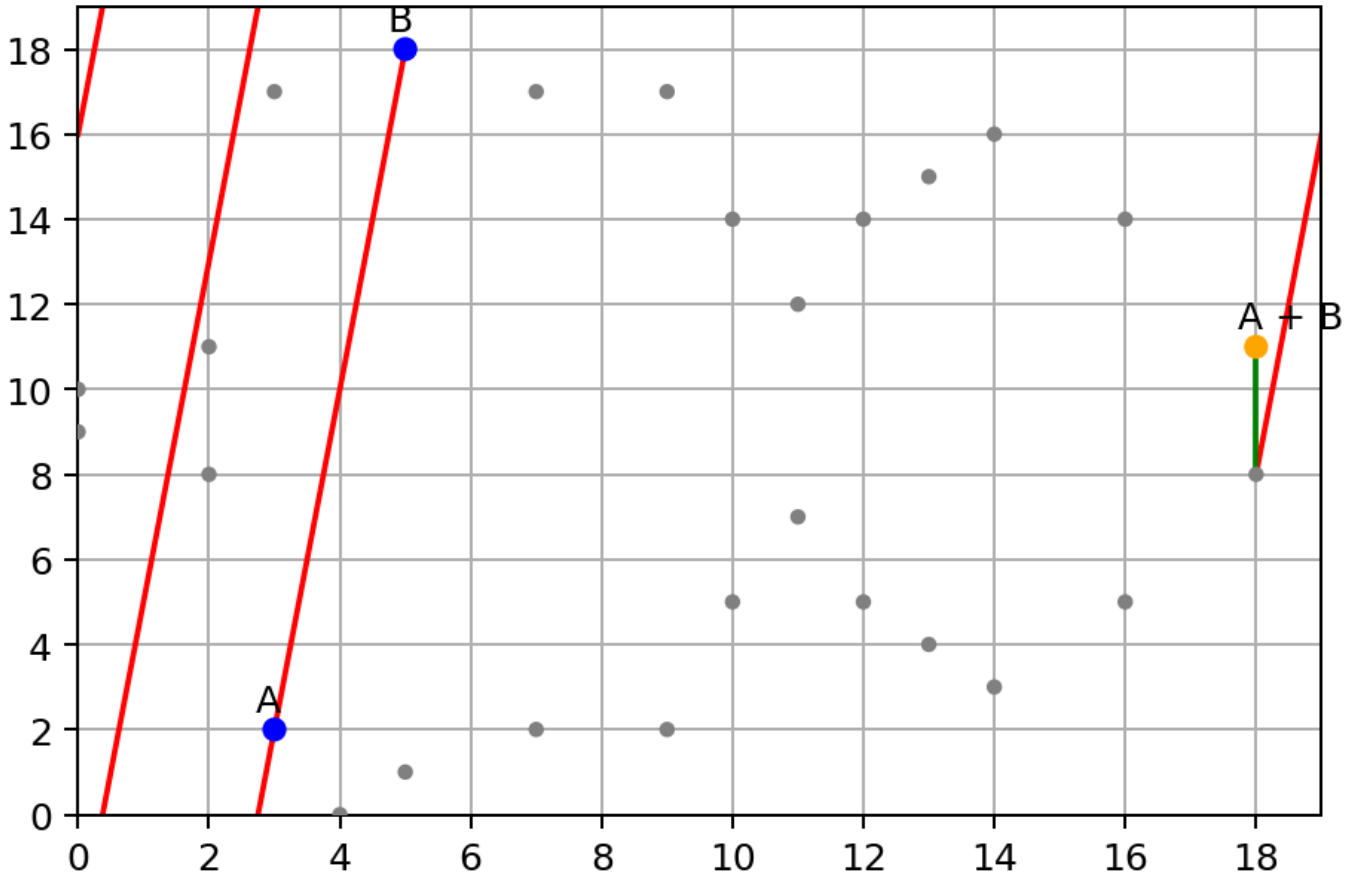
In [62]:

```
A = 0
B = 7
x = np.linspace(-4,3,500)
secp256k1 = lambda x: x**3 + A*x + B
ySquare = secp256k1(x)
x = x[ySquare>=0]
y = np.sqrt(ySquare[ySquare>=0])
y = np.append(y[::-1],-y)
x = np.append(x[::-1],x)

x1 = -1.5
x2 = -0.3
y1 = np.sqrt(secp256k1(x1))
y2 = np.sqrt(secp256k1(x2))
s = (y2-y1)/(x2-x1)
intercept = y1 - s*x1
#y = ax + b
x3 = s**2 - x1 - x2
y3 = s*(x1-x3) - y1
xStraight = np.linspace(x1,x3,100)
yStraight = s * xStraight + intercept

fig, ax = plt.subplots(1,1)
ax.plot(x,y)
ax.plot(xStraight,yStraight)
ax.scatter([x1],[y1],s = 20,color='r', zorder=3)
ax.scatter([x2],[y2],s = 20,color='r', zorder=3)
ax.scatter([x3],[-y3],s = 20,color='r', zorder=3)
ax.scatter([x3],[y3],s = 20,color='r', zorder=3)
ax.vlines(x3,-y3,y3,'r')
ax.annotate(r'$M$',
            xy=(x1, y1+0.3)
            )
ax.annotate(r'$N$',
            xy=(x2, y2+0.3)
            )

ax.annotate(r'$M+N$',
            xy=(x3-0.3, y3-0.6)
            )
ax.axhline(0, color='black', linestyle="--")
ax.axvline(0, color='black', linestyle="--")
plt.show()
```

```
<IPython.core.display.Javascript object>
```



3.3 Curve on Finite Field Set



4 Encryption (ECDSA)

No.

Date. . .

Cont. ⇒

$$-s^2(x-x_1)x + x^3 + (x_1 s^2 - 3x_1)x + 2x_1^3 - s^2 x_1^2 = 0$$

$$(x-x_1)(x^2 + (x_1 - s^2)x - 2x_1^2 + s^2 x_1) = 0$$

$$(x-x_1)(x-(s^2-2x_1))(x-x_1) = 0$$

$$x_3 = s^2 - 2x_1$$

Thus $y_3 = s(x_1 - x_3) - y_1$

Elliptic Curve Digital Signature Algorithm (ECDSA)

$P = eG$

G is a Elliptic Curve point, and has order n. $1G, 2G \dots nG$

P is the public key, e is the private key

Please note calculating $P/G = e$ is extremely difficult

If we want to prove our possession of e, we create a target $R = kG$, where k is not revealed. R's x coordinate is r. Then we have $uG + vP = kG$. Thus, $veG = (k-u)G$, now ve and $(k-u)$ can be regarded as Field element, following the rule of scalar multiplication of elliptic curves.

$$\therefore e = \frac{k-u}{v}$$

Since R is on the curve, $R \to r$. As long as we can provide (u, v, r), we can prove that we have e.

No.

Date . . .

How to create a signature:

① Given z, the signature hash, which is also the intent of the verification and verifies the messages already known.
E.g. $z = \text{hash256}(\text{'Programming Bitcoin'})$

② figure out a K
A unique K is very important! (see Plog of the book)
$R = kG \Rightarrow r$

③ $u = z/s \Rightarrow$ Finite Field Element division
$v = r/s$
where $s = \frac{z + re}{k}$

④ Then we have signature (r,s)
easy to verify $uG + vP = R = kG$

Verify the signature
① given (r,s) as the signature
② calculate $u = z/s$, $v = r/s$
③ calculate $uG + vP = R$
④ verify R's x coordinate equals r

No.

Date. . .

Fake the signature:

given z, we need to create a (r,s) satisfy:

$$uG + vP = R$$

$$\frac{r}{s}P = (k - \frac{z}{s})G$$

As mentioned, it's extremely difficult because if we create ans, $(k - \frac{z}{s})G$ is easy to calculate, but cannot know $(k - \frac{z}{s})G$ equal to how many P !!.

4.1 Create Secret Key

In [1]:

```
from ecc import PrivateKey
from helper import hash256, little_endian_to_int
secret = little_endian_to_int(hash256(b'Huobi Ventures Seminar Friday
74530sdfsdf'))
private_key = PrivateKey(secret)
print(private_key.point.address(testnet=True))
```

```
msNyWmcYJCfQeHE3Sz7cuuUXnN7DenEJnF
```

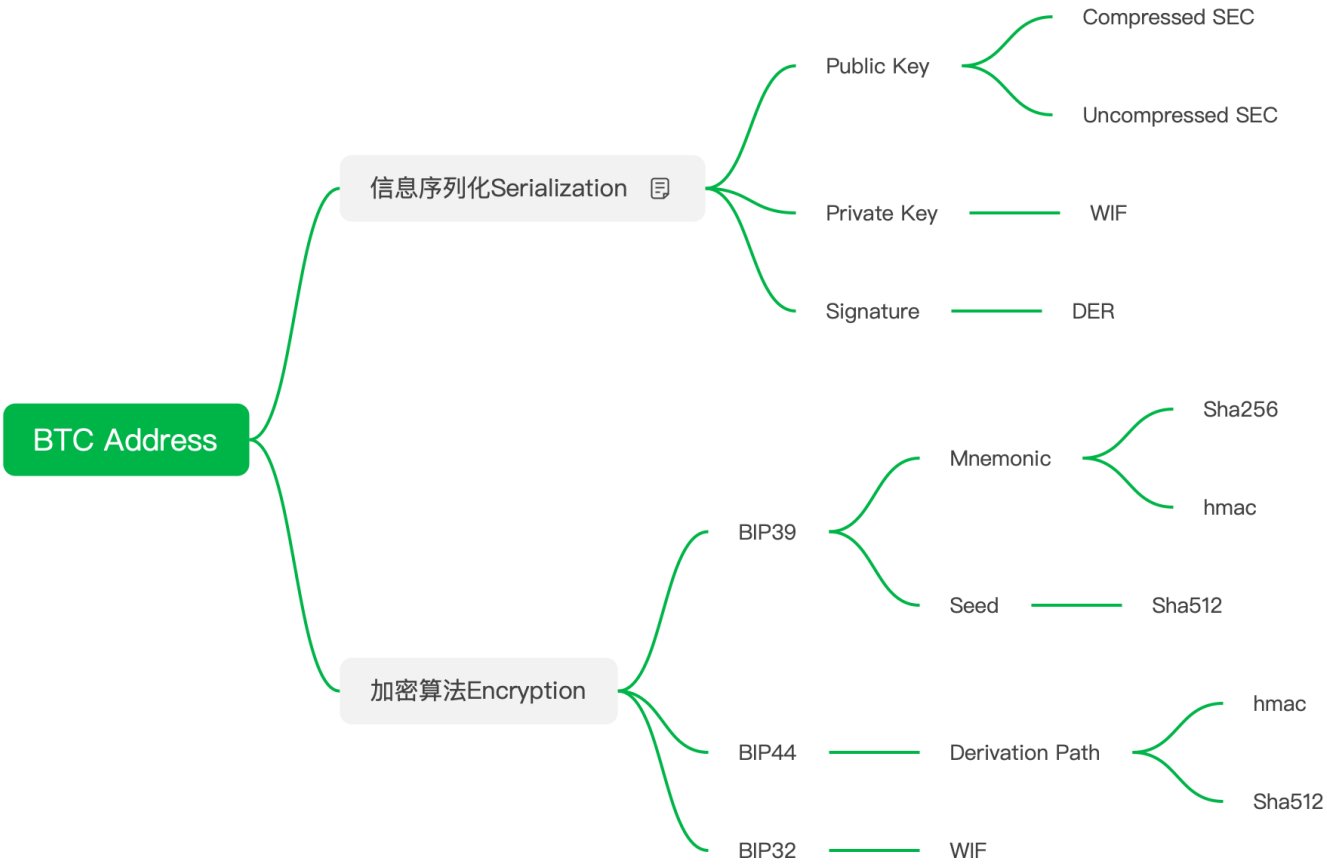
4.2 Transaction Hash To Script

In [2]:

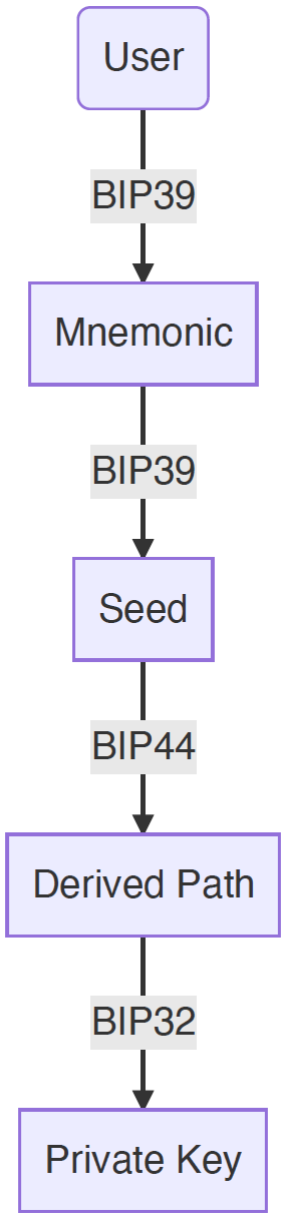
```
import re
from helper import run
import ecc
import helper
import script
from script import Script

import tx
from helper import (
    encode_varint,
    hash256,
    int_to_little_endian,
    little_endian_to_int,
    read_varint,
)

hexstr = '76a988ac'
script_pubkey = Script([eval('0x'+hexstr[i:i+2]) for i in range(0, len(hexstr),
2)])
print(script_pubkey)
```

```
OP_DUP OP_HASH160 OP_EQUALVERIFY OP_CHECKSIG
```

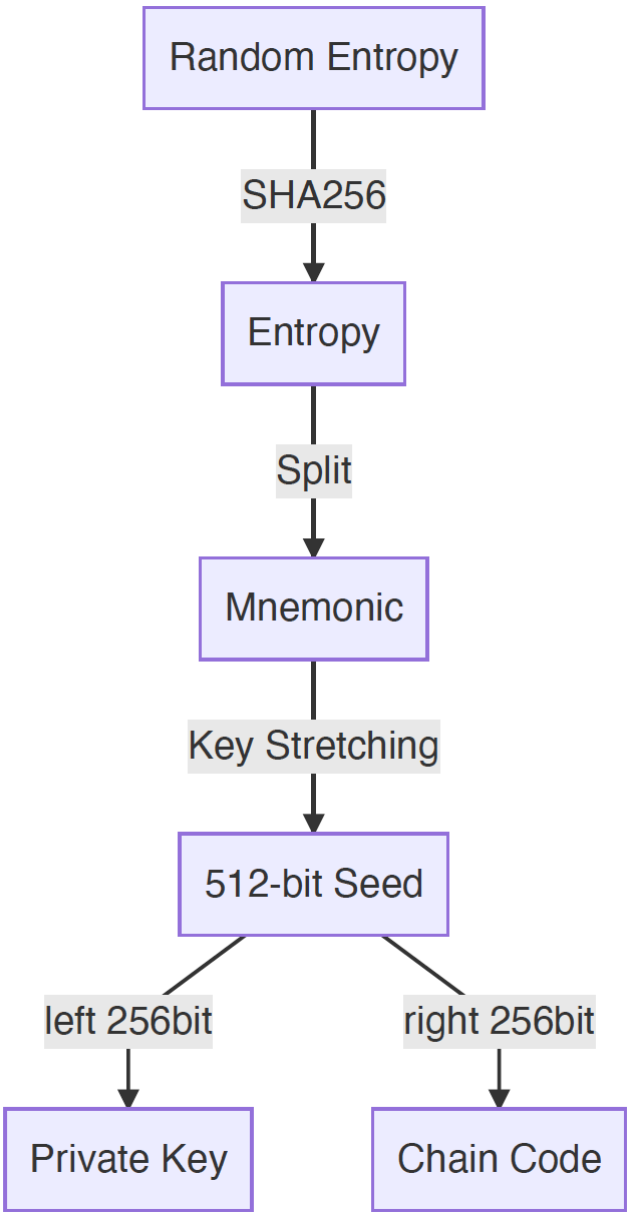
5 Mnemonic to Private Key



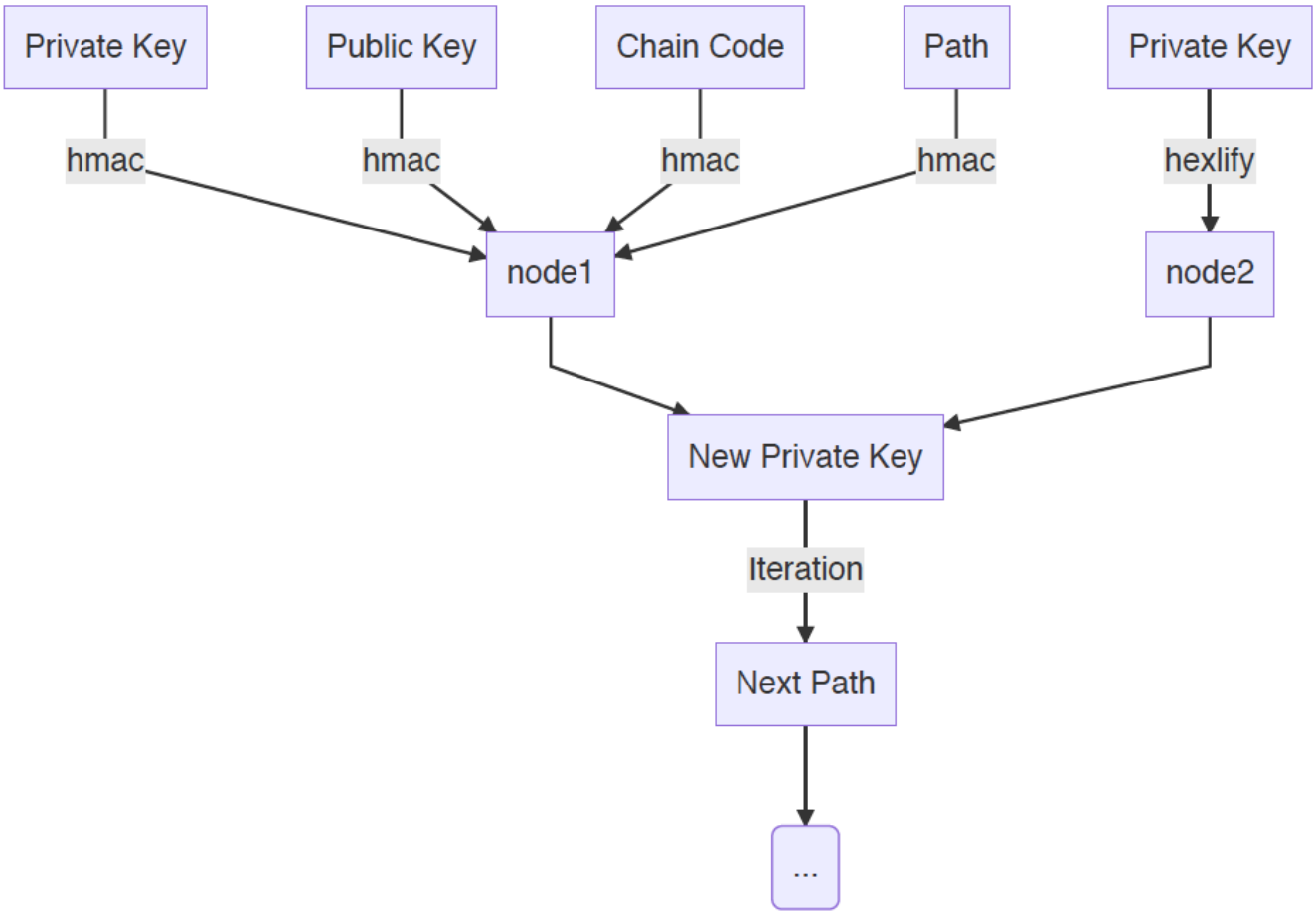
5.1 Procedure


User
BIP39
Mnemonic
BIP39
Seed
BIP44
Derived Path
BIP32
Private Key

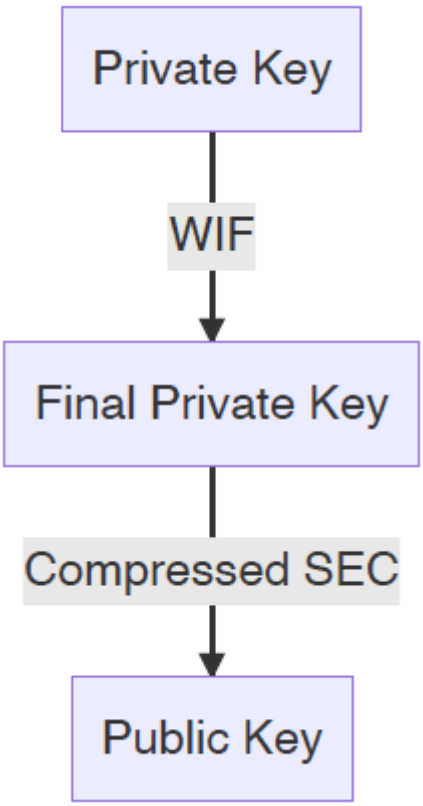
5.2 BIP 39


Random Entropy
SHA256
Entropy
Split
Mnemonic
Key Stretching
512-bit Seed
left 256bit
right 256bit
Private Key
Chain Code

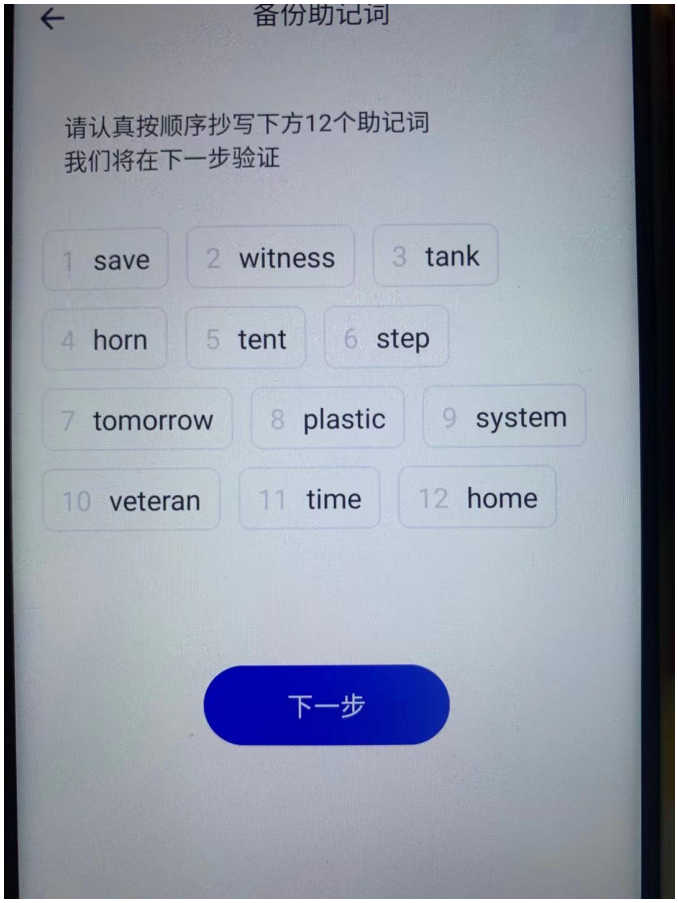
5.3 BIP 44

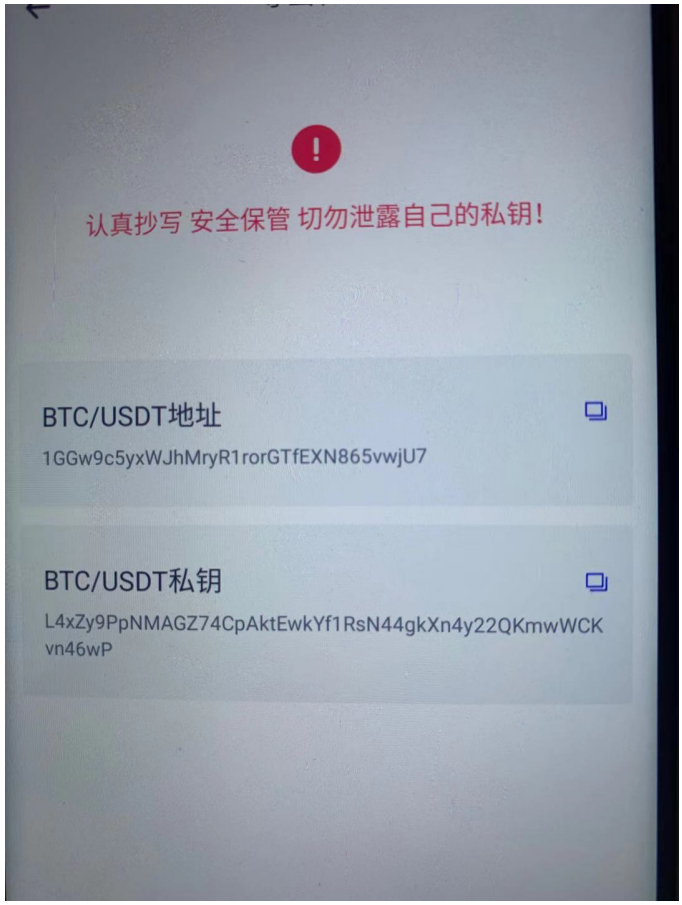


5.4 BIP 32



5.5 Get Private Key


备份助记词
请认真按顺序抄写下方12个助记词
我们将在下一步验证
1 save
2 witness
3 tank
4 horn
5 tent
6 step
7 tomorrow
8 plastic
9 system
10 veteran
11 time
12 home
下一步


认真抄写 安全保管 切勿泄露自己的私钥!
BTC/USDT地址
1GGw9c5yxWJhMryR1rorGTfEXN865vwjU7
BTC/USDT私钥
L4xZy9PpNMAGZ74CpAktEwkYf1RsN44gkXn4y22QKmwWCKvn46wP

In [3]:

```
from importlib import reload
from helper import run
import ecc
import helper
import ecdsa
import struct
from binascii import unhexlify,hexlify
import hmac, hashlib
from btc_hd_wallet import bip39, BaseWallet
import hmac
import hashlib
from ecc import PrivateKey

passphrase = 'save witness tank horn tent step tomorrow plastic system veteran time
home'
# the mnemonic
seed = bip39.bip39_seed_from_mnemonic(passphrase)
#print("Seed:",seed)
seed = hmac.new(b"Bitcoin seed", seed, hashlib.sha512).digest()
chain_code = seed[32:]
seed = seed[:32]

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

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

```
L4xZy9PpNMAGZ74CpAktEwkYf1RsN44gkXn4y22QKmwWCKvn46wP
```

6 Schnorr Signature

6.1 Generation

Remember P is the public key and $P = eG$, m is the info waiting for a signature, the procedure of signing of a Schnorr signature is:

- select a random k, let $R = kG$
- let $s = k + Hash(m||R||P) \times e$

Thus (R,s) is a Schnorr signature.

6.2 Group Verification

Assume now we have two Schnorr signatures: (R_1, s_1) and (R_2, s_2).

- let $R = R_1 + R_2$ and $s = s_1 + s_2$
- We know $s_1 = k_1 + Hash(m||R||P) \times e_1$ and $s_2 = k_2 + Hash(m||R||P) \times e_2$

Thus,

$$\begin{aligned} s &= s_1 + s_2 \\ &= (k_1 + k_2) + Hash(m||R||P) \times (e_1 + e_2) \end{aligned}$$

Finally,

$$\begin{aligned} sG &= (k_1 + k_2)G + Hash(m||R||P) \times (e_1 + e_2)G \\ &= (R_1 + R_2) + Hash(m||R||P) \times (P_1 + P_2) \\ &= R + Hash(m||R||P) \times P \end{aligned}$$

7 Taproot

7.1 BIP340 Schnorr Signatures for secp256k1

This document proposes a standard for 64-byte Schnorr signatures over the elliptic curve secp256k1.

- **Provable security:** Schnorr signatures are provably secure. In more detail, they are strongly unforgeable under chosen message attack (SUF-CMA)
- **Non-malleability:** The SUF-CMA security of Schnorr signatures implies that they are non-malleable. On the other hand, ECDSA signatures are inherently malleable[3]; a third party without access to the secret key can alter an existing valid signature for a given public key and message into another signature that is valid for the same key and message.
- **Linearity:** Schnorr signatures provide a simple and efficient method that enables multiple collaborating parties to produce a signature that is valid for the sum of their public keys. This is the building block for various higher-level constructions that improve efficiency and privacy, such as multisignatures and others (see Applications below).

7.2 BIP341 Taproot: SegWit version 1 spending rules

This document proposes a new SegWit version 1 output type, with spending rules based on Taproot, Schnorr signatures, and Merkle branches.

- **Permit key aggregation:** a public key can be constructed from multiple participant public keys, and which requires cooperation between all participants to sign for. Such multi-party public keys and signatures are indistinguishable from their single-party equivalents.
- **Batch validation:** allowing multiple signatures to be validated together more efficiently than validating each one independently, we make sure all parts of the design are compatible with this.
- **Public key** is directly included in the output in contrast to typical earlier constructions which store a hash of the public key or script in the output.

7.3 BIP342 Validation of Taproot Scripts

This document specifies the semantics of the initial scripting system under BIP341.

7.4 Transaction Cost

$$\text{Transaction Cost} = (input \times 148 + out \times 34 + 10) \times satoshis/byte$$

Input 148 bytes:

- Transaction ID (32 Bytes)
- Index (4B)
- Script Length (1 ~ 9B)
- ScriptSig(107B?)*
- Sequence (4B)

Output 34 bytes:

- Value (8B)
- Script Length (1 ~ 9B)
- Script (25B?)*
- Outpoint (36B)

Source 10 bytes:

- Version (4 Bytes)
- LockTime (4B)
- TxIn Count (1 ~ 9B)

- TxOut Count (1 ~ 9B)

Taproot's signature and Public Key are 64 bytes and 32 bytes respectively. In ECDSA, these two figures are 72 and 3 3. In a one-to-one transaction, the transaction fee can be lowered by 5 percent.

8 SegWit's Impact

Instead of the previous (legacy) consensus rule which stated that a block cannot be more than 1,000,000 bytes, the new, tighter, consensus rule is that the total block weight allowed for any block is 4,000,000. Notice that if you fill a block with only non-Segwit transactions, this is equivalent to a block size limit of 1,000,000 bytes (1,000,000 * 4 = 4,000,000).

- A 1000 byte Segwit transaction with 500 bytes of witness data
- A 1000 byte Segwit transaction with 300 bytes of witness data

The first would have a weight of $500 \times 3 + 1000 = 2500$ whereas the second would have a weight of $700 \times 3 + 1000 = 3100$. From a miner's perspective, if both transactions had the same fee, they would make more money including the former transaction as it takes up less weight in a block despite the same size.

8.1 Smaller weight of Transaction

Transaction 1

```
    {'hash': 'a8d0c0184dde994a09ec054286f1ce581bebf46446a512166eae7628734ea0
a5',
     'ver': 1,
     'vin_sz': 1,
     'vout_sz': 1,
     'size': 148,
     'weight': 592,
     'fee': 0,
     'relayed_by': '0.0.0.0',
     'lock_time': 0,
     'tx_index': 5827453744647893,
     'double_spend': False,
     'time': 1456417484,
     'block_index': 400000,
     'block_height': 400000,
     'inputs': [{'sequence': 4294967295,
       'witness': '',
       'script': '03801a060004cc2acf560433c30f37085d4a39ad543b0c000a42572053
7570706f727420384d200a666973686572206a696e78696e092f425720506f6f6c2f',
       'index': 0,
       'prev_out': None}],
     'out': [{'type': 0,
       'spent': True,
       'value': 2533349423,
       'spending_outpoints': [{'tx_index': 3843156473962473, 'n': 0}],
       'n': 0,
       'tx_index': 5827453744647893,
       'script': '76a914721afdf638d570285d02d3076d8be6a03ee0794d88ac',
       'addr': '1BQLNJtMDKmMZ4PyqVFfRuBNvoGhjigBKF'}]}
        // Legacy Address
```


Transaction 2

```
    {'hash': 'fe6c48bbfdc025670f4db0340650ba5a50f9307b091d9aaa19aa44291961c69f',
     'ver': 1,
     'vin_sz': 1,
     'vout_sz': 1,
     'size': 215,
     'weight': 533,
     'fee': 1704,
     'relayed_by': '0.0.0.0',
     'lock_time': 0,
     'tx_index': 5621580204157077,
     'double_spend': False,
     'time': 1513622125,
     'block_index': 500000,
     'block_height': 500000,
     'inputs': [{'sequence': 4294967295,
       'witness': '0247304402205f39ccbab38b644acea0776d18cb63ce3e37428cbac06dc23b59c61607aef69102206b8610827e9cb853ea0ba38983662034bd3575cc1ab118fb66d6a98066fa0bed01210304c01563d46e38264283b99bb352b46e69bf132431f102d4bd9a9d8dab075e7f',
       'script': '1600142b2296c588ec413cebd19c3cbc04ea830ead6e78',
       'index': 0,
       'prev_out': {'spent': True,
        'script': 'a914994394dbd20b7752e272458c738ae9b7666271b787',
        'spending_outpoints': [{'tx_index': 5621580204157077, 'n': 0}],
        'tx_index': 7505813802705764,
        'value': 34676070,
        'addr': '3FfQGY7jqsADC7uTVqF3vKQzeNPiBPTqt4',
        'n': 0,
        'type': 0}}],
     'out': [{'type': 0,
       'spent': True,
       'value': 34674366,
       'spending_outpoints': [{'tx_index': 1069003796305379, 'n': 3}],
       'n': 0,
       'tx_index': 5621580204157077,
       'script': 'a91487e4e5a7ff7bf78b8a8972a49381c8a673917f3e87',
       'addr': '3E5ZMVMzm4iZKAid54GeuebvBHvDkfukxh'}]}
        // Compatible Address
```


Transaction 3

```
    {'hash': '761619410d9f2a7fa69336583a0351dd86caf9ed2cee8765132a93caa44f77
de',
     'ver': 1,
     'vin_sz': 1,
     'vout_sz': 2,
     'size': 223,
     'weight': 565,
     'fee': 41800,
     'relayed_by': '0.0.0.0',
     'lock_time': 499981,
     'tx_index': 7827328597578341,
     'double_spend': False,
     'time': 1513622125,
     'block_index': 500000,
     'block_height': 500000,
     'inputs': [{'sequence': 4294967294,
       'witness': '02473044022017768d3be171e0ef48141ea07c66e332d58fd4cd3dd0cb
f69f26feb8a7f32a2d1022025bf48619989d9357dd7e53da40f586512210629256ecbc1f5269
ac72bbd454c012102995773fd0848fb11d4e73007b0f65bf42915f39d18d5ef20f9491739b61
3c38b',
       'script': '',
       'index': 0,
       'prev_out': {'spent': True,
        'script': '0014f2c78dda8d18a90545045942989406bf90966f7f',
        'spending_outpoints': [{'tx_index': 7827328597578341, 'n': 0}],
        'tx_index': 7014719209568386,
        'value': 237293200,
        'addr': 'bc1q7trcmk5drz5s23gyt9pf39qxh7gfvmmlr8kc2n',
        'n': 1,
        'type': 0}}],
     'out': [{'type': 0,
       'spent': True,
       'value': 50000000,
       'spending_outpoints': [{'tx_index': 8461589482919976, 'n': 0}],
       'n': 0,
       'tx_index': 7827328597578341,
       'script': 'a914cefc39c263c83303bcbd25b7be401b04e527c39087',
       'addr': '3LZTH1CiBnETzKvEB1eqfjacgxu5nJ84du'},
      {'type': 0,
       'spent': True,
       'value': 187251400,
       'spending_outpoints': [{'tx_index': 5327395442585504, 'n': 2}],
       'n': 1,
       'tx_index': 7827328597578341,
       'script': '00145f361cf1b9d1a5dbbf8c98b176edb82e30a7d3f1',
       'addr': 'bc1qtumpeude6xjah0uvnzchdmdc9cc205l34e95ej'}]}
       // Segwit Address
```


8.2 Save Cost

Saved Cost (%)	Legacy	Compatible	Segwit

Saved Cost (%)	Legacy	Compatible	Segwit
Legacy	0	24	N/A
Compatible	0	24	24
Segwit	N/A	24	70

8.3 Chain Impact

In [1]:

```
import pandas as pd
import matplotlib.pyplot as plt

df1 = pd.read_csv("averageTransactionPerBlock.csv")
df2 = pd.read_csv("transactionFee.csv")
df3 = pd.read_csv("confirmedTransaction.csv")
df4 = pd.read_csv("BTCPrice.csv")
df1.set_index("Timestamp",inplace=True)
df2.set_index("Timestamp",inplace=True)
df3.set_index("Timestamp",inplace=True)
df4.set_index("Timestamp",inplace=True)

df1 = df1.iloc[1000:1000+200]
df2 = df2.iloc[1000:1000+200]
df3 = df3.iloc[1000:1000+200]
df4 = df4.iloc[1005:1005+200]
```

In [3]:

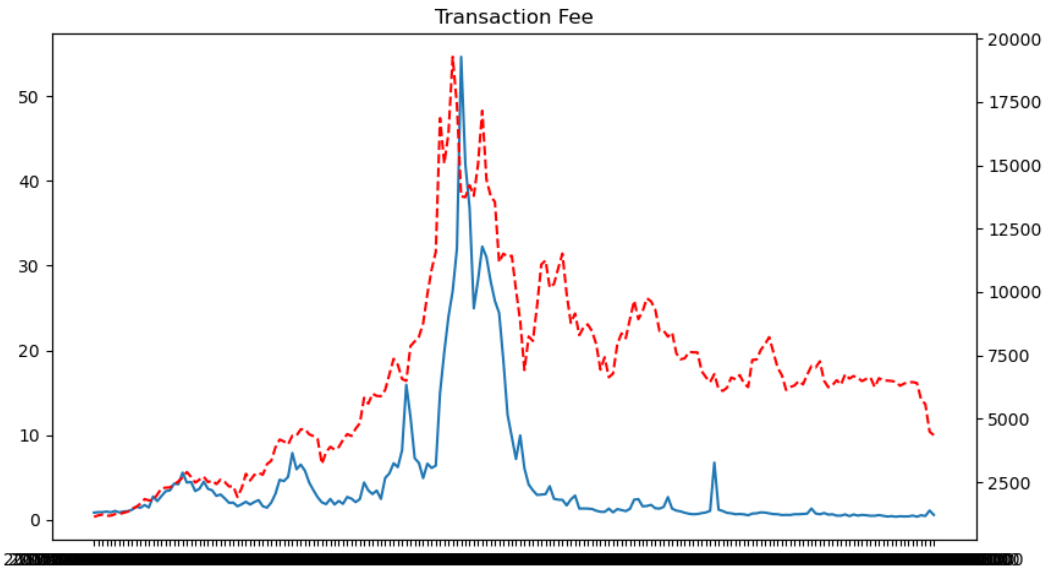
```
%matplotlib notebook
fig, ax1 = plt.subplots()
l1 = ax1.plot(df1.index,df1.values)

ax2 = ax1.twinx()
l2 = ax2.plot(df1.index,df4.values,'r--')

#plt.xticks(X_axis, label)
#plt.legend([l1,l2],['Average Transactions Per Block','Bitcoin Price'])
#plt.ylabel("Option Trading Volume")
plt.title("Average Transactions Per Block")
plt.show()
```

```
<IPython.core.display.Javascript object>
```

In [145]:

```
fig, ax1 = plt.subplots()
l1 = ax1.plot(df2.index,df2.values)

ax2 = ax1.twinx()
l2 = ax2.plot(df1.index,df4.values,'r--')

#plt.xticks(X_axis, label)
#plt.legend([l1,l2],['Average Transactions Per Block','Bitcoin Price'])
#plt.ylabel("Option Trading Volume")
plt.title("Transaction Fee")
plt.show()
```

```
<IPython.core.display.Javascript object>
```



9 Structured Wallet

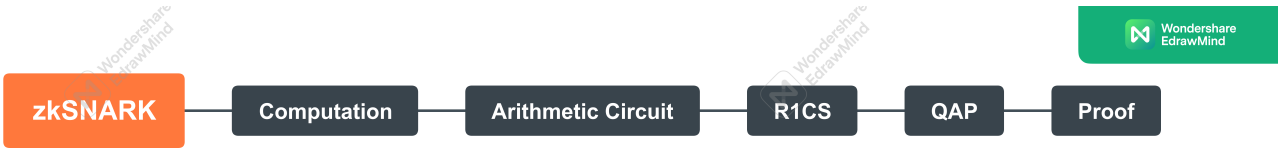
10 Taproot v.s XMR

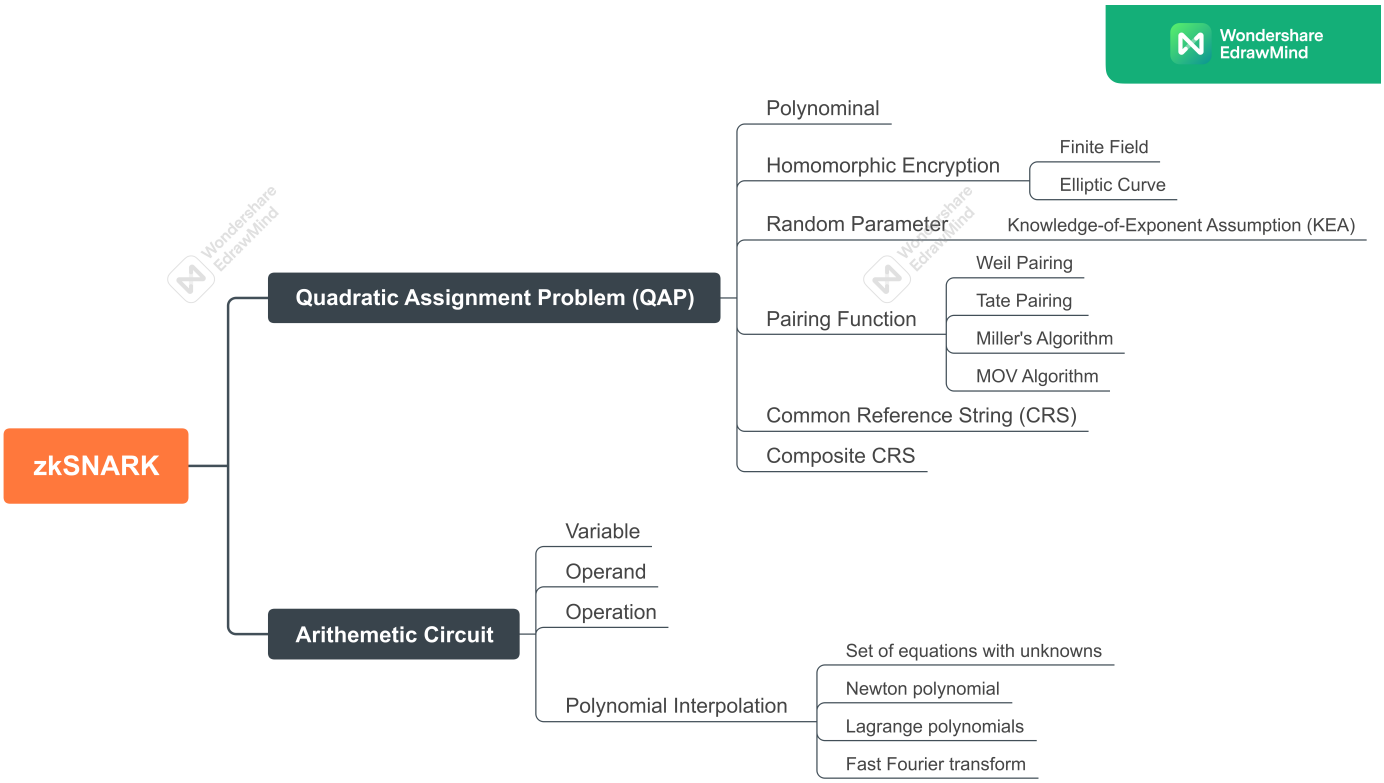
Indeed, im pretty confident any privacy thats built into bitcoin will be optional, so it will be kinda useless. Privacy can't be bolted on. Monero's strength re: privacy (or confidentiality if you want to sound professional and not like a ner-do-well) comes from the fact that its on by default. Its always on. Every transaction has the same level of confidentiality.

In addition to the confidentiality tech of Monero, I'd also recommend some of the other strengths when thinking about it. Monero has a flexible blocksize - there will never be a blocksize / base-layer scaling debate in Monero. Monero is also dedicated to maintaining a decentralized network, recently manifest in the fight for ASIC resistance. Monero also has a perpetual emission - it doesn't need a mythical fee market to support the proof of work that secures the blockchain.

11 zkSNARK

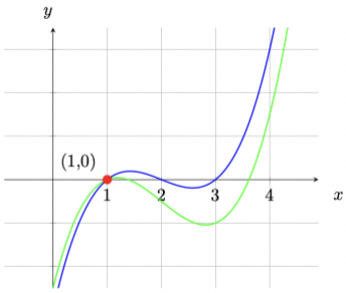
- Succint
- Non-interactive
- Argumented
- Zero Knowledge





12 Polynomial

Polynomials have an advantageous property, namely, if we have two non-equal polynomials of degree at most d, they can intersect at no more than d points. A tiny change produces a dramatically different result. In fact, it is impossible to find two non-equal polynomials, which share a consecutive chunk of a curve



Procedure:

- Verifier chooses a random value for x and evaluates his polynomial locally
- Prover evaluates his polynomial at x and gives the result to the verifier
- Verifier gives x to the prover and asks to evaluate the polynomial in question
- Verifier checks if the local result is equal to the prover's result, and if so then the statement is proven with a high confidence

12.1 Homomorphic Encryption & Encrypted Polynomial

- Encrpytion: $5^3 = 6(mod 7)$
- Multiplication: $6^2 = (5^3)^2 = 1(mod 7)$
- Encrpytion: $5^3 5^2 = 5^5 = 3(mod 7)$

1. **Verifier**

- samples a random value s, i.e., secret
- calculates encryptions of s for all powers *i* in 0, 1, ..., d, i.e.: $E(s^i) = g^{si}$
- evaluates unencrypted target polynomial with $s : t(s)$
- encrypted powers of s are provided to the prover: $E(s^0), E(s^1), \ldots, E(s^d)$

2. **Prover**

- calculates polynomial $h(x) = \frac{p(x)}{t(x)}$
- using encrypted powers $g^{s0}, \ldots, g^{sd}$ and coefficients $c_0, \ldots, c_n$evaluates $E(p(s)) = g^{p(s)} = (g^{sd})^{cd} \cdots (g^{s0})^{c0}$ and similarly $E(h(s)) = g^{h(s)}$
- the resulting g^p and g^h are provided to the verifier

3. **Verifier**

- The last step for the verifier is to checks that $p = t(s)h$ in encrypted space: $g^p = g^{t(s)h}$

12.2 Knowledge-of-Exponent Assumption (KEA, Zero Knowledge)

- Verifier chooses random s, α and provides evaluation for $x = s$ for power 1 and its "shift":$(g^s, g^{\alpha s})$
- Prover applies the coefficient $c : ((g^s)^c, (g^{\alpha s})^c) = (g^{cs}, g^{\alpha cs})$
- Verifier checks: $(g^{cs})^\alpha = g^{\alpha cs}$

12.3 Bilinear Pairings on Elliptic Curves (Zero Knowledge / Non-Interactivity)

See *An introduction to mathematical cryptography* 6.8 and *The arithmetic of elliptic curves* III.4

12.4 CRS (Non-Interactivity)

12.5 Complete Proof

- Setup
  - sample random values s, α
  - calculate encryptions g^{α} and $\left\{g^{s^i}\right\}_{i\in[d]}, \left\{g^{\alpha s^i}\right\}_{i\in\{0,\ldots,d\}}$
  - proving key: $\left(\left\{g^{s^i}\right\}_{i\in[d]}, \left\{g^{\alpha s^i}\right\}_{i\in\{0,\ldots,d\}}\right)$
  - verification key: $\left(g^{\alpha}, g^{t(s)}\right)$
- Proving
  - assign coefficients $\{c_i\}_{i\in\{0,\ldots,d\}}$ (i.e., knowledge), $p(x) = c_d x^d + \cdots + c_1 x^1 + c_0 x^0$
  - calculate polynomial $h(x) = \frac{p(x)}{t(x)}$
  - evaluate encrypted polynomials $g^{p(s)}$ and $g^{h(s)}$ using $\left\{g^{s^i}\right\}_{i\in[d]}$
  - evaluate encrypted shifted polynomial $g^{\alpha p(s)}$ using $\left\{g^{\alpha s^i}\right\}_{i\in\{0,\ldots,d\}}$
  - sample random δ
  - set the randomized proof $\pi = \left(g^{\delta p(s)}, g^{\delta h(s)}, g^{\delta \alpha p(s)}\right)$
- Verification
  - parse proof π as $\left(g^{p}, g^{h}, g^{p'}\right)$
  - check polynomial restriction $e\left(g^{p'}, g\right) = e\left(g^{p}, g^{\alpha}\right)$
  - check polynomial cofactors $e\left(g^{p}, g\right) = e\left(g^{t(s)}, g^{h}\right)$

13 Circuit

After all, the logic operation program is composed of basic operation logic such as the basic addition, subtraction, multiplication and division conditional judgment. The addition, subtraction, multiplication and division polynomials are naturally supported. It seems that conditional judgments cannot be expressed by polynomials. However, it could be understood as a switch (0,1). It can also be expressed in terms of polynomials.

for the following program:

In [ ]:

```
def qeval(a):
    b = a**3
return a + b + 5
```

Target Polynomials: $P(x) = L(x) * R(x) - O(x)$

It can be done by:

$$a * a = sym1 \tag{1}$$
$$a * sym1 = sym2 \tag{2}$$
$$sym2 + a = sym3 \tag{3}$$
$$sym3 + 5 = f \tag{4}$$

For (1),(2):

- L(1)=a, R(1)=a, O(1) = sym1
- L(2)=a, R(2)=sym1, O(2) = sym2

For (3),(4), if we define:

$$L(x) = L1(x) + L2(x) + \ldots + Ld(x)$$
$$R(x) = R1(x) + R2(x) + \ldots + Rd(x)$$
$$O(x) = O1(x) + O2(x) + \ldots + Od(x)$$

So we could do addition:

$$a * a = sym1 \tag{1}$$
$$a * sym1 = sym2 \tag{2}$$
$$(sym2 + a) * 1 = sym3 \tag{3}$$
$$(sym3 + 5) * 1 = f \tag{4}$$

The Constraints are as follows:

$\mathrm{L1(1) = a, Li(1) = 0\{\ d \geq i \geq 2\}, R1(1) = a, Ri(1) = 0\{\ d \geq i \geq 2\}, O1(1) = sym1, Oi(1) = 0\{\ d \geq i \geq}$
$\mathrm{L1(2) = a, Li(2) = 0\{\ d \geq i \geq 2\}, R1(2) = sym1, Ri(2) = 0\{\ d \geq i \geq 2\}, O1(2) = sym2, Oi(2) = 0\{\ d \geq}$
$L1(3) = sym2, L2(3) = a, Li(3) = 0\{d \geq i \geq 3\}, R1(3) = 1, Ri(3) = 0\{d \geq i \geq 2\}, O1(3) = sym3, Oi($
$L1(4) = Sym3, L2(4) = 5, Li(3) = 0\{d \geq i \geq 3\}, R1(3) = 1, Ri(3) = 0\{d \geq i \geq 2\}, O1(3) = f, Oi(3) =$

It can be seen that the L1(x) polynomial is a polynomial of the four points (1,a), (2,a), (3,sym2), (4,sym3), and then it can be constructed by the Lagrange interpolation. Eligible L1(x) polynomials, L2(x), R1(x), R2(x), O1(x), O2(x) can be constructed in the same way, and then P(x) can be constructed. Then you can execute the setup, prove, verifier process to do zero-knowledge proof.

Different inputs (a,f), P(x) are different. We know that the expressions are the same, but each value is different. So we modify the decomposition methods of L(x), R(x), O(x) and add coefficients to each decomposition term, and Ld(x), Rd(x), Od(x) can only be 0/1 Two values, since there are only two values, are similar to circuits, so this polynomial construction process is also called constructing a circuit.

Create Circuit:

$$L(x) = A1 * L1(x) + A2 * L2(x) + \ldots + Am * Lm(x)$$
$$R(x) = B1 * R1(x) + B2 * R2(x) + \ldots + Bm * Rm(x)$$
$$O(x) = C1 * O1(x) + C2 * O2(x) + \ldots + Cm * Om(x)$$

Suppose we have parameters (a, sym1, sym2, sym3, 1, f):

$$r = \begin{bmatrix} a & sym1 & sym2 & sym3 & 1 & f \end{bmatrix}$$

Then,

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

In [4]:

```
# Source: Ethereum Research
def multiply_polys(a, b):
    o = [0] * (len(a) + len(b) - 1)
    for i in range(len(a)):
        for j in range(len(b)):
            o[i + j] += a[i] * b[j]
    return o

# Add two polynomials
def add_polys(a, b, subtract=False):
    o = [0] * max(len(a), len(b))
    for i in range(len(a)):
        o[i] += a[i]
    for i in range(len(b)):
        o[i] += b[i] * (-1 if subtract else 1) # Reuse the function structure for
subtraction
    return o

def subtract_polys(a, b):
    return add_polys(a, b, subtract=True)

# Divide a/b, return quotient and remainder
def div_polys(a, b):
    o = [0] * (len(a) - len(b) + 1)
    remainder = a
    while len(remainder) >= len(b):
        leading_fac = remainder[-1] / b[-1]
        pos = len(remainder) - len(b)
        o[pos] = leading_fac
        remainder = subtract_polys(remainder, multiply_polys(b, [0] * pos +
[leading_fac]))[:-1]
    return o, remainder

# Evaluate a polynomial at a point
def eval_poly(poly, x):
    return sum([poly[i] * x**i for i in range(len(poly))])

# Make a polynomial which is zero at {1, 2 ... total_pts}, except
# for `point_loc` where the value is `height`
def mk_singleton(point_loc, height, total_pts):
    fac = 1
    for i in range(1, total_pts + 1):
        if i != point_loc:
            fac *= point_loc - i
    o = [height * 1.0 / fac]
    for i in range(1, total_pts + 1):
        if i != point_loc:
            o = multiply_polys(o, [-i, 1])
    return o

# Assumes vec[0] = p(1), vec[1] = p(2), etc, tries to find p,
# expresses result as [deg 0 coeff, deg 1 coeff...]
def lagrange_interp(vec):
    o = []
    for i in range(len(vec)):
        o = add_polys(o, mk_singleton(i + 1, vec[i], len(vec)))
    for i in range(len(vec)):
        assert abs(eval_poly(o, i + 1) - vec[i] < 10**-10), \
            (o, eval_poly(o, i + 1), i+1)
```

```
    return o

def transpose(matrix):
    return list(map(list, zip(*matrix)))

# A, B, C = matrices of m vectors of length n, where for each
# 0 <= i < m, we want to satisfy A[i] * B[i] - C[i] = 0
def r1cs_to_qap(A, B, C):
    A, B, C = transpose(A), transpose(B), transpose(C)
    new_A = [lagrange_interp(a) for a in A]
    new_B = [lagrange_interp(b) for b in B]
    new_C = [lagrange_interp(c) for c in C]
    Z = [1]
    for i in range(1, len(A[0]) + 1):
        Z = multiply_polys(Z, [-i, 1])
    return (new_A, new_B, new_C, Z)

def create_solution_polynomials(r, new_A, new_B, new_C):
    Apoly = []
    for rval, a in zip(r, new_A):
        Apoly = add_polys(Apoly, multiply_polys([rval], a))
    Bpoly = []
    for rval, b in zip(r, new_B):
        Bpoly = add_polys(Bpoly, multiply_polys([rval], b))
    Cpoly = []
    for rval, c in zip(r, new_C):
        Cpoly = add_polys(Cpoly, multiply_polys([rval], c))
    o = subtract_polys(multiply_polys(Apoly, Bpoly), Cpoly)
#     for i in range(1, len(new_A[0]) + 1):
#         assert abs(eval_poly(o, i)) < 10**-10, (eval_poly(o, i), i)
#         print("test:",(eval_poly(o, i), i))
    return Apoly, Bpoly, Cpoly, o

def create_divisor_polynomial(sol, Z):
    quot, rem = div_polys(sol, Z)
#     for x in rem:
#         assert abs(x) < 10**-10
    return quot

r = [2, 4, 8, 10, 1, 15]

A = [[1, 0, 0, 0, 0, 0],
     [1, 0, 0, 0, 0, 0],
     [1, 1, 0, 0, 0, 0],
     [0, 0, 1, 0, 1, 0]]
B = [[1, 0, 0, 0, 0, 0],
     [0, 1, 0, 0, 0, 0],
     [0, 0, 0, 0, 1, 0],
     [0, 0, 0, 0, 1, 0]]
C = [[0, 1, 0, 0, 0, 0],
     [0, 0, 1, 0, 0, 0],
     [0, 0, 0, 1, 0, 0],
     [0, 0, 0, 0, 0, 1]]

# r = [1, 3, 35, 9, 27, 30]
# A = [[0, 1, 0, 0, 0, 0],
#      [0, 0, 0, 1, 0, 0],
#      [0, 1, 0, 0, 1, 0],
#      [5, 0, 0, 0, 0, 1]]
# B = [[0, 1, 0, 0, 0, 0],
#      [0, 1, 0, 0, 0, 0],
```

```
#       [1, 0, 0, 0, 0, 0],
#       [1, 0, 0, 0, 0, 0]]
# C = [[0, 0, 0, 1, 0, 0],
#       [0, 0, 0, 0, 1, 0],
#       [0, 0, 0, 0, 0, 1],
#       [0, 0, 1, 0, 0, 0]]

Ap, Bp, Cp, Z = r1cs_to_qap(A, B, C)
print('Ap')
for x in Ap: print(x)
print('Bp')
for x in Bp: print(x)
print('Cp')
for x in Cp: print(x)
print('Z')
print(Z)
Apoly, Bpoly, Cpoly, sol = create_solution_polynomials(r, Ap, Bp, Cp)
print('Apoly')
print(Apoly)
print('Bpoly')
print(Bpoly)
print('Cpoly')
print(Cpoly)
print('Sol')
print(sol)
print('Z cofactor')
print(create_divisor_polynomial(sol, Z))
```

```
Ap
[2.0, -1.833333333333333, 1.0, -0.16666666666666663]
[4.0, -7.0, 3.5, -0.5]
[-1.0, 1.8333333333333333, -1.0, 0.16666666666666666]
[0.0, 0.0, 0.0, 0.0]
[-1.0, 1.8333333333333333, -1.0, 0.16666666666666666]
[0.0, 0.0, 0.0, 0.0]
Bp
[4.0, -4.333333333333333, 1.5, -0.16666666666666666]
[-6.0, 9.5, -4.0, 0.5]
[0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0]
[3.0, -5.166666666666667, 2.5, -0.33333333333333337]
[0.0, 0.0, 0.0, 0.0]
Cp
[0.0, 0.0, 0.0, 0.0]
[4.0, -4.333333333333333, 1.5, -0.16666666666666666]
[-6.0, 9.5, -4.0, 0.5]
[4.0, -7.0, 3.5, -0.5]
[0.0, 0.0, 0.0, 0.0]
[-1.0, 1.8333333333333333, -1.0, 0.16666666666666666]
Z
[24, -50, 35, -10, 1]
Apoly
[11.0, -15.166666666666666, 7.0, -0.8333333333333331]
Bpoly
[-13.0, 24.166666666666668, -10.5, 1.3333333333333335]
Cpoly
[-7.0, 16.16666666666667, -6.0, 0.8333333333333335]
Sol
[-136.0, 446.8333333333333, -567.0277777777778, 353.08333333333337, -
113.86111111111111, 18.083333333333332, -1.111111111111111]
```

```
Z cofactor
[-5.25, 6.972222222222223, -1.111111111111111]
```

14 zkEVM - Hermez

https://www.youtube.com/watch?v=17d5DG6L2nw (https://www.youtube.com/watch?v=17d5DG6L2nw)

Plookup: https://eprint.iacr.org/2020/315.pdf (https://eprint.iacr.org/2020/315.pdf)