

密钥管理

目 录

- 以太坊
- NEO
- 比特币
- 莱特币
- 量子
- 比特股

- **关于私钥存储:**

- 加密存储

- 加密密钥: 根据【登录密码】计算得到

- 加密算法: 对称加密算法, 一般为AES

以太坊

公私钥生成与存储

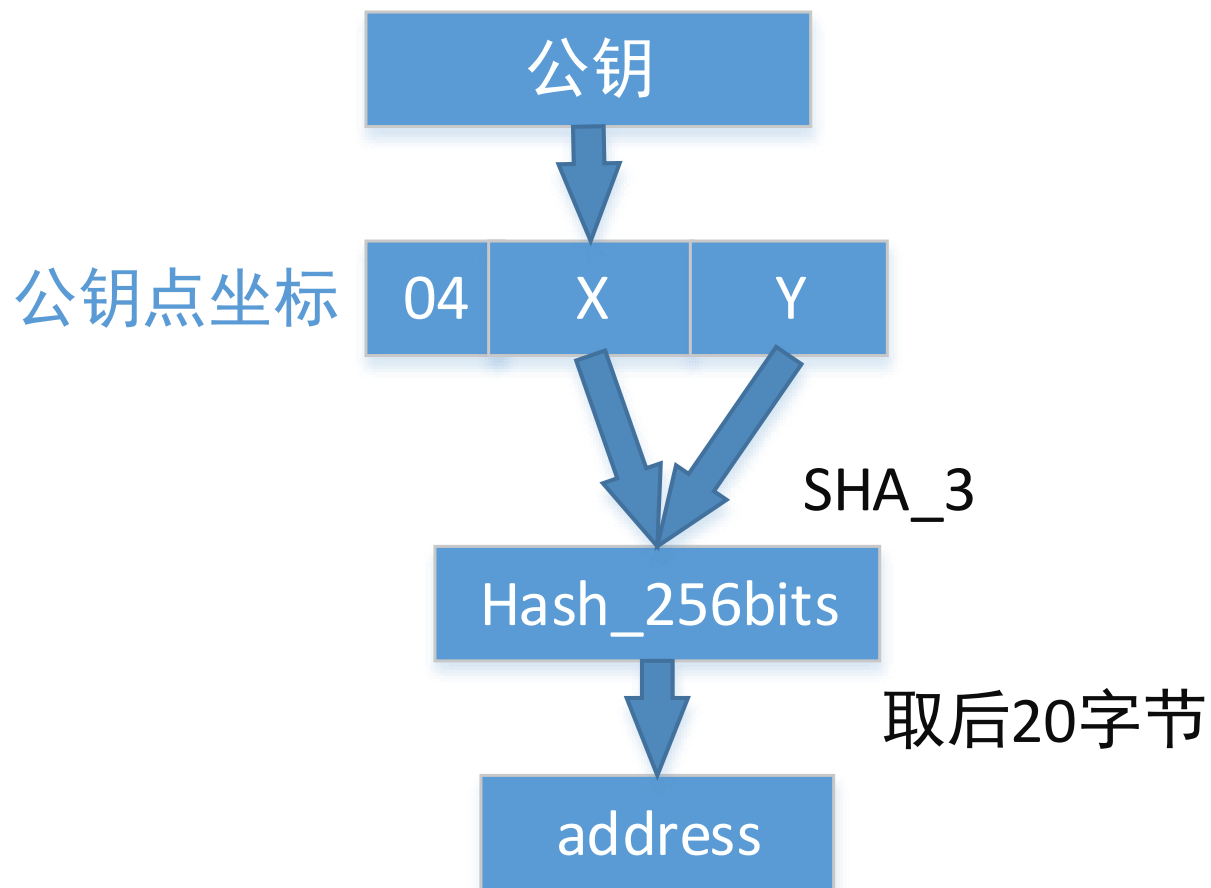
- 以太坊使用ECC算法的secp256k1曲线。
- 首先使用随机数发生器生成一个私钥（32字节）。
- 私钥经过SECP256K1算法处理生成了公钥。
- 公钥计算出钱包地址（不可逆运算）。并存储在钱包文件中。
- 私钥加密存储在钱包文件中。加密算法为128bit分组的AES算法，加密模式为CTR模式。加密密钥根据用户的口令，使用密钥生成函数生成。

钱包结构

```
008aeeda4d805471df9b2a5b0f38a0c3bcba786b 地址
{
  "crypto" : {
    "cipher" : "aes-128-ctr", 加密私钥的算法
    "cipherparams" : { 加密需要使用的参数
      "iv" : "83dbcc02d8ccb40e466191a123791e0e"
    },
    "ciphertext" :  私钥加密后的密文
    "d172bf743a674da9cdad04534d56926ef8358534d458ffccd4e6ad2fbde479c",
    "kdf" : "scrypt", 密钥生成函数使用的算法
    "kdfparams" : {  scrypt函数使用的参数
      "dklen" : 32,
      "n" : 262144,
      "r" : 1,
      "p" : 8,
      "salt" :
    "ab0c7876052600dd703518d6fc3fe8984592145b591fc8fb5c6d43190334ba19"
    },
    "mac" : 用于校验正确性
    "2103ac29920d71da29f15d75b4a16dbe95cfd7ff8faea1056c33131d846e3097"
  },
  "id" : "3198bc9c-6672-5ab3-d995-4942343ae5b6",
  "version" : 3
}
```

- **cipher**: 对称算法的名称, 该算法用于加密私钥
- **cipherparams**: 上述 *cipher* 算法需要的参数;
- **ciphertext**: 以太坊私钥使用上述 *cipher* 算法进行加密的结果
- **kdf**: [密钥生成函数](#), 用密码生成加密 keystore 文件的密钥
- **kdfparams**: 上述 kdf 算法需要的参数;
- **Mac**: 用于验证密码的正确性。

以太坊地址

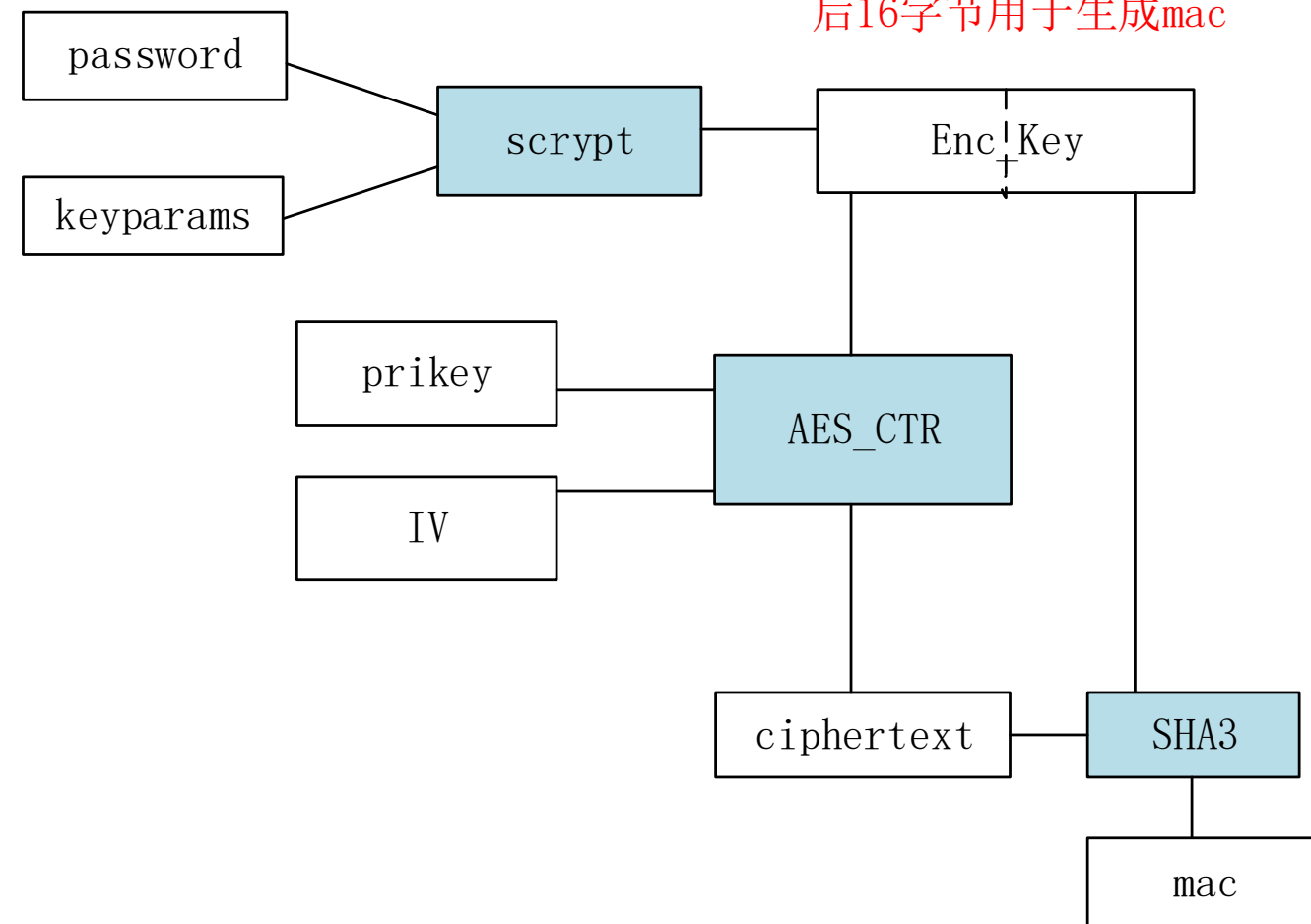


以太坊地址示例：

0x2Dd94041d9c2d4b1bf553A726B8390E50882FceC

私钥加密过程

32字节
前16字节作为加密密钥
后16字节用于生成mac



- 首先，根据密码和参数keyparams经过scrypt函数来计算**加密密钥**。
- 然后，用计算出的**加密密钥前16字节**和参数IV对prikey使用AES_CTR进行加密，得到私钥的密文ciphertext。
- 最后，将**加密密钥后16字节**和 **ciphertext** 密文一起用SHA3计算得到**mac**。

计算加密密钥

- 加密私钥所使用的密钥是根据password使用scrypt算法（一种密钥派生函数）计算出。
- scrypt算法使用的参数如下图所示：

```
"kdfparams" : {  
    "dklen" : 32,  
    "n" : 262144,  
    "r" : 1,  
    "p" : 8,  
    "salt" :  
    "ab0c7876052600dd703518d6fc3fe8984592145b591fc8fb5c6d43190334ba19"  
},
```

scrypt函数使用的参数
生成密钥长度

在第一次存储时随机生成

scrypt算法(RFC 7914)的过程

Input:

P Passphrase, an octet string.
S Salt, an octet string.
N CPU/Memory cost parameter, must be larger than 1,
 a power of 2, and less than $2^{(128 * r / 8)}$.
r Block size parameter.
p Parallelization parameter, a positive integer
 less than or equal to $((2^{32}-1) * hLen) / MLen$
 where hLen is 32 and MLen is $128 * r$.
dkLen Intended output length in octets of the derived
 key; a positive integer less than or equal to
 $(2^{32} - 1) * hLen$ where hLen is 32.

Output:

DK Derived key, of length dkLen octets.

Steps:

1. Initialize an array B consisting of p blocks of $128 * r$ octets each:
 $B[0] || B[1] || \dots || B[p - 1] =$
 PBKDF2-HMAC-SHA256 (P, S, 1, $p * 128 * r$)
2. for i = 0 to p - 1 do
 $B[i] = \text{scryptROMix}(r, B[i], N)$
 end for
3. $DK = \text{PBKDF2-HMAC-SHA256}(P, B[0] || B[1] || \dots || B[p - 1],$
 1, dkLen)

注：scrypt算法所需计算时间长，而且占用的内存也多，使得并行计算多个摘要异常困难，因此暴力攻击更加困难。

scryptROMix：是一个块混淆函数

PBKDF：Password-Based Key Derivation Function, 基于口令的密钥分发函数，详细过程见PKCS5。

PBKDF2 算法(RFC 2898)的描述和主要过程

$DK = \text{PBKDF2}(\text{PRF}, \text{Password}, \text{Salt}, c, \text{dkLen})$

PRF是一个伪随机函数，例如HASH_HMAC函数，它会输出长度为hLen的结果。

Password是用来生成密钥的原文密码。

Salt是一个加密用的盐值。

c是进行重复计算的次数。

至少1000次

dkLen是期望得到的密钥的长度。

DK是最后产生的密钥。

DK的值由一个以上的block拼接而成。block的数量是 dkLen/hLen 的值。

$$DK = T1 || T2 || \dots || T_{\text{dklen}/\text{hlen}}$$

而每个block则通过函数F得到：

$$\begin{aligned} T_i &= F(\text{Password}, \text{Salt}, c, i) \\ &= U1 \oplus U2 \oplus \dots \oplus U_c \end{aligned}$$

其中：

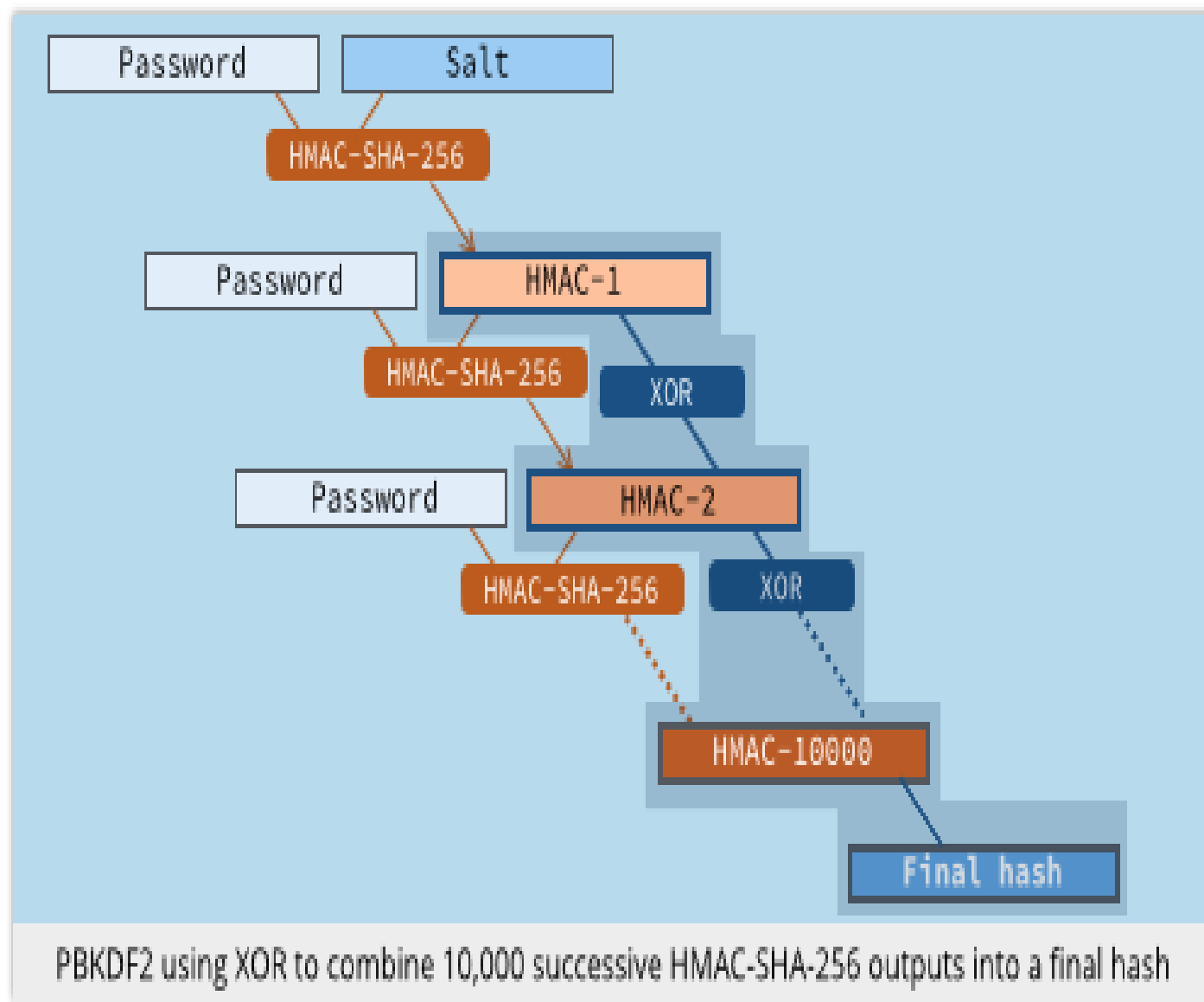
$$U1 = \text{PRF}(\text{Password}, \text{Salt} || \text{INT_32_BE}(i))$$

$$U2 = \text{PRF}(\text{Password}, U1)$$

...

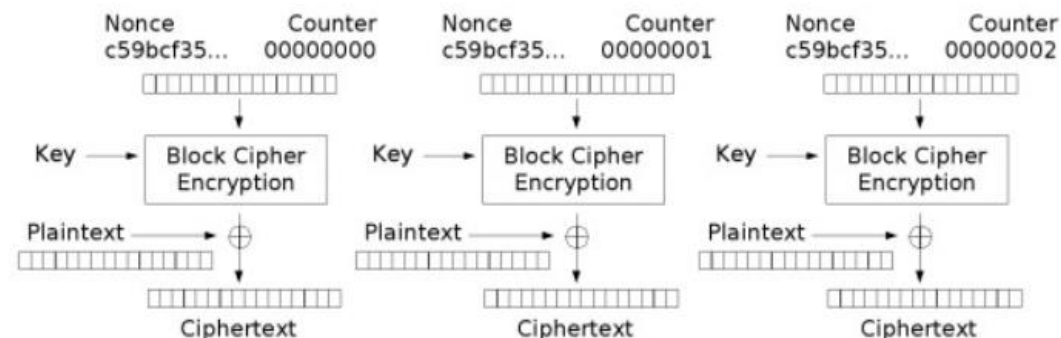
$$U_c = \text{PRF}(\text{Password}, U_{c-1})$$

函数F详解图：

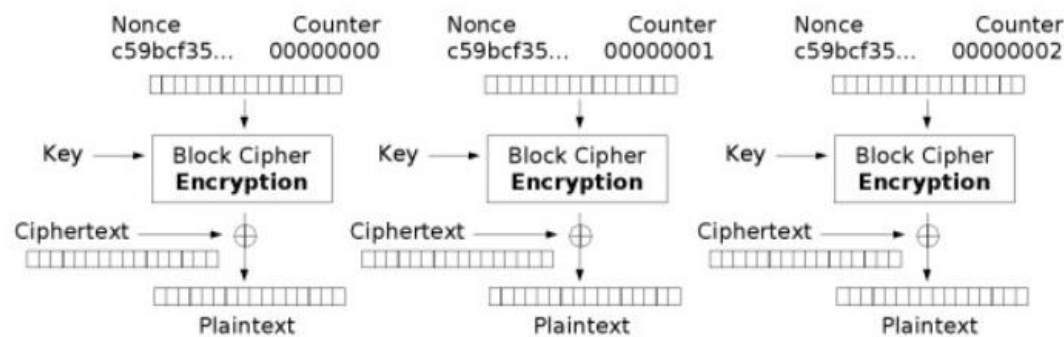


私钥加密

- 私钥以加密方式存储在钱包文件中。加密使用算法为 AES_128_CTR.
- **KEY: KDF (scrypt算法) 计算出来的密钥（取前16字节）。**
- **分组长度为128 bit.**
- **CTR模式需要用到计数器，第一个计数器即为钱包文件的IV（128 bit，第一次存储时随机生成）。**



Counter (CTR) mode encryption



Counter (CTR) mode decryption

CTR加密原理：用密钥对输入的计数器加密，然后同明文异或得到密文。解密原理：用密钥对输入计数器加密，然后同密文异或得到明文。

CTR不需要Padding，而且采用了流密钥方式加解密，适合于并行运算，CTR涉及参量：Nounce随机数、Counter计数器和密钥。Nounce随机数和Counter计数器整体可看作计数器，因为只要算法约定好，就可以回避掉串行化运算。

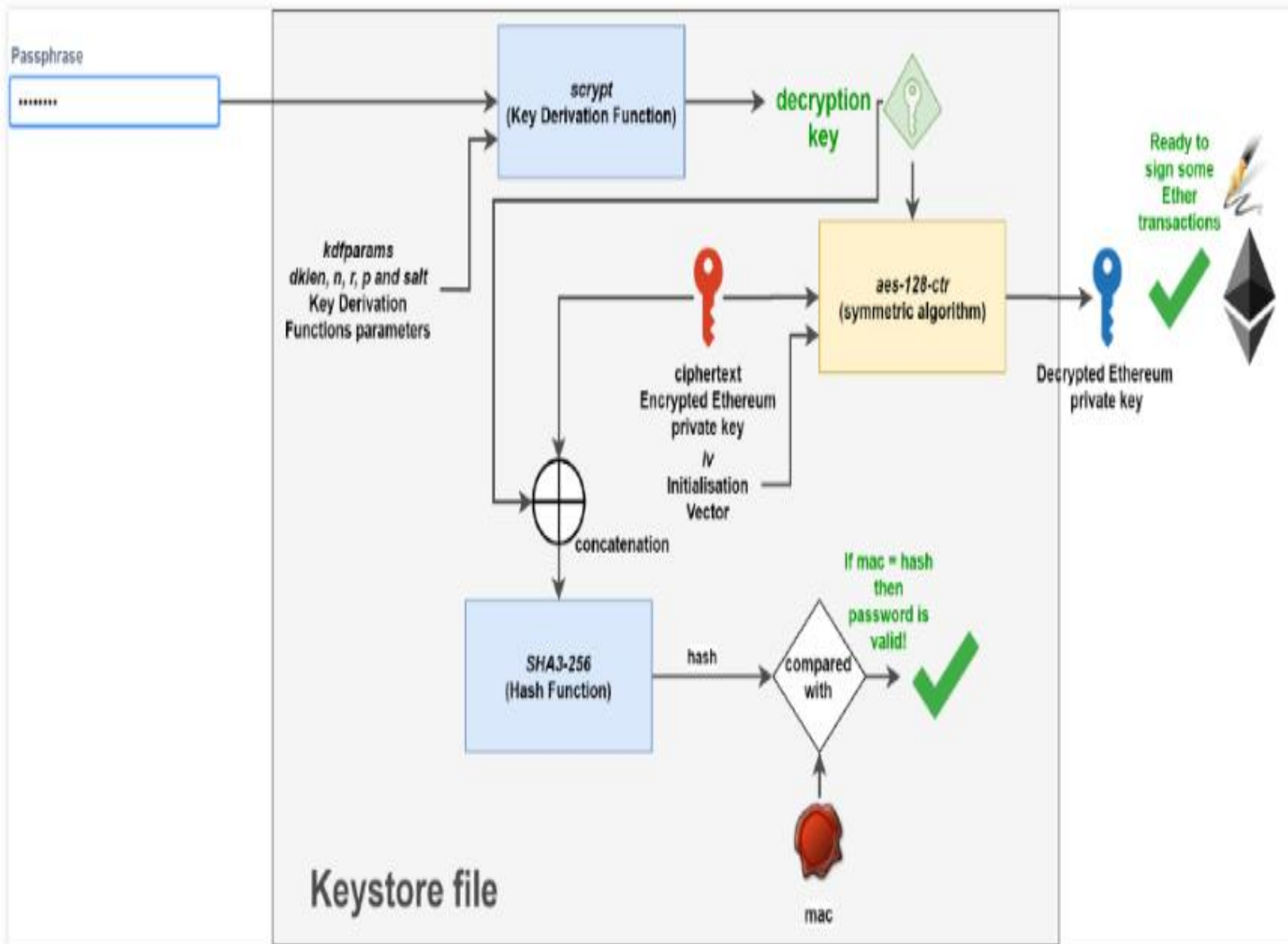
校验值: *MAC*

- MAC值用于校验password与cipher的匹配性，以保证解密出的私钥是正确的。
- MAC的计算方式：

```
mac := crypto.Keccak256(derivedKey[16:32], cipherText)
```

- **Derivekey:** password通过KDF计算出来的密钥（取后16字节）。
- **cipherText:** 钱包文件的cipherText，私钥的密文。

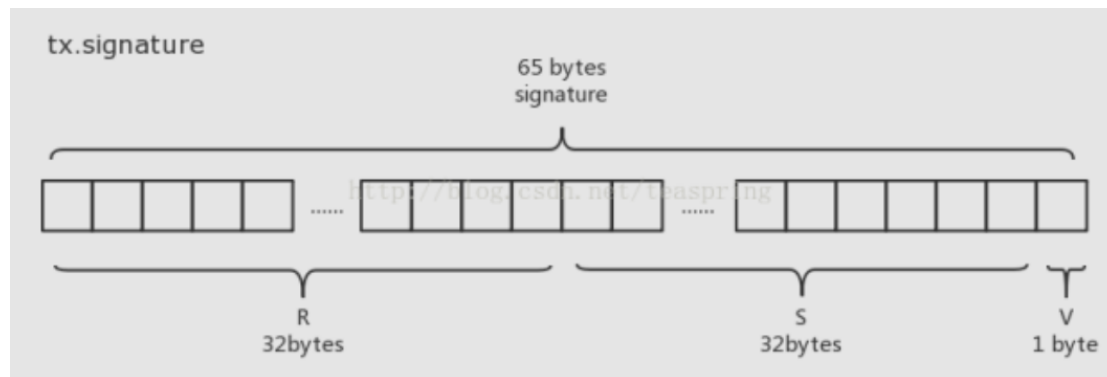
私钥解密过程



- 首先，输入密码，这个密码作为 **kdf** 密钥生成函数的输入，来计算**解密密钥**。
- 然后，计算出的**解密密钥**和 **ciphertext** 密文连接做SHA_3，和 **mac** 比较来确保密码是正确的。
- 最后，用**解密密钥**对 **ciphertext** 密文解密。

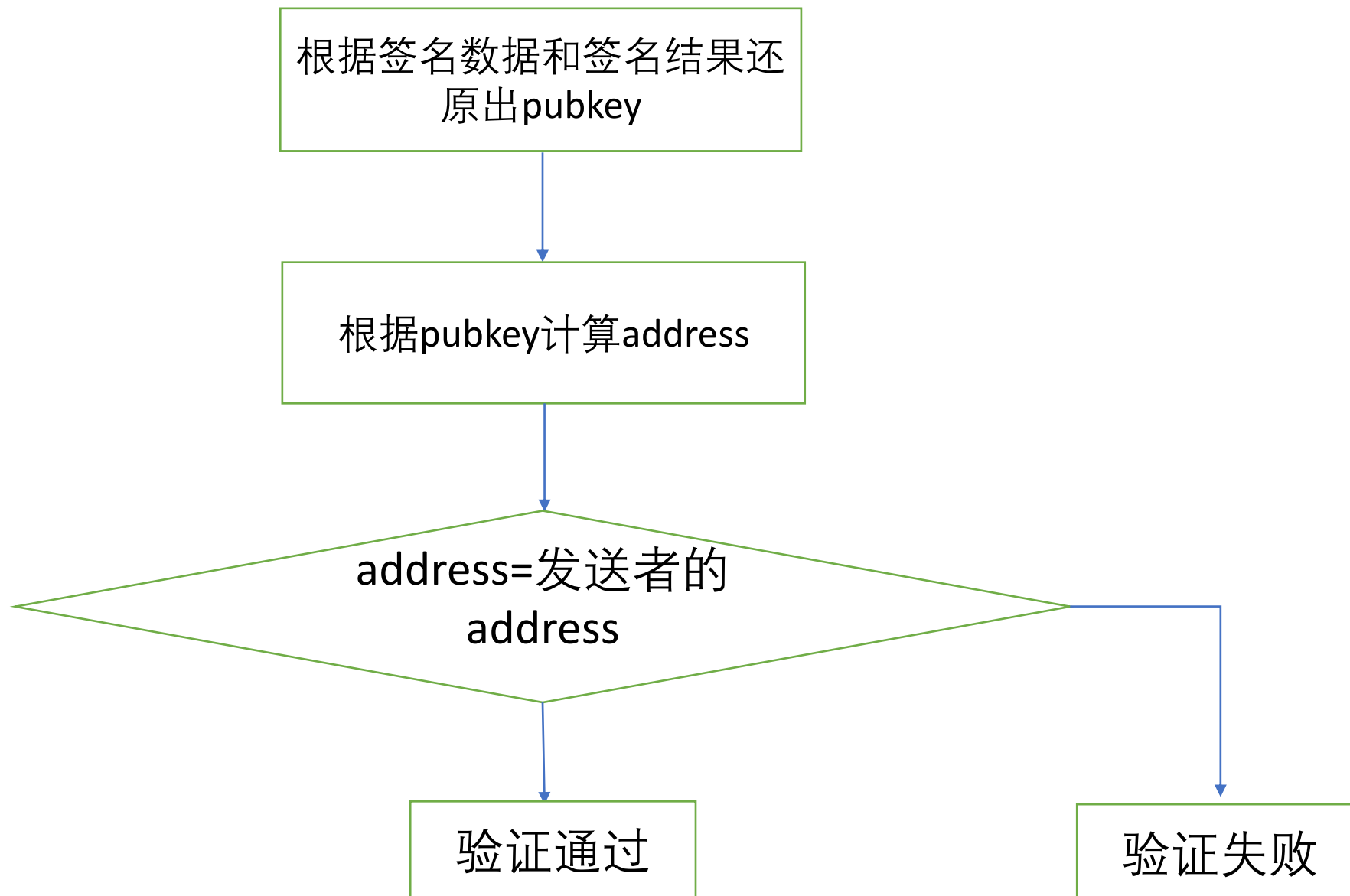
交易签名

- 以太坊的交易使用私钥来签名。因此发起交易之前必须要进行对钱包的解锁操作，得到私钥。
- 签名结果的结构如下：



- 其中前64字节为签名值。最后一字节为标志位（有效标志位为0,1），用于签名验证时还原公钥。签名算法步骤及标志位的标志方法如下：
- 输入：椭圆曲线参数(q, FR, S, a, b, P, n, h)；私钥 d ；消息的哈希 $e=\text{hash}(m)$
- 输出：签名结果(r, s, v)

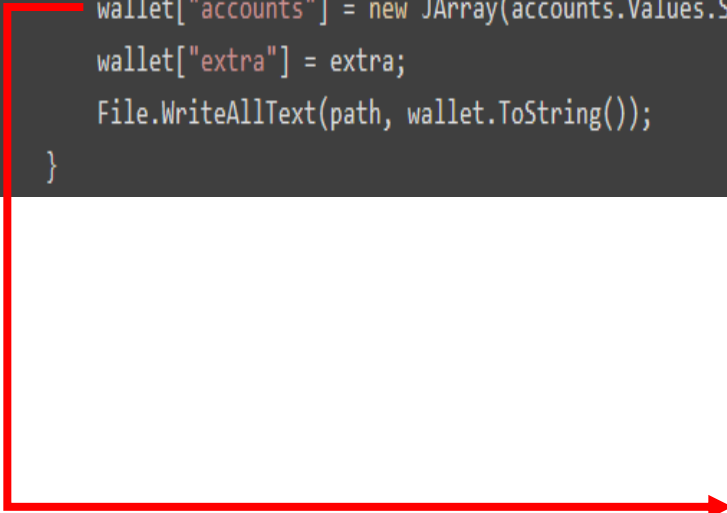
交易验证



NEO

钱包结构

```
public void Save()
{
    JObject wallet = new JObject();
    wallet["name"] = name; //钱包名
    wallet["version"] = version.ToString(); //钱包版本
    wallet["script"] = Script.ToJson(); //script加密参数
    wallet["accounts"] = new JArray(accounts.Values.Select(p => p.ToJson())); //账户转json
    wallet["extra"] = extra;
    File.WriteAllText(path, wallet.ToString());
}
```



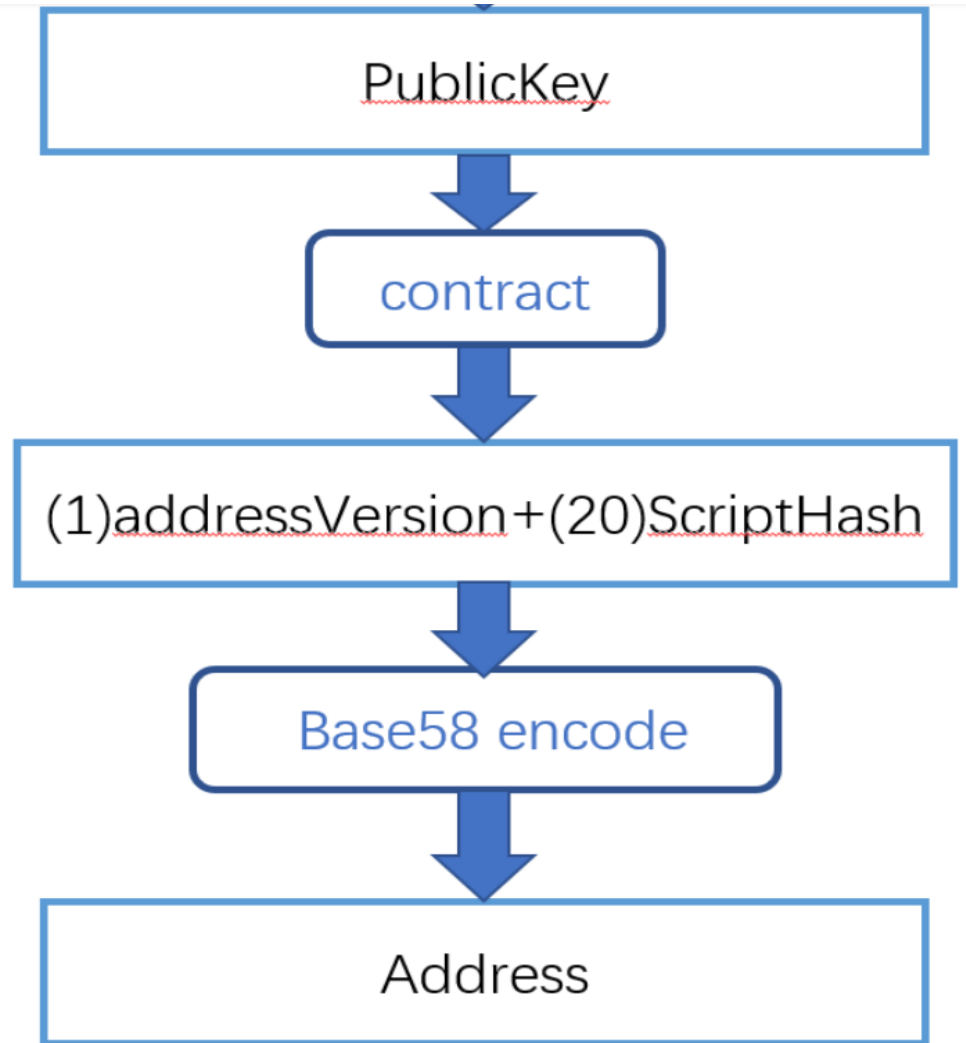
```
public JObject ToJson()
{
    JObject account = new JObject();
    account["address"] = Wallet.ToAddress(ScriptHash); //地址
    account["label"] = Label; //账户标签
    account["isDefault"] = IsDefault;
    account["lock"] = Lock;
    account["key"] = nep2key; //nep2key
    account["contract"] = ((NEP6Contract)Contract)?.ToJson(); //账户合约
    account["extra"] = Extra; //补充信息
    return account;
}
```

密钥管理

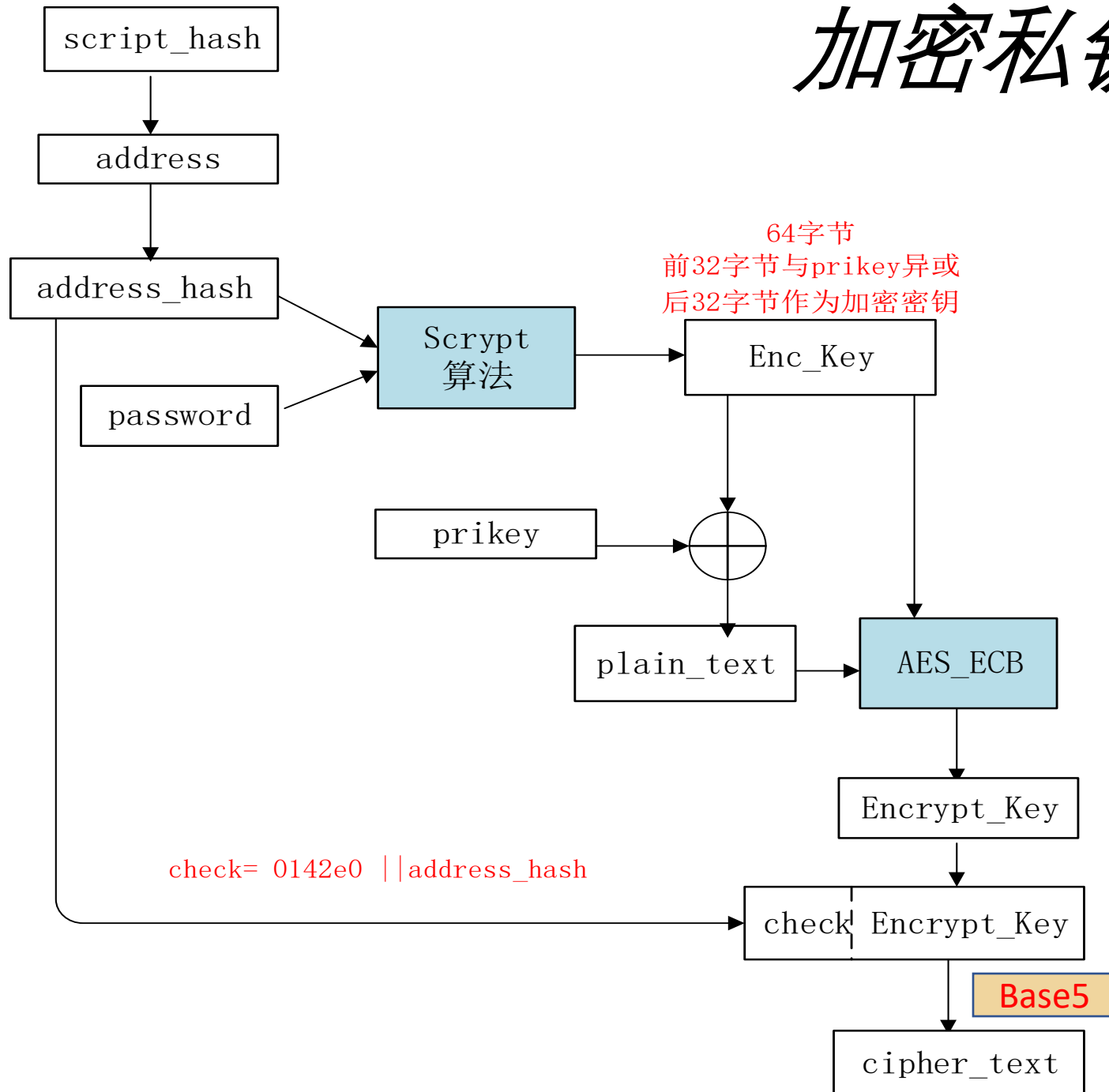
- 密钥生成
- NEO选取的是椭圆曲线的secp256r1曲线。
- NEO的私钥是随机生成的长度为32的字节数组；
- 根据生成的私钥生成公钥；
- 然后再根据公钥来计算地址。

公钥生成地址

- NEO的地址也是根据pubkey计算出来的，但是计算的过程与比特币、以太坊等有很大的不同。
- NEO在创建账户的时候会根据公钥创建一个鉴权合约（主要是将公钥写到合约中），返回合约的脚本，地址就是根据这个脚本的哈希值得来的。在生成地址的时候，会传入这个合约脚本的哈希值。

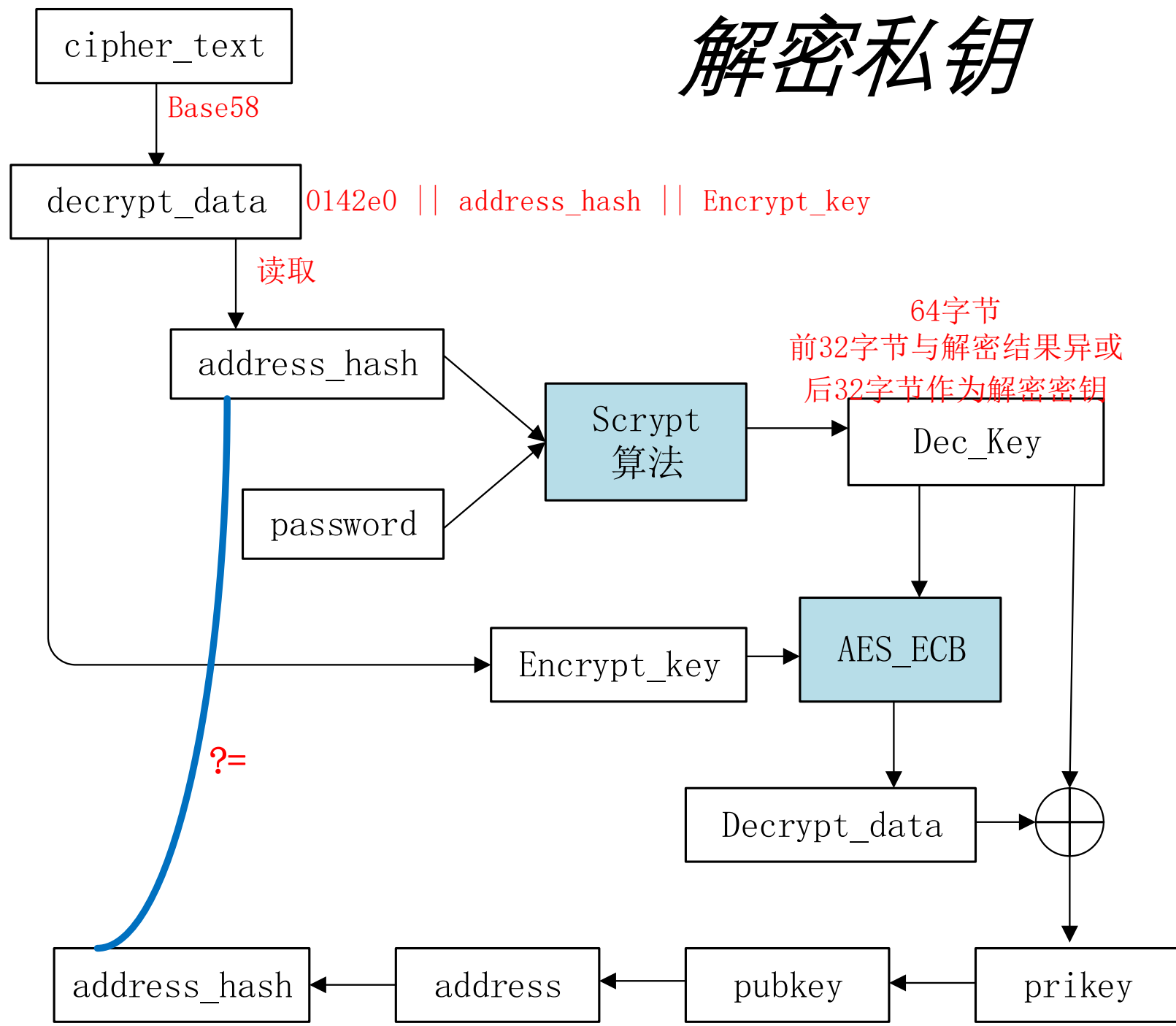


加密私钥



- 私钥经过加密后存储在文件中。
- 加密私钥的密钥通过钱包的口令计算生成
- 存储address_hash是为了做验证。与以太坊的MAC作用类似。

解密私钥



- 相应的，在使用私钥进行签名的时候，需要将文件中的私钥解密得到私钥。解密时也需要用到钱包的密码。
- 同时，会对解密结果的正确性做验证：验证存储的 `address_hash` 与计算出来的 `address_hash` 是否一致。

交易签名与验证

- NEO的交易为UTXO交易。
- NEO交易的签名结果是 (r,s) 两部分，长度为64字节。
- 交易验证：使用公钥对签名结果进行验证。

比特股

BTS

钱包文件结构

- {
- chain_id_type **chain_id**; /** Chain ID this wallet is used with */
- account_multi_index_type **my_accounts**; //account_object, 具体的账户信息
- vector<char> **cipher_keys**; //加密的私钥信息 /** encrypted keys */
- map<account_id_type, set<public_key_type>> **extra_keys**; //导入钱包的账户id 与 对应的公钥
- map<string, vector<string>> **pending_account_registrations**; //未注册成功的账户信息
- map<string, string> **pending_witness_registrations**; //未注册成功的证人节点信息
- //隐私交易相关信息
- key_label_index_type **labeled_keys**;
- blind_receipt_index_type **blind_receipts**;
- string **ws_server** = "ws://localhost:8090";
- string **ws_user**;
- string **ws_password**;
- }
- 钱包文件实例: [wallet_example.json](#)

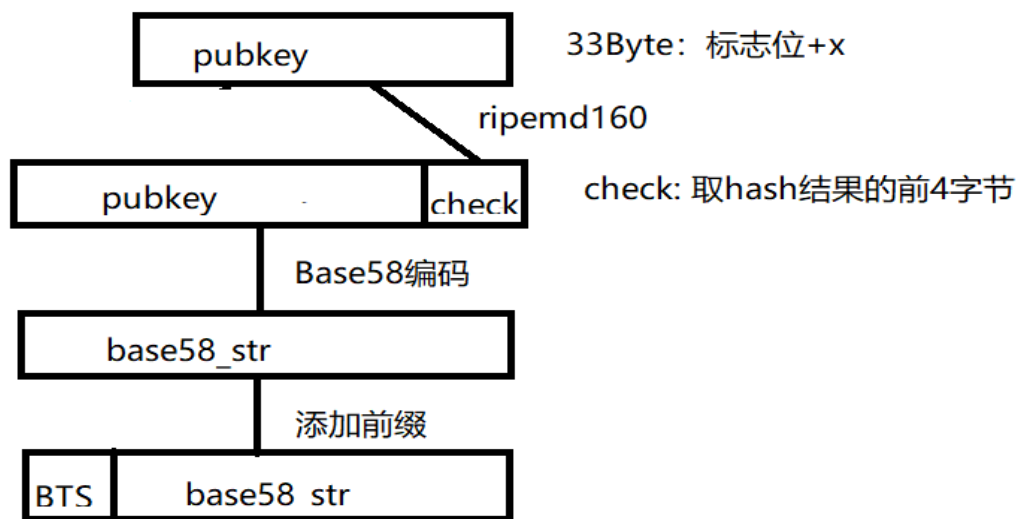
密钥生成

- 创建账户时，根据BrainKey生成账户的公私钥对。账户有三对公私钥对。
- ■ owner_prikey/ owner_pubkey: 控制账户，如备份、升级、导入账户等
- ■ active_prikey/ active_pubkey: 控制资金，如交易
- ■ memo_prikey/ memo_pubkey: 保证交易时所填备注信息的隐私性
- 私钥的具体计算方法如下：
- $\text{owner_prikey} = \text{SHA_256}(\text{SHA_512}(\text{normailize}(\text{barin_key}) || " " || 0))$;
- $\text{active_prikey} = \text{SHA_256}(\text{SHA_512}(\text{key_to_wif}(\text{owner_privkey}) || " " || \text{active_index}))$;
- $\text{memo_prikey} = \text{SHA_256}(\text{SHA_512}(\text{key_to_wif}(\text{active_prikey}) || " " || \text{memo_index}))$;
- active index/ memo_index的存在是为了保证active/memo的唯一性。active_key和memo_key会在交易时使用，因此要保证唯一性。owner_prikey直接与barin_key有关，因此可能相同。由于active_key是由owner_key和active_index一起做hash计算出来的，对于相同的owner_key，使用不同的active_index，计算出的active_key会看起来有很大差别。因此，不需要担心，使用相同的barin_key会导致密钥信息泄露。

公钥

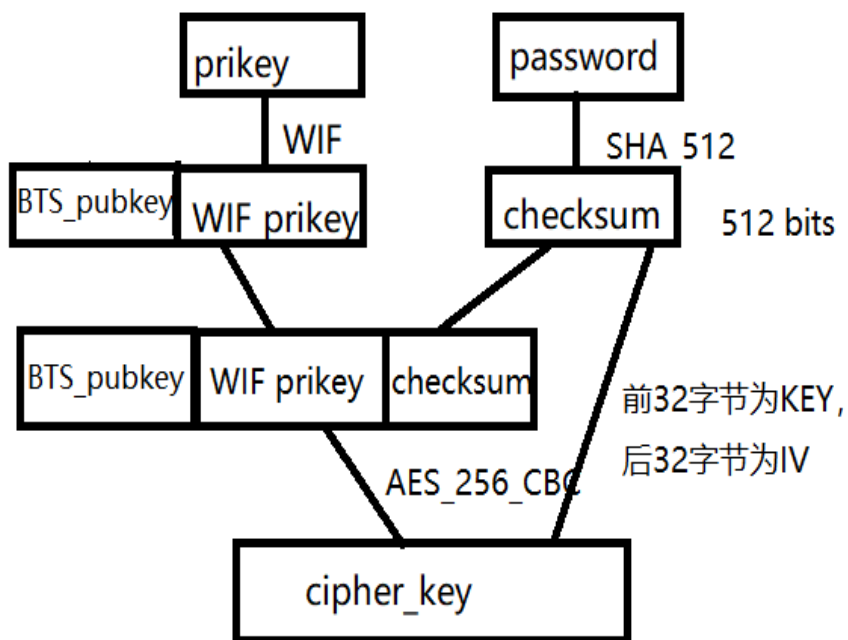
- pubkey由对应的prikey根据ECC算法计算得出。用户的公钥信息里只存储了pubkey的x部分。在实际使用时可以根据x求出相应的y。
- 通过钱包文件可看到的公钥格式如下：

- 公钥到 "BTS6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW5CV"



私钥存储

- 用户的私钥加密存储在钱包文件的cipher_keys选项中。具体加密方式如下：



- 私钥加密存储时不止加密了私钥，同时还加密了与私钥对应的公钥，以及加密用的密码。其中公私钥以map形式存储。明文的存储结构如下图：

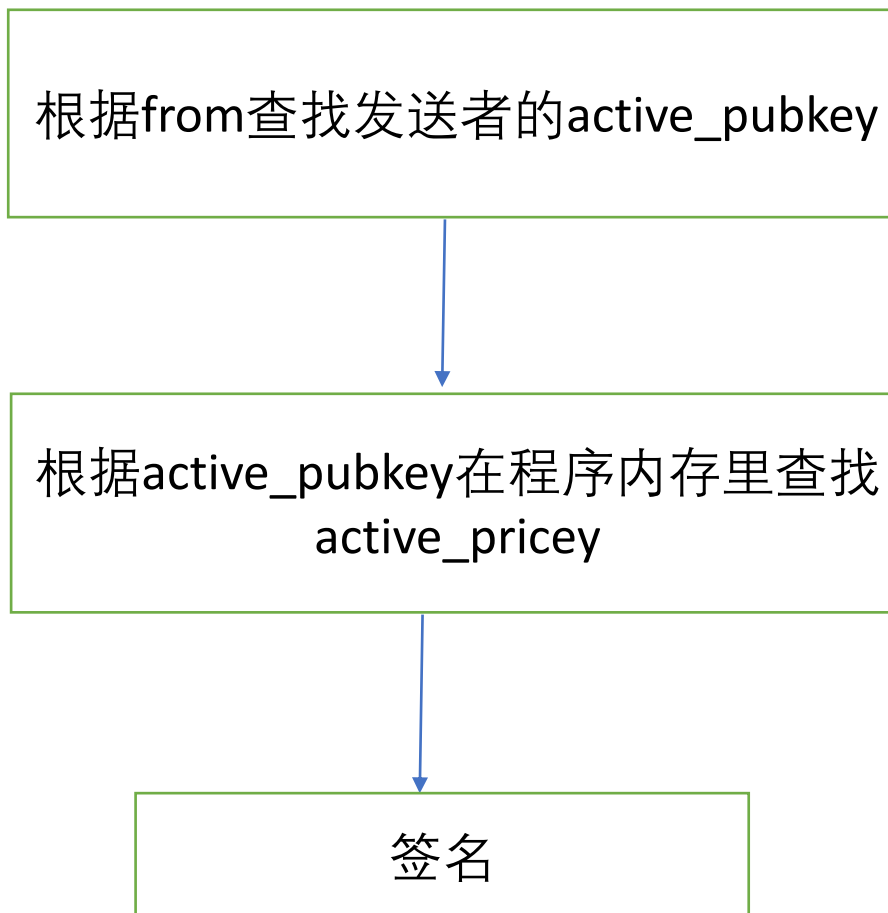
struct plain_keys

```
{
    map<public_key_type, string> keys;
    fc::sha512 checksum;
};
```

Keys只包括active_key和memo_key, 对于owner_key, 只显示owner_pubkey, 不存储owner_prikey

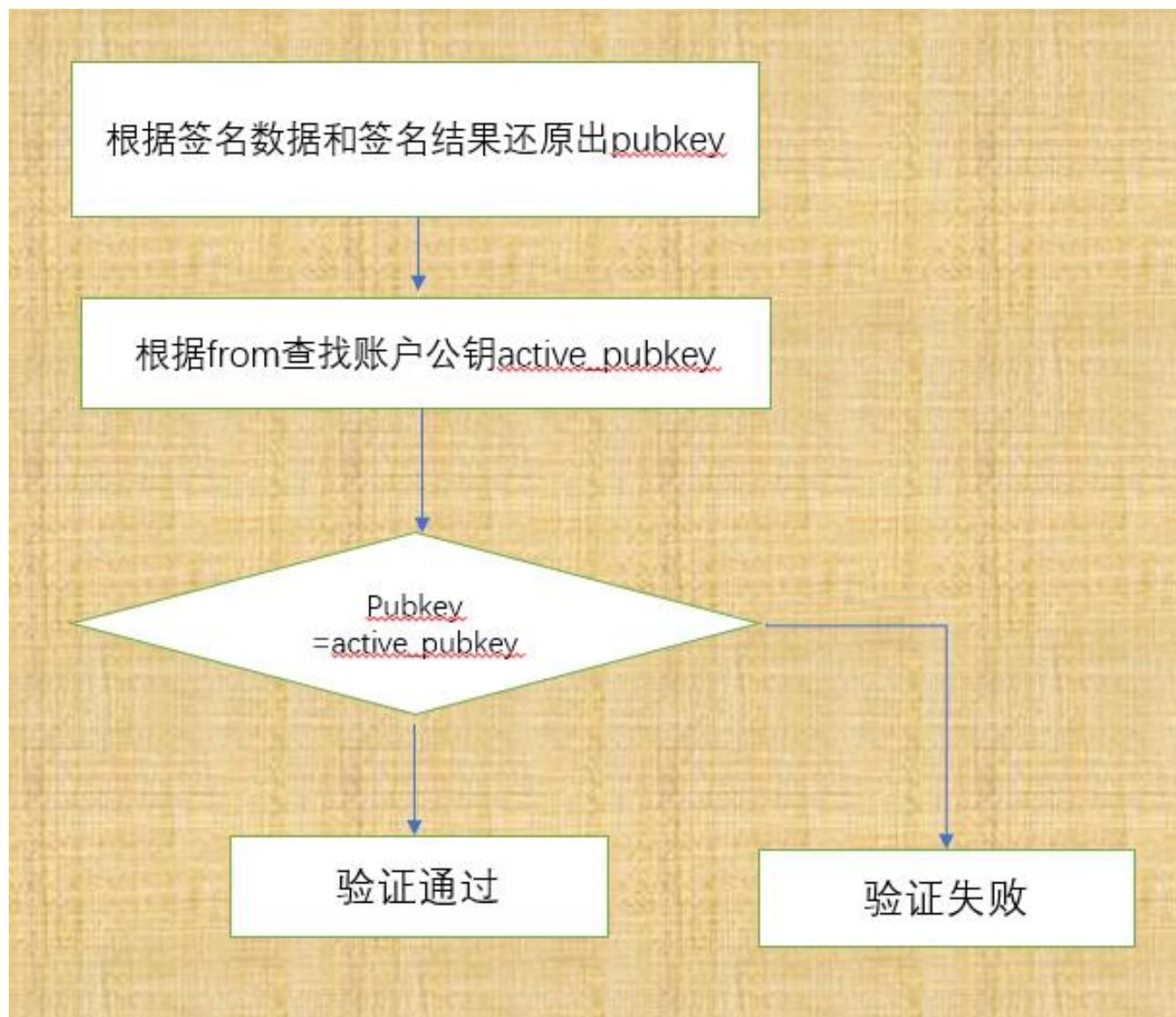
交易签名

- 交易中使用
active_prikey进行签名，
使用active_prikey进行
签名的具体操作流程
如下：



交易签名验证

- 验证交易签名的有效性时，没有使用验签函数，而是根据签名结果和签名数据将公钥恢复出来，再与交易发送者的公钥进行比较。



交易备注信息加密

- 备注信息加密使用 AES_256_CBC 算法
- 加密密钥和IV根据发送者和接收者的 memo_key 通过 ECDH 算法计算。

ECDH（椭圆曲线密钥交换协议）算法：

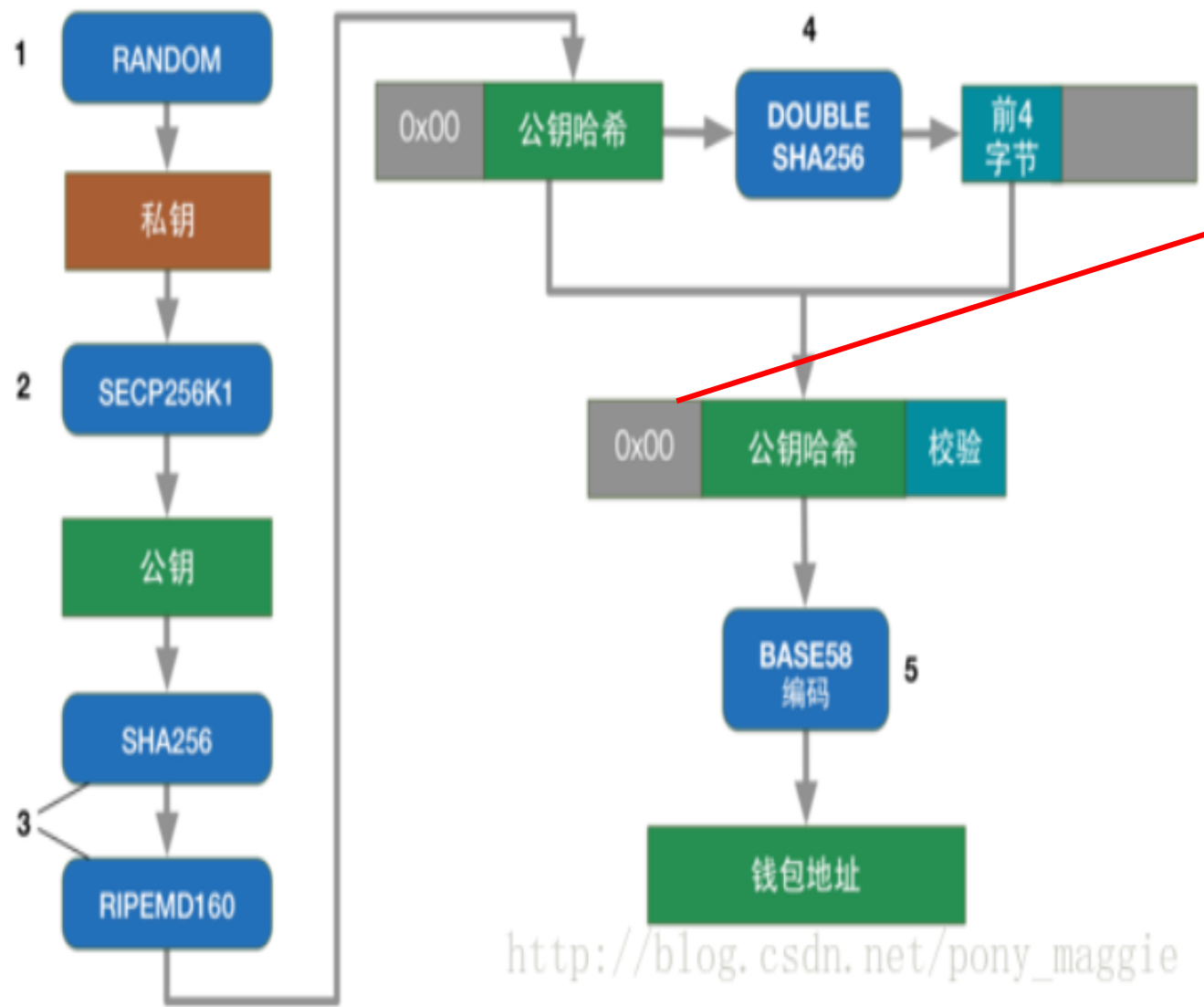
- 1) Alice生成随机整数 a ，计算 $A=a*G$ 。 #生成Alice公私钥
- 2) Bob生成随机整数 b ，计算 $B=b*G$ 。 #生产Bob公私钥
- 3) Alice将 A 传递给Bob。
- 4) Bob将 B 传递给Alice。
- 5) Bob收到Alice传递的 A ，计算 $Q=b*A$ #Bob通过自己的私钥和Alice的公钥得到对称密钥 Q
- 6) Alice收到Bob传递的 B ，计算 $Q'=a*B$ #Alice通过自己的私钥和Bob的公钥得到对称密钥 Q'

Alice、Bob双方即得

$Q=b*A=b*(a*G)=(b*a)*G=(a*b)*G=a*(b*G)=a*B=Q'$ (交换律和结合律)，即双方得到一致的密钥 Q 。

比特币

密钥、地址生成



比特币地址以“1”开头：在Base58编码时在hash前加了前缀0x00

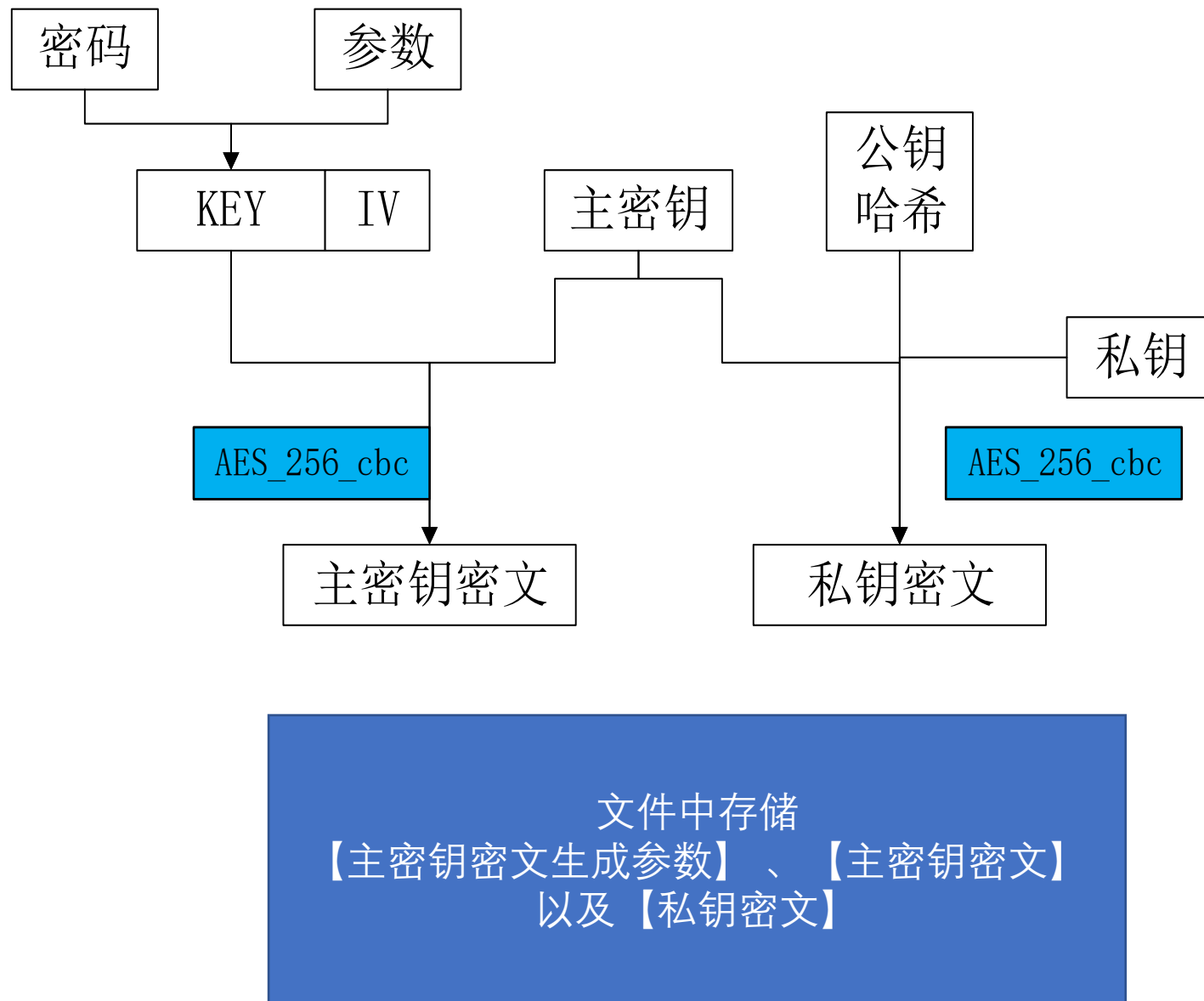
表4-1 Base58Check版本前缀和编码后的结果

种类	版本前缀 (hex)	Base58格式
Bitcoin Address	0x00	1
Pay-to-Script-Hash Address	0x05	3
Bitcoin Testnet Address	0x6F	m or n
Private Key WIF	0x80	5, K or L
BIP38 Encrypted Private Key	0x0142	6P
BIP32 Extended Public Key	0x0488B21E	xpub

密钥保存

- 账户私钥加密存储在钱包文件中。
- 加密方案采用NIST建立的BIP0038方案。
- 加密算法：AES_256_CBC
- 加密密钥：根据用户的password计算生成

私钥加密存储过程



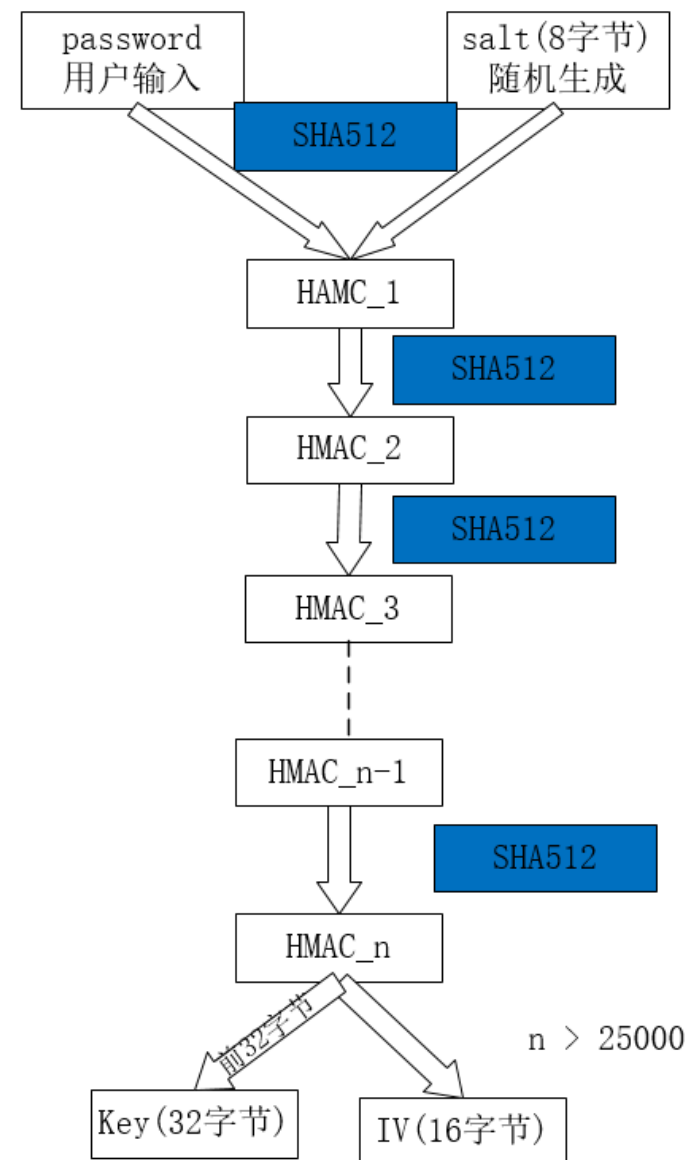
主密钥：32字节，随机生成，用于加密私钥

主密钥密文：主密钥加密后存储在钱包文件中，加密主密钥的算法为AES_256_CBC，加密密钥和IV由【密码】和【主密钥密文生成参数】计算得到

私钥密文：私钥的加密结果，加密算法为AES_256_CBC，加密密钥为【主密钥】、【IV为公钥哈希】

加密主密钥的KEY和IV的计算

- 加密密钥根据用户的【password】和【主密钥密文生成参数】计算生成
- Key: 32字节
- IV : 16字节
- 迭代次数: $n > 25000$, 是为了增加暴力破解的难度。在1.86 GHz Pentium M环境下, 25000 轮迭代所需时间不到0.1s
- 具体计算过程如右图:



私钥解密

- 私钥解密过程：
- 首先，根据外部输入的【密码】结合保存的【主密钥密文】和【主密钥密文生成参数】恢复出【主密钥】。
- 解密钱包就此终止。所以，解密钱包并没有解密所有【私钥密文】，而是恢复出【主密钥】的过程。
- 在每次需要使用【私钥】的时候，都是通过【主密钥】去解密一次【私钥密文】，用完之后我们依然保存的是【私钥密文】。

交易签名

- 比特币定义了五种UTXO类型的交易，分别为：
- P2PKH （Pay to Public Key Hash） ；
- P2PK （Pay to Public Key） ；
- 多重签名（少于15个私钥签名） ；
- P2SH （Pay to Script Hash） ；
- OP_RETURN。

在进行交易时需要使用私钥对交易进行签名，并将签名结果写到锁定脚本中

交易验证

- 在UTXO中，使用公钥对签名函数进行验证。

莱特币

密钥管理

- 莱特币的密钥生成、存储、使用都与比特币类似。
- 不同之处在于量子链的地址：
 - LTC的钱包地址以L开头，长为34字节，如
LViZNqKo2JuXLvRWx38vha8oPhfsWpvXfr

交易

- LTC的交易和比特币的交易类似，目前支持比特币定义的五种UTXO类型的交易，分别为：P2PKH（Pay to Public Key Hash）、P2PK（Pay to Public Key）、多重签名（少于15个私钥签名）、P2SH（Pay to Script Hash）和OP_RETURN。利用这五种交易标准，客户端可以满足复杂的支付逻辑。

量子

Qtum

密钥管理

- 量子链的密钥生成、存储、使用都与比特币类似。
- 不同之处在于量子链的地址：
 - QTUM的钱包地址以Q开头，34字节，如
QiozLhNSHWi8f6NGxvgK6iJkZjCkoyJKvX

交易

- Qtum区块链的普通交易和比特币的交易是兼容的，目前支持比特币定义的五种UTXO类型的交易，分别为：P2PKH（Pay to Public Key Hash）、P2PK（Pay to Public Key）、多重签名（少于15个私钥签名）、P2SH（Pay to Script Hash）和OP_RETURN。利用这五种交易标准，客户端可以满足复杂的支付逻辑。

Thank you !