深入理解 Bitcoin 钱包的密钥体系

Jia, longcpp

yin.jia987@gmail.com, longcpp9@gmail.com

July 29, 2019

Bitcoin 系统中对 Bitcoin (BTC) 的控制权实际上是对基于 secp256k1 的 ECDSA 签名机制的私钥的控制权, 私钥是 256 比特的随机值. 为了构建一笔合法的交易, 需要利用私钥对这笔交易进行 ECDSA 签名值. 但由于私钥本身是一串 256 位的比特串, 直接以二进制形式访问私钥对用户来说非常不友好. 为了方便用户记录和识别, 衍生出了 Hex, WIF (Wallet Import Format), 压缩的 WIF 等各种展示和存储私钥的新格式. 相比 256 比特的二进制比特串, 这些格式的字符长度更短, 更易于辨别.

为了在所有交易信息都公开的前提下保护 Bitcoin 用户的隐私, 中本聪建议每次交易时都启用新的 Bitcoin 地址[1]. 遵循中本聪的建议, 可以预期相当比例的 Bitcoin 钱包中会管理多个地址及其相应的公私钥对. 维护和保管多个零散的公私钥会使用户体验变差, 用户在使用中也更容易犯错. 如果多个私钥都可以从某个值确定性派生而来, 则管理多个私钥的复杂性就简化为对这一个值的管理.

基于这样的思路以及椭圆曲线 secp256k1 的 ECDSA 签名机制的公私钥之间的关系, Pieter Wuille 在 BIP-32[2] 中提出了分层确定性钱包 (Hierarchical Deterministic Wallets, HDW) 的理念以及技术说明. HDW 允许从一个种子 (Seed) 确定性地派生出一整棵密钥树, 从而简化了对多个地址及相应公私钥对的管理. BIP32 着重介绍分层确定性钱包的原理, 并没有就如何分层以及每一层的逻辑功能给规范, 为了在不同的分层确定性钱包实现之间保持兼容性, Marek Palatinus 和 Marek Palatinus 在 2014 年提出了 BIP-44[3], 在 BIP-32 的基础上就逻辑分层等给出了具体的描述. 为了方便种子的导入导出和保管,

[1] *A new key pair should be used for each transaction to keep them from being linked to a common owner.* 摘录自 Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008. https://bitcoin.org/bitcoin.pdf

[2] Pieter Wuille. BIP-32: Hierarchical Deterministic Wallets. 2012. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki

[3] Marek Palatinus, Pavol Rusnak. BIP-44: Multi-Account Hierarchy for Deterministic Wallets. 2014. https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki

BIP-39[4] 给出了将种子和助记词 (Mnemonic Code) 相互转化的操作过程. 另外 BIP-38[5] 给出了在纯软件环境下保护种子信息的措施: 从用户口令 (Passphrase) 派生密钥实现对种子信息的加密保护. 每次需要使用种子信息都要求用户输入口令并派生密钥来解密出种子信息. 这种通过口令进行种子信息加解密的方式, 不仅保证了种子的存储安全, 也同时引入了通过口令进行鉴权的功能, 可谓一举两得.

前述的各个 BIP 协议主要是在如何简化用户与钱包软件的交互以及在不过多增加用户负担的情况下提高安全性等方面进行的探索, 主要是希望由用户掌握一个短的方便记忆的字符串, 通过密码算法, 派生出不同用途的密钥 (签名密钥, 加解密密钥). 因此这些方案中除了基本的加解密算法 (如 AES) 和签名算法 (如 ECDSA) 之外, 用到的另外一类算法就是基于口令的密钥派生函数 (Password-based Key Derivation Function, PBKDF).

[4] Marek Palatinus, Pavol Rusnak, Aaron Voisine, Sean Bowe. BIP-39: Mnemonic code for generating deterministic keys. 2013. https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki

[5] Mike Caldwell, Aaron Voisine. BIP-38: Passphrase-protected private key. 2012. https://github.com/bitcoin/bips/blob/master/bip-0038.mediawiki

1 基于口令的密钥派生函数

PBKDF 用来从一个较短的便于记忆的口令 (Passphrase/Password) 派生出密码学方案中所需的密钥. PBKDF 的输入除了口令之外, 通常还有一些其他的输入参数, 如盐值 (Salt), 迭代次数 (Iteration Count) 等. 盐值用来引入更多的熵值 (Entropy) 以进一步提升 PBKDF 的安全性, 迭代次数等信息则用来调整一次 PBKDF 计算所需的计算量. PBKDF 通常是计算密集型的, 计算需要消耗较为可观的计算量. 从口令派生密码学方案中使用的密钥时, 由于只需要进行一次操作, 通过调整参数, 可以在用户体验和计算量之间进行权衡. 而对暴力穷举攻击, 由于攻击者需要进行海量的 PBKDF 计算, 提升每次 PBKDF 计算的难度意味着攻击者需要付出更多的计算资源. 并且由于迭代次数等参数的存在, 可以根据计算机硬件的发展做对计算难度进行灵活的调整.

1.1 PBKDF2

PBKDF2 全称为 Password-Based Key Derivation Function 2, 在 PKCS#5 中取代了输出长度不大于 160 比特的 PBKDF1 作为该推荐使用的密钥派生函数. BIP39 协议使用该函数从助记词组派生一个一串随机比特, 作为 BIP32 的分层确定性钱包中用于生成主密钥对的种子值. PBKDF2 最早被包含在 PKCS#5 中发表, 后来该规范又被 IETF 以 RFC 2898[6]重新发表. 在 2017 年发布的 RFC 8018[7] 中, 仍然建议用 PBKDF2 函数作为处理口令时的哈希函数.

记 PBKDF2 函数为 $DK = \mathsf{PBKDF2}(P, S, c, dkLen)$, 其中 DK 为派生的密钥,$dkLen$ 为 DK 的字节数, P 为口令,S 为盐值,c 为迭代次数. 另外还需要为 PBKDF2 配置要使用的伪随机函数, 伪随机数函数输出为 $hLen$ 字节. 例如选用 HMAC-SHA256 函数时, $hLen = 32$. 由于用户的口令取值空间一般较小, 容易被穷举攻击, 因此 PBKDF2 的输入参数除了口令 P 和派生密钥的字节长度 $dkLen$ 之外还有盐值 S 用来增加熵值. 同样的口令在选用不同的盐值时, 可以映射成为不同的密钥. 如果 S 为 64 比特, 就意味着一个口令可以被映射成为 2^{64} 个不同的密钥, 这样可以防范彩虹表攻击 (Rainbow Table Attack)[8], 另外也可以防止同一个口令通过 PBKDF 计算出来的密钥相同, 例如在选用 64 比特的 S 时, 根据生日攻击[9] 的理论, 发生碰撞的概率 (也即同样口令不同的盐值通过 PBKDF 生成的密钥值相同) 大约为 2^{-32}. 参数为迭代次数 c, 用于调整一次 PBKDF2 计算的代价,

[6] RFC 2898: PKCS #5: Password-Based Cryptography Specification Version 2.0. https://tools.ietf.org/html/rfc2898

[7] RFC 8018: PKCS #5: Password-Based Cryptography Specification Version 2.1 https://tools.ietf.org/html/rfc8018

[8] Wikipedia: Rainbow table. https://en.wikipedia.org/wiki/Rainbow_table

[9] Wikipedia: Birthday problem. https://en.wikipedia.org/wiki/Birthday_problem

因此也会改变攻击的难度. 通过增加迭代次数, 可以显著增加攻击者海量穷举时海量计算时的成本, 而对一次诚实的派生计算没有显著影响. 因此, 在密钥派生的计算代价可以承受的范围内 (不影响用户体验),c 的取值越大越好. PKCS#5 建议 c 的最小取值为 1000, 在一些关键的应用场景中甚至需要取到 10000000.

PBKDF2 算法的大致流程为: 根据算法选取的伪随机函数 (PRF) 的输出的字节数 $hLen$ 和需要派生的密钥的字节数 $dkLen$, 将计算过程划分为 $l = \lceil (dklen/hlen) \rceil$ 个可并行的过程. 在每一个独立进行的过程中, 需要迭代 c 次 PRF 的计算, 并且后面的计算依赖于前面的计算结果, 因此必须顺序执行. 将这 l 个计算结果级联起来即得到了最终派生的密钥.

1.2 Scrypt 算法

Scrypt 算法由 Colin Percival 设计[10], 并收录在 RFC 7914[11]中. Scrypt 算法基于 memory-hard 函数, 因此在执行时对内存的消耗较高, 限制了利用硬件实现进行并行搜索空间的能力, 从而降低暴力攻击复杂度的可行性, 以此来抵抗大规模的专用硬件攻击 (例如使用 ASIC 来加速暴力穷举攻击). BIP38 使用 Scrypt 算法作为密钥派生函数, 从口令派生 AES 的加密密钥.

Scrypt 算法可以写为 $DK = \mathsf{scrypt}(P, S, N, r, p, dkLen)$, 其中 DK 为派生的密钥, P 为口令, S 为盐值, N 为 CPU/内存消耗参数, r 为区块大小参数, p 为并行参数, $dkLen$ 为 DK 的字节数. scrypt 内部首先调用 $\mathsf{PBKDF2\text{-}HMAC\text{-}SHA256}(P, S, 1, p \times 128 \times r)$ 生成包含 p 个区块的数组 B, 每个区块的长度为 $128 \times r$ 字节. 数组中的每个区块然后经由函数 scryptROMix 进行更新 $B[i] = \mathsf{scryptROMix}(r, B[i], N), 0 \le i \le p-1$, 更新后的数组 B 作为盐值再次调用 PBKDF2 得到最终结果 $DK = \mathsf{PBKDF2\text{-}HMAC\text{-}SHA256}(P, B, 1, dkLen)$. Memory-hard 的特性主要来自于对子函数 scryptROMix 的调用, 此处不再展开详述. 算法对内存的消耗与 $N * r$ 成线性关系, 用户可以根据当前 CPU 性能, 内存容量, 以及要求的并行度对进行它们调节, 以调整算法的 CPU/内存消耗.

[10] Colin Percival. Stronger key derivation via sequential memory-hard functions. 2009. http://www.tarsnap.com/scrypt/scrypt.pdf

[11] RFC 7914: The scrypt Password-Based Key Derivation Function https://tools.ietf.org/html/rfc7914

2 分层确定性钱包 (BIP32, BIP44)

在遵循中本聪的每次交易都使用新地址的建议时, 为了防止每次交易后都需要对新生成的公私钥进行备份, 钱包客户端需要维护一个预留的密钥池. 这种方式给用户在不同客户端之间进行切换, 以及钱包对用户私钥的管理带来了困难: 在切换时, 用户需要导出导入所有相关地址的私钥, 并且不能在不同的系统上同时使用钱包. 分层确定性钱包从根节点确定性派生密钥树的方法很好地解决了这个问题.

分层确定性钱包是由 Pieter Wuille 在 2012 年于 BIP32 中提出的. 分层确定性钱包中, 从根节点出发按照层级结构以一种确定性的方式从 parent key 派生 child key, 从而建立起一棵由根节点完全派生的树. 因此, 当用户在两个支持该协议的不同客户端之间进行切换时, 密钥的导入、导出只需要复制根节点（主密钥）的信息, 钱包可以根据根节点和该协议规定的派生方法确定性地派生出整棵密钥树. 因此, 用户可以方便地在不同客户端之间切换, 钱包也可以依据密钥的派生层级对密钥进行逻辑上的分层管理.

此外, 该协议允许子私钥和子公钥的派生过程相互独立, 即父私钥可以派生出子私钥和子公钥, 而父公钥只能派生出子公钥. 这种机制允许在不安全的环境中, 在没有父私钥访问权限的情况下, 依然可以进行子公钥的派生, 从而防止父私钥的泄露. 同时, 钱包的树状结构有助于用户对访问权限进行选择性的共享 (这取决于共享的密钥所处的层级, 以及共享的是私钥还是公钥）.

BIP32 着重讲述分层确定性钱包的原理, 对于客户端如何实现并没有做严格的限制, 因此，2014 年,Marek Palatinus 和 Pavol Rusnak 提出了 BIP44, 旨在 BIP32 的基础上对钱包具体的实现方式, 不同层级的逻辑含义进行了规定.

2.1 BIP32: 分层确定性钱包的密钥派生

在使用该协议从父节点派生子节点时, 实际上使用的是 512 比特的扩展密钥 $(k, c)/(K, c)$, 其中 k 代表私钥, K 代表公钥, c 则称为链码 (Chain Code), 作为额外的 256 比特的熵. 对于扩展公私钥对, 其中只有公私钥部分不同, 链码是相同的.

每一个扩展密钥至多可以有 2^{31} 个平凡子密钥 (Normal Child Key) 和 2^{31} 个增强子密钥 (Hardened Child Key), 平凡子密钥对应的索引 (Index) 从 0 到 $2^{31}-1$, 增强子密钥的索引则从 2^{31} 到 $2^{32}-1$. 为便于表示, 我们用 i_H 表示增强子密钥的索引 $i+2^{31}$. 增强子密钥的引入是为了增强整个方案的安全性, 具体原理在 Section 2.3 一节会详细介绍. 下面首先介绍从父私钥 (Private Parent Key) 到子私钥 (Private Child Key) 的派生过程.

在上述算法的第 4 步中, *Data* 参数中的前缀 $0x00$ 字节将私钥补齐为 33 个字节, 与压缩形式的公钥 (第 6 步) 一样长. 而第 10 步中, 计算结果不合法发生的概率大概为

Algorithm 1 子私钥派生算法

函数 CKDpriv$((k_{par}, c_{par}), i) \to (k_i, c_i)$ 从父扩展私钥计算子扩展私钥:
判断索引 i 是否大于等于 2^{31}, 也即判断子密钥的类型
if hardened child **then**
    let $I = \text{HMAC-SHA512}(Key = c_{par}, Data = 0x00||ser_{256}(k_{par})||ser_{32}(i))$
else
    let $I = \text{HMAC-SHA512}(Key = c_{par}, Data = ser_P(k_{par}G)||ser_{32}(i))$
end if
拆分 I 为两个 32 字节: $I_L||I_R = I, I_L = I[0, \cdots, 31], I_R = I[32, \cdots, 63]$.
子私钥 $k_i = parse_{256}(I_L) + k_{par} \mod n$, 对应的链码 $c_i = I_R$.
如果 $parse256(I_L) \geq n$ 或者 $k_i = 0$, 则计算结果不合法, 此时递增 i 并重新计算.

$1/2^{127}$. 根据上述算法也可以发现, 派生增强子密钥和平凡子密钥时 HMAC-SHA512 计算的密钥均为父链码 c_{par}, 然而输入参数 $Data$ 的构造方式不同. 派生平凡子私钥 (Private Normal Child Key) 时, $Data = ser_P(k_{par}G)||ser_{32}(i))$, 而在派生增强子私钥 (Private Hardened Child Key) 时, $Data = 0x00||ser_{256}(k_{par})||ser_{32}(i))$. 可以注意到平凡子私钥的派生只需要父公钥 $k_{par}G$, 则增强子私钥的派生则需要父私钥 k_{par}, 这也就限定了增强子密钥只能通过父私钥进行派生 (无论是计算增强子私钥还是增强子公钥).

从父公钥派生子公钥的过程在算法 2 中给出, 注意该过程只适用于派生平凡子公钥的情形. 从父公钥派生子公钥时, 最终的子公钥 $K_i = parse_{256}(I_L) + K_{par}$.

Algorithm 2 子公钥派生算法

函数 CKDpub$((K_{par}, c_{par}), i) \to (K_i, c_i)$ 从父扩展公钥计算子扩展公钥
判断索引 i 是否大于等于 2^{31}, 也即判断子密钥的类型
if hardened child **then**
    return failure
else
    let $I = \text{HMAC-SHA512}(Key = c_{par}, Data = ser_P(K_{par})||ser_{32}(i))$.
end if
拆分 I 为两个 32 字节: $I_L||I_R = I, I_L = I[0, \cdots, 31], I_R = I[32, \cdots, 63]$.
子公钥 $K_i = parse_{256}(I_L) + K_{par}$, 对应的链码为 $c_i = I_R$.
如果 $parse_{256}(I_L \geq n)$ 或者 K_i 为无穷远点, 则递增 i 重新计算

椭圆曲线点群上点加法与 $\mathbb{Z}_n$ 上模 n 加法的同态性保证了按照算法 2 派生而来的子公钥与算法 1 中根据相同索引派生出的子私钥是一一对应的. 这也就是在计算平凡子密钥时, 平凡子公钥和平凡子私钥的派生可以分开独立进行的原因. 记 $E(\mathbb{F}_p)$ 为基于有限域 $\mathbb{F}_p$, 基点为 G, 阶数为 n 的椭圆曲线点群, 则存在一个从 $\mathbb{Z}_n$ 到 $E(\mathbb{F}_p)$ 的映射

$f(x) = xG, x \in \mathbb{Z}_n$, 并且该映射 f 是保持加法操作的：即对于 $\mathbb{Z}_n$ 上的任意值 x, 都有

$$f(x + \Delta_x) = f(x) + f(\Delta_x) \Leftrightarrow (x + \Delta_x)G = xG + \Delta_x G, \text{其中 } \Delta_x \in \mathbb{Z}_n.$$

把上式中的 x 当做父私钥, 而 Δ_x 当做 HMAC-SHA512 输出的 I_L, 则 f 即为从私钥计算相应公钥的过程. 这也就是说, 将私钥 x 先加上一个偏移量 Δ_x, 再通过 f 变换得到的结果, 与先将私钥 x 和偏移 Δ_x 映射到公钥, 再做加法得到的结果是相同的.

根据算法 1 和算法 2 可知, 有两种方式从父私钥派生子公钥. 可以先根据算法 1 从父私钥派生出子私钥然后得到子公钥:

$$k_i, c_i = \mathsf{CKDpriv}((k_{par}, c_{par}), i),\ K_i = k_i G.$$

也可以先根据父私钥计算父公钥, 然后根据算法 2 得到子公钥

$$K_{par} = k_{par} G,\ K_i, c_i = \mathsf{CKDpub}((K_{par}, c_{par}), i).$$

如前所述, 算法 2 只适用于平凡子密钥派生的情形.

根据算法 1 和算法 2 可知, 可以从一个扩展密钥出发, 逐层派生出一棵密钥树, 通常称这个扩展密钥为扩展主密钥. 扩展主密钥通常不是直接生成的, 而是从一个可能更短的种子派生而来. BIP-32 中给出了具体的计算步骤. 首先利用密码学安全的的伪随机数发生器 (Cryptographically Secure Pseudo-Random Number Generator) 生成种子 $Seed$. 种子的比特长度通常介于 128 比特到 512 比特之间, 推荐使用 256 比特. 然后执行如下计算:

$$I = \mathsf{HMAC\text{-}SHA512}(Key = \text{“}Bitcoinseed\text{”}, Data = Seed),$$

将 I 划分 $I_L = [0, \cdots, 31], I_R = [32, \cdots, 63]$, 则扩展主私钥为 I_L, 主链码 (Master Chain Code) 为 I_R. 有了扩展主密钥之后, 就可以根据算法 1 和算法 2 逐层派生密钥树. 密钥派生的整体结构图参见 Figure 1 , 其中绿色表示 Recover 算法, Section 2.3 中具体阐述.

2.2 BIP44: 分层确定性钱包的逻辑分层

为了方便标记和说明, 将 $\mathsf{CKDpriv}(\mathsf{CKDpriv}(\mathsf{CKDpriv}(m, 3_H), 2), 5)$ 记为 $m/3_H/2/5$, 将 $\mathsf{CKDpub}(\mathsf{CKDpub}(\mathsf{CKDpub}(M, 3), 2), 5)$ 记为 $M/3/2/5$. 其中 m 代表主私钥, M 代表主公钥. BIP44 对确定性钱包的逻辑层次做了如下限定:

$$m/\text{purpose}'/\text{coin-type}'/\text{account}'/\text{change}/\text{address-index},$$

其中符号 $'$ 代表了这一层使用的是增强子密钥的派生方式. purpose 字段设定为常量 44‘, 代表了逻辑分层符合 BIP44 规范; coin-type 字段用于表示不同的数字货币, 从而根据数

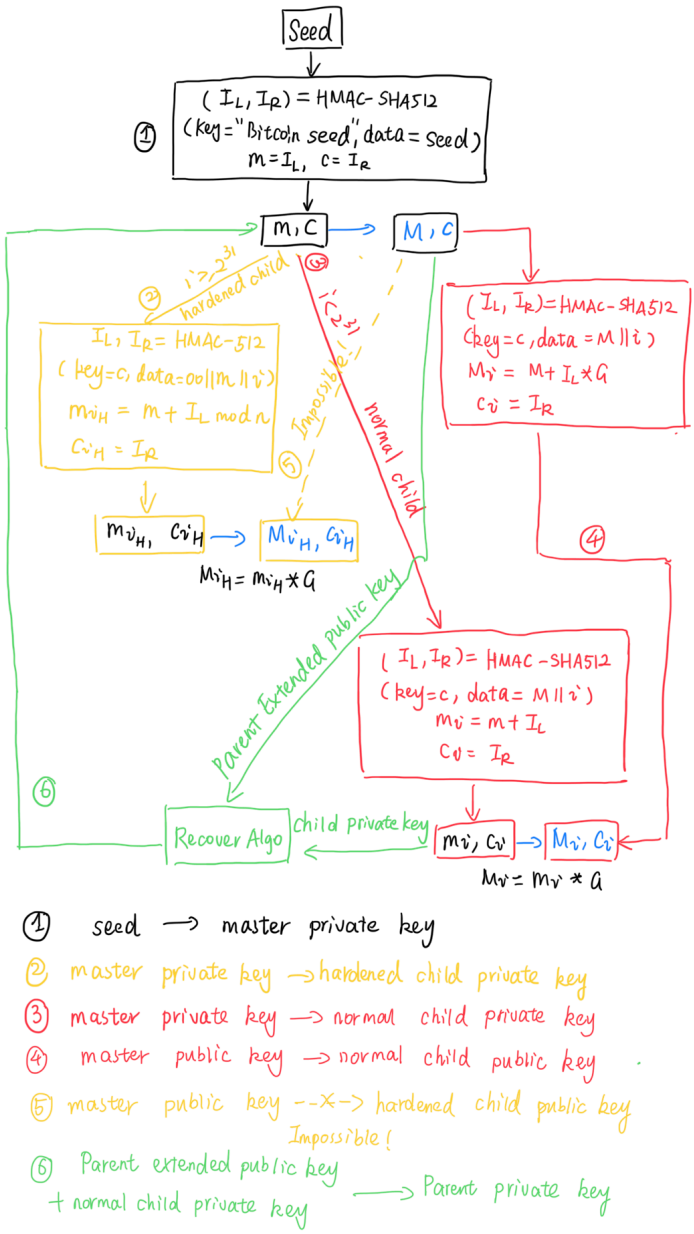

Figure 1: 分层确定性钱包密钥派生原理

字货币种类对密钥空间进行划分, 对于一种数字货币而言该字段为一个常量, 开发者需要为他们的数字货币申请尚未使用的值; account 字段用于根据不同的用户身份和使用目的对密钥空间进行划分, 允许钱包将不同账户隔离开来, 字段值从 0 开始递增, 在当前账户没有任何交易历史的情况下, 钱包软件应当阻止新账户的生成, 并且在用户从外部导入种子后, 钱包需要恢复所有已经使用过的账户; change 字段为 0 代表外部的密钥链, 也即暴露给其他人用来收款的地址, 值为 1 则代表内部的密钥链, 对于外部是不可见的, 如用作找零地址等; address-index 字段从 0 开始连续递增, 该索引即为 BIP32 中定义的用于派生子密钥的索引值.

Table 1: **数字货币的密钥分层示例**

Coin	Account	Chain	Address	Path
Bitcoin	first	external	first	m/44'/0'/0'/0/0
Bitcoin	second	external	first	m/44'/0'/1'/0/0
Bitcoin	first	change	first	m/44'/0'/0'/1/0
Bitcoin	first	external	second	m/44'/0'/0'/0/1
Bitcoin Testnet	first	external	first	m/44'/1'/0'/0/0

前面已经提及到, HD 钱包的优势在于将很多私钥的管理简化对种子的管理, 尤其是进行导入导出时只需要导出导入种子. 钱包方面需要根据用户导入的种子值, 对用户使用过的账户进行恢复. 具体步骤如下:

1. 派生第一个账户节点 (index = 0)。
2. 派生该账户的外部链节点。
3. 根据下述的 gap limit 对外部链进行扫描。
4. 如果该外部链的地址上没有发生过交易，则停止扫描。
5. 反之，递增账户的索引序号，并重复上述步骤 1。

Gap limit 目前被设置为 20, 即当扫描到某个账户的外部链中有连续 20 个没有被使用的地址时, 就停止扫描. 该算法有效的前提是, 存在账户没有被使用时, 钱包需要阻止用户通过递增索引继续生成下一个新账户, 并且在同一个账户下, 存在 20 个连续的地址没有被使用时, 阻止用户跨越这些地址生成下一个新地址.

2.3 安全性分析

分层确定性钱包的密钥派生方案能够保证给定一个公钥 K, 攻击者计算出相应私钥 k 的难度至少和椭圆曲线点群上的离散对数问题一样难, 这是由基于椭圆曲线的密码学方案所提供的. 通过给定的扩展子私钥 (k_i, c_i) 和索引值 i 恢复父私钥 k_{par} 的难度至少和攻击 HMAC-SHA512 一样难, 而 HMAC-SHA512 提供了 256 比特的安全强度. 判断给定的 N 个 (索引, 扩展子私钥) 对 $(i_j, (k_j, c_j)), 0 \leq j \leq N, 2 \leq N \leq 2^{32} - 1$ 是否由同一个扩展父私钥派生而来, 至少和攻击 HMAC-SHA512 一样难, 也是不现实的.

给定一个扩展父公钥 (K_{par}, c_{par}) 和一个平凡子公钥 K_i, 找到该子公钥对应的索引 i 是容易的. 平凡子密钥的索引值 i 的取值范围为 $0 \leq 0 \leq 2^{31} - 1$, 在前述条件下, 可以遍历 i 来重复子公钥的派生的过程, 并比较派生出的子公钥是否与给定的子公钥相等. 如果相等, 则 i 就是该子公钥对应的索引值.

给定一个扩展父公钥 (K_{par}, c_{par}) 和平凡子私钥 (k_i), 计算父私钥 k_{par} 是容易的. 很容易根据平凡子私钥计算平凡子公钥, 则根据前述说明, 很容易计算出该子密钥对应的索引值 i. 由于 $I = \text{HMAC-SHA512}(Key = c_{par}, Data = ser_P(K_{par})||ser_{32}(i))$, 并且有 $I_L = I[0, \cdots, 31]$, $I_R = I[32, \cdots, 63]$ 以及 $k_i \equiv parse_{256}(I_L) + k_{par} \mod n$. 则 $k_{par} \equiv k_i - parse_{256}(I_L) \mod n$.

Listing 1 中的代码展示是前述攻击过程的 PoC 示例, 代码基于分层确定性钱包的 Python 实现库 bip32utils[12].

Listing 1: 基于父扩展公钥和平凡子私钥恢复父私钥的攻击示例

```
CURVE_GEN  = ecdsa.ecdsa.generator_secp256k1
CURVE_ORDER  = CURVE_GEN.order()
BIP32_HARDEN = 0x80000000
def recover_parent_privkey(parent_pubkey,child_privkey):
    # traverse all possible non-hardened child index (0~2^31-1) to
    find the corresponding index of child_privkey
    for i in xrange(0, BIP32_HARDEN +1):
        child=parent_pubkey.CKDpub(i)
        if(child.PublicKey()==child_privkey.PublicKey()):
            break
    # if i is larger than 2^31-1, means that it corresponds to a hardened child,
    then the recovery is impossible
    if i & BIP32_HARDEN:
        print "can not recover parent private key with a hardened child node"
        return
    # data is composed of public key || i
```

[12] https://github.com/lyndsysimon/bip32utils

```
    data=parent_pubkey.PublicKey()+struct.pack(">L",i)
    (Il,Ir)=parent_pubkey.hmac(data)
    Il_int = string_to_int(Il)
    if Il_int > CURVE_ORDER:
        return None
    cpk_int=string_to_int(child_privkey.k.to_string())
    # recover parent private key ppk_int from cpk_int= ppk_int + Il_int mod n
    ppk_int=(cpk_int-Il_int)%CURVE_ORDER
    return int_to_string(ppk_int).encode('hex')
```

按照 BIP32[13] 测试向量 2 中给定的 seed，计算出相应路径下的 child private/public key，按照上述算法 1，通过调用 recover_parent_privkey(M,m/0) 可成功恢复 M 的私钥 m。

这对于分层确定性钱包的安全性有重要影响, 因为这意味着知道一个扩展父公钥和平凡子私钥, 就可以推算出父私钥. 因此管理扩展公钥需要比通常的公钥更为谨慎. 对于采用增强方式派生的密钥则不存在上述问题, 因为增强密钥只能通过扩展父私钥派生而来. 这也是在 BIP44 中规定在 purpose, coin-type, account 层使用 hardened 节点的原因, 以防止这三层上一层的私钥被攻击者按照前述程恢复出来.

[13] https://github.com/bitcoin/bips/blob/33e040b7bdf5d937599d2401454878d6293476c9/bip-0032.mediawiki#Test_vector_2

3 BIP39: 从助记词生成确定性密钥

BIP39 提出了一种从助记词 (Mnemonic Code) 派生 HD 钱包种子的方法, 使得用户只需要掌握这组助记词即可借助钱包实现派生出 HD 钱包的密钥树, 从而优化用户的交互体验. 该协议主要包含两个过程: 从熵值 (Entropy) 生成助记词的过程以及从助记词派生种子的过程. 其中, 助记词是从事先定义好的单词列表 (Wordlist) 里选出的符合相应长度要求的单词集合, 目前针对英语, 日语, 韩语, 西班牙语, 中文 (简体/繁体), 法语, 意大利语都有相应的单词列表, 在使用前都对它们进行 UTF-8 编码处理, 详见单词列表[14].

用来产生助记词的熵值的比特长度为 128 到 512 比特, 要求长度必须是 32 比特的整数倍. 熵值的比特长度越大, 安全性就越高, 但同时生成助记词中单词的个数也越多. 助记词的产生过程: 假设初始的熵值为 e, 比特长度为 $\mathsf{bits}(e)$, 取 $\mathsf{SHA256}(e)$ 的前 $\mathsf{bits}(e)/32$ 比特作为检验和 (Checksum) 缀在 e 的后面, 随后将级联后的比特串以 11 比特为单位进行分组, 即每个分组的数字范围是为 $[0, \cdots, 2047]$, 将其作为索引从单词列表中读取相应的单词, 最终选出的词语即构成了一个助记词句子 (Mnemonic Sentence). 当 $\mathsf{bits}(e) = 128, 192, 256$ 时, 对应的助记词的个数为 12, 18, 24. 参见 Figure 2(截取自 "Mastering Bitcoin 2nd Edition - Programming the Open Blockchain"[15]).

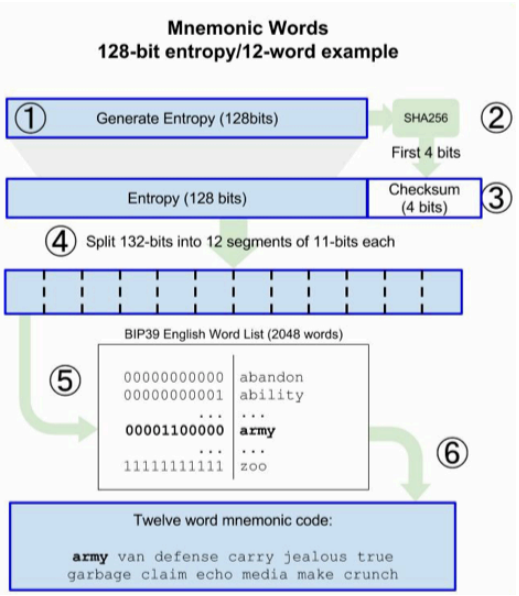

Figure 2: 从熵值生成助记词

BIP39 规定使用 PBKDF2 函数从助记词生成种子, PBKDF2 各参数设置如下: 助

[14] Wordlists. https://github.com/bitcoin/bips/blob/master/bip-0039/bip-0039-wordlists.md
[15] https://github.com/bitcoinbook/bitcoinbook

记词以 UTF-8 NFKD (UTF-8 using Normalization Form Compatibility Decomposition) 编码后作为 P, "mnemonic" 级联 P 同样使用 UTF-8 NFKD 进行编码后作为盐值 S, $c = 2048$, $PRF = \text{HMAC-SHA512}$, $dklen = 512$. 最终该算法输出的 512 比特作为分层确定性钱包的种子. Listing 2 中的代码基于 Trezor 的 python-mnemonic 库[16]:

Listing 2: 基于助记词句子的种子派生

```
def seed_derivation_from_mnemonic():
    import binascii
    import sys
    from bip32utils import BIP32Key
    if len(sys.argv) > 1:
        data = sys.argv[1]
    else:
        data = sys.stdin.readline().strip()
    data = binascii.unhexlify(data)
    m = Mnemonic('english')
    code=m.to_mnemonic(data)
    seed=m.to_seed(code,'TREZOR')
    xprv = BIP32Key.fromEntropy(seed).ExtendedKey()
    print('mnemonic : %s (%d words)' % (code, len(code.split(' '))))
    print('seed     : %s (%d bits)' % (binascii.hexlify(seed),len(seed) * 4))
    print('xprv     : %s' % xprv)
```

Listing 3: Listing 2 的执行结果示例

```
mnemonic : void come effort suffer camp survey warrior heavy shoot primary
    clutch crush open amazing screen patrol group space point ten exist slush
    involve unfold (24 words)

seed : 01f5bced59dec48e362f2c45b5de68b9fd6c92c6634f44d6d40aab69056506f0e35524a
       518034ddc1192e1dacd32c1ed3eaa3c3b131c88ed8e7e54c49a5d0998 (256 bits)

xprv : xprv9s21ZrQH143K39rnQJknpH1WEPFJrzmAqqasiDcVrNuk926oizzJDDQkdiTv
       NPr2FYDYzWgiMiC63YmfPAa2oPyNB23r2g7d1yiK6WpqaQS
```

[16] https://github.com/trezor/python-mnemonic

4 BIP38: 基于口令保护私钥

BIP38 提出了基于口令 (Passphrase/Password) 对私钥进行加密保护的方案. 该方案只考虑了针对私钥的机密性保护, 而没有考虑提供完整性保护 (从密码学角度来讲不算是最佳实践), 因此其主要针对纸钱包 (Paper Wallet) 等私钥密文不容易遭受篡改的应用场景. BIP38 中给出了两种加密私钥的方法, 其中一种方法使用了 EC 乘法操作, 另一种没有使用 EC 乘法操作. 两种方法实现的功能有很大的区别. 方便起见, 后续我们用方法一指代没有使用 EC 乘法操作的加密方法, 用方法二指代使用了 EC 乘法操作的加密方法.

未使用 EC 乘法操作的方法一中, 对于已经产生的私钥, 使用用户设置的口令对私钥进行加密. 加密后的密文与口令在解密私钥时缺一不可. 由于两者可以独立管理, 可以提高私钥的安全性.

使用了 EC 乘法操作的方法二中, 用户首先会生成一个中间口令码 (Intermediate Passphrase Code), 并将其传输给一个第三方. 第三方可以为用户生成新的地址, 并将新地址和计算地址对应的私钥所必须的信息发给用户用户拿到前述信息后结合口令, 即可算出新地址对应的私钥. 这种方式允许用户将生成新地址的工作委托给无需可信的第三方来做, 但同时保证只有知道口令和密文的人才能计算出对应私钥.

4.1 不使用 EC 乘法的加密方法一

该方法对已有私钥 k 进行加密, 加密过程分为两步: 用 Scrypt 算法从口令 P 派生 AES 算法加密所需密钥; 使用 AES 算法对私钥信息进行加密. 记该私钥对应的 Bitcoin 地址为 $address$, 按照以下 Algorithm 3 展示的步骤进行.

Algorithm 3 不使用 EC 乘法的加密方法一

1: 计算地址哈希值 $addresshash = \mathsf{SHA256}(\mathsf{SHA256}(address))[0, 1, 2, 3]$
2: 用 Scrypt 根据口令等派生 64 字节值 $v = v_1||v_2, v_1 = v[0, \cdots, 31], v_2 = [32, \cdots, 63]$
3: 计算 $ek_1 = \mathsf{AES256Encrypt}(block = k[0...15] \oplus v_1[0, \cdots, 15], key = v_2)$
4: 计算 $ek_2 = \mathsf{AES256Encrypt}(block = k[16...31] \oplus v_1[16...31], key = v_2)$
5: 密文为 $0x01||0x42||flagbyte||addrhash||ek_1||ek_2$ 的 $\mathsf{Base58CheckEncode}$

在使用 Scrypt 派生密钥时, 参数 P 为用 UTF-8 编码并用 NFC (Unicode Normalization Form C) 正则化后的口令, S 设置为 $addresshash$, 也即 $\mathsf{SHA256}(\mathsf{SHA256}(address))$ 的前 4 个字节, 该字段无需保密, 目的是为了引入更多的熵值, 以增加攻击者进行暴力破解时的搜索空间. n, r, p 的值为系统设置参数, 并且 $n = 16384$, $r = 8$, $p = 8$, $length = 64$. 最终的密文为 39 个字节, 参见 Figure 3. 其中 $flagbyte$ 为 1 个字节, 最高两位用来区分是否使用 EC 乘法: $flagbyte_{7,6} = 11$ 表示不使用 EC 乘法, $flagbyte_{7,6} = 00$ 表示使用了

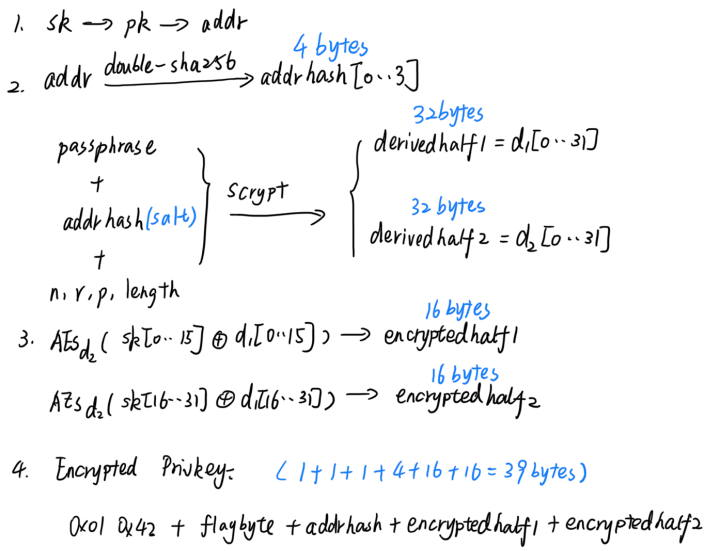

Figure 3: 不使用 EC 乘法的加密方法一: 加密过程

EC 乘法. 比特 $flagbyte_5 = 1$ 表示使用比特币地址是根据压缩形式的公钥计算而来的, 注意 Bitcoin 中同一个私钥可以有两种比特币地址, 分别对应压缩形式公钥和不压缩形式公钥. $flagbyte_2 = 1$ 表示在使用 EC 乘法加密时还使用了 lot+sequence 字段, 这两个字段的作用会在后面说明. 其余比特位保留.

解密过程首先根据用户掌握的 passphrase 计算出 AES 的解密密钥, 随后对密文解密, 首先也需要通过 Scrypt 计算 v_1, v_2, 然后利用 AES256Encrypt 解密得到私钥的值, 通过私钥根据 $flagbyte$ 中的信息派生 Bitcoin 地址, 并进一步计算 $addresshash$ 并与密文中的该字段进行对比. 如果不匹配则报告输入的口令错误. 加密和解密的 PoC 代码实现参见 Listing 4, 执行结果参见 Listing 5.

Listing 4: 不使用 EC 乘法的加解密过程示例

```
1
2  import hashlib
3  import secrets
4  from Crypto.Cipher import AES
5  import base58check
6  import scrypt
7  from bitcoinlib.keys import *
8  import ecdsa
9  from ecdsa.curves import SECP256k1
10 from ecdsa.ecdsa import int_to_string, string_to_int
11
```

```
12 # parameter for Scrypt function
13 SCRYPT_N=16384
14 SCRYPT_R=8
15 SCRYPT_P=8
16
17 def b58check(data):
18     # return base58 encoding of data with checksum
19     checksum=hashlib.sha256(data).digest()
20     checksum=hashlib.sha256(checksum).digest()
21     data+=checksum[:4]
22     return base58check.b58encode(data)
23
24 def enc_without_ec_multi(privkey,address,passphrase,compressed):
25     # compute addrhash of corresponding address to known private key
26     addrhash=hashlib.sha256(address).digest()
27     addrhash=hashlib.sha256(addrhash).digest()
28     addrhash=addrhash[:4]
29     # derive encryption key for AES
30     passphrase=passphrase.decode('utf-8')
31     d=scrypt.hash(passphrase,addrhash,SCRYPT_N,SCRYPT_R,SCRYPT_P,64)
32     derivedhalf1=d[:32]
33     derivedhalf2=d[32:64]
34     # encryption process
35     m1=[a^b for a,b in zip(privkey[:16],derivedhalf1[:16])]
36     m2=[a^b for a,b in zip(privkey[16:32],derivedhalf1[16:32])]
37     e1=AES.new(derivedhalf2).encrypt(bytes(m1))
38     e2=AES.new(derivedhalf2).encrypt(bytes(m2))
39    # pack encryption data
40     flagbyte=b'\xe0' if compressed==1 else b'\xC0'
41     res=b'\x01\x42'+flagbyte+addrhash+e1+e2
42     return b58check(res)
43
44 def dec_without_ec_multi(encrypted_privkey,passphrase):
45     # unpack data
46     data=base58check.b58decode(encrypted_privkey)
47     addrhash_v=data[3:7]
48     passphrase=passphrase.decode('utf-8')
49     # derive decryption key for AES
50     d=scrypt.hash(passphrase,addrhash_v,SCRYPT_N,SCRYPT_R,SCRYPT_P,64)
51     derivedhalf1=d[:32]
52     derivedhalf2=d[32:64]
53     # decryption process
```

```
    m1=AES.new(derivedhalf2).decrypt(data[7:23])
    m2=AES.new(derivedhalf2).decrypt(data[23:39])
    privkey1=bytes([a^b for a,b in zip(m1,derivedhalf1[:16])])
    privkey2=bytes([a^b for a,b in zip(m2,derivedhalf1[16:32])])
    privkey=privkey1+privkey2
    # verify whether addrhash is correct
    k=Key(privkey)
    addr=k.address_uncompressed() if data[2]==0xC0 else k.address()
    addrhash=hashlib.sha256(addr.encode('ascii')).digest()
    addrhash=hashlib.sha256(addrhash).digest()
    if addrhash!=addrhash_v:
        assert("address is not valid!")
    else:
        return privkey1+privkey2
```

Listing 5: Listing 4 的执行结果示例

```
Encryption with no EC multiplication:
Privkey is : b'\xcb\xf4\xb9\xf7\x04p\x85k\xb4\xf4\x0f\x80\xb8~\xdb\x90\x86Y\x97\
    xff\xeem\xf3\x15\xab\x16mq:\xf43\xa5'
Encrypted private key is : b'6
    PRVWUbkzzsbcVac2qwfssoUJAN1Xhrg6bNk8J7Nzm5H7kxEbn2Nh2ZoGg'

Decryption with no EC multiplication:
Decryption succeed!
Private key is  b'\xcb\xf4\xb9\xf7\x04p\x85k\xb4\xf4\x0f\x80\xb8~\xdb\x90\x86Y\
    x97\xff\xeem\xf3\x15\xab\x16mq:\xf43\xa5'
```

4.2 使用 EC 乘法的加密方法二

该方法利用了 EC 乘法的同态性质, 主要思想类似于一个基于椭圆曲线的密钥协商方案. 用户事先由口令和熵值等生成一个随机数 x, 并计算对应的椭圆曲线群上的点 $P = xG$, 将其发给第三方. 随后第三方选择一个随机数 k 与该点进行 EC 的乘法操作, 也即新生成的公钥 $P' = kP = (x \cdot k)G$. 随后将 k 加密后发送给用户 (加密过程与方法一中类似). 用户收到数据解密后, 计算出新私钥 $x \cdot k$. 假设用于生成 x 的口令没有泄露, 则用户就可以确认新私钥是足够安全的 (只有知道口令的人才能计算出新私钥), 方法二主要包括两个阶段: 初始化阶段和私钥生成阶段.

初始化阶段主要功能是用户利用口令和熵值计算中间口令码 (Intermediate Passphrase Code) 并发送给第三方. 在计算时, 用户可以选择性的生成一个 20 比特的批号 (Lot Num-

ber) 和 12 比特的序列号 (Sequence Number). 在用户向第三方请求大量的加密私钥时, 需要检查返回的私钥对应的批号和序列号与用户自己提供的中间口令码中的一致. 4 个字节批号和序列号并不是必须的. 根据是否使用这两个字段, 初始化阶段稍微不同.

在使用批号和序列号时, 按照 Algorithm 4 中的步骤进行初始化. 其中第 3 步中 Scrypt 输入参数中的 P 为 UTF-8 编码的并用 NFC 形式正则化后的用户口令, 盐值 S 为第 1 步中生成的 $ownersalt$, $n = 16384, r = 8, p = 8, length = 32$. 第 5 步中的点的压缩表示形式约束仅限于这一步的计算, 与生成 Bitcoin 地址时使用的公钥的表示形式无关.

Algorithm 4 使用批号和序列号时的初始化阶段

1: 生成 4 个字节的随机数 $ownersalt$
2: 批号和序列号用大端法形式的 4 字节的整数 $lotsequence = lotnumber * 4096 + sequencenumber$
3: 用 Scrypt 从口令等信息派生 32 字节密钥 $prefactor$
4: 计算 $passfactor = \mathsf{SHA256}(\mathsf{SHA256}(prefactor||ownerentropy))$
5: 计算 $passfactor * G$, 该点的压缩表示形式记为 $passpoint$
6: 传送 $ownerentropy = ownersalt||lotsequence$ 以及 $passpoint$ 给第三方

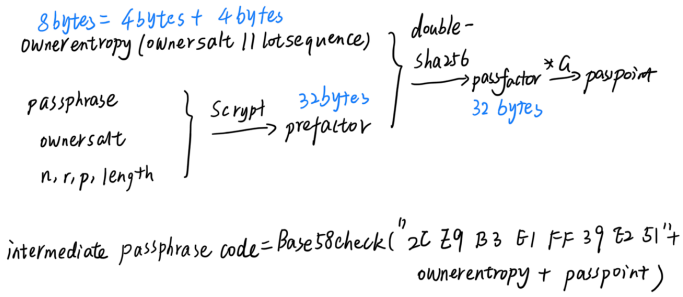

Figure 4: 使用批号和序列号时方法二的初始化过程

最后一步中真正发送给第三方的是下面名为 *intermediate_passphrase_string* 的字符串 (也就是前述的中间口令码):

$$\mathsf{Base58CheckEncode}(2C\ E9\ B3\ E1\ FF\ 39\ E2\ 51||ownerentropy||passpoint)$$

由于魔数 (Magic Number) "2C E9 B3 E1 FF 39 E2 51" 的采用, 编码之后的字符串会以 "passphrase" 开头. 72 个字符的 *intermediate_passphrase_string* 编码了 49 个字节的信息以及校验码 49 个字节的信息包括 8 字节的魔数, 8 字节的 *ownerentropy* 以及 33 字节的 *passpoint*. 参见 Figure 4.

不使用批号和序列号时的初始化过程相对简单. 首先生成 8 字节的随机数 *ownersalt* 而不是 Algorithm 4 中的 4 个字节. 略过批号和序列号字段, 此时 *ownerentropy* 是

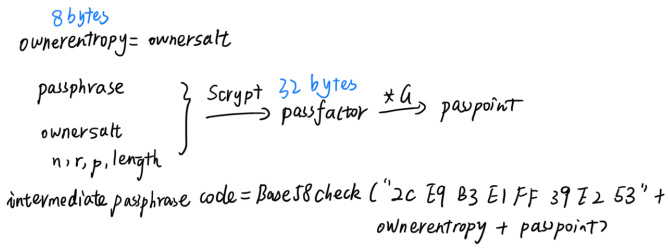

Figure 5: 不使用批号和序列号时方法二的初始化过程

ownersalt 的别名. Algorithm 4 中的第 4 步从 $prefactor$ 生成 $passfactor$ 的计算也略过, Scrypt 的输出直接作为 $passfactor$. Base58CheckEncode 编码时使用魔数 "2C E9 B3 E1 FF 39 E2 53" (最后一个字节从 $0x53$ 变成 $0x51$). 参见 Figure 5.

4.2.1 加密私钥计算

第三方拿到中间口令码之后, 可以用它为用户生成新的加密私钥. 确切地说, 第三方可以为用户生成新的公钥, 并将计算对应私钥的所需的信息加密后一并发送给用户, 同时拥有口令和密文信息的人才能从中计算出对应的私钥.

计算时首先需要设置 *flagbyte*, 由于使用了 EC 乘法, $flagbyte_{7,6} = 00$. 如果采用压缩形式的公钥派生 Bitcoin 地址, 则设置 $flagbyte_5 = 1$. 如果字段 *ownerentropy* 中包含批号系列号字段, 则设置 $flagbyte_3 = 1$. 然后生成 24 个字节的随机数 *seedb*, 并计算 $factorb = \mathsf{SHA256}(\mathsf{SHA256}(seedb))$. 计算 $generatedpubkey = factorb * passpoint$ 并计算对应的 Bitcoin 地址 $generatedaddress = \mathsf{RIPEMD160}(\mathsf{SHA256}(generatedpubkey))[0, \cdots, 19]$, 可使用未压缩或者压缩形式的公钥表示, 这就是第三方为用户新生成的 Bitcoin 地址.

前述过程完成了新公钥的生成, 下面介绍具体的加密过程, 加密过程与方法一中的加密过程类似: 利用 Scrypt 派生 AES 的加密密钥, 然后对与新公钥关联的 *seedb* 进行加密. 具体过程在 Algorithm 5 中给出.

Algorithm 5 使用 EC 乘法的方法二的加密过程

1: 计算 $addresshash = \mathsf{SHA256}(\mathsf{SHA256}(generatedaddress))[0, 1, 2, 3]$
2: 用 Scrypt 从 $passpoint$ 派生 32 字节值 $v = v_1||v_2, v_1 = v[0, \ldots, 31], v_2 = [32, \ldots, 63]$
3: 计算 $ek_1 = \mathsf{AES256Encrypt}(block = (seedb[0, \ldots, 15] \oplus v_1[0, \ldots, 15]), key = v_2)$
4: 计算 $ek_2 = \mathsf{AES256Encrypt}(block = ((ek_1[8, \ldots, 15]||seedb[16, \ldots, 23]) \oplus v_1[16, \ldots, 31]), key = v_2)$
5: 密文为 $0x01||0x43||flagbyte||addresshash||ownerentropy||ek_1[0...7]||ek_2$ 的 Base58CheckEncode

第 2 步中函数 Scrypt 的参数 P 为用户提供的 33 字节的 *passpoint*, 参数 S 为 $addresshash||ownerentropy$, $n = 1024, r = 1, p = 1, length = 64$. 这里最终数据的格式

略有些奇怪, 推测是为了方法一中的结果保持长度一致 (都是 39 个字节), 参见 Figure 6.

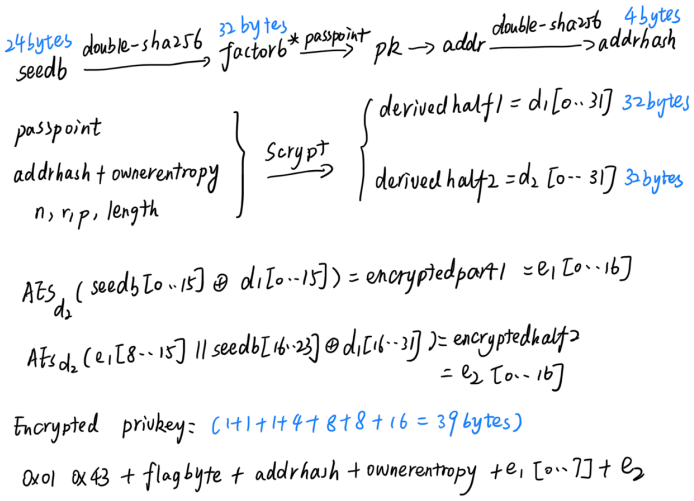

Figure 6: 使用 EC 乘法的方法二的加密过程

4.2.2 计算新地址对应的私钥

根据第三方返回的密文计算新私钥的过程在 Algorithm 6 中展示. 由于新公钥为 $passpoint \cdot factorb = passfactor \cdot factorb * G$, 对应的私钥即为 $passfactor \cdot factorb$.

Algorithm 6 使用 EC 乘法的方法二的解密过程与私钥计算

1: 用 Scrypt 函数从 *ownersalt* 和口令派生 $passfactor$ 并重新计算 *passpoint*
2: 用 Scrypt 函数从 $passpoint, addresshash, ownerentropy$ 派生 *seedb* 的解密密钥
3: 用 AES256Decrypt 解密 ek_2 得到 $seedb[16, \ldots, 23]$ 和 $ek_1[8, \ldots, 15]$
4: 用 AES256Decrypt 解密 ek_1 得到 $seedb[0, \ldots, 15]$
5: 用 *seedb* 计算 $factorb$
6: 计算地址 *generatedaddress* 对应的私钥为 $passfactor \cdot factorb$
7: 根据 *flagbyte* 中的公钥表示形式, 从私钥计算对应的 Bitcoin 地址
8: 根据 Bitcoin 地址计算 *addresshash* 并与密文中的字段比较. 若不匹配, 则口令错误

在恢复了新私钥后, 客户端还需要验证计算出的私钥与之前 *addresshash* 给出的地址是否对应, 如果不是, 则需要向用户返回错误信息, 即用户输入了错误的口令. 由于该协议描述的目的场景是针对纸钱包, 所以可假定上述编码后的密文的完整性是保证的. 使用 EC 乘法的加解密过程示例在 Listing 6中给出, 执行结果在 Listing 7中给出.

Listing 6: 使用 EC 乘法的加解密过程示例

```
1 def init_enc_with_ec_multi(passphrase,ownersalt,lotsequence):
```

```
    if len(ownersalt+lotsequence)!=8:
        assert("ownerentropy must be 8 bytes!")
    # derive prefactor from scrypt
    prefactor=scrypt.hash(passphrase,ownersalt,SCRYPT_N,SCRYPT_R,SCRYPT_P,32)
    #derive passpoint & pack intermediate passphrase code
    if len(lotsequence)==0:
        intermediate_passphrase_code=b'\x2C\xE9\xB3\xE1\xFF\x39\xE2\x53'
        intermediate_passphrase_code+=ownersalt
        passfactor=prefactor
        print("ownersalt",ownersalt)
    else:
        intermediate_passphrase_code=b'\x2C\xE9\xB3\xE1\xFF\x39\xE2\x51'+
            ownersalt+lotsequence
        prefactor+=(ownersalt+lotsequence)
        passfactor=hashlib.sha256(prefactor).digest()
        passfactor=hashlib.sha256(passfactor).digest()
    k=Key(bytes(passfactor))
    passpoint=k.public_compressed_byte
    intermediate_passphrase_code+=passpoint
    return b58check(intermediate_passphrase_code)

def enc_with_ec_multi(intermediate_passphrase_code,flag_lotseq,flag_compr):
    #unpack intermediate passphrase code
    intermediate_passphrase_code=base58check.b58decode(
        intermediate_passphrase_code)
    ownerentropy=intermediate_passphrase_code[8:16]
    passpoint=intermediate_passphrase_code[16:49]

    # generate random seedb to derive new public key & address
    seedb=secrets.token_bytes(24)
    factorb=hashlib.sha256(seedb).digest()
    factorb=hashlib.sha256(factorb).digest()
    k=Key(bytes(passpoint))
    (x,y)=k.public_point()
    c=ecdsa.ellipticcurve.CurveFp(SECP256k1.curve.p(),SECP256k1.curve.a(),
        SECP256k1.curve.b())
    p=ecdsa.ellipticcurve.Point(c,x,y)
    new_point=p.__mul__(string_to_int(factorb))
    k=Key(new_point.x()+new_point.y())
    addr=k.address() if flag_compr==1 else k.address_uncompressed()
    addrhash=hashlib.sha256(base58check.b58decode(addr)).digest()
    addrhash=hashlib.sha256(addrhash).digest()
```

```
41      addrhash=addrhash[:4]
42
43      # derive encryption key and use AES to encrypt seedb
44      salt=addrhash[:4]+ownerentropy
45      d=scrypt.hash(passpoint,salt,1024,1,1,64)
46      derivedhalf1=d[:32]
47      derivedhalf2=d[32:64]
48      m1=[a^b for a,b in zip(seedb[:16],derivedhalf1[:16])]
49      e1=AES.new(derivedhalf2).encrypt(bytes(m1))
50      m2=[a^b for a,b in zip(e1[8:16]+seedb[16:24],derivedhalf1[16:32])]
51      e2=AES.new(derivedhalf2).encrypt(bytes(m2))
52
53      # put all together and encode with base58check
54      flagbyte=(0x04 if flag_lotseq==1 else 0)^(0x20 if flag_compr==1 else 0)
55      flagbyte=b'\x00' if flagbyte==0 else bytes([flagbyte])
56      res=b'\x01\x43'+flagbyte+addrhash[:4]+ownerentropy+e1[:8]+e2
57
58      return b58check(res)
59
60  def dec_with_ec_multi(passphrase,enc_priv):
61      #unpack encrpted private key
62      enc_priv=base58check.b58decode(enc_priv)
63      flagbyte=enc_priv[2]
64      addrhash=enc_priv[3:7]
65      ownerentropy=enc_priv[7:15]
66      ownersalt=ownerentropy[:4] if flagbyte&0x04!=0 else ownerentropy
67
68      # derive passpoint from passphrase
69      prefactor=scrypt.hash(passphrase,ownersalt,SCRYPT_N,SCRYPT_R,SCRYPT_P,32)
70      if (flagbyte&0x04==0):
71          passfactor=prefactor
72      else:
73          prefactor+=ownerentropy
74          passfactor=hashlib.sha256(prefactor).digest()
75          passfactor=hashlib.sha256(passfactor).digest()
76      k=Key(bytes(passfactor))
77      passpoint=k.public_compressed_byte
78
79      # derive encryption key for AES from passpoint and decrypt
80      d=scrypt.hash(passpoint,addrhash+ownerentropy,1024,1,1,64)
81      derivedhalf1=d[:32]
82      derivedhalf2=d[32:64]
```

```
83      m2=AES.new(derivedhalf2).decrypt(enc_priv[23:39])
84      seedb2=bytes([a^b for a,b in zip(m2,derivedhalf1[16:32])])
85      m1=AES.new(derivedhalf2).decrypt(enc_priv[15:23]+seedb2[:8])
86      seedb1=bytes([a^b for a,b in zip(m1,derivedhalf1[0:16])])
87      seedb=seedb1+seedb2[8:16]
88
89      #derive private key and verifiy validness of it
90      factorb=hashlib.sha256(seedb).digest()
91      factorb=hashlib.sha256(factorb).digest()
92      k=Key(factorb)
93      (x,y)=k.public_point()
94      c=ecdsa.ellipticcurve.CurveFp(SECP256k1.curve.p(),SECP256k1.curve.a(),
            SECP256k1.curve.b())
95      p=ecdsa.ellipticcurve.Point(c,x,y)
96      new_point=p.__mul__(string_to_int(passfactor))
97      k=Key(new_point.x()+new_point.y())
98      addrhash_v=k.address() if flagbyte&0x20==1 else k.address_uncompressed()
99      addrhash_v=hashlib.sha256(base58check.b58decode(addrhash_v)).digest()
100     addrhash_v=hashlib.sha256(addrhash_v).digest()
101     if addrhash_v[:4]==addrhash:
102         return k.wif()
103     else:
104         assert("privkey is invalid")
```

Listing 7: Listing 6 的执行结果示例

```
1  Intialization:
2  Passphrase :  b'
       passphraseaB8feaLQDENqCgr4gKZpmf4VoaT6qdjJNJiv7fsKvjqavcJxvuR1hy25aTu5sX'
3  Entropy : b'O\xcaZ\x97@@\xf0\x01'
4  Intermediate_passphrase_code is  b'
       passphraseaB8feaLQDENqCgr4gKZpmf4VoaT6qdjJNJiv7fsKvjqavcJxvuR1hy25aTu5sX'
5
6  Encryption:
7  Under this intermediate_passphrase_code, encrypted key is :
8  b'6PgKDE6dzgJ3cM1Z2fwdhxGyH3W59H1kZjCDHuETevCjCWDh6H7j95X6EF'
9
10 Decryption:
11 Decrypted key plaintext is:
12 L41y9jwBDLa1h2nHxNnhKGCtn7MDEpMNNvrd1o3WTuoXN8srsrD9
```

4.2.3 新地址的确认码机制

在第三方为用户生成新的 Bitcoin 地址之后, 用户可能并不需要立即进行解密, 私钥只有在花费对应地址的 UTXO 时才需要. 但用户需要对产生的新地址进行确认, 以防止出现自己无法计算新的地址对应的私钥. 因此, 对于使用 EC 乘法的私钥加密方式, 协议还设计了一个独立的验证方式: 第三方可以向用户返回一个以"cfrm38"开头的 75 个字符的确认码 (Confirmation Code), 用来协助用户验证新地址对应的私钥是用户可计算出来的. 确认码的生成过程参见 Algorithm 7.

Algorithm 7 确认码生成过程

1: 需要 Algorithm 5 加密过程中的值:$flagbyte, ownerentropy, factorb, v_1, v_2$
2: 计算 $pointb = factorb * G$, 压缩形式表示为 33 个字节, 第一个字节为 $0x02$ 或者 $0x03$
3: 计算 $pointbprefix = pointb[0] \oplus (v_2[31]\&0x01)$ The first byte is 0x02 or 0x03. XOR it by (derived-half2[31] & 0x01), call the resulting byte pointbprefix.
4: 计算 $pointbx_1 = \mathsf{AES256Encrypt}(block = (pointb[1...16] \oplus v_1[0...15]), key = v_2)$
5: 计算 $pointbx_2 = \mathsf{AES256Encrypt}(block = (pointb[17...32] \oplus v_1[16...31]), key = v_2)$
6: 连接几个值得到 33 字节 $epb = pointbprefix||pointbx1||pointbx2$
7: 确认码为 643BF6A89A$||flagbyte||addresshash||ownerentropy||epb$ 的 $\mathsf{Base58CheckEncode}$

除了常数部分, 确认码与私钥密文数据的不同之处在于 epb, 其中包含了 $factor * G$ 的信息, 允许用户收到确认码后计算出地址, 并可以根据 $addresshash$ 进行验证如果验证通过, 用户可以使用该地址进行交易, 并且需要花费时可以计算出相应的私钥.

4.2.4 使用 EC 乘法的私钥加解密机制的安全性分析

确认码机制存在一个安全隐患: 仅仅靠 4 个字节的 $addresshash$ 来关联确认码和加密后的密钥是不够的. 一个不诚实的第三方可以提供两个不同的 $factorb$, 并且满足经过 $\mathsf{SHA256}(\mathsf{SHA256}(\cdot))$ 计算后前 4 个字节相同的两个地址. 这时, 仅仅验证确认码是不能够保证用户一定能够计算出他所使用的地址的私钥的. 这也就是说, 确认码并不能起到所声称的作用. 比如, 第三方计算出 $seedb_1$ 以及相应的 $factorb_1$, 并计算出新私钥对应的地址 $addr_1$, $addrhash_1 = \mathsf{SHA256}(\mathsf{SHA256}(addr_1))[0, 1, 2, 3]$. 由于 $addrhash_1$ 只有 4 字节, 第三方可以遍历 $factorb$ 的取值, 直到找到满足 $addrhash_2 = addrhash_1$ 的 $factorb_2$, 计算复杂度为 $O(2^{16})$. 这时, 第三方就可以根据 $seedb_1$ 来计算加密的密钥, 根据 $factorb_2$ 来构造确认码. 由于加密的密钥和确认码中的 $addrhash$ 是相同的, 将确认码和加密的密钥发给用户后, 在用户侧确认码可以验证通过, 如果用户使用计算出的 $addr_2$ 来收款将无法花费相应地址中的资产, 因为他无法计算出 $addr_2$ 对应的私钥 (用户只知道 $factorb_2 * G$ 的值), 他从加密的密钥中计算出来的私钥对应于 $factorb1 * passfactor$, 所以 $addr_2$ 中的

钱将无法被花费.

在新私钥的生成中, 存在一个安全隐患: 在同一个中间口令码下, 如果第三方知道了其中一个加密私钥的明文, 那么它就可以绕开用户的口令, 得到在当前中间口令码下派生的所有私钥. 对于中间口令码 ipc_1, 第三方生成一个随机数 $seedb_1$, 按照方法二计算, 生成的新私钥是 $k_1 = passfactor * factorb_1 \mod n$, $factorb_1 = \mathsf{SHA256}(\mathsf{SHA256}(seedb_1))$. 假如第三方知道了该私钥 k_1, 他就可以计算 $passfactor = k1 * factorb1^{-1} \mod n$, 因此, 在同一个 ipc_1 下的所有私钥都可以被该第三方计算出来. 这种场景是存在的: 如用户委托第三方为自己生成新的加密私钥后, 不小心又委托他对该私钥按照方法一进行加密, 恶意的第三方可以通过对比公钥或地址知道到该私钥是它之前按照方法二生成的, 这时它就具备了作恶的条件.