Abstract

This document describes hierarchical deterministic wallets (or "HD Wallets"): wallets which can be shared partially or entirely with different systems, each with or without the ability to spend coins.

The specification is intended to set a standard for deterministic wallets that can be interchanged between different clients. Although the wallets described here have many features, not all are required by supporting clients.

The specification consists of two parts. In a first part, a system for deriving a tree of keypairs from a single seed is presented. The second part demonstrates how to build a wallet structure on top of such a tree.

Motivation

The Bitcoin reference client uses randomly generated keys. In order to avoid the necessity for a backup after every transaction, (by default) 100 keys are cached in a pool of reserve keys. Still, these wallets are not intended to be shared and used on several systems simultaneously. They support hiding their private keys by using the wallet encrypt feature and not sharing the password, but such "neutered" wallets lose the power to generate public keys as well.

Deterministic wallets do not require such frequent backups, and elliptic curve mathematics permit schemes where one can calculate the public keys without revealing the private keys. This permits for example a webshop business to let its webserver generate fresh addresses (public key hashes) for each order or for each customer, without giving the webserver access to the corresponding private keys (which are required for spending the received funds).

However, deterministic wallets typically consist of a single "chain" of keypairs. The fact that there is only one chain means that sharing a wallet happens on an all-or-nothing basis. However, in some cases one only wants some (public) keys to be shared and recoverable. In the example of a webshop, the webserver does not need access to all public keys of the merchant's wallet; only to those addresses which are used to receive customer's payments, and not for example the change addresses that are generated when the merchant spends money. Hierarchical deterministic wallets allow such selective sharing by supporting multiple keypair chains, derived from a single root.

Specification: Key derivation

Conventions

In the rest of this text we will assume the public key cryptography used in Bitcoin, namely elliptic curve cryptography using the field and curve parameters defined by secp256k1 (http://www.secg.org/sec2-v2.pdf). Variables below are either:

- Integers modulo the order of the curve (referred to as n).
- Coordinates of points on the curve.
- Byte sequences.

Addition (+) of two coordinate pair is defined as application of the EC group operation. Concatenation (||) is the operation of appending one byte sequence onto another.

As standard conversion functions, we assume:

- point(p): returns the coordinate pair resulting from EC point multiplication (repeated application of the EC group operation) of the secp256k1 base point with the integer p.
- $ser_{32}(i)$: serialize a 32-bit unsigned integer i as a 4-byte sequence, most significant byte first.
- $ser_{256}(p)$: serializes the integer p as a 32-byte sequence, most significant byte first.
- $ser_P(P)$: serializes the coordinate pair P = (x,y) as a byte sequence using SEC1's compressed form: (0x02 or 0x03) || $ser_{256}(x)$, where the header byte depends on the parity of the omitted y coordinate.
- $parse_{256}(p)$: interprets a 32-byte sequence as a 256-bit number, most significant byte first.

Extended keys

In what follows, we will define a function that derives a number of child keys from a parent key. In order to prevent these from depending solely on the key itself, we extend both private and public keys first with an extra 256 bits of entropy. This extension, called the chain code, is identical for corresponding private and public keys, and consists of 32 bytes.

We represent an extended private key as (k, c), with k the normal private key, and c the chain code. An extended public key is represented as (K, c), with K = point(k) and c the chain code.

Each extended key has 2^{31} normal child keys, and 2^{31} hardened child keys. Each of these child keys has an index. The normal child keys use indices 0 through 2^{31}-1. The hardened child keys use indices 2^{31} through 2^{32}-1. To ease notation for hardened key indices, a number i_H represents $i+2^{31}$.

Child key derivation (CKD) functions

Given a parent extended key and an index i, it is possible to compute the corresponding child extended key. The algorithm to do so depends on whether the child is a hardened key or not (or, equivalently, whether $i \geq 2^{31}$), and whether we're talking about private or public keys.

Private parent key → private child key

The function CKDpriv((k_{par}, c_{par}), i) → (k_i, c_i) computes a child extended private key from the parent extended private key:

- Check whether $i \geq 2^{31}$ (whether the child is a hardened key).
  - If so (hardened child): let I = HMAC-SHA512(Key = c_{par}, Data = 0x00 || $ser_{256}(k_{par})$ || $ser_{32}(i)$). (Note: The 0x00 pads the private key to make it 33 bytes long.)
  - If not (normal child): let I = HMAC-SHA512(Key = c_{par}, Data = $ser_P(point(k_{par}))$ || $ser_{32}(i)$).
- Split I into two 32-byte sequences, I_L and I_R.
- The returned child key k_i is $parse_{256}(I_L) + k_{par}$ (mod n).
- The returned chain code c_i is I_R.
- In case $parse_{256}(I_L) \geq n$ or $k_i = 0$, the resulting key is invalid, and one should proceed with the next value for i. (Note: this has probability lower than 1 in 2^{127}.)

The HMAC-SHA512 function is specified in RFC 4231.

Public parent key → public child key

The function CKDpub((K_{par}, c_{par}), i) → (K_i, c_i) computes a child extended public key from the parent extended public key. It is only defined for non-hardened child keys.

- Check whether $i \geq 2^{31}$ (whether the child is a hardened key).
  - If so (hardened child): return failure
  - If not (normal child): let I = HMAC-SHA512(Key = c_{par}, Data = $ser_P(K_{par})$ || $ser_{32}(i)$).
- Split I into two 32-byte sequences, I_L and I_R.
- The returned child key K_i is $point(parse_{256}(I_L)) + K_{par}$.
- The returned chain code c_i is I_R.
- In case $parse_{256}(I_L) \geq n$ or K_i is the point at infinity, the resulting key is invalid, and one should proceed with the next value for i.

Private parent key → public child key

The function N((k, c)) → (K, c) computes the extended public key corresponding to an extended private key (the "neutered" version, as it removes the ability to sign transactions).

- The returned key K is point(k).
- The returned chain code c is just the passed chain code.

To compute the public child key of a parent private key:

- N(CKDpriv((k_{par}, c_{par}), i)) (works always).
- CKDpub(N(k_{par}, c_{par}), i) (works only for non-hardened child keys).

The fact that they are equivalent is what makes non-hardened keys useful (one can derive child public keys of a given parent key without knowing any private key), and also what distinguishes them from hardened keys. The reason for not always using non-hardened keys (which are more useful) is security; see further for more information.

Public parent key → private child key

This is not possible.

The key tree

The next step is cascading several CKD constructions to build a tree. We start with one root, the master extended key m. By evaluating CKDpriv(m,i) for several values of i, we get a number of level-1 derived nodes. As each of these is again an extended key, CKDpriv can be applied to those as well.

To shorten notation, we will write CKDpriv(CKDpriv(CKDpriv(m,3_H),2),5) as m/3_H/2/5. Equivalently for public keys, we write CKDpub(CKDpub(CKDpub(M,3),2),5) as M/3/2/5. This results in the following identities:

- N(m/a/b/c) = N(m/a/b)/c = N(m/a)/b/c = N(m)/a/b/c = M/a/b/c.
- N(m/a_H/b/c) = N(m/a_H/b)/c = N(m/a_H)/b/c.

However, N(m/a_H) cannot be rewritten as N(m)/a_H, as the latter is not possible.

Each leaf node in the tree corresponds to an actual key, while the internal nodes correspond to the collections of keys that descend from them. The chain codes of the leaf nodes are ignored, and only their embedded private or public key is relevant. Because of this construction, knowing an extended private key allows reconstruction of all descendant private keys and public keys, and knowing an extended public keys allows reconstruction of all descendant non-hardened public keys.

Key identifiers

Extended keys can be identified by the Hash160 (RIPEMD160 after SHA256) of the serialized ECDSA public key K, ignoring the chain code. This corresponds exactly to the data used in traditional Bitcoin addresses. It is not advised to represent this data in base58 format though, as it may be interpreted as an address that way (and wallet software is not required to accept payment to the chain key itself).

The first 32 bits of the identifier are called the key fingerprint.

Serialization format

Extended public and private keys are serialized as follows:

- 4 byte: version bytes (mainnet: 0x0488B21E public, 0x0488ADE4 private; testnet: 0x043587CF public, 0x04358394 private)
- 1 byte: depth: 0x00 for master nodes, 0x01 for level-1 derived keys, ...
- 4 bytes: the fingerprint of the parent's key (0x00000000 if master key)
- 4 bytes: child number. This is $ser_{32}(i)$ for i in $x_i = x_{par}/i$, with x_i the key being serialized. (0x00000000 if master key)
- 32 bytes: the chain code
- 33 bytes: the public key or private key data ($ser_P(K)$ for public keys, 0x00 || $ser_{256}(k)$ for private keys)

This 78 byte structure can be encoded like other Bitcoin data in Base58, by first adding 32 checksum bits (derived from the double SHA-256 checksum), and then converting to the Base58 representation. This results in a Base58-encoded string of up to 112 characters. Because of the choice of the version bytes, the Base58 representation will start with "xprv" or "xpub" on mainnet, "tprv" or "tpub" on testnet.

Note that the fingerprint of the parent only serves as a fast way to detect parent and child nodes in software, and software must be willing to deal with collisions. Internally, the full 160-bit identifier could be used.

When importing a serialized extended public key, implementations must verify whether the X coordinate in the public key data corresponds to a point on the curve. If not, the extended public key is invalid.

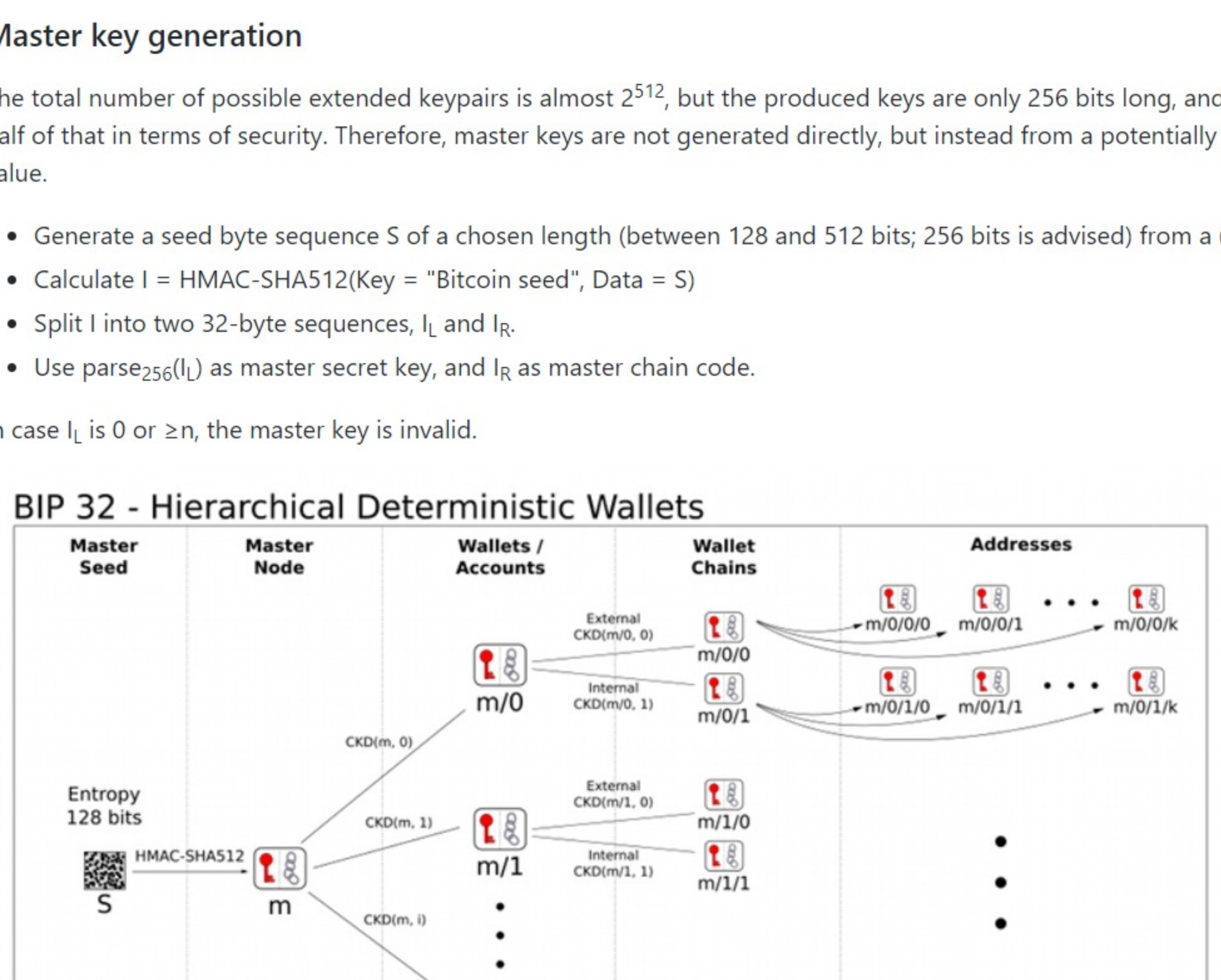
Master key generation

The total number of possible extended keypairs is almost 2^{512}, but the produced keys are only 256 bits long, and offer about half of that in terms of security. Therefore, master keys are not generated directly, but instead from a potentially short seed value.

- Generate a seed byte sequence S of a chosen length (between 128 and 512 bits; 256 bits is advised) from a (P)RNG.
- Calculate I = HMAC-SHA512(Key = "Bitcoin seed", Data = S)
- Split I into two 32-byte sequences, I_L and I_R.
- Use $parse_{256}(I_L)$ as master secret key, and I_R as master chain code.

In case I_L is 0 or $\geq n$, the master key is invalid.

BIP 32 - Hierarchical Deterministic Wallets



Child Key Derivation Function ~ CKD(x,n) = HMAC-SHA512(x_{Chain}, x_{PubKey} || n)

Specification: Wallet structure

The previous sections specified key trees and their nodes. The next step is imposing a wallet structure on this tree. The layout defined in this section is a default only, though clients are encouraged to mimic it for compatibility, even if not all features are supported.

The default wallet layout

An HDW is organized as several 'accounts'. Accounts are numbered, the default account ("") being number 0. Clients are not required to support more than one account - if not, they only use the default account.

Each account is composed of two keypair chains: an internal and an external one. The external keychain is used to generate new public addresses, while the internal keychain is used for all other operations (change addresses, generation addresses, ..., anything that doesn't need to be communicated). Clients that do not support separate keychains for these should use the external one for everything.

- m/i_H/0/k corresponds to the k'th keypair of the external chain of account number i of the HDW derived from master m.
- m/i_H/1/k corresponds to the k'th keypair of the internal chain of account number i of the HDW derived from master m.

Use cases

Full wallet sharing: m

In cases where two systems need to access a single shared wallet, and both need to be able to perform spendings, one needs to share the master private extended key. Nodes can keep a pool of N look-ahead keys cached for external chains, to watch for incoming payments. The look-ahead for internal chains can be very small, as no gaps are to be expected here. An extra look-ahead could be active for the first unused account's chains - triggering the creation of a new account when used. Note that the name of the account will still need to be entered manually and cannot be synchronized via the block chain.

Audits: N(m/*)

In case an auditor needs full access to the list of incoming and outgoing payments, one can share all account public extended keys. This will allow the auditor to see all transactions from and to the wallet, in all accounts, but not a single secret key.

Per-office balances: m/i_H

When a business has several independent offices, they can all use wallets derived from a single master. This will allow the headquarters to maintain a super-wallet that sees all incoming and outgoing transactions of all offices, and even permit moving money between the offices.

Recurrent business-to-business transactions: N(m/i_H/0)

In case two business partners often transfer money, one can use the extended public key for the external chain of a specific account (M/i h/0) as a sort of "super address", allowing frequent transactions that cannot (easily) be associated, but without needing to request a new address for each payment. Such a mechanism could also be used by mining pool operators as variable payout address.

Unsecure money receiver: N(m/i_H/0)

When an unsecure webserver is used to run an e-commerce site, it needs to know public addresses that are used to receive payments. The webserver only needs to know the public extended key of the external chain of a single account. This means someone illegally obtaining access to the webserver can at most see all incoming payments but will not be able to steal the money, will not (trivially) be able to distinguish outgoing transactions, nor be able to see payments received by other webservers if there are several.

Compatibility

To comply with this standard, a client must at least be able to import an extended public or private key, to give access to its direct descendants as wallet keys. The wallet structure (master/account/chain/subchain) presented in the second part of the specification is advisory only, but is suggested as a minimal structure for easy compatibility - even when no separate accounts or distinction between internal and external chains is made. However, implementations may deviate from it for specific needs; more complex applications may call for a more complex tree structure.

Security

In addition to the expectations from the EC public-key cryptography itself:

- Given a public key K, an attacker cannot find the corresponding private key more efficiently than by solving the EC discrete logarithm problem (assumed to require 2^{128} group operations).

the intended security properties of this standard are:

- Given a child extended private key (k_i,c_i) and the integer i, an attacker cannot find the parent private key k_{par} more efficiently than a 2^{256} brute force of HMAC-SHA512.
- Given any number ($2 \leq N \leq 2^{32}-1$) of (index, extended private key) tuples (i_j,(k_{i_j},c_{i_j})), with distinct i_j's, determining whether they are derived from a common parent extended private key (i.e., whether there exists a (k_{par},c_{par}) such that for each j in (0..N-1) CKDpriv((k_{par},c_{par}),i_j)=(k_{i_j},c_{i_j})), cannot be done more efficiently than a 2^{256} brute force of HMAC-SHA512.

Note however that the following properties does not exist:

- Given a parent extended public key (K_{par},c_{par}) and a child public key (K_i), it is hard to find i.
- Given a parent extended public key (K_{par},c_{par}) and a non-hardened child private key (k_i), it is hard to find k_{par}.

Implications

Private and public keys must be kept safe as usual. Leaking a private key means access to coins - leaking a public key can mean loss of privacy.

Somewhat more care must be taken regarding extended keys, as these correspond to an entire (sub)tree of keys.

One weakness that may not be immediately obvious, is that knowledge of a parent extended public key plus any non-hardened private key descending from it is equivalent to knowing the parent extended private key (and thus every private and public key descending from it). This means that extended public keys must be treated more carefully than regular public keys. It is also the reason for the existence of hardened keys, and why they are used for the account level in the tree. This way, a leak of account-specific (or below) private key never risks compromising the master or other accounts.