RECENT CHANGES:

- (16 Apr 2013) Added private derivation for i >= 0x80000000 (less risk of parent private key leakage)
- (30 Apr 2013) Switched from multiplication by I_L to addition of I_L (faster, easier implementation)
- (25 May 2013) Added test vectors

```
BIP: 32
Title: Hierarchical Deterministic Wallets
Author: Pieter Wuille
Status: Accepted
Type: Informational
Created: 11-02-2012
```

Table of Contents

Abstract

This document describes hierarchical determinstic 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/index.php?action=secg,docs_secg). Variables below are either:

- integers modulo the order of the curve (referred to as n), serialized as 32 bytes, most significant byte first.
- coordinates of points on the curve, serialized as specified in SEC1 in compressed form: [0x02] + 32 byte x coordinate), where the header byte depends on the parity of the omitted y coordinate.
- byte sequences

We shall denote the compressed form of point P by χ(P), which for secp256k1 will always be 33 bytes long.

Addition (+) of two coordinates is defined as application of the EC group operation. Multiplication (*) of an integer and a coordinate is defined as repeated application of the EC group operation. The generator element of the curve is called G. The public key K corresponding to the private key k is calculated as k*G. We do not distinguish between numbers or coordinates and their serialization as byte sequences.

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 = k*G and c the chain code.

Child key derivation functions

We allow for two different types of derivation: private and public.

- Private derivation: knowledge of the private key k_{par} and c_{par} is required to compute both k_i and K_i.
- Public derivation: knowledge of the public key K_{par} and c_{par} suffices to compute K_i (but not k_i).

We define the private and public child key derivation functions:

- CKD((k_{par}, c_{par}), i) → (k_i, c_i), defined for both private and public derivation.
- CKD'((K_{par}, c_{par}), i) → (K_i, c_i), defined only for public derivation.

We use the most significant bit of i to specify which type of derivation to use. i is encoded as a 32 bit unsigned integer, most significant byte first; '||' represents concatenation.

Private child key derivation

To define CKD((k_{par}, c_{par}), i) → (k_i, c_i):

- Check whether the highest bit (0x80000000) of i is set:
  - If 1, private derivation is used: let I = HMAC-SHA512(Key = c_{par}, Data = 0x00 || k_{par} || i) [Note:]
  - If 0, public derivation is used: let I = HMAC-SHA512(Key = c_{par}, Data = χ(k_{par}*G) || i)
- Split I = I_L || I_R into two 32-byte sequences, I_L and I_R.
- $k_i = I_L + k_{par}$ (mod n).
- $c_i = I_R$.

In case $I_L \geq n$ or $k_i = 0$, the resulting key is invalid, and one should proceed with the next value for i. [Note:]

Public child key derivation

To define CKD'((K_{par}, c_{par}), i) → (K_i, c_i):

- Check whether the highest bit (0x80000000) of i is set:
  - If 1, return error
  - If 0, let I = HMAC-SHA512(Key = c_{par}, Data = χ(K_{par}) || i)
- Split I = I_L || I_R into two 32-byte sequences, I_L and I_R.
- $K_i = (I_L + k_{par})*G = I_L*G + K_{par}$
- $c_i = I_R$.

In case $I_L \geq n$ or K_i is the point at infinity, the resulting key is invalid, and one should proceed with the next value for i.

Note that the extended public key corresponding to the evaluation of CKD(k_{ext}, i) with public derivation (so with i ext, i), with K_{ext} the extended public key corresponding to k_{ext}. Symbolically, CKD((k, c), i)*G = CKD'((k*G, c), i). This implies that CKD' can be used to derive all public keys corresponding to the private keys that CKD would find. It cannot be used to retrieve the private keys, however.

The HMAC-SHA512 function is specified in RFC 4231.

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 CKD(m,i) for several values of i, we get a number of first-degree derivative nodes. As each of these is again an extended key, CKD can be applied to those as well. To shorten notation, we will write CKD(CKD(CKD(m,0x8000003),0x2),0x5) as m/3'/2/5 now.

Each leaf node in the tree corresponds to an actual keypair, while the internal nodes correspond to the collection of keypairs that descends from them. The chain codes of the leaf nodes are ignored, and only their embedded private or public key is used. 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 public keys derived using public derivation.

Key identifiers

Extended keys can be identified by the Hash160 (RIPEMD160 after SHA256) of the serialized public key, 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 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 descendants, ...
- 4 bytes: the fingerprint of the parent's key (0x00000000 if master key)
- 4 bytes: child number. This is the number i in $x_i = x_{par}/i$, with x_i the key being serialized. This is encoded in MSB order. (0x00000000 if master key)
- 32 bytes: the chain code
- 33 bytes: the public key or private key data (0x02 + X or 0x03 + X for public keys, 0x00 + 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.

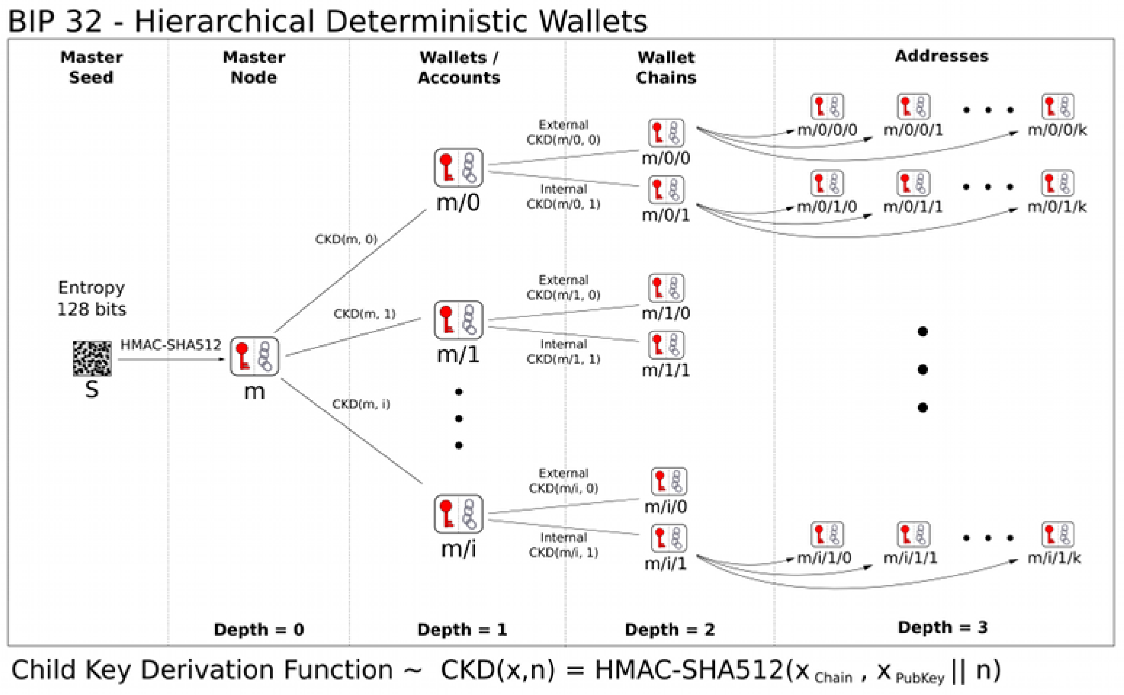
When importing a serialized extended public key, implementations must verify whether the X coordinate in the public key data corresponds to a point on the curve. If not, the extended public key is invalid.

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 S of a chosen length (at least 128 bits, but 256 is advised) from a (P)RNG.
- Calculate I = HMAC-SHA512(key="Bitcoin seed", msg=S)
- Split I into two 32-byte sequences, I_L and I_R.
- Use I_L as master secret key, and I_R as master chain code.

In case I_L is 0 or >=n, the master key is invalid.



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

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 mimick it for compatibility, even if not all features are supported.

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'/0/k corresponds to the k'th keypair of the external chain of account number i of the HDW derived
 * m/i'/1/k corresponds to the k'th keypair of the internal chain of account number i of the HDW derived
```

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: M

In case an auditor needs full access to the list of incoming and outgoing payments, one can share the master public extended key. 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'

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: M/i'/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'/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: M/i'/0

When an unsecure webserver is used to run an e-commerce site, it needs to know public addresses that be 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 (trivially) be able to distinguish outgoing transactions, nor see payments received by other webservers if there are several ones.

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 32-1) of (index, extended private key) tuples $(i_j,(p_j,c_j))$, with distinct i_j's, determining whether they are derived from a common parent extended private key (i.e., whether there exists a (p_{par},c_{par}) such that for each j in [0..N-1] $CKD((p_{par},c_{par}),i_j)=(p_j,c_j)$), 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 child private key (k_i) using public derivation, it is hard to find k_{par}.

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 the extended public key + a private key descending from it is equivalent to knowing the extended private key (i.e., every private and public key) in case public derivation is used. This means that extended public keys must be treated more carefully than regular public keys. This is the reason why accounts at the first level of the default wallet layout use private derivation, so a leak of account-specific (or below) private key never risks compromising the master.

Test Vectors

For more detailed information about these (such as intermediate values and other representations), see BIP_0032_TestVectors

Test vector 1

```
Master (hex): 000102030405060708090a0b0c0d0e0f
* [Chain m]
  * ext pub: xpub661MyMwAqRbcFtXgS5sYJABqqG9YLmC4Q1Rdap9gSE8NqtwybGhePY2gZ29ESFjqJoCu1Rupje8YtGqsefD265
  * ext prv: xprv9s21ZrQH143K3QTDL4LXw2F7HEK3wJUD2nW2nRk4stbPy6cq3jPPqjiChkVvvNKmPGJxWUtg6LnF5kejMRNNU3
* [Chain m/0']
  * ext pub: xpub68Gmy5EdvgibQVfPdqkBBCHxA5htiqg55crXYuXoQRKfDBFA1WEjWgP6LHhwBZeNK1VTsfTFUHCdrfp1bgwQ9x
  * ext prv: xprv9uHRZZhk6KAJC1avXpDAp4MDc3sQKNxDiPvvkX8Br5ngLNv1TxvUxt4cV1rGL5hj6KCesnDYUhd7oWgT11eZG7
* [Chain m/0'/1]
  * ext pub: xpub6ASuArnXKPbfEwhqN6e3mwBcDTgzisQN1wXN9BJcM47sSikHjJf3UFHKkNAWbWMiGj7Wf5uMash7SyYq527Hqc
  * ext prv: xprv9wTYmMFdV23N2TdNG573QoEsfRrWKQgWeibmLntzniatZvR9BmLnvSxqu53Kw1UmYPxLgboyZQaXwTCg8MSY3H
* [Chain m/0'/1/2']
  * ext pub: xpub6D4BDPcP2GT577Vvch3R8wDkScZWzQzMMUm3PWbmWvVJrZwQY4VUNgqFJPMM3No2dFDFGTsxxpG5uJh7n7epu4
  * ext prv: xprv9z4pot5VBttmtdRTWfWQmoH1taj2axGVzFqSb8C9xaxKymcFzXBDptWmT7FwuEzG3ryjH4ktypQSAewRiNMjAN
* [Chain m/0'/1/2'/2]
  * ext pub: xpub6FHa3pjLCk84BayeJxFW2SP4XRrFd1JYnxeLeU8EqN3vDfZmbqBqaGJAyiLjTAwm6ZLRQUMv1ZACTj37sR62cf
  * ext prv: xprvA2JDeKCSNNZky6uBCviVfJSKyQ1mDYahRjijr5idH2WwLsEd4Hsb2Tyh8RfQMuPh7f7RtyzTtdrbdqqsunu5Mm
* [Chain m/0'/1/2'/2/1000000000]
  * ext pub: xpub6H1LXWLaKsWFhvm6RVpEL9P4KfRZSW7abD2ttkWP3SSQvnyA8FSVqNTEcYFgJS2UaFcxupHiYkro49S8yGasTv
  * ext prv: xprvA41z7zogVVwxVSgdKUHDy1SKmdb533PjDz7J6N6mV6uS3ze1ai8FHa8kmHScGpWmj4WggLyQjgPie1rFSruoUi
```

Test vector 2

```
Master (hex): fffcf9f6f3f0edeae7e4e1dedbd8d5d2cfccc9c6c3c0bdbab7b4b1aeaba8a5a29f9c999693908d8a8784817e7
* [Chain m]
  * ext pub: xpub661MyMwAqRbcFW31YEwpkMuc5THy2PSt5bDMsktWQcFF8syAmRUapSCGu8ED9W6oDMSgv6Zz8idoc4a6mr8BDz
  * ext prv: xprv9s21ZrQH143K31xYSDQpPDxsXRTUcvj2iNHm5NUtrGiGG5e2DtALGdso3pGz6ssrdK4PFmM8NSpSBHNqPqm55Q
* [Chain m/0]
  * ext pub: xpub69H7F5d8KSRgmmdJg2KhpAK8SR3DjMwAdkxj3ZuxV27CprR9LgpeyGmXUbC6wb7ERfvrnKZjXoUmmDznezpbZb
  * ext prv: xprv9vHkqa6EV4sPZHYqZznhT2NPtPCjKuDKGY38FBWLvgaDx45zo9WQRUT3dKYnjwih2yJD9mkrocEZXo1ex8G81d
* [Chain m/0/2147483647']
  * ext pub: xpub6ASAVgeehLbnwdqV6UKMHVzgqAG8Gr6riv3Fxxpj8ksbH9ebxaEyBLZ85ySDhKiLDBrQSARLq1uNRts8RuJiHj
  * ext prv: xprv9wSp6B7kry3Vj9m1zSnLvN3xH8RdsPP1Mh7fAaR7aRLcQMKTR2vidYEeEg2mUCTAwCd6vnxVrcjfy2kRgVsFaw
* [Chain m/0/2147483647'/1]
  * ext pub: xpub6DF8uhdarytz3FWdA8TvFSvvAh8dP3283MY7p2V4SeE2wyWmG5mg5EwVvmdMVCQcoNJxGoWaU9DCWh89LojfZ5
  * ext prv: xprv9zFnWC6h2cLgpmSA46vutJzBcfJ8yaJGg8cX1e5StJh45BBciYTRXSd25UEPVuesF9yog62tGAQtHjXajPPdbR
* [Chain m/0/2147483647'/1/2147483646']
  * ext pub: xpub6ERApfZwUNrhLCkDtcHTcxd75RbzS1ed54G1LkBUHQVHQKqhMkhgbmJbZRkrgZw4koxb5JaHWkY4ALHY2grBGR
  * ext prv: xprvA1RpRA33e1JQ7ifknakTFpgNXPmW2YvmhqLQYMmrj4xJXXWYpDPS3xz7iAxn8L39njGVyuoseXzU6rcxFLJ8HF
* [Chain m/0/2147483647'/1/2147483646'/2]
  * ext pub: xpub6FnCn6nSzZAw5Tw7cgR9bi15UV96gLZhjDstkXXxvCLsUXBGXPdSnLFbdpq8p9HmGsApME5hQTZ3emM2rnY5ag
  * ext prv: xprvA2nrNbFZABcdryreWet9Ea4LvTJcGsqrMzxHx98MMrotbir7yrKCEXw7nadnHM8Dq38EGfSh6dqA9QWTyefMLE
```

Implementations

A Python implementation is available at https://github.com/richardkiss/pycoin

A Java implementation is available at https://github.com/bitsofproof/supernode/blob/1.1/api/src/main/java/com/bitsofproof/supernode/api/ExtendedKey.java

A C++ implementation is available at https://github.com/CodeShark/CoinClasses/tree/master/tests/hdwallets

A Ruby implementation is available at https://github.com/wink/money-tree

Acknowledgements

- Gregory Maxwell for the original idea of type-2 deterministic wallets, and many discussions about it.
- Alan Reiner for the implementation of this scheme in Armory, and the suggestions that followed from that.
- Mike Caldwell for the version bytes to obtain human-recognizable Base58 strings.