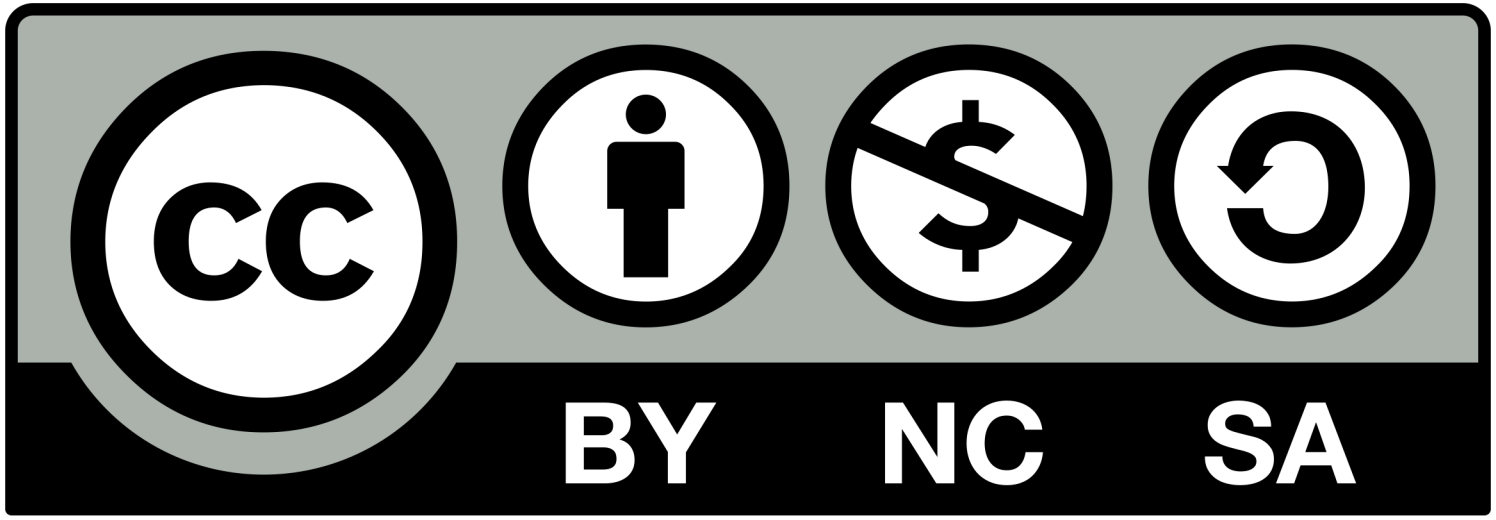
Cryptographic Keys, Addresses, Wallets

CONTROLLING OWNERSHIP OF FUNDS

Stéphane Roche

02-06-2019

CREATIVE COMMONS



ABOUT STEPHANE

2015

Work at Ledger - hardware wallet company

2016–2017

Work on Ethereum

- Learn and play
- Co-found non-profit organization Asseth
- Contribute to the ERC20 Consensys smart contracts
- Dether.io

2017–2019

Found Bitcoin Studio

Focus on Bitcoin education

Consultant at Chainsmiths

https://www.bitcoin-studio.com
@janakaSteph on Twitter
bitcoin-studio@protonmail.com

INTRODUCTION

- « Wallets » in the broad sense
  - Apps to control access to money, manage keys and addresses, track balance, create and signe transactions, etc.

- « Wallets » in the narrow technical sense
  - Data structure used to store and manage user’s keys

- Wallets never contain coins but keys to unlock the coins
  - Users sign tx with the keys, thereby proving they own the tx outputs

- Secure key generation and key management are of utmost importance for the safety of your funds
  - For the end-user: choice of the app wallet
  - For the wallet developer: mastering the cryptography and the Bitcoin standards

- Cryptography can be used for
  - Encryption
  - Proving knowledge of a secret without revealing that secret (digital signature)
  - Prove the authenticity of data (digital fingerprint)

- Used extensively in bitcoin to *control ownership of funds*, in the form of keys, addresses, and wallets

- Encryption is not an important part of bitcoin
  - Communications and transaction data are not encrypted and do not need to be encrypted to protect the funds

OUTLINE

1. Public key cryptography in Bitcoin
2. Mnemonic and root seed
3. HD wallet key derivation
4. HD wallet structure
5. Bitcoin addresses (types, encoding)

1 PUBLIC KEY CRYPTOGRAPHY IN BITCOIN

- ECDSA key pair to control access to bitcoin
  - Public key used to receive funds
  - Private key used to digitally sign transactions, in order to spend the funds
    - The signature can be validated against the public key without revealing the private key

- Keys are completely independent of the bitcoin protocol
  - They can be generated and managed by the user's wallet software without reference to the blockchain or access to the internet

- PKC enables many of the interesting properties of bitcoin
  - Decentralized trust and control
  - Ownership attestation
  - Cryptographic-proof security model
  - Open and permissionless system

- Bitcoin uses a specific EC and set of mathematical constants known as *secp256k1*
  - $y^2 = x^3 + ax + b$ where a = 0 and b = 7
  - Recommended by the Standards for Efficient Cryptography Group; not a NIST curve
  - Unlike popular NIST curves, secp256k1's constants were selected in a predictable way, which significantly reduces the possibility of backdoor

- Signature malleability issue
  - ECDSA is not Strong Existential Unforgeability against Adaptive Chosen Message Attack (SEUF-CMA)
    - Existential Unforgeability: hard to forge a signature on a new message
  - Fixed by a policy rule requiring a canonical « low-s » encoding (Bitcoin PR #6769)

PRIVATE KEY

- A private key is simply a number between 1 and 2^{256}, picked at random
  - The range of valid private keys is governed by the secp256k1 ECDSA standard
  - Need secure source of entropy (CSPRNG)

- *1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD*
  - 256 bits shown as 64 hexadecimal digits, each 4 bits
  - Approximately 10^{77} in decimal
    - For comparison, the visible universe is estimated to contain 10^{80} atoms

PUBLIC KEY

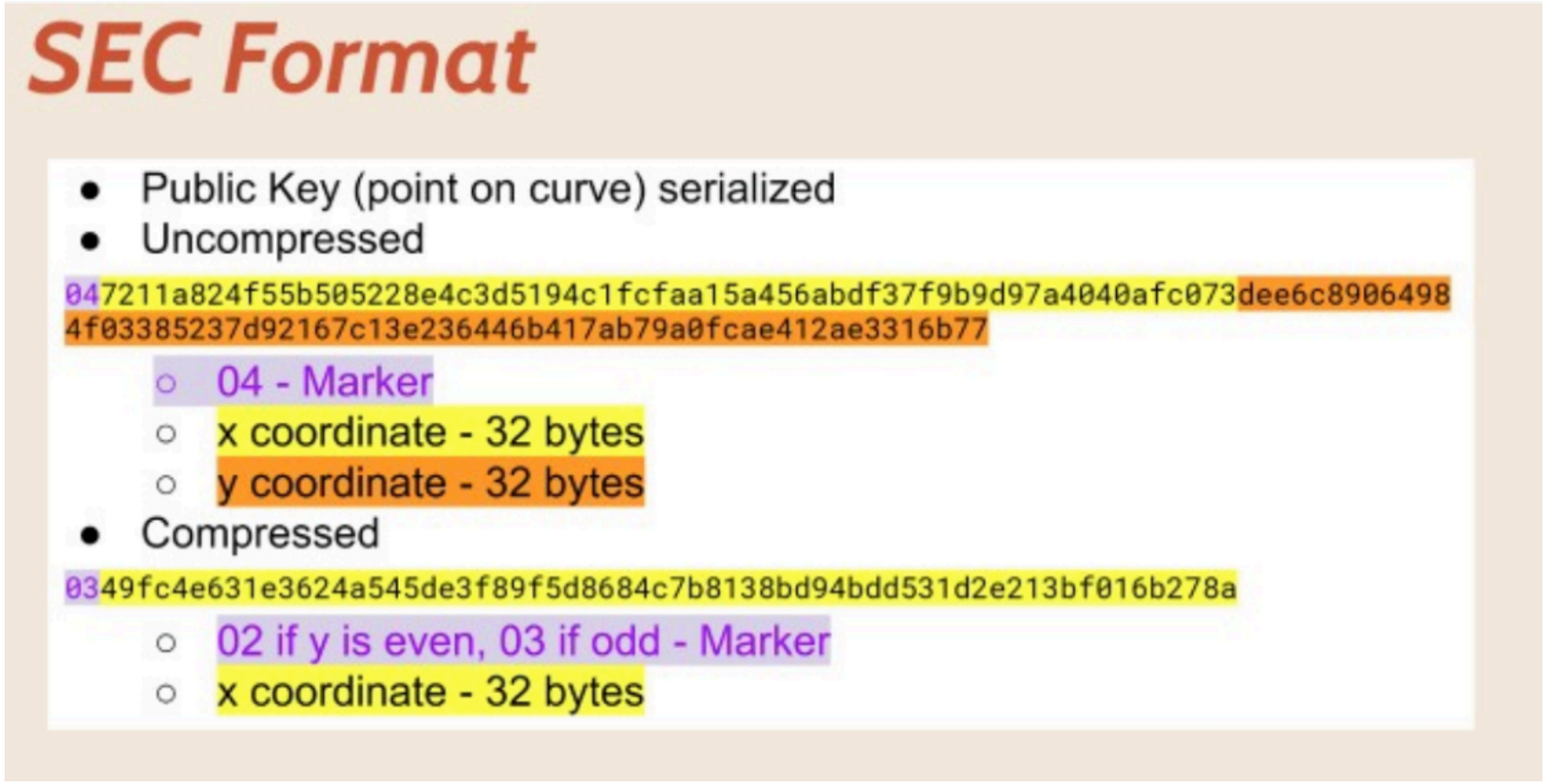
- The public key is calculated from the private key using EC multiplication
  - $K = k * G$
  - EC multiplication is irreversible (one-way function)

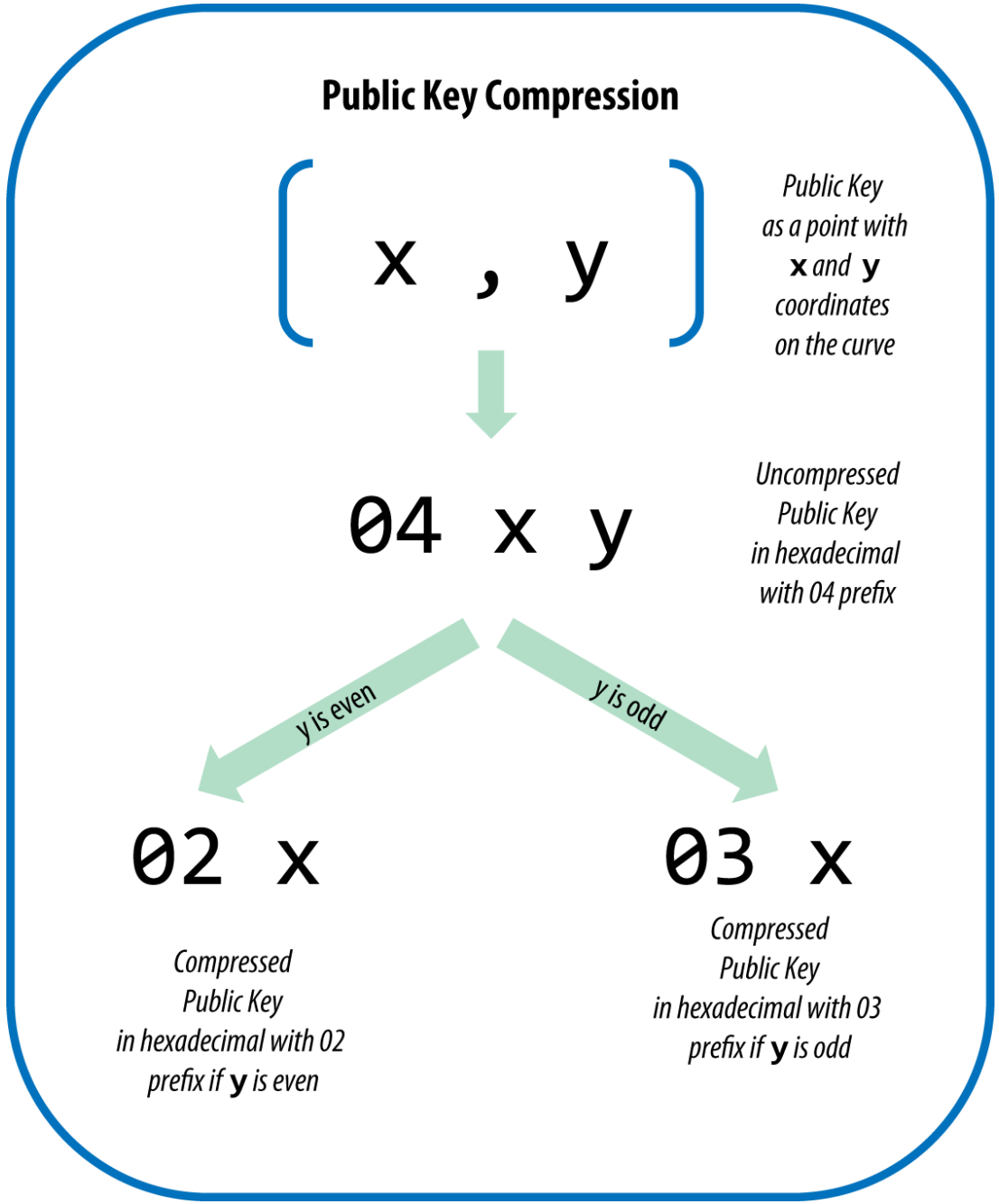
- A public key is a coordinate point (x,y) on an elliptic curve
  - With a P-256 curve, pubKey will be represented as two values in the 256-bit field

- Curve symmetry about a horizontal axis makes compression possible
  - Plotting a curve defined over a finite field yields two points for every value of x
  - One of these points will have an even y–value, and the other an odd y–value

PUBLIC KEY COMPRESSION

- SEC defines serialization formats (uncompressed and compressed)
  - Compressed pubKey is 33 bytes instead of 65




Public Key Compression
[x , y]
Public Key as a point with x and y coordinates on the curve
04 x y
Uncompressed Public Key in hexadecimal with 04 prefix
y is even
y is odd
02 x
Compressed Public Key in hexadecimal with 02 prefix if y is even
03 x
Compressed Public Key in hexadecimal with 03 prefix if y is odd

WIF FORMAT

- Wallet Import Format
  - A way of encoding a private key so as to make it easier to copy and export

- Two different pubKey formats (compressed and uncompressed) produces two different *valid* bitcoin addresses
  - So wallets need to know if the privKey has been used to produce compressed or uncompressed public keys
  - Add a 0x01 byte => compressed public key
  - WIF / WIF-compressed
  - BIP178 in discussion to integrate output type in the WIF

- Private keys are not themselves compressed and cannot be compressed

PRIVATE KEY FORMATS

Type	Prefix	Description
Raw	None	32 bytes
Hex	None	64 hexadecimal digits
WIF	5	Base58Check encoding: Base58 with version prefix of 128- and 32-bit checksum
WIF-compressed	K or L	As above, with added suffix 0x01 before encoding

MINI PRIVATE KEY FORMAT

- Method of encoding a Bitcoin private key in as few as 30 characters for the purpose of being embedded in a small space
- The first of the characters is always the uppercase letter S
- Casascius Series 1 holograms use a 22-character variant of the minikey format, instead of 30 characters
- To determine whether the minikey is valid
  - Add a question mark to the end of the mini private key string
  - Take the SHA256 hash of the entire string
  - If the first byte is 00, the string is a well-formed minikey
- Example: SzavMBLoXU6kDrqtUVmffv

BIP-38 - PASSPHRASE-PROTECTED PRIVATE KEY

- Proposed by Mike Caldwell aka Casascius in 2012

- Intended for use on paper wallets and physical Bitcoins
  - Encrypt a key with a passphrase
  - Delegate key and address *creation* to an untrusted peer

- Encrypting a private key without EC multiplication
  - Offers the advantage that any known private key can be encrypted
  - The party performing the encryption must know the passphrase

- Encrypting a private key with EC multiplication
  - The idea is to generate an *intermediate code*. With this *intermediate code*, a third party will be able to generate encrypted keys on the user behalf, without knowing password nor any clear private key
  - Only the person who knows the original passphrase can decrypt the private key
  - Does not offer the ability to encrypt a known private key

- Uses salting and the KDF Scrypt to resist brute-force attacks

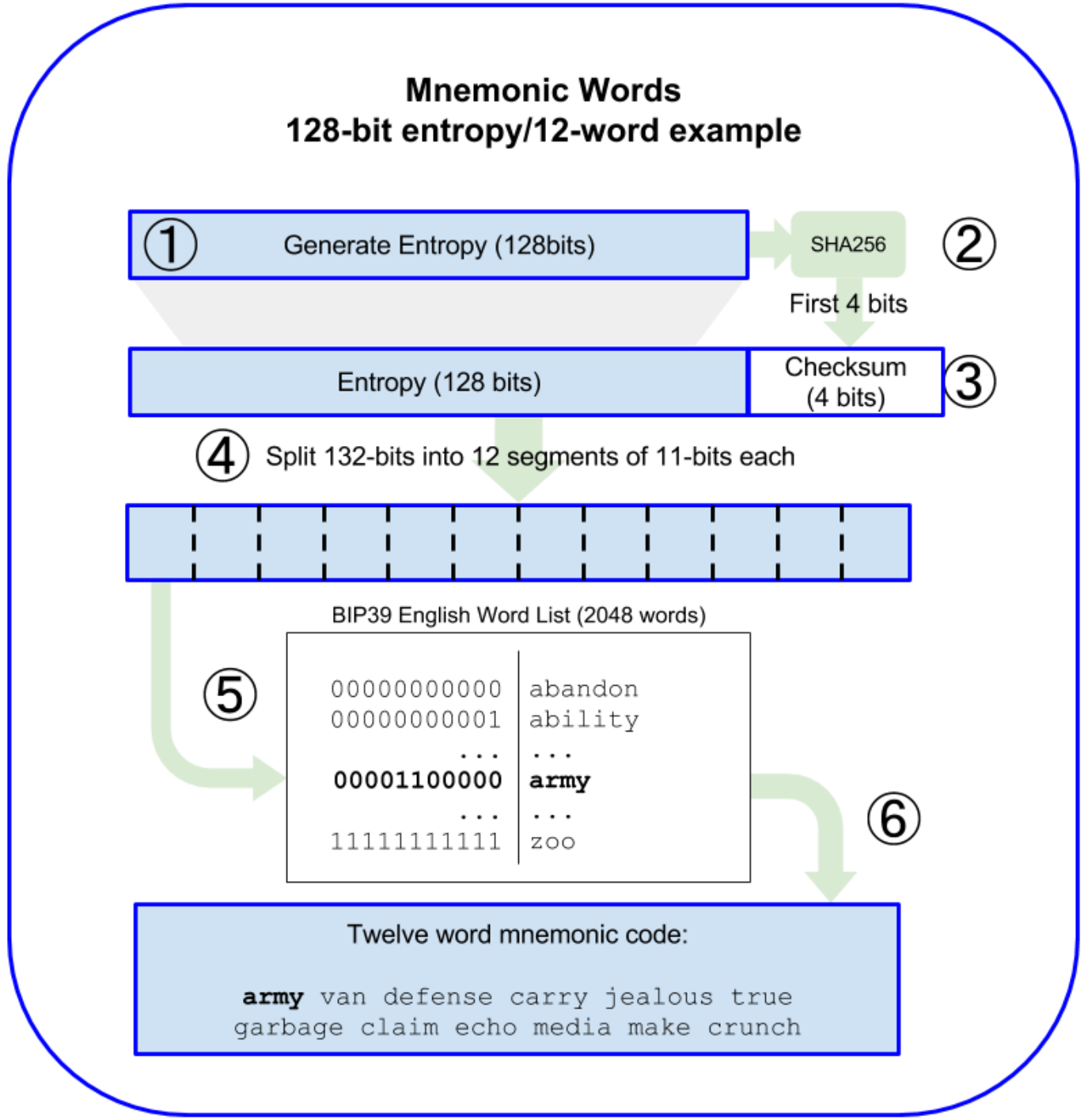
2

MNEMONIC AND ROOT SEED

- BIP-39 – Mnemonic code for generating deterministic keys
  - Describes how to generates a mnemonic code
  - Describes how to converts the mnemonic to a binary seed
  - Computer-generated randomness
  - Easy to back up and transport

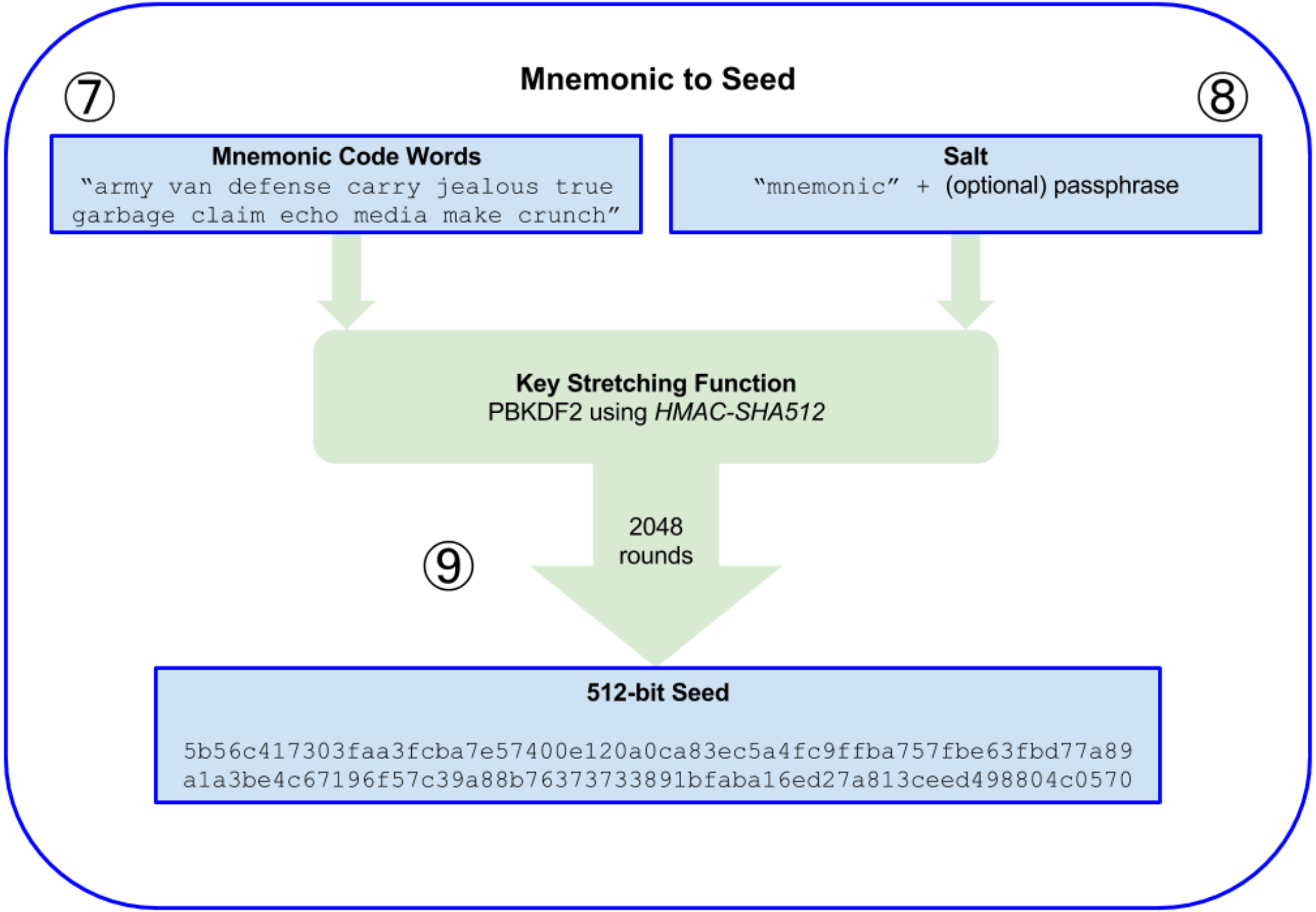
- Mnemonic are more practical than private keys
  - No need to back up regularly

- User-created sentences (aka brainwallets) are less secure
  - WarpWallet brainwallet: PBKDF2(password/email) ⊕ Scrypt(password/email)
    - Quite ok if user uses a unique password
    - But no BIP32 derivation, just a keypair
  - Nowallet in pre-alpha uses the same algorithm, with BIP32/44 support


Mnemonic Words
128-bit entropy/12-word example
1
Generate Entropy (128bits)
SHA256
2
First 4 bits
Entropy (128 bits)
Checksum
(4 bits)
3
4
Split 132-bits into 12 segments of 11-bits each
BIP39 English Word List (2048 words)
5
00000000000 abandon
00000000001 ability
... ...
00001100000 army
... ...
11111111111 zoo
6
Twelve word mnemonic code:
army van defense carry jealous true
garbage claim echo media make crunch

BIP39 WORDLISTS

- Available at https://github.com/bitcoin/bips/blob/master/bip-0039/bip-0039-wordlists.md
- 2048 natural language words
  - $(2047)_{10} = (11111111111)_2$ => 11-bits
- English, Japanese, Korean, Spanish, Chinese, French and Italian
- It's enough to type the first four letters to unambiguously identify the word
- Similar words are avoided
- Native characters must be encoded in UTF-8 using Normalization Form Compatibility Decomposition (NFKD)


Mnemonic to Seed
⑦
⑧
Mnemonic Code Words
"army van defense carry jealous true garbage claim echo media make crunch"
Salt
"mnemonic" + (optional) passphrase
Key Stretching Function
PBKDF2 using HMAC-SHA512
⑨
2048
rounds
512-bit Seed
5b56c417303faa3fcba7e57400e120a0ca83ec5a4fc9ffba757fbe63fbd77a89
a1a3be4c67196f57c39a88b76373733891bfaba16ed27a813ceed498804c0570

BIP39 MNEMONIC KEY STRETCHING

- PBKDF2
  - Applies a pseudorandom function (HMAC-SHA512) to *mnemonic* + *password*
  - Repeats the process many times to produce a *seed*
  - 2048 rounds is weak
    - This is equivalent to adding an extra 11 bits of security to the seed ($2048 = 2^{11}$)
    - BIP39 was made to work on hardware wallet (proposed by the Trezor team)
    - Not a problem as long as mnemonic is generated from CSPRNG

- Password (salt) optionally added to the mnemonic
  - Reduces the ability to use precomputed hashes (rainbow tables) for attacks
  - Provides *plausible deniability*, because every password generates a valid seed
    - Do not overestimate your ability to remember this password

Entropy (bits)	Checksum (bits)	Entropy + checksum (bits)	Mnemonic length (words)
128	4	132	12
160	5	165	15
192	6	198	18
224	7	231	21
256	8	264	24

ELECTRUM SEED VERSION SYSTEM

- Encode with a version number, directly into the seed, how to derive the keys
  - Pro: wallets don’t have to scan different keychains. Treat P2PKH, P2SH-P2WPKH, Bech32 as separate keychains
  - Drawback: you need to generate a new seed every time you want to use it for something new

- Electrum seed generation doesn’t follows BIP39
  - The method is not formalized in a BIP

- During seed generation, mnemonic is hashed until the hash begins with the correct *version number* prefix
  - Achieved by enumerating a nonce and re-hashing the seed phrase until the desired version number is created
  - The version number is also used to check seed integrity; in order to be correct, a seed phrase must produce a registered version number

ELECTRUM LIST OF RESERVED NUMBERS

Number	Type	Description
0x01	Standard	P2PKH and Multisig P2SH wallets
0x100	Segwit	Segwit: P2WPKH and P2WSH wallets
0x101	2FA	Two-factor authenticated wallets

3

HD WALLET KEY DERIVATION

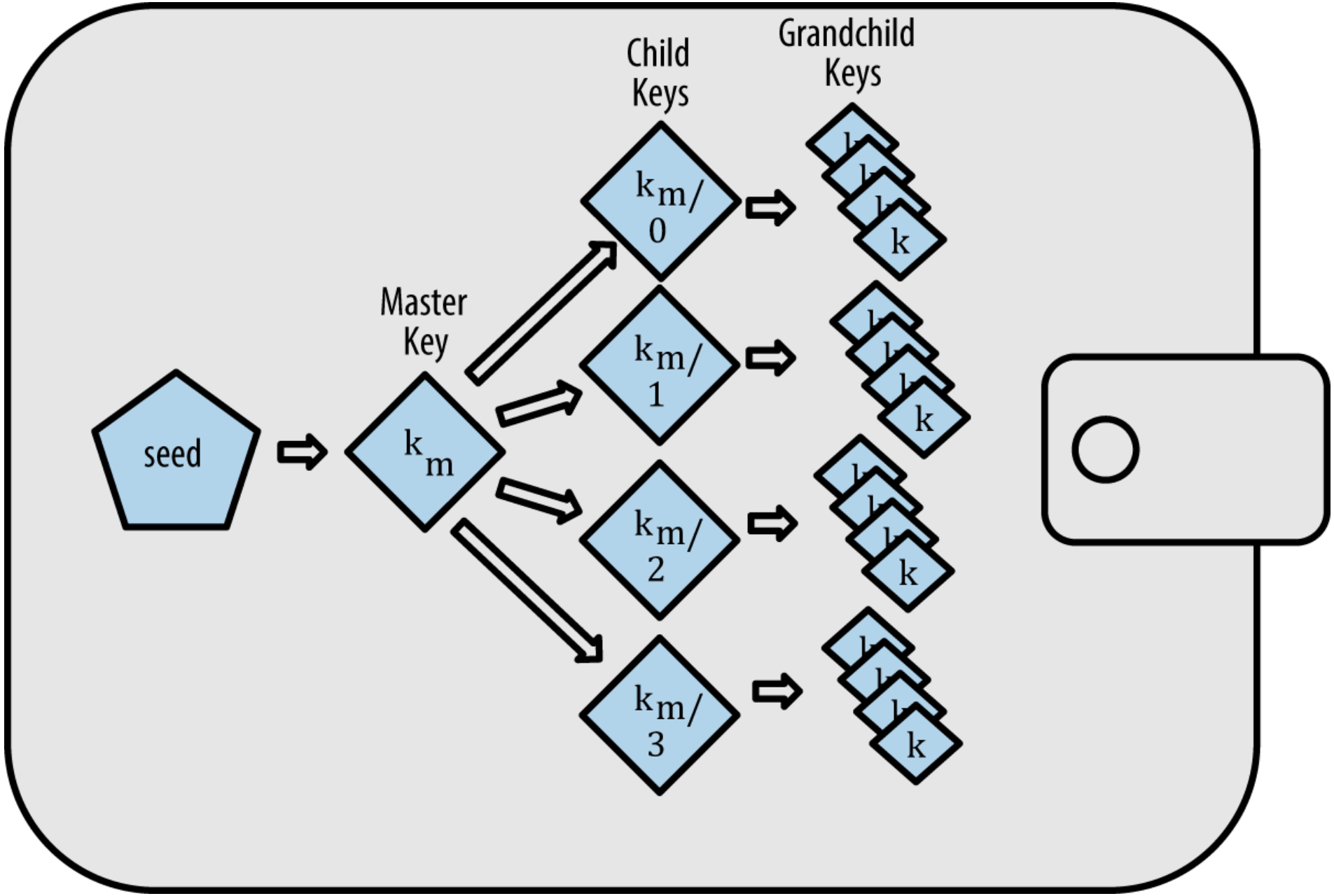
BIP32 - HIERARCHICAL DETERMINISTIC WALLETS

- BIP32 defines a framework for HD wallet (and suggests a structure)

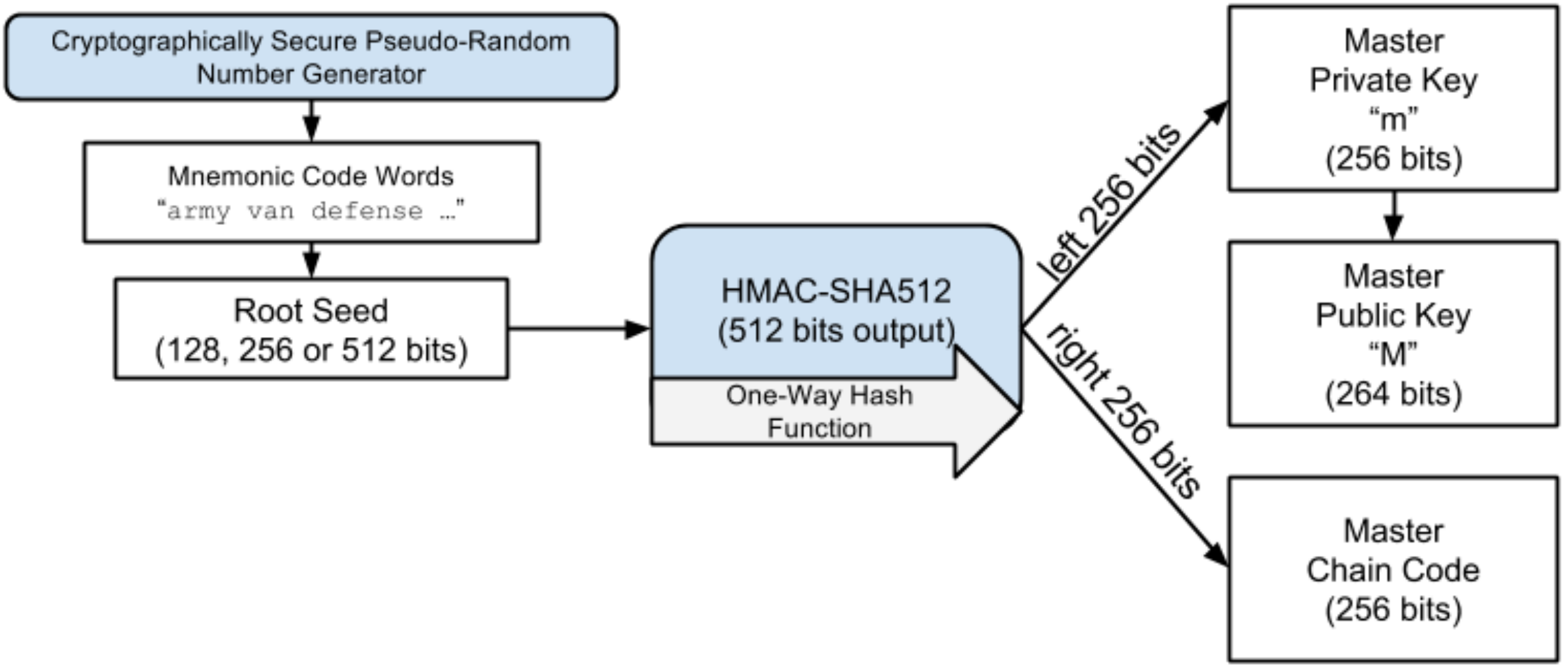
- System for deriving a tree of keypairs from a single seed
  - Tree-like structure => hierarchical
  - The tree structure can be used to express additional organizational meaning

- All the keys are derived from a single master key, known as the seed
  - Keys are related to each other
  - Can be generated again if one has the original seed => deterministic
  - Prevent outdated backups

- Wallets can be shared partially or entirely with different systems, each with or without the ability to spend coins


Child
Keys
Grandchild
Keys
Master
Key
seed
k_m
$k_{m/0}$
$k_{m/1}$
$k_{m/2}$
$k_{m/3}$
k

CREATING MASTER KEYS AND CHAIN CODE FROM A ROOT SEED



BIP32 CHILD KEY DERIVATION FUNCTIONS

- Given a parent extended key and an index i, it is possible to compute the corresponding child extended key

- Parent private key → Child private key
  - CKDpriv((k_{par}, c_{par}), i) → (k_i, c_i)
  - Computes a child XPRIV from the parent XPRIV
  - Check whether $i \geq 2^{31}$ (whether the child is a hardened key)
    - If so HMAC-SHA512(chain code, k, i)
    - If not HMAC-SHA512(chain code, point(k), i)

- Parent public key → Child public key
  - CKDpub((K_{par}, c_{par}), i) → (K_i, c_i)
  - Computes a child XPUB from the parent XPUB
  - Only defined for non-hardened child keys (return failure if $i \geq 2^{31}$)

- Parent private key (neutered) → Child public key
  - N((k, c)) → (K, c)
  - Computes the XPUB from the corresponding XPRIV

- The next step is cascading several CKD constructions to build a tree

- CKDpriv(CKDpriv(CKDpriv(m,3_H),2),5) == m/3_H/2/5
- CKDpub(CKDpub(CKDpub(M,3),2),5) == M/3/2/5

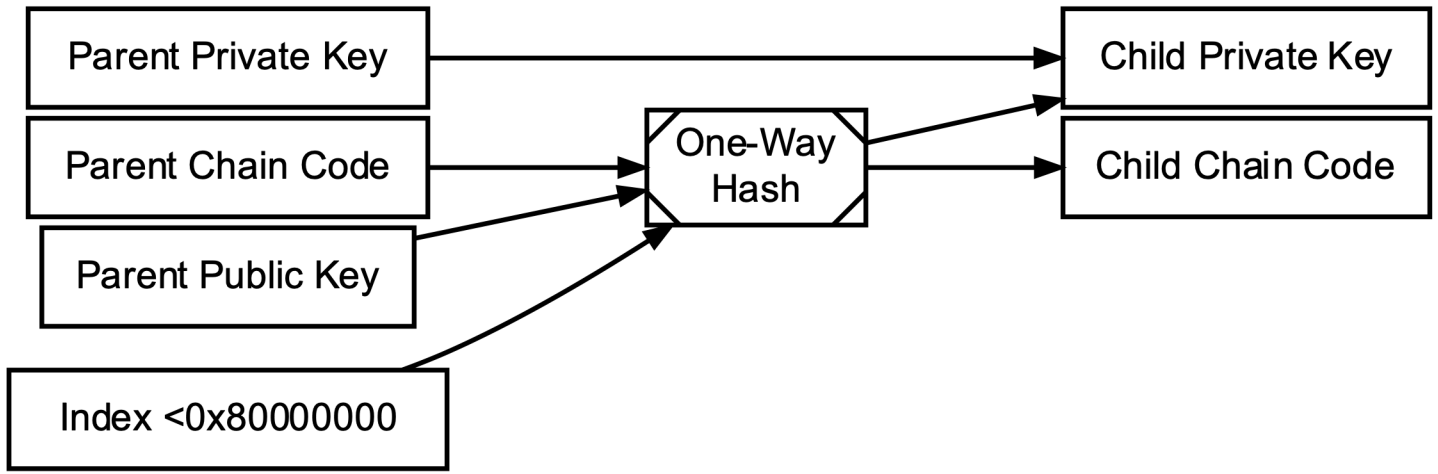
- Each leaf node in the tree corresponds to an actual key, while the internal nodes correspond to the collections of keys that descend from them

- The chain codes of the leaf nodes are ignored, and only their embedded private or public key is relevant

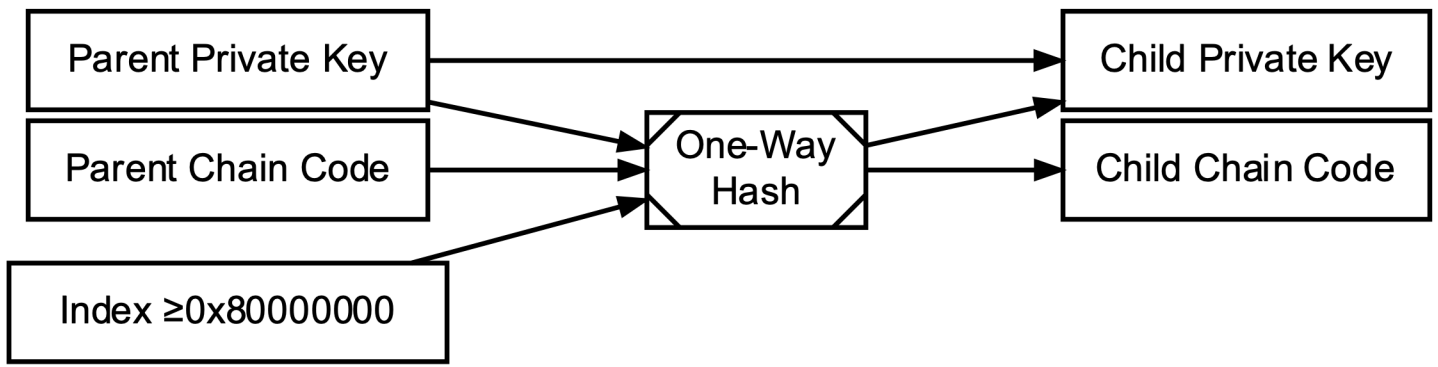
- Because of this construction
  - Knowing an XPRIV allows reconstruction of all descendant private keys and public keys
  - Knowing an XPUB allows reconstruction of all descendant non-hardened public keys

CHILD PRIVATE KEY DERIVATION

Normal CKDpriv

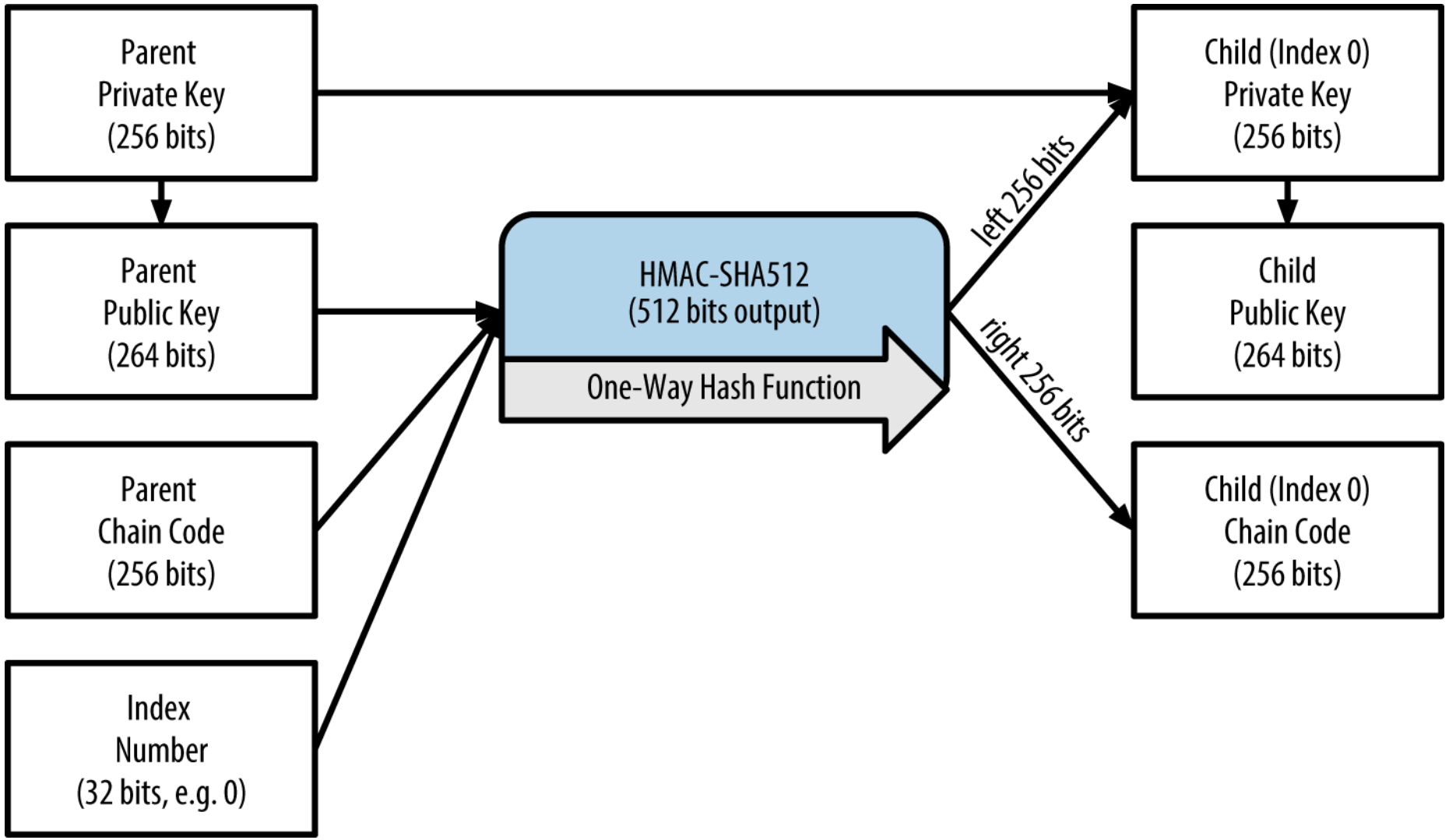


Hardened CKDpriv

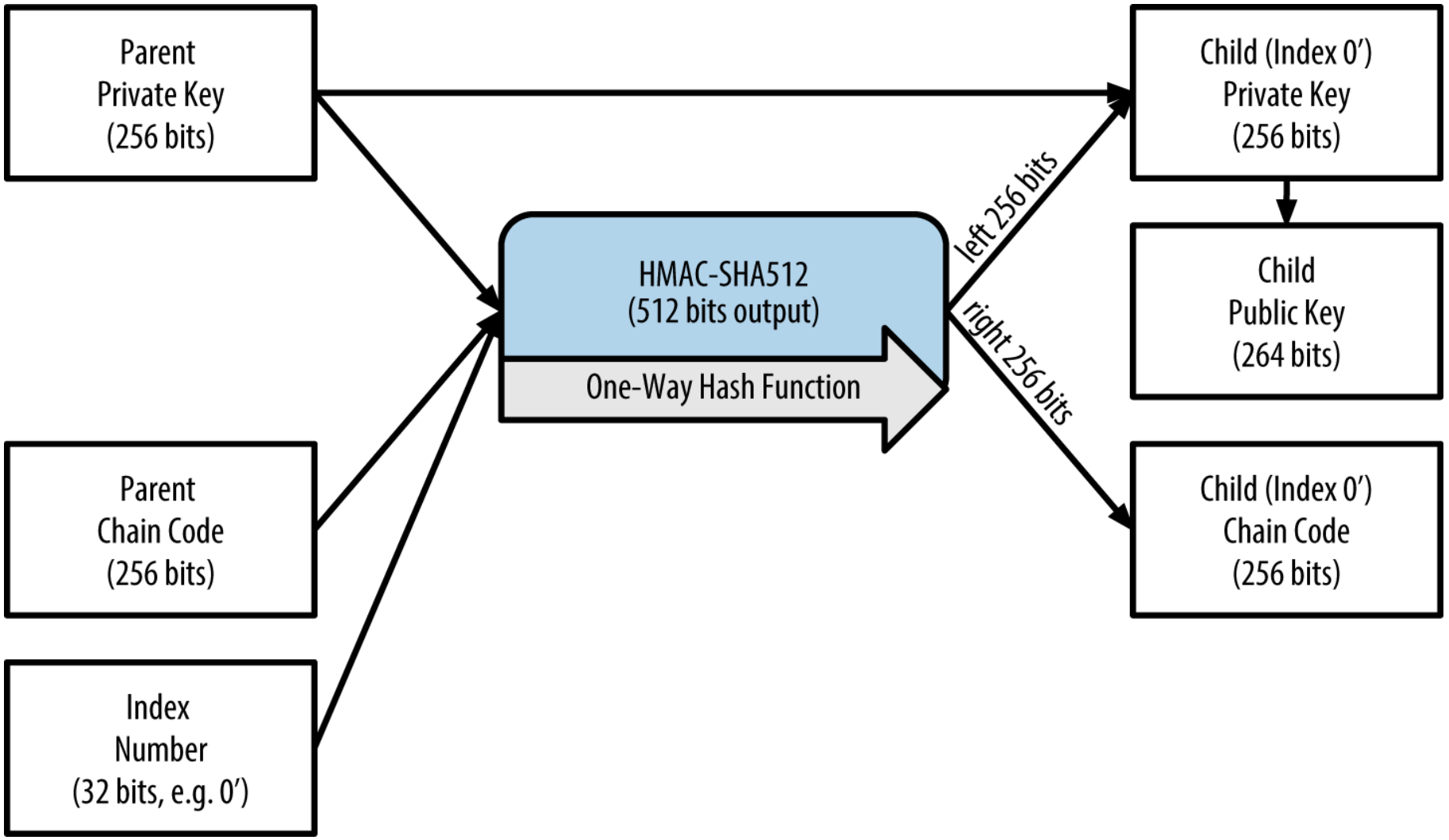


Norm

NORMAL CHILD PRIVATE KEY DERIVATION



HARDENED DERIVATION OF A CHILD KEY



HARDENED KEYS

- Non-hardened child key
  - Hash(parent public key + chain code + index 0 through 2^31-1)
  - An attacker who compromises a private key can climb the derivation path by combining it with an non-hardened XPUB chain code

- Hardened child key
  - Hash(parent private key + chain code + index 2^31 through 2^32-1)
  - Hardened XPRIV creates a firewall
    - Multi-level (cross-generational) key derivation compromises cannot happen
  - But *public derivation* from M is no longer possible

EXTENDED KEY

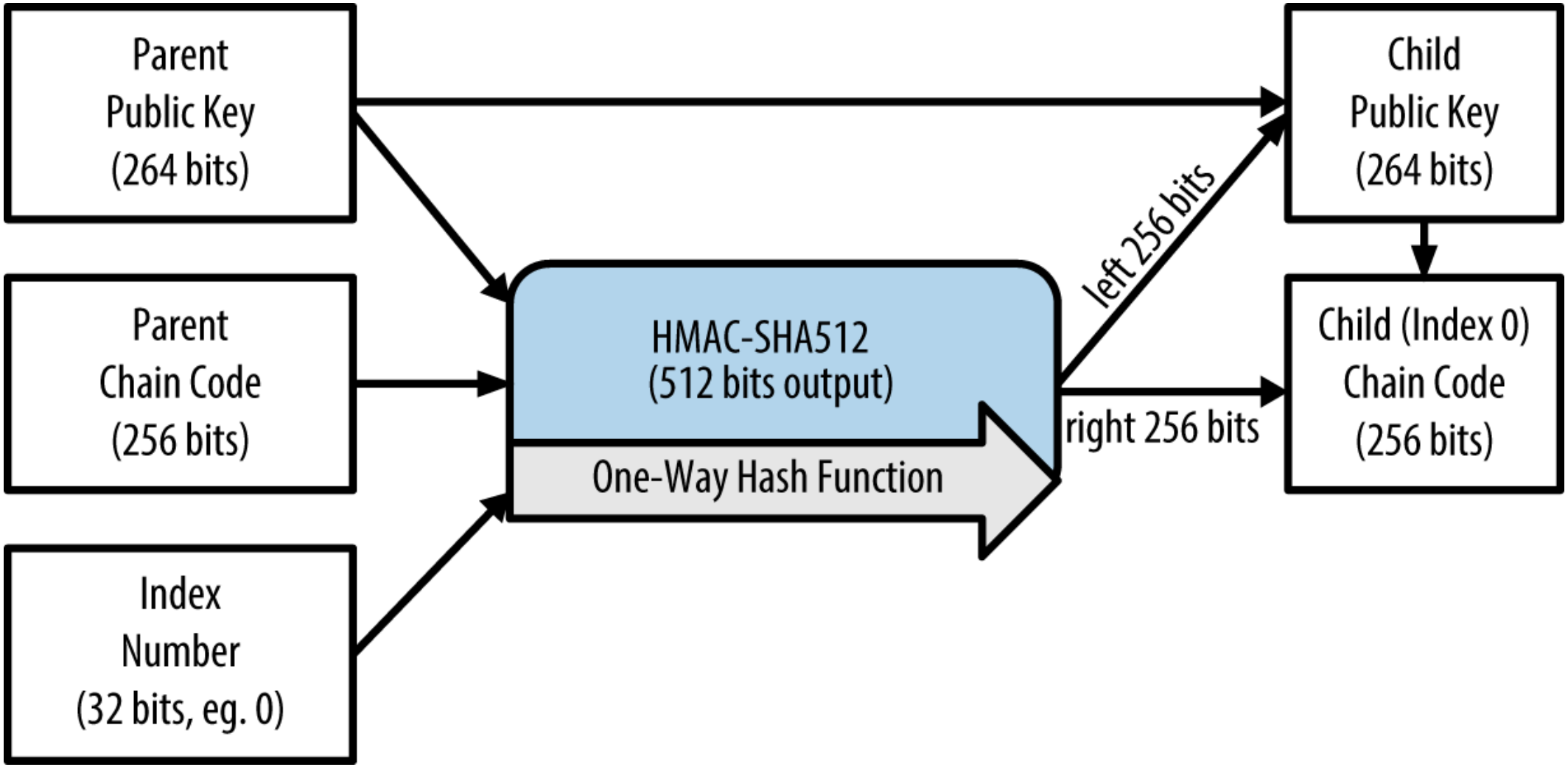
- Extended key = key (256-bit) || chain code (256-bit)
  - Serialization format: [magic][depth][parent fingerprint][key index][chain code][key]

- The term could also be thought of as an "extension key"
  - Such a key can be used to derive children
  - Root of a branch in the tree structure of the HD wallet
  - Sharing an extended key gives access to the entire branch

- XPUB
  - XPUB can derive non-hardened child public keys (only)
    - Can create a branch of public keys only
    - Allows HD wallets to be used on an insecure server or in a receive-only capacity
  - XPUB + parent private key can derive non-hardened child private keys

- XPRIV
  - XPRIV can derive non-hardened and hardened keys

CHILD PUBLIC KEY DERIVATION DIRECTLY FROM XPUB

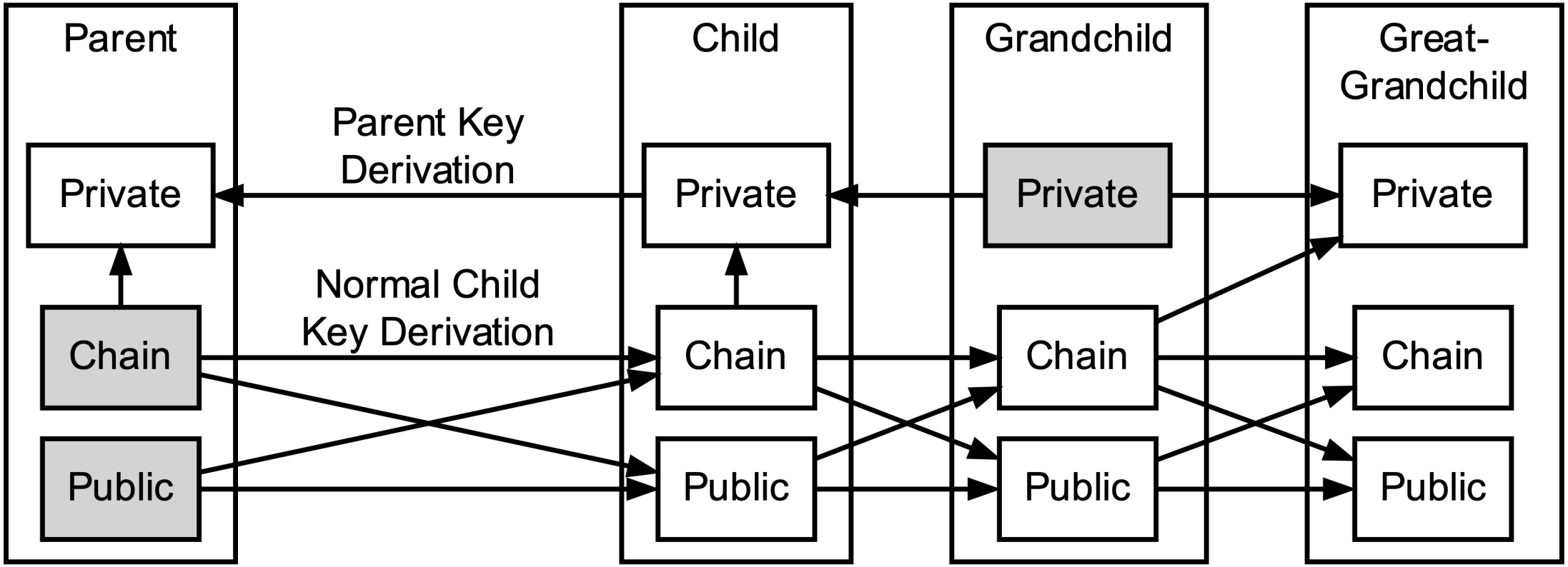


- Because the xpub contains the chain code, if a child private key is leaked, it can be used with the chain code to derive all the other child private keys
  - A single leaked child private key, together with a parent chain code, reveals all the private keys of all the children
  - Worse, the child private key together with a parent chain code can be used to deduce the parent private key

- Hardened derivation "breaks" the relationship between parent public key and child chain code
  - Creates a "firewall" in the parent/child sequence
  - Chain code cannot be used to compromise a parent or sibling private key
  - The resulting "branch" of keys can be used to produce extended public keys that are not vulnerable, because the chain code they contain cannot be exploited to reveal any private keys

- XPUB must be treated more carefully than regular public keys

CROSS-GENERATIONAL KEY COMPROMISE



SLIP132 - VERSION BYTES FOR BIP32 EXTENDED KEYS

- Currently hot debate on the bitcoin-dev mailing-list and on github issues
- Used first on Electrum wallet, followed by Trezor and others
- With the activation of SegWit, the number of ways of encoding an address public key has increased
- Use BIP32 version bytes to also encode the type of output scripts a wallet should derive along the HD subtree
  - Wallet program would not have to guess the address space
  - Give an indication to the user which derivation scheme is implemented

SLIP132 - REGISTERED HD VERSION BYTES

Coin	Public Key	Private Key	Address Encoding
Bitcoin	0x0488b21e - xpub	0x0488ade4 - xprv	P2PKH or P2SH
Bitcoin	0x049d7cb2 - ypub	0x049d7878 - yprv	P2WPKH in P2SH
Bitcoin	0x0295b43f - Ypub	0x0295b005 - Yprv	P2WSH in P2SH
Bitcoin	0x04b24746 - zpub	0x04b2430c - zprv	P2WPKH
Bitcoin	0x02aa7ed3 - Zpub	0x02aa7a99 - Zprv	P2WSH

4

HD WALLET STRUCTURE

HD WALLET STRUCTURE

- HD wallet structure is defined by a BIP32 *path*
  - Each level in that path have a special meaning
  - Each level splits the key space
- BIP32 also suggest a wallet structure (used by Bitcoin Core)
- BIP43 suggests a purpose field for deterministic wallets
- BIP44 is the Bitcoin standard
  - Also became the de-facto standard for a lot of blockchains
- BIP47 for reusable payment codes for HD wallet
- BIP49 for Segwit wallets (P2SH-P2WPKH)
- BIP84 for native Segwit wallets (P2WPKH)

BIP43 - PURPOSE FIELD FOR DETERMINISTIC WALLETS

- BIP43 suggests that wallet software will try various existing derivation schemes within the BIP32 framework
  - Or user need to move their coins (cf. BIP44 => BIP49)

- Rests on the assumption that future wallets will support all previously accepted derivation methods

- Having to explore the branches of the BIP32 tree in order to determine the type of wallet attached to a seed might be somewhat inefficient

BIP44 - MULTI-ACCOUNT HIERARCHY FOR DETERMINISTIC WALLETS

- Defines a logical hierarchy for deterministic wallets
  - Based on an algorithm described in BIP32

- Allows the handling of multiple coins, multiple accounts, external and internal chains per account

- m / purpose' / coin_type' / account' / change / address_index
  - Apostrophe indicates that BIP32 hardened derivation is used

- Purpose
  - Constant set to 44’
  - Follows the BIP43 recommendation
  - Indicates that the subtree of this node is used according to BIP44 specification
  - Hardened derivation

- Coin type
  - Constant defined in SLIP44
  - Bitcoin: 0, Ether: 60, Waves: 5741564, etc.
  - Creates a separate subtree for every cryptocoin
  - Avoids reusing addresses across cryptocoins, improving privacy issues
  - Hardened derivation

- Account
  - Index (used as child index in BIP32 derivation)
  - To organize the funds in different purposes (donation, savings, common expenses, ...)
  - Hardened derivation

- Change
  - Constant 0 for external keychain
    - Used to generate new public addresses, for receiving payments
    - Scan during account discovery, gap limit of 20 addr
  - Constant 1 for internal keychain
    - Addresses for change from associated external chain
    - Addresses are not communicated
  - Public derivation is used at this level

- Index
  - Index, used as child index in BIP32 derivation
  - Public derivation is used at this level

BIP32 PATH EXAMPLES

coin	account	chain	address	path
Bitcoin	first	external	first	m / 44' / 0' / 0' / 0 / 0
Bitcoin	first	external	second	m / 44' / 0' / 0' / 0 / 1
Bitcoin	first	change	first	m / 44' / 0' / 0' / 1 / 0
Bitcoin	first	change	second	m / 44' / 0' / 0' / 1 / 1
Bitcoin	second	external	first	m / 44' / 0' / 1' / 0 / 0
Bitcoin	second	external	second	m / 44' / 0' / 1' / 0 / 1
Bitcoin	second	change	first	m / 44' / 0' / 1' / 1 / 0
Bitcoin	second	change	second	m / 44' / 0' / 1' / 1 / 1

BIP47 - REUSABLE PAYMENT CODES FOR HD WALLET

- BIP47 protocol extends BIP44 HD wallet
  - Yet coins that were received to a Payment Code address can only be accessed using a BIP47 compatible wallet

- Payment codes add identity information to transactions
  - An extended public key and associated metadata which is associated with a particular identity/account
  - Can be publicly advertised

- Provides the privacy benefits of Darkwallet-style *Stealth Addresses* to SPV clients
- PayNym Bot: visual avatar uniquely generated from a hash of your BIP47 reusable payment code
- Be careful, BIP still controversial, some people say it is not well designed enough

- m / purpose' / coin_type' / identity' / 0
  - The 0th (non-hardened) child is the notification key

- m / purpose' / coin_type' / identity' / 0 through 2147483647
  - These (non-hardened) keypairs are used for ECDH to generate deposit addresses

- m / purpose' / coin_type' / identity' / 0' through 2147483647'
  - These (hardened) keypairs are ephemeral payment codes

BIP49 - DERIVATION SCHEME FOR P2WPKH-NESTED-IN-P2SH BASED ACCOUNTS

- P2SH-P2WPKH is the temporary format for Segwit (used for smooth transition, wallet compatibility)

- BIP49 defines the derivation scheme for HD wallets using the P2SH-P2WPKH (BIP 141) serialization format for segwit transactions

- User needs to create dedicated P2SH-P2WPKH segwit wallet
  - Ensures that only BIP49-compatible wallets will detect the accounts and handle them appropriately

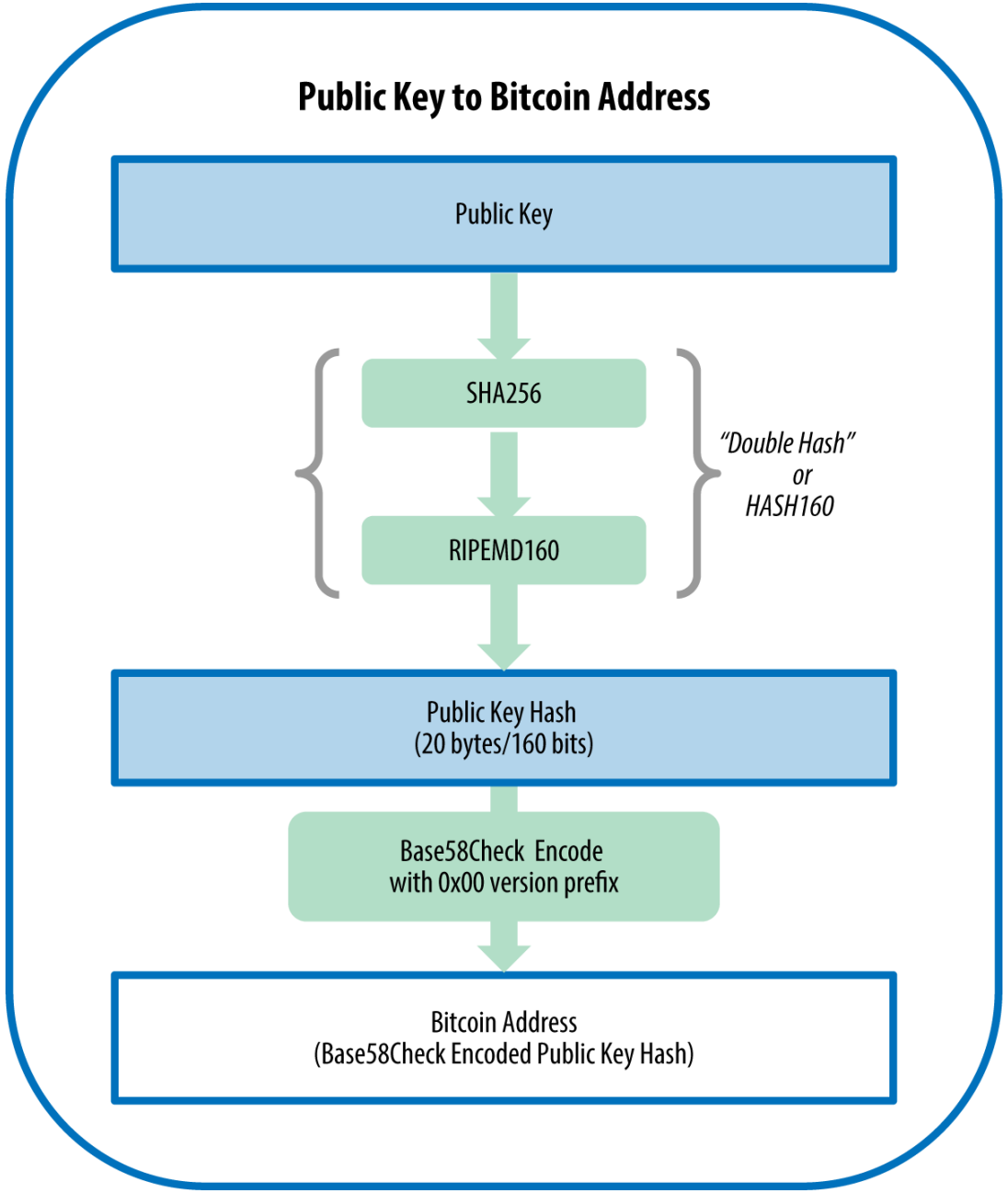
- m / purpose' / coin_type' / account' / change / address_index
  - Same BIP44 path except *purpose* is 49

BIP84 - DERIVATION SCHEME FOR P2WPKH BASED ACCOUNTS

- P2WPKH is the native format for Segwit
- BIP84 defines the derivation scheme for HD wallets using the P2WPKH (BIP173) serialization format for segwit transactions
- User needs to create dedicated Segwit wallets
  - Ensures that only wallets compatible with this BIP will detect the accounts and handle them appropriately
- m / purpose' / coin_type' / account' / change / address_index
  - Same BIP44 path except *purpose* is 84

5

BITCOIN ADDRESSES
(TYPES, ENCODINGS)



User level
But usually doesn't see it

Blockchain level

User level
What is actually displayed

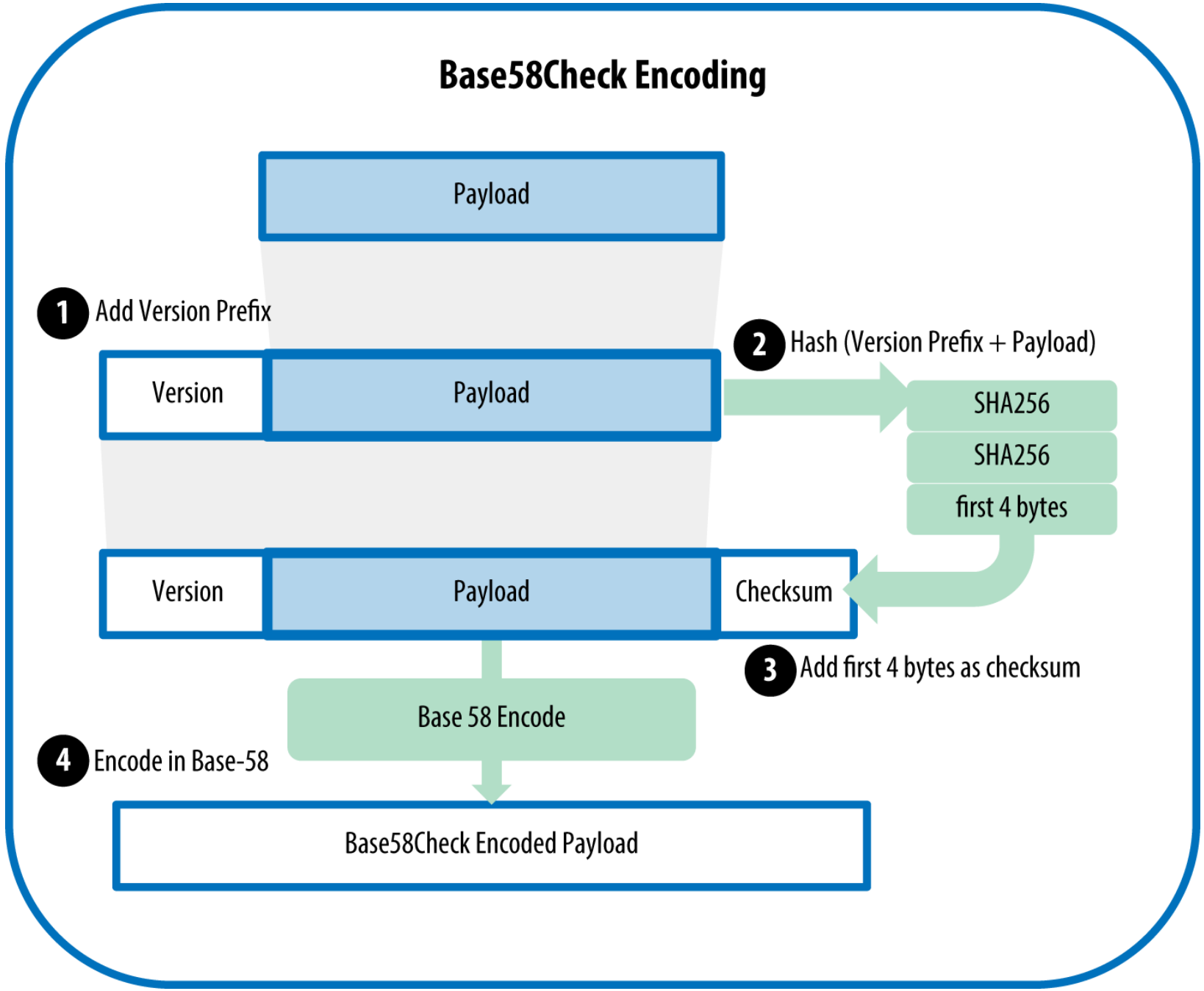
BASE58CHECK

- Most of the data presented to the user is Base58Check-encoded
  - Bitcoin address, private key, encrypted key, script hash, ...
  - Help human readability, avoid ambiguity, and protect against errors in transcription

- Uses 58 characters (a Base58 number system)
  - Set of lowercase and capital letters and numbers without 0, O, l, I
- Built-in error-checking code (checksum)
- Compact representation

- Base58Check Bitcoin address
  - [one-byte version 0x00][20-byte HASH160 pubKeyHash][4-byte checksum]

- Base58Check P2SH address
  - [one-byte version 0x05][20-byte HASH160 redeemScriptHash][4-byte checksum]


Base58Check Encoding
Payload
1
Add Version Prefix
Version
Payload
2
Hash (Version Prefix + Payload)
SHA256
SHA256
first 4 bytes
Version
Payload
Checksum
3
Add first 4 bytes as checksum
Base 58 Encode
4
Encode in Base-58
Base58Check Encoded Payload

Type	Version prefix (hex)	Base58 result prefix
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
BIP-38 Encrypted Private Key	0x0142	6P
BIP-32 Extended Public Key	0x0488B21E	xpub

BECH32 ENCODING

- New Segwit address format specified by BIP 173
  - Like Base58, it is end-user sugar, bech32 is never on the blockchain

- Only encodes witness (P2WPKH and P2WSH) scripts. Not compatible with non-segwit P2PKH or P2SH scripts

- A segwit address is a Bech32 encoding of
  - The human-readable part "bc" for mainnet, and "tb" for testnet
  - The separator « 1 »
  - The data-part values: checksum encoded witness script

- Several advantages over Base58Check
  - QR code is smaller
  - Detection of any error affecting at most 4 characters (BCH codes)
  - Error correction
  - Only consisting of lower cases, should be easier to type and read aloud
  - Excludes characters "1", "b", "i", and "o"
  - Increased security

CONCLUSION

- Wallet development is still in an exploratory phase

- We still have to design the perfect mnemonic/seed/derivation scheme

- Standards spread across BIPs, SLIPs and de-facto standard not in a formal doc
- Standards are never perfect, and in constant evolution
- Standards are not always followed, and it's fine
  - Electrum xkey version bytes, LND aezeed, ...
- Lot of implemented standards are still in draft status

- We have very basic tools for advanced features (timelocks, custom contracts, choosing UTXOs, privacy schemes, ...)

- With Bitcoin you are your own bank
  - Ultimately you can’t rely on anybody
  - Wallets provide no guarantee whatsoever as the quality or reliability of their product
  - Which means that at some point you need to learn all that
  - It is useful to know which derivation scheme is used by your wallet (BIP44, BIP49, custom…)

- Choose your wallet carefully
  - The most famous, used, renowned
  - The official version
  - No more than pocket money in beta version
  - Check which BIPs are supported
  - Check MD5 checksum, Gitian build
  - Use multiple wallets in case one has an issue

- Need proper key management
  - Leaking a mnemonic/privKey means loss of coins
  - Leaking a public key mean loss of privacy
  - More care must be taken regarding extended keys (entire tree of keys)
  - Shamir Secret Sharing your mnemonic
  - Be organized (Excel, regular check of access of funds, ...)

BIBLIOGRAPHY

- Advanced GUI for HD wallet key derivation
  - https://iancoleman.io/bip39/
- Bitcoin Developer Guide
  - https://bitcoin.org/en/developer-guide#wallets
- Mastering Bitcoin, Andreas Antonopoulos
  - https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch04.asciidoc
  - https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch05.asciidoc
- HD wallet BIPs
  - https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki
  - https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki
  - https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki

TERMINOLOGY

- Wallet
- HD wallet
- Seed
- Mnemonic
- Master key
- Chain code
- Extended key (xPriv, xPub)
- Neutered key
- Key derivation function
- Child key
- Hardened child key

- Public key cryptography
- Public/private keys
- Elliptic curve cryptography
- One-way function
- Trapdoor function
- Entropy
- Keyed-hash Message Authentication Code

IDEAL PROPERTIES OF A MNEMONIC/SEED

- Upgreadable
- Portable to multiple coins
- Wallet efficient (avoid unnecessary keychain scan)
- Informative for the end-user
- Self-descriptive
  - But is it a good thing to encode information/behavior at the mnemonic/seed level
  - Should be restricted to a specific purpose or purpose-agnostic?

KEY SERIALIZATION FORMAT

- 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)

LND AEZEED

- New scheme to replace BIP39

- Versioning
  - For wallets to be able to know how to re-derive the keys of the wallet
  - See also Electrum Seed Version System

- Wallet birthday
  - Expressed in Bitcoin Days Genesis (days since the timestamp in Bitcoin's genesis block)

- Much stronger KDF
  - Scrypt with modern parameters

- 1 byte internal version || 2 byte timestamp || 16 bytes of entropy
  - Converted to 33-byte cipher text using Scrypt and AEZ (AEAD)
  - Encoded using BIP 39 to produce 24 english words

LND DERIVATION SCHEME

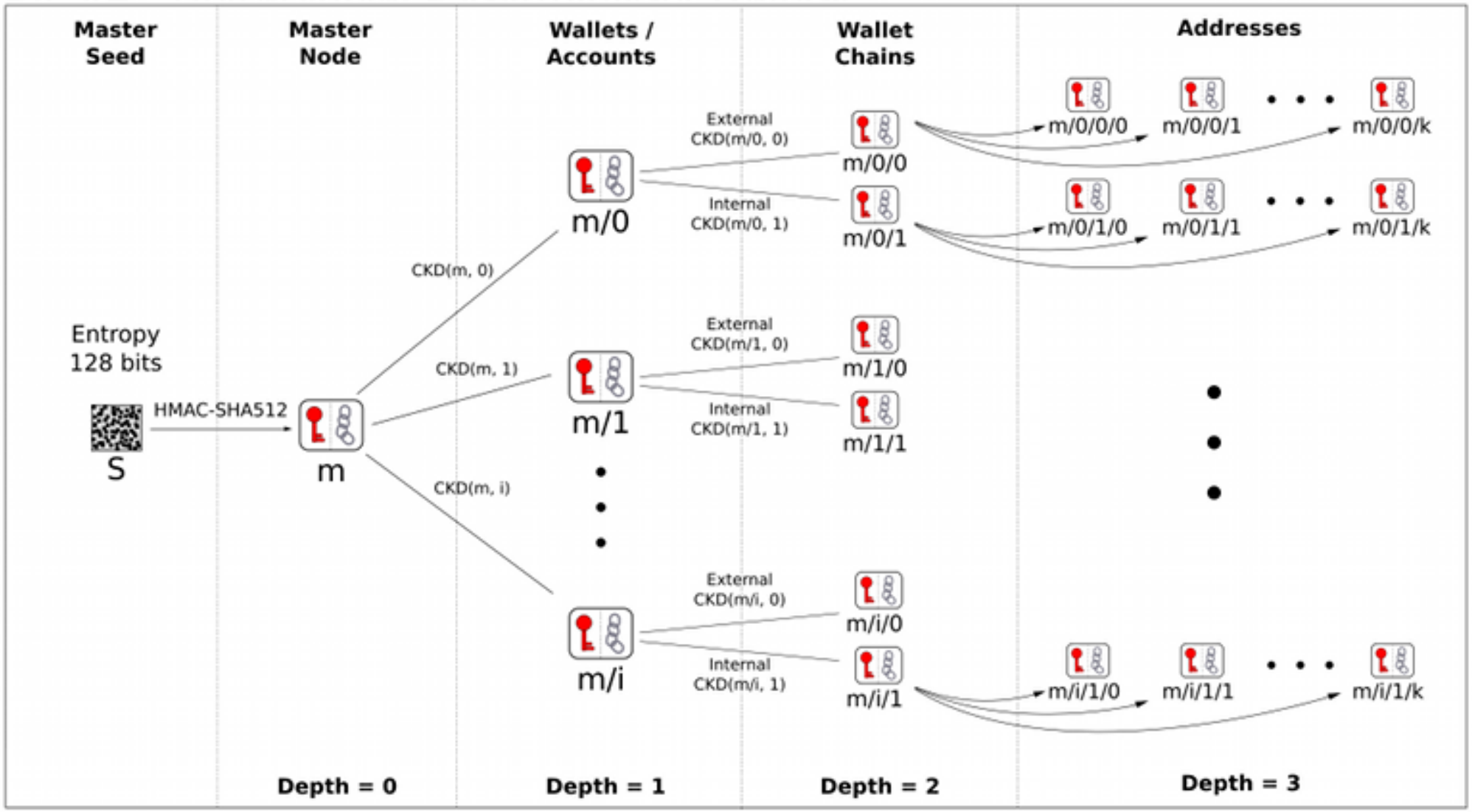
- Versioning of the key derivation scheme
  - Currently set to 0

- BIP43 *purpose* currently set to 1017
  - Meaning it is the first version of the scheme 0

- KeyFamily represents a "family" of keys that will be used within various contracts created by lnd
  - m/1017'/coinType'/keyFamily/0/index
  - These families are meant to be distinct branches within the HD key chain of the backing wallet
  - KeyFamily are just individual "accounts" in the nomenclature of BIP43

- KeyLocator is a two-tuple that can be used to derive *any* key that has ever been used under the current LND key derivation mechanisms
  - m/201'/coinType'/keyFamily/0/index

BIP 32 - Hierarchical Deterministic Wallets



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

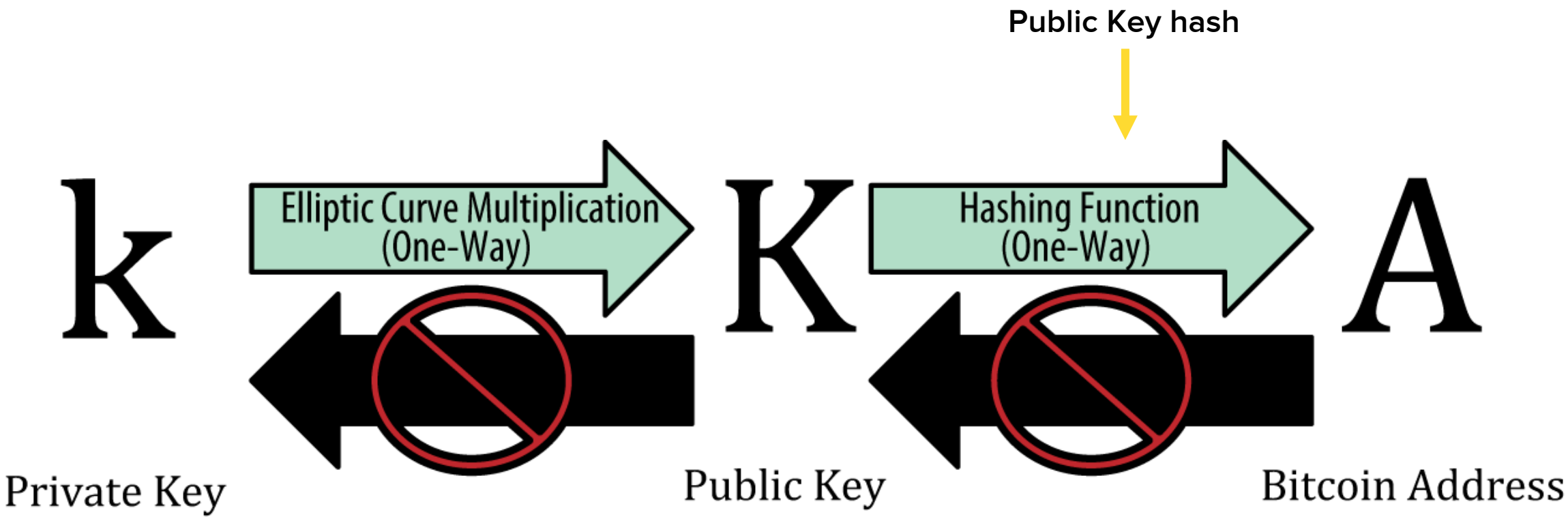
```
$ bitcoin-cli getnewaddress
1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy

$ bitcoin-cli dumpprivkey 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ


$ bx seed | bx ec-new | bx ec-to-wif
5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
```

Format	Private key
Hex	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
Hex-compressed	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD01
WIF-compressed	KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

Entropy input (128 bits)	0c1e24e5917779d297e14d45f14e1a1a
Mnemonic (12 words)	army van defense carry jealous true garbage claim echo media make crunch
Passphrase	(none)
Seed (512 bits)	5b56c417303faa3fcba7e57400e120a0ca83ec5a4fc9ffba757fbe63fbd77a89a1a3be4c67196f57c39a88b76373733891bfaba16ed27a813ceed498804c0570

Entropy input (128 bits)	0c1e24e5917779d297e14d45f14e1a1a
Mnemonic (12 words)	army van defense carry jealous true garbage claim echo media make crunch
Passphrase	SuperDuperSecret
Seed (512 bits)	3b5df16df2157104cfdd22830162a5e170c0161653e3afe6c88defeefb0818c793dbb28ab3ab091897d0715861dc8a18358f80b79d49acf64142ae57037d1d54

Entropy input (256 bits)	2041546864449caff939d32d574753fe684d3c947c3346713dd8423e74abcf8c
Mnemonic (24 words)	cake apple borrow silk endorse fitness top denial coil riot stay wolf luggage oxygen faint major edit measure invite love trap field dilemma oblige
Passphrase	(none)
Seed (512 bits)	3269bce2674acbd188d4f120072b13b088a0ecf87c6e4cae41657a0bb78f5315b33b3a04356e53d062e5f1e0deaa082df8d487381379df848a6ad7e98798404


Public Key hash
k
Elliptic Curve Multiplication
(One-Way)
K
Hashing Function
(One-Way)
A
Private Key
Public Key
Bitcoin Address

BITCOIN CORE WALLET

- Uses derivation path m/0'/0'/k

OTHER WALLET STRUCTURE EXEMPLES

- BIP80 – Hierarchy for Non-Colored Voting Pool Deterministic Multisig Wallets
  - Not in use

- BIP81 – Hierarchy for Colored Voting Pool Deterministic Multisig Wallets
  - Not in use

PRIVATE DERIVATION

- Public keys Qi can be found deterministically through private derivation
- Private derivation computes public key Qi through point multiplication of private key d_i with the generator point
- Private key d_i is in turn computed from a formula relating the master private key to index i
- The simplest formula for d_i merely adds i to m

PUBLIC DERIVATION

- Public keys Qi can be more securely generated through public derivation. Public derivation computes public key Qi independently of master private key m. Instead of point multiplication, public derivation uses point addition to find Qi from master public key M.

- Public derivation relies on the distributive property of point multiplication. Imagine two scalars, a and b, and a point on elliptic curve, C. Point multiplication of the sum (a + b) with point C gives point D (eq 23). The distributive property ensures that point D can also be found through point addition of aD and aC