

UNIVERSITY NAME

DOCTORAL THESIS

Hierarchical deterministic wallet

Author:

Daniele FORNARO

Supervisor:

Daniele MARAZZINA

*A thesis submitted in fulfillment of the requirements
for the degree of Mathematical Engineering*

in the

Research Group Name
Department or School Name

February 21, 2018

Declaration of Authorship

I, Daniele FORNARO, declare that this thesis titled, “Hierarchical deterministic wallet” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

“Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism.”

Dave Barry

UNIVERSITY NAME

Abstract

Faculty Name
Department or School Name

Mathematical Engineering

Hierarchical deterministic wallet

by Daniele FORNARO

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor...

Contents

Declaration of Authorship	iii
Abstract	vii
Acknowledgements	ix
1 Elliptic Curve Geometry	1
1.1 Introduction	1
1.2 Elliptic Curve over \mathbb{F}_p	1
1.2.1 Operations	1
Symmetry	1
Point addition	2
Scalar multiplication	3
1.2.2 Group order	3
Cyclic subgroups	3
1.3 Bitcoin private-public key cryptography	4
1.3.1 Bitcoin Elliptic Curve	4
1.3.2 Bitcoin keys representation and addresses	4
Uncompressed public key	4
Compressed public key	5
WIF Private Key	5
Address	5
2 Wallet	7
2.1 Nondeterministic (<i>random</i>) Wallet	7
Pros and Cons	7
2.2 Deterministic Wallets	8
2.2.1 Deterministic Wallet <i>type-1</i>	8
Pros and Cons	9
2.2.2 Deterministic Wallet <i>type-2</i>	9
Pros and Cons	10
2.2.3 Deterministic Wallet <i>type-3</i>	11
3 Hierarchical Deterministic Wallet	13
3.1 Elements	13
3.1.1 Seed	13
3.1.2 Extended Key	13
3.2 From SEED to Master Private Key	14
3.3 Child Key derivation	16
3.3.1 Normal derivation	17
3.3.2 Hardened derivation	18
3.4 Special derivation	19
3.4.1 Public derivation	19

3.4.2	Weakness of Normal Derivation	20
3.5	Advantages and disadvantages	21
3.5.1	When use Normal Derivation?	22
3.5.2	When use Hardened Derivation?	22
4	Mnemonic phrase	25
4.1	BIP39	25
4.1.1	Mnemonic Generation	25
4.1.2	From Mnemonic to Seed	27
4.2	Electrum Mnemonic	28
4.2.1	Mnemonic Generation	28
4.2.2	From Mnemonic to Seed	30
4.3	Comparison	30
5	Child Key Derivation	31
5.1	Functional explanation	31
5.2	Normal derivation	31
5.2.1	Derive public child from public parent	31
5.2.2	Possible Risk	31
5.3	Hardened derivation	31
6	Mnemonic to Seed	33
6.1	Functional explanation	33
6.2	BIP 39 derivation	33
6.2.1	Mnemonic generation	33
6.2.2	Seed derivation	33
6.3	Electrum derivation	33
6.3.1	Mnemonic generation	33
6.3.2	Seed derivation	33
6.4	BIP39 vs Electrum derivation	33
7	How to use a HD Wallet	35
7.1	Multi-coin wallet BIP 44	35
A	Frequently Asked Questions	37
A.1	How do I change the colors of links?	37

List of Figures

1.1	Points on the Elliptic Curve $y^2 = x^3 - 7x + 10 \pmod{p}$, with $p = 19, 97, 127, 487$	2
1.2	Elliptic Curve $y^2 = x^3 - 7x + 10 \pmod{97}$	2
3.1	From seed to master private key	16
3.2	Normal Derivation	18
3.3	Hardened Derivation	19
3.4	Hardened Derivation	21
3.5	Hardened Derivation	22

List of Tables

List of Abbreviations

LAH List Abbreviations **Here**
WSF What (it) Stands **For**

Physical Constants

Speed of Light $c_0 = 2.997\,924\,58 \times 10^8 \text{ m s}^{-1}$ (exact)

List of Symbols

a	distance	m
P	power	W (J s ⁻¹)
ω	angular frequency	rad

For/Dedicated to/To my...

Chapter 1

Elliptic Curve Geometry

1.1 Introduction

Bitcoin security is based on public and private key cryptography. The main concept is that it is simple to compute the public key, knowing the private, but it is infeasible to calculate the private key, knowing the public.

In order to obtain this result a particular Elliptic Curve is used.

1.2 Elliptic Curve over \mathbb{F}_p

A point Q , which coordinates are x and y , belong to an Elliptic Curve if and only if Q satisfies the following equation:

$$y^2 = x^3 + ax + b \quad \text{over } \mathbb{F}_p \quad (1.1)$$

Where \mathbb{F}_p is the finite field defined over the set of integers modulo p and a and b are the coefficients of the curve.

We can rewrite the equation 1.1 in the following way:

$$y^2 = x^3 + ax + b \quad \text{mod } p \quad (1.2)$$

Figure 1.1 shows some examples of Elliptic Curve over \mathbb{F}_p with $a = -7$ and $b = 10$

1.2.1 Operations

A point on the Elliptic Curve has some particular properties:

- Symmetry
- Point addition
- Scalar multiplication

Symmetry

For every point in the x axis exists two points in the y axis. Suppose that a point $P(x, y)$ belongs to the Elliptic Curve, then it must satisfy the equation 1.1. So it is easy to prove that the point $Q(x, p - y)$ belongs to the curve too.

Furthermore we have $P = -Q$, from the moment that $P + Q = 0$ (see addition below).



FIGURE 1.1: Points on the Elliptic Curve $y^2 = x^3 - 7x + 10 \bmod p$, with $p = 19, 97, 127, 487$

Point addition

We need to change our definition of addition in order to make it works in \mathbb{F}_p . In this framework we claim that if three points are aligned over the finite field \mathbb{F}_p , then they have zero sum.

So $P + Q = R$ if and only if P, Q and $-R$ are aligned, in the sense shown in figure 1.2



FIGURE 1.2: Elliptic Curve $y^2 = x^3 - 7x + 10 \bmod 97$

The equations for calculating point additions are the follow:
Suppose that A and B belong to the Elliptic Curve.

$$A = (x_1, y_1) \quad B = (x_2, y_2)$$

Let's defined $A + B := (x_3, y_3)$

So we have:

$$s = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1}, & \text{if } x_1 \neq x_2 \\ \frac{3x_1^2 + a}{2y_1}, & \text{if } x_1 = x_2 \end{cases}$$

$$\begin{aligned} x_3 &= s^2 - x_1 - x_2 \pmod{p} \\ y_3 &= s(x_1 - x_3) - y_1 \pmod{p} \end{aligned}$$

Scalar multiplication

Once defined the addition, any multiplication can be defined as:

$$nP = \underbrace{P + P + \dots + P}_{n \text{ times}}$$

When n is a very large number can be difficult or even infeasible to compute nP in this way, but we can use the *double and add algorithm* in order to perform multiplication in $\mathcal{O}(\log n)$ steps.

1.2.2 Group order

An elliptic curve defined over a finite field is a group and so it has a finite number of points. This number is called order of the group.

If the prime order is a very large number, it is impossible to count all the point in that field, but there is an algorithm that allows to calculate the order of a group in a fast and efficient way: *Schoof's algorithm*.

Cyclic subgroups

Let's consider a generic point P , we have:

$$nP + mP = \underbrace{P + \dots + P}_{n \text{ times}} + \underbrace{P + \dots + P}_{m \text{ times}} = \underbrace{P + \dots + P}_{n+m \text{ times}} = (n+m)P$$

So multiple of P are closed under addition and this is enough to prove that the set of the multiples of P is a cyclic subgroup of the group formed by the elliptic curve.

The point P is called **generator** of the cyclic subgroup.

Remark The order of P is linked to the order of the elliptic curve by Lagrange's theorem, which states that the order of a subgroup is a divisor of the order of the parent group.

Remark If the order of the group is a prime number, all the point P generate a subgroup with the same order of the group.

1.3 Bitcoin private-public key cryptography

1.3.1 Bitcoin Elliptic Curve

Bitcoin uses a specific Elliptic Curve defined over the finite field of the natural numbers, where $a = 0$ and $b = 7$.

The equation 1.1 becomes:

$$y^2 = x^3 + 7 \pmod{p} \quad (1.3)$$

The \pmod{p} (modulo prime number) indicates that this curve is over a finite field of prime order $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$.

The *order* of this Elliptic Curve is a very large prime number, close to 2^{256} , but smaller than p .

Let's consider a particular point G , called generator, with:

$$\begin{aligned} x = & 79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798 \\ y = & 483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8 \end{aligned}$$

From the moment that the order of the group is a prime number, the order of any subgroup is equal to the order of the entire group. In particular the order of the subgroup generated by G is equal to *order*.

Definition A private key is a number chosen in the range between 1 and *order*.

Definition A public key W is a point in the Bitcoin EC, derived from a private key k in the following way:

$$W = k \cdot G \quad (1.4)$$

Where the multiplication between k and G is defined in the previous chapter.

This is a *one way* function, in the sense that computing the scalar multiplication, knowing the private key is simple, but make the reverse is infeasible.

Remark It is infeasible to calculate a private key knowing the public key.

1.3.2 Bitcoin keys representation and addresses

In order to make it easy to store and recognise keys, some encodings were designed.

A public key, a point in the EC, can be represented in two ways: *uncompressed* or *compressed*.

Uncompressed public key

An uncompressed public key is represented in hexadecimal digits, and it is obtained simply concatenating the x coordinate with the y coordinate and adding 04 at the beginning, for a total of 130 hexadecimal digits.

Example of an uncompressed public key:

0450863AD64A87AE8A2FE83C1AF1A8403CB53F53E486D8511DAD8A04887E5B235
22CD470243453A299FA9E77237716103ABC11A1DF38855ED6F2EE187E9C582BA6

Compressed public key

A compressed public key is obtained simply taking the x coordinate and adding 02 at the beginning if the y coordinate is even, 03 otherwise.

This is due to the *symmetry properties* of a point of the EC.

Example of a public key compressed:

0250863AD64A87AE8A2FE83C1AF1A8403CB53F53E486D8511DAD8A04887E5B2352

WIF Private Key

WIF stands for wallet import format and is the standard way used to write down a private key.

- Add a version number (80 for Bitcoin) in front of the private key, in order to recognize quickly for what cryptocurrency that private key was used.
- Add 01 at the end of the private key if you want a WIF *compressed*, none if you want a WIF *uncompressed*. The difference between these two types is that from a *compressed* private key a *compressed* public key is expected and from a *uncompressed* private key a *uncompressed* public key is expected.
- Add a checksum at the end, obtained applying the SHA256 function twice to the string previously obtained, take the first 4 bytes (8 hexadecimal digits) and put them at the end of the string.
- Compute the Base58Encode, obtaining a 52 digit string.

Example of private key WIF:

KwdMAjGmerYanjeui5SHS7JkmpZvVipYvB2LJGU1ZxJwYvP98617

Address

Among the Bitcoin transactions, one of the most used is a *Pay-to-PubkeyHash*, meaning that in the transaction you will not write directly the public key, but the hash of that public key.

The hash function used in this framework is the HASH160 function, applied to the *compressed* public key. This is an irreversible procedure, so you cannot obtain the public key from the public key hash.

In order to obtain a valid Bitcoin address, it is needed to encode the *PubkeyHash* in base58, adding first the version in front, the checksum at the end and then encode everything with Base58Encode, obtaining a 34 digit string.

Example of an Address:

1BvBMSEYstWetqTFn5Au4m4GFg7xJaNVN2

Chapter 2

Wallet

A Bitcoin wallet is a structure used to store keys.

There are different type of wallet:

- Nondeterministic (*random*) Wallet
- Deterministic Wallet

Remark *Bitcoin wallets contains keys, not coins. Coins are in the Blockchain.*

2.1 Nondeterministic (*random*) Wallet

A nondeterministic wallet is the simplest type of wallet. Each Key is randomly and independently generated.

- (i) Consider a *Discrete Uniform Random Variable*

$$X \sim \mathcal{U}(S)$$

Where S is the finite set of natural number in the range from 1 to *order*.

- (ii) Take some realizations $k_1, k_2 \dots k_n$ of X using enough entropy to make these numbers (*private keys*) impossible to guess.

$$k_1 = X(\omega_1) \quad k_2 = X(\omega_2) \quad \dots \quad k_n = X(\omega_n)$$

- (iii) Go back to point (i) every time new *private keys* are needed.

With this procedure it is impossible to compute the *public key* without having already the *private key*.

Pros and Cons

Let's focus on the good and bad aspects of this wallet.

<i>Random Wallet</i>	
Pros	Cons
<ul style="list-style-type: none"> • Easy to implement 	<ul style="list-style-type: none"> • Difficult to find <u>real</u> new entropy for every new <i>private key</i>. • Every time new <i>private keys</i> are needed, you need to make new back up. • Difficult to store or back up in a <i>non digital way</i>. Awkward to write it down all yours keys on a paper.

The use of *random wallet* is strongly discouraged for anything other than simple test. There are no good reason to use it.

2.2 Deterministic Wallets

A deterministic wallet is a more sophisticated one, in which every key is generated from a common "*seed*". This means that knowing the *seed* means also to know all the keys in the wallet.

There are different types of deterministic wallets, in this text we will analyze three main types:

- Deterministic Wallet *type 1*
- Deterministic Wallet *type 2*
- Deterministic Wallet *type 3*

These wallet are in increasing order of complexity.

2.2.1 Deterministic Wallet *type-1*

The Deterministic Wallet *type-1* is one of the simplest Wallet among the deterministic ones. Each key is generated adding a number in a sequential order to the *seed* and then computing an *hash* function such as the **SHA256** function.

Let's see how to generate the n^{th} private key:

- Generate a *seed* (only once), a random number from a *Discrete Uniform Random Variable*

$$seed = X(\omega) \quad X \sim \mathcal{U}(S)$$

Where S is the finite set of natural number in the range from 1 to *order*.

- (ii) Consider the numbers *seed* and *n* as strings and concatenate *n* to *seed*, obtaining a *value*

$$value = seed|n$$

- (iii) Compute the SHA256 function to *value* and obtain the n^{th} *private key*.

- (iv) Go back to point (ii) every time new *private keys* are needed with $n = n + 1$.

With this procedure it is impossible to compute the *public key* without having already computed the *private key*.

Pros and Cons

Let's focus on the good and bad aspects of this wallet.

<i>Deterministic Wallet type-1</i>	
Pros	Cons
<ul style="list-style-type: none"> • In order to make a back up of the entire wallet it is needed to store the <i>seed</i> only. All <i>private keys</i> can be derived from it. • A single back up is needed. • The <i>seed</i> can be stored easily also in a <i>non digital way</i>, in a paper for example. 	<ul style="list-style-type: none"> • Every time new <i>public keys</i> are needed, you need to use the <i>seed</i>, to compute new <i>private keys</i> and then derive the <i>public</i> ones. This could compromise the entire wallet, if the <i>seed</i> is used in a non safe environment. • There is only a <i>key sequence</i>. No way to distinguish the "purpose" of each <i>private key</i>.

The use of this type of wallet is not recommended for everyday use, but it could be used to store Bitcoin in a safe place: *cold wallet*.

2.2.2 Deterministic Wallet type-2

The Deterministic Wallet *type-2* is more sophisticated. Each *private key* is generated in such a way that it is possible to compute the respective *public key* without knowing the *private*.

First let's introduce the necessary ingredients:

- ◇ **Master private key** (*mp*): a random number, generated from a *Discrete Uniform Random Variable*

$$mp = X(\omega) \quad X \sim \mathcal{U}(S)$$

Where *S* is the finite set of natural number in the range from 1 to *order*.

The master private key must be take secret.

- ◇ **Master public key (MP)**: a point on the EC, obtained from the mp :

$$MP = mp \cdot G$$

Where G is the *generator*.

This point can be consider non-secret.

- ◇ **Public random number (r)**: a random number, generated from a *Discrete Uniform Random Variable*

$$r = X(\omega) \quad X \sim \mathcal{U}(S)$$

This number can be consider non-secret.

Let's see how to generate the n^{th} private key: p_n

- (i) Compute the SHA256 function to the concatenation of r with n , considered as string:

$$h_{n|r} = \text{SHA256}(n|r)$$

$h_{n|r}$ can be consider non-secret, from the moment that it is derived from non secret information.

- (ii) Compute the n^{th} private key adding mp to $h_{n|r}$:

$$p_n = mp + h_{n|r} \quad \text{mod (order)}$$

In order to obtain the corresponding *public key* P_n , it is possible to compute the standard multiplication:

$$P_n = p_n \cdot G$$

It is also possible to compute P_n without knowing p_n , using only non-secret information: $h_{n|r}$ and MP .

- (i) Compute V :

$$V = h_{n|r} \cdot G$$

V can be see as the *public key* of $h_{n|r}$ and can be consider non-secret.

- (ii) Add MP to V :

$$P_n = MP + V$$

Where the sum in this contest is the one defined between two point in the EC.

It is easy to prove that P_n can be computed in these two way:

$$\begin{aligned} P_n &= p_n \cdot G \\ &= (mp + h_{n|r}) \cdot G \\ &= (mp \cdot G) + (h_{n|r} \cdot G) \\ &= MP + V \end{aligned}$$

Pros and Cons

Let's focus on the good and bad aspects of this wallet.

<i>Deterministic Wallet type-2</i>	
Pros	Cons
<ul style="list-style-type: none"> • In order to make a back up of the entire wallet it is needed to store the <i>master private key</i> and the random number <i>r</i>. All <i>private keys</i> can be derived from them. • A single back up is needed. • The <i>master private key</i> can be stored easily also in a <i>non digital way</i>, in a paper for example. • It is possible to derive a new <i>public key</i> using only non-secret information, with the procedure above. 	<ul style="list-style-type: none"> • There is only a <i>key</i> sequence. No way to distinguish the "purpose" of each <i>private key</i>.

The *type-2 deterministic wallet* it is an improvement of the *type-1* because it has the same benefits (except for the need to back up two number instead of only one), but with a great advantage: it is possible to generate new addresses also in a non safe environment, having only *r* and *MP*.

A thief can only steal your privacy, because if *MP* and *r* are stolen, he is not able to make any Bitcoin transactions from your wallet, but he can see all the previous transactions and the total amount of Bitcoin stored in the wallet.

2.2.3 Deterministic Wallet *type-3*

Deterministic Wallet *type-3* is the most elaborate among the ones considered. Starting from a *seed* it is possible to obtain different *keys* in a hierarchical way, with a structure similar to a tree.

Let's see roughly how an this Wallet works:

- (i) Generate a *seed*, a random number from a *Discrete Uniform Random Variable*, unique for each wallet.

$$seed = X(\omega) \quad X \sim \mathcal{U}(S)$$

Where *S* is a finite set of natural number.

- (ii) Generate a *master private key* from the *seed*, using a stretching function: PBKDF2.
- (iii) From this *master private key* it is possible to generate 2^8 *private key* using a irreversible function: HASH512

- (iv) Everyone of this *private key* "children" can derive 2^8 *private key* and all of these "grandchildren" can derive as many.

This procedure can produce a huge number of keys. They seem independent from an outside point of view: it is impossible to guess that two private-public key are derived from the same *seed*.

This particular type of Wallet is commonly known as **Hierarchical Deterministic Wallet**, one of the most used and widespread.

In the next chapter we will see in detail how it works.

Chapter 3

Hierarchical Deterministic Wallet

In this chapter we will see how an HD wallet works.

3.1 Elements

Let's focus on the main elements of the Wallet:

- ◇ Seed
- ◇ Extended keys

3.1.1 Seed

The entire Wallet is based on a *seed*.

It is a number taken from a *Discrete Uniform Random Variable*

$$seed = X(\omega) \quad X \sim \mathcal{U}(S)$$

Where S is the finite set of natural number in the range from 1 to an arbitrary value. Obviously the greater the set from which the number can be extracted, the better it is for the security of the seed itself.

This is an example of seed expressed in hexadecimal format:

```
seed=ffcf9f6f3f0edeae7e4e1dedbd8d5d2cfccc9c6c3c0bdbab7b4b1aeaba8a5a29f9c999
693908d8a8784817e7b7875726f6c696663605d5a5754514e4b484542
```

3.1.2 Extended Key

An Extended Key is a sequence of bytes, encoded in base58. It contains all the information necessary for the derivation. When the derivation is made for the first time from the seed, the extended key is called master key.

Once it is decoded we will obtain exactly 78 bytes, with a specific meaning and order:

- ⊗ 4 bytes are used to specified the **version**.
- ⊗ 1 byte is used to specified the **depth** in the hierarchical tree: the extended key derived directly from the seed has *depth* = 0, its first children have *depth* = 1, grandchildren have *depth* = 2 and so on.

- ⊗ 4 bytes are used for the **fingerprint**. It is a unique value that identify the parent. Compute the HASH160 function on the "parent" public key in a compressed form and then take the first 4 bytes, this is the fingerprint of the child:

$$fingerprint = HASH160(\text{parent public key})[0 : 4]$$

Where $[0 : 4]$ is a python notation.

For the master key the fingerprint is formed by 4 zeros bytes: $fingerprint = 0000000000$

- ⊗ 4 bytes are used to specified the **index** of the child.
For the master key the index is formed by 4 zeros bytes: $index = 0000000000$
- ⊗ 32 bytes are used for the **chain code**. The chain code is used in order to introduce a sort of entropy in the children generation. We will see below how it works.
- ⊗ 33 bytes are used for the **key**. It can be *private* or *public*.
Public key is expressed in compact form, so the first byte is always 02 or 03.
The first byte of the private key is always 00 in order to distinguish the key from the public one.

An extended key is called **Extended Private Key** if the lasts 33 bytes are used to specify the private key; it is called **Extended Public Key** if they are used to specify the public key.

For the Bitcoin mainnet it is used for the **version**: 0x0488ADE4 for an extended private key, 0x0488B21E for an extended public key. When this bytes are encoded in base58, they returns *xprv* and *xpub* respectively.

Remark Obviously it is possible to calculate the extended public key starting from the extended private key, but it is infeasible to do the opposite. The only difference between the two extended keys are the **key bytes** and the **version bytes**, all the others elements remain the same.

3.2 From SEED to Master Private Key

In this section we will see in detail how it is possible to switch from a *seed* to a *master private key*.

First of all we need to convert the seed into a string of bytes, where the most significant bytes come first (big endian). In order to do so, we need to know how much long we want the string of bytes.

Let's see a practical example:

$$byte_string_1 = 00\ 00\ 00\ 01$$

$$byte_string_2 = 00\ 00\ 01$$

$$byte_string_3 = 00\ 01$$

$$byte_string_4 = 01$$

These 4 byte strings are obtained from the same seed: $seed = 1$ and the only different is the length of the string.

Remark *Different length of string produce different master private key, even if the seed is the same number.*

In python:

```
1 byte_string = seed.to_bytes(seed_bytes, 'big')
```

Where $seed$ is a *int*, $seed_bytes$ is the number of bytes that the *byte_string* should have.

It is essential to specify the length of the byte string, otherwise there will be obtained different wallets.

Once we obtain a string of bytes, we will compute the HMAC algorithm. The hash function used for HMAC is the SHA512 and the *key* is a particular string of bytes: *b"Bitcoin seed"*. In python the implementation is the follow:

```
1 from hashlib import sha512
2 from hmac import HMAC
3
4 hashValue = HMAC(b"Bitcoin seed", byte_string, sha512).digest()
```

Where *.digest()* is used in order to return a string of bytes.

Now we have obtained an *hashValue* of 512 bits, so 64 bytes. Consider the firsts 32 bytes as the master private key and the next 32 bytes as the master chain code. A python implementation is the follow:

```
1 private_key_bytes = hashValue[0:32]
2 chain_code_bytes = hashValue[32:64]
```

Now we have two byte strings, one for the master private key and the other for the master chain code.

It is important to remember that a private key must be in the range between 1 and *order*, so the byte string for the private key should be converted in *int* and then take the *mod order*. In python we have:

```
1 private_key = int(private_key_bytes.hex(), 16) % order
```

Finally we will concatenate all the informations obtained in order to form a Master Extended Private Key (in bytes format):

- $vbytes = b'\x04\x88\xAD\xE4'$
- $depth = b'\x00'$
- $fingerprint = b'\x00\x00\x00\x00'$
- $index = b'\x00\x00\x00\x00'$
- chain code is the one previously computed
- private key = $b'\x00' +$ private key in bytes format, previously computed.

Then the Master Extended Private Key is formed by concatenation:

```
1 xkey = vbytes + depth + fingerprint + index + chain_code + key
```

In order to make it readable, a base58 encode is performed.

This is an example of Master Extended Private Key:

```
xprv9s21ZrQH143K3wEaiSjZ8jYCuZF1oJoXHiwFcx2WwXqQHD4ZLdyEAFZ22M4
BmQT82HRbWssLArj53YDQTj6vSN4iH6nTiSQ61C5CckxUtDq
```

Remark The SHA512 is an irreversible function, so it is infeasible to obtain the seed, knowing the Extended Key. (It is also useless because with the master key you can derive all the keys in the wallet).

Graphically these operation can be shown in figure 3.1

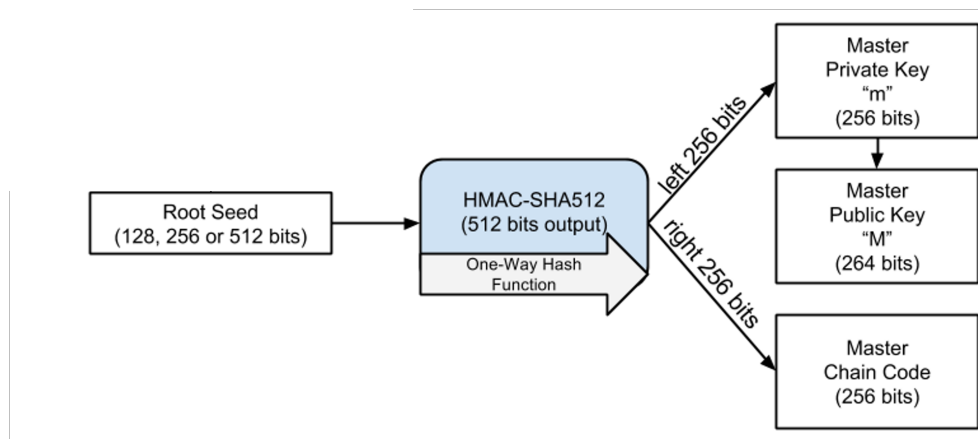


FIGURE 3.1: From seed to master private key

3.3 Child Key derivation

From any extended private key it is possible to obtain different child keys. There are two methods used in order to do so:

- Normal
- Hardened

Both methods have some advantage and disadvantage that we will discuss later. For every situation it is essential to use the method that best fits.

For both methods the derivation starts from an Extended Private Key. From this key some essential information is necessary:

- ★ Chain code
- ★ Private key

It is also required a number, used in order to specify the **index** of the child. This number should be in the range between 0 and 4294967295. This is due to the fact that in any Extended Key there are 4 bytes used to specify the index of the child:

$$\max \text{index} = (FF \ FF \ FF \ FF)_{16} = (4294967295)_{10}$$

In fact it is possible to generate even a greater number of children from the same parent, but it would not be possible to write the corresponding Extended Key in the format described above.

3.3.1 Normal derivation

As already mentioned we need only 3 ingredient in order to derive a key. Let's see how we can combine them together in order to obtain a new key (and chain code).

First we need to compute the Parent Public Key P . This is obtained from the usual scalar multiplication of an EC point (the Generator G) with the Parent Private Key p :

$$P = p \cdot G$$

Consider only the compress form of P and convert this value into a byte string, obtaining 33 bytes.

Concatenate this 33 byte string to the 4 byte string representing the index number:

$$msg = \text{compressed public key} \mid \text{index}$$

msg is a string of 37 bytes.

Apply the HMAC algorithm with the following input:

- ⊙ **Hash function:** SHA512
- ⊙ **Key:** chain code
- ⊙ **Message:** msg

The Python code is the follow:

```
1 from hmac import HMAC
2 from hashlib import sha512
3
4 msg=parent_public_key + index
5 hashValue = HMAC(parent_chain_code, msg, sha512).digest()
```

The result is a string of 64 bytes: $hashValue$.

Split this string of bytes in two: the last 32 are the child chain code. Take the first 32 bytes, convert them into an integer number and sum it to the parent private key (mod $order$), obtaining the child private key.

This is the python code:

```
1 child_chain_code = hashValue[32:]
2 q = int(hashValue[:32].hex(), 16)
3 child_private_key = (q + parent_private_key) % order
```

Graphically these operation can be shown in figure [3.2](#)

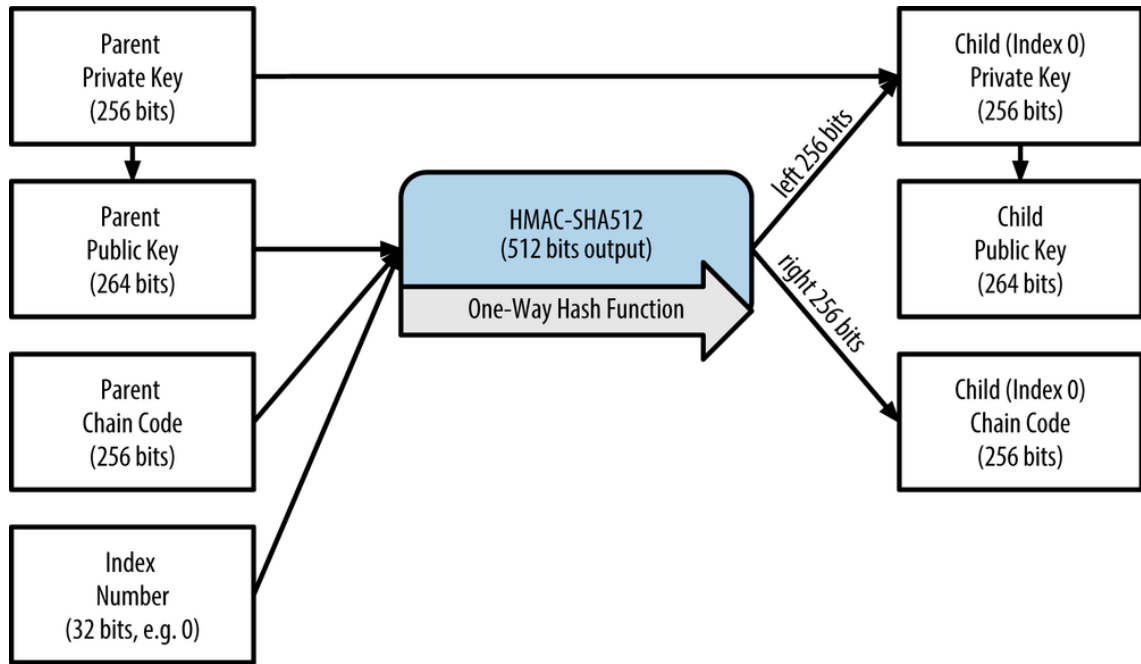


FIGURE 3.2: Normal Derivation

3.3.2 Hardened derivation

This method is similar to the previous one, the only difference is that as input of the *hash* function the private key is used instead of the public one.

Concatenate the 33 bytes of parent private key, considering also the 00 byte, with the 4 byte string representing the index number:

$$msg = 00 \mid private\ key \mid index$$

msg is a string of 37 bytes.

Apply the HMAC algorithm with the following input:

- ⊙ **Hash function:** SHA512
- ⊙ **Key:** chain code
- ⊙ **Message:** *msg*

The Python code is the follow:

```
1 from hmac import HMAC
2 from hashlib import sha512
3
4 msg=parent_private_key + index
5 hashValue = HMAC(parent_chain_code , msg, sha512).digest()
```

The result is a string of 64 bytes: *hashValue*. In this code the *parent_private_key* already has the first 00 byte, because it is taken directly from the parent extended private key.

Split this string of bytes in two (in the same way as the normal method): the last 32 are the child chain code. Take the first 32 bytes, convert them into an integer number

and sum it to the parent private key (mod *order*), obtaining the child private key.

This is the python code:

```
1 child_chain_code = hashValue[32:]
2 q = int(hashValue[:32].hex(), 16)
3 child_private_key = (q + parent_private_key) % order
```

Graphically these operation can be shown in figure 3.3

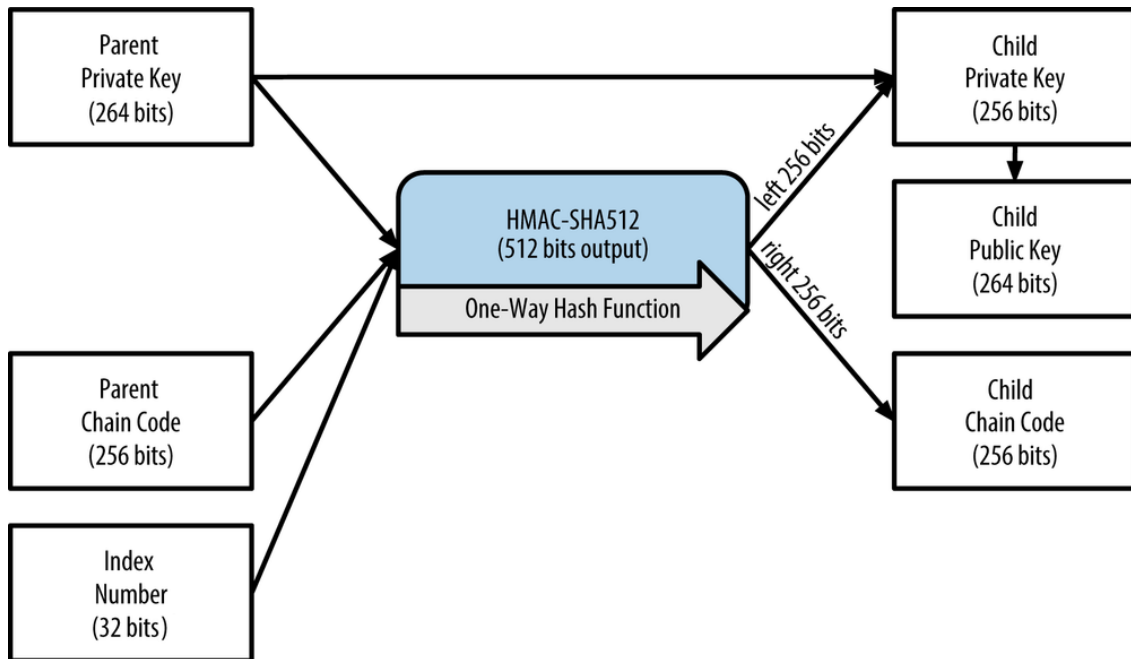


FIGURE 3.3: Hardened Derivation

3.4 Special derivation

Using a *normal derivation* it is possible to derive the extended public key, starting only from the extended public key of the parent.

3.4.1 Public derivation

The only essential elements are the ones contained in the extended public key, in particular:

- Public key P_{parent}
- Chain code
- Index

As shown for the normal derivation starting from the extended private key, we have all the input elements for the hash algorithm:

Concatenate the 33 byte string of the public key to the 4 byte string representing the index number:

$$msg = \text{compressed public key} \mid \text{index}$$

msg is a string of 37 bytes.

Apply the HMAC algorithm with the following input:

- ⊙ **Hash function:** SHA512
- ⊙ **Key:** chain code
- ⊙ **Message:** msg

The output of this function is the same of the normal derivation with the extended private key. The last 32 bytes formed as usual the child chain code, instead the first 32 bytes can be read as a special number: q .

Multiply the generator G to the integer number q and obtain Q , a point on the EC:

$$Q = q \cdot G$$

Compute the sum between two point on the elliptic curve: Q and P_{parent} , where P_{parent} is the parent public key.

$$Q + P_{parent} = P_{child}$$

Where P_{child} is the child public key. Let's prove that the child public key obtained in this way P_{child_2} is the same as that obtained starting from the private key, P_{child_1} :

Both the procedures start from q , number obtained from the first 32 bytes of the HMAC function. Let's call p_{parent} the parent private key and p_{child} the child private key.

$$\begin{aligned} P_{child_1} &= p_{child} \cdot G \\ &= (q + p_{parent}) \cdot G \\ &= (q \cdot G) + (p_{parent} \cdot G) \\ &= Q + P_{parent} = P_{child_2} \end{aligned}$$

cvd

Graphically this derivation can be shown in figure 3.4

3.4.2 Weakness of Normal Derivation

As shown above the normal derivation present a great advantage, but also a weakness. It is possible to derive the **parent extended private key** knowing the **parent extended public key** and only one of the **child extended private key**.

The HMAC-SHA256 function has as input three elements: the parent chain code, the parent public key and the child index. The firsts two information can be taken from the parent extended public key, instead the child index can be taken from the child extended private key.

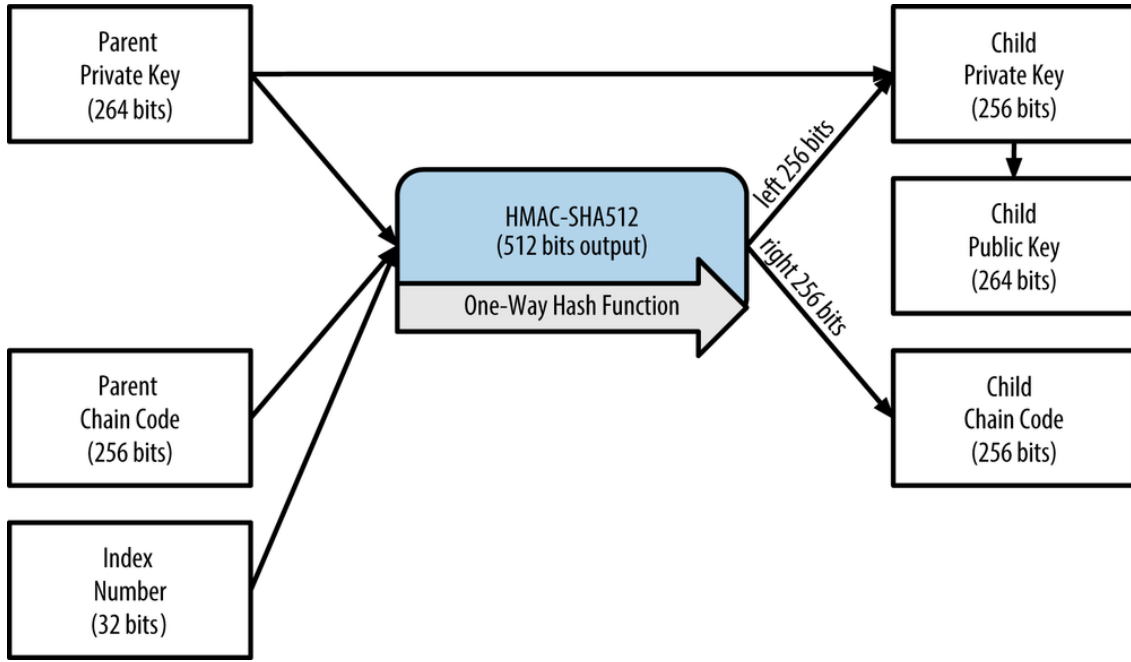


FIGURE 3.4: Hardened Derivation

$$msg = \text{compressed parent public key} \mid \text{child index}$$

Then apply the HMAC algorithm with the usual input:

- **Hash function:** SHA512
- **Key:** parent chain code
- **Message:** msg

Consider the first 32 bytes of the result of this function and consider it as an integer number, q .

Remembering that to get the child private key it is needed to compute a sum with the parent private key, it is possible to reverse the process.

Let's call p_{child} and p_{parent} the private keys of the child and the father respectively.

$$\begin{aligned}
 p_{child} &= q + p_{parent} \quad \text{mod } (order) \\
 \Downarrow \\
 p_{parent} &= p_{child} - q \quad \text{mod } (order)
 \end{aligned}
 \tag{3.1}$$

The implication 3.1 holds also with modular arithmetic.

So we have derived the private key of the parent. Graphically this derivation can be shown in figure 3.5

3.5 Advantages and disadvantages

We have seen that public-to-public operation is possible using the normal derivation. This is impossible with the hardened one:

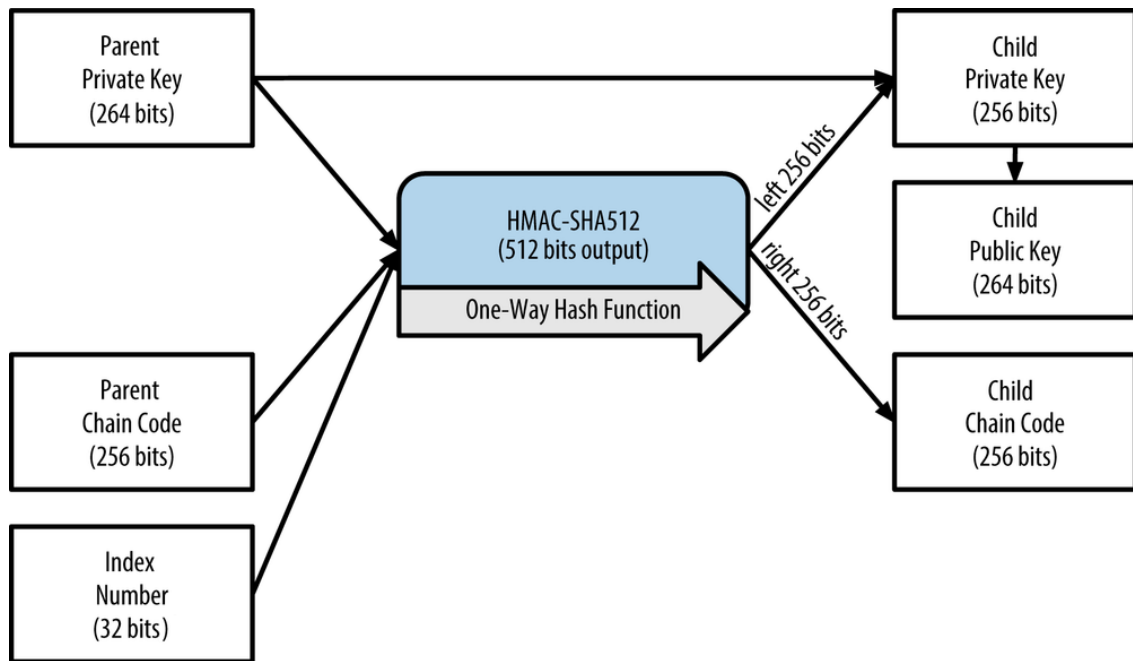


FIGURE 3.5: Hardened Derivation

The input of HMAC-SHA512 are different for the two derivation. If for the normal derivation is sufficient only the information in the public extended key, for the hardened derivation, the parent private key is needed. This makes impossible to obtain a public from a public, but also it makes impossible to derive the private key of the parent knowing the private key of the child and the public of the parent.

Knowing strengths and weaknesses of the two methods, it is advisable to use each of the two methods in the appropriate situations.

3.5.1 When use Normal Derivation?

Normal derivation should be used whenever all the child keys are collected in the same digital place and you should never give one key to someone else. As already mention a leak of a single private key can compromise the entire wallet.

However if all the child keys are used by the same person and you need to generate different public key, with this derivation it is possible to do so even in a "hot place". Let's suppose to have stored only the extended public key in a device, you can then receive payment but it is impossible to spend them as long as the private keys are hidden. If someone stole your device the only problem is a leak of privacy, because it is possible, by examining the blockchain, to discover all the UTXOs associated with that wallet.

3.5.2 When use Hardened Derivation?

Hardened derivation should be used whenever all the keys generated are used for different purpose and are maybe stored in different place. With this procedure it is possible to yield a branch of the tree to someone else in order to manage part of your money, without the risk to lose all the others keys in the wallet.

As a best practice it is always advisable to derive with the hardened method from the master. An hardened key should have both hardened or normal children, but from a normal child it is not reasonable to derive an hardened one, because it make no sense to decrease the security of the wallet at the last level.

Chapter 4

Mnemonic phrase

We have seen how it is possible to generate keys starting from a seed. But a seed is a long number, difficult to remember and not easy to write down on a paper. You may incur typos while transcribing it, and this can compromise the entire wallet.

Remark *Mistyping a single digit in the seed produce completely different keys*

In order to work around this problem, some solution were implemented. Among them, the most widespread and used is the one described by BIP39, *Bitcoin Improvement Proposal*. This is not the only one, in this chapter we will also see another solution proposed by Electrum, one of the most famous Bitcoin wallet.

Both these solutions used a *Mnemonic phrase*, from which the seed is obtained. This sentence is designed to avoid typing errors while maintaining the same level of security and entropy.

What is a Mnemonic phrase?

A Mnemonic phrase is a set of words taken from a specific dictionary. Although the choice of the dictionary is not binding, the most commonly used among the practitioners is the English one, defined by BIP39. It contains 2048 common words of the English language, each of them from 3 to 9 letters. The set of words that make up the dictionary must be chosen in such a way that the words within it are easy to remember and difficult to misinterpret with one another. It is better to avoid to insert into the dictionary two words with similar meaning or spelling.

4.1 BIP39

First we will see how to generate a mnemonic phrase in the framework of BIP39 and then how it is possible to obtain a seed from it.

4.1.1 Mnemonic Generation

In order to generate a Mnemonic phrase we will start from a given entropy, that can be seen as a large integer number. The way to obtain it can be left free to the user: he can obtain it by inserting arbitrarily chosen numbers (poor choice of randomness), roll a dice many times or with any other method he considers suitable. Many softwares provide a function that generate entropy with quite randomness, but if someone is skeptical about the reliability of software randomness, he must provide himself with such integer number.

Call ENT the number of binary digits of the given entropy. Then ENT should belong to a given set:

$$ENT \in \{128, 160, 192, 224, 256\}$$

The reason for a given length for the entropy will be clear in a moment.

Write the entropy in bytes format, obtaining a string of $ENT/8$ length. Compute the SHA256 algorithm and consider only the first $ENT/32$ bits as a checksum. Finally add these bits to the bottom of the entropy, obtaining an integer number, called *entropy_checked*, expressed in binary format of length equal to: $ENT + ENT/32$.

In python:

```
1 from hashlib import sha256
2
3 entropy_bytes = entropy.to_bytes(int(ENT/8), byteorder='big')
4 checksum = sha256(entropy_bytes).digest()
5 entropy_checked = entropy_bin + checksum_bin[:int(ENT/32)]
```

Where *entropy_bin* and *checksum_bin* are strings of bits that can be concatenated.

Now it is clear the reason for a constraint on the length of the entropy in input:

- i ENT must be a dividend of 32
- ii $ENT < 128$ could be not secure enough.
- iii $ENT > 256$ is useless.

The point (i) is due to the structure proposed by BIP39. It is only a convention to take the first $ENT/32$ bits as a checksum. However it is essential that the final length of the entropy plus the checksum must be a dividend of 11, from the moment that the dictionary is a set of 2^{11} words.

The point (ii) is just a suggestion because taking less entropy could bring to a leak of security. It will be easier for an attacker to guess your mnemonic phrase by trying out all the possible combinations if less words are involved. It is important to remember that adding even a single bit of entropy, doubles the difficulty of guessing it.

The point (iii) is another suggestion. A private key is a number smaller than 2^{256} therefore it would be useless to generate a seed starting from an entropy with more than 256 bits.

Thanks to constraint (i) we obtain that the length of *entropy_checked* is a dividend of 11:

$$\text{len}(\text{entropy_checked}) = ENT + \frac{ENT}{32} = \frac{33}{32} \cdot ENT = 11 \cdot \frac{3}{32} \cdot ENT$$

Consider *entropy_checked* as a string of bits and divide it in substring, each of 11 bits length, obtaining $(\frac{3}{32} \cdot ENT)$ strings of bits.

Each of these strings represents an integer number that can take values in the range between 0 and 2047, ie $2^{11} - 1$. Associate each of these numbers with a word in the

chosen dictionary, suppose to consider the English one sorted alphabetically. Write down all these words, separated by a space and obtain the Mnemonic Phrase.

All these steps can be summarize with the follow scheme, ($ENT = 128$):

$$\begin{array}{c}
 entropy_{16} = f012003974d093eda670121023cd03bb \\
 \Downarrow \\
 entropy_2 = \underbrace{1111000000010010000000...0111011}_{SHA256} \\
 \Downarrow \\
 \underbrace{0010}_{check\ sum} \quad \underbrace{010001000001001...}_{ignored} \\
 entropy\ checked = 1111000000010010000000...0111011 \mid 0010 \\
 \Downarrow \\
 \underbrace{11110000000}_{1920 \downarrow useless} \quad \underbrace{10010000000}_{1152 \downarrow mosquito} \quad \underbrace{...01110110010}_{946 \downarrow iron}
 \end{array}$$

Obtaining a sequence of 12 words:

useless mosquito atom trust ankle walnut oil across awake bunker domain iron

4.1.2 From Mnemonic to Seed

Once obtaining the Mnemonic phrase we need to derive the seed. In order to do so an hash function is used. So it will be infeasible to derive the Mnemonic phrase from the seed.

The function used is the PBKDF2 and it is used in order to avoid brute force attack, from the moment that the output has exactly the same length of a standard hash function, but it will take more times to calculate it from the moment that it will compute the same hash function many times.

It receives as input:

- ⊙ **Password:** Mnemonic phrase
- ⊙ **Salt:** 'mnemonic' + passphrase
- ⊙ **Number of iterations:** 2048
- ⊙ **Digest-module:** SHA512
- ⊙ **Mac-module:** HMAC

Summing up it can be said that it calculates the same hash function (HMAC-SHA512) 2048 times.

In order to introduce more complexity in the seed computation a *Salt* is introduced. If not specified the standard salt is simply the world 'mnemonic', otherwise it could be extended with an optional *passphrase*.

Although it is true that a human being is a scarce source of randomness, the passphrase is usually chosen by the user. This is due to the fact that it has a different purpose to the mnemonic phrase. We will analyze some useful applications in the next chapter.

Remark *The randomness should be guaranteed by the input entropy used to generate the mnemonic phrase, not by the passphrase.*

The python code is the follow:

```
1 from hashlib import sha512
2 from pbkdf2 import PBKDF2
3 import hmac
4
5 seed = PBKDF2(mnemonic, 'mnemonic' + passphrase, iterations = 2048,
               macmodule = hmac, digestmodule = sha512).read(64)
```

Where *mnemonic* is the mnemonic phrase previously computed and *passphrase* is chosen by the user

Remark *With this procedure we always produce a seed of specific length: 512 bits. It will always be enough because every private key can take value from a smaller set of value (1 to order).*

4.2 Electrum Mnemonic

Even if BIP39 is proposed, it is not the only solution adopted among the practitioners. One example is the one proposed by Electrum¹.

The main difference is in the way that the mnemonic phrase is generated and the purpose of it. Electrum chooses to assign a version to the seed in such a way that is possible to recognize the purpose of the keys and the way to generate them.

4.2.1 Mnemonic Generation

Whenever a new mnemonic phrase is required, Electrum starts from an entropy, generated through a random function. Obviously it is possible to generate a valid mnemonic phrase with an entropy chosen by the user, if he is skeptical or doesn't want to rely on the reliability of the randomness of random function.

To be consistent with the BIP39 section, consider the entropy as a large integer number and call *ENT* the number of its binary digits. Then *ENT* must be a multiple of 11 if the chosen dictionary is the same of BIP39. However the choice of dictionary is not binding.

The first important difference with BIP39 mnemonic is that the checksum is not performed on the entropy but on the Mnemonic phrase directly. In order to obtain a valid mnemonic phrase, the following instruction must be followed:

1. Divide the *entropy* in string of 11 bits each.
2. Associate each string with a word from the chosen dictionary of 2048 words.
3. Write down all these words, separated by a space and obtain a *candidate* Mnemonic phrase
4. Compute a particular HASH function on the Mnemonic phrase previously obtained and verify that the first digits correspond to the digits of the chosen version of the seed:

¹Electrum is a lightweight Bitcoin client, released on November 5, 2011

- '0x01' for a standard type seed
- '0x100' for a segwit type seed
- '0x101' for a two-factor authenticated type seed

5. If the initial digits are different, increase entropy by one and then go back to point 1, otherwise you have obtained a valid mnemonic phrase.

Point 5. has the simple effect of changing a single word of the mnemonic phrase and this will lead to a complete different HASH. Do this over and over again, until the first digits match the version digits required.

This procedure can be shown with the following python instruction:

```
1 import binascii
2 import hmac
3 from hashlib import sha512
4
5 def verify_mnemonic_electrum(mnemonic, version = "standard"):
6     x = hmac.new(b"Seed version", mnemonic.encode('utf8'), sha512).digest()
7     s = binascii.hexlify(x).decode('ascii')
8     if s[0:2] == '01':
9         return version == "standard"
10    elif s[0:3] == '100':
11        return version == "segwit"
12    elif s[0:3] == '101':
13        return version == "2FA"
14    else:
15        return False
16    return True
17
18 def mnemonic_electrum(entropy, number_words, version, dictionary):
19     is_verify = False
20     while not is_verify:
21         mnemonic = from_entropy_to_mnemonic(entropy, number_words, dictionary)
22         is_verify = verify_mnemonic_electrum(mnemonic, version)
23         if not is_verify:
24             entropy = entropy + 1
```

Let's see an example: suppose to be looking for a standard type seed, starting with $ENT = 132$

$$\begin{array}{c}
 entropy_{16} = ef938205cd78ab6d876398dcfd65dae32 \\
 \Downarrow \\
 entropy_2 = \underbrace{11101111100}_{1916 \downarrow usage} \underbrace{10011100000}_{1248 \downarrow orchard} \underbrace{10000001011}_{1035 \downarrow lift} \dots \underbrace{11000110010}_{1586 \downarrow shock}
 \end{array}$$

Mnemonic = *usage orchard lift online melt replace budget indoor table twenty issue shock*

$$HASH(\text{Mnemonic}) = 3d5d23737859601eeabe32d1e1\dots$$

Does $HASH(\text{Mnemonic})$ starts with '01'? \Rightarrow NO

$$\begin{array}{c}
 entropy_{16} = entropy_{16} + 1 = ef938205cd78ab6d876398dcfd65dae33 \\
 \Downarrow \\
 entropy_2 = \underbrace{11101111100}_{1916 \downarrow usage} \underbrace{10011100000}_{1248 \downarrow orchard} \underbrace{10000001011}_{1035 \downarrow lift} \dots \underbrace{11000110011}_{1587 \downarrow shoe}
 \end{array}$$

Mnemonic = *usage orchard lift online melt replace budget indoor table twenty issue shoe*

$$\text{HASH}(\text{Mnemonic}) = 2a3b2cf6a0506844a77baf8175...$$

Does $\text{HASH}(\text{Mnemonic})$ starts with '01'? \Rightarrow NO

After other 443 attempts we obtain:

$$\begin{array}{c} \text{entropy}_{16} = ef938205cd78ab6d876398dcfd65dafee \\ \downarrow \\ \text{entropy}_2 = \underbrace{11101111100}_{1916 \downarrow \text{usage}} \underbrace{10011100000}_{1248 \downarrow \text{orchard}} \underbrace{10000001011}_{1035 \downarrow \text{lift}} \dots \underbrace{1111101110}_{2030 \downarrow \text{worry}} \end{array}$$

Mnemonic = *usage orchard lift online melt replace budget indoor table twenty issue worry*

$$\text{HASH}(\text{Mnemonic}) = 01d133fdcc54bd0da3d717173e0f82127...$$

Does $\text{HASH}(\text{Mnemonic})$ starts with '01'? \Rightarrow YES

So we have finally found a valid Mnemonic phrase for the standard type seed.

4.2.2 From Mnemonic to Seed

Once obtaining the Mnemonic phrase the seed is derived in the same way described in BIP39, with only one exception:

The **Salt** used for the PBKDF2 function does not contain the word 'mnemonic', instead it contains the word 'electrum'. It is always concatenate with a *passphrase*, chosen by the user.

The python code it is therefore the following:

```
1 from hashlib import sha512
2 from pbkdf2 import PBKDF2
3 import hmac
4
5 seed = PBKDF2(mnemonic, 'electrum' + passphrase, iterations = 2048,
               macmodule = hmac, digestmodule = sha512).read(64)
```

4.3 Comparison

Once described how to generate the Mnemonic phrase for each of the two principal proposals, let's analyse the advantages and disadvantages.

Both BIP39 and Electrum are secure from a so called *brute force attack*, from the moment that the function PBKDF2 is used and so it is infeasible to guess a Mnemonic randomly generated by another user.

From a security point of view the two methods are really similar, indeed both of them use the last part of the phrase as a "checksum", but in two different ways. Indeed the first important difference is in the way that a Mnemonic phrase is considered valid. For BIP39, it is valid if and only if the checksum is correct. In order to

Chapter 5

Child Key Derivation

5.1 Functional explanation

5.2 Normal derivation

5.2.1 Derive public child from public parent

5.2.2 Possible Risk

5.3 Hardened derivation

Chapter 6

Mnemonic to Seed

6.1 Functional explanation

6.2 BIP 39 derivation

6.2.1 Mnemonic generation

6.2.2 Seed derivation

6.3 Electrum derivation

6.3.1 Mnemonic generation

6.3.2 Seed derivation

6.4 BIP39 vs Electrum derivation

Chapter 7

How to use a HD Wallet

7.1 Multi-coin wallet BIP 44

Appendix A

Frequently Asked Questions

A.1 How do I change the colors of links?

The color of links can be changed to your liking using:

```
\hypersetup{urlcolor=red}, or  
\hypersetup{citecolor=green}, or  
\hypersetup{allcolor=blue}.
```

If you want to completely hide the links, you can use:

```
\hypersetup{allcolors=.}, or even better:  
\hypersetup{hidelinks}.
```

If you want to have obvious links in the PDF but not the printed text, use:

```
\hypersetup{colorlinks=false}.
```