POLITECNICO DI MILANO

DOCTORAL THESIS

Elliptic Curve Hierarchical Deterministic Private Key Sequences: Bitcoin Standards and Best Practices

Author:
Daniele FORNARO

Supervisor:
Daniele MARAZZINA

A thesis submitted in fulfillment of the requirements for the degree of Mathematical Engeneering

in the

Research Group Name
Department or School Name

March 10, 2018

Declaration of Authorship

"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

POLITECNICO DI MILANO

Abstract

Faculty Name
Department or School Name

Mathematical Engeneering

Elliptic Curve Hierarchical Deterministic Private Key Sequences: Bitcoin Standards and Best Practices

by Daniele FORNARO

The cryptography used by most of the cryptocurrencies is mainly based on the private-public key pair. It is therefore fundamental the method used to generate private keys, which must be efficient, secure and suitable for the situation. Among the various methods used by now, it has become a standard the one described in BIP32, the Hierarchical Deterministic Wallet. Starting from a sufficiently large random number, called SEED, it is possible to generate numerous private keys in a hierarchical and deterministic way through particular HASH functions and thanks to the elliptic curve properties. Several wallets also use a special algorithm to store the seed and to be able to back it up in a readable form, through the use of the mnemonic phrase, words selected from a specific dictionary. Although it is trying to reach consensus on a single standard to use, not all major players in the industry use the same method. This paper aims to clarify the various techniques used for the derivation of the keys, with particular attention to the HD wallet. It will also be analyzed the two principal way of encoding the seed, the one described into BIP39 as opposed to the proposal of Electrum, one of the main Bitcoin Wallet, highlighting their respective advantages and disadvantages.

Acknowledgements

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

Contents

For/Dedicated to/To my…

Chapter 1

Cryptography

In order to have a clear understanding of this thesis, it is necessary to know the basic concepts of:

- ✓ HASH function
- ✓ Elliptic Curve

Only these two elements together can describe all the cryptography behind Bitcoin.

1.1 HASH function

In general a hash function is a mathematical process that takes input data of *any* size, performs an operation on it, and returns output data of a *fixed* size.

The input data is called *message* and the output data is called *hash value*.

A hash function must have at least these 6 properties:

(i) It is **deterministic**: if the message remains unchanged, the hash value is the same.

(ii) It is **quick**: it should not take too much time to compute the hash value from the message.

(iii) It is a **ONE-WAY function**: it is infeasible to find a message, knowing the hash value. The only way to find the message must be to try randomly all the possible combination.

(iv) It is **collision free**: it is infeasible to find two messages with the same hash value, even if it is theoretically possible.

(v) It has the **avalanche effect**: a very small change in the input message, even flipping a single bit, produces a completely different hash value.

(vi) It has **fixed size** output and could have input messages of **any size**.

In this thesis we will see hash functions as black-boxes, with all the proprieties described above.

There are various kinds, but for our purpose the main difference lies in the number of bits of the hash value. Among all the possible hash function, in Bitcoin cryptography 3 functions are used:

- **SHA256**: Secure Hash Algorithm 256

  - developed by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard (FIPS).
  - output size: 256 bits.

- **SHA512**: Secure Hash Algorithm 512
  - developed by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard (FIPS).
  - output size: 512 bits.
- **RIPEDM160**: RACE Integrity Primitives Evaluation Message Digest 160
  - developed by Hans Dobbertin, Antoon Bosselaers and Bart Preneel at the COSIC research group at the Katholieke Universiteit Leuven.
  - output size: 160 bits.

One important hash function used in the Bitcoin cryptography is the so called HASH160. It is simple the concatenation of SHA256 and RIPEDM160:

$$HASH160\,(\,msg\,) = RIPEDM160\,(\,SHA256\,(\,msg\,))$$

From the moment that the last operation made to compute the HASH160 function was the RIPEDM160 function, the output size is 160 bits.

1.1.1 Other functions

There are two other functions that will be heavily used in this thesis:

- **HMAC**: Hash-based Message Authentication Code
- **PBKDF2**: Password-Based Key Derivation Function 2

HMAC is a function that made some computation involving also a hash function. This algorithm provides better immunity against *length extension attacks*.

It receives 3 inputs:

- ⊙ **Hash function**: a hash function with the properties described above.
- ⊙ **Key**: a sequence of bytes.
- ⊙ **Message**: a sequence of bytes.

This function compute the following operations:

$$HMAC(H, k, m) = H\,(\,opad(k)\,||\,H\,(\,ipad(k)\,||\,m\,))$$

Where H is the hash function, k is the key, m is the message, $opad(\bullet)$ and $ipad(\bullet)$ are two padding function, applied to the key k and || is a symbol that denotes concatenation.

PBKDF2 is a function that applied a hash function to an input (message) along with a salt and repeat the process many times. This algorithm provides more computational work with respect to a single hash function, and so it reduces the risk of a brute force attack.

It receives 5 inputs:

- ⊙ **Message**: a sequence of bytes
- ⊙ **Salt**: a sequence of bytes
- ⊙ **Number of iterations**: the number of iteration to be computed.
- ⊙ **Digest-module**: a hash function with the properties described above.
- ⊙ **Mac-module**: a message authentication code module (e.g. HMAC).

Having a salt reduce the ability to use rainbow tables for attack. It is recommends to use at least 64 bits for the salt.

1.2 Elliptic Curve over $\mathbb{F}_p$

Elliptic curve is a *plane algebraic curve* defined by an equation, over a specific field. In cryptography the field is finite.

A point Q, which coordinates are x and $y \in \mathbb{N}$, belong to an Elliptic Curve if and only if Q satisfies the following equation:

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

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:

$$y^2 = x^3 + ax + b \mod p \tag{1.2}$$

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

1.2.1 Symmetry

The elliptic curve has an important property: the line $y = p/2$ is an axis of symmetry for the curve.

This can be shown, by proving that the point $P(x,y)$ belongs to the Elliptic Curve (EC) if and only if the point $Q(x, p-y)$ belongs to the curve too:

$$P(x,y) \in EC \iff Q(x, p-y) \in EC$$

Proof:

First analyze the implication in the right direction: ($\implies$).

From equation 1.2 and from the hypothesis we have that:

$$P(x,y) \in EC \implies y^2 = x^3 + ax + b \mod p$$

But we know also that:

$$Q(x,p-y) \in EC \iff (p-y)^2 = x^3 + ax + b \mod p$$

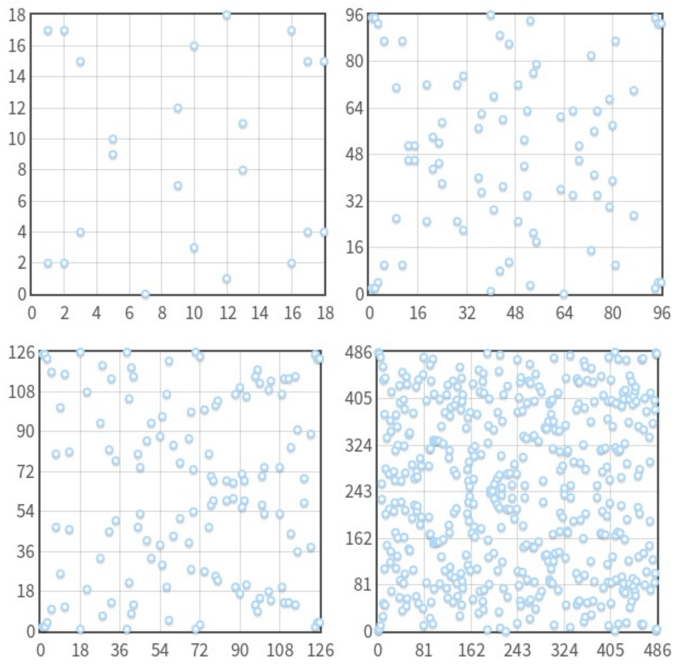

FIGURE 1.1: Points on the Elliptic Curve $y^2 = x^3 - 7x + 10 \mod p$, with $p = 19, 97, 127, 487$

From the moment that the right hand side of both the equations are equal, we only need to prove that:

$$(p - y)^2 = y^2 \mod p$$

This is true, indeed:

$$\begin{aligned}
(p-y)^2 &= p^2 - 2py + y^2 && \mod p \\
&= p \cdot (p - 2y) + y^2 && \mod p \\
&= 0 + y^2 && \mod p \\
&= y^2 && \mod p
\end{aligned}$$

This is due to the fact that

$$p \cdot k = 0 \mod p \qquad \forall k \in \mathbb{N}$$

The other implication ($\Longleftarrow$) is almost the same and follow the same logic.

c.v.d.

Once shown the symmetry property, it can be useful to denote the point $P(x, y)$ as the opposite of $Q(x, p - y)$:

$$P = -Q \implies P + Q = 0$$

Where the + operator between two point in the EC will be explained below.

1.2.2 Point addition

Once defined a point on the elliptic curve, let's introduce the addition between two point on this finite field.

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 some 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

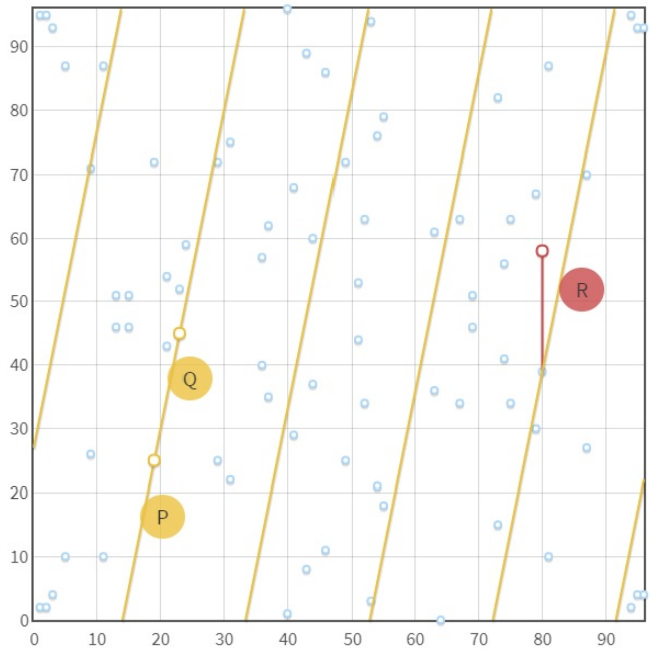

FIGURE 1.2: Elliptic Curve $y^2 = x^3 - 7x + 10 \bmod 97$

After defined when points in the EC have zero sum, it is possible to calculate the equations for point addition:

Suppose that A and B belong to the Elliptic Curve.

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

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

When $x_1 = x_2$ but $y_1 \neq y_2$, it is the case in which A and B are symmetric point and so the sum is zero:

$$A + B = 0$$

In all the other case we have:

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

Where s is a dummy variable, used to compute x_3 and y_3. It can be computed in two different way if we are performing a "real" point addition, when $A \neq B$ or if we are looking for the double of a point, when $A = B$.

$$x_3 = s^2 - x_1 - x_2 \mod p$$
$$y_3 = s(x_1 - x_3) - y_1 \mod p$$

Once we have s the value x_3 and y_3 are obtained followed this simple formula.

1.2.3 Scalar multiplication

Once defined the addition, any multiplication between a scalar and a point on the elliptic curve can be defined as:

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

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. Let's see and example:

Suppose that we need to compute $53 \cdot P$ where P is a point on the EC:

$$53 \cdot P = 110101_{base\,2} \cdot P = 2^5 \cdot P + 2^4 \cdot P + 2^2 \cdot P + P$$

Computing the common sub-terms only once we obtain a total of 5 doubling and 3 addition operations, much less of 52 addition operations. This algorithm is even more efficient if the scalar is a very large number.

1.2.4 Discrete Logarithm Problem

Once we have described the multiplication between a scalar and a point, let's see if it is possible to make the inverse operation. Let's suppose that:

$$Q = n \cdot P$$

Where Q and P are point on the EC and n is a scalar number.

Let's suppose to know Q and P. With these information it exists only one possible $n \in \mathbb{N}$, such that $0 < n < p$ and that the equation above holds true. Even so this number n is infeasible to find for large value of p.

This is due to the fact that there is not an efficient algorithm that is able to compute n given P and Q. The only way to find n is by trying. As already mention, this could became infeasible if the number of value that n can assume $(p - 1)$ is too large.

1.2.5 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 p 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*.

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

$$nG + mG = \underbrace{G + \cdots + G}_{n \text{ times}} + \underbrace{G + \cdots + G}_{m \text{ times}} = \underbrace{G + \cdots + G}_{n+m \text{ times}} = (n + m)G$$

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

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

Remark *The order of the subgroup generated by G 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 points belonging to the EC generate a subgroup with the same order of the group or with order 1.*

All these preliminary information are needed in order to introduce the private-public key cryptography used by Bitcoin.

1.2.6 Bitcoin private-public key cryptography

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:

$$y^2 = x^3 + 7 \mod p \tag{1.3}$$

The *mod 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 then p.

Let's consider a particular point G, called generator, expressed in hexadecimal digits:

$$x = 79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798$$
$$y = 483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8$$

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*.

We have now all the elements necessary to define the private and the public key.

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:*

$$W = k \cdot G \tag{1.4}$$

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.*

The purpose of having defined the private and public keys is to use them to cryptographically sign a message. It is not the scope of this thesis explain how a message is signed, but it is at least necessary to know the principal properties of a signed message.

Let's suppose to have a message that is needed to be signed, in Bitcoin this message is usually a *transaction*.

- A message is signed using a **private key**.
- Knowing the **public key** associated to the private key that sign the message, it is possible to verify that the message is signed using the corresponding private key (without knowing it).

For this reason the keys are called private and public, the private one is suppose to be kept **secret**, because it is able to sign a message, instead the other one is suppose to be **shared**, in order to let everyone else knows that who signs the message is in possession of the corresponding private key.

Chapter 2

Wallet

The way used to store keys is essential in private-public keys cryptography. For this reason different type of wallet were designed.

Definition *A wallet is a software used to store keys. It is able also to sign messages with the private key, but in this framework we will only consider wallets as key containers.*

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 $k_1, k_2 ... k_n$ of X using enough entropy to make these numbers (*private keys*) impossible to guess.

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

(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 computed the *private key*.

Pros and Cons

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

Random Wallet	
Pros	**Cons**
• Easy to implement	• Difficult to find real new entropy for every new *private key*. • Every time new *private keys* are needed, a new back up is needed. • Difficult to store or back up in a *non digital way*. 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* leads to know also 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, the **SHA256** function.

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

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

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

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.

Deterministic Wallet type-1

Pros	**Cons**
• In order to make a back up of the entire wallet it is needed to store the *seed* only. All *private keys* can be derived from it. • A single back up is needed. • The *seed* can be stored easily also in a *non digital way*, in a paper for example.	• Every time new *public keys* are needed, you need to use the *seed*, to compute new *private keys* and then derive the *public* ones. This could compromise the entire wallet, if the *seed* is used in a non safe environment. • There is only a *key* sequence. No way to distinguish the "purpose" of each *private key*.

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*

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

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*

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

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:

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

$h_{n|r}$ 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 $h_{n|r}$:

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

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.

Deterministic Wallet type-2

Pros	**Cons**
• In order to make a back up of the entire wallet it is needed to store the *master private key* and the random number *r*. All *private keys* can be derived from them. • A single back up is needed. • The *master private key* can be stored easily also in a *non digital way*, in a paper for example. • It is possible to derive a new *public key* using only non-secret information, with the procedure above.	• There is only a *key* sequence. No way to distinguish the "purpose" of each *private key*.

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, obtained from the public keys, also in a non safe environment, having only r and MP.

A thief can only steal your privacy, if he steal only MP and r. In fact he is able to see which messages (transaction) have you signed, without the possibility to sign new ones and spend your money.

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 this wallet works:

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

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

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^{32} *private key* using a irreversible hash function: SHA512

(iv) All of this *private key* "children" can derive 2^{32} *private key* and all of these "grand-children" 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 keys 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*

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

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=fffcf9f6f3f0edeae7e4e1dedbd8d5d2cfccc9c6c3c0bdbab7b4b1aeaba8a5a29f9c999
693908d8a8784817e7b7875726f6c696663605d5a5754514e4b484542

3.1.2 Extended Key

An Extended Key is a sequence of bytes, encoded in base58. It contains all the information necessary derive the keys. 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:

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

  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 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 compress 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 the length of 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:

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

Where *seed* is an integer number, *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:

```
from hashlib import sha512
from hmac import HMAC

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:

```
private_key_bytes = hashValue[0:32]
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:

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

Then the Master Extended Private Key is formed by concatenation:

```
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 obtained 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

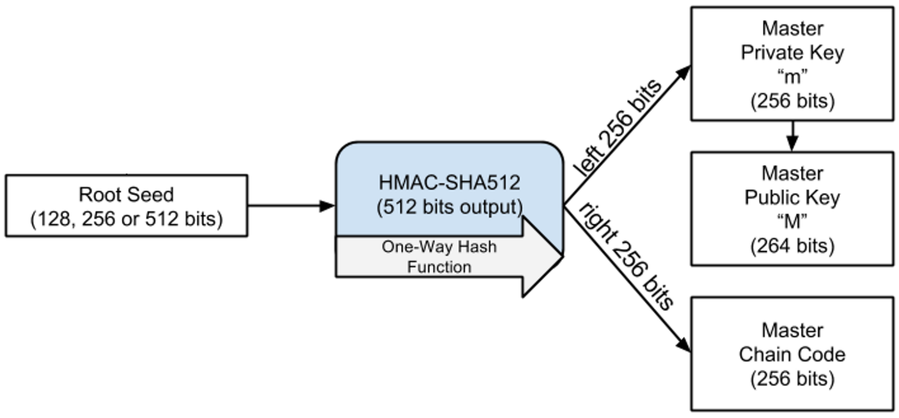

FIGURE 3.1: From seed to master private key

3.3 Child Key derivation

From any extended private key it is possible to obtain different child keys. There are two method 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 fit.

For both the method the derivation starts from a Extended Private Key. From this key some essential information are 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:

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

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

First we need to compute the Parent Public Key P. This is obtained from the usual scalar multiplication between a point on the EC (the Generator G) and 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:

$$msg = compressed\ public\ key \mid index$$

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:

```python
from hmac import HMAC
from hashlib import sha512

msg=parent_public_key + index
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:

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

Graphically these operation can be shown in figure 3.2

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:

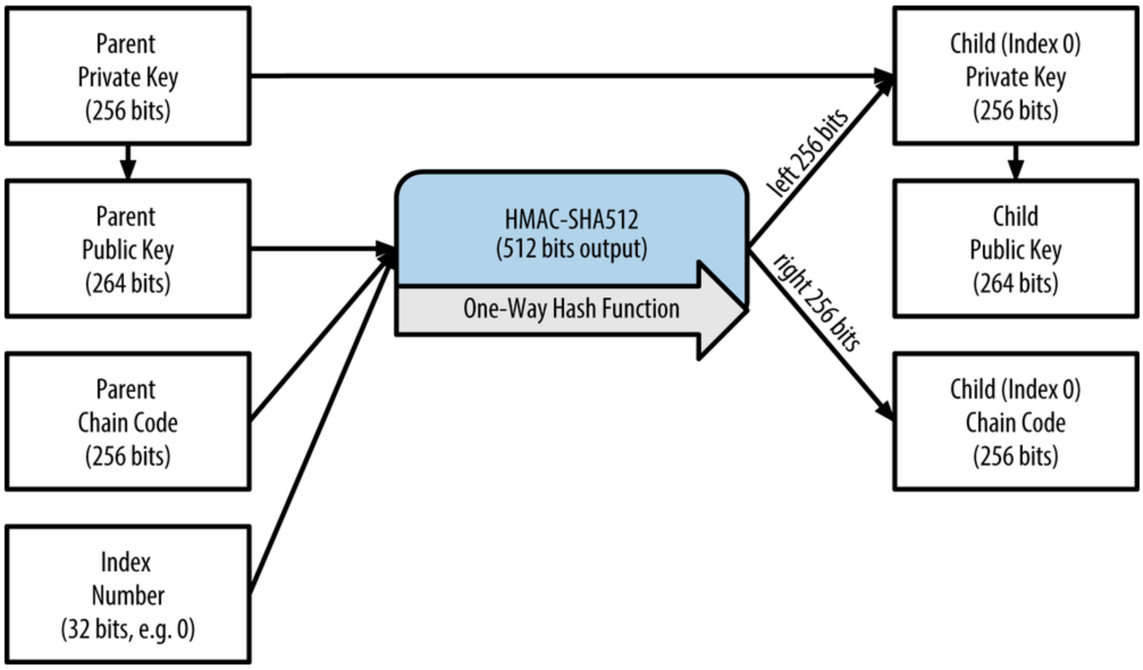
Parent Private Key (256 bits)
Parent Public Key (264 bits)
Parent Chain Code (256 bits)
Index Number (32 bits, e.g. 0)
HMAC-SHA512 (512 bits output)
One-Way Hash Function
left 256 bits
right 256 bits
Child (Index 0) Private Key (256 bits)
Child Public Key (264 bits)
Child (Index 0) Chain Code (256 bits)

FIGURE 3.2: Normal Derivation

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

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:

```python
from hmac import HMAC
from hashlib import sha512

msg=parent_private_key + index
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:

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

Graphically these operation can be shown in figure 3.3

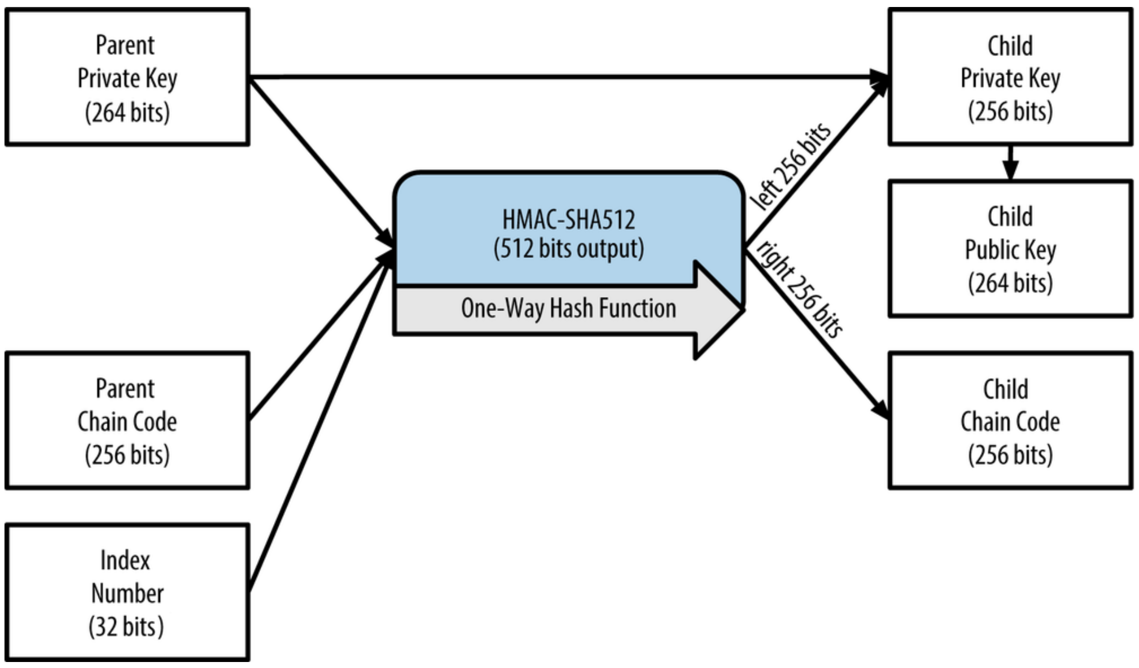

FIGURE 3.3: Hardened Derivation

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:

$$msg = compressed\ public\ key \mid index$$

msg is a string of 37 bytes.

Apply the HMAC algorithm with the following input:

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

The 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.

$$msg = compressed\ parent\ public\ key \mid child\ index$$

Then apply the HMAC algorithm with the usual input:

- ⊙ **Hash function**: SHA512
- ⊙ **Key**: parent chain code

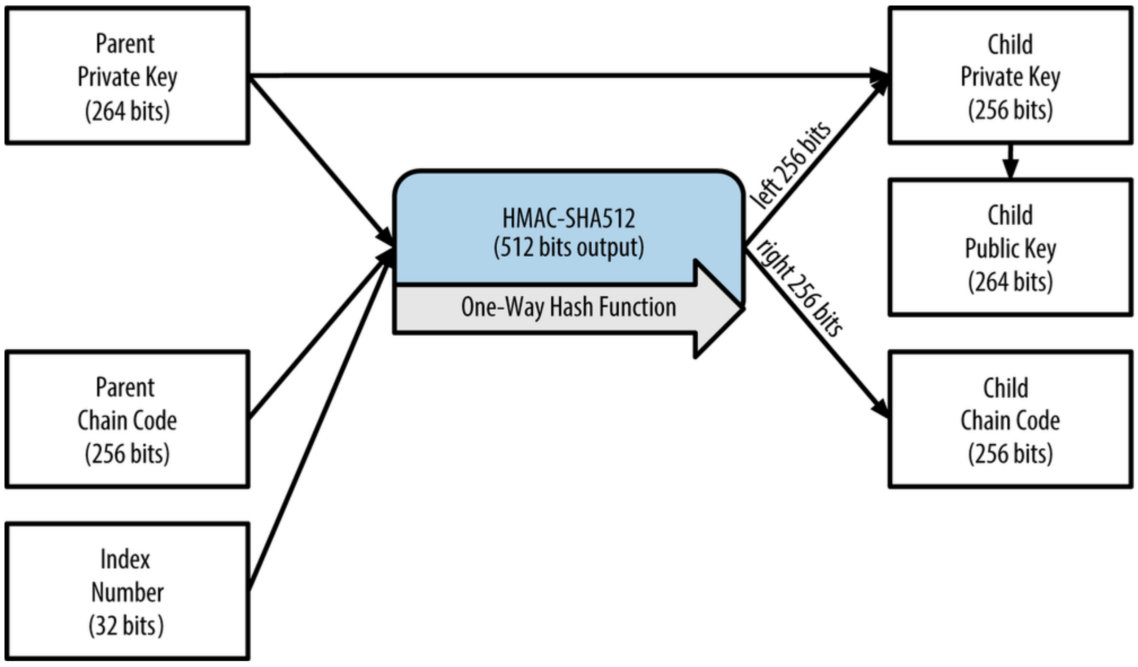

FIGURE 3.4: Hardened Derivation

- ⊙ **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.

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

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:

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.

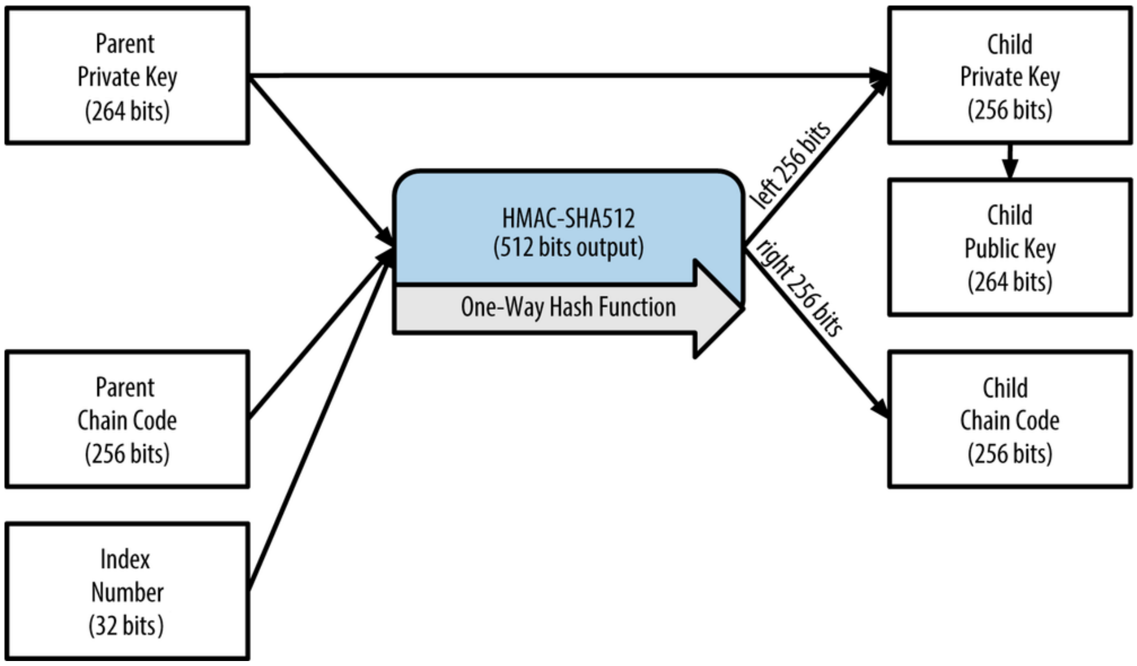

FIGURE 3.5: Hardened Derivation

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 belongs 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:

```python
from hashlib import sha256

entropy_bytes = entropy.to_bytes(int(ENT/8), byteorder='big')
checksum = sha256(entropy_bytes).digest()
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 then 2^{256} therefore it would be useless to generate a seed starting from an entropy with more then 256 bits.

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

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

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, ($ENT = 128$):

$$entropy_{16} = f012003974d093eda670121023cd03bb$$
$$\Downarrow$$
$$entropy_2 = \underbrace{111100000001001000000000...0111011}_{\substack{SHA256 \\ \Downarrow}}$$
$$\underbrace{0010}_{check\ sum}\ \ \underbrace{010001000001001...}_{ignored}$$
$$entropy\ checked = 111100000001001000000000...0111011 \mid 0010$$
$$\Downarrow$$
$$\underbrace{11110000000}_{\substack{1920 \\ \Downarrow \\ useless}}\underbrace{10010000000}_{\substack{1152 \\ \Downarrow \\ mosquito}}\ldots\underbrace{01110110010}_{\substack{946 \\ \Downarrow \\ iron}}$$

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:

```python
from hashlib import sha512
from pbkdf2 import PBKDF2
import hmac

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 Electrum[1].

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 en 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 relay 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:

[1]Electrum 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:

```python
import binascii
import hmac
from hashlib import sha512

def verify_mnemonic_electrum(mnemonic, version = "standard"):
  x = hmac.new(b"Seed version", mnemonic.encode('utf8'), sha512).digest()
  s = binascii.hexlify(x).decode('ascii')
  if s[0:2] == '01':
    return version == "standard"
  elif s[0:3] == '100':
    return version == "segwit"
  elif s[0:3] == '101':
    return version == "2FA"
  else:
    return False
  return True

def mnemonic_electrum(entropy, number_words, version, dictionary):
  is_verify = False
  while not is_verify:
    mnemonic = from_entropy_to_mnemonic(entropy, number_words, dictionary)
    is_verify = verify_mnemonic_electrum(mnemonic, version)
    if not is_verify:
      entropy = entropy + 1
```

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

$$entropy_{16} = ef938205cd78ab6d876398dcfd65dae32$$
$$\Downarrow$$
$$entropy_2 = \underbrace{11101111100}_{\substack{1916\\ \Downarrow\\ usage}}\underbrace{10011100000}_{\substack{1248\\ \Downarrow\\ orchard}}\underbrace{10000001011}_{\substack{1035\\ \Downarrow\\ lift}}\ldots\underbrace{11000110010}_{\substack{1586\\ \Downarrow\\ shock}}$$

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

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

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

$$entropy_{16} = entropy_{16} + 1 = ef938205cd78ab6d876398dcfd65dae33$$
$$\Downarrow$$
$$entropy_2 = \underbrace{11101111100}_{\substack{1916\\ \Downarrow\\ usage}}\underbrace{10011100000}_{\substack{1248\\ \Downarrow\\ orchard}}\underbrace{10000001011}_{\substack{1035\\ \Downarrow\\ lift}}\ldots\underbrace{11000110011}_{\substack{1587\\ \Downarrow\\ shoe}}$$

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

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

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

After other 443 attempts we obtain:

$$entropy_{16} = ef938205cd78ab6d876398dcfd65dafee$$
$$\Downarrow$$
$$entropy_2 = \underbrace{11101111100}_{\substack{1916\\ \Downarrow \\ usage}}\underbrace{10011100000}_{\substack{1248\\ \Downarrow \\ orchard}}\underbrace{10000001011}_{\substack{1035\\ \Downarrow \\ lift}}...\underbrace{11111101110}_{\substack{2030\\ \Downarrow \\ worry}}$$

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

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

Does $HASH$(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:

```python
from hashlib import sha512
from pbkdf2 import PBKDF2
import hmac

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 to generate the seed from the mnemonic phrase and so it is infeasible to guess a Mnemonic randomly generated by another user. In fact each time a valid phrase has been found, it is required to compute the seed, then the first child keys and then look at the public ledger, the blockchain, in order to see if some of this private keys are used to sign a transaction. All these passages are computational consuming and it would take too much time to be tried, even with all the most powerful computer in the world working together, it will take a time of many order of magnitude greater then the age of the universe itself.

Even if they are both secure, there is a difference. Suppose to be looking for a 12 words mnemonic already used: for BIP39 it is "only" needed to try 128 bits of entropy and then the others 4 bits, the checksum, are obtained through an HASH function, instead with Electrum it is needed to try all the 132 bits of entropy and then check, through an HASH function, if they are valid. With this example the difference is in 4 bits, but it increase with the number of words:

Words	BIP39	Electrum	Difference
12	128	132	4
15	160	165	5
18	192	198	6
21	224	231	7
24	256	264	8

The first column represents the number of words in a mnemonic phrase, the second and the third represent respectively the bits of entropy needed to be checked in order to find a specific BIP39 or Electrum mnemonic phrase. The last column is simply the difference between the previous two. Although this additional difficulty is not necessary, the difference between the two methods is not negligible. In fact, to find a specific Electrum phrase with 12 words you have to do 16 (2^4) times the attempts needed to find the same number of words in BIP39 framework. If the words become 24 the number of attempts would be 256 (2^8) times.

Another difference is in the way that a phrase is consider valid. BIP39 used a checksum based on the input entropy. This means that the knowledge of the mnemonic phrase alone is not enough to know if it is valid or not. A fixed dictionary is always required. On the other side for Electrum it is useless the knowledge of the dictionary, because the validation is based directly on the HASH of the mnemonic phrase. Furthermore Electrum allows the use of any kind of dictionary and not only the standard ones.

The most important difference is the Electrum introduction of version type for the seed. While BIP39 only check if a mnemonic is valid, Electrum check the validity of phrase looking if it correspond to a specific version. Directly from the Mnemonic, with Electrum, it is possible to understand how to derive all the keys required and their purpose. On the other side, with BIP39, it is impossible to say *a priori* the purpose of keys derived from that seed.

To avoid this problem a new BIP was proposed: BIP43. In order to identify a particular purpose for a bunch of keys, a particular derivation scheme was used:

$$m \ / \ purpose' \ / \ *$$

Changing the derivation path, will change also the purpose of the keys. For more information see the next chapter.

Chapter 5

How to use a HD Wallet

As already mentioned there are various way to use a Hierarchical Deterministic Wallet. It is possible to use this sequence of keys also for a non-monetary purpose, outside the cryptocurrency world. In fact these keys are simple number and they can be used for every possible purpose. Here we will focus only on the cryptocurrency application, in particular Bitcoin.

5.1 BIP 43

This BIP introduces a "Purpose Field" in the Hierarchical deterministic wallet. The first child of the extended master private key specify the purpose of the entire branch.

$$m \ / \ purpose' \ / \ *$$

For example if $purpose = 44$ it means that this is a multi coin wallet, if $purpose = 49$ it means that the keys generated follow the BIP49 specification (P2WPKH nested in P2SH).

5.1.1 Multi-coin wallet BIP 44

The derivation scheme for a Multi-coin wallet should be the follow:

$$m/purpose'/coin\_type'/account'/change/address\_index$$

Let's see the meaning of each field:

- *Purpose*: it must be equal to 44' (or 0x8000002C) and it indicates that the subtree of this node is used according to this specification. Hardened derivation is used at this level.
- *Coin_type*: This level creates a separate subtree for every cryptocoin, avoiding reusing addresses across cryptocoins and improving privacy issues. Coin type is a constant, set for each cryptocoin. Hardened derivation is used at this level.

  Some example:

Path	Cryptocoin
m / 44′ / 0′	Bitcoin (mainnet)
m / 44′ / 1′	Bitcoin (testnet)
m / 44′ / 2′	Litecoin
m / 44′ / 3′	Dogecoin
m / 44′ / 60′	Ethereum
m / 44′ / 128′	Monero
m / 44′ / 144′	Ripple
m / 44′ / 1815′	Cardano
⋮	⋮

A list with the complete set of cryptocoin is available and in continuous update:
https://github.com/satoshilabs/slips/blob/master/slip-0044.md

- *Account*: This level splits the key space into independent user identities, so the wallet never mixes the coins across different accounts. Users can use these accounts to organize the funds in the same fashion as bank accounts; for donation purposes, for saving purposes, for common expenses etc. Hardened derivation is used at this level.

- *Change*: It can take only two value: 0 and 1. 0 is used for addresses that are meant to be visible outside of the wallet (e.g. for receiving payments). 1 is used for addresses which are not meant to be visible outside of the wallet and is used for return transaction change. Public derivation is used at this level.

- *Index*: This is the last derivation, used to have many keys for each cryptocoin. It can take values from 0 in sequentially increasing manner. Public derivation is used at this level.

The principal advantage of this BIP is the possibility to easily back up all the cryptocoins of the user just by remembering a particular set of world (BIP39 mnemonic phrase). If these method is used to store more then one coins, keep attention in not to lose the master private key, otherwise all coins were lost.

5.1.2 SegWit addresses BIP 49

From August 2017 a new way to sign a Bitcoin transaction has been allowed, Segregated Witness: SegWit. It is not the purpose of this thesis to look inside of this topic, but it is important to say that a software not updated is not able to spend coins received in this format. So it is necessary to have a different branch in the derivation scheme just for SegWit addresses, in such a way that only software aware of SegWit are able to spend thus coins.

The derivation scheme is the follow:

$$m/49'/coin\_type'/account'/change/address\_index$$

The logic of BIP44 was followed and it let the user the possibility to have a multicoin wallet just by choosing a specific *coin_type*.

Remark *In order to make SegWit a soft fork (backward compatible), a SegWit transaction is nested in a pay-to-script-hash, so the corresponding address must begin with '3'.*

Appendix A

Bitcoin keys representation and addresses

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

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

A.1 Uncompressed public key

An uncompressed public key is rapresented 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

A.2 Compressed public key

A compressed public key is obtained simply taking the x coordinate and adding 02 at the begging 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

A.3 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.

- Addd 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

A.4 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 freamwork 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