Universidad de Valladolid

ESCUELA DE INGENIERIA INFORMATICA (SG)

Grade en Ingeniería Informática de Servicios y Aplicaciones

Bitcoin Wallet: simulación de una criptomoneda

Alumno: Guillermo Nicolas Leiro Arroyo

Tutor: José Ignacio Farrán Martín

ÍNDICE

1 Introducción 5
2 Estructuración de la memoria 6
3 Manual de usuario 7
3.1 Jupyter notebooks en el navegador 7
3.2 VSCode 8
3.3 Desinstalación 9
4 Algoritmos hash 10
4.1 Hashes en Bitcoin 10
4.1.1 Hash160 11
4.1.2 Hash256 12
4.2 RIPEMD-160 13
4.3 SHA-512 14
4.4 SHA-256 15
4.5 ¿Como funciona SHA-256? 15
4.5.1 Operaciones atómicas 16
4.5.1.1 SHR 16
4.5.1.2 ROTR 16
4.5.1.3 XOR 16
4.5.1.4 SUM 17
4.5.2 Operaciones compuestas 17
4.5.2.1 σ0 17
4.5.2.2 σ1 18
4.5.2.3 Σ0 19
4.5.2.4 Σ1 19
4.5.2.5 Choice 20
4.5.2.6 Majority 20
4.5.3 Constantes K 21
4.5.4 Codificación 21
4.5.5 Padding 22
4.5.6 Iteradores de bloques y mensajes 23

4.5.7 Message Schedule 24
4.5.8 Extended Message Schedule 24
4.5.9 Inicialización Registros de Compresión 25
4.5.10 Compresión 26
4.5.11 Retoques finales 28
5 Curvas Elípticas (Firmas Digitales) 30
5.1 ECDSA 32
5.1.1 Curvas elípticas 32
5.1.2 Operaciones en curvas elípticas 35
5.1.2.1 Inverso de un entero módulo p 35
5.1.2.2 Doblar 37
5.1.2.3 Suma 39
5.1.2.4 Multiplicación de un punto por un número 42
5.1.3 Generación de claves 44
5.1.4 Firma 45
5.1.4.1 ¿Qué pasa si usamos un Salt no seguro en nuestra firma? 47
5.1.5 Verificación 48
5.1.6 Codificaciones 49
5.1.6.1 DER 49
5.1.6.2 SEC sin compresión 50
5.1.6.3 SEC con compresión 50
6 Frase Mnemónica 52
6.1 Entropía 52
6.2 Código de comprobación de errores 52
6.3 Mapeo de palabras 53
6.4 Creación de la frase mnemónica 54
6.5 Validación de la frase mnemónica 55
6.6 Creación de la semilla 56
7 Claves extendidas 59
7.1 Derivación de la clave extendida maestra 59
7.2 Clave privada extendida hija ordinaria 61
7.3 Clave privada extendida hija curtida 63
7.4 Clave pública extendida hija ordinaria 66

7.5 Clave pública extendida hija curtida ... 68
8 Árbol de derivación ... 69
8.1 Test de derivación aislada ... 76
9 Codificación y serialización de claves ... 78
9.1 Base 58 ... 78
9.2 serialización y codificación claves extendidas ... 78
9.3 WIF (Wallet Import Format) ... 82
10 Nuestra cartera HD ... 85
11 Script ... 95
11.1 P2PK ... 101
11.1.1 ScriptPubKey ... 101
11.1.2 ScriptSig ... 101
11.2 P2PKH ... 102
11.2.1 ScriptPubKey ... 102
11.2.2 ScriptSig ... 102
11.3 P2MS ... 102
11.3.1 ScriptPubKey ... 103
11.3.2 ScriptSig ... 103
11.4 P2SH ... 103
11.4.1 ScriptPubKey ... 104
11.4.2 ScriptSig ... 104
11.4.3 NULL DATA ... 104
12 Transacción ... 105
12.1 TxIn ... 111
12.2 TxOut ... 113
13 Herramientas usadas ... 119
14 Conclusiones y futuras líneas de trabajo ... 120
15 Bibliografía ... 121
15.1 Libros ... 121
15.2 Recursos Web ... 121

1 INTRODUCCIÓN

La idea del tema seleccionado para este trabajo surgió durante el desarrollo del TFG anteriormente escogido, en el cual tenía como propósito la creación de una criptomoneda en C++. Sin embargo, mucho más simple y sencilla que cualquier criptomoneda desarrollada con el fin de ser lanzada al público y obtener un gran número de usuarios, me di cuenta de que el mundo de las criptomonedas es mucho más extenso y tiene mucha más profundidad de la que yo me podía haber imaginado. Por estas razones, decidí tomar la decisión de cambiar y acotar el tema en el cual me iba a centrar para mi trabajo de fin de grado.

En este trabajo me gustaría aprender a la vez que mostrar, los procedimientos y tecnologías usadas en las carteras Bitcoin deterministas. Desde algoritmos criptográficos como funciones hash o firmas digitales, hasta cómo construir una transacción.

La curiosidad por este tema en concreto surgió una vez comprendí los aspectos más básicos de las criptomonedas, pero a su vez tenía muchas dudas y fascinación por cómo una simple clave puede guardar toda una cartera de criptomonedas, o el por qué ciertas monedas o transacciones dentro de estas no son compatibles con algunas carteras. Como veremos a lo largo de esta memoria muchas de estas dudas y en cierta medida lo que quería replicar o, mejor dicho, ver cómo funciona una cartera criptográfica llamada (Electrum) de la cual hablaremos en esta memoria.

A la hora de adentrarme en el mundo de las criptomonedas y más concretamente en el mundo de Bitcoin, necesitamos de una manera de interactuar con estas redes, y la elegida por mí en aquel momento fue la cartera criptográfica (Electrum y Micelium) las cuales en aquel momento me presentaron una gran variedad de posibilidades y opciones las cuales no entendían ni que hacían ni el cómo funcionaban. Y por ello decidí el mostrar y replicar el funcionamiento de estas para poder entender que opción usar y como ha de ser usada.

Puesto que es un proyecto vinculado a una carrera de informática no podía faltar algo de código desde mi punto de vista, y por ello he decidido el llevar a cabo el desarrollo de los ejemplos en Júpiter Notebooks, puesto que son una gran plataforma para aprender e interactuar con el código. Las distintas celdas de código en estos notebooks están escritas en Python, ya que es un lenguaje de programación con sintaxis amigable, lo que hace seguir el código mucho más fácil y simple.

El objetivo de este trabajo es el aprender y mostrar las distintas tecnologías de las cuales una cartera determinista en Bitcoin hace uso, y ver como cada uno de estos componentes funciona al más bajo nivel en cuanto al código se refiere. Es en parte por estas razones las cuales me gustaría alcanzar con este trabajo por la que decidí hacer uso de las herramientas antes brevemente citadas.

2 ESTRUCTURACIÓN DE LA MEMORIA

La memoria sigue una estructura (Bottom up), la cual empieza describiendo algunas de las partes más fundamentales para nuestra cartera (o para las criptomonedas en general), y culmina poniendo las distintas piezas del puzle juntas para ver como una transacción es creada en una cartera.

La memoria está pensada para ser acompañada por los Jupyter notebooks con el código de cada una de las secciones. Puesto que la lectura de esta memoria en solitario quedaría muy sosa y transmitiría las mismas preguntas al lector de "Pero ¿cómo lo hace?" que yo he experimentado leyendo otras fuentes relacionadas con este mismo tema, incluiré el código y resultados generados por la ejecución de este para que pueda ser seguido sin ordenador.

El código adjuntado con esta memoria está distribuido en cuatro notebooks, los cuales podremos encontrar en la carpeta (*hd-wallet*):

- **SHA-256.ipynb**
  En este notebook encontraremos el código de manera interactiva y con alguna que otra animación de cómo funciona un hash SHA-256. He elegido este puesto que uno de los más usados en Bitcoin y será usado en esta memoria para explicar los tipos y utilidades de las funciones hash.

- **EllipticCurve.ipynb**
  Este notebook nos muestra algunos fundamentos básicos de las curvas elípticas y criptografía asimétrica, y concluye con la creación y validación de una firma digital.

- **HD_wallet.ipynb**
  Nuestro tercer notebook comprende muchos aspectos y tecnologías usados en las carteras criptográficas. Incluye muchas de las BIP (Bitcoin Improvement Proposals) las cuales introdujeron alguna mejora o novedad en el ámbito de las carteras criptográficas como por ejemplo semillas mnemónicas, claves extendidas o árboles de derivación.

- **Transactoins.ipynb**
  Para finalizar en este notebook veremos y construiremos una transacción para poder poner a prueba lo anteriormente visto.

Cada uno de los puntos en esta memoria estará ligado a un notebook del repositorio, el cual será indicado al comienzo de dicho apartado.

Los notebooks se encuentran en inglés, por lo que me disculpo si hay algún inconveniente por ello, pero la gran mayoría de información, herramientas y proyectos en el campo de la informática está ligado al inglés, por lo que me parecía lógico el crear los notebooks y hacer toda mi labor de investigación en esta lengua. Las pequeñas descripciones encontradas en algunas de las celdas de estos pueden ser encontradas en esta memoria de manera extendida y en castellano, aunque algunos de los términos hayan sido mancillados por el camino.

3 MANUAL DE USUARIO

Opino como la gran mayoría cuando digo que no me gusta el tener que instalar aplicaciones o dependencias en mi ordenador, de las cuales solo haré uso una vez, y luego quedarán allí de manera indefinida, bien porque el proceso de desinstalación sea tan enrevesado que no lo quiera revertir, o porque simplemente no haya un proceso definido para su desinstalación. Es por ello por lo que intentaré hacer una guía sencilla y rápida que muestre como crear un entorno para la ejecución de estos y el cómo deshacerse de este una vez hallamos concluido con su uso.

Para poder hacer uso de los Jupyter Notebooks necesitaremos tener el intérprete de Python instalado en nuestro ordenador versión 3 o superior.

https://www.python.org/

Además del intérprete necesitaremos un administrador de paquetes como PIP el cual nos ayudará a instalar las dependencias necesarias. Este suele venir incluido con la instalación del intérprete, pero en caso de no ser así podemos encontrar la guía de como instalarlo en el siguiente enlace:

https://pip.pypa.io/en/stable/installation/

Finalmente necesitaremos instalar el paquete Pipenv con la ayuda de PIP, el cual podremos instalar con la ayuda del siguiente comando si la utilidad PIP se encuentra en nuestro PATH:

```
$ pip install pipenv
```

https://pypi.org/project/pipenv/

Ahora que ya tenemos todas las herramientas necesarias podemos empezar creando nuestro entorno virtualizado, en el cual correremos posteriormente los notebooks. Esto solo requerirá de un comando:

```
$ pipenv install
```

El cual tendremos que ejcutar en el directorio en donde se encuentran nuestros Notebooks. En caso de no tener Pipenv en nuestro PATH, debemos de ejecutarle desde nuestro intérprete:

```
$ python -m pipenv install
```

Una vez tengamos nuestro entorno simplemente accederemos a este con:

```
$ pipenv shell
```

Llegados a este punto hay dos caminos que me gustaría ofrecer o mencionar. El primero es el de abrir nuestros Notebooks en el navegador con un servidor en local, siendo este el más simple, rápido y sencillo. El segundo sería el de abrir los Notebooks con VSCode el cual requiere tener VSCode instalado, así como la extensión de Python para este. Recomiendo la última opción ya que esta ha sido la manera empleada para su desarrollo, y durante este me encontré algunos problemas de compatibilidad con las celdas de Mark Down, el cual esta temporalmente resuelto con HTML.

3.1 JUPYTER NOTEBOOKS EN EL NAVEGADOR

Si elegimos esta opción simplemente tendremos que correr el siguiente comando una vez nos encontremos en el entorno virtual recién creado y esto nos abrirá una pestaña en el navegador.

```
$ jupyter notebook
```

3.2 VSCODE

Para esta otra opción tendremos que instalar VSCode y la extensión de Python que se muestra a continuación.

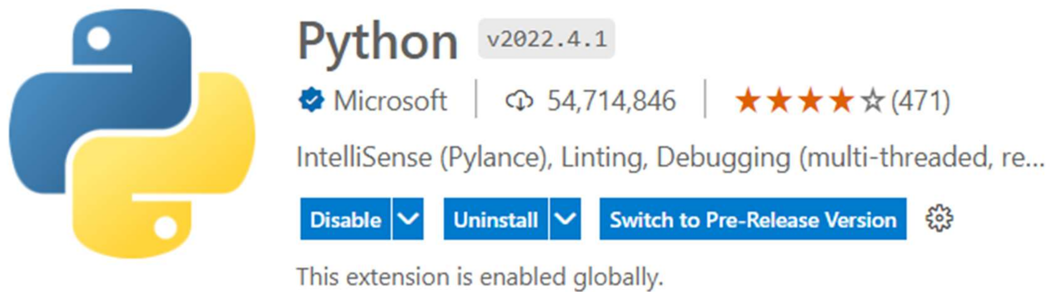


Ilustración 1

Desde la terminal correremos el siguiente comando:

```
$ code .
```

Y nos cercionaremos de que el kernel indicado en la imagen inferior es el entorno virtual anteriormente creado.

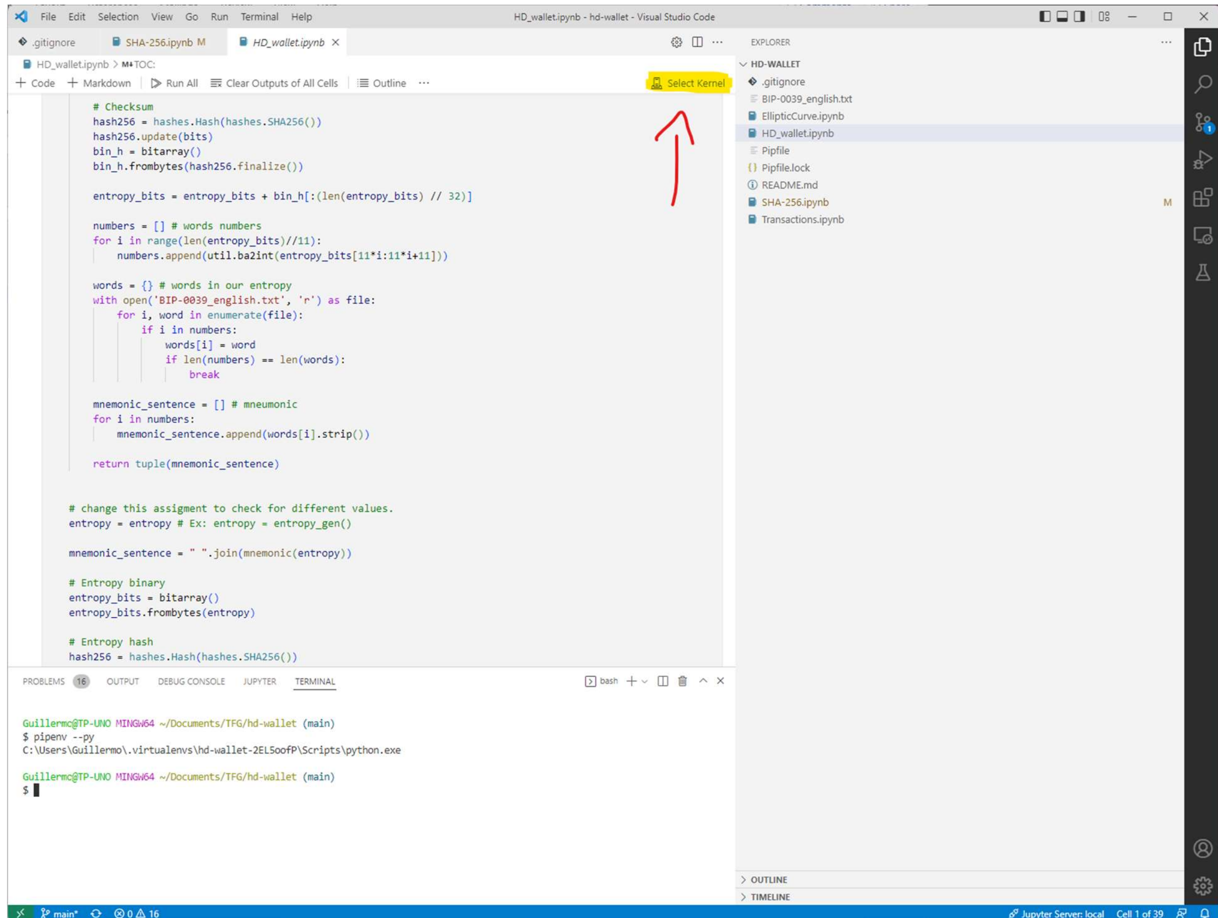


Ilustración 2

3.3 DESINSTALACIÓN

Para desinstalar o deshacernos de las dependencias instaladas anteriormente simplemente tendremos que desplazarnos hasta el directorio en el cual se encuentre nuestros notebooks, y correr el siguiente comando:

```
$ pipenv --rm
```

Esto eliminará el entorno virtual de nuestro sistema, así como todo sus contenidos y cambios creados en él. En caso de no saber si está más en el entorno virtual o no, o en cual estamos, usar los comandos:

Mostrar PATH a tu intérprete de Python:

```
$ pipenv --py
```

Mostrar entornos virtuales:

```
$ pipenv --venv
```

4 ALGORITMOS HASH

Una función hash es un pequeño programa de ordenador que baraja y entremezcla los datos suministrados a este, y devuelve un resultado de longitud fija. Una de las mejores descripciones que he oído de la definición de algoritmo hash es la de una compresión con perdida muy agresiva.

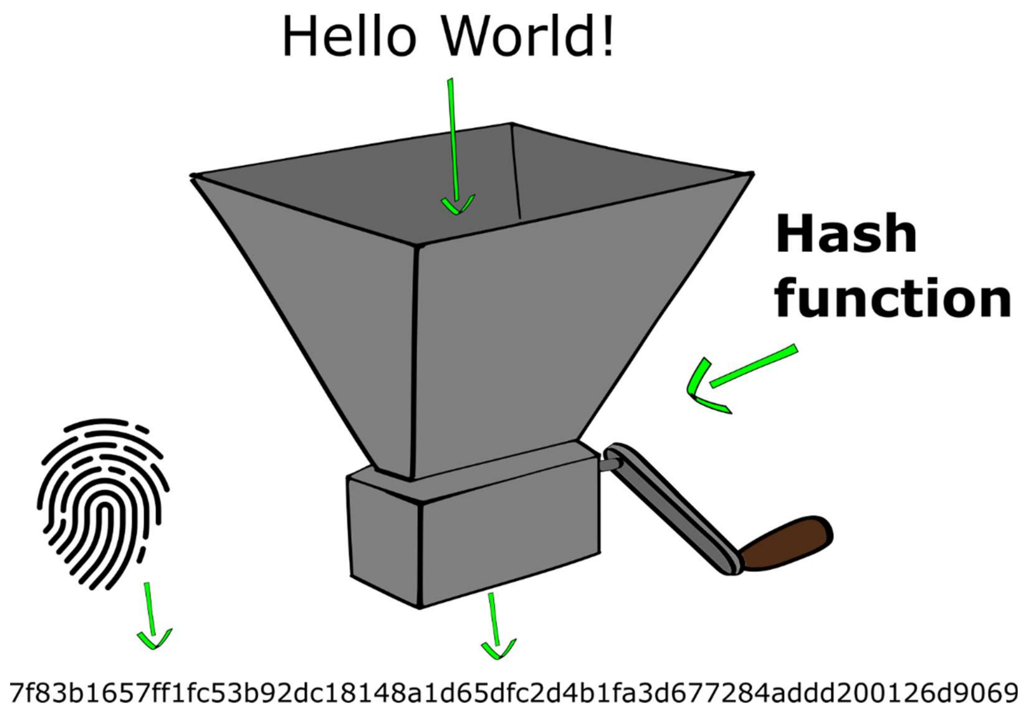


Ilustración 3

Como me mencionó mi tutor cabe recalcar que las funciones hash que veremos es este trabajo pertenecen al grupo de funciones hash criptográficamente seguras, las cuales son funciones hash en las que sus resultados tienen una probabilidad tan baja de colisionar que son usadas como si estos fueran únicos para cada distinta entrada.

4.1 HASHES EN BITCOIN

Bitcoin hace uso extensivo de los algoritmos hash criptográficamente seguros. Estos son empleados para todo tipo de tareas, desde comprobación de errores hasta compresión de datos.

Una particularidad de los hashes de Bitcoin es la de usar hashes dobles, esto ayuda con la seguridad ya que al hashear los mismos datos con dos algoritmos distintos hace que si en un futuro uno de estos algoritmos fuese roto y dejase de ser seguro el otro lo respaldaría.

4.1.1 Hash160

Primero se aplica el hash SHA-256 a los datos, y al resultado de este se le aplica el hash RIPEMD-160, para finalizar con un resultado de 20 bytes. Este doble hash incrementa la seguridad o por decirlo de otra forma añade una segunda capa de seguridad, ya que en caso de que se consiga revertir uno, será siendo imposible recuperar los datos de inicio.

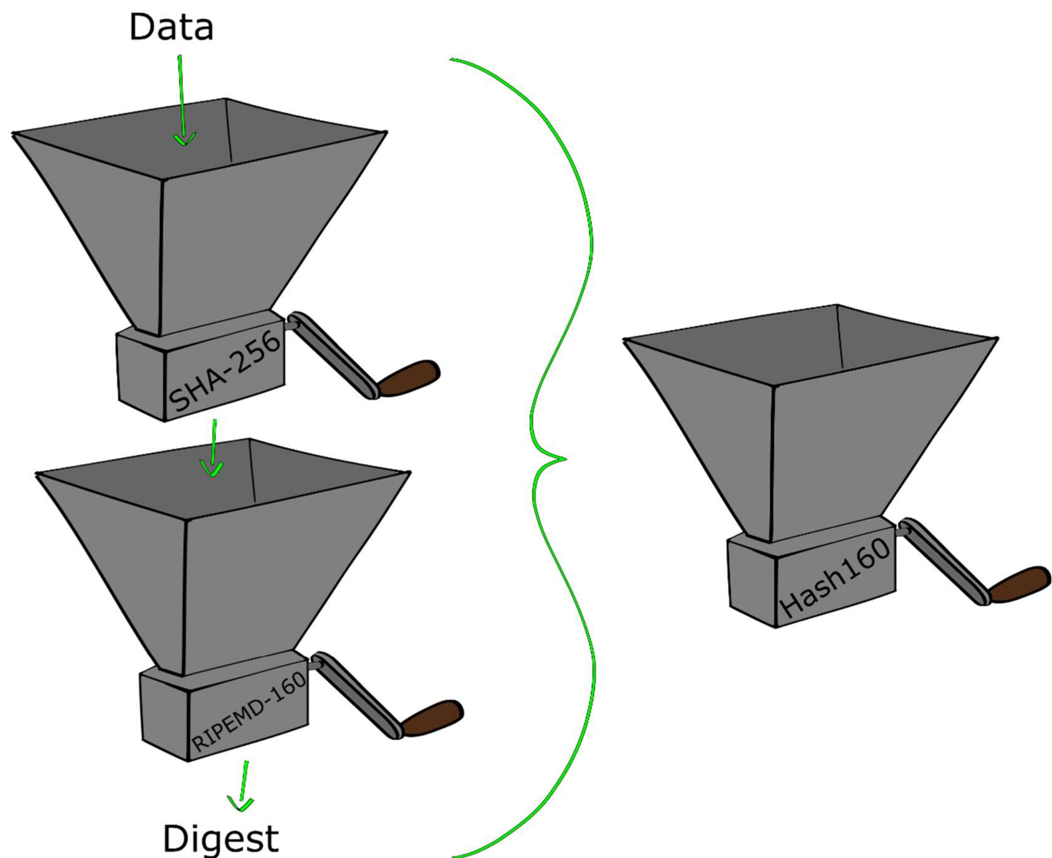


Ilustración 4

```
def hash160(data):
    return new('ripemd160', sha256(data).digest()).digest()
```

Transactions.ipynb

4.1.2 Hash256

Este algoritmo simplemente corre el hash SHA-256 dos veces sobre los datos. A diferencia del anterior, este no incrementa la seguridad puesto que el correr dos veces el mismo hash no nos da esa segunda capa de refuerzo.

Hay bastantes teorías de porque esto es así en lugar de simplemente correr una única iteración. Las dos que yo encuentro más acertadas son la de que el correr el mismo hash dos veces ralentiza un ataque de fuerza bruta, pero funciones hash como SHA-256 son algoritmos muy optimizados para correr de manera muy rápida en dispositivos muy poco potentes por lo que si se quisiera disuadir un ataque de fuerza bruta contra este el número de iteraciones debería ser mucho mayor a dos.

La otra teoría y por la que yo me decanto es la de que Satoshi Nakamoto (el creador de Bitcoin) decidió crearlo de esta manera por estandarización del uso de las funciones hash a lo largo de toda la implementación de Bitcoin.

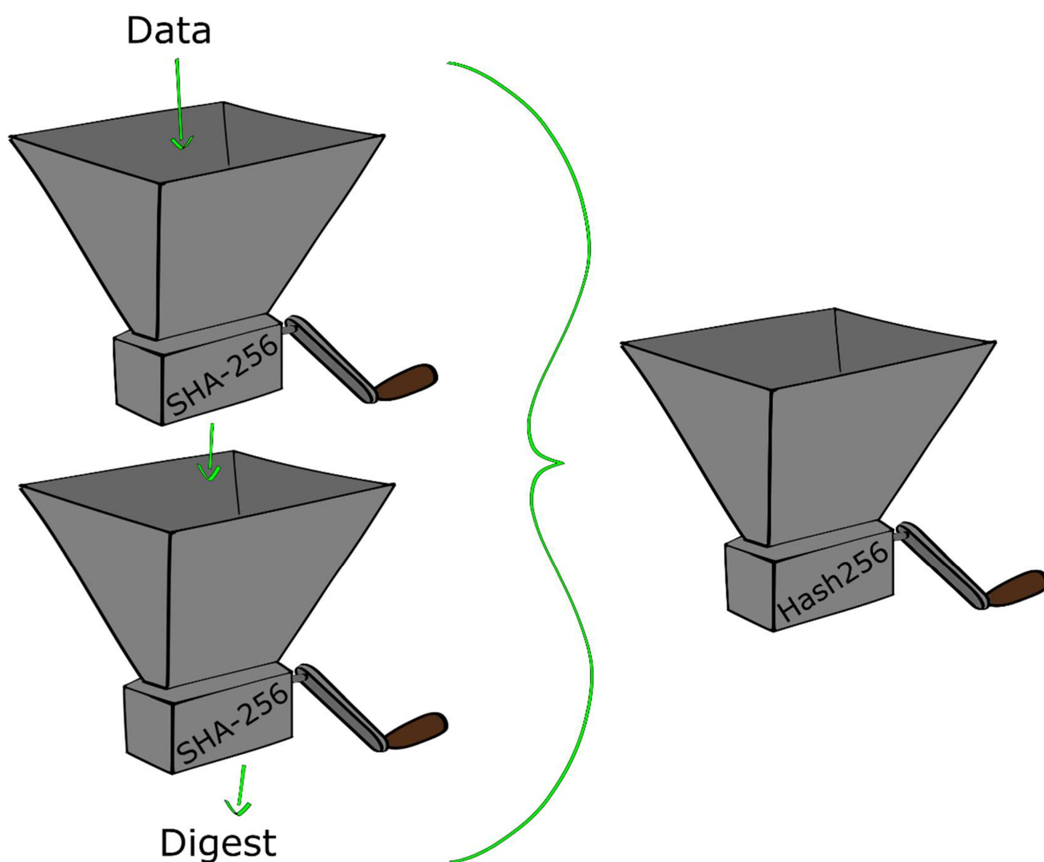


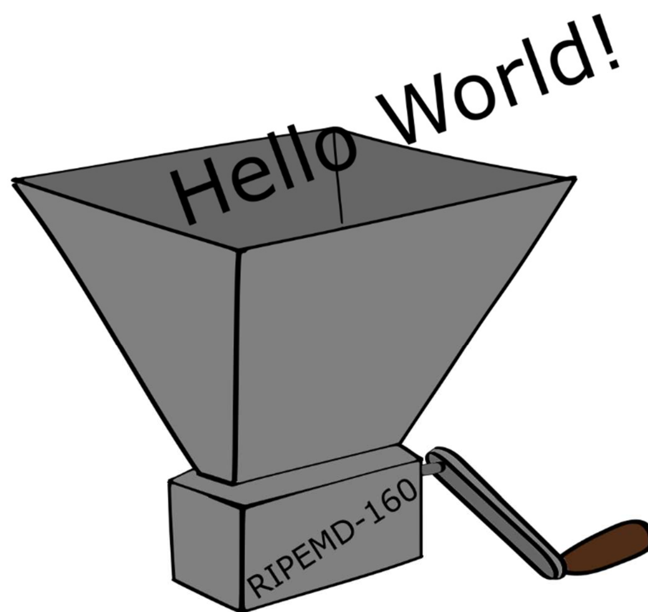
Ilustración 5

```
def hash256(data):
```

```
    return sha256(sha256(data).digest()).digest()
```
Transactions.ipynb

4.2 RIPEMD-160

Este hash tiene la peculiaridad de que el resultado de su digestión es de 160 bits o 20 bytes haciéndolo considerablemente más corto que el otro algoritmo fuertemente utilizado en Bitcoin, SHA-256. Además, se diferencia de este bastante ya que no pertenece a la familia SHA-2, la cual es una de las más usadas en el ámbito de hashes criptográficamente seguros en la que se incluye la popular variante SHA-256, esto como ya mencionamos anteriormente ayuda con la seguridad.



8476ee4631b9b30ac2754b0ee0c47e161d3f724c

Ilustración 6

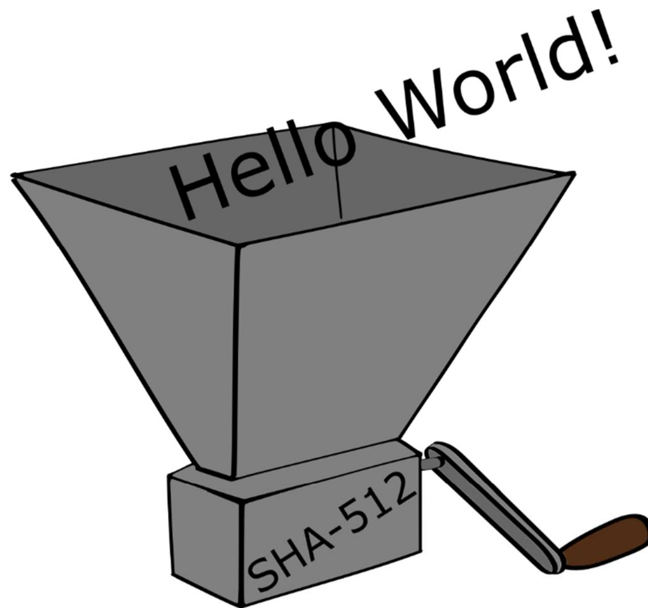
Como podemos ver su reducido tamaño hace que este sea más manejable por lo que es usado principalmente en la creación de direcciones de pago, las cuales en ciertas ocasiones han de ser escritas a mano por usuarios de Bitcoin.

```
Hash type: ripemd160
Void digest: 9c1185a5c5e9fc54612808977ee8f548b2258d31
Digest size: 20

String: Hello World!
Digest: 8476ee4631b9b30ac2754b0ee0c47e161d3f724c
```

4.3 SHA-512

Este hash con su longitud de 512 bits o 64 bytes el más seguro de los utilizados en Bitcoin, aunque en ciertos casos este extra de seguridad nos beneficie, la longitud de este hace que no sea considerado en multitud de áreas las cuales están pensadas como interfaces de usuario de Bitcoin (ej. Cripto monedero).



861844d6704e8573fec34d967e20bcfef3d424cf48be04e6dc08f2bd58c72974
3371015ead891cc3cf1c9d34b49264b510751b1ff9e537937bc46b5d6ff4ecc8

Ilustración 7

Aunque también perteneciente a la popular familia de hashes criptográficamente seguros SHA-2, este hash es menos popular que su hermano SHA-256 debido a que de momento su gran longitud no compensa el incremento en robustez en muchas aplicaciones.

Hash type: sha512

Void digest:
cf83e1357eefb8bdf1542850d66d8007d620e4050b5715dc83f4a921d36ce9ce47d0d13c5d85f2b0ff8318d2877eec2f63b931bd47417a81a538327af927da3e

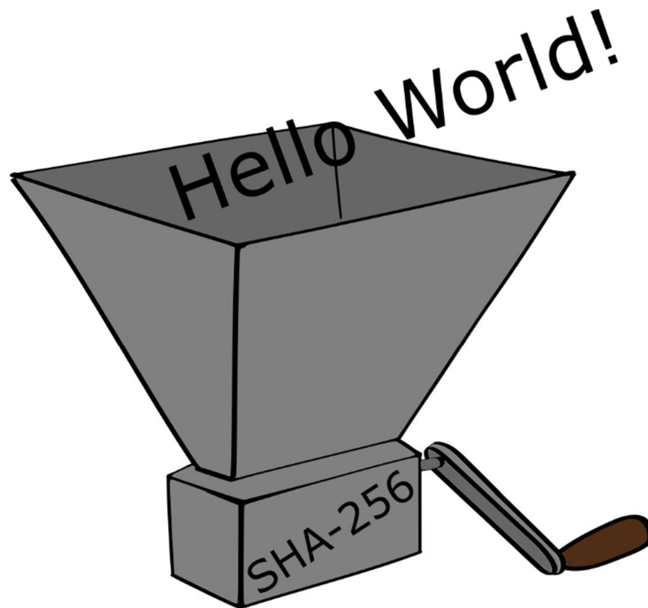
Digest size: 64

String: Hello World!

Digest:
861844d6704e8573fec34d967e20bcfef3d424cf48be04e6dc08f2bd58c729743371015ead891cc3cf1c9d34b49264b510751b1ff9e537937bc46b5d6ff4ecc8

4.4 SHA-256

El hash más popular sin lugar a duda no solo de la familia SHA-2, si no del gran catálogo de algoritmos de hashes criptográficamente seguros. Este, como podemos deducir de su nombre, presenta una longitud de 256 bits o 32 bytes, lo cual le da un equilibrio perfecto entre seguridad facilidad de manejo.



7f83b1657ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284addd200126d9069

Ilustración 8

Este es sin lugar a duda uno de los principales componentes de Bitcoin, por lo que en las siguientes páginas nos vamos a centrar en cómo funciona este algoritmo con algunos simples ejemplos de código.

```
Hash type: sha256
Void digest: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
Digest size: 32

String: Hello World!
Digest: 7f83b1657ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284addd200126d9069
```

4.5 ¿CÓMO FUNCIONA SHA-256?

Este apartado trata de explicar cómo funciona el algoritmo desde un punto de vista del software y no del porque ciertas operaciones son utilizadas en lugar de otras.

Dicho esto, hay algunos rasgos que podemos nombrar de un algoritmo como este:

- Ha de ser determinista dando siempre un mismo resultado, para unos mismos datos de entrada.

- La salida ha de ser siempre de 256 bits, 32 bytes.
- Un simple y pequeño cambio en los datos de entrada tiene que producir una salida completamente distinta.
- El número de colisiones de este ha de ser extremadamente pequeño para poder ser utilizado de manera segura.

4.5.1 Operaciones atómicas

SHA-256 trabaja realizando operaciones con los bits que conforman los datos de entrada, estas son las operaciones más simples realizadas en el algoritmo. El resto de las operaciones no son más que una combinación de estas. La mayoría de las operaciones tienen lugar con arrays de 32 bits de largo, que llamaremos longitud de la palabra.

Puesto que SHA-256 trabaja con los bits, la codificación de los datos es importante para el resultado. En este proyecto como datos usaremos strings los cuales serán siempre codificados usando ASCII o el superconjunto UTF-8 como extensión.

4.5.1.1 SHR

Desplazamiento a la derecha (Shift Right), esta operación corre todos los bits una posición a la derecha desechando los bits que sobrepasen el largo de la palabra, y añadiendo ceros por la izquierda para siempre mantener un largo de 32 bits.

```
def shr(bits, shifts):
    for _ in range(shifts):
        bits.pop()
        bits.insert(0, 0)
    return bits
```

SHA-256.ipynb

A: 11111111000000001111111100000000
SHR 32: 00000000000000000000000000000000

4.5.1.2 ROTR

Rotación a la derecha (Rotation Right) rota los bits de una palabra añadiendo el overflow de bits creado por la derecha al comienzo de esta.

```
def rotr(bits, shifts):
    for _ in range(shifts):
        bits.insert(0,bits.pop())
    return bits
```

SHA-256.ipynb

A: 11111111000000001111111100000000
ROTR 32: 11111111000000001111111100000000

4.5.1.3 XOR

En esta operación aplicamos la puerta lógica exclusive OR a cada par de bits de las palabras a y b siguiendo la siguiente tabla de verdad:

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

```
def xor(a, b):
  return a^b
```

SHA-256.ipynb

A: 11111111000000001111111100000000

B: 00000000111111111111111100000000 XOR

11111111111111110000000000000000

4.5.1.4 SUM

Suma binaria módulos 2^32, lo cual viene a decir que se ignora el overflow en caso de que se produzca.

```
def add(a, b):
    result = bitarray(endian='big')
    carry = 0
    for i in range(32-1, -1, -1):
        result.insert(0,(a[i]^b[i]^carry))
        carry = 1 if (a[i] + b[i] + carry) > 1 else 0
    return result
```

SHA-256.ipynb

A: 10000000001100001111111111111111

B: 10000000010100000000000000000000 +

00000000100000001111111111111111

4.5.2 Operaciones compuestas

Las operaciones compuestas no son más que operaciones de una complejidad mayor las cuales operan con tres palabras y a su vez hacen uso de las anteriores mencionadas operaciones atómicas.

4.5.2.1 σ0

Sigma minúscula cero tiene tres argumentos a, b y c a los cuales se les aplican las siguientes operaciones respectivamente:

- ROTR (7)
- ROTR (18)
- SHR (7)

Para finalizar estos tres valores son pasados por la operación atómica XOR y esto nos dará un resultado con la nueva palabra de 32 bits denominada σ0.

```
def sigma1(a, b, c):
    rotr(a, 17)
    rotr(b, 19)
    shr(c, 10)
    return xor(xor(a,b),c)
```

SHA-256.ipynb

```
   Origin: 00000000000000000011111111111111
   ----------------------------------------------------------------
   ROTR 7: 11111110000000000000000001111111
  ROTR 18: 00001111111111111100000000000000 XOR
    SHR 7: 00000000000000000000011111111111 XOR
   ----------------------------------------------------------------
       σ0: 11110001111111111100011110000000
```

4.5.2.2 σ1

Sigma minúscula uno viene a ser la misma operación anterior simplemente alterando el número de ROTR y SHR de la siguiente manera:

- ROTR(17)
- ROTR(19)
- SHR(10)

```
def sigma1(a, b, c):
    rotr(a, 17)
    rotr(b, 19)
    shr(c, 10)
    return xor(xor(a,b),c)
```

SHA-256.ipynb

```
   Origin: 00000000000000000011111111111111
   ----------------------------------------------------------------
  ROTR 17: 00011111111111111000000000000000
  ROTR 19: 00000111111111111110000000000000 XOR
   SHR 10: 00000000000000000000000000001111 XOR
   ----------------------------------------------------------------
       σ1: 00011000000000000110000000001111
```

4.5.2.3 Σ0

Sigma mayúscula cero es muy similar, lo único que cambian es que la operación SHR es remplazada por otra ROTR por lo que la combinación de operaciones es:

- ROTR(2)
- ROTR(13)
- ROTR(20)

```
def Sigma0(a, b, c):
    rotr(a, 2)
    rotr(b, 13)
    rotr(c, 22)
    return xor(xor(a,b),c)
```

SHA-256.ipynb

```
Origin: 00000000000000000011111111111111
-------------------------------------------------------------
ROTR 2: 11000000000000000000111111111111
ROTR 13: 11111111111110000000000000000001 XOR
ROTR 22: 00000000111111111111110000000000 XOR
-------------------------------------------------------------
Σ0: 00111111000001111111001111111110
```

4.5.2.4 Σ1

Al igual que σ0 y σ1, sigma mayúscula uno difiere de su hermana (Σ0) en el número de rotaciones aplicadas a la palabra.

```
def Sigma1(a, b, c):
    rotr(a, 6)
    rotr(b, 11)
    rotr(c, 25)
    return xor(xor(a,b),c)
```

SHA-256.ipynb

```
Origin: 00000000000000000011111111111111
-------------------------------------------------------------
ROTR 6: 11111100000000000000000011111111
ROTR 11: 11111111111000000000000000000111 XOR
ROTR 25: 00000000000111111111111110000000 XOR
-------------------------------------------------------------
Σ1: 00000011111111111111111101111000
```

4.5.2.5 *Choice*

Choice del inglés elección recibe tres palabras como argumentos y crea una cuarta la a partir de la siguiente lógica que es la que devuelve como resultado.

```
def choice(a, b, c):
    result = a.copy()
    for i in range(32):
        if a[i] == 1:
            result[i] = b[i]
        elif a[i] == 0:
            result[i] = c[i]
    return result
```

SHA-256.ipynb

La palabra A funcionara como discriminador, y dependiendo del valor que un determinado bit de esta obtenga seleccionaremos el bit en la misma posición de la palabra B (en caso del bit de A ser 1) o de la palabra C (en caso del bit de A ser 0).

```
A: 00000000111111110000000011111111
B: 00000000000000001111111111111111
C: 11111111111111110000000000000000
----------------------------------------------------------------
OUT: 11111111000000000000000011111111
```

4.5.2.6 *Majority*

Finalmente, la última de las operaciones compuestas es Majority, mayoría en castellano. Esta recibe tres palabras como argumentos y si la suma de A(n) + B(n) + C(n) es mayor a o igual a dos añade un 1 en OUT(n) o un cero en caso contrario, devolviendo el bit más abundante “mayoría” de cada columna.

```
def majority(a, b, c):
    result = bitarray("0"*32)
    for i in range(32):
        result[i] = 1 if a[i]+b[i]+c[i] > 1 else 0
    return result
```

SHA-256.ipynb

```
A: 00000000111111110000000011111111  0
B: 00000000000000001111111111111111  0
C: 11111111111111110000000000000000  1
----------------------------------------------------------------
OUT: 00000000111111110000000011111111
```

4.5.3 Constantes K

Las constantes K son simplemente números, los cuales nos ayudan a introducir ruido y tener más datos para mezclar.

Estos números son los primeros 32 bits o 4 bytes (una palabra) de la parte decimal de la raíz cúbica de los primeros 64 números primos positivos.

```
first64primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151,
157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241,
251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311]

# The list is indexed from 0 even tho is printed starting from 1.
K = []

def frac2bin(frac):
    x = floor(frac*(2**32))
    bin32 = ""

    for _ in range(32):
        bin32 += str(x % 2)
        x = x // 2

    return bin32[::-1]


for j, i in enumerate(first64primes):
    sqrt3 = i**(1./3.)
    deciaml_part = float(str(sqrt3)[1:])
    K.append(bitarray(frac2bin(deciaml_part)))
```

SHA-256.ipynb

Por lo que tengo entendido se hace uso de estos números tan rebuscados por razones de seguridad. Puesto que la raíz cúbica de un número primo produce un número irracional y la parte decimal de este no seguirá ningún patrón, lo que disminuye la posibilidad de creaciones de puertas traseras en el algoritmo.

$1 \rightarrow \sqrt[3]{2} \rightarrow 0.2599210498948732 \rightarrow 01000010100010100010111110011000$

$2 \rightarrow \sqrt[3]{3} \rightarrow 0.4422495703074083 \rightarrow 01110001001101110100010010010001$

$3 \rightarrow \sqrt[3]{5} \rightarrow 0.7099759466766968 \rightarrow 10110101110000001111101111001111$

...

$63 \rightarrow \sqrt[3]{307} \rightarrow 0.745996711705424 \rightarrow 10111110111110011010001111110111$

$64 \rightarrow \sqrt[3]{311} \rightarrow 0.775168952273312 \rightarrow 11000110011100010111100011110010$

4.5.4 Codificación

Una vez ya tenemos nuestras funciones atómicas y compuestas definidas, así como las constantes K, podemos iniciar el procedimiento de hashing por la codificación de los datos a hashear.

Usaremos como ejemplo el string “Hello World!”.

Hay que tener en cuenta la codificación de los datos, puesto que como ya mencionamos, estas funciones trabajan con los bits. En caso de usar una codificación distinta en futuros hashes obtendremos un resultado complemente diferente.

Input: "Hello World!"

Encoding: ASCII

Message:
01001000011001010110110001101100011011110010000001010111011011110111001001101100011
0010000100001

En este ejemplo haremos uso de codificación ASCII ya que solo nos interesa el poder codificar los caracteres incluidos en este standard. La codificación que se suele usar para texto es UTF-8, pero al ser esta un superset de ASCII los hace compatibles, siempre que no se usen caracteres de 8 bits (ascii > 127).

4.5.5 Padding

SHA-256 opera en bloques. Esto quiere decir que divide el mensaje en trozos más pequeños de igual tamaño (bloques) y opera con estos de uno en uno. En el caso de SHA-256 cada uno de estos bloques tiene una longitud de 512 bits o 64 bytes.

Como podemos ver nuestro mensaje no alcanza a rellenar un bloque entero por sí mismo, y puesto que nuestra función hash solo opera con múltiplos de bloques, tendremos que extender este hasta los 512 bits.

```
def padding(message):

    p_message = message.to01()
    blocks = len(message) // 512 +1 # Number of block the message is gonna
take

    if (len(message) % 512 >= (512 - 65)):
        blocks += 1

    p_message += '1' # Padding flag

    # Same result as the commented loop
    p_message += '0'*(blocks*512 - 64 - len(p_message))
    #for _ in range(len(p_message), blocks*512 - 64):
    #     p_message += '0'

    # Last 64 bits with message length
    bin_len_message = bin(len(message))[2:]
    len_bin_len_message = len(bin_len_message)
    p_message += '0'*(64-len_bin_len_message) + bin_len_message

    return bitarray(p_message)
```
SHA-256.ipynb

Padding message size: 512 bits

Message: Hello World!

01001000011001010110110001101100011011110010000001010111011011110111001001101100011
00100001000011000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000001100000

Message

Padding flag

Padding

Reserved bits for message length

Message length

Para todo mensaje que no sea justo un múltiplo de 512 bits (64 bytes), le añadiremos los siguientes bits hasta completar el próximo bloque:

- Un bit a 1 que usaremos como señalización de padding.
- Bits a 0 hasta completar el bloque dejando únicamente libre los últimos 64 bits.
- Y en estos últimos 64 bits incluiremos la longitud del mensaje en big-endian.

En caso de que la longitud de nuestro mensaje sea múltiplo de los 512bits de un bloque, este se expandirá al siguiente puesto que los bits reservados para la longitud del mensaje son necesarios.

4.5.6 Iteradores de bloques y mensajes

Como ya sabemos, se podría decir que hay dos unidades de las cuales SHA-256 hace uso:

- Palabras
- Bloques

Y debido a que vamos a hacer uso de estas unidades de manera secuencial, nos vendría muy bien el tener un par de iteradores como los siguientes.

```
def blck_iter(message):
    N_blck = len(message) // 512
    for blck in range(N_blck):
        yield message[blck*512:(blck+1)*512]

def word_iter(block):
    N_words = len(block) // 32
    for word in range(N_words):
        yield block[word*32:(word+1)*32]
```

SHA-256.ipynb

Como veremos a continuación la función que constituye el corazón del algoritmo hash coge un bloque, lo divide en palabras entre las cuales realiza diversas operaciones, y el resultado obtenido con este bloque es añadido al siguiente sobre el cual realizaremos las mismas operaciones.

4.5.7 Message Schedule

Message Schedule es el nombre que recibe el área de trabajo (registros en memoria) que utilizaremos para correr nuestras operaciones sobre cada uno de los bloques del mensaje. Este está formado por 16 palabras (32 bits) lo que nos da un total de 512 bits o un bloque.

```
def init_message_schedule(message):

    words = []

    for i in blck_iter(message):
        words = []
        for j in word_iter(i):
            words.append(j)
        yield words

word_list = next(init_message_schedule(p_message))
```

SHA-256.ipynb

Como podemos ver a continuación estos registros son inicializados con los bits del primer bloque de nuestro mensaje.

W1: 01001000011001010110110001101100
W2: 01101111001000000101011101101111
W3: 01110010011011000110010000100001
W4: 10000000000000000000000000000000
W5: 00000000000000000000000000000000
W6: 00000000000000000000000000000000
W7: 00000000000000000000000000000000
W8: 00000000000000000000000000000000
W9: 00000000000000000000000000000000
W10: 00000000000000000000000000000000
W11: 00000000000000000000000000000000
W12: 00000000000000000000000000000000
W13: 00000000000000000000000000000000
W14: 00000000000000000000000000000000
W15: 00000000000000000000000000000000
W16: 00000000000000000000000001100000

4.5.8 Extended Message Schedule

De igual manera que Message Schedule, podemos encontrar el Extended Message Schedule el cual está constituido por 64 registros de los cuales ya tenemos los 16 primeros poblados, y tendremos que extender el mensaje de 16 a 64 palabras para el bucle posterior.

```
def extend_message_schedule(words):
    for i in range(64-16):
```

```
        N = len(words)
        a = words[N-16].copy()
        b = sigma0(*[words[N-15].copy() for _ in range(3)]) # Weird unpaking
to match args
        c = words[N-7].copy()
        d = sigma1(*[words[N-2].copy() for _ in range(3)]) # Weird unpaking
to match args
        words.append(add(add(a, b), add(c, d)))
    return words
```

SHA-256.ipynb

Para computar los 48 registros restantes, usaremos la siguiente formula y añadiremos la palabra resultante con la siguiente formula y añadimos la palabra resultante al final de nuestra lista.

$$W_x = \sigma 1(W(x-2)) + W(x-7) + \sigma 0(W(x-15)) + W(x-16)$$

Donde x indica el valor del índice del registro a poblar.

W49: 00101000010011000001010111100011 —-—-→ 00101000010011000001010111100011
W50: 01100000100110100111100100010101 — σ0 → 10111000100101110010001111110110
W51: 00101110010011110010110010011011
W52: 00111100111101010110101001101111
W53: 00101111111000100011111111011010
W54: 10111010110110111100001001000000
W55: 11111100111000011110000010011110
W56: 00100101011000010000011010011001
W57: 10001010111000101101111000001001 —-—-→ 10001010111000101101111000001001
W58: 00001101100101110101010101110011
W59: 11010000101011001011001011101101
W60: 01111001010011101000000001100111
W61: 01111110101111010001101011100100
W62: 10100000011110010111000001000111 — σ1 → 10010110000000110011101001101111
W63: 10111111101001110000111111010111
W64: 01000110110000011000010010000010 ←-—-— σ1(W62) + W57 + σ0(W50) + W49

4.5.9 Inicialización Registros de Compresión

Los pasos anteriores nos han ayudado a generar un gran número de datos con un alto nivel de entropía, pero como ya sabemos nuestra función ha de devolver una huella digital de estos de exactamente 256 bits. Para esto los siguientes pasos serán los de comprimir nuestras 64 palabras a tan solo 8 palabras o registros.

El primer paso al igual que con el Message Scheduler es el de inicializar nuestros registros, para lo que volveremos a hacer uso de los números primos.

```
def init_stage_registers():
    regs = [2, 3, 5, 7, 11, 13, 17, 19]
    for i in range(len(regs)):
        aux = bin(int((sqrt(regs[i])%1)*2**32))[2:]
        regs[i] = bitarray("0"*(32-len(aux))+aux)
    return regs

stage_reg = init_stage_registers()
```

SHA-256.ipynb

Los registros que a partir de ahora denotaremos con las letras de (a-h), serán inicializados con los 32 primeros bits de la parte fraccional de las raíces cuadradas de nuestros 8 primeros números primos.

2 → √2 → 1.4142135623730951 → 0.41421356237309515 → 01101010000010011110011001100111
3 → √3 → 1.7320508075688772 → 0.7320508075688772 → 10111011011001111010111010000101
5 → √5 → 2.23606797749979 → 0.2360679774997898 → 00111100011011101111001101110010
7 → √7 → 2.6457513110645907 → 0.6457513110645907 → 10100101010011111111010100111010
11 → √11 → 3.3166247903554 → 0.3166247903553998 → 01010001000011100101001001111111
13 → √13 → 3.605551275463989 → 0.6055512754639891 → 10011011000001010110100010001100
17 → √17 → 4.123105625617661 → 0.12310562561766059 → 00011111100000111101100110101011
19 → √19 → 4.358898943540674 → 0.358898943540674 → 01011011111000001100110100011001

4.5.10 Compresión

Para este paso haremos uso de nuestras constantes K, registros de compresión y Extended Message Schedulers creados anteriormente. Al finalizar este paso obtendremos los valores de nuestros 8 registros de compresión, y estas palabras son las que formaran el valor hash.

```
def block_compress(stage_reg, word_list, K, animation=False):

    working_stage_reg = stage_reg

    temp1 = bitarray(endian='big')
    temp2 = bitarray(endian='big')

    for i in range(64):
        temp1 = add(
                    add(
                        add( Sigma1(*[working_stage_reg[4].copy() for _ in 
range(3)]),
                            choice(working_stage_reg[4].copy(),
                            working_stage_reg[5].copy(),
                            working_stage_reg[6].copy())),
                        add( working_stage_reg[7].copy(),
                            K[i])),
                    word_list[i])

        temp2 = add(Sigma0(*[working_stage_reg[0].copy() for _ in range(3)]),
                    majority(working_stage_reg[0].copy(),
                    working_stage_reg[1].copy(),
                    working_stage_reg[2].copy()))

        top = working_stage_reg[0].copy()
```

```
        working_stage_reg.insert(0, add(temp1, temp2))
        trash = working_stage_reg.pop()
        working_stage_reg[4] = add(working_stage_reg[4], temp1)

        if animation:
            for scene in range(3):
                print("W{}: {}".format(i+1, word_list[i].to01()))
                print("K{}: {}\n".format(i+1, K[i].to01()))

                print("T1: {:>32} = {}".format(temp1.to01(), "\u03A31(e) +
Choice(e, f, g) + h + K"+str(i+1)+" + W"+str(i+1)))
                print("T2: {:>32} = {}\n".format(temp2.to01(), "\u03A30(a) +
Majority(a, b, c)"))

                if(scene == 0):
                    right_side = "\u2193"
                    print("{:>4} = {:>32} {}".format(chr(97), top.to01(),
right_side))
                    for j, reg in enumerate(working_stage_reg[:-1]):
                        print("{:>4} = {:>32} {}".format(chr(j+97+1),
reg.to01(), right_side))
                    print("")
                elif(scene == 1):
                    for j, reg in enumerate(working_stage_reg):
                        if( j == 0):
                            reg = "..."
                            right_side = ""
                            print("{:>4} = {:<32} {}".format(chr(j+97), reg,
right_side))
                        else:
                            right_side = ""
                            print("{:>4} = {:>32} {}".format(chr(j+97),
reg.to01(), right_side))
                    print("{:>4} {:>32} {}".format("\U0001F6AE",
trash.to01(), right_side))
                elif(scene == 2):
                    for j, reg in enumerate(working_stage_reg):
                        if( j == 0):
                            right_side = "\u2190"+"\u2014 T2 + T1"
                        elif( j == 4):
                            right_side = "\u2190"+"\u2014 e + T1"
                        else:
                            right_side = ""
                        print("{:>4} = {:>32} {}".format(chr(j+97),
reg.to01(), right_side))
                    print("")

                if i < 12:
                    sleep(0.5)
                else:
                    sleep(0.05)
                clear_output(wait=True)

    return working_stage_reg

stage_reg = block_compress(stage_reg.copy(), word_list, K, True)
```
SHA-256.ipynb

Además de nuestros registros de compresión debemos de a ver uso de otros dos registros auxiliares sobre los cuales podamos operar y almacenar los valores de las siguientes fórmulas:

$$\mathbf{T1 = \Sigma1(e) + Choice(e, f, g) + h + Ki + Wi}$$

$$\mathbf{T2 = \Sigma0(a) + Majority(a, b, c)}$$

El algoritmo de compresión itera 64 veces por bloque, y en cada una de estas iteraciones hace uso de palabra enésima del Extended Message Scheduler denotada por (Wi) y de las constantes K denotadas por (Ki).

W64: 01000110110000001100001001000010
K64: 11000110011100010111100011110010

T1: 00111010011110110111000110111100 = Σ1(e) + Choice(e, f, g) + h + K64 + W64
T2: 11011010111111100101100101000010 = Σ0(a) + Majority(a, b, c)

a = 00010101011110011100101011111110 ←— T2 + T1
b = 11000100100010100100110111001110
c = 01111100101111101100111000001111
d = 10100011010100011110000100100011
e = 10101011000111101111100010100000 ←— e + T1
f = 00001000110100010000111010011100
g = 00101011010110011111100001010101
h = 10110110100011001100001101010000

En el ejemplo superior podemos observar los valores tomados por los registros en la última iteración, donde la palabra proveniente del Extended Message Scheduler es (W64) y la constante K es (K64).

Computamos nuestras dos fórmulas anteriormente mencionadas y almacenamos sus resultados en las variables T1 y T2.

El último paso es el correr los registros de compresión una posición hacia abajo, dejando (a) libre, la cual poblaremos con el valor (T2 + T1). Desecharemos la palabra en el registro (h), y añadiremos el valor de T1 al registro (e).

En caso de que nuestro mensaje contenga más de un bloque repetiremos este mismo proceso por cada bloque con la única diferencia de que para los bloques sucesivos los registros de compresión serán inicializados a los valores resultantes de la anterior iteración.

4.5.11 Retoques finales

Para concluir simplemente nos queda por añadir a los valores resultantes de nuestros registros de compresión los valores con los que en un principio estos fueros inicializados (32 primeros bits de la parte fraccional de las raíces cuadradas de nuestros 8 primeros números primos). Después codificaremos las

palabras resultantes de esta operación en hexadecimal y concatenaremos estas en orden alfabético en lo relativo al nombre del registro para obtener nuestro resultado hash final.

```
def sum_up(stage_reg):
    result = []
    hexa = []

    for init_reg, temp_reg in zip(init_stage_registers(), stage_reg):
        result.append(add(init_reg, temp_reg))
        hexa.append(util.ba2hex(result[-1]))

    return "".join(hexa)
```

SHA-256.ipynb

a = 7f83b165 ← 01101010000001001111001100110011 1 + 00010101011110011100101011111110

b = 7ff1fc53 ← 10111011011001111010111010000101 + 11000100100010100100110111001110

c = b92dc181 ← 00111100011011101111001101110010 + 01111100101111101100111000001111

d = 48a1d65d ← 10100101010011111111010100111010 + 10100011010100011110000100100011

e = fc2d4b1f ← 01010001000011100101001001111111 + 10101011000111101111100010100000

f = a3d67728 ← 10011011000001010110100010001100 + 00001000110100010000111010011100

g = 4addd200 ← 00011111100000111101100110101011 + 00101011010110011111100001010101

h = 126d9069 ← 01011011111000001100110100011001 + 10110110100011001100001101010000

Digest: 7f83b1657ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284addd200126d9069

5 CURVAS ELÍPTICAS (FIRMAS DIGITALES)

Este está enfocado principalmente a ECDSA (Elliptic Curve Digital Signature Algorithm), firmas digitales con curvas elípticas. Una firma digital es una manera de validar la proveniencia de los datos. Está compuesta por dos bloques principales, uno que ya hemos visto es un algoritmo hash criptográficamente seguro como podría ser SHA-256, y el segundo y más importante es la encriptación asimétrica.

La encriptación asimétrica es un método de encriptación en el cual los datos son encriptados por una clave y desencriptados por otra, a diferencia de la encriptación clásica o simétrica en la cual tendremos una única clave o palabra secreta, la cual desempeñará ambas funciones.

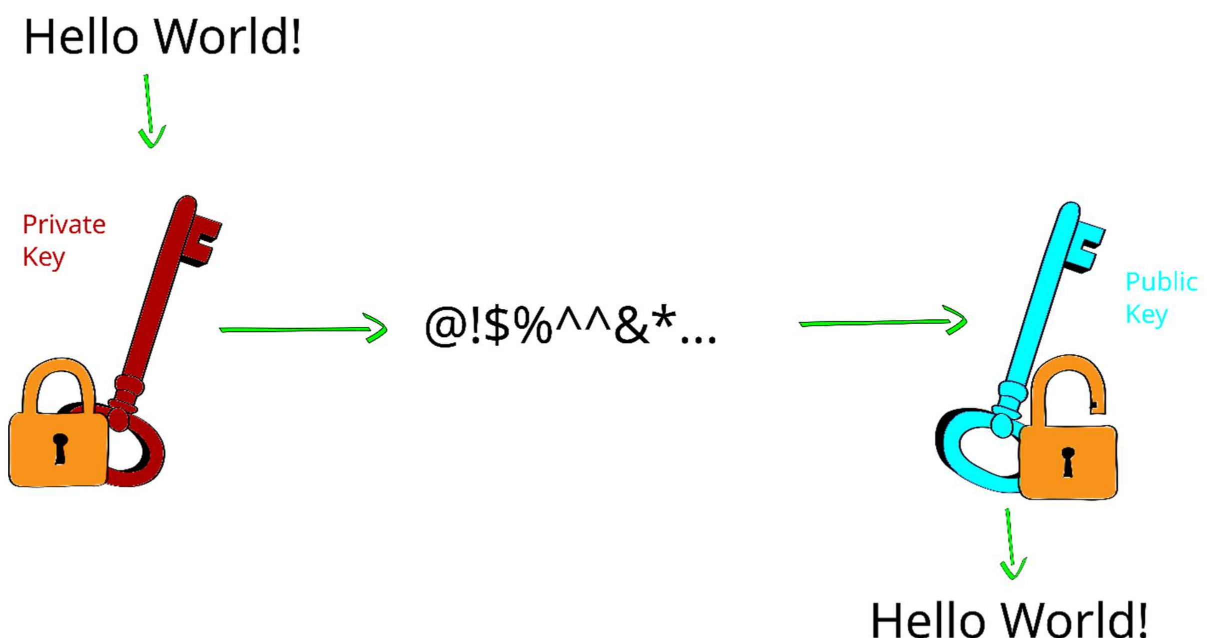


Ilustración 9

Como se muestra en la ilustración superior los datos de nuestro mensaje, los cuales en este caso están constituidos por el string “Hello World!”, son encriptados por nuestra clave privada. Los datos encriptados serán únicamente desencriptados por la clave pública la cual es derivada por medio de unas funciones matemáticas de la clave privada. Estas funciones matemáticas hacen que sabiendo el tiempo de algoritmo de encriptación asimétrica y la clave privada, podamos hallar la clave pública, pero el saber la clave pública no nos permite el hallar la clave privada. Son estas cualidades las que nos permiten difundir la clave pública junto con nuestro mensaje (de ahí su nombre de clave pública).

El receptor de nuestros datos desencriptaría el mensaje con la clave pública, y puesto que la clave pública solo desencripta datos previamente encriptados por la clave privada, la cual mantenemos de manera “privada” bajo nuestra custodia, el receptor del mensaje puede verificar que este mensaje ha sido enviado por nosotros.

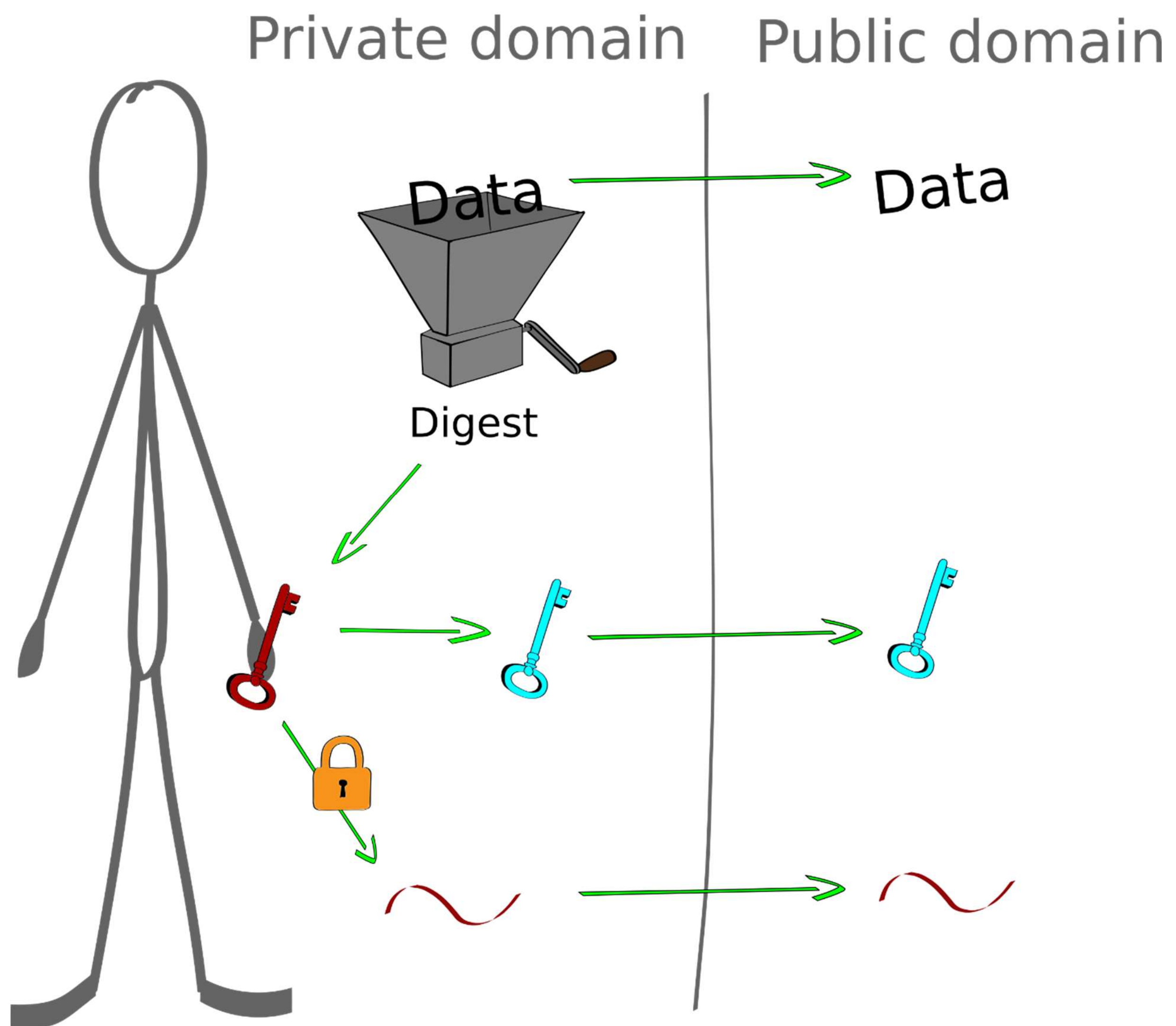


Ilustración 10

Aunque esto nos serviría, como método para validar la procedencia de unos datos, la firma digital va más allá. El propósito de esta no es el encriptar los datos, si no el validarlos. Es por esto por lo que se publican o transmite los siguientes tres elementos:

- Los datos
- La clave pública
- Datos encriptados asimétricamente

Ahora la manera que el receptor tiene de validar la integridad y procedencia de los datos es desencriptando los datos con la llave pública proporcionada y comparando estos con los datos transferidos sin encriptación alguna.

Este esquema, aunque nos permita el enviar los datos sin encriptación, de manera visible para todo el mundo y luego validarlos en el otro extremo, presenta un problema del que ya nos hemos dado cuenta

todos, y es que estamos enviando de manera duplicada los datos; encriptados y sin encriptar. Además de la duplicidad tenemos el problema de que si la cantidad de datos es muy grande (como por ejemplo un archivo de video), nos va a llevar mucho tiempo el encriptar estos. Es aquí donde entra en juego el hash criptográficamente seguro el cual vimos anteriormente. En lugar de encriptar los datos, encriptaremos el hash de estos y transmitiremos lo siguiente:

- Los datos
- La llave pública
- Firma digital

Ahora los datos encriptados, en este caso llamados firma digital, no son más que el hash cifrado de nuestros datos, y el procedimiento para su verificación será el siguiente:

1- Desencriptar la firma digital con la clave pública lo cual nos dará un hash.
2- Correr la misma función hash que se usó en la firma sobre los datos.
3- Comparar ambos resultados hash.

Es muy importante que para este proceso se use una función hash criptográficamente segura, ya que esto nos garantiza una probabilidad de colisiones muy reducida. Así, si alguien quisiera encontrar una colisión alterando los datos, tendría que invertir tanto tiempo y recursos que se consideraría prácticamente imposible.

5.1 ECDSA

El tipo de firma digital empleado en Bitcoin es el llamado ECDSA (Elliptic Curve Digital Signature Algorithm). Este puede ser dividido en dos:

1. DSA: (Digital Signature Algorithm) o algoritmo de firma digital el cual ya hemos visto anteriormente.
2. EC: (Elliptic Curve) o curvas elípticas. Esto nos indica el tipo de encriptación asimétrica de la cual haremos uso en este algoritmo.

Como podremos deducir hay distintos tipos de firmas digitales las cuales podemos identificar o distinguir por los métodos empleados para la encriptación asimétrica, así como los algoritmos hash usados. En este caso en particular usaremos ECDSA puesto que es el algoritmo usado por Bitcoin para firmar las transacciones.

5.1.1 Curvas elípticas

Las claves públicas y las firmas no son más que puntos en una curva elíptica. Estos son derivados de una clave privada, la cual está constituida por un número (un gran número), que deberemos generar de forma aleatoria y mantener guardado de forma muy segura ya que si este se diera a conocer estaríamos dando la llave de lo que estemos encriptando o firmando con esta tecnología; Bitcoin es este caso. Estas claves mantienen una relación geométrica que prueba que la persona que creó la firma fue la misma que creó la clave pública. Y es por medio de esta relación por la que podemos verificar que la información firmada por una persona o entidad realmente fue firmada por esta.

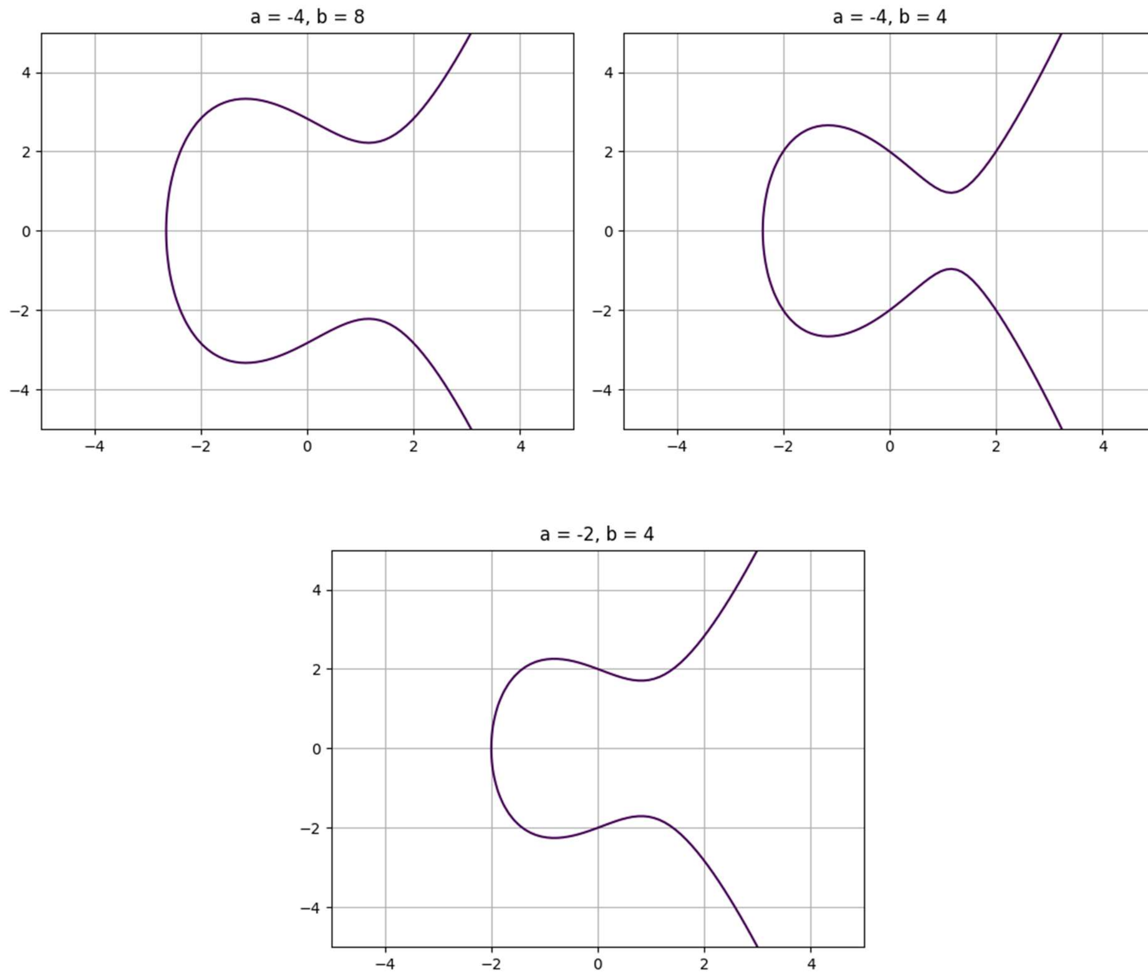


Ilustración 11

Una curva elíptica es una curva plana sin puntos singulares definida sin puntos singulares definida por un polinomio en X, Y de grado 3, su propiedad fundamental es que sus puntos se pueden “sumar”, y da un “grupo conmutativo”.

Hay un gran número de curvas elípticas, todas ellas siguen la fórmula indicada abajo donde (a) y (b) son las dos constantes que nos darán una de tantas variaciones. Estas curvas están estandarizadas con la intención de crear un algoritmo sólido, robusto y que al mismo tiempo pueda ser replicado por distintos usuarios obteniendo siempre un mismo resultado. Bitcoin en particular hace uso de la una curva llamada “secp256k1”, en la cual (a = 0) y (b = 7).

$$\boldsymbol{y^3 = x^2 + ax + b}$$

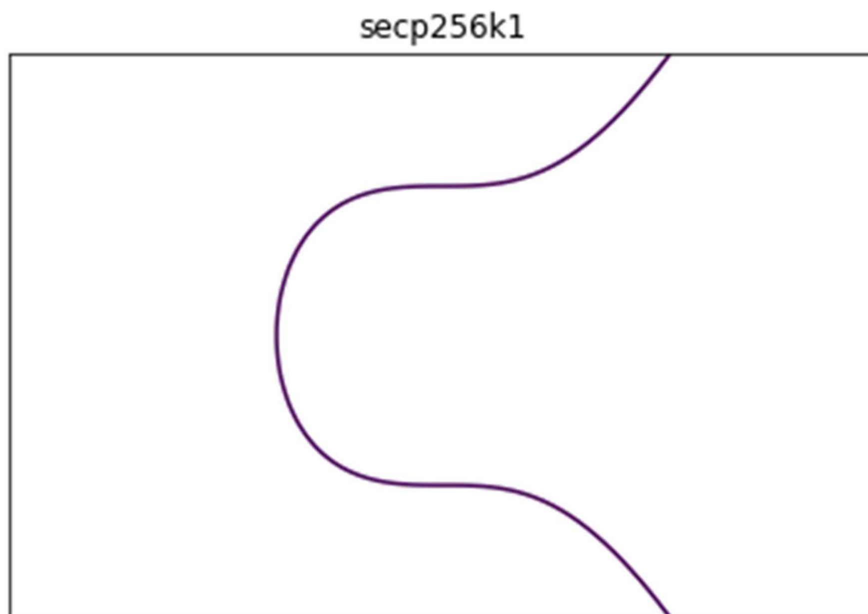


Ilustración 12

El nombre ecp256k1 viene a indicar lo siguiente:

- sec = Standard for Efficient Cryptography – un standard.
- p = Prime – Número primo usado para la creación del campo finito.
- 256 = 256 bits – Tamaño del campo finito.
- k = Koblits – Tipo de curva.
- 1 = Primera curva de este grupo.

Esta curva fue la elegida por Satoshi Nakamoto (creador de Bitcoin) para ser usada en el algoritmo de firma digital ECDSA. Y contiene los siguientes parámetros:

- (a & b) Constantes: que como mencionamos anteriormente son los parámetros que definen la forma de la curva en este caso toman los valores (0 & 7).
- (p) Cuerpo primo finito: hace que todos los puntos de la curva se encuentren dentro de un campo o set finito de números. Aunque representemos las curvas como funciones lineales en cuales (x e y) puede tomar cualquier valor, esto no es así, y los puntos de los cuales podemos hacer uso en la curva están restringidos a este espacio anteriormente nombrado. El cual pose una extremadamente grande cantidad de posibles valores primos, que es imposible de graficar.
- (n) Orden: el orden de la curva no es más que la cantidad de posibles valores en los cuales podremos aterrizar en la curva.

- (G) Generador: este es un punto de la curva elegido por proporciona un gran orden además de no presentar flaquezas o vectores de ataque criptográficos conocidos. Como veremos más adelante este punto será el comienzo de nuestras operaciones sobre la curva.

```
a = 0
b = 7

# prime field
p = 2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 - 1

# order
n =
115792089237316195423570985008687907852837564279074904382605163141518161494433
7

G = [# genarator point
55066263022277343669578718895168534326250603453777594175500187360389116729240
,
32670510020758816978083085130507043184471273380659243275938904335757337482424
]
```

EllipticCurve.ipynb

5.1.2 Operaciones en curvas elípticas

Para poder firmar y verificar vamos a necesitar simplemente de cuatro operaciones. Estas operaciones, aunque nombradas y denotadas de igual manera que sus análogas operaciones en aritmética, sus ejecuciones son diferentes puesto que estas son ejecutadas en el dominio de las curvas elípticas. Simplemente conservar el nombre puesto que comparten ciertas propiedades matemáticas con sus hermanas de toda la vida.

5.1.2.1 Inverso de un entero módulo p

La siguiente función hace uso del algoritmo extendido de Euclides para calcular el inverso de un número. Tiene más relación con campos finitos que con curvas elípticas; pero al trabajar con curvas elípticas definidas en campos finitos, haremos buen uso de ella.

```
def inverse(a, m = p):
    m_orig = m
    if a == 0:
        raise ZeroDivisionError(f"Variable a={a}; should be bigger than 0.")
    a = a % m
    prevy, y = 0, 1
    while a > 1:
        q = m // a
        y, prevy = prevy - q * y, y
        a, m = m % a, a
    return y % m_orig
```

EllipticCurve.ipynb

Al usar una curva definida en un campo finito nos encontramos con el problema de definir la división en esta. Pero podemos obtener el resultado de la división al multiplicar por el inverso de un número en un

campo primo finito. Esto solo funciona para campos finitos con números primos, puesto que estos solo son divisibles por 1 y por sí mismos.

```
def animation(y, m, x = 8):

    for i in range(x, x*y+1):
        print("Finite Fields: Multiplication")
        pos = i % m

        print(" "+" "*pos+str(pos))
        print("0"+" "*pos+"\u21E9"+" "*(m-pos-1)+str(m))
        print("|"+"\u2022"*m+"|")
        print(" "+" "*x+"\u21EA")

        print("\n{} * {} = {}".format(x, y, pos))

        if i == x:
            sleep(1)
        else:
            sleep(.05)
        clear_output(wait=True)
    result = pos
    sleep(1.2)
    hit = []
    for i in range(result + 1, result * pow(y, -1, m) + 1):
        print("Finite Fields: Multiplication (Find Inverse)")
        pos = i % m
        if not i % result:
            hit.append(pos)

        print(" "+" "*pos+str(pos))
        print("0"+" "*pos+"\u21E9"+" "*(m-pos-1)+str(m))
        print("|"+"".join(['\u2022' if i not in hit else
'\033[92m\u2022\033[0m' for i in range(m)])+"|")
        print(" "+" "*x+"\u21EA")

        print("\n{} * {} = {}".format(x, y, result))
        print("{} * {} = {}".format(result, i // result, pos))

        sleep(.02)
        clear_output(wait=True)
```

EllipticCurve.ipynb

La manera en la que lo visualizo e intento mostrar con esta pequeña animación es que para si tenemos nuestros números representados en una línea recta, la multiplicación por un número positivo sería el avanzar hacia adelante saltando por los diversos múltiplos de nuestro número hasta aterrizar en el resultado. La división entre un número positivo sería la de retroceder saltando por los múltiplos de ese número hasta aterrizar en el resultado.

Ahora bien, en el campo finito la división es saltar hacia adelante por lo diversos múltiplos del número aplicando resto módulo p en caso de que rebasemos el rango de nuestro campo finito hasta aterrizar en nuestro resultado.

Finite Fields: Multiplication (Find Inverse)

8 * 13 = 10
10 * 29 = 8

El código y resultado anterior es una pequeña animación la cual ayuda bastante a visualizar el funcionamiento de la inversa modular.

5.1.2.2 Doblar

Doblar un punto viene a ser el equivalente a multiplicar un punto por dos o añadir dicho número a sí mismo en aritmética. Se le llama doblar puesto que tiene propiedades en común con el doblar un número en las matemáticas ordinarias.

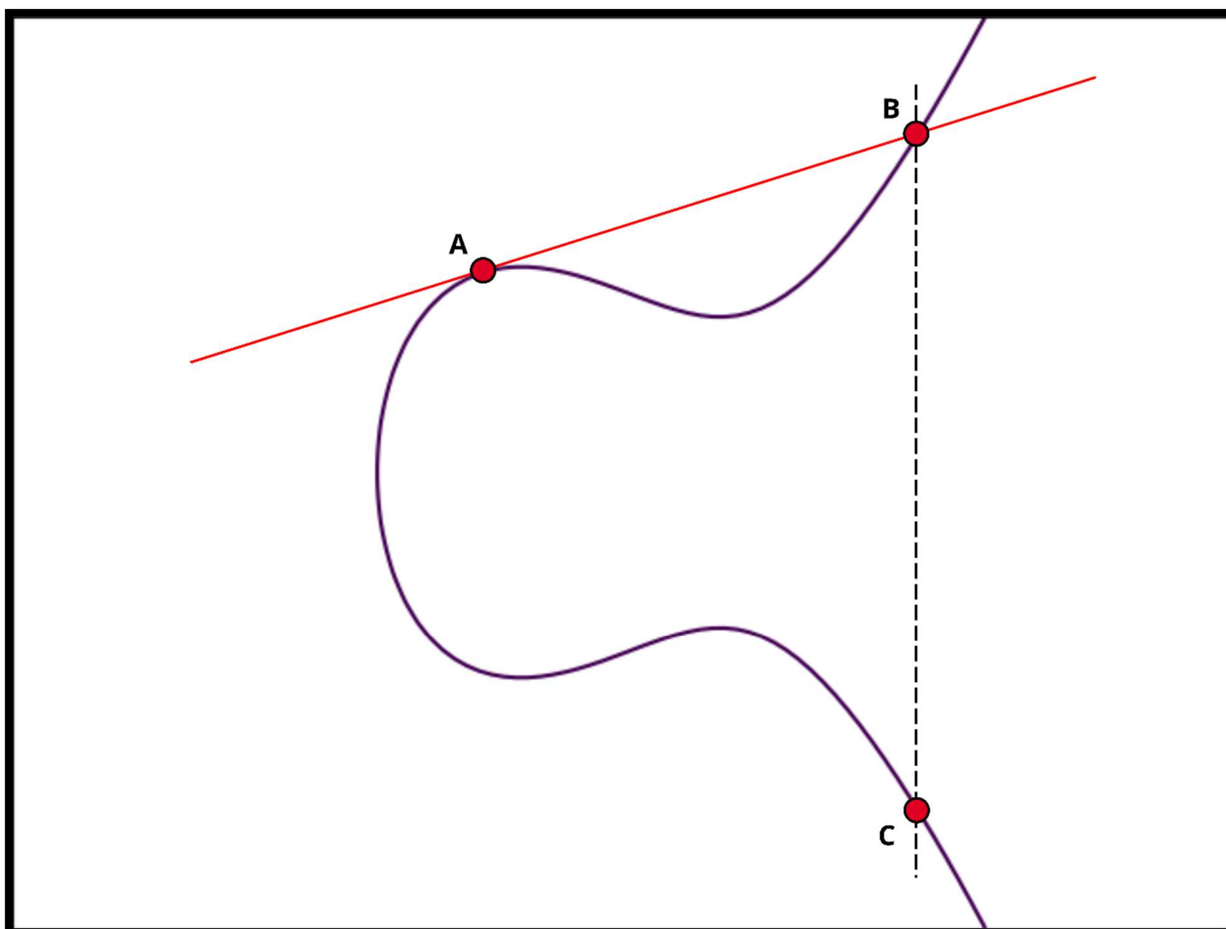


Ilustración 13

Digamos que queremos doblar el punto (A), primero trazamos una tangente a la curva la cual pase por dicho punto; esta tangente cortará la curva en otro punto (B). Encontramos el punto simétrico en la curva con respecto al eje X y este es nuestro resultado.

```
def double(point):
    slope = ((3 * point[0] ** 2 + a) * inverse((2 * point[1]), p)) % p
    x = (slope ** 2 - (2 * point[0])) % p
    y = (slope * (point[0] - x) - point[1]) % p
    return (x, y)
```

EllipticCurve.ipynb

Tal y como podemos ver el código, doblar un punto en la curva consta de tres pasos:

1. Primero hallamos la pendiente de la recta tangente a la curva y que contiene al punto a doblar con la fórmula inferior. La división la realizaremos con la función "inverse()" la cual hace uso del algoritmo de Euclides extendido y nos ayuda con la comprobación del denominador.

$$s = \frac{3x^2 + a}{2y}$$

2. Después se halla X con la fórmula inferior, esta es la X tanto en (B) como en su reflejo puesto que las curvas poseen simetría sobre el eje x.

$$x_{new} = s^2 - 2x$$

3. Y finalmente se calcula la Y; ya que para esta necesitaremos haber calculado previamente nuestra X denotada en la formula inferior por Xnew.

$$y_{new} = s(x - x_{new}) - y$$

El siguiente punto sería el resultado de doblar el punto anteriormente definido como punto generador (G).

```
double(G): {
x: 89565891926547004231252920425935692360644145829622209833684329913297188986597,
y: 12158399299693830322967808612713398636155367887041628176798871954788371653930
}
```

Hay un caso especial el cual sería el de doblar el punto con y = 0. Ya que la tangente de este sería perpendicular al eje vertical, la línea trazada por este no interceptaría la curva en ningún otro lugar. Llamaremos a este punto, punto del infinito o elemento identidad en lo que a las curvas elípticas se refiere.

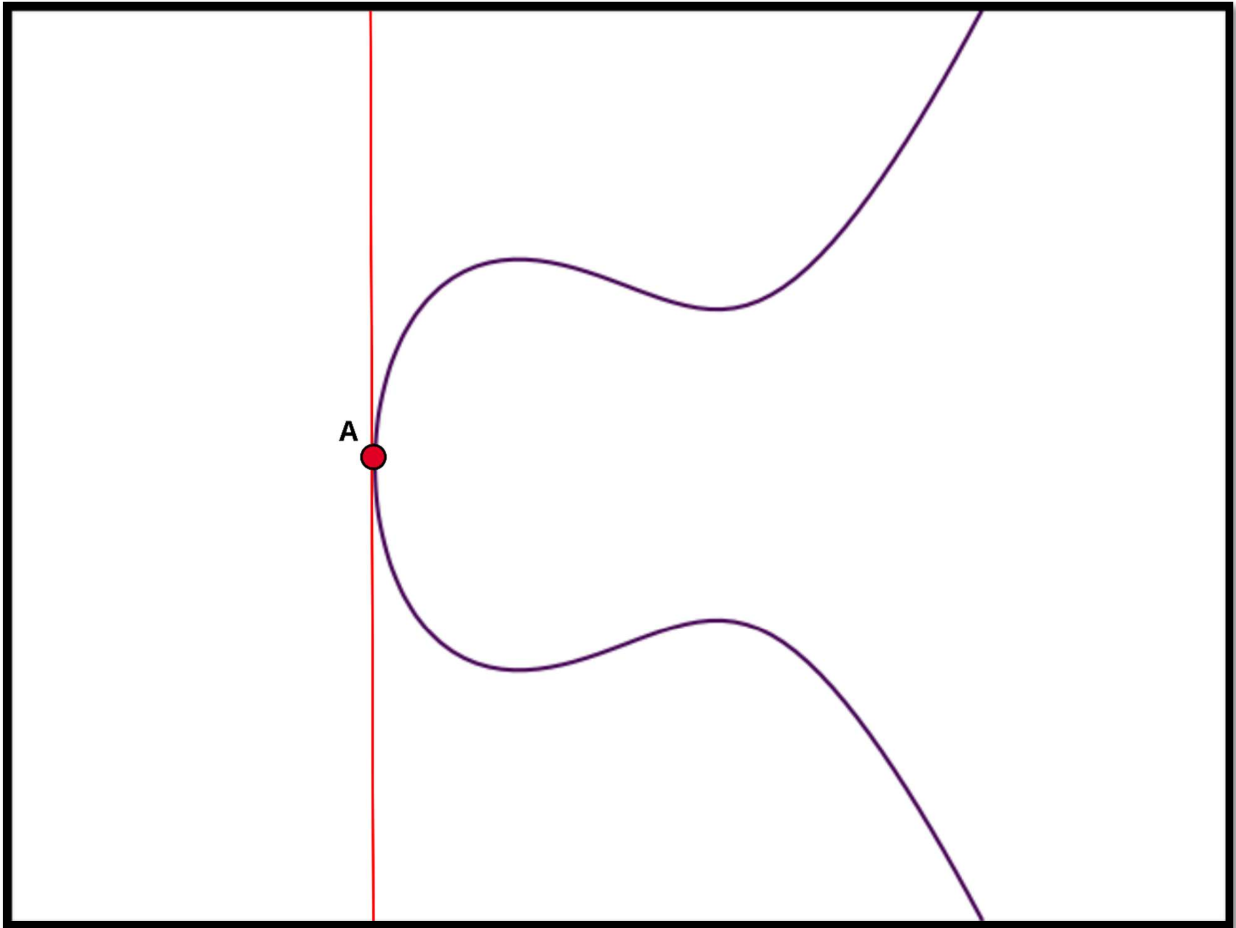


Ilustración 14

5.1.2.3 Suma

De igual manera sumar un punto viene a ser el equivalente al sumar un número con otro en aritmética. Se le llama suma puesto que tiene propiedades en común con la suma aritmética.

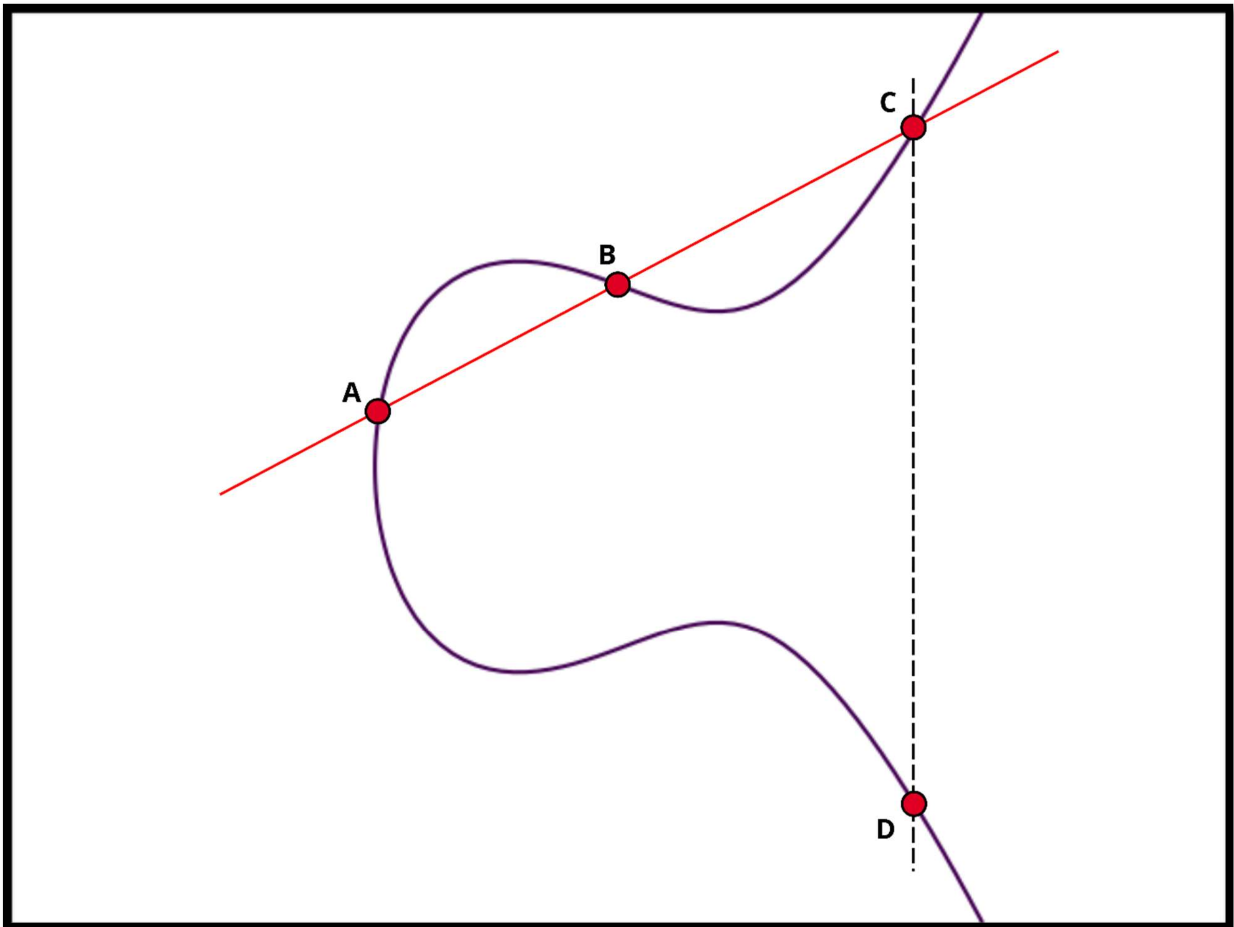


Ilustración 15

Para la suma empezaremos con dos puntos en la curva en este caso (A) y (B), trazaremos una línea recta que pase por estos dos, y una vez más hallaremos el punto simétrico respecto al eje X donde la línea interseca la curva en (C).

```
def add(point1, point2):
  if point1 == point2:
    return double(point1)
  slope = ((point1[1] - point2[1]) * inverse(point1[0] - point2[0], p)) % p
  x = (slope ** 2 - point1[0] - point2[0]) % p
  y = ((slope * (point1[0] - x)) - point1[1]) % p
  return (x, y)
```

EllipticCurve.ipynb

Pare ello seguiremos los pasos parecidos a los anteriormente utilizados en la operación doblar:

1. Primero hallar la pendiente de la curva; en este case la curva que pase por (punto1, punto2). Para ello usaremos nuestra función la cual hace uso del algoritmo extendido de Euclides, así como comprueba que no acabemos con un cero en el denominador.

$$s = \frac{y_1 - y_2}{x_1 - x_2}$$

2. Después se calcular la tercera intersección que esta línea genera con la curva elíptica.

$$x_{new} = s^2 - x_1 - x_2$$

3. Y finalmente se calcula el reflejo de este tercer punto sobre el eje horizontal.

$$y_{new} = s(x_1 - x_{new}) - y_1$$

De igual manera que la aritmética básica, sumar un número consigo mismo nos dará el mismo resultado que el multiplicarlo por dos o doblarlo. Por lo que en el siguiente resultado ha de ser el mismo que obtuvimos al doblar el punto generador (G).

```
G + G =: {
x: 89565891926547004231252920425935692360644145829622209833684329913297188986597,
y: 12158399299693830322967808612713398636155367887041628176798871954788371653930
}
```

De igual manera que se describió anteriormente se puede dar el caso de que los puntos a sumar sean los puntos simétricamente opuestos sobre el eje X de nuestra curva. Y de igual manera que sucedía al doblar el punto con y=0, el resultado de esta operación seria la del punto infinito.

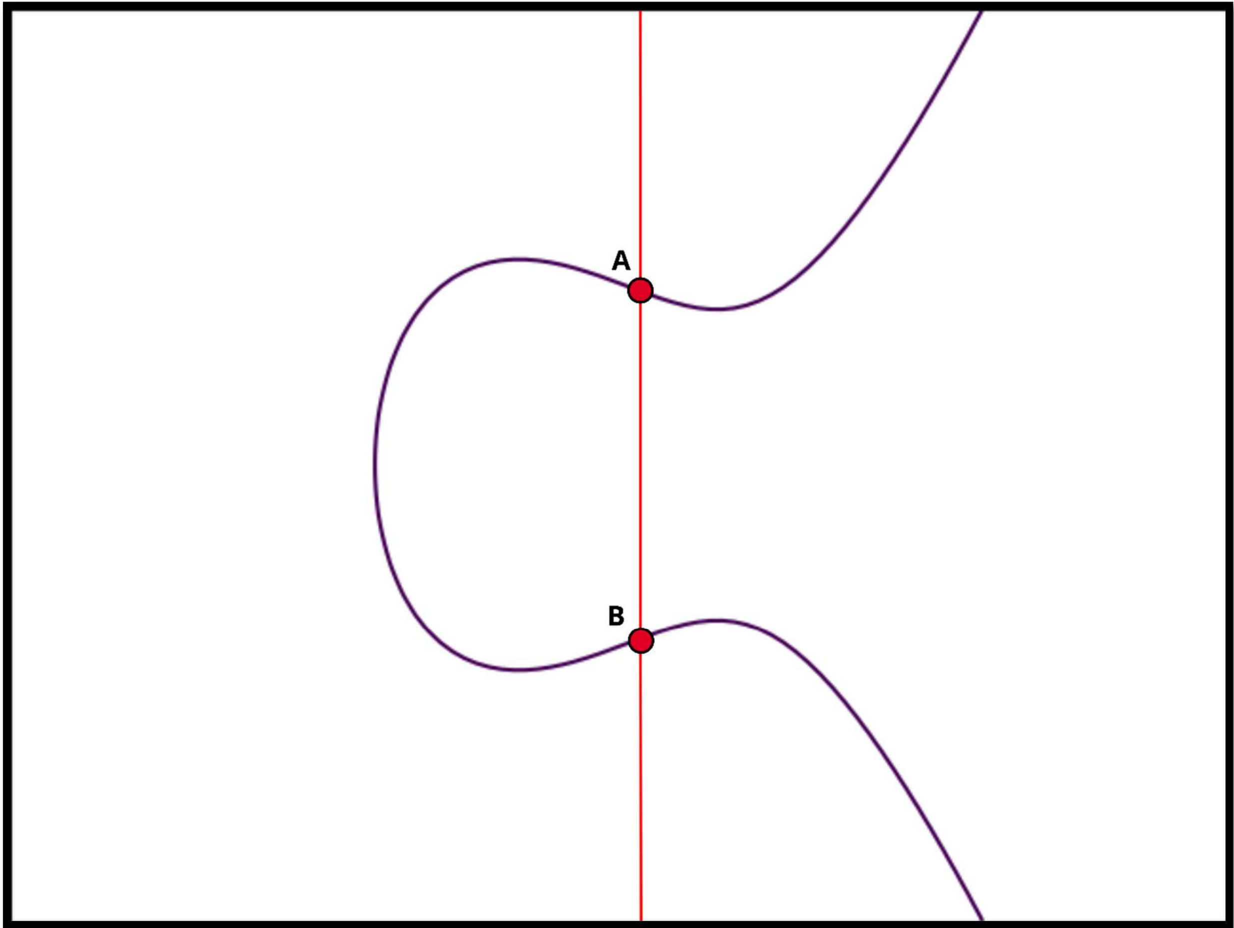


Ilustración 16

5.1.2.4 Multiplicación de un punto por un número

Para acabar necesitaremos de multiplicación en curvas elípticas, este es el verdadero corazón detrás de la criptografía de curvas elípticas.

```
def multiply(constant, point = G):
    acu = add(point, point)
    for _ in range(constant-2):
        acu = add(acu, point)
    return acu

start = time()
result = multiply(40000)
print(f"Execution time: {(time() - start)*1000} ms")
print(f"Result: {{\n\tx: {result[0]},\n\ty: {result[1]}\n}}")

del start, result
```

EllipticCurve.ipynb

Para ello podríamos hacer un uso repetido de la operación suma sobre curvas elípticas del mismo modo que se hace en aritmética, una sucesión de sumas del mismo número. El problema que tenemos con este enfoque es la lentitud del algoritmo, la cual solo empeora cuando hacemos uso de los grandes números con los que trabajaremos en este espacio.

```
Execution time: 2028.754711151123 ms
Result: {
    x: 45144681238022915379838256640648974390095578890737782493692183842300544374804,
    y: 15493911184894630123643928078266102650003862558761004463646228136434723692345
}
```

Es por ello por lo que haremos uso del método de multiplicación denominado como “Double-and-add algorithm” o en castellano algoritmo de doblar y sumar equivalente a la exponenciación modular de enteros (cuadrados y productos). Este algoritmo combina las operaciones de doblar un número en curvas elípticas y la suma en curvas elípticas para agilizar el proceso de la multiplicación.

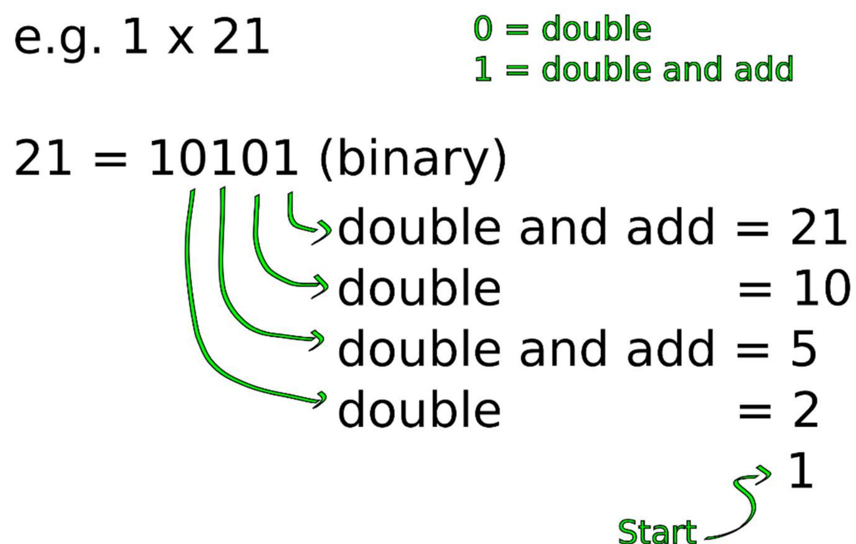


Ilustración 17

Para este procedimiento tendremos que hacer lo siguiente. Primero pasaremos el número a binario, este nos proporcionará la secuencia con la que debemos llevar a cabo la multiplicación.

El primer bit se ignora ya que siempre se empieza doblando; y después se sigue la secuencia; (0 == doblar), (1 == doblar y sumar) hasta haber recorrido todos los bits.

```
def double_add__multiply(constant, point = G): # Double and add algorithm
    current = point
    binary = bin(constant)[3:]
    for i in binary:
        current = double(current)
        if i == '1':
```

```
            current = add(current, point)
    return current

start = time()
result = double_add_multiply(40000)
print(f"Execution time: {(time() - start)*1000} ms")
print(f"Result: {{\n\tx: {result[0]},\n\ty: {result[1]}\n}}")

del start, result
```

EllipticCurve.ipynb

Al realizar la multiplicación de esta manera nos ahorramos bastantes operaciones de suma sobre curva elíptica que extrapolada a los gigantes números con los que se operan en ECDSA nos supondrá una gran mejora en tiempo de ejecución.

```
Execution time: 0.9989738464355469 ms
Result: {
x: 45144681238022915379838256640648974390095578890737782493692183842300544374804,
y: 15493911184894630123643928078266102650003862558761004463646228136434723692345
}
```

En ambos resultados hemos multiplicado sobre la curva elíptica el punto generador (G) por 4000, y podemos ver como el algoritmo "Double-and-add-algorithm" muestra una reducción de tiempo de ejecución muy considerable, donde el tamaño del bucle principal del algoritmo tiene tantas iteraciones como bits del número "constant". Tan considerable que al hacer uso de números tan grandes como para ser criptográficamente seguros el primer método quedaría totalmente inutilizable.

5.1.3 Generación de claves

Como ya sabemos en criptografía asimétricas tendremos dos claves llamadas clave privada la cual denotaremos con la letra (d) y nuestra clave pública la cual denotaremos por la letra (Q).

```
d =
112757557418114203588093402336452206775565751179231977388358956335153294300646 # Pvivate key
Q = double_add_multiply(d)

print("Public {}: {{\n\tx: {},\n\ty: {}\n}}".format("\U0001F511",Q[0], Q[1]))
```

EllipticCurve.ipynb

La clave pública será un número comprendido entre 0 y (n – 1). Este número ha de ser obtenido de una fuente aleatoria criptográficamente segura para garantizar su robustez.

```
Private 🔑:
112757557418114203588093402336452206775565751179231977388358956335153294300646
```

La clave pública la derivaremos de nuestra clave privada con ayuda de la operación de multiplicación sobre curvas elípticas. El procedimiento es bastante simple, solo hay que multiplicar el punto generador de nuestra curva (G) por la clave pública y el punto resultante de esta operación será nuestra clave pública.

$$Q = dG$$

Public 🔑: {
x: 33886286099813419182054595252042348742146950914608322024530631065951421850289,
y: 9529752953487881233694078263953407116222499632359298014255097182349749987176
}

La clave pública es la que podremos compartir posteriormente para validar nuestra firma digital. Esto no presenta ningún riesgo puesto que la multiplicación sobre curvas elípticas es una función logarítmica discreta en curvas elípticas, la cual es muy fácil y sencilla de calcular en una dirección, pero extremadamente computacionalmente intensiva en el sentido contrario. Es por esto por lo que no se puede obtener la clave privada conociendo únicamente la clave pública, el punto generador y la curva elíptica usada.

5.1.4 Firma

Para firmar un mensaje necesitaremos lo siguiente:

- (k) Salt: un número aleatorio el cual introducirá ruido y entropía en nuestra firma para que esta no se pueda revertir al cruzar la información de distintas firmas creadas con el mismo par de claves.
- (z) Hash del mensaje: el resultado hash de nuestro mensaje; el cual en este caso será el digest del algoritmo hash SHA-256 el cual mostramos anteriormente.
- (d) Clave privada: nuestra clave privada con la que firmaremos el mensaje.

```
k = 1818181818181818181818181
print(f"Salt: {k}")

message = "ECDSA (Elliptic Curve Digital Signature Algorithm)"
print(f"Message: {message}")

z = sha256(message.encode('ascii')).digest()
z = int.from_bytes(z, 'big')
print(f"Hash digest in decimal: {z}")

print(f"Private key: {d}")
```

EllipticCurve.ipynb

En este ejemplo usaremos un Salt el cual no es considerado seguro, pero esto nos asegura que el resultado de la firma resultante será siempre el mismo para el mismo mensaje y clave.

También hay que destacar que el resultado del hash en la función usada en este caso es un string en hexadecimal el cual tendremos que decodificar como un número natural en big-endian.

Salt: 1818181818181818181818181

```
Message: ECDSA (Elliptic Curve Digital Signature Algorithm)

Hash digest in decimal:
49410833204725140173526597860441273720434304053996485655430536493502439593083
Private key:
112757557418114203588093402336452206775565751179231977388358956335153294300646
```

Una vez hayamos inicializado nuestras variables podremos continuar con la firma digital, para ello dividiremos este proceso en dos partes.

```python
def sign(private_k, m_hash, salt = None):

    if salt == None:
        salt = randbelow(n) # Random number within finite field order.

    r = double_add_multiply(salt)[0] % n

    s = (inverse(salt, n) * (m_hash + private_k * r)) % n

    return (r, s)

signature = sign(d, z, k)

print(f"Signature: {{\n\tx: {signature[0]},\n\ty: {signature[1]}\n}}")
```

EllipticCurve.ipynb

Puesto que una firma digital sobre curvas elípticas no es más que un punto en el dominio de esta. Este punto está formado por dos componentes, una componente (x) y una componente (y).

```
Signature: {
x: 54767518919804132415261250426954620798001685863099591615502827291246864032477,
y: 98602259575904705330380232146105755454407891657434652824036155561569926413905
}
```

Primero hallaremos el valor de nuestra (x) con la siguiente fórmula. Esta multiplicará nuestro punto generador (G) por la Salt (k) anteriormente definida, y finalmente se calculará el módulo con el orden de nuestra CE (n) para cerciorarnos de que el valor aterriza en uno de los posibles valores del conjunto que define nuestra curva.

$$\boldsymbol{r} = (\boldsymbol{G} \times \boldsymbol{k})\boldsymbol{mod}(\boldsymbol{n})$$

Esta operación nos dará un punto como resultado del cual solo utilizaremos la componente (x) que será almacenada en nuestra variable (r) y que usaremos posteriormente como componente (x) de nuestra firma, y con la ayuda de la siguiente función para hallar nuestra componente (y).

$$\boldsymbol{s} = (\boldsymbol{k}^{-1} \times (\boldsymbol{z} + \boldsymbol{d} \times \boldsymbol{r}))\boldsymbol{mod}(\boldsymbol{n})$$

En este paso sumaremos nuestro mensaje (z) a nuestra clave privada (d), multiplicaremos el resultado por nuestra componente anteriormente hallada (r) y posteriormente por la inversa de nuestra Salt (k).

Finalmente, y de igual forma que en operación anterior, realizaremos la operación módulo con el orden de nuestra CE (n) para cerciorarnos de que el resultado quede en nuestro conjunto de posibles valores.

5.1.4.1 *¿Qué pasa si usamos un Salt no seguro en nuestra firma?*

Como ya mencionamos es de gran importancia que el Salt que utilicemos en nuestra firma sea un número realmente aleatorio el cual varíe con cada una de las firmas que generemos. Ya que si este no es lo suficientemente seguro se podría revertir la operación para hallar nuestro Salt (k) y con este y una firma la cual haga uso de el podríamos incluso llegar a hallar la clave privada. Llegados a este punto todos los datos firmados con esta quedarían comprometidos inclusive los firmados usando Salt criptográficamente seguros.

```
prv_k = 111122223333444455556666777788889999 0000
pub_k = double_add_multiply(prv_k)

k = 12345 # No random on pourpose.

digest1 = int.from_bytes(sha256(b'Just a simple message.').digest(), 'big')
bad_sig1 = sign(prv_k, digest1, k)

digest2 = int.from_bytes(sha256(b'I have used the same k value.').digest(),
'big')
bad_sig2 = sign(prv_k, digest2, k)

k_recovery = ((digest1 - digest2) * inverse(bad_sig1[1] - bad_sig2[1], n)) %
n

d_recovery = ((k_recovery * bad_sig1[1] - digest1) * inverse(bad_sig1[0], n))
% n

print("Private key ({}) reversed: {}".format(prv_k, d_recovery == prv_k))
```

EllipticCurve.ipynb

El primer paso será el de hallar el valor del Salt (k) usado en una firma digital, para ello necesitaremos tener dos firmas digitales las cuales hayan usado el mismo valor Salt (k), hayan firmados distintos mensajes y empleen la misma clave privada (d).

Private key (1111222233334444555566667777888899990000) reversed: True

Denotaremos a los hashes de estos mensajes como (d1 y d2) respectivamente y a las firmas digitales de estos como (s1 y s2).

Empleando la siguiente fórmula siguiente fórmula extraeremos el valor de la Salt (k) usada:

$$\boldsymbol{k = ((d1 - d2) \times (s1_y - s2_y)^{-n})mod(n)}$$

Donde Sx sub-x denota la componente (x) del punto y Sx sub-y denota la componente (y) del punto.

Una vez hemos recuperado el valor del Salt (k), podremos extraer la clave pública de una firma la cual hallar sido firmada con este Salt.

$$d = ((k \times s1_y - d1) \times (s1_x)^{-n}) mod(n)$$

De este ejemplo podemos aprender que debemos usar una Salt segura para cada una de nuestras firmas, así como que debemos tener cuidado con el no revelar esta, no hay necesidad de guardar la Salt ya que esta es de un solo uso, pero tener siempre en cuenta que el proceso de firma es un proceso delicado puesto que este hace uso de valores privados o secretos como pueden ser las claves privadas o Salt.

5.1.5 Verificación

Al igual que para firmar, en el caso de la verificación vamos a necesitar lo siguiente:

- (Q) Clave pública: La clave pública relacionada con la clave privada que usamos para firma.
- (z) Hash del mensaje: el resultado hash de nuestro mensaje; el cual en este caso será el digest del algoritmo hash SHA-256 el cual mostramos anteriormente.
- ([r, s]) Firma digital: firma digital la cual hallamos en el punto anterior.

```
def verify(public_k, signature, hash):

    point1 = double_add_multiply(inverse(signature[1], n) * hash)
    point2 = double_add_multiply((inverse(signature[1], n )* signature[0]),
public_k)

    point3 = add(point1, point2)
    return point3[0] == signature[0]


print(f"SHA-256 digest: {z}")
print(f"Signature [r, s]: {{\n\tx: {signature[0]},\n\ty: {signature[1]}\n}}")
print(f"Public key Q: {{\n\tx: {Q[0]},\n\ty: {Q[1]}\n}}")

verify(Q, signature, message)
```

EllipticCurve.ipynb

El procedimiento es el siguiente. Tendremos que hallar dos puntos en la curva con las fórmulas que podremos encontrar a continuación. Acto seguido sumaremos estos dos puntos con la ayuda de la nuestra función (add()) la cual suma dos puntos sobre una curva elíptica. Y si la componente (x) del punto resultante coincide con la componente (x) de nuestra firma electrónica, la firma será válida.

```
SHA-256 digest:
49410833204725140173526597860441273720434304053996485655430536493502439593083

Signature [r, s]: {
x: 54767518919804132415261250426954620798001685863099591615502827291246864032477,
y: 98602259575904705330380232146105755454407891657434652824036155561569926413905
}

 Public key Q: {
x: 33886286099813419182054595252042348742146950914608322024530631065951421850289,
y: 95297529534878812336940782639534071162224996323592980142550971823497499871 76
```

}

☑ ECDSA (Elliptic Curve Digital Signature Algorithm)

Podemos agrupar las operaciones mencionadas anteriormente en una sola, y denominaremos el resultado de esta como (R).

$$\boldsymbol{R} = (\boldsymbol{s}^{-1} \times \boldsymbol{z})\boldsymbol{G} + (\boldsymbol{s}^{-1} \times \boldsymbol{r})\boldsymbol{Q}$$

Este punto resultante será del cual extraeremos la primera componente para compararla con nuestra firma digital y verificar que los datos fueron firmados por la clave privada correspondiente a la clave pública usada en la verificación.

5.1.6 Codificaciones

Puesto que de ahora en adelante haremos uso de las firmas digitales necesitaremos una forma robusta y consistente de manejar las claves. Es por ello por lo que en este punto vamos a ver las formas más usadas a la hora de codificar las claves y firmas.

5.1.6.1 DER

La codificación DER (Distinguished Encoding Rules) esta será la forma que usaremos para codificar nuestras firmas digitales. La manera en la que esta funciona es la siguiente:

1. Un byte a 0x30
2. El largo de la firma digital el cual suele ser 0x44 o 0x45
3. Un byte que usaremos como separador con el valor 0x02
4. El largo del campo codificado a continuación.
5. El valor (r) de nuestra firma como un número entero en bit-endian. Si el primer byte de (r) es mayor o igual a 0x80 añadiremos el byte 0x00 al principio de este.
6. Otro byte separador con valor 0x02
7. El largo del campo a seguir.
8. El valor (s) de nuestra firma como un número entero en bit-endian. Si el primer byte de (s) es mayor o igual a 0x80 añadiremos el byte 0x00 al principio de este.

```
def der_encoding(signature):

    rbin = signature[0].to_bytes(32, byteorder='big')
    rbin = rbin.lstrip(b'\x00')

    if rbin[0] & 0x80:
        rbin = b'\x00' + rbin
    result = bytes([2, len(rbin)]) + rbin

    sbin = signature[1].to_bytes(32, byteorder='big')
    sbin = sbin.lstrip(b'\x00')

    if sbin[0] & 0x80:
        sbin = b'\x00' + sbin
    result += bytes([2, len(sbin)]) + sbin
```

```
    return bytes([0x30, len(result)]) + result

encoded_sig = der_encoding(signature)

hex_sig = encoded_sig.hex()
```
EllipticCurve.ipynb

```
3045022079155134 5a2e80dc53dd2d607429c98be30c5bfa38258dfeb722e049b696e2dd022100d9f
ee6b4e7fef80cf80770eb04f01db3be7fe024cc53f4b73435cc782d3a7e51
```
- Marker
- Length of sig
- [r, s] markers
- [r, s] lengths
- [r, s]

5.1.6.2 SEC sin compresión

Este método lo usaremos para codificar nuestras claves públicas. Este es muy sencillo ya que solo tendremos que añadir un byte a valor 0x04 al comienzo de nuestra clave.

```
def uncompress_sec(public_key):
    return b'\x04' + public_key[0].to_bytes(32, 'big') +
public_key[1].to_bytes(32, 'big')

uncompressed_pk = uncompress_sec(Q)
uncompressed_pk_hex = uncompressed_pk.hex()
```
EllipticCurve.ipynb

```
044aeaf55040fa16de37303d13ca1dde85f4ca9baa36e2963a27a1c0c1165fe2b11511a626b232de4
ed05b204bd9eccaf1b79f5752e14dd1e847aa2f4db6a52768
```
- Marker
- X coordinate
- Y coordinate

5.1.6.3 SEC con compresión

Puesto que sabemos la curva elíptica con la que estamos trabajando podemos simplemente transmitir el punto o componente (x) de nuestra clave pública y así ahorrarnos casi la mitad de los datos. Solo hay un pequeño problema con esto y es que la curva es simétrica sobre el eje (x) lo que hace que para un único valor de la primera componente habrá dos puntos que satisfagan esta condición.

```
def sec_pk(public_key, compressed=True):
        # return byte version of the PK
        if compressed:
            if public_key[1] % 2 == 0:
                return b'\x02' + public_key[0].to_bytes(32, 'big')
            else:
                return b'\x03' + public_key[0].to_bytes(32, 'big')
        else:
            return b'\x04' + public_key[0].to_bytes(32, 'big') +
public_key[1].to_bytes(32, 'big')
```

```
compressed_pk = sec_pk(Q)
compressed_pk_hex = compressed_pk.hex()
```

EllipticCurve.ipynb

La manera que tenemos de solucionar esto es la de usar dos valores en nuestro byte añadido al comienzo con los valores:

- 0x02 en caso de que la segunda componente de la clave pública sea positiva.
- 0x03 en caso contrario (negativa).

```
024aeaf55040fa16de37303d13ca1dde85f4ca9baa36e2963a27a1c0c1165fe2b1
█ Marker
█ X coordinate
```

6 FRASE MNEMÓNICA

En este apartado vamos a ver como guardar de manera simple e intuitiva un gran número del cual podremos derivar nuestras claves privadas de forma determinista y por consiguiente nuestra cartera Bitcoin. Este método/estándar fue introducido por BIP (Bitcoin Improvement Proposal) 39 y hoy en día es utilizado por un gran número de criptomonedas y transacciones compatibles.

6.1 ENTROPÍA

Lo primero que necesitaremos será ese gran número que acabamos de mencionar el cual usaremos como semilla de nuestra cartera. Este ha de ser completamente aleatorio.

Preferiblemente creado por el sistema operativo, hay un gran número de librerías que hacen uso de este tipo de archivos de bits aleatorios o llamadas al sistema. En este caso y juzgando por su documentación yo me he decantado por la librería o paquete de Python: “secrets”.

```
def entropy_gen(n = 128):
    if n < 128 or n > 256:
        raise Exception("Entropy size must be between 128 and 256 bits 
inclusive.")
    if n % 32 != 0:
        raise Exception("Entropy size must be multiple of 32.")
    return token_bytes(n // 8) # Secret module function

entropy = entropy_gen(32*8)
```

HD_wallet.ipynb

Nuestro string aleatorio ha de ser de longitud comprendida entre 128 y 256 bits, ambos incluidos y a su vez múltiplo de 32. Números tan grandes son usados para que la probabilidad de generar dos veces el mismo número de manera aleatoria es tan baja, como para considerarlo criptográficamente seguro.

Entropy (hex): 5558bc76602e06b585ecb25a5b80b56d0cdf0838efd96abb5aad949512270334

Con el fin de que ensenar los resultados de nuestras celdas de código y el procedimiento de obtener una cartera criptográfica de esta semilla, usaremos la entropía aquí mostrada para que los ejemplos aquí mostrados puedan ser replicados y observados de la misma forma.

6.2 CÓDIGO DE COMPROBACIÓN DE ERRORES

Una vez tengamos nuestra entropía calcularemos un checksum de esta el cual incluiremos de forma permanente en nuestra frase mnemónica y el cual nos servirá para comprobar errores en esta. Cabe decir que este checksum puede detectar errores, pero no corregirlos, por lo que es de vital importancia que no perdamos nuestra frase mnemónica o perderemos acceso a los fondos guardados en nuestra cartera.

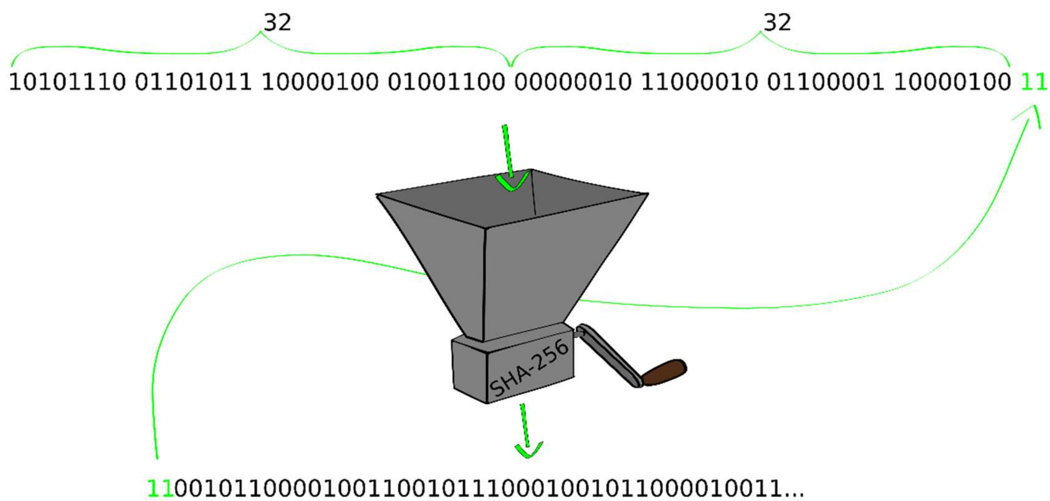


Ilustración 18

Para calcular el checksum, realizaremos el hash SHA-256 de nuestra entropía y por cada 32 bits o 4 bytes de esta, cogeremos un bit del hash resultante y lo añadiremos al final de nuestro string de entropía.

Entropy (hex): 5558bc76602e06b585ecb25a5b80b56d0cdf0838efd96abb5aad949512270334
Entropy SHA-256: ae2b298947e83beadfdb346e5a536de8a1a918246987dea9a7128e384b9f1a30

```
0101010101011000101111000111011001100000001011100000001101011010110000101111011001
0110010010110100101101110000000101101010110110100001100110111110000100000011100011
1011111101100101101010101110110101101010101101100101001001010100010010001001110001
00011001101001010111 0
```

▮ Entropy
▮ Checksum (hash)

6.3 MAPEO DE PALABRAS

El mapeo de palabras está pensado como una forma intuitiva y fácil para ser capaz de recordar este gran número o string de bytes que es la entropía. Usa palabras escogidas cuidadosamente por su simplicidad y falta de ambigüedad para codificar nuestro string de bytes, el cual de ahora en adelante estará formado por entropía más checksum o código comprobador de errores.

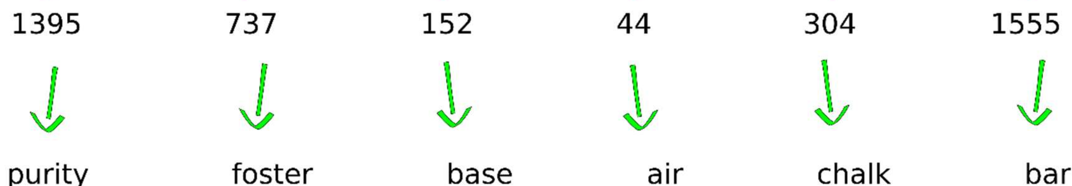


Ilustración 19

Para mapear nuestro string a palabras dividiremos este en substrings de 11 bits de largo, los cuales pasaremos a números decimales en big-endian. Cada uno de estos números hace referencia a una palabra de la lista que podemos encontrar en el archivo de texto (*BIP-0039_english.txt*). Esta lista de palabras está definida por la BIP (Bitcoin Improvement Proposal) 39, la cual especifica listas de palabras, también llamados diccionarios, en una gran variedad de idiomas, de los cuales escogeremos el inglés para llevar a cabo nuestros ejemplos.

Estas palabras son mapeadas de un diccionario de 2048 entradas, en el que cada palabra es única en sus cuatro primeras letras lo que hace más difícil el confundirlas.

6.4 Creación de la frase mnemónica

En el código mostrado a continuación pondremos los cuatro bloques mostrados con anterioridad a trabajar juntos para generar una frase mnemónica.

```
def mnemonic(bits):
    # Entropy 2 binary
    entropy_bits = bitarray()
    entropy_bits.frombytes(bits)

    # Checksum
    hash256 = hashes.Hash(hashes.SHA256())
    hash256.update(bits)
    bin_h = bitarray()
    bin_h.frombytes(hash256.finalize())

    entropy_bits = entropy_bits + bin_h[:(len(entropy_bits) // 32)]

    numbers = [] # words numbers
    for i in range(len(entropy_bits)//11):
        numbers.append(util.ba2int(entropy_bits[11*i:11*i+11]))

    words = {} # words in our entropy
    with open('BIP-0039_english.txt', 'r') as file:
        for i, word in enumerate(file):
            if i in numbers:
                words[i] = word
                if len(numbers) == len(words):
```

```
                    break

    mnemonic_sentence = [] # mnemonic
    for i in numbers:
        mnemonic_sentence.append(words[i].strip())

    return tuple(mnemonic_sentence)

mnemonic_sentence = " ".join(mnemonic(entropy))
```
HD_wallet.ipynb

Finalmente acabaremos con una frase de entre 12 y 24 palabras de largo la cual nos servirá como copia de seguridad de nuestra cartera Bitcoin.

fetch shine budget scare there food bless north fog swing coil sure social lottery together will height suspect problem circle eye bean art noise

Como ya veremos más adelante no todas las carteras Bitcoin se pueden almacenar de esta forma ya que en algunos casos como por ejemplo p2sh (Pay to Script Hash), además de nuestra frase mnemónica tendremos que guardar información adicional, como por ejemplo los scripts de las transacciones que todavía no hemos gastado.

6.5 Validación de la frase mnemónica

Gracias al checksum introducido anteriormente podremos comprobar la integridad de nuestra entropía antes de derivar nuestra semilla.

Esto es muy importante puesto que, si introdujésemos una combinación de palabras erróneas y no hubiese ningún comprobante, estas nos derivarían una semilla y consecuentemente una cartera sin ningún error. La cual estaría condenada y todos los fondos almacenados en esta serían irrecuperables, a no ser que seamos capaces de replicar el mismo error consistentemente. Es por ello por lo que se ha de validar la frase mnemónica antes de proseguir con la creación de claves cuando esta es importada en una cartera.

```
def mnemonic_check(mnemonic):

    words = mnemonic.split(" ")
    numbers = [None] * len(words)

    with open('BIP-0039_english.txt') as file:
        for i, word in enumerate(file):
            if word.strip() in words:
                for j, item in enumerate(words):
                    if item == word.strip():
                        numbers[j] = i

    entropy = "".join([f'{i:011b}' for i in numbers])
    entropy = bitarray(entropy)

    checksum_len = len(entropy) % 32
```

```
    hash256 = hashes.Hash(hashes.SHA256())
    hash256.update(entropy[:-checksum_len].tobytes())
    bin_h = bitarray()
    entropy_hash = hash256.finalize()
    bin_h.frombytes(entropy_hash)

    if entropy[-checksum_len:] != bin_h[:checksum_len]:
        # Just raised in case checksum for mneumonic fails.
        raise Exception(f"Checksum fail, hash: {entropy_hash.hex()}")

    return entropy[:-checksum_len].tobytes()
```

HD_wallet.ipynb

Para ello simplemente revertimos el procedimiento, pasando nuestras palabras a números con la lista de palabras, (muy importante tener en cuenta la codificación de estas), la cual ha de ser UTF-8. En nuestro caso usamos ASCII puesto que este es un subset de UTF-8 donde se engloba todos los caracteres que usaremos.

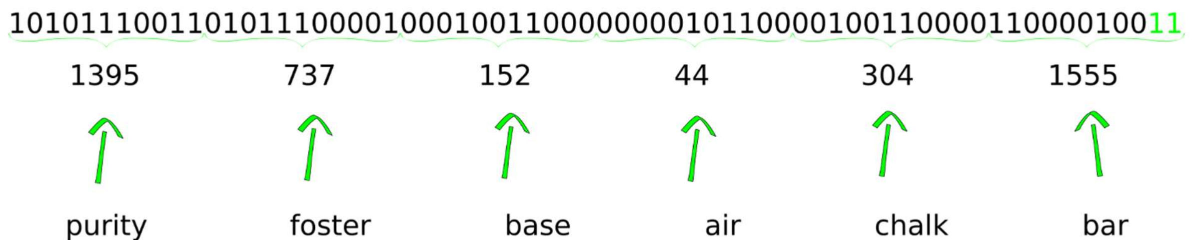


Ilustración 20

Pasamos los números a binario en big-endian y aislamos los bits del checksum. Para finalizar simplemente hay que computar el checksum sobre que los bits que formen la entropía, y comparar el principio de este con los bits anteriormente aislados para así validar nuestra frase mnemónica.

6.6 Creación de la semilla

Aunque podríamos usar la entropía como semilla para nuestra cartera Bitcoin, haremos caso de la especificación y haremos lo correcto derivando una semilla de nuestra frase mnemónica. Este proceso nos ofrece dos grandes beneficios o ventajas:

1. Aumenta la seguridad contra ataques de diccionario o fuerza bruta.
2. Nos ofrece la posibilidad de añadir una contraseña de derivación a nuestra frase mnemónica o backup.

```
def mnemonic2seed(mnemonic_sentence, salt=""):

    mnemonic_dev = PBKDF2HMAC(
        algorithm = hashes.SHA512(),
        length = 64, #bytes
        salt = ("mnemonic"+salt).encode('ascii'),
        iterations = 2048
```

```
    )

    mnemonic_bytes = mnemonic_sentence.encode('ascii')
    seed = mnemonic_dev.derive(mnemonic_bytes)

    return seed

recovery_pass = "" # In case we want to add a salt to our mneumonic, add in
here

seed = mnemonic2seed(mnemonic_sentence, recovery_pass)
```
HD_wallet.ipynb

Para su creación haremos uso del algoritmo PBKDF2 (Password-Based Key Derivation Function 2); a su vez esta función hace uso de HMAC (hash-based message authentication code). Esta función no es más que un simple salted hash SHA-512. Puesto que este algoritmo no admite la omisión del Salt, usaremos como Salt predeterminado el string "mnemonic" y en caso de que queramos añadir una contraseña a nuestra frase mnemónica, se la concatenaremos a este al final.

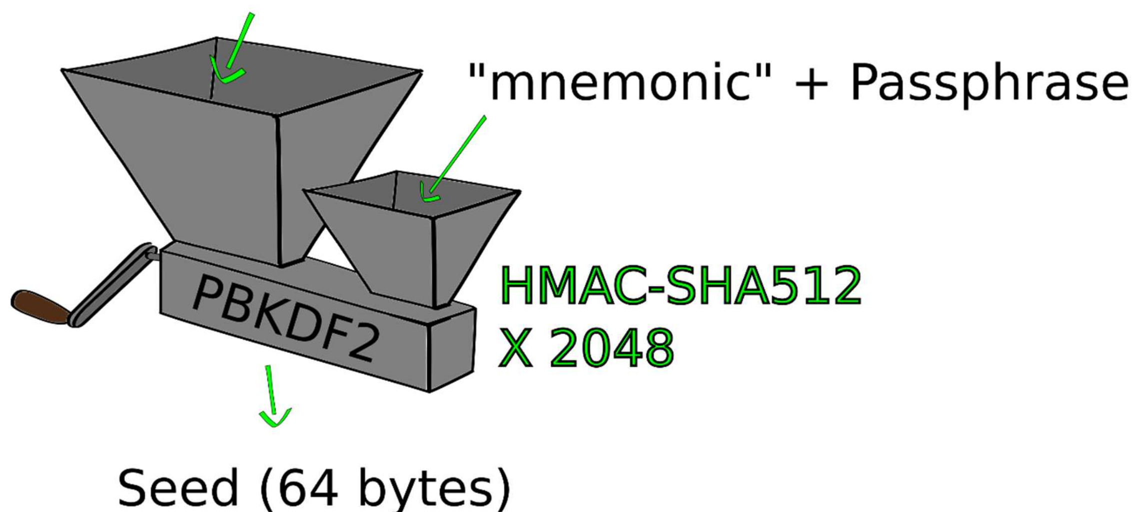


Ilustración 21

Volviendo a PBKDF2, el propósito de este algoritmo es el de ralentizar un posible ataque por fuerza bruta a nuestra semilla; esto lo consigue corriendo el algoritmo HMAC de su interior un número elevado de veces, 2048 para ser más exactos. Haciendo que el atacante tuviese que correr dicha operación 2048 veces por cada combinación que quiera comprobar.

Seed:
b0b04fe4e7b77c4c037d98dbf788811e7ac02eb247f6ba53526b061dc20cff365f15e579cc248a9f020132b32ceeac5a7a1e5b48e16975e8af4b5154a1eee649

El resultado final será de la misma longitud que el algoritmo hash utilizado; en este caso 512 bits o 64 bytes, al que llamaremos la semilla de la cartera.

7 CLAVES EXTENDIDAS

BIP 32, entre otras muchas modificaciones y mejoras, introduce la utilización de claves extendidas. Estas no son más que una clave acompañada de un número pseudoaleatorio, el cual nos servirá para derivar sucesivas claves de manera determinista.

Por razones de seguridad, la manera óptima de realizar transacciones en Bitcoin es que cada una de ellas se realice con un par de claves distinto. De esta manera es más difícil el saber la cuantía contenida en una cartera, y en caso de que una clave se vea comprometida tendremos la certeza de que el resto de los fondos seguirán bajo nuestro poder.

7.1 DERIVACIÓN DE LA CLAVE EXTENDIDA MAESTRA

Esta clave recibe su propio apartado puesto que al ser la primera clave extendida que crearemos a partir de nuestra semilla, seguirá un procedimiento algo diferente.

Llamamos clave maestra a la primera clave derivada de la semilla. Y denotamos a esta con la letra “m”, y que explicaremos más adelante.

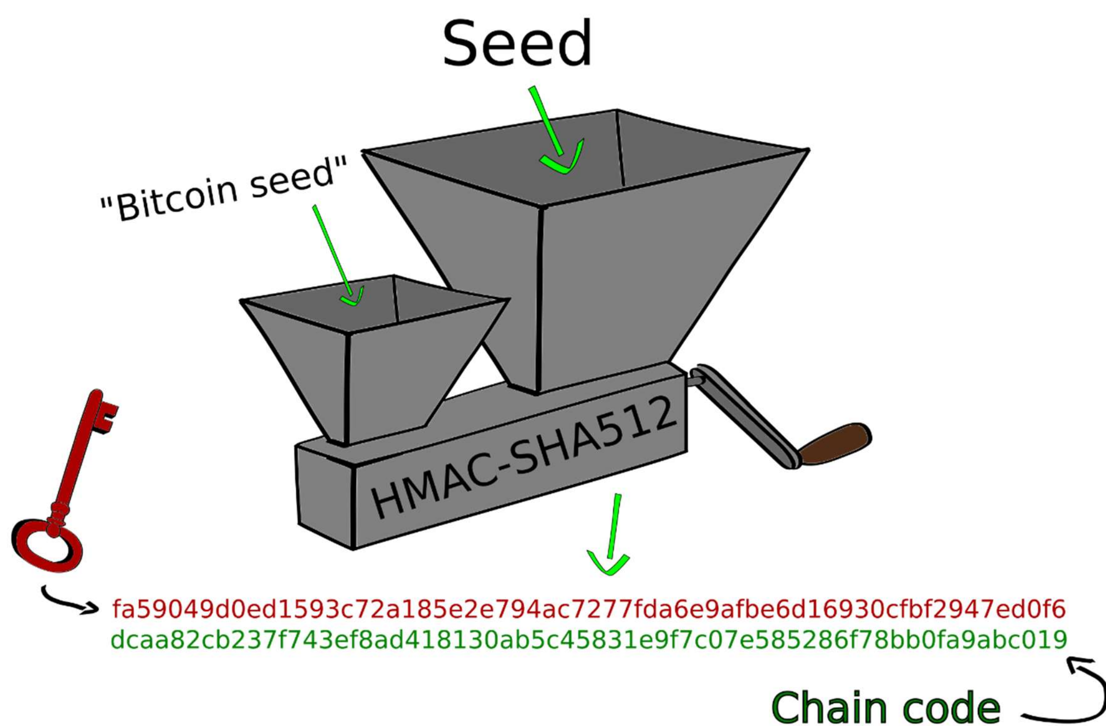


Ilustración 22

Para la creación de la clave maestra simplemente corremos la función HMAC-SHA512 con la semilla antes creada a partir de nuestra frase mnemotécnica y usando como Salt el Sting “*Bitcoin seed*”.

Esta nos dará como resultado el digest de 512 bits o 64 bytes, el cual dividiremos en dos; la primera parte (32 bytes) de este será nuestra clave privada, y la segunda mitad (los 32 bytes restantes) será nuestro Chain-code, el cual se usará para derivar las subsecuentes claves.

```
def master_key_extended(seed):

    expression = "Bitcoin seed" # Used for master key

    hmac_master = hmac.HMAC(expression.encode('ascii'), hashes.SHA512())
    hmac_master.update(seed)
    output = hmac_master.finalize()

    secret_num = output[:32]
    chain_code = output[32:]

    return ('m', secret_num, chain_code)

_, prv_master_key, chain_master_code = master_key_extended(seed)

private_key = ec.derive_private_key(int.from_bytes(prv_master_key, 'big'),
ec.SECP256K1())
public_key = private_key.public_key()
pub_master_key = public_key.public_bytes(Encoding.X962,
PublicFormat.CompressedPoint)

print(f"Chain code: {chain_master_code.hex()}")
print(f"Private master key (d): {prv_master_key.hex()}")
print(f"Public master key (compressed): {pub_master_key.hex()}")

del private_key, public_key
```

HD_wallet.ipynb

La llamada clave extendida no es más que la combinación de estos dos números (Key, Chain-code). Para hallar la correspondiente clave pública extendida, tendríamos que hacer uso de del método anteriormente visto, el cual multiplica sobre la curva elíptica el punto generador (G) por la clave privada (p); y añadir a esta el mismo Chain-code que su análoga clave privada.

A continuación, podemos apreciar el resultado de nuestra clave maestra extendida derivada de nuestra semilla mnemotécnica creada en los apartados anteriores

Chain code:
e57babbd57369704b4258a218c0d6158ea81bf4549b601baa85709b4688b6f4d

Private master key (d):
a8da07e2340e2d8055220121dde7b961c5d1d23319a4a369d97579cd7f43820b

Public master key (compressed):
03248a445f08350e1dadec79d1e6982118951ba65bc196619f138122d723fb9e05

Como veremos a continuación es esta cualidad de que tanto clave pública como privada compartan el mismo Chain-code la que nos ofrece algunas características muy interesantes.

7.2 CLAVE PRIVADA EXTENDIDA HIJA ORDINARIA

A diferencia de nuestra clave maestra las sucesivas claves serán creadas a partir de una clave extendida existente y no a partir de una semilla. Obviando este detalle el procedimiento de derivación es bastante similar.

Hay que recordar que estos procesos son deterministas, lo que quiere decir que para los mismos datos de entrada siempre obtendremos los mismos datos de salida, y esta cualidad es la que, si se dan las condiciones adecuadas, nos permitirá guardar o almacenar toda una cartera Bitcoin en una simple frase mnemónica.

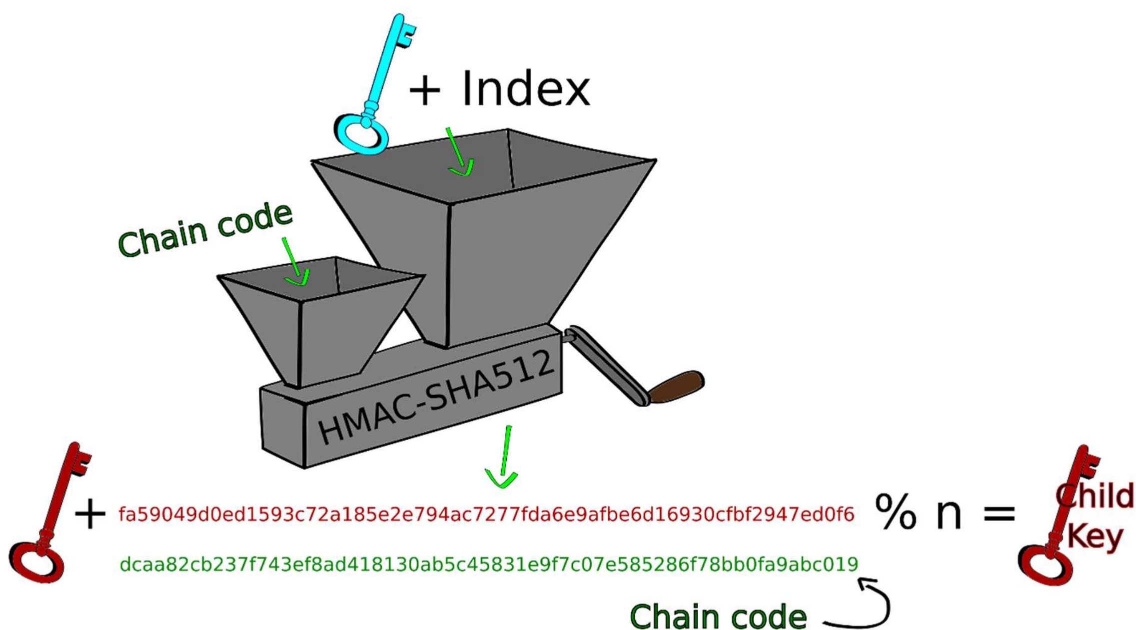


Ilustración 23

Antes de pasar a la implementación, me gustaría explicar el papel que tienen los índices a la hora de derivar claves extendidas. Un índice no es más que un numero de 32 bits o 4 bytes el cual nos indica cuantas claves hijas pueden ser derivadas de cada clave. Dos elevado a los 32 bits que podemos usar para el índice nos dan un rango de 4,294,967,296 el cual dividiremos en dos partes iguales.

La primara parte de 0 a 2,147,483,647 serán claves normales; y la segunda mitad comprendida entre 2,147,483,647 y 4,294,967,296 serán los llamados hijas curtidas (hardened childs).

```
def key_derivation(index, parent_key, parent_chain_code):

    index_bytes = index.to_bytes(32 // 8, 'big')
    data = parent_key + index_bytes
    salt = parent_chain_code
```

```
    hmac_512 = hmac.HMAC(salt, hashes.SHA512())
    hmac_512.update(data)
    output = hmac_512.finalize()

    return (output[:32], output[32:])
```

HD_wallet.ipynb

En este caso estamos derivando una clave privada; para ello necesitaremos tener tanto la clave padre pública como la privada, así como el Chain-code el cual recordemos que es el mismo para ambas.

El proceso es bastante simple; simplemente tendremos que correr el algoritmo HMAC-SHA512 al igual que hicimos a la hora de derivar la clave maestra, pero en este case usaremos como datos nuestra clave pública padre concatenada con nuestros 4 bytes al final, los cuales denotan el índice. Y como Salt usaremos nuestro Chain-code, el cual acompaña a nuestras claves padre extendidas.

De igual manera que vimos anteriormente esto dará un digest de 512 bits el cual dividiremos en dos; la primera mitad nuestra clave privada, y la segunda nuestro nuevo Chain-code. Esto introduce un nuevo problema puesto que si nuestra clave pública extendida padre se da a conocer sería tan simple como iterar por las 2,147,483,647 combinaciones de índices para obtener todos los posibles hijos de esta. Es por ello por lo que a nuestra nueva clave privada le sumaremos nuestra clave privada padre, para que de esta manera aquel que conozca nuestra clave pública extendida padre no pueda hacer nada sin averiguar antes nuestra clave privada padre.

Finalmente, hallemos el módulo de nuestra nueva clave pública para que este entre en el orden de la curva elíptica utilizada en nuestro encriptado asimétrico.

```
def private_normal_child(index, parent_prv_key, parent_chain_code):
    cv = Curve.get_curve('secp256k1')

    # Order of SECP256K1
    n = cv.order

    # Index range checking
    if index > 2147483647 | index < 0:
        raise f"Index: {index}, out of range for soft derivation."

    # Generating public key
    private_key = ec.derive_private_key(int.from_bytes(parent_prv_key,
'big'), ec.SECP256K1())
    public_key = private_key.public_key()
    parent_pub_key = public_key.public_bytes(Encoding.X962,
PublicFormat.CompressedPoint)

    # Derivation function
    child_prv_key, child_chain_code = key_derivation(index, parent_pub_key,
parent_chain_code)

    # Checking child_chain_code
    if int.from_bytes(child_chain_code, 'big') > n:
        raise f"Chain code bigger than curve order \"n\"; Try with index
{index + 1}"
```

```
    # Calculating child private key
    child_prv_key = ((int.from_bytes(child_prv_key, 'big') +
int.from_bytes(parent_prv_key, 'big')) % n).to_bytes(32, 'big')

    return (index, child_prv_key, child_chain_code)


index, key, chain_code = private_normal_child(0, prv_master_key,
chain_master_code)

private_key = ec.derive_private_key(int.from_bytes(key, 'big'),
ec.SECP256K1())
public_key = private_key.public_key()
public_compress_child = public_key.public_bytes(Encoding.X962,
PublicFormat.CompressedPoint)

print(f"Index: {index}")
print(f"Chain code: {chain_code.hex()}")
print(f"Private key (d): {key.hex()}")
print(f"Public key (compressed): {public_compress_child.hex()}")
```

HD_wallet.ipynb

A continuación, podemos ver el resultado obtenido al derivar el primer hijo normal de nuestra clave maestra. La derivación nos da como resultado la nueva clave privada y nuevo Chain-code, y como vimos anteriormente simplemente tendremos que multiplicar sobre la curva elíptica nuestro generador por la clave privada para hallar la clave pública, y al añadirle a esta nuestro Chain-code la haremos nuestra nueva clave pública extendida.

Index: 0
Chain code:
3575e4dcd7952e13f6dfa34aa29701ebf267c65bf65301a091da6698124948f8

Private key (d):
f5bad2e51b299946bbd7b611a9360ea49285b6606659e0b2154e03289e366cf3

Public key (compressed):
02fdf3c8ce073816f5a91452490ed309e39188d352e073aa3f6a8a2fc2a8e0cc11

Puesto que este es un algoritmo para derivar un hijo ordinario, el rango de nuestro índice ha de estar comprendido entre 0 y 2,147,483,647. Si el índice se encuentra fuera de este rango levantaríamos una excepción. Esto no quiere decir que no puedan derivar más hijos, pero puesto que el estándar está fijado de esta manera, modificar los rangos de los índices haría que nuestra cartera no fuese soportada en un gran número de aplicaciones.

7.3 CLAVE PRIVADA EXTENDIDA HIJA CURTIDA

En este caso vamos a derivar nuestro primer hardened child, en castellano hijo o hija curtida. Como podemos imaginar el procedimiento es muy sencillo, ya que este es prácticamente igual al superior. La

única diferencia reside en que para derivar una hija curtida usaremos como datos de entrada para HMAC-SHA512 nuestra clave privada acompañada del correspondiente índice en lugar de nuestra clave pública.

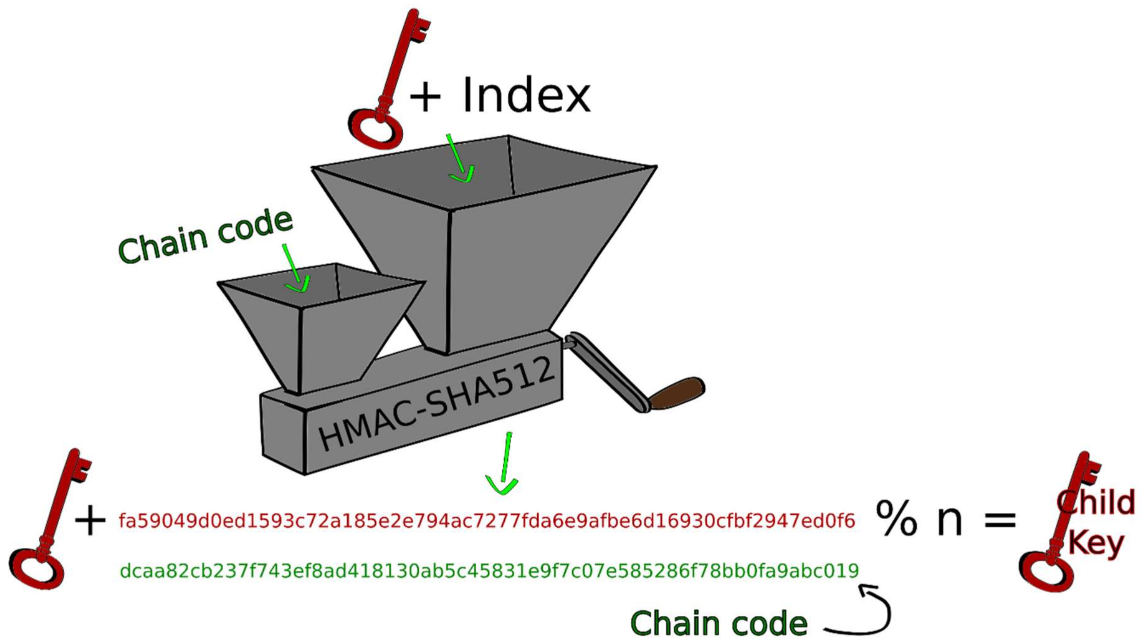


Ilustración 24

Hay que recordar que los índices comprendidos como curtidos son los que se encuentra entre el rango 2,147,483,647 y 4,294,967,296.

Una vez calculado el resultado del algoritmo HMAC, de igual manera que al calcular el hijo ordinario, sumaremos nuestra clave privada padre y haremos el módulo del orden de nuestra curva para hallar nuestra nueva clave. En este caso no sería necesario sumarle nuestra clave privada, puesto que los datos de entrada no van a ser difundidos en el dominio público, pero se hace esto para mantener la simetría con el otro algoritmo.

```
def private_hardened_child(index, parent_prv_key, parent_chain_code):
    cv = Curve.get_curve('secp256k1')

    # Order of SECP256K1
    n = cv.order

    # Index range checking
    if index > 2147483647 | index < 0:
        raise f"Index: {index}, Hard index: {index + 2147483648}, out of
range for hard derivation."

    # Generating public key --> No required

    # Derivation function
    child_prv_key, child_chain_code = key_derivation(
        index + 2147483648, # Hard index range
```

```
        b'\x00' + parent_prv_key, # Leading bit for public key compensation
        parent_chain_code
    )

    # Checking child_chain_code
    if int.from_bytes(child_chain_code, 'big') > n:
        raise f"Chain code bigger than curve order \"n\"; Try with index
{index + 1}"

    # Calculating child private key
    child_prv_key = ((int.from_bytes(child_prv_key, 'big') +
int.from_bytes(parent_prv_key, 'big')) % n).to_bytes(32, 'big')

    return (index, child_prv_key, child_chain_code)


index, key, chain_code = private_hardened_child(0, prv_master_key,
chain_master_code)

private_key = ec.derive_private_key(int.from_bytes(key, 'big'),
ec.SECP256K1())
public_key = private_key.public_key()
public_compress_child = public_key.public_bytes(Encoding.X962,
PublicFormat.CompressedPoint)

print(f"Index: {index}, Hard index: {index + 2147483648}")
print(f"Chain code: {chain_code.hex()}")
print(f"Private key (d): {key.hex()}")
print(f"Public key (compressed): {public_compress_child.hex()}")
```

HD_wallet.ipynb

A continuación, podemos ver la cual sería primer hijo curtido derivado de la clave maestra; como podemos ver, el campo (Index) hace referencia al índice dentro del rango reservado para claves curtidas, y (Hard index) hace referencia al índice que esta clave obtendría cuando englobamos los dos tipos de derivación posibles para un hijo.

Index: 0, Hard index: 2147483648

Chain code:
8474132cc761b149452dd9fafad6f671906dca61e939fcabf5e44a916cfa7f4e

Private key (d):
52da336b82908d8f282d1faa4bab0faddc7c15ed8134a26582fa2331eb897ee3

Public key (compressed):
03159256715cdc3501758cd5c32ff02a52540c44ae0ccbb5b523d22d8ddde69ef4

De igual manera que en el anterior procedimiento, la clave privada extendida sería nuestra nueva clave privada acompañada de Chain-code, y para averiguar nuestra clave pública extendida, simplemente hay que multiplicar el punto generador de la curva elegida por nuestra clave privada y añadir el mismo Chain-code a nuestra resultante clave pública.

7.4 CLAVE PÚBLICA EXTENDIDA HIJA ORDINARIA

En este ejemplo es donde podemos encontrar la magia de este procedimiento. Mediante la derivación de un hijo ordinario, la clave pública resultante podrá ser calculada por cualquier persona que tenga la clave pública padre en su poder.

Esto nos permite el poder derivar un gran número de claves públicas, con las que podremos firmar distintas transacciones incrementando así la seguridad de nuestros movimientos, ya que cada uso de estos estará dirigido a distintas direcciones, y por lo tanto después deberá de ser firmado por distintas claves públicas.

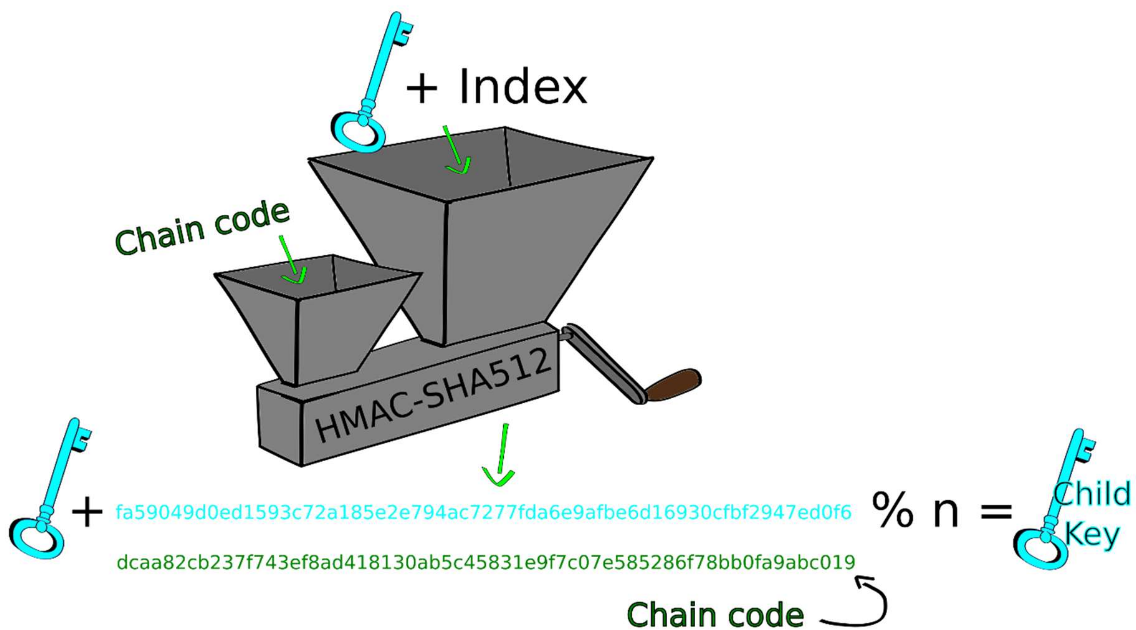


Ilustración 25

Además, nos ofrece otra importante cualidad, como es la de poder crear nuevas claves sin la necesidad de involucrar a nuestra clave principal; esto lo hace más seguro y nos permite externalizar la generación y compartición de claves a un tercero.

La generación de esta clave sigue una estructura muy similar a las anteriores con pequeñas variaciones, las cuales hacen posible el poder crear un hijo a partir del padre sin la necesidad de usar ningún dato privado como claves o semillas.

```
def public_normal_child(index, parent_pub_key, parent_chain_code):
    cv = Curve.get_curve('secp256k1')

    # Order of SECP256K1
    n = cv.order

    # Index range checking
    if index > 2147483647 | index < 0:
        raise f"Index: {index}, out of range for soft derivation."
```

```
    # Generating public key --> No required, we pass the pub key as parameter

    # Derivation function
    child_pub_key, child_chain_code = key_derivation(index, parent_pub_key,
parent_chain_code)

    # Checking child_chain_code
    if int.from_bytes(child_chain_code, 'big') > n:
        raise f"Chain code bigger than curve order \"n\"; Try with index
{index + 1}"

    # Calculating child private key
    parent_point = cv.decode_point(parent_pub_key)
    child_point = cv.mul_point(int.from_bytes(child_pub_key, 'big'),
cv.generator)
    result_point = parent_point + child_point

    # Encoding in weird format because ECPy doesn't support compress point
encoding
    child_pub_key = ec.EllipticCurvePublicKey.from_encoded_point(
        ec.SECP256K1(),
        bytes(cv.encode_point(result_point))).public_bytes(
            Encoding.X962,
            PublicFormat.CompressedPoint
        )

    return (index, child_pub_key, child_chain_code)


index, key, chain_code = public_normal_child(0, pub_master_key,
chain_master_code)

print(f"Index: {index}")
print(f"Chain code: {chain_code.hex()}")
print(f"Parent public key (compressed): {pub_master_key.hex()}")
print(f"Public key (compressed): {key.hex()}")
```

HD_wallet.ipynb

Partiremos de una clave pública extendida, que como ya hemos visto está compuesta de la clave pública, así como de un Chain-Code que es compartido entre las componentes pública y privada de nuestras claves asimétricas.

Como inputs de nuestro algoritmo HMSC-SHA512 usaremos la clave pública más el índice del hijo a derivar, recordando que los índices designados para hijos ordinarios son los comprendidos entre 0 y 2147483647, y como Salt usaremos el Chain-Code.

El digest resultante de 64 bytes será dividido en dos partes iguales de 32 bytes cada una. A la primera parte le sumaremos la clave pública padre y luego realizaremos el módulo (n) de esta para asegurarnos de que nuestra nueva clave se encuentre entre los posibles puntos de nuestra curva elíptica, y los 32 bytes resultantes de esta operación serán nuestro Chain-Code para este nuevo par de claves.

A continuación, podemos ver el que sería nuestro primero curtido hijo público de la clave maestra pública de la clave maestra.

Index: 0

Chain code:
3575e4dcd7952e13f6dfa34aa29701ebf267c65bf65301a091da6698124948f8

Parent public key (compressed):
03248a445f08350e1dadec79d1e6982118951ba65bc196619f138122d723fb9e05

Public key (compressed):
02fdf3c8ce073816f5a91452490ed309e39188d352e073aa3f6a8a2fc2a8e0cc11

Como podemos observar, el Chain-Code resúltante ha de ser igual al Chain-Code de la equivalente clave privada, la cual podrá desencriptar los mensajes encriptados por esta. Para corroborar esto podemos comparar nuestro Chain-Code con la anteriormente calculada clave hija ordinaria con índice 0, la cual obtuvimos en el apartado 6.2.

7.5 CLAVE PÚBLICA EXTENDIDA HIJA CURTIDA

Por último, al igual que existe un método para la creación de un hijo privado curtido, debería de existir un equivalente que nos permita derivar la clave pública correspondiente a esta clave privada. Y de cierta manera lo hay, el caso es que la derivación de este es muy distinta a las vistas anteriormente puesto que para ello necesitaríamos primero calcular nuestra clave privada curtida, y una vez obtenida esta calcular la clave pública por medio de matemáticas sobre la curva elíptica, como ya hemos experimentado.

```
def public_hardened_child(index, parent_pub_key, parent_chain_code):
    raise f"You need the private key to derive its peer public key."
```

HD_wallet.ipynb

Haciendo uso de estos mecanismos de derivación veremos cómo construir un árbol de derivación, el cual es el núcleo de una cartera determinista, que es de carteras más potentes y usadas en gran variedad de criptomonedas como por ejemplo Bitcoin.

8 ÁRBOL DE DERIVACIÓN

Un árbol de derivación es la pieza la principal en una cartera determinista. Este concepto fue introducido en la BIP 32 (Bitcoin Improvement Proposal) como una manera en la cual organizar nuestras claves en una cartera, y recibió posteriores modificaciones en BIP 43, 44, 49, 84, añadiendo características nuevas o modificando las ya existentes.

Este método surge para solucionar el problema de hacer uso de un nuevo par de claves para cada transacción; puesto que si se sigue esta práctica necesitaremos manejar un gran número de claves, y como ya hemos visto en caso de perder nuestra clave privada los fondos asociados a dicha clave quedan serán irrecuperables. Se podría decir que quedan en el limbo, puesto que si se vuelve a dar con la clave privada se recuperaría el acceso a los fondos vinculados a esta, tarea casi imposible si se intenta abordar probando todas las posibles combinaciones. Es por ello por lo que se ha de poner especial atención en la forma es la que estas claves son almacenadas.

En algunas de las implementaciones más populares de Bitcoin, como puede ser Bitcoin Core, la cartera integrada hace uso de bases de datos muy robustas y probadas como puedes ser SQLite o BerkleyDB, por completo, y no se implementan muchas de las tecnologías aquí explicadas como pueden ser frases mnemónicas, claves extendidas o árboles de derivación. Pero hoy en día estas son el estándar que nos brinda grandes beneficios, y es por ello por lo que cada vez más carteras y criptomonedas son desarrolladas bien para ser compatibles con estas, o con su propia versión de estas tecnologías.

Esta pieza del puzle es la última junto con frases mnemónicas y claves extendidas que necesitaremos para resolver el siguiente problema de una forma simple y elegante.

Imaginemos que queremos tener una cartera criptográfica almacenada en papel para mayor seguridad. Si esta es una cartera que tiene un historial de transacciones relativamente grande la tarea sería imposible de llevar a cabo, ya que necesitaríamos anotar en el papel cada una de las claves privadas usadas por esa cartera, que como sabemos son grandes números de difícil manejo, y esto daría pie a la aparición del error humano a la hora de copiar una gran cantidad de dígitos hexadecimales. Incluso haciendo uso de otras técnicas como WIF (Wallet Import Format), que es una manera de codificar las claves privadas de forma más funcinal, que veremos en este capítulo.

La solución, es guardar una semilla en lugar de toda la cartera con la ayuda de una frase mnemónica, y con las técnicas de derivación de claves extendidas seguir los árboles que mostraremos a continuación para replicar de forma determinista todas y cada uno de los pares de claves que forman nuestra cartera criptográfica.

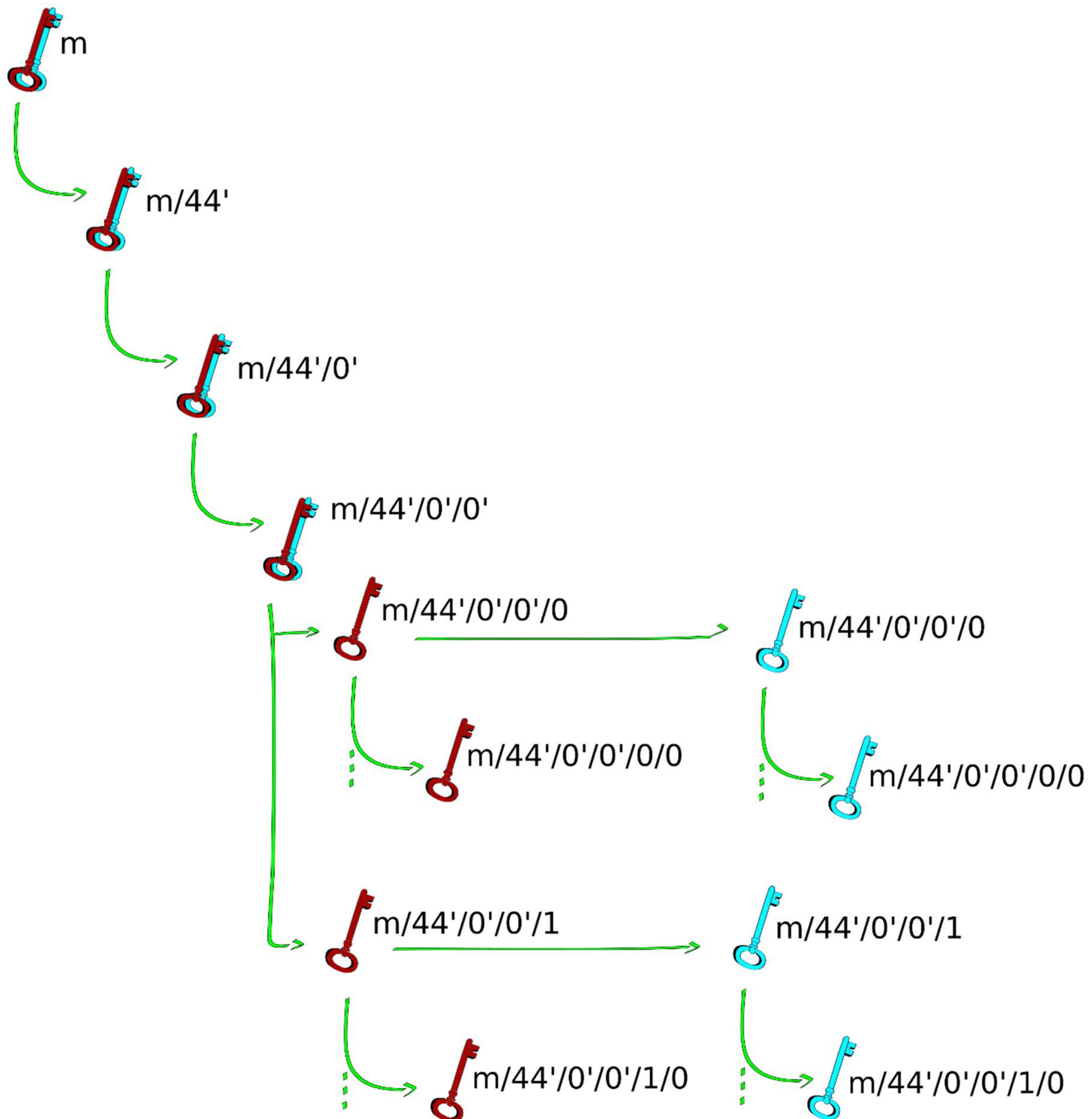


Ilustración 26

Para entender y poder implementar un árbol de derivación necesitamos 5 bloques esenciales los cuales ya hemos visto con anterioridad:

- Derivación clave extendida maestra
- Derivación clave privada extendida hija curtida
- Derivación clave privada extendida hija ordinaria
- Derivación clave pública extendida hija curtida

• Derivación clave pública extendida hija ordinaria

En la raíz del árbol empezamos siempre con la clave maestra, la cual como ya hemos visto puede ser obtenida de una entropía criptográficamente segura, que es codificada en una frase mnemónica y a su vez procesada en una semilla.

La “m” denota la clave maestra, “/” como delimitador de los distintos campos y cada uno de los campos es el índice de derivación de dicha clave. En caso de que dicho índice contenga una tilde la derivación será curtida y en caso contrario una derivación ordinaria. Siempre recordando que los índices para cada rango son los siguientes:

Ordinaria 0 hasta 2,147,483,647 incluido.

Curtida 2,147,483,648 hasta 4,294,967,295 incluido.

Por lo que el índice de derivación el árbol 0’ corresponderá al índice 2,147,483,648.

Cada uno de los niveles del esquema de derivación tiene su propio significado; en el caso de BIP32, el árbol tiene una profundidad de 4 niveles con el siguiente esquema:

1. El primer nivel siempre es el reservado para la clave extendida maestra.
2. El segundo nivel nos indica la cuenta a la que hacemos referencia. Bajo una misma cartera y moneda podemos manejar distintas cuentas empezando siempre por la primera denotada por 0.
3. En el tercer nivel encontraremos con un nivel un tanto especial ya que este es el encargado de dividir las claves en los grupos (External) y (Internal) las cuales hacen referencia a las criptomonedas que serán enviadas a nosotros por otros usuarios de Bitcoin (External, la cual adquiere el índice 0), y las criptomonedas enviadas por nosotros a nuestra propia cartera ya sea como cambio o para desfragmentación de esta (Internal, la cual adquiere el índice 1).
4. Y finalmente tenemos el nivel de cada uno de los pares de claves pública y privada que creemos para manejar nuestras criptomonedas.

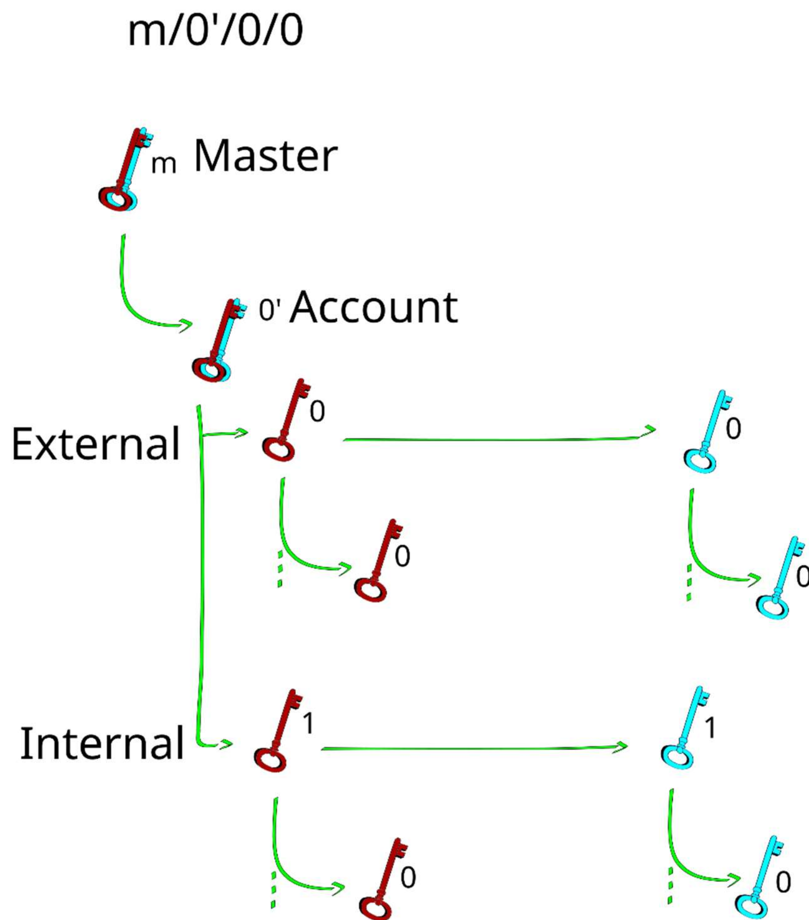


Ilustración 27

En la imagen superior podemos ver el aspecto de un árbol de derivación como definido por BIP32 (Bitcoin Improval Proposal), este consta de cuatro niveles con el siguiente formato:

m / cuenta' / cambio / índice

El sucesor que introdujo cambios considerables en el formato de árboles de derivación fue BIP43, donde se añade un nivel más dando lugar a los siguientes 5 niveles:

1. El primer nivel siempre es el reservado para la clave extendida maestra.
2. El nuevo nivel es denominado como (propósito) y obtiene el valor del BIP que se esté usando en dicha aplicación (32, 43, 44 ...) Las cuales crean modificaciones en las distintas rutas de derivación, así como en la forma de codificar las claves públicas para creación de cuentas y o direcciones.

Y el resto de los niveles a seguir serían los restantes que mostramos en anteriormente en referencia a BIP32; cuenta, cambio, índice.

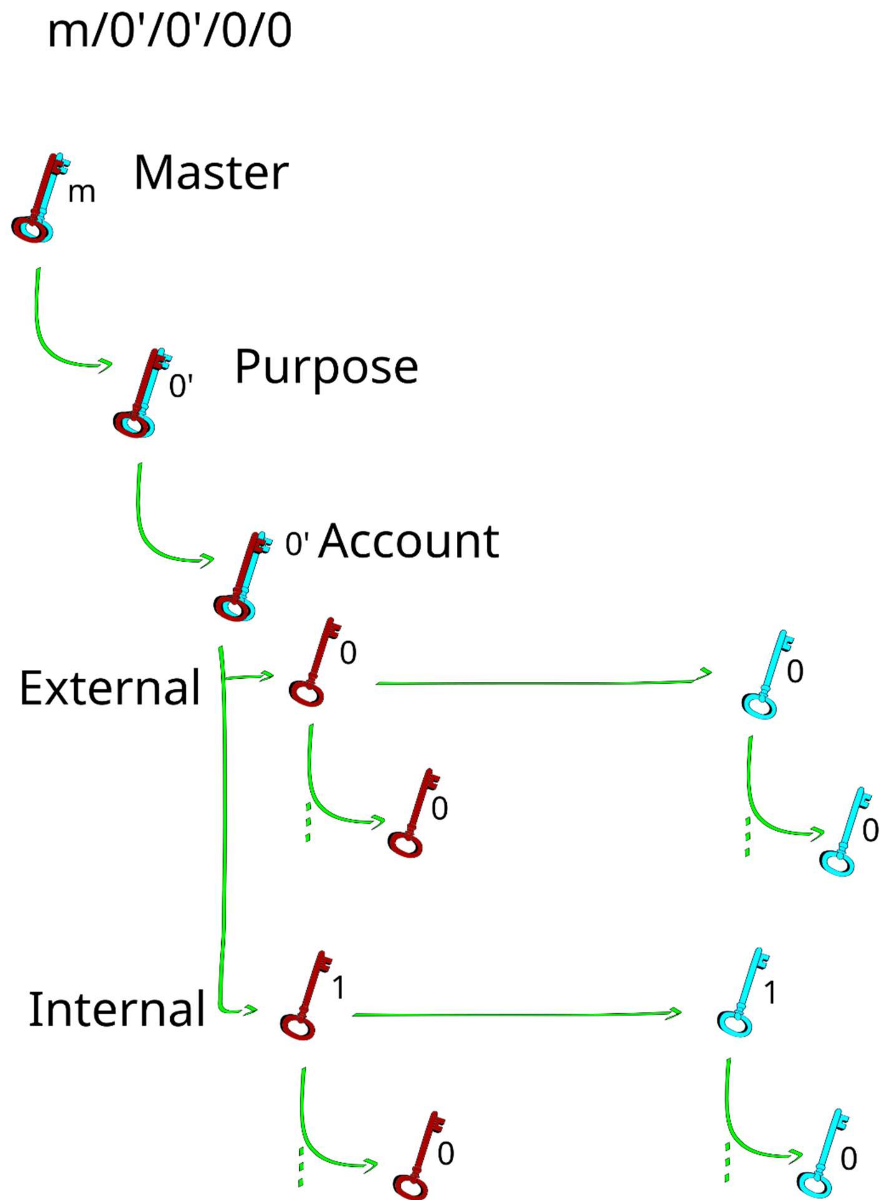


Ilustración 28

En la imagen superior podemos ver el aspecto de un árbol de derivación como definido por BIP43 (Bitcoin Improval Proposal), este consta de cinco niveles con el siguiente formato:

m / propósito' / cuenta' / cambio / índice

Finalmente mencionaremos la que puede ser la modificación o característica que más ha ayudado a la amplia adopción de este estándar. Esta es la introducida por BIP44 la cual añade un nivel más haciendo

así un árbol de derivación de profundidad 6, con los mismos niveles que los indicados en la BIP43, a excepción de un nuevo nivel comprendido entre el propósito y la cuenta.

Este nuevo nivel es denominado como (propósito) y obtiene el valor del BIP que se esté usando en dicha aplicación (32, 43, 44 ...) Las cuales crean modificaciones en las distintas rutas de derivación, así como en la forma de codificar las claves públicas para creación de cuentas y o direcciones.

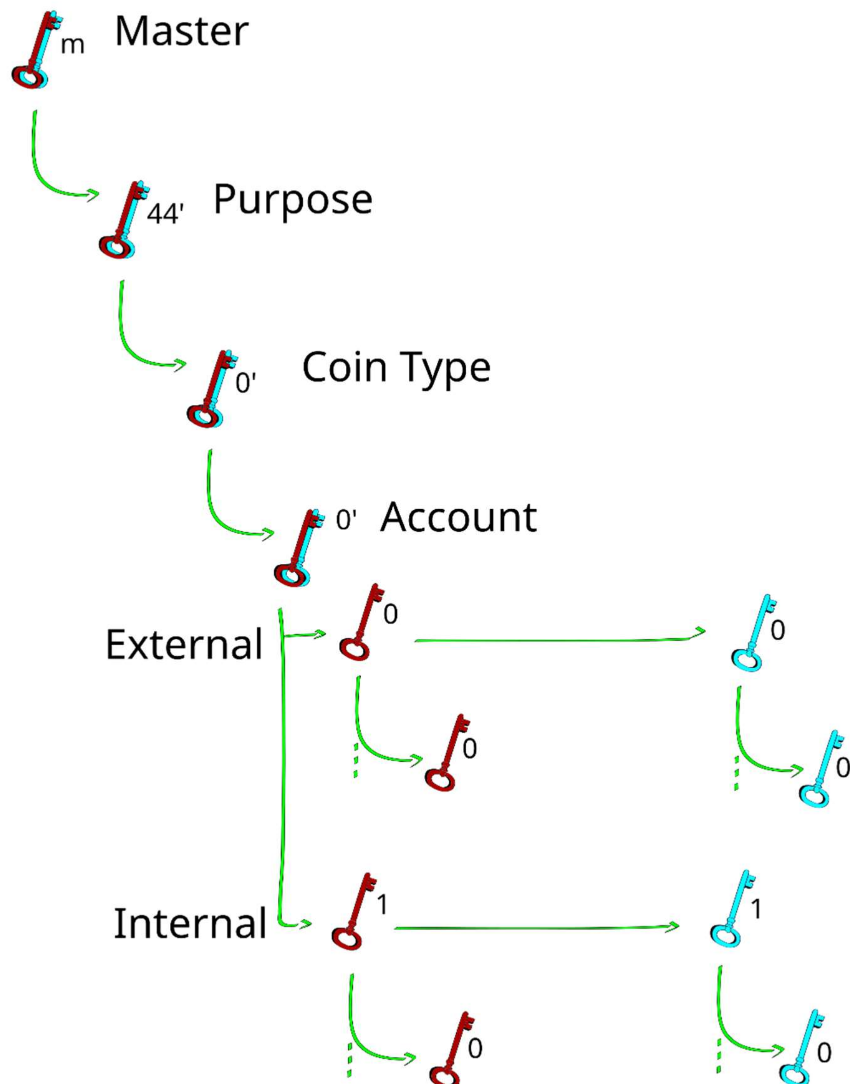


Ilustración 29

Gracias a este nivel incluido con la BIP44 no ya solo podemos almacenar toda nuestra cartera Bitcoin en una sola frase mnemónica, si no todas nuestras carteras criptográficas y criptomonedas o Blokchains hagan uso de este standard definido por el formato:

m / propósito' / moneda' / cuenta' / cambio / índice

He de destacar que el campo moneda también separa las Blockchain de prueba de las de producción, esto es importante puesto que en capítulos siguientes haremos uso de estas Blockchain de prueba. A continuación, algunos ejemplos de los valores que puede tomar el campo de este nivel:

Índice	Moneda	Blockchain
0	BTC	Bitcoin Mainet
1	BTC	Bitcoin Testnet
2	LTC	Litecoin
3	DOGE	Dogecoin
60	ETH	Ether
1815	ADA	Cardano

Podemos encontrar la lista completa de las distintas criptodivisas en el siguiente enlace:

https://github.com/satoshilabs/slips/blob/master/slip-0044.md

Además de estas BIP (Bitcoin Improval Proposal) hay muchas otras que, aunque no nos modifiquen el árbol de derivación, tendrán un impacto en el resultado de nuestra cartera y en el cómo esta interactuará con la red Bitcoin. Como por ejemplo BIP49 sigue el mismo esquema que su predecesor BIP44 con la característica de que las direcciones que generamos en este caso son direcciones P2WPKH (Pay to Witness Public Key Hash) y P2SH (Pay to Script Hash) que se verán en futuros capítulos; o BIP84 que es otra versión la cual simplemente cambia el tipo de direcciones creadas a P2WPKH (Pay to Witness Public Key Hash), las cuales son compatibles con la tecnología SegWit (Segregated Witness).

Hasta el momento los árboles de derivación que hemos visto hacen uso de P2PK (Pay to Public Key), el cual fue remplazado por el más seguro P2PKH (Pay to Public Key Hash), que también veremos en breve.

```
def derive_path(path, seed):
    path_list = path.split('/')
    index, key, chain_code = None, None, None
    for term in path_list:
        if term == 'm':
            # Master extended key from seed
            _, key, chain_code = master_key_extended(seed)
            continue
        if "'" in term:
            # Case term is hard
            _, key, chain_code = private_hardened_child(int(term.strip("'")),
key, chain_code)
        else:
            # Case the term is soft of normal
            _, key, chain_code = private_normal_child(int(term), key,
chain_code)
    return key, chain_code
```

```
derivation_path = "m/44'/0'/0'/0/0" # Path extipulated on BIP 44

path_key, _ = derive_path(derivation_path, seed)

print(f"Private key on {derivation_path}:\n\t{path_key.hex()}")
```

HD_wallet.ipynb

En el siguiente ejemplo podremos encontrar la que sería la primera clave privada de nuestros fondos recibidos de cuentas externas de nuestra primera cuenta de la criptomoneda Bitcoin de tipo BIP44.

Private key on m/44'/0'/0'/0/0:
b333fbdc1e42198b24e08710010ddfd65ee7111220b491335dde61a424ad6b73

8.1 Test de derivación aislada

En este apartado demostraremos una de las grandes ventajas que nos brindan este tipo de carteras, como es la de poder derivar de manera determinista las claves públicas hijas a una clave pública dada, y que estas estén ligadas a sus claves privadas particulares las cuales serán derivadas de forma completamente aislada.

Esto no es de gran utilidad en el caso de que queramos hacer transferencias de una criptomoneda de manera reiterada en el tiempo, por ejemplo, digamos que queremos hacer una transferencia a la semana.

Siguiendo las buenas prácticas de Bitcoin, estas transferencias han de ser realizadas con un nuevo par de claves para cada una de ellas, pero esto implica el tener que generar y transmitir nuestra clave pública una vez a la semana, lo cual no es muy práctico y puede introducir gran número de inconvenientes.

La solución es la de acordar una rama del árbol de derivación la cual será utilizada con este propósito, y facilitar a la persona que realiza las transferencias la clave pública extendida raíz de esta rama, de tal modo que pueda derivar subsecuentes hijos con la ayuda de derivación de clave pública extendida ordinaria.

```
 # Finding the master pub, prv and chain-code
prv_secret, chain_code_m = derive_path("m", seed)
prv_key_m = ec.derive_private_key(int.from_bytes(prv_secret, 'big'),
ec.SECP256K1())
pub_key_m = prv_key_m.public_key().public_bytes(Encoding.X962,
PublicFormat.CompressedPoint)

# Derive our path for our private key
prv_secret, _ = derive_path("m/444/8", seed) # paths must be the same
prv_key = ec.derive_private_key(int.from_bytes(prv_secret, 'big'),
ec.SECP256K1())
```

```
# We derive the same path for our public key
_, pub_key, chain_code_child = public_normal_child(444, pub_key_m,
chain_code_m) # 444/
_, pub_key, chain_code_child = public_normal_child(8, pub_key,
chain_code_child) # 8/
pub_key = ec.EllipticCurvePublicKey.from_encoded_point(ec.SECP256K1(),
pub_key)

# We sign a message with our derived private key
message = b"Testing derivaiton of extended keys."
signature = prv_key.sign(message, ec.ECDSA(hashes.SHA256()))

# If no exception raised then the signature is correct
pub_key.verify(signature, message, ec.ECDSA(hashes.SHA256()))
```
HD_wallet.ipynb

En el ejemplo de arriba primero creamos una nueva semilla y posteriormente una clave maestra para nuestro árbol de derivación. Derivamos la clave pública que usaremos para este ejemplo, la cual se encuentra bajo la rama (m/444/8) y consecuentemente haremos lo mismo, pero de manera totalmente independiente para hallar la clave pública a partir de la clave pública maestra.

Finalmente firmaremos un mensaje con esta, y si nuestra clave pública es capaz de verificarlo, podremos estar seguros de que nuestra implementación de árboles de derivación funciona correctamente.

En caso de que la firma sea validada correctamente, no habrá ningún mensaje de salida corroborándolo; solo en caso contrario se levantará un error.

9 CODIFICACIÓN Y SERIALIZACIÓN DE CLAVES

Hasta este momento hemos visto como guardar un número aleatorio criptográficamente seguro en una frase mnemónica, como pasar está a una semilla, y usar esta semilla para crear y posteriormente derivar nuestras claves mediante un árbol de derivación. Ahora necesitamos una manera de guardar, almacenar y transmitir estas claves, árboles, ramas o caminos de derivación.

Para ello haremos uso de dos métodos, uno para trabajar con claves extendidas dentro de un árbol de derivación, y otro que nos permita codificar nuestras claves privadas en un estándar reconocible por otras carteras Bitcoin.

9.1 BASE 58

Antes de meternos de lleno en la codificación y serialización de claves me gustaría explicar brevemente la codificación base 58, puesto que esta es muy usada en Bitcoin.

Base 58 no es más que una forma de codificar la información, como ya hemos visto Bitcoin hace uso de grandes números. Ya sean para la semilla de nuestra cartera, claves o como veremos más adelante para ciertos rasgos de nuestras transacciones. Es aquí donde base 58 nos es de gran ayuda, ya que comprime los números y disminuye el error humano a la hora de trabajar con estos.

BASE 10	23452632384750928374590823529034875092375092750923875092750923
BASE 16	0e98384b04ba4789f00ecf7f4bde3748374e8496847bc493624b
BASE 58	SvNr9HoNwBxekQBdNBeVWwCpMjByX3BnSQi

¿Cómo hace esto?, de la misma manera que al pasar un número de base 10 o decimal a base 16 o hexadecimal, la cantidad de caracteres para representarlo disminuye, base 58 consigue el resultado, pero con una compresión mucho mayor al hacer uso de tanto caracteres de números como letras. Ahora bien, ¿por qué 58?

0 1 2 3 4 5 6 7 8 9
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z

Base 58 usa los caracteres de números y letras del alfabeto inglés, tanto mayúsculas como minúsculas a excepción de las cuatro resaltadas en rojo. Esto hace más fácil el leer e introducir nuestros números y que estos cuatros caracteres son muy semejantes entre sí, y en ciertos estilos caligráficos se pueden llegar a confundir.

9.2 SERIALIZACIÓN Y CODIFICACIÓN CLAVES EXTENDIDAS

Como hemos visto, el proceso de creación de una clave extendida contiene más información que la simple clave; esta información es muy importante puesto que sin ella muchas de las ventajas y posibilidades que nos brindan las carteras deterministas quedarían invalidadas. Es por ello por lo que necesitamos una forma estandarizada, en la que podamos incluir toda esta información, bien sea para su almacenamiento o difusión.

Los datos que incluidos en esta serialización serán los indicados abajo en ese mismo orden:

1. Versión
2. Profundidad
3. Huella clave padre
4. Índice
5. Chain-code
6. Clave

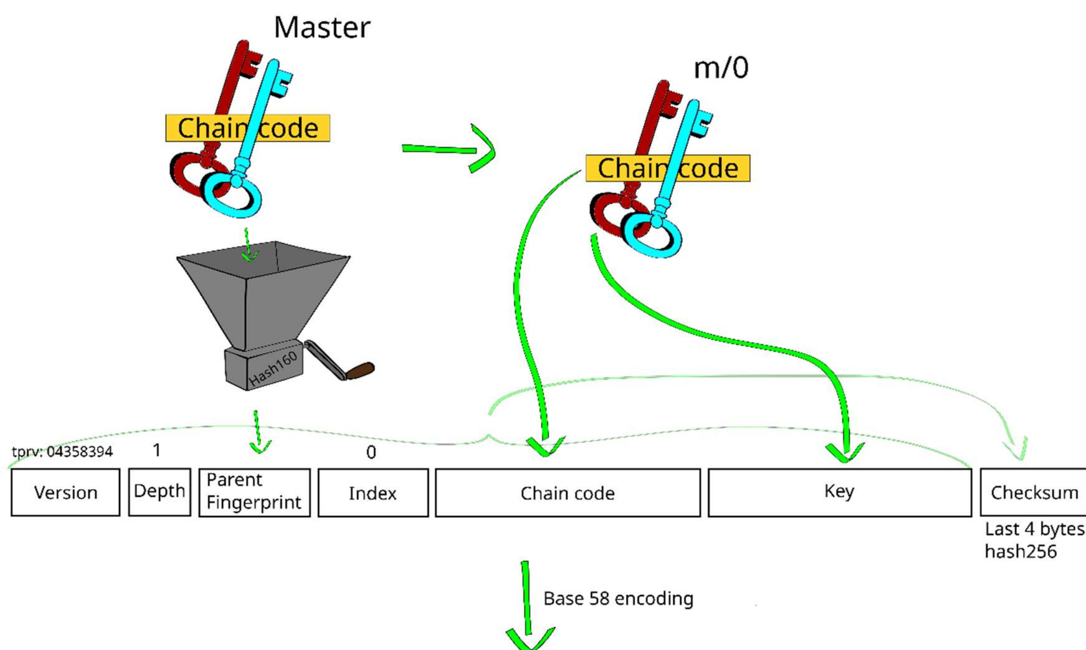


Ilustración 30

Para serializar nuestra clave extendida primero tendremos que codificar los datos anteriormente mencionados junto con el checksum de todos estos para el cual añadiremos para poder verificar la integridad de estos.

Longitud	Nombre	Descripción
4 bytes	Versión	Tipo o propósito de clave.
1 byte	Profundidad	Profundidad de la clave con respecto a la clave maestra.
4 bytes	Huella clave padre	Primeros 4 bytes del hash160 de la clave padre.
4 bytes	Índice	Índice de derivación de la clave.
32 bytes	Chain-Code	Chain-Code asociado con dicha clave.
33 bytes	Clave	Clave privada o pública.
4 bytes	Checksum	Últimos 4 bytes hash256 del resto de los datos.

Versión: estos cuatro primeros bytes nos indican la procedencia de la clave para que en la posterior importación o deserialización de la clave podremos encontrar el correspondiente árbol de derivación, así como el nivel en la que esta se encuentra. A continuación, podemos observar algunos de los bytes que identifican tanto claves privadas como públicas para distintos arboles de derivación.

Hex-code	Blockchain	Clave	Base58	BIP
0x0488ade4	Mainet	Privada	xprv	BIP44
0x0488b21e	Mainet	Pública	xpub	BIP44
0x049d7878	Mainet	Privada	xprv	BIP49
0x049d7cb2	Mainet	Pública	xpub	BIP49
0x04b2430c	Mainet	Privada	xprv	BIP84
0x04b24746	Mainet	Pública	xpub	BIP84
0x04358394	Testnet	Privada	xprv	BIP44
0x043587cf	Testnet	Pública	xpub	BIP44
0x044a4e28	Testnet	Privada	xprv	BIP49
0x044a5262	Testnet	Pública	xpub	BIP49
0x045f18bc	Testnet	Privada	xprv	BIP84
0x045f1cf6	Testnet	Pública	xpub	BIP84

Profundidad: Valor de un único byte que nos indica el nivel de profundidad en el cual dicha clave se encuentra en nuestro árbol de derivación.

Huella clave padre: Los primeros cuatro bytes obtenidos al computar el hash160 de la clave padre a la serializada.

Índice: Cuatro bytes que contienen el número que denota el hijo del padre.

Chain-Code: Código característico de las claves extendidas (compartido entre pareja de clave pública y clave privada).

Clave: Nuestra clave; esta ha de corresponder con lo que podríamos denominar su meta-data, que son todos los valores anteriores.

Checksum: Últimos cuatro bytes del hash256 de los datos anteriores, los cuales nos ayudaran con la consistencia e identificación de errores en el futuro.

```
def extended_serialization(depth, parent_key, child_number, chain_code, key,
version = None, testnet = True):

    # Version BIP 44
    if version == None and testnet == False:
        serialized_key = b"\x04\x88\xad\xe4" if len(key) == 32 else
b"\x04\x88\xb2\x1e"
    elif version == None and testnet == True:
        serialized_key = b"\x04\x35\x83\x94" if len(key) == 32 else
b"\x04\x35\x87\xcf"
    else:
        serialized_key = verison
```

```
    # Depth
    serialized_key += depth.to_bytes(1, 'big')

    # Fingerpirnt of the pub father key
    hash1 = hashlib.sha256(parent_key)
    hash2 = hashlib.new('rmd160')
    hash2.update(hash1.digest())
    fingerprint = hash2.digest()[:4]

    # Child index
    index = child_number.to_bytes(4, 'big')

    serialized_key += fingerprint + index + chain_code

    # If pvt key prepend 0x00
    if len(key) == 32:
        serialized_key += b"\x00"

    serialized_key += key

    # Checksum
    hash1 = hashlib.sha256(serialized_key)
    hash2 = hashlib.sha256(hash1.digest())

    serialized_key += hash2.digest()[:4]

    return base58.b58encode(serialized_key)

# Derivaiton paths
derivation_path = "m/0" #44'/0'/0'/0/0
parent_derivation_path = "/".join(derivation_path.split('/')[:-1])

# Generating necesary keys
prv_key, chain_code = derive_path(derivation_path, seed)
parent_prv_key, _ = derive_path(parent_derivation_path, seed)
parent_prv_key = ec.derive_private_key(int.from_bytes(parent_prv_key, 'big'),
ec.SECP256K1())
parent_pub_key = parent_prv_key.public_key().public_bytes(Encoding.X962,
PublicFormat.CompressedPoint)

# Getting index and depth
derivation_count = len(derivation_path.split('/'))-1
child_n = int(derivation_path.split('/')[-1].strip("'"))

# Serialization
serialized_key = extended_serialization(derivation_count, parent_pub_key,
child_n, chain_code, prv_key)

# Printing
# Code for printing was ommited of this sniped in order to make it cleaner
```

HD_wallet.ipynb

En el código anterior se define una función para codificar y serializar una clave extendida, y a continuación se deriva una clave nueva y se serializa con esta. La parte del código que imprime la salida ha sido omitida para mantener la memoria más limpia, pero esta se puede encontrar en el Playbook correspondiente.

Parent derivation path: m
Parent public key: 03248a445f08350e1dadec79d1e6982118951ba65bc196619f138122d723fb9e05

Derivation path: m/0
Key: f5bad2e51b299946bbd7b611a9360ea49285b6606659e0b2154e03289e366cf3
Chain code: 3575e4dcd7952e13f6dfa34aa29701ebf267c65bf65301a091da6698124948f8

0435839401cbdba7ad000000003575e4dcd7952e13f6dfa34aa29701ebf267c65bf65301a091da66981249
48f800f5bad2e51b299946bbd7b611a9360ea49285b6606659e0b2154e03289e366cf3899aadbf

- Version
- Depth
- Fingerprint
- Child index
- Chain code
- Key
- Checksum

Finalmente haremos uso de la codificación Base 58 la cual hace la clave más amigable para las personas gracias al acortar su longitud, así como omitir ciertos caracteres similares los cuales pueden dar lugar a confusión.

tprv8d4zoWN8RJCtBzFPjujLoh9XSirGw5yC2VyHfZi9AEgFaYbyHTtikuvdt2aijK3vUbfESC9soM4AQruKuyrm
op1bAwxFmtJFPaumEBRakCE

9.3 WIF (Wallet Import Format)

Una clave WIF de (Wallet Import Format) es una manera simple de trabajar con claves privadas. Como hemos visto, teniendo una única clave privada podremos derivar de esta un gran número de claves de manera determinista siempre que sigamos una misma especificación. Es por esto por lo que este método obtiene su nombre, puesto que importando la clave raíz de nuestro árbol tendremos acceso a toda la cartera.

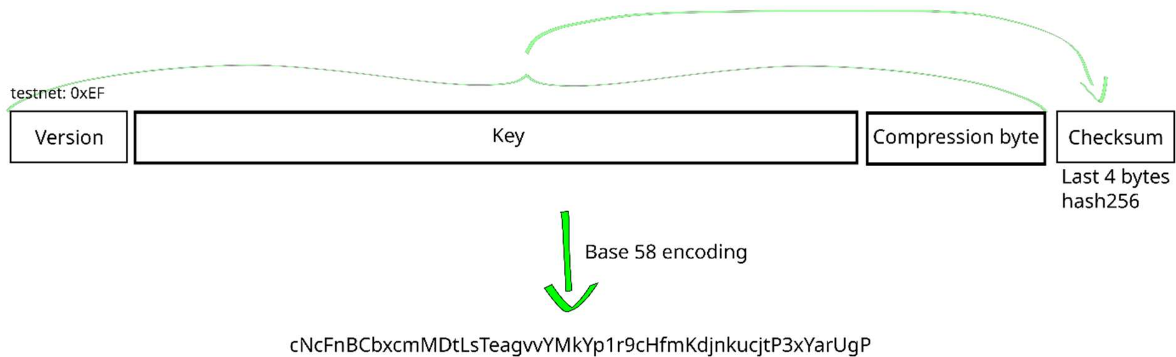


Ilustración 31

El procedimiento para construir una de estas claves es bastante simple puesto que no requiere de la serialización de muchos campos.

Lo primero será la versión, al igual que en el método posterior encontraremos un campo de versión al comienzo de este, el cual poblaremos con los bytes:

0x80	Mainet
0xEF	Testnet

A continuación de este capo incluiremos nuestra clave privada, y para indicar si esta es usada para derivar claves públicas comprimidas o no, haremos uso del siguiente campo, incluyendo el byte 0x01 en caso de que las claves públicas resultantes sean comprimidas, u omitiremos este campo en caso contrario. Y al igual que en el método anterior computaremos un checksum de los campos anteriores para facilitar las labores de detección de errores.

```
def wif(key, compressed = True, testnet = True):

  # Network
  network = b"\xef" if testnet else b"\x80"

  # Network + key + compress byte
  result = network + key + (b"\x01" if compressed else b"")

  # Checksum
  hash1 = hashlib.sha256(result)
  hash2 = hashlib.sha256(hash1.digest())
  result += hash2.digest()[0:4]
```

```
  return base58.b58encode(result)

wif_key = wif(prv_master_key)

# Printing
# Code for printing was ommited of this sniped in order to make it cleaner
```
HD_wallet.ipynb

efa8da07e2340e2d8055220121dde7b961c5d1d23319a4a369d97579cd7f43820b0134cc404c

- Network
- Key
- Compress byte
- Checksum

Simplemente nos quedara codificar nuestra clave WIF con ayuda de Base 58, que como ya mencionamos anteriormente ayuda y facilita el trabajo con estos grandes números.

cTEvjWiBoBR5Lymro3sTAb3b9DunrgSd57LcnFdhtBz2FzU9JC35

10 NUESTRA CARTERA HD

En este capítulo crearemos unas cuantas direcciones de nuestra cartera digital, enviaremos fondos a esta mediante el uso de una faucet Bitcoin, y posteriormente la importaremos en una aplicación compatible con los estándares que hemos estado haciendo uso. Pero antes de esto tendremos que ver cómo crear nuestras direcciones Bitcoin; estas direcciones serán del tipo P2PKH (Pay to Public Key Hash), lo cual viene a significar que a la hora de hacer una transacción los fondos o valor transferido será al hash de nuestra clave y no a nuestra clave en sí. Ya veremos con más detalle cómo funciona esto sucesivos apartados.

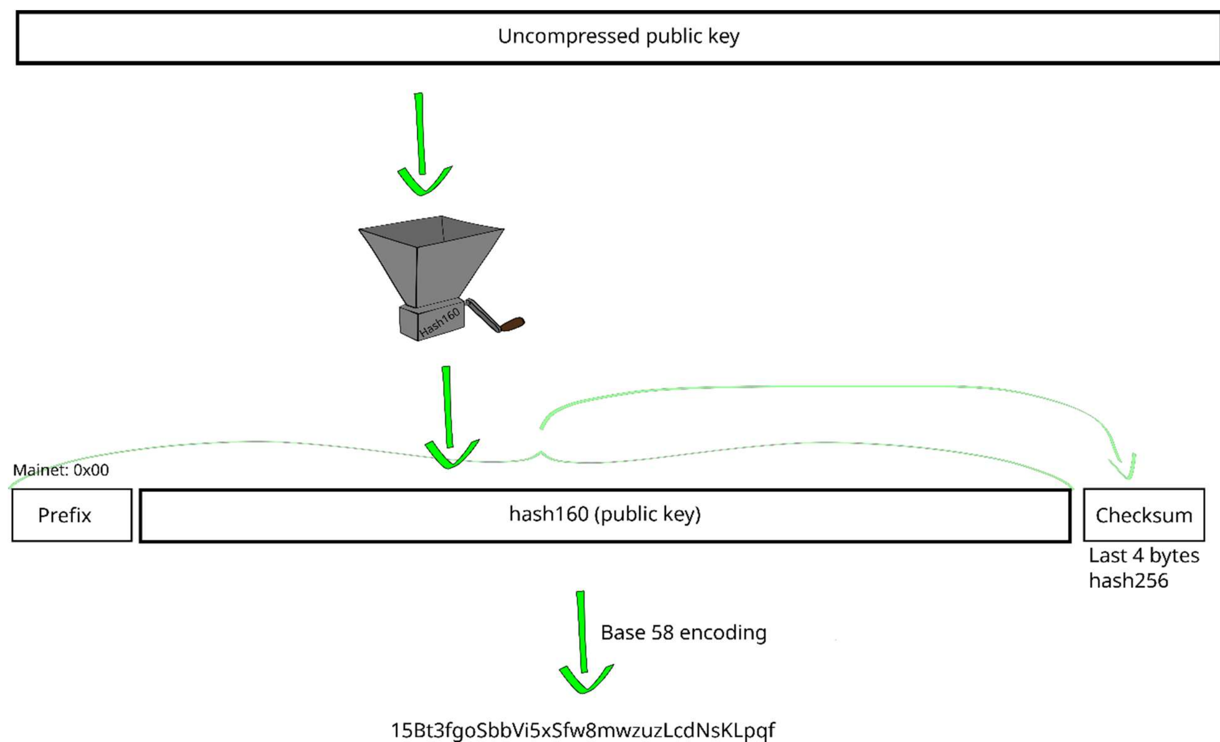


Ilustración 32

Para su creación empezaremos por un byte que denote la blockchain en la cual esta dirección será válida:

0x00	Mainet
0x6F	Testnet

Seguido del hash160 de la clave pública. Cabe decir que la clave ha de ser sin comprimir, reflejando las dos coordenadas del punto. Si cometemos un error al crear la clave de que no pueda ser detectado por el checksum que sigue continuación la transacción con destino esta dirección será ejecutada pero los fondos serán perdidos, puesto que no podremos desbloquearlos posteriormente, por lo que hemos de tener cuidado en este paso.

```
def P2PKH(pub_key, testnet = True):
  # Network
  network = b"\x6f" if testnet else b"\x00"

  # hash160
  hash1 = hashlib.sha256(pub_key)
  hash2 = hashlib.new('rmd160')
  hash2.update(hash1.digest())
  result = network + hash2.digest()

  # Checksum
  hash1 = hashlib.sha256(result)
  hash2 = hashlib.sha256(hash1.digest())
  result += hash2.digest()[:4]

  return base58.b58encode(result)

address = P2PKH(pub_master_key, False)

# Printing
# Code for printing was ommited of this sniped in order to make it cleaner
```
HD_wallet.ipynb

00cbdba7ad324c5045a35e32fbe2687e6d0aca73568f65313d

- Version
- hash160(pub_key)
- Checksum

Finalmente, como ya está siendo costumbre, codificaremos el Sting resultante en Base 58 para evitar errores y hacerlo así más amigable.

1KauJqagKnTMbtaMDtZZ5iGPKFi6nVozVJ

Finalmente llegamos a nuestro objetivo: la creación de una cartera determinista. En este apartado pondremos juntos todas las piezas que hemos ido examinando anteriormente para crear nuestra cartera, y posteriormente ingresaremos fondos en ella e importaremos en una aplicación de carteras Bitcoin deterministas, para ver como esta es reconstruida de la misma manera por dos códigos distintos.

```
if not 'mnemonic_sentence' in vars():
  entropy = entropy_gen(256)
  mnemonic_sentence = " ".join(mnemonic(entropy))
```

```
print("\n\u250F"+"\u2501"*(len(mnemonic_sentence)+2)+"\u2513")
print("\u2503 "+mnemonic_sentence+" \u2503")
print("\u2517"+"\u2501"*(len(mnemonic_sentence)+2)+"\u251B")

seed = mnemonic2seed(mnemonic_sentence)

def encoded_key_pairs(prv_key, chain_code, testnet = True):
  # WIF for private key
  wif_key = wif(prv_key, True, testnet)

  # Getting public key
  prv_key = ec.derive_private_key(int.from_bytes(prv_key, 'big'),
ec.SECP256K1())
  pub_key = prv_key.public_key().public_bytes(Encoding.X962,
PublicFormat.CompressedPoint)

  # Address from public key
  address = P2PKH(pub_key, testnet)

  return (wif_key, address)

derivation_path = "m/44'/1'/0'/0/0" # Path testnet BIP 44

recieve_addresses = []

if True:
  print("{:18}{:38}{:55}".format('Path'.center(18),'Address'.center(38),'WIF'
.center(55)))
  print(colors['gray']+"-"*111, end='\033[0m\n')

  # Ordinary
  for i in range(10): # Wallets search on the first 20 addresses
    derivation = derivation_path.split('/')
    derivation[-1] = str(int(derivation_path[-1]) + i)

    extended_key = derive_path("/".join(derivation), seed)
    key_pair = encoded_key_pairs(*extended_key, True)

    print("{:>18}".format("/".join(derivation)), end="")
    print("{}{:>38}".format(colors['blue'], key_pair[1].decode('ascii')),
end="")
    print("{}{:>55}".format(colors['red'], key_pair[0].decode('ascii')),
end="\033[0m\n")
    recieve_addresses.append(key_pair[1].decode('ascii'))

  # Change
  for i in range(5):
    derivation = derivation_path.split('/')
```

```
    derivation[-2] = '1'
    derivation[-1] = str(int(derivation_path[-1]) + i)

    extended_key = derive_path("/".join(derivation), seed)
    key_pair = encoded_key_pairs(*extended_key, True)

    print("{:>18}".format("/".join(derivation)), end="")
    print("{}{:>38}".format(colors['yellow'], key_pair[1].decode('ascii')),
end="")
    print("{}{:>55}".format(colors['red'], key_pair[0].decode('ascii')),
end="\033[0m\n")
```

HD_wallet.ipynb

El código superior genera un Sting de bytes completamente aleatorio, el cual es convertido en una frase mnemotécnica. Esta es impresa por pantalla, así como las primeras 10 direcciones de nuestra cartera y las primeras 5 direcciones para recibir cambio. A continuación, podremos ver nuestra frase mnemotécnica, así como una lista de las direcciones, las cuales separamos entre azules para direcciones normales y amarillas para el cambio. Justo debajo de cada una de nuestras direcciones podremos apreciar en rojo la clave privada en formato WIF para dicha dirección.

fetch shine budget scare there food bless north fog swing coil sure social lottery together will height suspect problem circle eye bean art noise

Path
P2PKH
WIF

---

m/44'/1'/0'/0/0
n194pq5LLXpDRzBZssuQayP9A7pfSfAjbF
cPnztkJGDnweFad2TXjqXdM1r7EBFNQUQ1XoiEfsy8mfED5kaQib

m/44'/1'/0'/0/1
mkYTxuQmgrfDQ9WzVoN43A1y4PR2zPw1ec
cPrSqshyySHqtF4ZE8ttk4TW7w6ck88jTQfeU415ngTPq9RWrSPs

m/44'/1'/0'/0/2
miYWGe3a8GQaMTdTFf8HLmjHEQGq75CozJ
cRHXZkGZeevh8SbCXYLHcRwsGx1FmuKBLMe8h3LT4M9oYUUiDKqg

m/44'/1'/0'/0/3
mjqcz6yV8UGP5mZnxqDY9BRcvAUAY2AS4T
cScqWqFzrWhiK3ndTLpKFPuE5kCzi7degAWeqTEJaqWZTCopbAoV

m/44'/1'/0'/0/4

mrCA9AoGpUNbjue8nW1vupeZKRzvGfk4TX
cQmCKs93e6x9MFTEVeyRzBjKsdjKZHLT16zHgaJFmzFhqiaSbYT2

m/44'/1'/0'/0/5
n38RBjtXLv1kx8FfBLcgbAdPz5wtvtUvaL
cUeg9WsyLEtThJVqQBBS2Q6dedp5Y4WfQXn2WUMt7FbSZj3PXZUJ

m/44'/1'/0'/0/6
miEHYUEEUMfhkpBY5DLNtaL2yNLApFpp4x
cPumoRdWcKncpmh7vi1Bh4JWdYn42JZeBiBXQhF5edvU8RXvZPfB

m/44'/1'/0'/0/7
n3Vt8vd6DX5MHdcD8JQFsPaPtbC3TGptU1
cR1oM445Bkx2LaoMNhw2PaVJBEzDcE8xj5jWpHf5sRYBunRuvG5u

m/44'/1'/0'/0/8
mu3xJnYC53FimrPhhHWvoKVmrf6emKNAF9
cPBX3x2dzSK6sgTFHEkcMfebr7ucUqVEAFmt8NDzWQ54R89GPLXM

m/44'/1'/0'/0/9
mh9aeMiJ8tdYr2mZujqiqbQTFLP5e3dQK9
cNPikC9SUHX4TxqAWcg1ULLRC8P88ZRhDJfPmfRC4ssEbBHehJt4

m/44'/1'/0'/1/0
mgAHZvH2yETQB8SbkPLHP7CoiXY5dVfeG7
cPkFDUb1dNErCCXYNBDJagaKWuAiYpWi4s4f3GCbausQCSetBB37

m/44'/1'/0'/1/1
mwNMcimwUwUeH7rakyMRpqm9jqJjbsChYT
cSw2Kru9XGEYShg34ixXQRjWr3vHGfpkpxHSUVhmWYNddEWWafpx

m/44'/1'/0'/1/2
mzvDM9MpgoKabWPyLHMhgx6K82CgfhYffZ
cU5mZ93MKYzFwY1tkYS4xWwXTDDt9yjbY4WkKQBd4eY33aHY4QBd

m/44'/1'/0'/1/3
mzYQrfriPJ4YrivvpqLBBw4QW4fzGNbJZ1
cToMX9smy1U3Qc4z8yZabPE3niqTwwB7vVaZGJ7HwqF8HNUVpXRA

m/44'/1'/0'/1/4
mhpSP2g6qxaWx8SZtRXoDHvWXUHJJodKFC
cUCJpAvtjzTBVkvsW27qs9SAEVwJUSw2npQRQHF861gRUYwGK649

Para añadir fondos a nuestra cartera simplemente tendremos que ejecutar el código especificando que queremos una cartera para ser usada con Bitcoin Testnet, y de esta manera las direcciones serán creadas con este fin.

Después deberemos copiar la primera dirección e ir a nuestra Faucet favorita; una Faucet no es más que un pool de las criptomonedas especificas a una blockchain, las cuales son distribuidas bajo demanda para fines de testeo.

Faucets:

- https://bitcoinfaucet.uo1.net/
- https://testnet.help/en/btcfaucet/testnet
- https://testnet-faucet.mempool.co/
- https://coinfaucet.eu/en/btc-testnet/

Una ver que hayamos introducido nuestra dirección y enviado algunos Satoshis (1 Bitcoin = 100,000,000 Satoshis) a nuestra nueva cartera, podremos ver el progreso de estos desde cualquiera explorador de Bitcoin testnet blockchain.

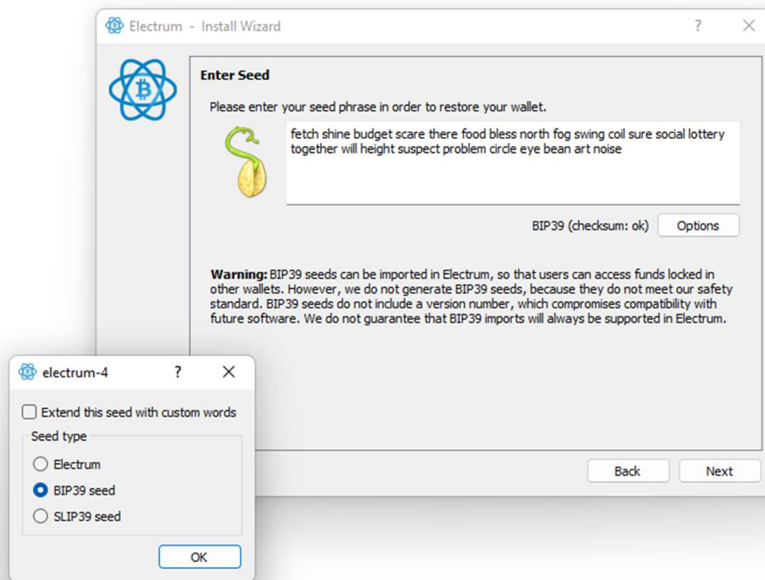
Exploradores de Bitcoin Testnet:

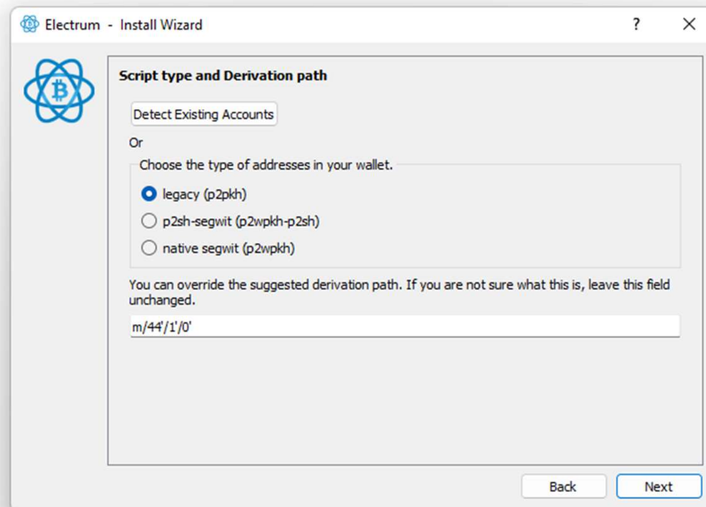
- https://www.blockchain.com/explorer?view=btc-testnet
- https://blockchair.com/bitcoin/testnet
- https://tbtc.bitaps.com/

También podremos descargarnos una aplicación como Electrum (Una cartera para Bitcoin), la cual nos ayudara a ver nuestras transacciones, así como validar que nuestra cartera es funcional.

Electrum Bitcoin Wallet

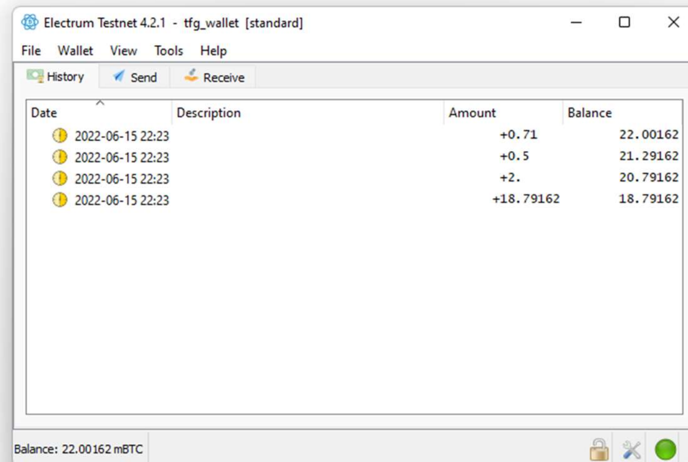
Para importar nuestra frase tendremos primero que correr Electrum Testnet, el cual nos creará y escaneará la Bitcoin Testnet en busca de transacciones y direcciones asociadas con nuestra cuenta.





Tendremos que importar nuestra cartera especificando la BIP correspondiente, en este caso BIP39, así como el tipo de direcciones que se crearán, las cuales en este caso serán P2PKH. En caso de que el tercer miembro de la derivación sea un “0” el cambiarlo manualmente no nos sirve, puesto que las direcciones seguirán siendo creadas para la Mainet y no Testnet. Tendremos el mismo problema si corremos Electrum Testnet e intentamos derivar un árbol reservado para Mainet.

Una vez hallamos importado nuestra cartera Bitcoin, podremos validar que los fondos enviados por nuestra faucet a las direcciones que hemos creado han sido recibidos correctamente.



Posteriormente podremos clicar en (View >> Show Addresses) localizado en la barra de herramientas para poder acceder a una visualización de direcciones Bitcoin. Veremos que esta estructura es muy parecida a la que hemos creado, debido a que la inspiración ha sido tomada de esta pestaña.

Electrum Testnet 4.2.1 - tfg_wallet [standard]

File Wallet View Tools Help

History Send Receive Addresses

Type	Address	Label	Balance	Tx
receiving	n194pq5LLXpDRzBZssuQayP9A7pfSfAjbF		0.71	1
receiving	mkYTxuQmgrfDQ9WzVoN43A1y4PR2zPw1ec		0.5	1
receiving	miYWGe3a8GQaMTdTFf8HLmjHEQGq75CozJ		2.	1
receiving	mjqcz6yV8UGP5mZnxqDY9BRcvAUAY2AS4T		18.79162	1
receiving	mrCA9AoGpUNbjue8nW1vupeZKRzvGfk4TX		0.	0
receiving	n38RBjtXLv1kx8FfBLcgbAdPz5wtvtUvaL		0.	0
receiving	miEHYUEEUMfhkpBY5DLNtaL2yNLApFpp4x		0.	0
receiving	n3Vt8vd6DX5MHdcD8JQFsPaPtbC3TGptU1		0.	0
receiving	mu3xJnYC53FimrPhhHWvoKVmrf6emKNAF9		0.	0
receiving	mh9aeMiJ8tdYr2mZujqiqbQTFLP5e3dQK9		0.	0
receiving	mpDJQXQw5U14GXYurdV8a88oMJNrobqAzy		0.	0
receiving	mtjG6YuMCG9ZheuWnfyqhhMYxyBhxFPVjV		0.	0
receiving	mk6UW1JnTrBe9mAYVW188cyWUj3TMr8jJW		0.	0
receiving	mju14C7dxkdh5hbdHkvSX67fDRiH3ihGHV		0.	0
receiving	mirasDzgB8otKVpvHAAELDi5b6fQGyLwem		0.	0
receiving	mhcRSwPehkCrHXH7eSn7Y429xdPyhTJNeF		0.	0
receiving	mu2cwKMiJTUHrmMos1FVMceTGDqBtRJ3gz		0.	0
receiving	n3GBpFxTjb4i6rYVyrukBbmRqaYrCKqNFA		0.	0
receiving	mw2JRzvXce67yFXSwshhvpLMGG8jzviKH3		0.	0
receiving	n4JCNXaZiFQbokS3QhokLqqY9qCDnhyGRb		0.	0
receiving	moEGRSTGhSySzaYdsLAbpPiq7Gqn8RkmZq		0.	0
receiving	n36bQQrJ3gCsYp5ju1d8K5j2y6NSpTBtCw		0.	0
receiving	mu4HmgmxvMm92PaNFNqTThJDxFT5dK3AVr		0.	0
receiving	mrsF3ffbAdQ2QMYWVCEe5onDVmAXjvJ7SW		0.	0
change	mgAHZvH2yETQB8SbkPLHP7CoiXY5dVfeG7		0.	0
change	mwNMcimwUwUeH7rakyMRpqm9jqJjbsChYT		0.	0
change	mzvDM9MpgoKabWPyLHMhgx6K82CgfhYffZ		0.	0
change	mzYQrfriPJ4YrivvpqLBBw4QW4fzGNbJZ1		0.	0
change	mhpSP2g6qxaWx8SZtRXoDHvWXUHJJodKFC		0.	0
change	moMUbX5htAAWuwuZHeSWqZo9vRVJcDxKJN		0.	0
change	n118dvyamXVJxZgMUb2zaJgLTMGmp9ca3r		0.	0
change	mhCoEvscieD2Utdehs2yf8jxYz4TtWnqBs		0.	0
change	mkhbSFxrLr1sGzT1Bj2HJwYfmVtRWbFYW2		0.	0
change	myKrBx5pGWpc3y4ueAY2to6m7hF9wDjyMC		0.	0

Balance: 22.00162 mBTC

En la sección (Addresses) podemos ver las distintas direcciones, que coinciden con las posteriormente calculadas, así como las cantidades transferidas. Por razones de seguridad las carteras Bitcoin modernas tienden a hacer uso de una dirección nueva para cada transacción.

Cuando una cartera trata de averiguar si hay fondos en ella, esta busca transacciones que contengan las primeras 12 direcciones; en caso de que estas 12 direcciones hayan sido usadas con anterioridad, la cartera seguirá generando y buscando direcciones en la blockchain correspondiente hasta toparse con 12 direcciones consecutivas que no hayan sido utilizadas. Es por ello por lo que las direcciones han de ser usadas desde el índice "0" en adelante sin saltan aleatorios, puesto que esto podría hacer que algunas aplicaciones tomen nuestra cartera como vacía.

Por lo que parece, Electrum hace uso de las primeras 20 en lugar de las primeras 12 lo que da algo más de juego en caso de que no hayan sido utilizadas en orden.

Muchas carteras de criptomonedas hacen uso de códigos QR como medio para transaccionar rápidamente, ya que este método es mucho más rápido y sencillo que el introducir una clave dígito por

dígito, así como el que añade un pequeño código corrector de errores que hace el sistema muy robusto en comparación con jugar al teléfono escacharrado con un String considerablemente largo.

```
address = recieve_addresses[0] # Substitude for address to encode

QRCode = pyqrcode.create(address)
buffer = io.BytesIO()
QRCode.svg(buffer, scale=6)

SVG(buffer.getvalue())
```

HD_wallet.ipynb

11 SCRIPT

Script es un micro lenguaje de programación usado para la liberación y bloqueo de fondos:

- Locking script o también llamado scriptSig es el script que bloqueará los fondos, y que usaremos para enviar valor a distintas direcciones.
- Unlocking script o scriptPubKey es el script que desbloquea los fondos asignados a una de nuestras direcciones, para que podamos hacer uso de ellos y así enviar estos a un tercero.

Un script está compuesto por dos entidades: los datos, piezas de información como números, claves o firmas y los opcodes, pequeñas funciones o subrutinas que toman datos como entrada y devuelven datos como salida. A continuación, podemos ver una lista de algunos opcodes:

OP_CHECKSIG: 0xac
OP_EQUAL: 0x87
OP_HASH160: 0xa9
OP_EQUALVERIFY: 0x88
OP_CHECKMULTISIG: 0xae
OP_DUP: 0x76
OP_0: 0x00
OP_1: 0x51
OP_2: 0x52
OP_3: 0x53
OP_4: 0x54
OP_5: 0x55
OP_6: 0x56
OP_7: 0x57
OP_8: 0x58
OP_9: 0x59
OP_10: 0x5a
OP_11: 0x5b
OP_12: 0x5c
OP_13: 0x5d
OP_14: 0x5e
OP_15: 0x5f
OP_16: 0x60

```
OP_CODES = {
  'OP_CHECKSIG': b'\xac',
  'OP_EQUAL': b'\x87',
  'OP_HASH160': b'\xa9',
  'OP_EQUALVERIFY': b'\x88',
  'OP_CHECKMULTISIG': b'\xae',
  'OP_DUP': b'\x76',
```

```
  'OP_0': b'\x00'
}

for i, code in enumerate(range(81,97)):
  name = f'OP_{i+1}'
  OP_CODES[name] = code.to_bytes(1, 'big')

for i in OP_CODES:
  print(i, end='')
  print(": 0x"+OP_CODES[i].hex())
```

Transactions.ipynb

Los datos no son más que pequeños fragmentos de información, que usaremos para alimentar los distintos opcodes y así operar con ellos. Estos pequeños trozos de información son codificados y descodificados con la utilización de las siguientes dos funciones:

```
def data_encoding(data):
  count = len(data)
  if count < 76:
    return count.to_bytes(1, 'little') + data
  elif count < 2**8:
   return b'\x4a' + count.to_bytes(1, 'little') + data #OP_PUSHDATA1
  elif count < 2**16: # cap at 520
    return b'\x4d' + count.to_bytes(2, 'little') + data #OP_PUSHDATA2
  elif count < 2**32:
    return b'\x4e' + count.to_bytes(4, 'little') + data #OP_PUSHDATA4

def data_decoding(data):
  code = int.from_bytes(data.read(1), 'little')
  if code == 0x4a:
    size = int.from_bytes(data.read(1), 'little') #OP_PUSHDATA1
    return data.read(size)
  elif code == 0x4d:
    size = int.from_bytes(data.read(2), 'little') #OP_PUSHDATA2
    return data.read(size)
  elif code == 0x4e:
    size = int.from_bytes(data.read(4), 'little') #OP_PUSHDATA4
    return data.read(size)
  else:
    return data.read(code)
```

Transactions.ipynb

Una transacción en Bitcoin no es más que una cantidad de Bitcoin que ha sido bloqueada a una de nuestras direcciones, de las cuales nosotros somos los únicos dueños de la clave privada. Para poder usar estos

fondos nosotros tendremos que desbloquear estos fondos verificando así que somos los dueños de dicha clave.

El lenguaje Script nos permite una gran flexibilidad a la hora de crear distintos métodos para bloquear y desbloquear nuestros fondos, pero por razones de seguridad los nodos de la red Bitcoin descartan cualquier tipo de script que no haya sido estandarizado, haciendo de Bitcoin un entorno menos flexible, pero a su vez más seguro. Más adelante veremos algunos de los scripts usados en Bitcoin, pero antes de eso veamos cómo funciona la ejecución de un script.

Para ejecutar un script necesitaremos tanto la parte que bloquea los fondos como la que la desbloquea para poder comprobar la validez de estas. La parte encargada de bloquear los fondos o ScriptSig es aportada por la persona que nos realizó la transacción de los fondos con los que vamos a operar, y la parte que desbloquea estos fondos es la llamada ScriptPubKey, que añadiremos para poder desbloquear el acceso a estos.

Para el siguiente ejemplo haremos uso de una transacción P2PKH. La ejecución de todo script en Bitcoin se lleva a cabo en una estructura Stack que usaremos a modo de registro escribiendo y leyendo de esta frecuentemente.

1. Introduciremos los datos que nos encontremos antes del primer opcode.

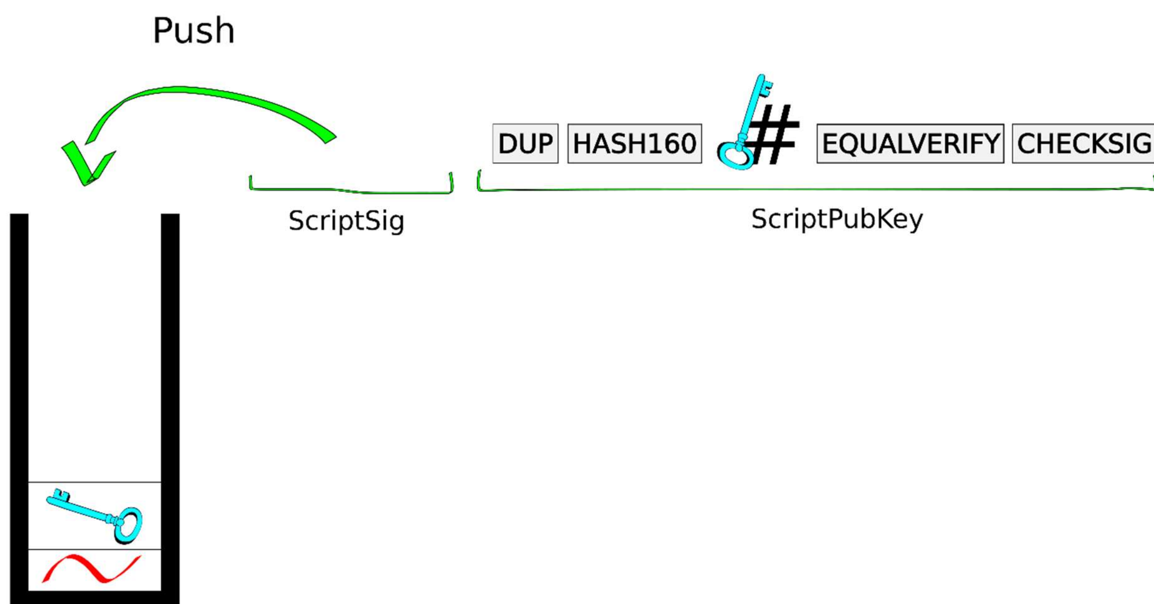


Ilustración 33

2. El primer opcode en este caso no es más que una simple duplicación del primer elemento del Stack, el cual en este caso es nuestra clave pública. Procedemos a duplicar e introducir el resultado en el Stack y desechamos el opcode.

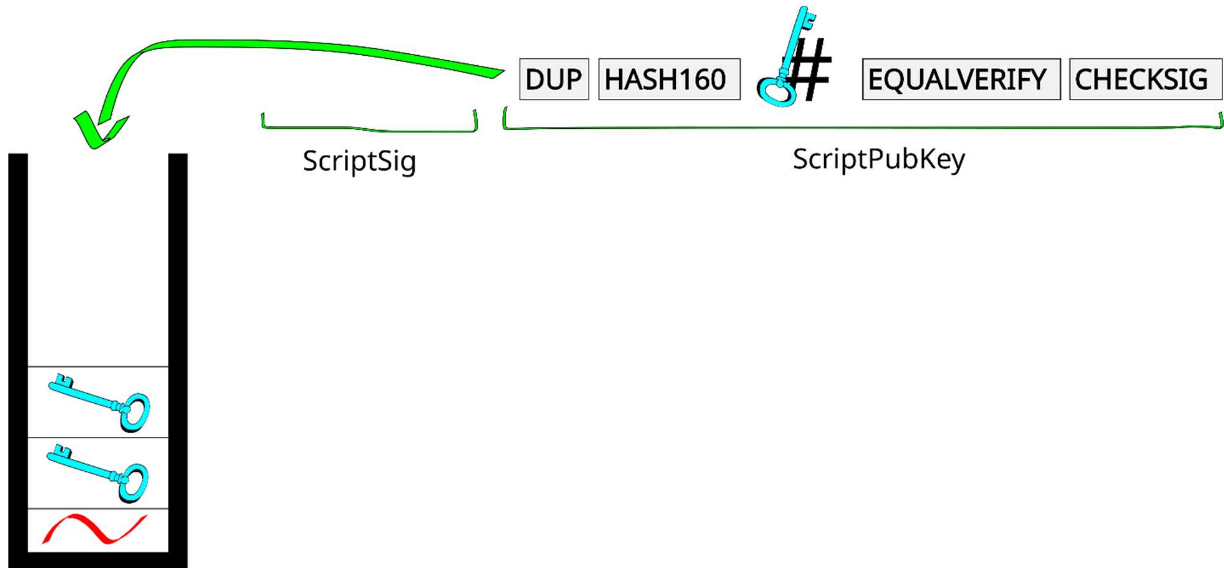


Ilustración 34

3. El siguiente opcode llamado hash160 realiza un hash160 del primer dato del Stack, por lo que hacemos pop de este, se lo pasamos al opcode, e introduciremos en el Stack el nuevo valor generado de vuelta al Stack.

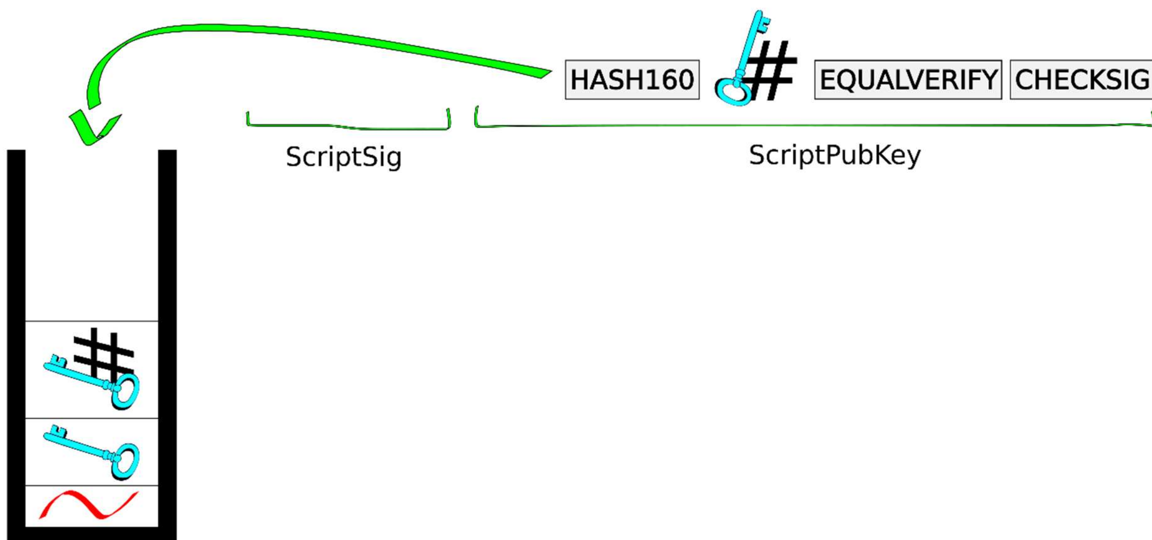


Ilustración 35

4. Puesto que la siguiente parte es un fragmento de datos, al igual que con los datos anteriores lo empujamos al Stack.

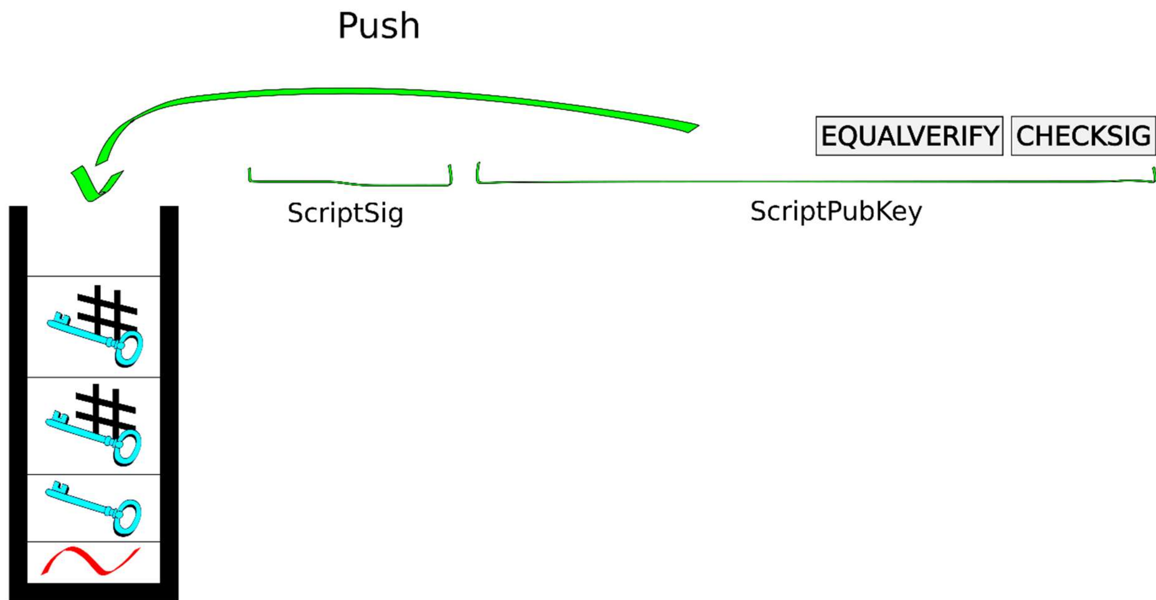


Ilustración 36

5. El siguiente opcode toma dos datos como inputs y verifica que estos sean iguales; en caso de que no sea así la rutina se parará y el proceso de verificación habrá fallado.

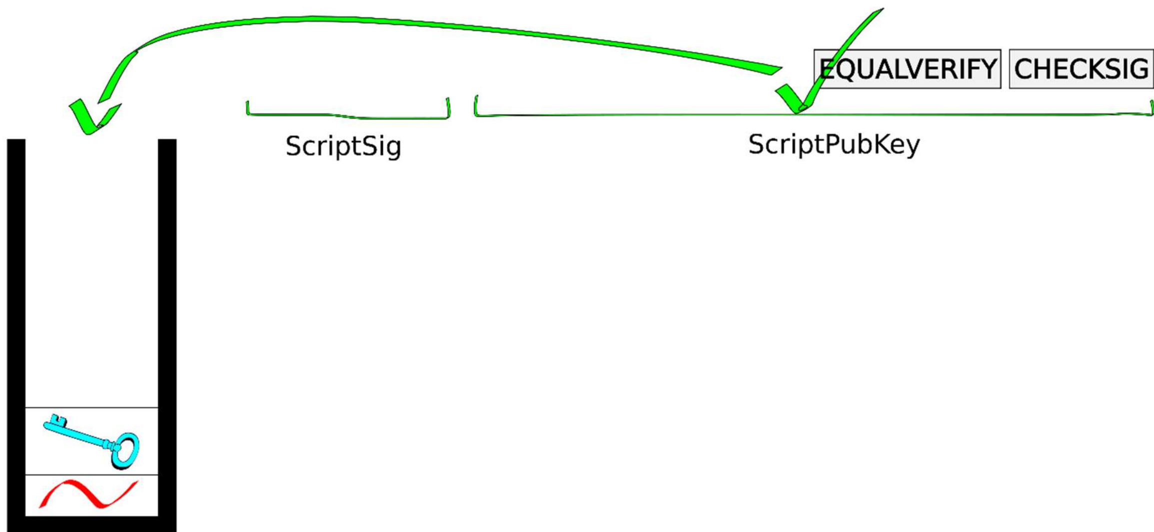


Ilustración 37

6. Finalmente se verificarán los dos pedazos de datos restantes, que serán la firma digital contra nuestra clave pública, y si la firma es validada correctamente se añade un 1 o True en el Stack, resultando en la correcta y completa ejecución del script.

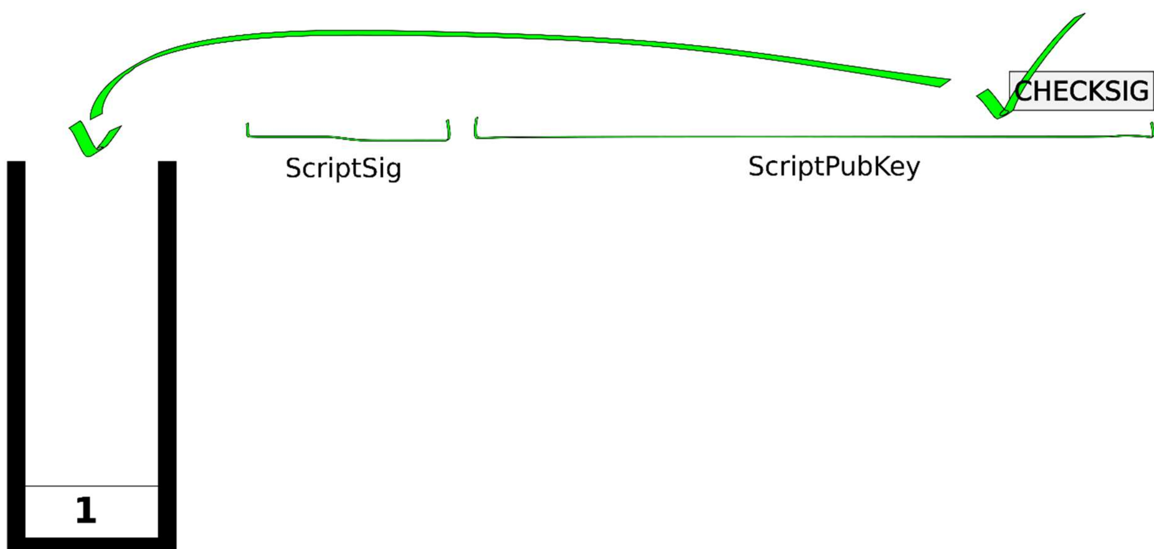


Ilustración 38

A continuación, vamos a describir/mostrar algunos de los scripts más populares y utilizados, para ver qué datos y opcodes son utilizados para bloquear y desbloquear los fondos a la hora de realizar transacciones.

11.1 P2PK

El script P2PK o (Pay to Public Key) fue uno de los primeros utilizados en Bitcoin, pero ha caído en desuso ya que es considerado una mala práctica desde el punto de vista de la seguridad, al exponer la clave pública desde el primer momento, y aunque como ya vimos anteriormente la criptografía asimétrica basada en curvas elípticas es muy robusta y confiable, nuevos tipos de scripts nos ofrecen mayor seguridad.

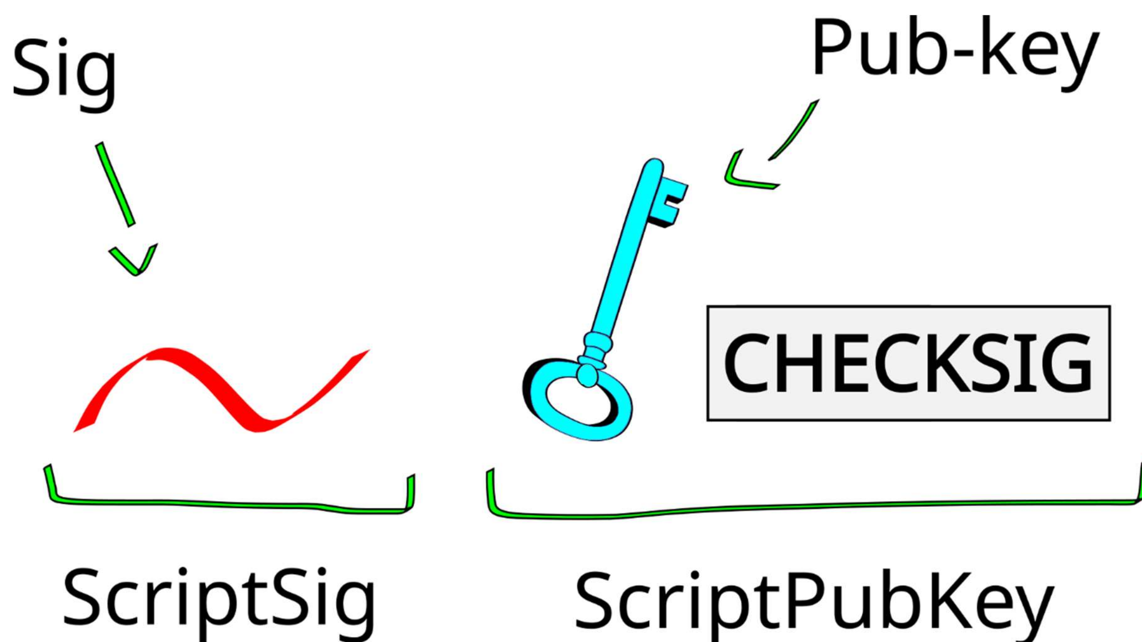


Ilustración 39

11.1.1 ScriptPubKey

```
def p2pk_scriptPubKey(pub_key):
  return data_encoding(pub_key) + OP_CODES['OP_CHECKSIG']
```

Transactions.ipynb

11.1.2 ScriptSig

```
def p2pk_scriptSig(sig):
  return data_encoding(sig)
```

Transactions.ipynb

11.2 P2PKH

El script P2PKH soluciona el problema de P2PK añadiendo el hash de la clave pública y no dando a conocer esta hasta el momento en el cual los fondos son gastados, lo cual al ser acompañado de la política de solo usar una clave o dirección para una única transacción resulta que, si por cualquier motivo se descubriese el cómo obtener la clave privada a partir de nuestra clave pública, nadie podría robar nuestros fondos puesto que estos estarían a su vez protegidos por una función hash.

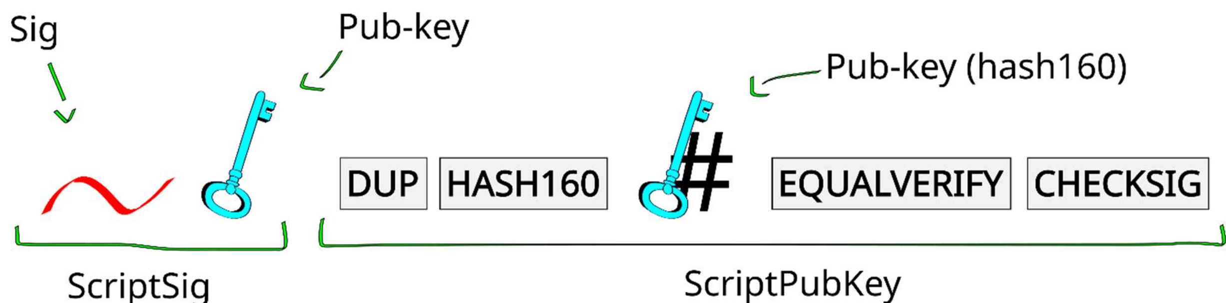


Ilustración 40

11.2.1 ScriptPubKey

```
def p2pkh_scriptPubKey(pub_key):
  hashed_key = hash160(pub_key)
  return OP_CODES['OP_DUP'] + OP_CODES['OP_HASH160'] +
data_encoding(hashed_key) + OP_CODES['OP_EQUALVERIFY'] +
OP_CODES['OP_CHECKSIG']
```

Transactions.ipynb

11.2.2 ScriptSig

```
def p2pkh_scriptSig(sig, pub_key):
  return data_encoding(sig) + data_encoding(pub_key)
```

Transactions.ipynb

11.3 P2MS

P2MS o (Pay to Multi Sig) es un script que demuestra el potencial y posibilidades de Script (el pseudo lenguaje de programación); funciona bloqueando los fondos bajo una póliza que dicta que solo si un numero “x” de firmas “y” son verificadas, los fondos podrán ser usados. En este caso tres claves son dadas, de las cuales se han de incluir al menos dos firmas que correspondan a estas como parte del ScriptSig para desbloquear los fondos.

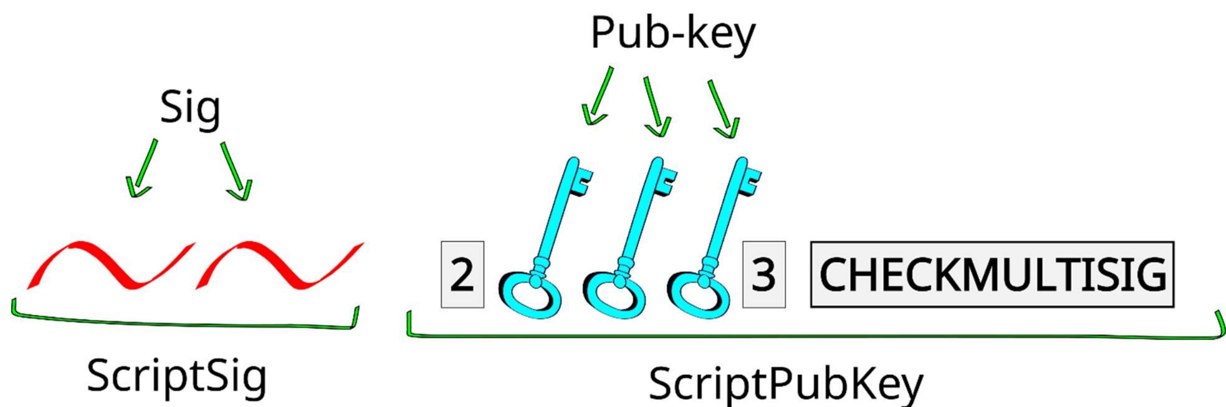


Ilustración 41

11.3.1 ScriptPubKey

```
def p2ms_scriptPubKey(key_array, n_sig):
  result = b''
  result += OP_CODES[f'OP_{n_sig}']
  for key in key_array:
    result += data_encoding(key)
  result += OP_CODES[f'OP_{len(key_array)}']

  return result + OP_CODES['OP_CHECKMULTISIG']
```

Transactions.ipynb

11.3.2 ScriptSig

```
def p2ms_scriptSig(sig_array):
  result = b''
  for sig in sig_array:
    result += data_encoding(sig)
  return OP_CODES['OP_0'] + result
```

Transactions.ipynb

11.4 P2SH

El script P2SH o (Pay to Script Hash) se ha convertido en uno de los favoritos últimamente por dos cualidades principalmente. La primera es que este modelo de script es compatible con una tecnología llamada SegWit o (Segregated Witness), que hace posible el reducir el tamaño de la transacción, haciendo así el coste de transacción más económico. Y la segunda es que, como podemos observar, el ScriptPubKey es bastante pequeño y a su vez que predecible en longitud, lo que hace de este muy cómodo para la persona que envía la transacción.

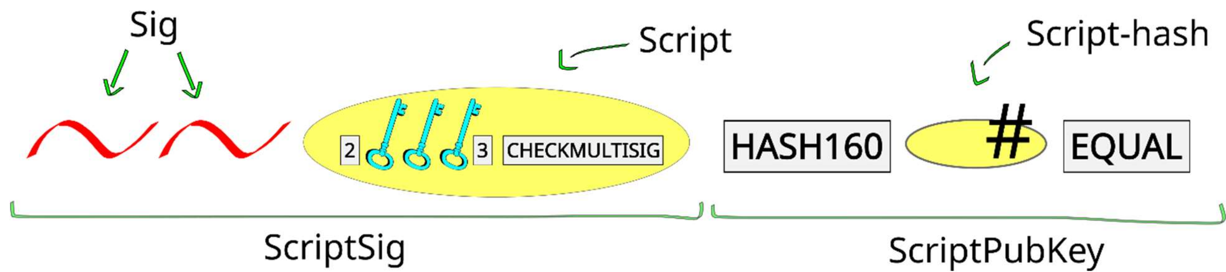


Ilustración 42

Pero como con todo, esto tiene sus ventajas y desventajas. Para desbloquear los fondos de estas transacciones tendremos que incluir el script en nuestro ScriptSig, lo que hace que nuestra cartera tenga que mantener un historial de las transacciones y scripts relacionados con estas, lo cual desde mi punto de vista mata la magia de una cartera determinista.

11.4.1 ScriptPubKey

```
def p2sh_scriptPubKey(script):
  result = data_encoding(hash160(script))
  return OP_CODES['OP_HASH160'] + result + OP_CODES['OP_EQUAL']
```

Transactions.ipynb

11.4.2 ScriptSig

```
def p2sh_scriptSig(sig_script, lock_script):
  return sig_script + data_encoding(lock_script)
```

Transactions.ipynb

11.4.3 NULL DATA

Finalmente quería también incluir el siguiente script el cual me ha parecido interesante; este es un script que es utilizado simplemente para almacenar datos en la blockchain y así hacer uso de la robustez e integridad de esta tecnología a la hora de guardar información.

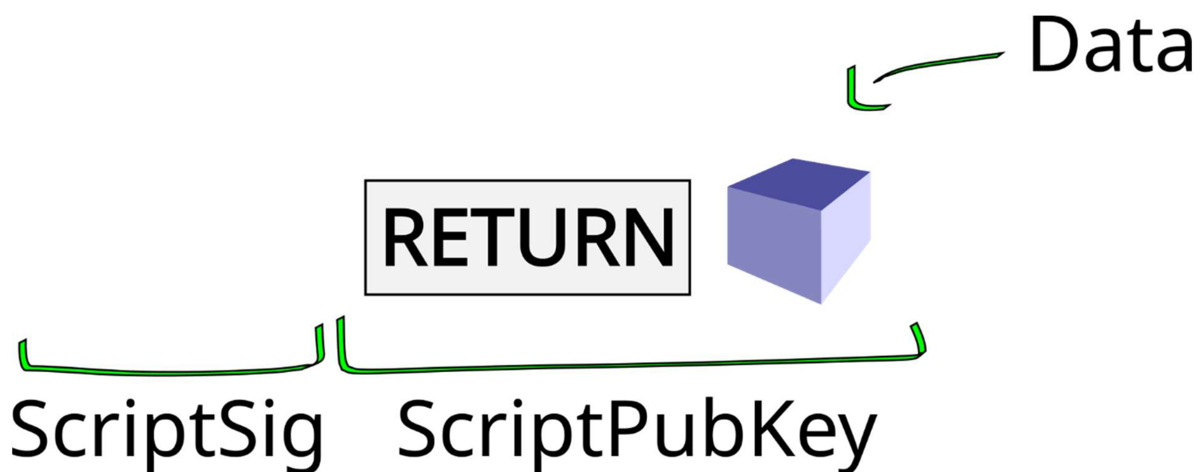


Ilustración 43

12 TRANSACCIÓN

Uno de los elementos que veremos más repetidos a la hora de crear una transacción es el llamado VarInt, esta es la forma usada en Bitcoin para codificar números de tamaño variable. Puesto que los bloques y transacciones de Bitcoin no tienen un tamaño fijo y por consiguiente no tienen un número de entidades fijo, se decidió hacer uso del siguiente método para el envío y almacenamiento de números variables.

```
def read_varint(s):

  flag = s.read(1)
  if flag == 0xfd:
    # 0xfd next two bytes are the number
    return flag.read(2)
  elif flag == 0xfe:
    # 0xfe next four bytes are the number
    return flag.read(4)
  elif flag == 0xff:
    # 0xff next eight bytes are the number
    return flag.read(8)
  else:
    # the flag is the number
    return flag
```

HD_wallet.ipynb

Los VarInt pueden o bien indicar el número de elementos que viene a continuación de este, o el largo en bytes del campo a continuación.

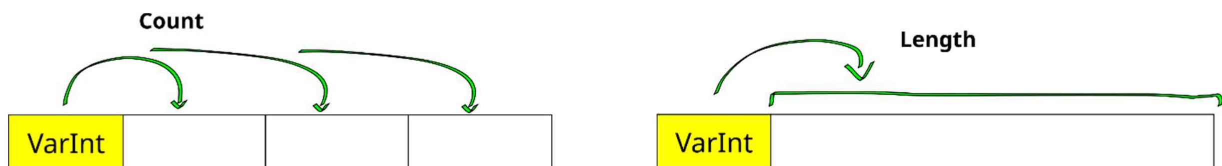


Ilustración 44

Un VarInt es construido de la siguiente manera:

Tamaño	Ejemplo	Descripción
<= 0xfc	12	Simplemente el número.
<= 0xffff	fd1234	Prefijo (fd), y los 2 bytes siguientes.
<= 0xffffffff	fe12345678	Prefijo (fe), y los 4 bytes siguientes.
<= 0xffffffffffffffff	ff1234567890abcdef	Prefijo (ff), y los 8 bytes siguientes.

```
def encode_varint(i):

  if i < 0xfd:
    return bytes([i])
  elif i < 0x10000:
    return b'\xfd' + i.to_bytes(2, 'litte')
  elif i < 0x100000000:
    return b'\xfe' + i.to_bytes(4, 'litte')
  elif i < 0x10000000000000000:
    return b'\xff' + i.to_bytes(8, 'litte')
  else:
    raise ValueError('integer too large: {}'.format(i))
```
HD_wallet.ipynb

Ahora que sabemos cómo funciona Script y hemos definido el funcionamiento de los VarInts, podemos crear una transacción, que es la última pieza de una cartera Bitcoin. Una buena práctica a la hora implementar una cartera es la de que esta sea la encargada de crear la transacción en su totalidad, y solo una vez esta sea creada podremos difundirla por la red. De esta manera nos aseguramos de que en ningún momento hemos cedido nuestras claves a un tercero para operar con nuestros fondos.

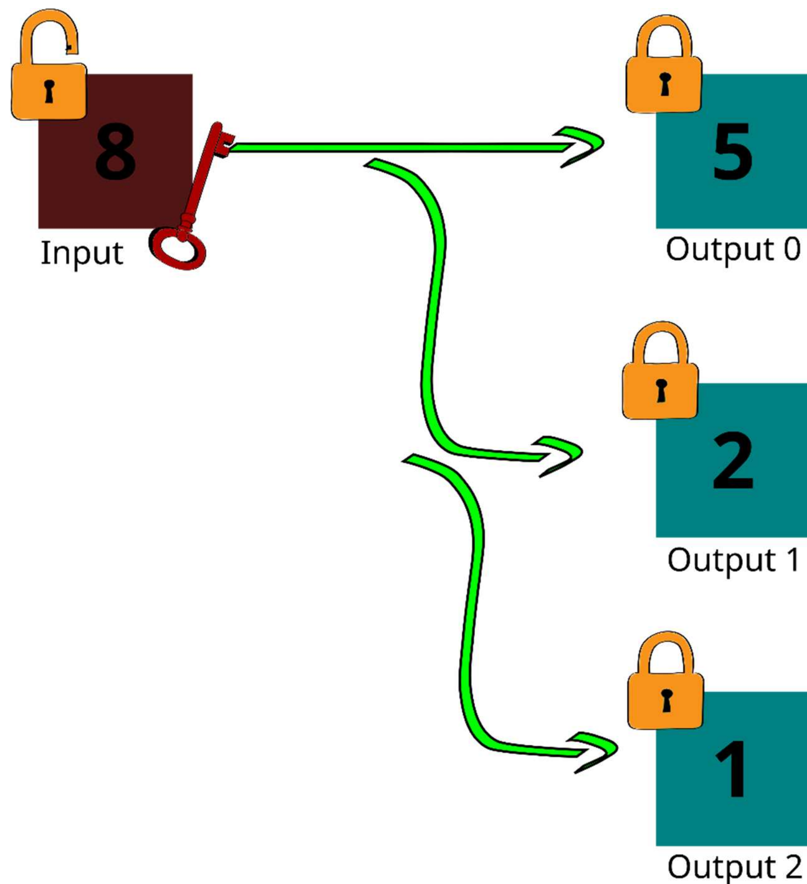


Ilustración 45

Como podemos ver en la ilustración superior, una transacción no es más que un conjunto de entradas y salidas, siendo las entradas los fondos enviados nuestras direcciones y las salidas los fondos que nosotros enviamos a distintos destinatarios, entre los cuales podemos ser incluidos.

El hecho de que nos podamos incluir como destinatarios en una transacción es la forma que Bitcoin tiene de lidiar con las distintas cantidades, y el factor de que Bitcoin en sí no mantiene el balance de nuestras cuentas, lo cual le da el valor añadido de la privacidad.

```
class TX:
  def __init__(self, inputs, outputs, version=b'\x00\x00\x00\x01' ,
locktime=b'\x00\x00\x00\x00'):
    self.inputs = inputs
    self.outputs = outputs
    self.version = version
    self.locktime = locktime

  def serialize(self):
    serialized_tx = self.version[::-1]
    serialized_tx += encode_varint(len(self.inputs))
    for i in self.inputs:
```

```
      serialized_tx += i.serialize()

    serialized_tx += encode_varint(len(self.outputs))
    for i in self.outputs:
      serialized_tx += i.serialize()

    serialized_tx += self.locktime[::-1]
    return serialized_tx

  @classmethod
  def parse(cls, tx):
    stream = BytesIO(bytes.fromhex(tx))

    version = stream.read(4)[::-1]

    input_count = int.from_bytes(read_varint(stream), 'little')
    inputs = []
    for i in range(input_count):
      inputs.append(TxIn.parse(stream, True))

    output_count = int.from_bytes(read_varint(stream), 'little')
    outputs = []
    for i in range(output_count):
      outputs.append(TxOut.parse(stream, True))
    locktime = stream.read(4)[::-1]

    return cls(inputs, outputs, version, locktime)

  def __repr__(self):
    string = ''
    string += f'Version: {int.from_bytes(self.version, "big")}\n'
    string += f'Input count: {len(self.inputs)}\n'
    string += '{\n'
    for input in self.inputs:
      string += input.__repr__() + ',\n'
    string += '}\n'
    string += f'Output count: {len(self.outputs)}\n'
    string += '{\n'
    for output in self.outputs:
      string += output.__repr__() + ',\n'
    string += '}\n'
    string += f'Locktime: {int.from_bytes(self.locktime, "big")}\n'
    return string

  def colors(self):
```

```
    result = ''
    result += '\033[94m' + self.version[::-1].hex()

    result += '\033[93m' + encode_varint(len(self.inputs)).hex()
    result += '\033[96m'
    for i in self.inputs:
      result += i.serialize().hex()

    result += '\033[93m' + encode_varint(len(self.outputs)).hex()
    result += '\033[95m'
    for i in self.outputs:
      result += i.serialize().hex()

    result += '\033[92m' + self.locktime[::-1].hex()

    result += '\n\033[94m\u2589 Version'
    result += '\n\033[93m\u2589 Counter'
    result += '\n\033[96m\u2589 Inputs'
    result += '\n\033[95m\u2589 Outputs'
    result += '\n\033[92m\u2589 Sequence'

    return result
```
Transactions.ipynb

Como ya mencionamos anteriormente una transacción en Bitcoin está limitada por su tamaño, puesto que, a mayor tamaño, mayores son los costes transaccionales. Es por ello por lo que en las transacciones se hace uso extenso de VarInts los cuales nos denotan la longitud o número de campos que viene a continuación.

En el diagrama inferior podemos observar cada uno de los campos que constituye una transacción, remarcado en rojo tenemos los inputs, que se añadirán tantos como sean necesarios para igualar o sobrepasar el valor que queremos transferir. O en caso de que dicha transacción tenga el propósito de desfragmentar nuestra cartera, incluiremos tantos inputs como sea posible y rentable, para reenviarlos la cantidad resultante como cambio de manera consolidada.

Name	Datos	Size	Description
Version	01000000 ↺	4 bytes	Which version of transaction data structure we're using.
Input Count	01	Var	Indicates the upcoming number of inputs.
Input(s)			
Out Count	01	Var	Indicates the upcoming number of outputs.
Output(s)			
Locktime	00000000 ↺	4 bytes	Set a minimum block height or Unix time that this to be mine.

Input(s):

Name	Datos	Size	Description
TXID	7967...9efc ↺	32 bytes	Refer to an existing transaction.
VOUT	01000000 ↺	4 bytes	Select one of its outputs.
ScriptSig Size	6a	Var	Indicates the upcoming size of the unlocking code.
ScriptSig	47304...1a825		A script that unlocks the input.
Sequence	ffffffff ↺	4 bytes	

Output(s):

Name	Datos	Size	Description
Value	4baf210000000000 ↺	8 bytes	The value of the output in satoshis.
ScriptPubKey Size	19	Var	Indicates the upcoming size of the locking code.
ScriptPubKey	76a914db4...68b0e88ac		A script that locks the output.

Ilustración 46

Los campos principales de una transacción son los siguientes:

- txid & vout: indican a que transacción y output de la blockchain estamos haciendo referencia.
- signature: verifica que los fondos a los que hacemos referencia son nuestros y podemos probarlos verificando el script al cual fueron bloqueados.
- amount: cantidad que enviaremos a cada uno de los destinatarios que incluiremos como outputs.
- lockscript: método que usaremos para bloquear cada una de las cuantías de los distintos outputs.

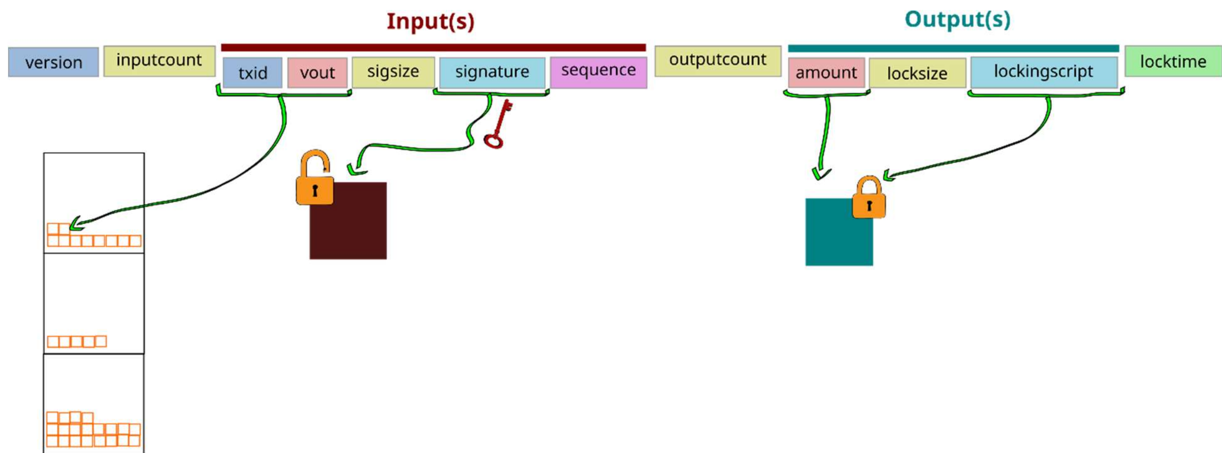


Ilustración 47

```
0100000004b674693a82cdcf1060186c92e530d280120c53952fd8b9cf66adff6b1e1ea76b0100000
06a47304402203c390e403858b3d4683ecd857675278e70b1c7af2a45fbe316328b0bb45c0f130220
41c05ef823d6f86af1d69c9359dddbae209005451f7d3de3eba65f5b4620d951012103b577a59ddc5
7ce173c058c66586064809fe88cd15dd9516614f1e8e5550de5f1ffffffff71dd3443b4ebbd256d70
90e540d471c0b696de5fceb30f2f65ea020245733f14010000006a47304402200be80959e1e9a0318
10b41934e667cd883af0a42bace12314d759ac173ed8c0f02204d3b05ee9177624facaaf86971eaa2
5b9362c95c3c8e90fd8cf039dd3bb71dce012102294f00ee94128a71d134fede09001463e46c447cb
1683e2b122e9ea323ef5319ffffffff0e1e6f6008f2a46e4dc069d2a51e95ec10295f7832c48934cb
0b46bf495627dd000000006a47304402202f7e4d368d50f4cf72a07922ded69e7aa8f5c106da57136
130ccff48bfc7ed56022038eddfbc42e53800870b476998606c5c2f26a71950e55d79ef1337cc64b6
a002012103e25df6966fe91e3bb7656b2b833d1ced860a22ac72c4ee1fbbefbed735918ca8ffffff
f3da3e5197ddb87a8f025b5bc230ae040c2c776dd804475619bb43eb1fc985b1301000006b483045
022100b0c7736d2a6f9c87b88c49289cf485962c511211d86e93e307cd03ca6ae69bac02203bb0666
0c13e228c261bb9b9396129c26336232138711622735523880a4748340121032ec518e302bd1a8652
92817db538fb3e75bca805ed16a02c7df5d9cf4126357ffffffff02880d01000000000001976a9147
51b5e9c9cf3ea616fc4f62f1827da9597c67dda88ac200b2000000000001976a914070f7ab537f647
f048c2d1cd89dc7dfed0e6384988ac00000000
```

- Version
- Counter
- Inputs
- Outputs
- Sequence

12.1 TxIn

```
class TxIn:
    def __init__(self, txID, outID, scriptSig = bytes(b''), sequence =
bytes(b'\xff\xff\xff\xff')):
        self.txID = txID # hex string
        self.outIndex = outID # int
        self.scriptSig = scriptSig #
        self.sequence = sequence
```

```
    def serialize(self):
        serialized_tx = self.txID[::-1]
        serialized_tx += self.outIndex[::-1]
        if self.scriptSig != None:
            serialized_tx += encode_varint(len(self.scriptSig))
            serialized_tx += self.scriptSig
        serialized_tx += self.sequence[::-1]

        return serialized_tx

    @classmethod
    def parse(cls, input, hex=False):
        if hex:
            stream = input
        else:
            stream = BytesIO(bytes.fromhex(input))
        TxID = stream.read(32)[::-1]
        index = stream.read(4)[::-1]
        script_size = int.from_bytes(read_varint(stream), 'little')
        script = stream.read(script_size)
        secuence = stream.read(4)[::-1]
        return cls(TxID, index, script, secuence)

    def __repr__(self):
        string = '\t[\n\t'
        string += f'TxID: {self.txID.hex()}\n\t'
        string += f'OutIndex: {self.outIndex.hex()}\n\t'
        if self.scriptSig != None:
            string += f'Script size: {len(self.scriptSig)}\n\t'
            string += f'ScriptSig: {self.scriptSig.hex()}\n\t'
        else:
            string += f'Script size: {None}\n\t'
            string += f'ScriptSig: {self.scriptSig}\n\t'
        string += f'Sequence: {self.sequence.hex()}\n' + '\t]'
        return string

    def colors(self):
        result = ''
        result += '\033[94m' + self.txID[::-1].hex()
        result += '\033[91m' + self.outIndex[::-1].hex()
        if self.scriptSig != None:
            result += '\033[93m' + encode_varint(len(self.scriptSig)).hex()
            result += '\033[96m' + self.scriptSig.hex()
        result += '\033[95m' + self.sequence[::-1].hex()
        result += '\n\033[94m\u2589 TxID'
        result += '\n\033[91m\u2589 Index'
        result += '\n\033[93m\u2589 Script Size'
        result += '\n\033[96m\u2589 ScriptSig'
        result += '\n\033[95m\u2589 Sequence'

        return result
```

Transactions.ipynb

b674693a82cdcf1060186c92e530d280120c53952fd8b9cf66adff6b1e1ea76b010000006a4730440
2203c390e403858b3d4683ecd857675278e70b1c7af2a45fbe316328b0bb45c0f13022041c05ef823

```
d6f86af1d69c9359dddbae209005451f7d3de3eba65f5b4620d951012103b577a59ddc57ce173c058
c66586064809fe88cd15dd9516614f1e8e5550de5f1ffffffff
▌ TxID
▌ Index
▌ Script Size
▌ ScriptSig
▌ Sequence
```

12.2 TxOut

```
class TxOut:

    def __init__(self, value, scriptPubKey):
        self.value = value
        self.scriptPubKey = scriptPubKey

    def serialize(self):
        serialized_tx = self.value[::-1]
        serialized_tx += encode_varint(len(self.scriptPubKey))
        serialized_tx += self.scriptPubKey

        return serialized_tx

    @classmethod
    def parse(cls, output, hex=False):
        if hex:
            stream = output
        else:
            stream = BytesIO(bytes.fromhex(output))
        value = stream.read(8)[::-1]
        script_size = int.from_bytes(read_varint(stream), 'little')
        script = stream.read(script_size)
        return cls(value, script)

    def __repr__(self):
        string = '\t[\n\t'
        string += f'Value: {int.from_bytes(self.value, "big")}\n\t'
        string += f'Script size: {len(self.scriptPubKey)}\n\t'
        string += f'ScriptPubKey: {self.scriptPubKey.hex()}\n' + '\t]'
        return string

    def colors(self):
        result = ''
        result += '\033[91m' + self.value[::-1].hex()
        result += '\033[93m' + encode_varint(len(self.scriptPubKey)).hex()
        result += '\033[96m' + self.scriptPubKey.hex()
        result += '\n\033[91m\u2589 Value'
        result += '\n\033[93m\u2589 Script Size'
        result += '\n\033[96m\u2589 ScriptPubKey'
        return result
```

Transactions.ipynb

```
880d0100000000001976a914751b5e9c9cf3ea616fc4f62f1827da9597c67dda88ac
▌ Value
▌ Script Size
▌ ScriptPubKey
```

Para concluir con esta memoria pondremos junta una transacción con los diversos recursos mostrados anteriormente y la difundiremos en la red Bitcoin donde será confirmada y será minada en un bloque si esta es correcta.

Para esta transacción cogeremos las cuatro transacciones que realizamos anteriormente para añadir fondos a nuestra cartera Bitcoin:

- Transaction: 6ba71e1e6bffad66cfb9d82f95530c1280d230e5926c186010cfcd823a6974b6
- Transaction: 143f73450202ea652f0fb3ce5fde96b6c071d440e590706d25bdebb44334dd71
- Transaction: dd275649bf460bcb3489c432785f2910ec951ea5d269c04d6ea4f208606f1e0e
- Transaction: 135b98fcb13eb49b61754480dd76c7c240e00a23bcb525f0a887db7d19e5a33d

Las emplearemos como (TxIn), y también crearemos dos (TxOut), de las cuales la primera será una transacción a nuestra propia cartera, para la cual utilizaremos la quinta dirección de esta, y con una cuantía de 0.69 mBTC. Y la segunda también estará destinada a nuestra cartera, pero en este caso como cash-back con una cuantía de 21 mBTC.

```
# Inputs
in_0 =
TxIn(bytes.fromhex('6ba71e1e6bffad66cfb9d82f95530c1280d230e5926c186010cfcd823
a6974b6'), bytes(b'\x00\x00\x00\x01'))
in_1 =
TxIn(bytes.fromhex('143f73450202ea652f0fb3ce5fde96b6c071d440e590706d25bdebb44
334dd71'), bytes(b'\x00\x00\x00\x01'))
in_2 =
TxIn(bytes.fromhex('dd275649bf460bcb3489c432785f2910ec951ea5d269c04d6ea4f2086
06f1e0e'), bytes(b'\x00\x00\x00\x00'))
in_3 =
TxIn(bytes.fromhex('135b98fcb13eb49b61754480dd76c7c240e00a23bcb525f0a887db7d1
9e5a33d'), bytes(b'\x00\x00\x00\x01'))
input_list = [in_0, in_1, in_2, in_3]

# Output
lock_script_0 =
p2pkh_scriptPubKey(bytes.fromhex('03024bbfa47bd639ebf6ed77a785175b6b3224a1547
aa7d1401f95e84a6263b7f0'))
amount_0 = 69000
out_0 = TxOut(amount_0.to_bytes(8, 'big'), lock_script_0)

#Cash-back
lock_script_1 =
p2pkh_scriptPubKey(bytes.fromhex('02278c58a944431b425875b60cdb229b1195923843a
83f915779a451fae85fe63c'))
amount_1 = 2100000
cash_back = TxOut(amount_1.to_bytes(8, 'big'), lock_script_1)
```

```
output_list = [out_0, cash_back]

# Transaction
lock_time = 0
tx = TX(input_list, output_list, b'\x00\x00\x00\x01', lock_time.to_bytes(4,
'big'))

# Variables needed
signatures = []
locking_scripts = [
    bytes.fromhex('76a914d73ed0e5506a1b9c4c82fcc4c8d2705d308a1c7988ac'),
    bytes.fromhex('76a9143721cfdffb37ea053c8507c26ee68b92b44ce4d788ac'),
    bytes.fromhex('76a91421337c36b4d44802ed2379b55b2b66a3433fd87588ac'),
    bytes.fromhex('76a9142f6849a431d91d44ba958b0a14c5d9e22222b14888ac')

]

public_keys = []
private_wifs = [
    'cPnztkJGDnweFad2TXjqXdM1r7EBFNQUQ1XoiEfsy8mfED5kaQib',
    'cPrSqshyySHqtF4ZE8ttk4TW7w6ck88jTQfeU415ngTPq9RWrSPs',
    'cRHXZkGZeevh8SbCXYLHcRwsGx1FmuKBLMe8h3LT4M9oYUUiDKqg',
    'cScqWqFzrWhiK3ndTLpKFPuE5kCzi7degAWeqTEJaqWZTCopbAoV'
]

# Cleaning scriptSigs
for input in tx.inputs:
    input.scriptSig = b''

# Signing each input
for wif, lock_s, input_index in zip(private_wifs, locking_scripts,
range(len(tx.inputs))):
    # Creating a copy to work on
    clean_tx = copy.deepcopy(tx)
    # Adding scriptPubKey to the scriptSig field as placeholder
    clean_tx.inputs[input_index].scriptSig = lock_s
    # Decoding private key
    secret_num = wif_decode(wif) # private key wif
    private_key = ec.derive_private_key(int.from_bytes(secret_num, 'big'),
ec.SECP256K1())
    # Signing
    chosen_hash = hashes.SHA256()
    digest = hash256(clean_tx.serialize() + b'\x01\x00\x00\x00')# Adding hash
code
    sig = private_key.sign(
        digest,
        ec.ECDSA(utils.Prehashed(chosen_hash))
    )
    # Adding SigScripts to each input
    signatures.append(sig + b'\x01')# Hash code after digest

    public_keys.append(private_key.public_key().public_bytes(Encoding.X962,
PublicFormat.CompressedPoint))

# Comment this following lines generate new signatures
if True:
```

```
    example_sigs =
['304402203c390e403858b3d4683ecd857675278e70b1c7af2a45fbe316328b0bb45c0f13022
041c05ef823d6f86af1d69c9359dddbae209005451f7d3de3eba65f5b4620d95101',
                    '304402200be80959e1e9a031810b41934e667cd883af0a42bace1231
4d759ac173ed8c0f02204d3b05ee9177624facaaf86971eaa25b9362c95c3c8e90fd8cf039dd3
bb71dce01',
                    '304402202f7e4d368d50f4cf72a07922ded69e7aa8f5c106da571361
30ccff48bfc7ed56022038eddfbc42e53800870b476998606c5c2f26a71950e55d79ef1337cc6
4b6a00201',
                    '3045022100b0c7736d2a6f9c87b88c49289cf485962c511211d86e93
e307cd03ca6ae69bac02203bb06660c13e228c261bb9b9396129c26336232138711622735523 8
80a47483401']
    signatures = [bytes.fromhex(i) for i in example_sigs]

# Adding SigScripts to each input
for sig, pub_key, tx_input in zip(signatures, public_keys, tx.inputs):
    tx_input.scriptSig = p2pkh_scriptSig(sig,pub_key)

print('TxID: '+hash256(tx.serialize())[::-1].hex())
print(tx.serialize().hex())
```

Transactions.ipynb

En el código superior generaremos una transacción con los detalles mencionados, y obtendremos un Sting hexadecimal con los datos de la transacción para ser difundidos en la red Bitcoin, así como el ID de la transacción (TxID), con el cual podremos identificar esta en cualquier explorador Bitcoin.

TxID: 1826038de274c38c4f754e72bfce408d260bebc8686a2ef666873893764e61a1

0100000004b674693a82cdcf1060186c92e530d280120c53952fd8b9cf66adff6b1e1ea76b010000006a47
304402203c390e403858b3d4683ecd857675278e70b1c7af2a45fbe316328b0bb45c0f13022041c05ef823
d6f86af1d69c9359dddbae209005451f7d3de3eba65f5b4620d951012103b577a59ddc57ce173c058c6658
6064809fe88cd15dd9516614f1e8e5550de5f1ffffffff71dd3443b4ebbd256d7090e540d471c0b696de5fce
b30f2f65ea020245733f14010000006a47304402200be80959e1e9a031810b41934e667cd883af0a42bace
12314d759ac173ed8c0f02204d3b05ee9177624facaaf86971eaa25b9362c95c3c8e90fd8cf039dd3bb71dc
e012102294f00ee94128a71d134fede09001463e46c447cb1683e2b122e9ea323ef5319ffffffff0e1e6f6008
f2a46e4dc069d2a51e95ec10295f7832c48934cb0b46bf495627dd000000006a47304402202f7e4d368d50
f4cf72a07922ded69e7aa8f5c106da57136130ccff48bfc7ed56022038eddfbc42e53800870b476998606c5c
2f26a71950e55d79ef1337cc64b6a002012103e25df6966fe91e3bb7656b2b833d1ced860a22ac72c4ee1f
bbefbed735918ca8ffffffff3da3e5197ddb87a8f025b5bc230ae040c2c776dd804475619bb43eb1fc985b13
010000006b483045022100b0c7736d2a6f9c87b88c49289cf485962c511211d86e93e307cd03ca6ae69bac
02203bb06660c13e228c261bb9b9396129c26336232138711622735523880a4748340121032ec518e302
bd1a865292817db538fb3e75bca805ed16a02c7df5d9cf4126357ffffffff02880d0100000000001976a9147
51b5e9c9cf3ea616fc4f62f1827da9597c67dda88ac200b2000000000001976a914070f7ab537f647f048c2
d1cd89dc7dfed0e6384988ac00000000

Antes de continuar me gustaría mencionar que en el código anterior se han sobrescrito las firmas de cada una de las (TxIn) para que al ejecutarse la celda la transacción resultante coincida. Como vimos anteriormente en el capítulo relacionado con las firmas digitales, estas requieren de un número aleatorio criptográficamente seguro por cada firma para garantizar la seguridad. Esto hace que cada ejecución de esta celda genere una transacción valida, pero con ligeras variaciones.

Para difundirla copiaremos la transacción en hexadecimal y la copiaremos en Electrum (Tools >> Load transaction >> From text).

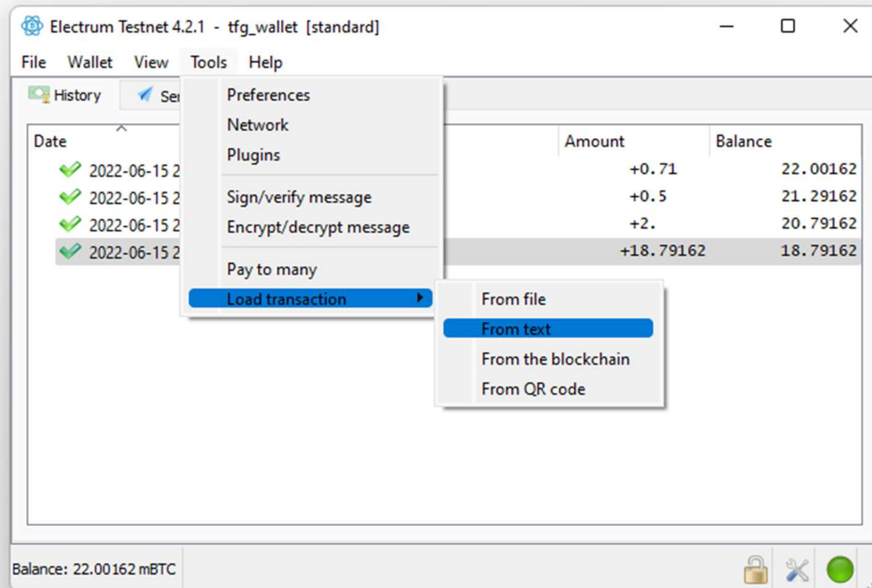


Ilustración 48

En esta ventana simplemente pegamos la transacción y la difundimos por la red Bitcoin donde los mineros la validarán e incluirán en uno de los bloques. Podremos ver y seguir el progreso de este proceso desde nuestra cartera Electrum o bien desde cualquier explorador Bitcoin.

Transaction

Transaction ID:

1826038de274c38c4f754e72bfce408d260bebc8686a2ef666873893764e61a1

Status: Signed
Amount sent: 0. mBTC
Fee: 0.31162 mBTC (46.7 sat/byte)

Size: 667 bytes
Replace by fee: False
LockTime: 0 (height)

Inputs (4)

```
6ba71e1e6bffad66cfb9d82f95530c1280d230e5926c186010cfcd823a6974b6:1    n194pq5LLXpDRzBZssuQayP9A7pfSfAjbF     0.71
143f73450202ea652f0fb3ce5fde96b6c071d440e590706d25bdebb44334dd71:1    mkYTxuQmgrfDQ9WzVoN43A1y4PR2zPw1ec     0.5
dd275649bf460bcb3489c432785f2910ec951ea5d269c04d6ea4f208606f1e0e:0    miYWGe3a8GQaMTdTFf8HLmjHEQGq75CozJ     2.
135b98fcb13eb49b61754480dd76c7c240e00a23bcb525f0a887db7d19e5a33d:1    mjqcz6yV8UGP5mZnxqDY9BRcvAUAY2AS4T    18.79162
```

Outputs (2)

= Receiving Address = Change Address

```
mrCA9AoGpUNbjue8nW1vupeZKRzvGfk4TX     0.69
mgAHZvH2yETQB8SbkPLHP7CoiXY5dVfeG7    21.
```

Export Save Sign Broadcast Close

Ilustración 49

13 HERRAMIENTAS USADAS

En este proyecto hemos hecho uso de un gran abanico de distintas herramientas, desde diseño hasta librerías código. En este apartado intentaré explicar la funcionalidad de las herramientas más recalcables de este proyecto, así como en que ámbitos han sido empleadas.

La gran pieza central y fuente de inspiración para muchos de los rasgos de este proyecto es la cartera que hemos usado a lo largo de este; principalmente nos hemos enfocado en la forma y uso de la cartera de Bitcoin (Electrum), la cual nos ofrece una interfaz de usuario en el ordenador y muchas opciones las cuales no podremos encontrar en otras carteras más dirigidas al público. Pero al igual que este, cabe destacar que muchas preguntas han surgido del uso de la anterior mencionada, así como de la cartera Bitcoin para dispositivos móviles (Micelium).

- https://electrum.org/#home
- https://wallet.mycelium.com/

Para las herramientas que conforman el entorno de desarrollo podremos encontrar nuestro editor de texto, JupyterNotebooks, el intérprete Python y nuestros entornos virtuales, los cuales mencionamos y explicamos brevemente al principio del proyecto.

- https://code.visualstudio.com/
- https://vscodium.com/
- https://www.python.org/
- https://pypi.org/project/pipenv/
- https://jupyter.org/

En la confección de esta memoria se han involucrado dos grupos de herramientas, las cuales están relacionadas con la creación de las ilustraciones en esta y con la edición de texto.

- https://www.office.com/launch/word
- https://www.libreoffice.org/
- https://www.adobe.com/acrobat/pdf-reader.html
- https://www.gimp.org/
- https://inkscape.org/
- https://krita.org/en/

Finalmente hay que destacar los módulos de Python usados para el código empleadas para obtener funcionalidades como: hash, gráficos o QRcodes.

- https://numpy.org/
- https://matplotlib.org/
- https://pypi.org/project/cryptography/
- https://pypi.org/project/bitarray/
- https://pypi.org/project/base58/
- https://pypi.org/project/ECPy/
- https://pypi.org/project/PyQRCode/

14 CONCLUSIONES Y FUTURAS LÍNEAS DE TRABAJO

Llegados a este punto creo que solo queda ver que hemos aprendido. En este caso este trabajo no ha sido más que una pequeña parte de un viaje de aprendizaje más largo. Todo esto comenzó hará más de un año cuando me empecé a interesar por las criptomonedas y leí el primer libro relacionado con el tema, (Mastering Ethereum - by Andreas M. Antonopoulos && Gavin Wood Ph.D). Aprendí el funcionamiento de las blockchain así como muchos conceptos relacionados con las criptomonedas. He de confesar que este es el segundo tema que escojo para el trabajo de fin de grado, ya que el primer tema que me vino a la cabeza fue el de implementar una criptomoneda; y durante el desarrollo de este me di cuenta de la titánica tarea que esto conlleva y la gran cantidad de cosas que desconozco en esta materia.

Finalmente decidí hacer borrón y cuenta nueva centrándome más en un tema en particular, que en aquel momento eran las carteras deterministas. Este trabajo me ha ayudado a ampliar mis conocimientos sobre las firmas digitales basadas en curvas elípticas, los algoritmos hash (de los cuales siempre había hecho uso a modo de caja negra y sin profundizar mucho en el tema) o el lenguaje Script dentro de Bitcoin.

Además, me ha hecho poder hacer uso de algunas herramientas las cuales me gustaría llegar a aprender en su totalidad algún día como el editor vectorial grafico Inkscape o los Jupyter notebooks.

Es por estas razones por lo que hacer este trabajo ha sido una gran experiencia de aprendizaje, algunas veces algo tedia, puesto que he tenido que repasar o reescribir alguna cosa múltiples veces hasta el punto de saturarme con ella, pero sin duda algo que repetiría.

Para el futuro me gustaría continuar esta línea de aprendizaje puesto que esta es una tecnología relativamente nueva en la que hay bastantes innovaciones, y el siguiente paso lógico a la hora de ampliar los conocimientos aquí recogidos sería ver como una cartera interactúa con scripts más recientes, así como tecnologías hechas para trabajar sobre Bitcoin u otras criptomonedas, como podrían ser SegWit o Lightning Network.

15 Bibliografía

15.1 Libros

- **Mastering Ethereum**
  - By Andreas M. Antonopoulos && Gavin Wood Ph. D.
- **Mastering Bitcoin**
  - By Andreas Antonopoulos
- **Programming Bitcoin**
  - By Jimmy Song
- **The Bitcoin Standard**: The Decentralized Alternative to Central Banking
  - By Saifedean Ammous

15.2 Apuntes

- **Fundamentos de Algoritmia** - *(Protocolos y Comunicaciones Seguras)*
  - José Ignacio Farran Martín
- **Criptografía de Clave Pública** – *(Protocolos y Comunicaciones Seguras)*
  - José Ignacio Farran Martín

15.3 Recursos Web

- **Version bytes for BIP32 extended public and private keys — Electrum 3.3 documentation**
  - https://electrum.readthedocs.io/en/latest/xpub_version_bytes.html
- **What Are Those Other Weird QR Codes? - YouTube**
  - https://www.youtube.com/watch?v=KMsvtqQqz5g
- **Bitcoin Core**
  - https://bitcoin.org/en/bitcoin-core/
- **Electrum Bitcoin Wallet**
  - https://electrum.org/#home
- **Mycelium Bitcoin Wallet**
  - https://wallet.mycelium.com/
- **Elliptic curve point multiplication - Wikipedia**
  - https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication
- **Computerphile – YouTube**
  - https://www.youtube.com/user/Computerphile
- **DevCentral - YouTube**
  - https://www.youtube.com/c/devcentral
- **learnmeabitcoin - YouTube**
  - https://www.youtube.com/c/learnmeabitcoin
- **How Does Bitcoin Work? (learnmeabitcoin.com)**
  - https://learnmeabitcoin.com/
- **Python Tutorial (w3schools.com)**
  - https://www.w3schools.com/python/default.asp

- **Pushdata Opcodes - Bitcoin Wiki (bitcoinsv.io)**
    - https://wiki.bitcoinsv.io/index.php/Pushdata_Opcodes
- **Script - Bitcoin Wiki**
    - https://en.bitcoin.it/wiki/Script
- **RPC API Reference — Bitcoin**
    - https://developer.bitcoin.org/reference/rpc/index.html
- **A from-scratch tour of Bitcoin in Python (karpathy.github.io)**
    - https://karpathy.github.io/2021/06/21/blockchain/
- **transactions - How to redeem a basic Tx? - Bitcoin Stack Exchange**
    - https://bitcoin.stackexchange.com/questions/3374/how-to-redeem-a-basic-tx
- **MultiSig wallet BTG transaction error · Issue #5 · Ayms/bitcoin-transactions · GitHub**
    - https://en.bitcoin.it/wiki/Main_Page
- **Bitcoin Wiki**
    - https://en.bitcoin.it/wiki/Main_Page
- **Electrum Bitcoin Wallet**
    - https://electrum.org/#home
- **Bitcoin explorer (bitaps.com)**
    - https://tbtc.bitaps.com/
- **Blockchain Explorer - Search the Blockchain | BTC | ETH | BCH**
    - https://www.blockchain.com/explorer
- **Bitcoin Explorer — Blockchair**
    - https://blockchair.com/bitcoin/
- **Bitcoin Testnet Faucet**
    - https://testnet.help/en/btcfaucet/testnet
- **Bitcoin Testnet Faucet (uo1.net)**
    - https://bitcoinfaucet.uo1.net/
- **Bitcoin testnet3 faucet (coinfaucet.eu)**
    - https://coinfaucet.eu/en/btc-testnet/
- **Opcodes used in Bitcoin Script - Bitcoin Wiki (bitcoinsv.io)**
    - https://wiki.bitcoinsv.io/index.php/Opcodes_used_in_Bitcoin_Script
- **Broadcast bitcoin testnet transaction (bitaps.com)**
    - https://tbtc.bitaps.com/broadcast