Pruebas de conocimiento cero, Bulletproofs

TRABAJO FIN DE GRADO

Curso 2023/2024

UNIVERSIDAD COMPLUTENSE MADRID

FACULTAD DE CIENCIAS MATEMÁTICAS

Grado en Matemáticas

Alumna: Elena de la Fuente

Tutor: Martín Eugenio Avendaño González

Madrid, 4 de septiembre de 2024

Resumen

En este trabajo de fin de grado se presenta un estudio detallado de varios protocolos criptograficos, y la aplicación de las pruebas de conocimiento cero poniendo particular atención a los protocolos Bulletproof.
Las criptomonedas son seguras pero a menudo, no proveen de privacidad a los usuarios. En esta investigación se estudia la forma de mejorar la privacidad sin sacrificar la verificabilidad de las transacciones. El protocolo Bulletproof, mejora la privacidad de las transacciones en criptomonedas como Grin en comparación con Bitcoin. Como conclusión, Bulletproof presenta una solución viable para transacciones privadas y seguras.

Abstract

This final degree project provides a detail study of various cryptographic protocols and their application in zero-knowledge proofs, with a particular focus on the bulletproof protocol.
Cryptocurrencies, while secure, often fail to protect user privacy. This research addresses enhancing privacy without sacrificing transaction verifiability. This protocol significantly enhance transaction privacy in cryptocurrencies like Grin compared to Bitcoin. In conclusion, Bulletproof provides a viable solution for private and secure transactions.

Índice general

1. Protocolos criptográficos **4**

1.1. Funciones hash 4

1.2. Problema del logaritmo discreto 5

1.3. Firmas digitales 11

2. Pruebas de conocimiento cero **22**

2.1. Conceptos básicos 22

2.2. Interactivas y no interactivas via Fiat-Shamir 23

3. Criptomonedas **28**

3.1. Bitcoin 32

3.2. Grin 34

4. Bulletproofs **38**

4.1. Range Proof 39

5. Conclusiones **52**

Introducción

Las criptomonedas surgen como respuesta a la crisis financiera global de 2008, y en particular a la pérdida generalizada de confianza en los sistemas financieros tradicionales y los bancos asociada con estos. En este contexto histórico nace el concepto de una moneda digital no controlada por ningún banco o entidad gubernamental.

En 2009 una persona, o un grupo de personas bajo el seudónimo "Satoshi Nakamoto", pública el artículo "Bitcoin: A Peer-to-Peer Electronic Cash System", en el que se describe un proyecto para crear una especie de divisa virtual que serviría para contabilizar y transferir valor a bajo coste. De esta propuesta nació Bitcoin. [1]

Desde sus inicios, las criptomonedas han sido una herramienta usada por muchos, porque son de fácil acceso, así como una posible inversión, y no están controladas por ninguna entidad central. Por otra parte, hay que admitir que estas monedas conllevan un riesgo, pues su falta de regulación las hace muy volátiles y podrían ser usadas para actividades ilegales.

Bitcoin es la criptomoneda más popular, en parte por ser la primera que se desarrolló. En Bitcoin se usa la tecnología blockchain, de la que hablaremos más adelante. Una característica de esta tecnología es que todos los integrantes de la red puedan ver las transacciones. El lado bueno de esto es que facilita enormemente la verificación de estas, lo que convierte a blockchain en una herramienta de registro muy útil. Pero esto hace también que toda la red tenga acceso a cuánto dinero tiene cada integrante, aún cuando este utilice un seudónimo, pues es fácil identificar a los dueños de cada cuenta. En muchas ocasiones, esto puede ser un problema. Este inconveniente, junto a la popularidad de las criptomonedas, hace que se creen otras monedas digitales. Un ejemplo que estudiaremos es Grin.

En este TFG hablaremos de las pruebas de conocimiento cero, en concreto, de las conocidas como bulletproof, que en algunas criptomonedas se utilizan para que las transferencias sean públicas y verificables, pero sin desvelar las identidades de las personas ni la cantidad de dinero involucrada.

Antes de hablar de las pruebas de conocimiento cero, estudiaremos en el primer capítulo algunos protocolos criptográficos como las funciones hash y el problema del logaritmo discreto, que pueden utilizarse para generar firmas digitales.

En el segundo capítulo hablaremos de pruebas de conocimiento cero, haciendo hincapié en las pruebas no interactivas que son las que usaremos para el algoritmo bulletproof, así como de la aplicación del problema del logaritmo discreto a la construcción de dichas pruebas.

En el tercer y penúltimo capítulo estudiaremos el funcionamiento de las criptomonedas, en concreto Bitcoin y Grin. Esta última usa el algoritmo que vamos a estudiar.

Por último hablaremos de Bulletproof, lo implementaremos en Python y se explicará su funcionamiento.

Capítulo 1

Protocolos criptográficos

1.1. Funciones hash

Definición 1.1.1. *Una función hash o función resumen recibe una cadena de entrada y devuelve un valor. Este valor, llamado valor hash, sirve como una representación compacta de la cadena de entrada. Es una función computable,* $H : U \longrightarrow M$ *tal que el conjunto de entrada* U *puede tener un cardinal infinito y el conjunto de salida* M *es un conjunto de cadenas de una longitud fija, por tanto este último es un conjunto finito. Podemos decir que* H *actúa como una proyección de* U *a* M*.*

Definición 1.1.2. *Dada una función hash, decimos que existe una colisión cuando dos entradas distintas devuelven el mismo código hash.*

En las funciones hash que usaré en este TFG el conjunto de entrada tendrá un cardinal mayor que el del conjunto de salida, por eso es trivial decir que producirán colisiones.

Definición 1.1.3. *Definimos una buena función hash como aquella con colisiones difíciles de encontrar.*

Propiedades importantes de las funciones hash

- Bajo coste: Calcular la función hash necesita poco coste computacional, de memoria etc.
- Compresión: Por definición, comprime datos de longitud muy grande a una longitud más pequeña.
- Uniformidad: Dada una cadena de entrada aleatoria, la probabilidad de que el código hash sea un valor cualquiera es la misma para todo el conjunto de salida.
- Determinista: Una función hash es determinista si dada una cadena de entrada, siempre devuelve el mismo valor hash.

Propiedades para analizar la resistencia frente a colisiones

- Resistencia a la primera preimagen: Una función hash, H, tiene resistencia a la primera preimagen si cumple que dado un valor hash y es computacionalmente imposible encontrar x tal que $H(x) = y$. Esto no significa que x no exista.
- Resistencia a la segunda preimagen: Se dice que una función hash es resistente a la segunda preimagen, si dado un mensaje x_1 es computacionalmente imposible encontrar un mensaje x_2 distinto que cumpla $H(x_1) = H(x_2)$.
- Resistencia a colisiones: Una función hash es resistente a colisiones si encontrar un par de mensajes (x, y), $x \neq y$ tal que $H(x) = H(y)$, es computacionalmente imposible.

Algunos ejemplos de funciones hash, funciones SHA

Las funciones SHA (Secure Hash Algorithm), son una familia de funciones hash, aprobadas por el NIST (National Institute of Standars and Technology) como un FIPS (Federal Information Processing Standar), que son unos estándares también aprobados en Europa. Su característica principal es la no reflexividad o resistencia a la primera preimagen, es decir, no es posible determinar el mensaje dado un código hash. Entre estas funciones encontramos:

1. SHA-1: El algoritmo de la SHA-1 produce como resultado un hash de 160 bits. Este algoritmo dejó de usarse en 2017 porque se encontró una manera de crear colisiones. [2]
2. SHA-2: La SHA-2 es una familia de seis funciones criptográficas que se construyen usando el método Merkle-Damgård. Entre esas seis funciones encontramos: [3]
   a) SHA-256: devuelve un hash de 32 bytes
   b) SHA-512: devuelve un hash de 64 bytes, por tanto es el más seguro de las SHA de las que se habla en este documento.
3. SHA-3: Es el último de las SHA publicado por la NIST en 2015, aún no es muy utilizada.

1.2. Problema del logaritmo discreto

Definición del logaritmo discreto

Antes de empezar a explicar el problema del logaritmo discreto hay que definir un concepto básico.

Definición 1.2.1. *Sea G un grupo abeliano finito multiplicativo y sea g de orden n perteneciente a G. Dado un elemento x perteneciente al subgrupo generado por g, definimos el logaritmo discreto de x en base g como el menor entero positivo k, tal que:*

$$x = g^k \iff k = log_g(x)$$

Problema del logaritmo discreto (PLD)

Este problema se usa en criptografía porque es muy fácil calcular $g^k = x$, en cambio se cree que es computacionalmente díficil encontrar k dados g y x.

Dado $p > 2$ primo, podemos definir el grupo $\mathbb{Z}_p^*$ que tiene un generador g, es decir $\mathbb{Z}_p^* = \langle g \rangle$. El siguiente isomorfismo es fácil de calcular computacionalmente.

$$exp_g : (\mathbb{Z}/(p-1)\mathbb{Z}, +) \longrightarrow ((\mathbb{Z}/p\mathbb{Z})^*, \cdot)$$
$$[k]_{p-1} \mapsto [g^k]_g$$

Su función inversa es el logaritmo

$$log_g : (\mathbb{Z}/p\mathbb{Z})^* \longrightarrow \mathbb{Z}/(p-1)\mathbb{Z}$$

que se conjetura difícil de calcular computacionalmente, ya que todos los algoritmos dados para resolver el problema hasta el momento no acaban en un tiempo polinómico. Algunos de los algoritmos poco eficientes son:

- **Busqueda exhaustiva o Brute Force:**

  Dada una base, se va calculando la exponencial empezando por el 0 hasta que se encuentre el valor buscado. Normalmente se pone un valor límite para que el algoritmo acabe en algún momento. Este algoritmo es uno de los más sencillos de todos y puede ser útil de cara a resolver logaritmos de números pequeños, pero al ser de orden lineal no es útil para números grandes. Por ejemplo, quiero resolver $log_b(n) = x$ en $\mathbb{Z}/p\mathbb{Z}$:

Listing 1.1: Busqueda exhaustiva

```
#Quiero resolver logb(n) = x con b y n conocidos en Zp
# logb(n) = x sii b^x = n modp

def fuerzaBruta(b,n,p,limite):
    a = 1
    for i in range(limite):
        a = a%p
        if a == n:
            return i
        else:
            a *= b
```

- **Algoritmo de Shanks o Baby-Step Giant-Step:**

  Este algoritmo se crea con la intención de optimizar Brute Force, pero aún así su complejidad es de orden $\sqrt{p}$ y por tanto sigue sin ser una opción factible para la resolución del PLD. Se basa en la idea de encontrar dos listas de exponentes, de eso el nombre Baby-Step Giant-Step, luego buscar una coincidencia entre ellas.
  Dado un grupo cíclico G de orden N, un generador g y el problema a resolver $h = g^x$. El algoritmo empieza calculando el tamaño del paso, que se suele elegir de manera que n sea la raíz cuadrada del orden de G, es decir $n = \lfloor \sqrt{N} \rfloor + 1$.

Después calculamos las listas, la baby steps es $\{g^0, g^1, g^2, \ldots, g^n\}$ y la giant steps $\{hg^0, hg^{-n}, hg^{-2n}, \ldots, hg^{-n^2}\}$. Buscamos que haya un elemento igual en cada lista, de tal forma que llegamos a $g^i = hg^{-jn} \iff h = g^i g^{jn}$ y como $h = g^x$, llegamos a $g^x = g^{i+jn} \iff x = i + jn$.

Listing 1.2: Algoritmo de Shanks

```
1  from math import isqrt
2  from sympy import mod_inverse
3
4  def baby_step_giant_step(g, h, p):
5      # Calcular m
6      m = isqrt(p) + 1
7
8      # Precomputación (Baby-steps)
9      baby_steps = {}
10     for j in range(m):
11         baby_step = pow(g, j, p)
12         baby_steps[baby_step] = j
13
14     # Calcular el inverso multiplicativo de g^m
15     g_m = pow(g, m, p)
16     g_m_inv = mod_inverse(g_m, p)
17
18     # Búsqueda (Giant-steps)
19     for i in range(m):
20         giant_step = (h * pow(g_m_inv, i, p)) % p
21         if giant_step in baby_steps:
22             j = baby_steps[giant_step]
23             return i * m + j
24
25     return None  # Si no se encuentra una solución
```

Sobre el código, hay que tener en cuenta que la línea 21 es eficiente ya que Python utiliza una representación de datos para los diccionarios que permite determinar rápidamente si un índice aparece en él.

Criptosistemas basados en el problema del logaritmo discreto

- **Intercambio de claves Diffie-Hellmann**:
  El intercambio de claves Diffie-Hellmann es un metodo matemático de intercambio de claves por un canal inseguro o público. Tradicionalmente, para hacer un intercambio de claves seguro se requería un encuentro entre las dos personas. Este es el primer metodo en el que esto no es necesario.
  Vamos a llamar a las personas que intercambian la clave Alice y Bob. Alice decide un numero secreto a y Bob elige otro numero secreto b, Alice y Bob deciden un elemento de orden muy alto g, que pertenece a un grupo que se intercambiaran por un canal inseguro. Alice sabiendo g y a puede calcular $A = g^a$, mientras que Bob puede calcular $B = g^b$. Seguidamente Alice y Bob se intercambian A y B publicamente,

luego Bob calcula A^b y Alicie calcula B^a. Dado que $A^b = (g^a)^b = (g^b)^a = B^a$ a esto lo llamamos C y es la clave secreta.Teniendo en cuenta el problema del logaritmo discreto aunque una persona externa supiera g, A y B no podría calcular C. [4]

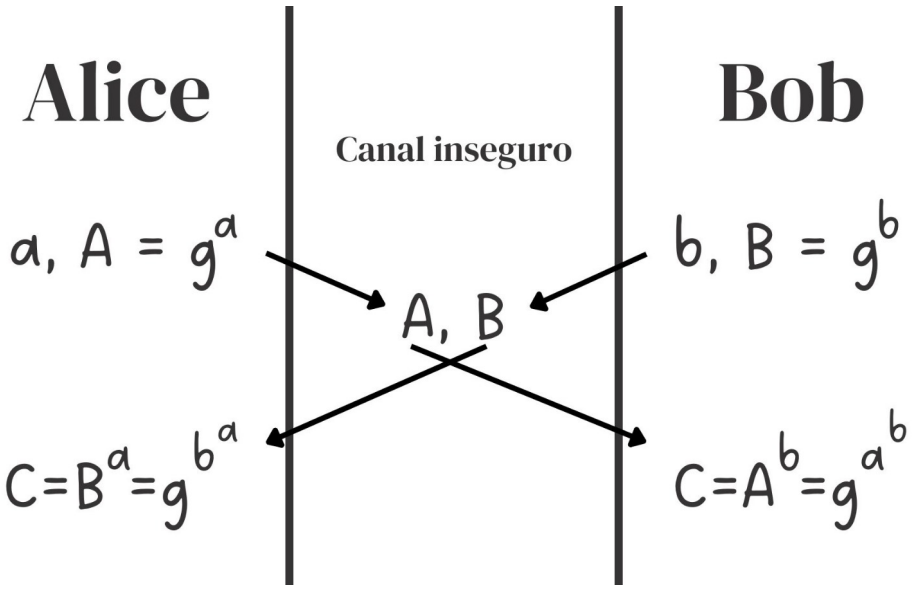

Figura 1.1: Esquema Intercambio Diffie Hellman

Voy a explicar un ejemplo concreto, supongamos que tomo $p = 23$, entonces elegiría un g que sea un generador del grupo $\mathbb{Z}_{23}^*$. Por ejemplo tomamos $g = 5$. Por su lado, Alice elige una clave privada $0 < a < 23$, $a = 6$. En cambio, Bob elige su clave privada $0 < b < 23$, $b = 15$.

Para calcular las claves públicas se hace de la siguiente manera, Alice calcula $A = g^a \pmod p$, es decir, $A = 5^6 \pmod{23} = 8$. Por otro lado, Bob calcula $B = g^b \pmod g$, por tanto $B = 5^15 \pmod{23} = 19$.

El intercambio de claves se hace por un canal público, Alice manda a Bob $A = 8$ y Bob manda a Alice $B = 19$. Entonces, Alice calcula $C = B^a \pmod p = 2$ y Bob calcula $C = A^b \pmod p = 2$.

Ahora ambas personas comparten la clave privada $C = 2$, sin que esta pueda ser deducida por nadie más.

- **Criptosistema ElGamal**: El sistema de cifrado ElGamal es un algoritmo de cifrado de clave asimétrica para criptografía de clave pública que se basa en el intercambio de claves Diffie-Hellman.

  - **Generación de clave**: Alice escoge un número primo p tal que su logaritmo no es resoluble en un tiempo asumible. Para encontrar dicho número lo que se hace computacionalmente es encontrar un q lo suficientemente grande y que cumpla $p = qz + 1$.
    Alice también elige dos números aleatorios: g, que es un generador del grupo cíclico $\mathbb{Z}_p^*$ y un número aleatorio $x \in \{0, \ldots, p-1\}$. Ahora, Alice calcula $y = g^x \pmod p$.
    Defino la clave publica como (g,p,y) y la clave secreta x.

Listing 1.3: Generación de claves cifrado ElGamal

```python
import random
import sympy

def key_generation():
    # empiezo por buscar q primo de 160 bits.
    found = False
    while not found:
       q = random.randint(2**159, (2**160)-1)
       found = sympy.isprime(q)

    # Ahora elijo p de L = 512 bits
    # Busco p = q * 2z + 1 y que sea primo,
    # tq z sea primo también y distinto de q
    L = 512
    izq = (2**(L-1)-1)//q
    der = ((2**L)-1)//q
    found = False
    while not found:
       z = random.randint(izq, der-1)
       if sympy.isprime(z):
           p = 2*q*z+1
           found = sympy.isprime(p)

    # g es un generador de del grupo ciclico de orden p-1
    found = False
    while not found:
       g = random.randint(1, p-1)
       if pow(g, 2*q, p) != 1 and pow(g, 2*z, p) \
           != 1 and pow(g,q*z,p) != 1:
           found = True

    # x es la clave secreta
    x = random.randint(2,p-2)

    # Calcular y = g^x(modp)
    y = pow(g,x,p)

    # La clave publica es pk = (p,g,y)
    # La clave secreta sk = x
    pk = (p,g,y)
    sk = x
    return (pk, sk)
```

- **Cifrado:** Dado un mensaje $m \in \{1, \ldots, p-1\}$ quiero convertirlo en un elemento de $\mathbb{Z}_p$. Bob elige aleatoriamente $b \in \{1, \ldots, p-1\}$, número secreto. También calcula $y_1 = g^b \pmod{p}$ y $y_2 = (y^b \cdot m) \pmod{p}$ y devuelve el mensaje $m = (y_1, y_2)$.

Listing 1.4: Cifrado ElGamal

```python
def partirnum(num,p):
    n = 10**p
    lista = []

    while num > n:
        l = num % n
        lista = [l] + lista
        num = num // n
    lista = [num] + lista

    return lista

def conv(m,p):
    # Primero pasamos la cadena de caracteres a un numero
    n = int.from_bytes(m, byteorder ='big', signed = False)
    l = partirnum(n,p)

    return l

def cifradoEG(g,p,y,m):
    b = random.randint(1,p-1)
    m1 = conv(m,math.floor(math.log(p,10)))
    n = len(m)
    y1 = pow(g,b,p)
    y2 =[((pow(y,b,p))* i )%p for i in m1]
    c = (y1,y2)

    return (c,n)
```

- **Descifrado:**
  Para descifrar el mensaje, hay que calcular $y_1^{-a}y_2 \pmod p$ aprovechando que se cumple:

$$\begin{aligned} y_1^{-a}y_2 \pmod p &= (g^b)^{-a}k^b m \pmod p \\ &= g^{-ab}(g^a)^b m \pmod p \\ &= (g^a)^{-b}(g^a)^b m \pmod p \\ &= m \pmod p \end{aligned}$$

Listing 1.5: Descifrado ElGamal

```python
def juntarnum(l,p):
    n = 0
    while len(l)>1:
        n = n*(10**p) + l[0]*(10**p)
        l.pop(0)
    n = n + l[0]
```

```
7
8       return n
9
10  def dconv(l,n,p):
11      # n el la longitud del mensaje
12      m = juntarnum(l,p)
13
14      return m.to_bytes(n,byteorder = 'big', signed = False)
15
16  def descifradoEG(y1,y2,x,p,n):
17      m = [( pow(y1,-x,p) * i ) % p for i in y2]
18
19      return dconv(m,n,math.floor(math.log(p,10)))
```

- **Ejemplo:** Para generar la clave buscamos un numero primo $q = 52673$, con este calculamos otro numero primo de la siguiente manera $p = 2 \cdot z \cdot q + 1$, tomando $z = 3 \cdot 7^2$, así que $p = 2 \cdot 3 \cdot 7^2 \cdot 52673 = 15485863$. Busco un generador de $\mathbb{Z}_p^*$, $g = 7$. Ahora eligimos la clave privada $x \in 0, \ldots, p-1$, $x = 21702$. Calculamos la clave privada de la siguiente manera $y = g^x \pmod p = 7^{21702} \pmod{15485863} = 8890431$. La clave pública es (g, p, y) y la privada es x.
  El cifrado se hace de la siguiente manera. Tomamos un mensaje $m = b'hey'$, que usando el código de Python llegamos a que su codificación, en este caso, es $m = 6841721 \in [1, p-1]$. Elegimos un numero random $b \in [1, p-1]$, tomamos $b = 480$. Ahora calculamos:
  - $y_1 = g^b \pmod p = 7^{480} \pmod{15485863} = 12001315$
  - $y_2 = y^b \cdot m \pmod p = (8890431^{480} \cdot 6841721) \pmod{15485863} = 6568011$

  Alice envia a Bob lo siguiente (y_1, y_2)
  Para descifrar el mensaje Bob calcula lo siguiente:

  $$y_1^{-x} \cdot y_2 \pmod p = 12001315^{-21702} \cdot 6568011 \pmod{15485863} = 6841721$$

  Ahora Bob ha encontrado la codificación del mensaje, con 6841721 puede ir a Python y descifrar el mensaje haciendo el camino contrario.

1.3. Firmas digitales

Una firma digital es un mecanismo criptográfico que permite a una entidad o persona mandar un mensaje a un receptor, garantizando que ese mensaje es suyo y no ha sido modificado. La validez de la firma se apoya en la imposibilidad de falsificarla, siempre y cuando no se conozca la clave secreta del firmante. Hay que tener en cuenta que las firmas digitales sirven exclusivamente para firmar, no para cifrar información.
Para esto necesitamos que las firmas digitales cumplan ciertas características:

- Unicidad: Solo puede ser generada por el firmante.
- Infalsificable: La firma solo se puede generar si se conoce la clave privada, por tanto solo la puede crear el firmante.

- Verificables: Cualquiera que disponga de la clave pública del firmante debe ser capaz de verificar la firma de manera fácil.
- Innegable: Una vez firmado, el firmante no puede ser capaz de negar su autenticidad.
- Viable: Las firmas deben ser fáciles de generar por el firmante, además el tiempo computacional es reducido.

Las firmas que vamos a estudiar son algoritmos asimétricos, es decir, se utilizan dos claves, una pública y una privada. Ambas claves estan conectadas, siendo la clave privada, únicamente conocida por el firmante, usada para firmar y la clave pública, conocida por todos, usada para verificar.

DSA (Digital Signature Algorithm)

El DSA es un sistema criptográfico para la verificación de firmas basado en el problema del logaritmo discreto. En este apartado se describe su funcionamiento y se implementa en python.

1. **Generación de claves:**
   Para generar la clave necesitamos elegir un primo q de 160 bits. Con ese número buscamos un primo p de L bits, donde $512 \leq L \leq 1024$, y que cumpla $p = qz + 1$ con $z \in \mathbb{N}$. Después, elegimos un h que cumpla $1 < h < p-1$ y $g = h^z \pmod{p} > 1$. Por último, se elige un x de manera aleatoria que cumpla $1 < x < q-1$. Y se calcula $y = g^x \pmod{p}$.

Listing 1.6: Generación de claves DSA

```
import random
import hashlib
import sympy

# Firma digital DSA
def key_generation():
    # empiezo por buscar q primo de 160 bits.
    found = False
    while not found:
        q = random.randint(2**159, (2**160)-1)
        found = sympy.isprime(q)

    # ahora elijo p de L = 512 bits
    L = 512

    izq = (2**(L-1)-1)//q
    der = ((2**L)-1)//q
    found = False
    while not found:
        z = random.randint(izq, der-1)
        p = q*z+1
        found = sympy.isprime(p)
```

```
23
24      # ahora elijo h en (1,p-1) tal que g = h**z mod p > 1
25      found = False
26      while not found:
27          h = random.randint(2,p-2)
28          g = pow(h, z, p)
29          found = g > 1
30
31      # elegir x en (1,q-1) aleatorio y hacer y = g**x mod p
32      x = random.randint(2,q-2)
33      y = pow(g, x, p)
34      pk = (p,q,g,y)
35      sk = x
36      return (pk, sk)
```

La función devuelve como datos públicos $pk = (p, q, g, y)$ y como clave privada $sk = x$.

2. **Firma:**
Dadas la clave pública y la clave privada, elegimos un número aleatorio k tal que $1 < k < q$. Con k se calcula $r = (g^k \pmod p) \pmod q$, después $s = k^{-1}(H(m)+rx) \pmod q$, siendo $H(m)$ la función SHA-1 del mensaje m. Hay que tener en cuenta que si $r = 0$ o $s = 0$ se repite el proceso.

Listing 1.7: Firma DSA.py

```
1  def H(m):
2      # Función hash SHA1
3      # m tiene que ser de la forma b'mensaje'
4      return int(hashlib.sha1(m).hexdigest(), base=16)
5
6  def firmaDSA(p, q, g, x, m):
7      k = random.randint(2, q-1)
8      r = (pow(g,k,p))%q
9      s = (pow(k,-1,q) * (H(m) + x * r)) % q
10     return (r,s)
```

La función firma DSA devuelve el par (r, s) siendo esta la firma.

3. **Verificación:**
Cuando una persona recibe un mensaje con una firma digital, debe ser capaz de verificarla. Se hace de la siguiente manera: se calculan $w = s^{-1} \pmod q$, $u_1 = H(m)w \pmod q$, $u_2 = rw \pmod q$ y $v = (g^{u_1}y^{u_2} \pmod p) \pmod q$. Esto debe cumplir $v = r$, sino la firma no es válida.

Listing 1.8: Verificación DSA.py

```
1  def verDSA(r, s, m, p, q, g, y):
2      if (1<r<(q-1)) and (1<s<(q-1)):
3          w = pow(s,-1,q)
```

```
        u1 = (H(m)*w)%q
        u2 = (r*w)%q
        v = ((powmod(g, u1, p) * powmod(y,u2,p)) % p) % q
        return v==r
    else:
        return False
```

4. **Ejemplo concreto:**

   Supongamos que Alice quiere firmar el mensaje $m = b'Hola'$, para simplificar los calculos usaremos numeros más pequeños de los que se debería.

   Para generar las claves, Alice toma $q = 13$, calcula $p = q \cdot z + 1$, con $z = 3834$, $p = 49843$. Elige $h = 17248$ y calcula $g = h^z \pmod p = 39245$. Para elegir la clave privada, Alice elige un numero aleatorio del intervalo $(1, q-1)$, $x = 8$. Teniendo esto, calcula la clave pública $y = g^x \pmod p = 23157$. Las claves públicas son $pk = (p, q, g, y) = (49843, 13, 39245, 23157)$ y la clave privada $sk = 8$.

   Para firmar, Alice elige un $k \in (1, q)$, $k = 9$. Después, calcula la firma

   - $r = g^k \pmod q = 11$
   - $s = [(k^{-1} \pmod q) \cdot (H(m) + x \cdot r)] \pmod q = 2$

   Siendo $H(m)$ la función hash del mensaje m. La firma del mensaje es $(r, s) = (11, 2)$

   Dado el mensaje, la firma y las claves públicas, el que recibe el mensaje, hace los siguientes calculos. Primero calcula $w = s^{-1} \pmod q = 7$, luego $u_1 = 4$ y $u_2 = 12$, como se ha explicado anteriormente. Y verifica que v es igual a r. Calcula $v = [(g^{u_1} \pmod p) \cdot (y^{u_2} \pmod p)] \pmod q = 11$ y se cumple que es igual a r. Esto quiere decir que el mensaje no ha sido modificado desde que ha salido de Alice.

Proposición 1.3.1. *Suponiendo que el firmante es honesto y el texto no ha sido modificado, la persona que recibe el mensaje quedará convencida de su validez. Es decir, se cumplirá $r = v$.*

Demostración. La persona que recibe el mensaje tiene (r, s) y el mensaje. Suponiendo que la función hash H que se usa en el algoritmo es resistente a colisiones. Despejando k de $s = k^{-1}(H(m) + rx) \pmod q$ y sustituyendo s^{-1} por w llegamos a $k = H(m)w + xrw$.

Buscamos que se cumpla $r = v$:

$$r = g^k = g^{H(m)w} g^{xrw} = g^{H(m)w} y^{rw} = g^{u_1} y^{u_2} = v$$

□

ElGamal

El esquema de firma ElGamal permite confiar en la autenticidad de un mensaje enviado por un canal de comunicación inseguro. Para generar esta firma se necesita: una función hash H resistente a colisiones, un número primo p muy grande tal que el cómputo del logaritmo discreto módulo p sea difícil y un generador pseudoaleatorio g para el grupo $\mathbb{Z}_p^*$.

1. **Generación de claves:**
   Este algoritmo lo realiza el firmante y no comparte con nadie su clave secreta. Primero buscamos un número aleatorio x tal que $1 < x < p-1$. Con eso calculamos $y = g^x \pmod{p}$. Definimos como clave pública (p, g, y) y como clave secreta x.

Listing 1.9: Generación de claves EG.py

```
import random
import hashlib
import sympy
import math

def key_generation():
    # empiezo por buscar q primo de 160 bits.
    found = False
    while not found:
        q = random.randint(2**159, (2**160)-1)
        found = sympy.isprime(q)

    # Ahora elijo p de L = 512 bits
    # Busco p = q * 2z + 1 y que sea primo,
    # tq z sea primo también y distinto de q
    L = 512
    izq = (2**(L-1)-1)//q
    der = ((2**L)-1)//q
    found = False
    while not found:
        z = random.randint(izq, der-1)
        if sympy.isprime(z):
            p = 2*q*z+1
            found = sympy.isprime(p)

    #  g es un generador de del grupo ciclico de orden p-1
    found = False
    while not found:
        g = random.randint(1, p-1)
        if pow(g, 2*q, p) != 1 \
            and pow(g, 2*z, p) != 1 \
                and pow(g,q*z,p) != 1:
            found = True

    # x es la clave secreta
    x = random.randint(2,p-2)

    # Calculo y
    y = pow(g,x,p)

    #La clave publica es pk = (p,g,y), la clave secreta sk = x
    pk = (p,g,y)
    sk = x
    return (pk, sk)
```

2. **Firma:**
Para firmar, buscamos un k aleatorio que cumpla $0 < k < p-1$ y k coprimo con $p-1$. Luego calculamos $r = g^k \pmod{p}$ y $s = (H(m) - xr)k^{-1} \pmod{p-1}$ Si $s = 0$, entonces se vuelve a empezar el proceso, en caso contrario se devuelve el par (r, s) siendo este la firma digital de m.

Listing 1.10: Firma EG.py

```
def H(m):
    # Función hash SHA-512
    # m tiene que ser de la forma b'mensaje'
    return int(hashlib.sha512(m).hexdigest(), base=16)

def firmaEG(m, p, g, x):
   k = 0
   while math.gcd(k, p-1) != 1:
       k = random.randint (1, p-2)

   r = pow(g,k, p)
   s = (((H(m) - (x*r)) % (p-1)) * pow(k,-1,p-1)) % (p-1)
   if s == 0 :
       return firmaEG(m,p,g,x)
   else:
       return (r,s)
```

3. **Verificación:**
El verificador acepta el mensaje si y solo si se cumple que $0 < r < p$, $0 < s < p-1$ y $g^{H(m)} = y^r r^s \pmod{p}$.

Listing 1.11: Verificación EG.py

```
def verificarEG(p,g,y,r,s,m):
    if (0<r<p) and (0<s<(p-1)):
        # Resolver la congruencia:
        # g^H(m) congruente ((y^r)*(r^s))modp
        u1 = pow(g,H(m),p)
        u2 = (pow(y,r,p) * pow(r,s,p)) % p
        return u1 == u2

    else:
        return False
```

4. **Ejemplo concreto:**
Usare el mismo ejemplo, Alice quiere mandar el mensaje $m = Hola$ a Bob. Primero la generación de claves, tomando $q = 17$ y $z = 7069$, calculamos $p = 2 \cdot q \cdot z + 1 = 240347$. Luego buscamos un generados g del grupo ciclico de orden $p - 1 = 240346$, por tanto, tomamos $g = 168784$. Alice elige una clave privada $x \in (1, p-1)$, $x = 197757$, con la clave privada calcula la clave privada $y = g^x \pmod{p} = 48859$. Los datos públicos son los siguientes; $p_k = (p, g, y) = (240347, 168784, 48859)$ y la clave privada $s_k = x = 197757$.

Para firmar, Alice elige una $k \in (0, p-1)$, $k = 75523$ y con esto calcula lo siguiente:

- $r = g^k \pmod p = 17554$
- $s = [(H(m) - x \cdot r) \pmod{p-1} \cdot k^{-1} \pmod{p-1}] \pmod{p-1} = 28581$

Por último, Bob verifica la firma calculando lo siguiente:

- $u_1 = g^{H(m)} \pmod p = 126018$
- $u_2 = (y^r \pmod p \cdot r^s \pmod p) \pmod p = 126018$

Como se cumple que $u_1 = u_2$, la firma de verifica.

Proposición 1.3.2. *Suponiendo que el firmante es honesto y el texto no ha sido modificado, la persona que recibe el mensaje quedará convencida de su validez. Es decir, se cumplirá* $g^{H(m)} = r^s y^r \pmod p$.

Demostración. De s, despejamos $H(m) = sk + xr$, como cualquier persona que tenga el mensaje puede calcular $H(m)$, y asumiendo que la función hash es resistente a colisiones tenemos:

$$g^{H(m)} = g^{sk} g^{xr} = r^s y^r \pmod p$$

□

Firma de Schnorr

La firma de Schnorr es un esquema de firma digital conocido por su simplicidad y eficiencia, basado en la dificultad del problema del logaritmo discreto en grupos cíclicos. Genera firmas cortas y eficientes mediante la combinación de un valor aleatorio y una función hash del mensaje. Por otro lado, la firma de Schnorr multiparty extiende este esquema permitiendo que múltiples partes colaboren en la firma de un mensaje.

Igual que en las firmas anteriores, se necesita una función hash H resistente a colisiones, un número p primo, y un g que sea generador de $\mathbb{Z}_p^*$.

1. **Generación de claves**:
   Primero el firmante busca su clave privada $x \in [1, p-1]$, y genera su clave pública $y = g^{-x} \in \mathbb{Z}_p^*$.

Listing 1.12: Generación de claves firma de Schnorr

```
import random
import hashlib
import sympy
from gmpy2 import powmod

def key_generation():
    found = False
    while not found:
        q = random.randint(2**159, (2**160)-1)
        found = sympy.isprime(q)
```

```
    L = 512
    izq = (2**(L-1)-1)//q
    der = ((2**L)-1)//q
    found = False
    while not found:
        z = random.randint(izq, der-1)
        p = q*z+1
        found = sympy.isprime(p)

    found = False
    while not found:
        h = random.randint(2,p-2)
        g = pow(h, z, p)
        found = g > 1
    x = random.randint(2,q-2)
    y = pow(g, -x, p)
    pk = (p,q,g,y)
    sk = x
    return (pk, sk)
```

2. **Firma**:
   La firma la genera el firmante. Dado el mensaje M, se genera aleatoriamente una $k \in [1, q-1]$, con esto se calcua $r = g^k$. Por último, genera la firma $s = k + x * e$ con $e = H(r||M)$. La firma es (r, s).
   Hay que tener en cuenta que $||$ significa concatenación.

Listing 1.13: Firma Schnorr

```
def H(m):
    return int(hashlib.sha1(m).hexdigest(), base=16)

def firmaSchnorr(p, q, g, x, m):
    k = random.randint(2, q-1)
    r1 = pow(g,k,p)
    r = str(r1).encode()
    m1 =  r + m
    e = H(m1) % q
    #e = H(m1)
    s = (k + x* e)%p

    return (r1,s)
```

3. **Verificación**:
   El verificador solo acepta el mensaje si se cumple que $e_v = H(r_v||M)$, con $r_v = g^s y^e$

Listing 1.14: Verificación Firma Schnorr

```
def verSchnorr(r, s, m, p, q, g, y):
```

```
    e = H(str(r).encode()+m) %q
    rv = (pow(g,s,p)*powmod(y,e,p))%p
    return r==rv
```

Proposición 1.3.3. *Suponiendo que el firmante es honesto y el texto no ha sido modificado, la persona que recibe el mensaje quedará convencida de su validez. Es decir, se cumplirá* $e_v = e$.

Demostración. Se cumple que $e_v = e$ si y solo si $H(r_v||M) = H(r||M)$ y esto solo se cumple cuando $r_v = r$. Como se cumple $r_v = g^s y^e = g^{k+xe} g^{-xe} = g^k = r$, entonces $e_v = e$.

□

Firma de Schnorr multiparty

En el caso de la firma de Schnorr multiparty, cada firmante genera su propia firma (r_i, s_i) y la firma final es (R, s) con $R = \prod_{i=1}^{n} r_i$ y $s = \sum_{i=1}^{n} s_i$.

1. **Generación de claves**:
   Cada firmante elige una clave privada $x_i \in [1, p-1]$, luego genera su clave pública correspondiente $y_i = g^{x_i} \in \mathbb{Z}_p^*$. Ahora, cada uno comparte su clave pública y se calcula la clave público agregada $Y = \prod_{i=1}^{n} y_i \in \mathbb{Z}_p^*$.

Listing 1.15: Generación de claves firma de Schnorr Multiparty

```
#Firma de Schnorr multiparty
import random
import hashlib
import sympy
from gmpy2 import powmod

def key_generation(n):
   # empiezo por buscar q primo de 160 bits.
   found = False
   while not found:
       q = random.randint(2**159, (2**160)-1)
       found = sympy.isprime(q)
   # ahora elijo p de L = 512 bits
   L = 512
   izq = (2**(L-1)-1)//q
   der = ((2**L)-1)//q
   found = False
   while not found:
       z = random.randint(izq, der-1)
       p = q*z+1
       found = sympy.isprime(p)
```

```
    # ahora elijo h en (1,p-1) tal que g = h**z mod p > 1
    found = False
    while not found:
        h = random.randint(2,p-2)
        g = pow(h, z, p)
        found = g > 1
    # elegir x en (1,q-1) aleatorio y hacer y = g**(x) mod p
    lx = []
    Y=1
    for i in range(0,n):
        x = random.randint(2,q-2)
        lx.append(x)
        y = pow(g, x, p)
        Y =( Y*y )%p

    pk = (p,q,g,Y)
    sk = lx
    return (pk, sk)
```

2. **Firma:**
   Igual que antes, cada participante elige un $k_i \in [1, q-1]$, luego calcula $r_i = g^{k_i}$ y se comparte. A continuación se calcula $R = \prod_{i=1}^{n} r_i$.

   Con esto, se calcula $e = H(m||R)$, después cada participante calcula su clave privada $s_i = k_i + e \cdot x_i$ y se comparte para generar la firma agregada $s = \sum_{i=1}^{n} s_i$

Listing 1.16: Firma Schnorr Multiparty

```
# Firma
def H(m):
    return int(hashlib.sha1(m).hexdigest(), base=16)

def firmaSchnorr(p, q, g, lx, m,n):
    lk = []
    R=1
    for i in range(0,n):
        k = random.randint(2, q-1)
        lk.append(k)
        r1 = pow(g,k,p)
        R = R*r1
    r = str(R).encode()
    m1 =  r + m
    e = H(m1) % q
    S= 0
    for i in range(0,n):
        s = (lk[i] + lx[i]* e)%p
        S = S+s
    return (R,S)
```

3. **Verificación:** El verificador conoce las siguientes variables:

   - La firma: (R, s)
   - Clave pública agregada: Y
   - Mensaje: m
   - Generador del grupo: g

   Con lo anterior se puede calcular $e_v = H(m||R)$ y tomamos la firma como válida si se cumple que $g^s = R \cdot Y^{e_v}$.

Listing 1.17: Verificación Firma Schnorr Multiparty

```
# Verificación
def verSchnorr(R, S, m, p, q, g, Y):

    e = H(str(R).encode()+m) %q
    a = pow(g,S,p)
    b = (R * pow(Y,e,p))%p

    return a==b
```

Proposición 1.3.4. *Suponiendo que el firmante es honesto y el texto no ha sido modificado, la persona que recibe el mensaje quedará convencida de su validez.*

Demostración. Dado s, podemos calcular g^s, e_v y $R \cdot Y^{e_v}$. Ahora comprobamos que estos dos valores son lo mismo:

$$g^s = g^{\sum_{i=1}^{n} s_i} = \prod_{i=1}^{n} g^{k_i + e \cdot x_i} = \prod_{i=1}^{n} g^{k_i} \prod_{i=1}^{n} g^{e \cdot x_i} = \prod_{i=1}^{n} r_i (\prod_{i=1}^{n} g^{x_i})^e = R \cdot Y^e$$

□

Capítulo 2

Pruebas de conocimiento cero

2.1. Conceptos básicos

Según la criptografía, las pruebas de conocimiento cero son un método por el cual una persona puede demostrar a otra que determinada declaración es verdadera, sin revelar ninguna información adicional. Hay que tener en cuenta que estas pruebas se basan en la probabilidad y también en el supuesto de que los participantes son honrados, es decir, no hacen trampas. Las pruebas de conocimiento cero deben satisfacer:

1. Completitud: Si la declaración es verdadera, un verificador honrado será convencido del hecho por un probador honesto.
2. Solidez: La probabilidad de que siendo una declaración falsa, esta se de por verdadera es casi nula.
3. Conocimiento cero: Si la declaración es verdadera, ningún verificador aprende nada más del hecho de que la declaración sea verdadera.

Ahora voy a explicar dos casos hipotéticos de pruebas de conocimiento cero, que se usan para explicar la lógica detrás de estas pruebas. En lo que sigue, llamaremos Alice al probador y Bob al verificador.

- **Amigo Daltónico**
  Bob tiene dos bolas iguales salvo por el color, una bola roja y otra verde, al ser daltónico no es capaz de diferenciarlas. Alice quiere demostrar a Bob que ella es capaz de diferenciarlas, sin decirle cual es verde y cual roja.
  Bob tiene la bola roja en una mano y la verde en la otra, se las enseña a Alice, se pone las manos detrás de la espalda y las cambia o no de mano. Ahora se las enseña a Alice y ella tiene que decir si las ha cambiado de mano o no. Si Alice falla, es porque no es capaz de ver las diferencias, aquí se acabaría el experimento. En caso contrario, Alice podría haber tenido suerte o estar diciendo la verdad sobre ser capaz de ver colores. Este proceso se repite las veces suficientes hasta que Bob esté convencido o hasta que Alice falle una vez.

Si Alice no fuera capaz de diferenciarlas tendría un 50 % de probabilidades de acertar. Repitiendo este proceso, la probabilidad de acertar siempre tiende a cero.
Al acabar el proceso, Bob solo ha aprendido que Alice es capaz de ver colores pero sigue sin poder diferenciar las bolas.

- **La cueva de Alibaba**
  Alice tiene el código para abrir una puerta en una cueva, con forma de anillo. Bob quiere saber si esto es verdad. Nombran cada camino de entrada de la cueva por A ó B. Bob espera fuera mientras Alice entra. Alice coje cualquier camino sin que Bob sepa por donde ha entrado. Bob dice A ó B refiriéndose por el camino que quiere que vuelva Alice. Como Alice conoce la contraseña, se puede mover libremente y escogerá el camino que ha dicho Bob. En el caso de que Alice no conociese la contraseña, solo tendría un 50 % de probabilidades de acertar. Repitiendo este proceso suficientes veces, la probabilidad de que Alice acierte sin conocer la contraseña es casi cero. Al final Bob llega a la conclusión de que Alice muy probablemente tenga la clave de la puerta pero él no sabe la contraseña.

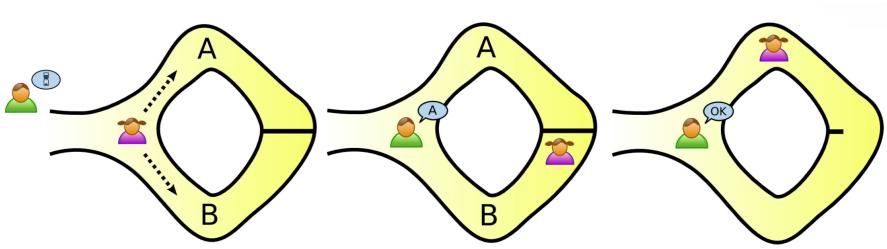

Figura 2.1: Ejemplo cueva de Alibaba [5]

Definición 2.1.1. *Un commitment o compromiso es un mecanismo criptográfico que permite a una parte demostrar de manera convincente a otra parte que dispone de cierta información, sin revelar la información en sí y garantizando que no pueda ser alterada posteriormente. [6]*

Un compromiso se podría comparar a cerrar una caja fuerte virtual con los datos dentro y mostrar a la otra parte que tienes la capacidad de desbloquearla sin mostrar lo que hay dentro. Este concepto es fundamental para garantizar la privacidad y la seguridad en situaciones en las que se necesita demostrar la posesión de cierta información sin exponerla, como en protocolos de autenticación y transacciones financieras.

2.2. Interactivas y no interactivas via Fiat-Shamir

Algoritmos interactivos

En las pruebas de conocimiento cero interactivas, tanto el probador como el verificador, necesitan estar presentes durante la ejecución del protocolo. Las pruebas de conocimiento cero suelen tener la misma estructura:

- El probador genera un mensaje de compromiso que no se puede modificar, indicando que conoce el secreto.
- El verificador devuelve un desafío al probador, sin ver el compromiso. Este desafío suele ser aleatorio.
- El probador envía una respuesta al verificador. El cálculo de la respuesta necesita tener en cuenta el compromiso, el desafío y el secreto.

Este proceso se suele repetir varias veces para asegurar la verificación.

- **Algoritmo de identificación de Schnorr:**

  Este algoritmo, como su propio nombre indica, se puede usar para demostrar que la clave secreta de un algoritmo de cifrado ElGamal es tuya, sin revelar la clave secreta.

  Dados los parametros públicos, (p, q, g, t), con p primo suficientemente grande $p \approx 2^{1024}$, q primo grande divisor de $p-1$, $q \approx 2^{160}$, $g \in \mathbb{Z}_p^*$ tiene orden q, por lo que es un generador de $G_q =< g >$ siendo este un subgrupo de $\mathbb{Z}_p^*$ y por último t es un parametro de seguridad tal que $q > 2^t$. Luego la clave secreta se calcula escogiendo de manera aleatoria $x \in \mathbb{Z}_q = \{a : a \in [0, q-1]\}$.[7]

  Algoritmo:
  Alice calcula $y = g^{-x} \pmod p$, siendo y también público, luego elige $c \in \mathbb{Z}_q$ y calcula $w = g^c \pmod p$. Alice manda w a Bob.
  Bob envía a Alice un commitment aleatorio $e \in \mathbb{Z}_q$ y $r \in [1, 2^t]$.
  Alice calcula $s = c + xe \pmod p$ y lo manda a Bob.
  Por último, Bob verifica la identidad de p verificando la siguiente ecuación $w = g^s y^e \pmod p$.

Listing 2.1: Schnorr

```
import random
import sympy

# El numero secreto es x
def generar():

    #Encontrar q primo de 10 digitos
    found = False
    while not found:
      q = random.randint(999999999, 9999999999)
      found = sympy.isprime(q)

    # Encontrar p primo de la forma p=q*2z+1 con z primo
    found = False
    while not found:
      z = random.randint(99999, 999999)
      if sympy.isprime(z):
          p = q * 2 * z + 1
          found = sympy.isprime(p)

```

```
    # Encontrar g generador de el subgrupo cíclico de orden q
    found = False
    while not found:
       g = random.randint(2, p-1)
       if pow(g, q, p) == 1:
            found = True

    # Encontrar a t, parametro de seguridad, tq q > 2**t
    found = False
    while not found :
        t = random. randint(2,30)
        found = q > 2**t

    # Generar x totalmente random
    x = random.randint(2,q-2)

    # Generar y
    y = pow(g,-x,p)

    return (p,q,g,t,x,y)

# Ejemplo
(p,q,g,t,x,y) = generar()

# 1. Alice busca c perteneciente Zq y calcula w = g^c (modp)
c = random.randint(0,q-1)
w = pow(g,c,p)

# 2. Bob escoge de forma aleatoria e perteneciente Zq
# y 1 <= r <= 2^t
e = random.randint(0,q-1)

# r = random.randint(1,2**t)

# 3. Alice calcula s = c + x*e (modq)
s =(c + x*e) % q

# 4. Se tiene que verificar que w = g^s * y^e (modp)
print( (pow(g,s, p)*pow(y,e,p))%p == w )
```

Algoritmos no interactivos

Las pruebas de conocimiento cero no interactivas, también conocidas NIZKP (Non Interactive Zero Knowledge Proof), no necesitan que el verificador ni el probador estén presentes durante la ejecución del protocolo. El probador genera una transcripción del protocolo de tal forma que el verificador pueda verificarlo más tarde.

- **Heurística de Fiat-Shamir** La heurística de Fiat-Shamir es una técnica crip-

tográfica para obtener una prueba de conocimiento cero no interactiva a partir de una prueba de conocimiento cero interactiva. Si la prueba interactiva es un protocolo de identificación, entonces la versión no interactiva puede ser usada directamente como una firma digital. [8] [9]

Mecanismo
El probador genera un mensaje de compromiso o commintment indicando que conoce el secreto. Luego toma el compromiso y más información como entradas y devuelve un desafío aplicando alguna función hash. Por último, el probador calcula la respuesta y envía la transcripción que incluye el compromiso, el desafío y la respuesta al verificador.

Listing 2.2: Fiat-Shamir EG.py

```
1
2  import random
3  import sympy
4  import hashlib
5
6  # Generar un numero primo de 10 digitos
7
8  def generar():
9
10     #Encontrar q primo de 10 digitos
11     found = False
12     while not found:
13       q = random.randint(999999999, 9999999999)
14       found = sympy.isprime(q)
15
16     # Encontrar p primo de la forma p=q*2z+1 con z primo
17     found = False
18     while not found:
19       z = random.randint(99999, 999999)
20       if sympy.isprime(z):
21           p = q * 2 * z + 1
22           found = sympy.isprime(p)
23
24     # Encontrar g generador de el subgrupo cíclico de orden q
25     found = False
26     while not found:
27        g = random.randint(2, p-1)
28        if pow(g, q, p) == 1:
29            found = True
30
31     # Encontrar a t, parametro de seguridad, tq q > 2**t
32     found = False
33     while not found :
34         t = random. randint(2,30)
35         found = q > 2**t
36
37     # Generar x totalmente random
38     x = random.randint(2,q-2)
```

```
39
40      # Generar y
41      y = pow(g,-x,p)
42
43      return (p,q,g,t,x,y)
44
45  # Ejemplo
46  (p,q,g,t,x,y) = generar()
47
48  # Alice busca c perteneciente a Zq y calcula w = g^c (modp)
49  c = random.randint(0,q-1)
50
51  w = pow(g,c,p)
52
53  # Alice calcula e = H(g|y|t)
54  m = str(g)+str(y)+str(t)
55  m1 = b'm'
56  e = int(hashlib.sha1(m1).hexdigest(), base=16)
57
58  # Alice calcula s = c+xe (modq)
59  s =(c + x*e) % q
60
61  # Bob recive w y s. Comprueba que se cumple w = g^s * y^e (modp)
62  print( (pow(g,s, p)*pow(y,e,p))%p == w )
```

Capítulo 3

Criptomonedas

En este capítulo explicaré de manera muy general los conceptos básicos para la comprensión del funcionamiento de las criptomonedas. Seguidamente, describiré como funcionan Bitcoin y Grin, que son dos de las criptomonedas más usadas.

Una de las principales diferencias entre estas dos monedas es la privacidad. Algunos expertos dicen que bitcoin es una red basada en seudónimos, por que, al tener un sistema de transacciones completamente público, se podría decir que no es seguro. Una persona externa al sistema podría acceder a la información de los usuarios. Por eso se han creado varias opciones para hacer una criptomoneda totalmente anónima. Grin se basa en los algoritmos de conocimiento cero explicados en el PDF llamado Mimblewimble para hacer las transacciones seguras, anónimas y además verificables. [10]

Definición 3.0.1. *Un nodo en este contexto es un ordenador conectado a la red de la criptomoneda, que sigue las reglas y comparte información. Los nodos participan en la red de forma directa y pueden hacer y retransmitir transacciones al igual que minar y validar bloques y transacciones.*

Definición 3.0.2. *Los dispositivos de almacenamiento de criptoactivos, tales como las carteras o monederos digitales, desempeñan un papel crucial al posibilitar la realización de transacciones de pagos en activos digitales. Dada la naturaleza completamente digital de estas monedas, carecen de existencia física, y lo que se guarda en dichos dispositivos no es la propia moneda, sino las claves pública y privada asociadas a la misma.*

En este contexto, las criptomonedas se conceptualizan como registros de transacciones previas que están inmersas en una cadena de bloques o blockchain. En consecuencia, la autorización para efectuar gastos de la criptomoneda radica en la posesión de las claves pública y privada correspondientes a la última transferencia realizada.

Transacciones

En Peer-to-peer se define una moneda electrónica como una cadena de firmas digitales. Cada transacción contiene el código hash de la transacción anterior con la clave pública de la persona que recibe la moneda y por último se añade la firma del propietario anterior.

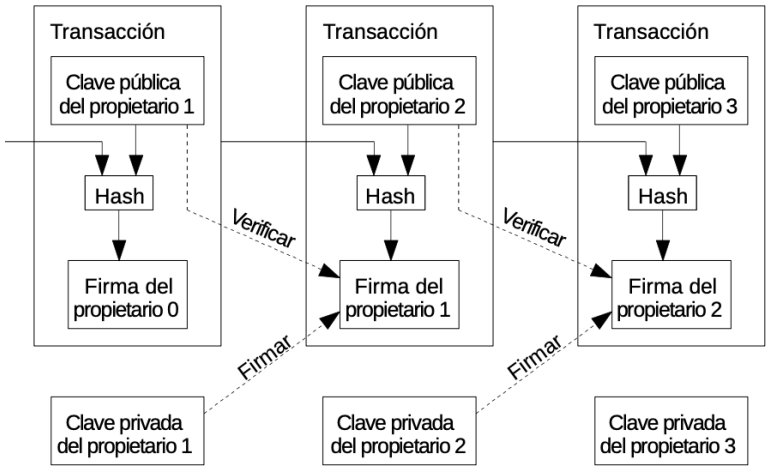

Figura 3.1: Esquema sacado de Peer 2 Peer

Definición 3.0.3. *UTXOs o Unspent Transaction Output es la lista de monedas que aún no se han gastado. Cada UTXO tiene un valor distinto.*

Para evitar que un usuario gaste dos veces la misma moneda, se hacen públicas todas las transacciones de la red. El beneficiario de cada transacción necesita pruebas de que en ese momento la mayor parte de los nodos están de acuerdo en que este UTXO no ha sido enviado a más usurarios.

Cuando Alice quiere transferir unidades monetarias a Bob, construye una transacción, en la que especifica la cantidad de criptos que cede de los UTXOs, la dirección del destinatario, la firma con su clave privada y la transmite a la red. Los nodos que reciben la transacción verifican la firma criptográfica y la válidez de la posesión de los UTXOs antes de aceptarla y retransmitirla. Este procedimiento propaga la transacción de manera indefinida hasta alcanzar a todos los nodos de la red. Finalmente, la transacción es validada por un minero y minada en un bloque. Una vez que una transacción se encuentra en la cadena de bloques y ha recibido la confirmación de un número razonable de bloques posteriores, la transacción se puede considerar parte permanente de la cadena de bloques y, por tanto, es aceptada por todos los participantes.

Cuando hay muchas transacciones el minero las empaqueta, hace una prueba de trabajo, de las que hablamos más adelante, y las añade a la blockchain.

Pruebas de trabajo o PoW

Para explicar qué es una prueba de trabajo empezaré con un ejemplo, el HashCash. Este algoritmo fue concebido con el propósito de combatir el correo no deseado. Antes de la implementación de la PoW, no existía una distinción en el costo computacional entre enviar un único correo o mil correos. HashCash propone solicitar a la entidad o persona que envía el correo que realice un trabajo computacional por cada mensaje enviado.

Para una persona común, esta prueba de trabajo solo lleva unos pocos segundos. Sin embargo, al intentar enviar el correo a miles de destinatarios y tener que resolver miles de pruebas, el proceso se vuelve considerablemente más prolongado y más costoso. De esta manera, HashCash introduce una barrera efectiva para desalentar el envío masivo de correos no deseados, al imponer una carga computacional que se vuelve significativa a medida que aumenta la escala del envío.

Definición 3.0.4. *El concepto de Prueba de Trabajo en la criptografía, también conocido como Proof of Work (PoW), se configura como un algoritmo diseñado para desincentivar o dificultar ciertos comportamientos dentro de un sistema. Su funcionamiento radica en que, para llevar a cabo una acción en la red, un individuo debe resolver un problema computacional. Este proceso se ejecuta de manera automática por la máquina u ordenador del usuario, y posteriormente se verifica por otros usuarios de la red.*

La característica fundamental de estas pruebas reside en su asimetría. Es esencial que la Prueba de Trabajo imponga un significativo costo computacional al realizar la acción, pero al mismo tiempo, la verificación de dicha acción debe ser sencilla y con poco costo.

Dentro de las posibles bases para una Prueba de Trabajo, se puede tomar como ejemplo el problema del logaritmo o las funciones hash, como veremos más adelante.

Las pruebas de trabajo siguen,en términos generales, el siguiente esquema:

1. Se inicia una acción en la red (una transacción, enviar un mail...).
2. Para completar dicha acción, la máquina u ordenador del usuario debe resolver un problema computacional complejo, consumiendo esto gran cantidad de recursos.
3. El usuario o la máquina presenta el resultado de la prueba de trabajo.
4. Otros usuarios de la red verifican la prueba.
5. Si el resultado es válido, se aprueba la acción que quiere hacer el usuario.

Este proceso se repite cada vez que un usuario quiera hacer una acción en una red. La prueba también cambia para evitar que esta sea fácil de resolver.

Blockchain y minería

Antes de hablar de blockchain debo definir el concepto de nonce.

Definición 3.0.5. *Un nonce o 'number that can only be used once', número que solo puede usarse una vez, es un número arbitrario que se emplea en criptografía dentro de los denominados protocolos de autenticación. Como ya dice el propio nombre, estos solo se usan una vez.*

En particular, en el contexto de la minería de criptomonedas, el nonce es un número que los mineros ajustan constantemente en el proceso de resolución de una prueba de trabajo. Esta requiere encontrar un valor de nonce que, al ser combinado con otros datos en el bloque, genere un hash que cumpla con ciertos criterios predeterminados, como comenzar con un cierto número de ceros.

La construcción de la cadena de bloques se hace por medio de esta actividad, la cual permite mantener una red Peer-to-Peer basada en la tecnología blockchain actualizada y segura.

Definición 3.0.6. *Una blockchain, como su nombre indica, es una cadena de bloques de información. Esto es, un registro compartido e inmutable que simplifica la documentación de las transacciones y el seguimiento de activos en una red. La relevancia de esta estructura reside en su capacidad para resistir modificaciones y en la verificación descentralizada, eliminando la necesidad de una entidad central. Además, cada acción queda registrada en la red. [11] [12]*

En una red blockchain basada en pruebas de trabajo el nonce funciona en combinación con el hash como un elemento de control para evitar la manipulación de la información de los bloques. En este caso el nonce es la solución al PoW que se usa para minar bloques, es probabilísticamente casi imposible que se repita. El cálculo del nonce se realiza de manera forzada, o lo que es lo mismo: se requieren grandes cantidades de recursos de cómputo y también de tiempo, por lo que conseguir este valor es una prueba de trabajo o PoW. Es imposible predecir la combinación de bits, que normalmente es de 32 bits y que dará como resultado un hash correcto.

Seguridad de la blockchain

En el caso de que una persona quiera cambiar un bloque de la historia de bitcoin, debe volver a minar el código hash de ese mismo bloque, ponerlo en el bloque siguiente y por tanto minar ese también, así sucesivamente. Mientras esa persona intenta cambiar la historia, el resto de los usuarios de bitcoin están minando, así que la persona sola no llegaría nunca a minar a la misma velocidad que el resto y la cadena de bloques se vería algo así:

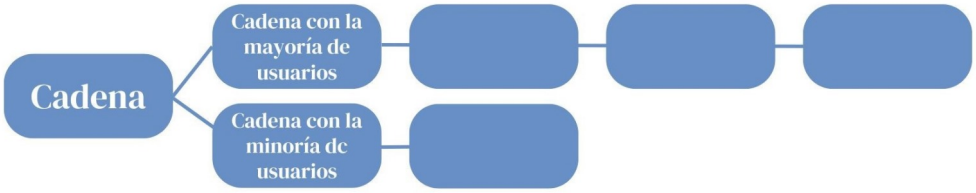

Figura 3.2: Cadena de bloques dividida

De esta manera se desincentiva al usuario a ser malicioso. No solo eso, cada vez que un usuario logra minar un bloque, este será recompensado con una cantidad de la criptomoneda, haciendo así que la gente use su energía para sellar bloques. La criptomoneda minada en la cadena más larga tendrá mayor valor porque está respaldada por más usuarios de la red. He de decir que no es el minero quien intenta resolver estas pruebas de trabajo, sino su máquina. Por el funcionamiento de los hashes, su resolución solo se puede hacer a partir de prueba y error. Por tanto, estas máquinas usan mucha energía, haciendo que esta gente tenga un gasto eléctrico muy grande.

Un blockchain o cadena de bloques es una base de datos formada por cadenas de bloques diseñadas para evitar modificaciones. Cada bloque contiene un conjunto de transacciones, un identificador único llamado hash y otros datos, en función de la moneda

que se esté usando. El enlace entre bloques se consigue mediante las pruebas de trabajo explicadas anteriormente.

Minería

Todos los mineros de la red compiten para ser los primeros en encontrar la solución al problema criptográfico del que hemos hablado antes. El objetivo de los mineros es buscar una solución válida para sellar el bloque que están minando. Este proceso se hace por prueba y error por el ordenador, generando así el coste computacional que se busca en las pruebas de trabajo. El primer minero en encontrar una solución válida se lleva un tipo de recompensa, lo que incentiva a los usuarios de las criptomonedas a invertir su trabajo. Cuando un minero encuentra la solución al problema criptográfico de su bloque, lo transmite al resto de los nodos a los que está conectado. En el caso de que dicho bloque sea válido dichos nodos lo retransmiten y lo agregan a la cadena de bloques. Este proceso se repite indefinidamente hasta que el bloque ha alcanzado todos los nodos de la red. Para que un bloque sea válido el minero que lo produjo debe incluir como referencia en la cabecera de este un hash o resumen criptográfico del último bloque de la cadena más larga de la que tienen conocimiento. [13]

Para modificar un bloque en la "historia" de cualquier criptomoneda hay que cambiar la solución de la prueba de trabajo dada al acabar el bloque, es decir que la persona que quiere cambiar las transacciones tiene que buscar las soluciones de las pruebas de trabajo de todos los bloques siguientes, esto hace que la minería sea más lenta porque hay menos ordenadores minando. Por tanto, cuando la mayoría vaya por un bloque, la minoría que hubiera intentado cambiar la historia tendrá muchos menos bloques. Este sistema no solo incentiva a los usuarios a ser honestos sino a la democracia dentro de los usuarios ya que lo mejor para todos es tener más ordenadores juntos para minar.

La construcción de la cadena de bloques se hace por medio de esta actividad, la cual permite mantener una red Peer to Peer basada en la tecnología blockchain actualizada y segura.

3.1. Bitcoin

Como ya hemos comentado en la introducción la criptomoneda nace de un pdf llamado 'Bitcoin: A Peer-to-Peer Electronic Cash System'. Aquí se describe el funcionamiento de las transacciones, blockchain, pruebas de trabajo y demás algoritmos para el correcto funcionamiento de la moneda.

Bitcoin es un sistema basado en UTXO ('Unspent Transacction Output'). Las cantidades de los UTXO están vinculadas a las direcciones que las puedan gastar por medio del registro de la cadena de bloques.

Blockchain y transacciones en bitcoin

Satoshi Nakamoto buscaba poder hacer transacciones sin una entidad central como los bancos, por lo que la blockchain no está almacenada en un ordenador principal sino

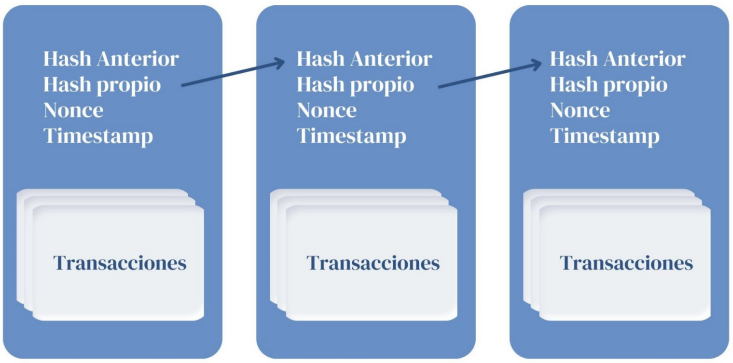

Figura 3.3: Blockchain en bitcoin

en la red, haciendo que cada usuario tenga la misma responsabilidad. La blockchain que usa bitcoin funciona de la siguiente manera:

Para sellar un bloque de transacciones se busca que los usuarios de la red resuelvan una prueba de trabajo. Esta consiste en encontrar un número que, junto a las transacciones del bloque, el hash del código anterior y una marca de tiempo, devuelva un código hash que cumpla ciertas condiciones predefinidas. Esto es lo que llamamos minería. El código hash se pone en el bloque como medida de seguridad contra posibles cambios.

Aquí explico un poco como funciona la red de bitcoin

1. Las transacciones nuevas se transmiten a todos los nodos.
2. Cada nodo recoge todas las transacciones en un bloque.
3. Cada nodo trabaja para resolver la PoW para su bloque.
4. Cuando un nodo resuelve su PoW, lo transmite al resto de los bloques.
5. Los nodos aceptan el bloque si todas las transacciones dentro de él son válidas.
6. Los nodos aceptan el bloque cuando se ponen a trabajar en crear el siguiente bloque en la cadena usando el código hash previo.

El valor de la moneda depende, en parte, de la cantidad de gente que forme parte de esta red. Por tanto para hacer que el valor de la moneda se reduzca, los nodos deberían considerar como correcta la cadena más larga y trabajar para extenderla. Si dos nodos transmiten simultáneamente diferentes versiones del siguiente bloque, algunos nodos recibirán una antes que la otra. En ese caso, trabajan sobre la primera que hayan recibido, pero guardan la otra por si esa ramificación se hace más larga.

3.2. Grin

Grin busca ser una moneda con más privacidad, y que necesite menos almacenamiento. Para ello se usa el protocolo Mimblewimble. Al usar ese protocolo la información que queda registrada en las transacciones de Grin no revela casi información de la gente mandando y recibiendo dinero ni de la cantidad.

Mimblewimble

Mimblewimble es un protocolo para preservar la privacidad de los usuarios que usan criptomonedas. Las principales características son el anonimato y el poco espacio que ocupan las transacciones. Para que sea anónimo totalmente no hay direcciones ni transacciones en la blockchain, para eso se usa la prueba de conocimiento cero Bulletproof. Al usar bulletproof también hacemos que la cantidad de información necesaria sea menor. [14]

La criptomoneda Grin está basada en el protocolo conocido como MimbleWimble. A diferencia del Bitcoin, en este protocolo la construcción del blockchain es totalmente anónima.

Los bloques están ordenados en el tiempo, y en cada bloque encontramos el número de bloque, las transacciones, una prueba de trabajo y por último un output de minería:

Numero de bloque
Transacciones
Kernel del bloque
Prueba de trabajo
Output de la minería = $g^r h^v$

Siendo r un número que solo conoce el minero y v una cantidad fija.

- Transacciones: Dentro de cada transacción encontramos el input, el output y los kernels.

Input	Monedas y sus bulletproofs
Output	Monedas y sus bulletproofs
Kernels	Firma de las transacciones

  - Input: Una lista de las monedas de entrada de todas las transacciónes del bloque, junto con sus bulletproofs. En esta lista no hay más datos, haciendo anonimas las transacciones.

Input	
$g^{r_1} h^{v_1}$	bulletproof
$g^{r_2} h^{v_2}$	bulletproof
...	...
$g^{r_n} h^{v_n}$	bulletproof

- Output: Lista de los siguientes UTXOS, es decir, la lista de monedas de llegadas o lista de monedas que aún no han sido gastadas.

Output	
$g^{r_1}h^{v_1}$	bulletproof
$g^{r_2}h^{v_2}$	bulletproof
...	...
$g^{r_m}h^{v_m}$	bulletproof

- Kernel: Firma digital conjunta de Schnorr, creada usando las firmas de todas las transacciones del bloque.

Kernel	
(s, g^k)	exceso

con $s = \sum s_n$ y $g^k = \prod g^{k_n}$ con (s_i, g^{k_i}) la firma de la transacción i.

Transacciones de Grin

Para describir como funcionan las transacciones en Grin utilizaremos un ejemplo básico. Para este trabajo no se tendrá en cuenta el coste monetario de hacer una transacción.

Alice tiene una moneda $g^{r_0}h^{v_0}$, de un valor de v_0, y le quiere dar a Bob v_1. Por tanto, Alice tendrá que generar dos transacciones, una moneda para Bob y otra para si misma de un valor de $v_2 = v_0 - v_1$.

Input	$g^{r_0}h^{v_0}$
Output	$g^{r_1}h^{v_1}, g^{r_2}h^{v_2}$
Kernel	Firma

Para que Alice, se pueda gastar esa moneda debe demostrar que conoce la clave secreta asociada a la misma, r_0, usando la firma de Schnorr.

Lo primero que hace Alice es generar de manera random la clave secreta del cambio r_2, después encuentra la diferencia entre la clave secreta de la moneda nueva y la original, $r_s = r_0 - r_2$. Genera un k_2 random que utilizará posteriormente para la firma de la transacción. Alice manda a Bob la siguiente información: la moneda original ($g^{r_0}h^{v_0}$), el cambio ($g^{r_2}h^{v_2}$), la clave secreta de k_2 (g^{k_2}) y la clave secreta de r_2 (g^{r_2}).

Bob con esta información, genera un mensaje M concatenando metadatos públicos, entre ellos, el número de bloque y la clave de identificación de la transacción. Luego crea de manera aleatoria su r_1 y k_1, busca el desafío de la firma de Schnorr e. Con todo esto Bob puede generar su firma parcial:

$$s_1 = k_1 + er_1$$

$$e = SHA256(M||g^{k_1+k_2}||g^{r_s+r_1})$$

Por último, Bob envía a Alice los siguientes datos: la firma parcial de Bob (s_1), la clave secreta de k_1 (g^{k_1}) y la clave secreta de r_1 (g^{r_1}).

Lo primero que tiene que hacer Alice es verificar la firma de Bob, para asegurarse de que Bob conoce la clave secreta de su moneda. Para ello, Alice genera e y verifica que se cumpla la siguiente igualdad:

$$g^{s_1} = g^{k_1}(g^{r_1})^e$$

Después genera su propia firma:

$$s_2 = k_2 + er_s$$

Alice manda a Bob su firma y este la verifica.

Una vez verificadas ambas firmas Alice genera la firma de la transacción:

$$(s_1 + s_2, g^{k_1+k_2})$$

$$s_1 + s_2 = (k_1 + er_1) + (k_2 + er_s) = (k_1 + k_2) + e(r_1 + r_s)$$

La firma, junto con $g^{r_s+r_1}$, que es el exceso de las claves secretas, es el kernel de la transacción. Con esto cualquier persona de la red puede verificar que las personas involucradas en la transacción conocen sus respectivas claves secretas. Estas deben poder verificar que tampoco se ha generado dinero nuevo, basandose en que $v_0 - (v_1 + v_2) = 0$, debería cumplirse lo siguiente:

$$(g^{r_1}h^{v_1} + g^{r_2}h^{v_2}) - g^{r_0}h^{v_0} = g^{r_1+r_2-r_0}h^0 = g^{r_s+r_1}$$

Otra medida de seguridad son las bulletproof, de las que hablaremos más en el siguiente capítulo. Estas pruebas se utilizan como medida de seguridad para demostrar que los v_i son valores positivos, dentro de un intervalo. Se usa para que no se puedan crear monedas de valores negativos y que la longitud de estos valores no sea mayor a cierta cantidad de bits.

Este ejemplo se puede generalizar a una cantidad mayor de inputs y outputs, es decir que cada transacción puede tener una cantidad de monedas de entrada y otra cantidad de monedas de salida, y funciona de la misma manera.

Una vez verificada la transacción se pone en el bloque los inputs y los outputs por separado sin importar el orden.

Número de bloque	
Transacción 1	Kernel 1
Transacción 2	Kernel 2
...	...
Transacción n	Kernel n

Por último se genera una firma y un exceso conjuntos con todos los kernels de todas las transacciones del bloque. Así cada bloque es tratado como una única transacción y se borran los kernels de las transacciones anteriores para que la cadena de bloques no pese tanto.

Número de bloque
Transacción 1 Transacción 2 ... Transacción n
Kernel del bloque

Capítulo 4

Bulletproofs

Una preocupación fundamental en la tecnología blockchain es la confidencialidad de los datos en la cadena de bloques. Para llegar a un consenso entre todos los nodos independientes en una red blockchain, cada nodo debe poder validar todas las transacciones; en la mayoría de los casos, esto significa que el contenido de las transacciones es visible para todos los nodos. Afortunadamente, existen varias soluciones que preservan la confidencialidad en una cadena de bloques, como las transacciones privadas, los canales HyperLedger Fabric, los canales de pago, el cifrado homomórfico, la combinación de transacciones y las pruebas de conocimiento cero.

En este capítulo, exploraremos en detalle los Bulletproofs, una técnica de pruebas de conocimiento cero no interactivas que no requieren una configuración confiable. Los Bulletproofs permiten a los usuarios demostrar, entre otras cosas, que un número cifrado se encuentra dentro de un rango específico sin revelar información adicional sobre el número en cuestión.

Los Bulletproofs fueron desarrollados por Jonathan Bootle de la University College de Londres y Benedikt Bünz de Stanford University. Este protocolo fue diseñado con la idea de implementarse en la cadena de bloques utilizada en Bitcoin, ya que con él el tamaño total del conjunto de UTXOs de Bitcoin se puede reducir de 160 GB a 17 GB. Aunque este protocolo no se utiliza actualmente en Bitcoin, criptomonedas como Monero y Grin lo emplean en sus transacciones como medida de seguridad.

En este capítulo se describe la implementación de una prueba de rango de conocimiento cero. Esta permite a la red blockchain validar que un número secreto está dentro de límites conocidos sin revelar el número en sí. Para ello, se requiere un compromiso sobre un número por parte de una entidad confiable; un usuario de Grin puede utilizar este compromiso para generar una prueba de rango, que será verificada por la red de Grin.

Describiremos la implementación en Python de la prueba de conocimiento cero interactiva Bulletproof. Luego, utilizando el algoritmo de Fiat-Shamir, convertiremos la prueba en no interactiva, ya que necesitamos que esta prueba pueda ser validada por cualquier nodo de la cadena de bloques en cualquier momento. Finalmente, demostraremos cómo funciona el algoritmo en detalle.

4.1. Range Proof

El problema que queremos solucionar con las pruebas de conocimiento cero es el siguiente: Alice quiere dar una cantidad v de criptomonedas a Bob, y quiere demostrar a Bob (y a la red) que esa cantidad está entre 0 y 2^n para cierto $n \in \mathbb{N}$. [15]

4.1.1. Algoritmo interactivo

Consideremos los siguientes datos públicos: h, g y p, donde p es un número primo suficientemente grande, y $h, g, g_1, \ldots, g_n, h_1, \ldots, h_n$ son elementos del grupo G. Aquí, G representa un grupo multiplicativo de orden p.

Listing 4.1: Inicialización

```
import random
import sympy

v  = 2024
L = 160
n = 20

def gen_resto_cuad(q):
    found = False
    while not found:
        x = random.randint(2,q-1)
        found = pow(x, (q-1)//2, q) == 1
    return x

def generarGH(n, L):
   found = False
   while not found:
      p = random.randint(2**(L-1), 2**L)
      q = 2*p+1
      found = sympy.isprime(p) and sympy.isprime(q)
   g = gen_resto_cuad(q)
   h = gen_resto_cuad(q)
   gi = [gen_resto_cuad(q) for i in range(n)]
   hi = [gen_resto_cuad(q) for i in range(n)]
   return (p, q, g, h, gi, hi)

(p, q, g, h, gi, hi) = generarGH(n, L)
```

- Alice:

  Alice conoce el valor de v en el intervalo $[0, 2^n - 1]$. Para proceder, representa v como número binario y lo asigna al vector a_L, de modo que $a_L = (a_0, \ldots, a_{n-1})$. Luego define $a_R = (a_0 - 1, \ldots, a_{n-1} - 1)$.

Después, elige al azar $\gamma, \delta, \rho \in \mathbb{Z}_p$. Además, busca de manera aleatorioa s_L,$s_R \in \mathbb{Z}_p^n$. A continuación, Alice realiza los siguientes cálculos:

- $V = h^{\gamma} g^{v} \in G$
- $A = h^{\delta} \bar{g}^{a_L} \bar{h}^{a_R} \in G$
- $S = h^{\rho} \bar{g}^{s_L} \bar{h}^{s_R} \in G$

Por último, Alice manda los valores de V, A y S a Bob.

Listing 4.2: Alice 1

```
def Alice1(v):
    # Calculo V
    gamma = random.randint(0,p-1)
    V = (pow(h, gamma, q) * pow(g, v, q)) % q

    # Calculo A
    #Calculo aL y aR
    a = bin(v)[2:]
    if len(a) < n:
        a = "0"*(n-len(a)) + a
    aL = []
    for i in range(n):
        if a[n-1-i] == '0':
            aL.append(0)
        else:
            aL.append(1)
    aR = [(aL[i]-1) % p for i in range(n)]
    alpha = random.randint(0, p-1)
    A = pow(h, alpha, q)
    for i in range(n):
        A = (A * pow(gi[i], aL[i], q)) % q
        A = (A * pow(hi[i], aR[i], q)) % q

    # Calculo S
    # Calculo sL y sR
    [sL, sR ]= [[],[]]
    for i in range(n):
        sL.append(random.randint(0,p-1))
        sR.append(random.randint(0,p-1))

    rho = random.randint(0,p-1)
    S = pow(h, rho, q)
    for i in range(n):
        S = (S * pow(gi[i], sL[i], q)) % q
        S = (S * pow(hi[i], sR[i], q)) % q
    return (gamma,V, aL, aR, alpha, A, sL, sR, rho, S)
```

- Bob:

  Elige de manera aleatoria $y, z \in \mathbb{Z}_p$ y se lo manda a Alice. Estos dos valores actúan como compromisos, de esta manera se asegura de que Alice no pueda elegir números que le resulten favorables.

Listing 4.3: Bob 1

```
def Bob1():
    [y, z]= [random.randint(1, p-1),random.randint(1, p-1)]
    return (y,z)
```

- Alice:

  Alice define los siguientes polonomios:

  - $l(X) = (a_L - z(1, \ldots, 1)) + s_L X \in \mathbb{Z}_p^n[X]$
  - $r(X) = (1, \ldots, y^{n-1}) \cdot (a_R + z(1, \ldots, 1)) + s_R X + z^2(1, \ldots, 2^{n-1}) \in \mathbb{Z}_p^n[X]$

  Con estos dos calcula $t(X) = \langle l(X), r(X) \rangle = t_0 + t_1 X + t_2 X^2$ donde:

  - $t_0 = \langle a_L - z(1, \ldots, 1), (1, y, \ldots, y^{n-1}) \cdot (a_R + z(1, \ldots, 1)) + z^2(1, 2, \ldots, 2^{n-1}) \rangle = z^2 v + \delta(y, z)$
  - $t_1 = \langle s_L, (1, y, \ldots, y^{n-1}) \cdot (a_R + z(1, \ldots, 1)) + z^2(1, \ldots, 2^{n-1}) \rangle + \langle a_L - z(1, \ldots, 1), s_R \rangle$
  - $t_2 = \langle s_L, s_R \rangle$

  A continuación, calcula:

  - $T_1 = h^{\tau_1} g^{t_1}$
  - $T_2 = h^{\tau_2} g^{t_2}$

  Donde τ_1, $\tau_2 \in \mathbb{Z}_p$ son valores elegidos al azar.

  Finalmente, Alice envia a Bob T_1 y T_2 como compromisos.

Listing 4.4: Alice 2

```
def Alice2(gamma,V, aL, aR, alpha, A, sL, sR, rho, S, y, z):
    # Calculo t0, t1, t2
    t0 = 0
    for i in range(n):
        t0 += (aL[i]-z) * (pow(y,i,p)*(aR[i]+z)
                           + pow(z,2,p)*pow(2,i,p))
        t0 %= p

    t1 = 0
    for i in range(n):
        t1 += (aL[i]-z)*sR[i]*pow(y,i,p)
        t1 += sL[i] * (pow(y,i,p)*(aR[i]+z)
                       + pow(z,2,p)*pow(2,i,p))
        t1  %= p

    t2 = 0
```

```
    for i in range(n):
        t2 = (t2 + sL[i]*pow(y,i,p)*sR[i]) % p

    # Calculo tao1, tao2
    tao1 = random.randint(0, p-1)
    tao2 = random.randint(0, p-1)

    # Calculo T1, T2
    T1 = (pow(g, t1, q) * pow(h, tao1, q)) % q
    T2 = (pow(g, t2, q) * pow(h, tao2, q)) % q
    return (t0, t1, t2, tao1, tao2, T1,T2)
```

- Bob:
  Bob elige otro compromiso $x \in \mathbb{Z}_p^*$ al azar y se lo manda a Alice.

Listing 4.5: Bob 2

```
def Bob2(T1,T2):
    x = random.randint(1, p-1)
    return x
```

- Alice:
  Alice realiza los siguientes calculos:
  - $\bar{l} = l(x) = a_L - z(1,\ldots,1) + s_L x \in \mathbb{Z}_p^n$
  - $\bar{r} = r(x) = (1, y, \ldots, y^{n-1}) \cdot (a_R + z(1,\ldots,1) + s_R x) + z^2(1,2,\ldots,2^{n-1}) \in \mathbb{Z}_p^n$
  - $\bar{t} = \langle \bar{l}, \bar{r} \rangle \in \mathbb{Z}_p$
  - $\tau_x = \tau_2 x^2 + \tau_1 x + z^2 \gamma \in \mathbb{Z}_p$
  - $\mu = \alpha + \rho x \in \mathbb{Z}_p$

  Finalmente, Alice le envia a Bob los valores $\bar{l}, \bar{r}, \bar{t}, \tau_x$ y μ.

Listing 4.6: Alice 3

```
def Alice3(gamma,V, aL, aR, alpha, A, sL, sR, rho, S, y, z,
           t0, t1, t2, tao1, tao2, T1,T2,x):

    l = [(aL[i]-z+sL[i]*x)%p for i in range(n)]

    r = [(pow(y,i,p) * (aR[i] + z + sR[i]*x)
          + z**2 * pow(2, i, p)) % p for i in range(n)]

    t = sum([l[i]*r[i] % p for i in range(n)]) % p

    taox = (tao2 * x**2 + tao1 * x + z**2 * gamma) % p

    mu = (alpha + rho*x) % p
    return (l,r,t,taox, mu)
```

- Bob:

  Para acabar el algoritmo, Bob debe comprobar que las siguientes igualdades se cumplen.

  - $g^t h^{\tau_x} = V^{z^2} g^{\delta(y,z)} T_1^x T_2^{x^2}$
  - $\bar{g}^{\bar{l}} \bar{h'}^{\bar{r}} = h^{-\mu} A S^x \bar{g}^{-z(1,\ldots,1)} \bar{h'}^{z(1,y,\ldots,y^{n-1})+z^2(1,2,\ldots,2^{n-1})}$
  - $\bar{t} = \langle \bar{l}, \bar{r} \rangle$

  Siendo $h' = (h'_1, \ldots, h'_n)$ con $h'_j = h_j^{y^{-j+1}}$.

  Si se cumple, Bob queda convencido de que Alice dice la verdad sobre $v \in [0, 2^{n-1}]$.

Listing 4.7: Bob 3

```
def Bob3(gamma,V, aL, aR, alpha, A, sL, sR, rho, S, y, z,
         t0, t1, t2, tao1, tao2, T1,T2,x,l,r,t,taox, mu):

    a1 = (pow(g, t, q) * pow(h, taox, q)) % q

    d = ((z-z**2) * sum([pow(y,i,p) for i in range(n)])
         - z**3 * sum([pow(2,i,p) for i in range(n)])) % p

    a2 = pow(V, pow(z,2,p), q) * pow(g, d, q)
        * pow(T1, x, q) * pow(T2, pow(x,2,p), q) % q

    if a1 != a2:
        print("error1")

    vh = [pow(hi[i], pow(y, -i, p), q) for i in range(n)]

    b1 = A * pow(S, x, q) % q
    for i in range(n):
        b1 = b1 * pow(gi[i], -z, q) % q
        b1 = b1 * pow(vh[i], (z*pow(y,i,p)
                              + pow(z,2,p)*pow(2,i,p))%p, q) % q

    b2 = pow(h, mu, q)
    for i in range(n):
        b2 = b2 * pow(gi[i], l[i], q) % q
        b2 = b2 * pow(vh[i], r[i], q) % q
    if b1 != b2:
       print("error2")

    c1 = t
    c2 = sum([l[i] * r[i] % p for i in range(n)]) % p
    if c1 != c2:
       print("error3")

    print("ok")
```

4.1.2. Algoritmo no interactivo Range Proof

Para que este protocolo pueda ser verificado por todos los usuarios de la red, sin depender de otra persona o entidad central. Usaremos el algoritmo de Fiat Shamir para pasar a un algoritmo no interactivo. En este caso sustituimos los compromisos de Bob por funciones SHA consiguiendo así unos compromisos pseudoaleatorios, que serán públicos.

El código es el mismo que el de la sección anterior pero las funciones Bob1 y Bob2 son sustituidas por funciones hash donde los elementos de entrada son los datos que recibe Bob de Alice. Como los datos que recive Bob son siempre públicos, cualquier persona puede generar la misma función hash. Suponiendo que la función usada por el algorimto es resistente a colisiones no hay manera de falsificarla.

Primero inicializaremos buscando las g y las h.

Listing 4.8: Inicialización

```
import random
import sympy
import hashlib

v  = 2024
L = 160
n = 20

def gen_resto_cuad(q):
    found = False
    while not found:
        x = random.randint(2,q-1)
        found = pow(x, (q-1)//2, q) == 1
    return x

def generarGH(n, L):
   found = False
   while not found:
      p = random.randint(2**(L-1), 2**L)
      q = 2*p+1
      found = sympy.isprime(p) and sympy.isprime(q)
   g = gen_resto_cuad(q)
   h = gen_resto_cuad(q)
   gi = [gen_resto_cuad(q) for i in range(n)]
   hi = [gen_resto_cuad(q) for i in range(n)]
   return (p, q, g, h, gi, hi)

(p, q, g, h, gi, hi) = generarGH(n, L)
```

Luego Alice sigue todos los pasos que explicamos en la seccion anterior y sustituye los compromisos que hace Bob por funciones SHA pseudoaleatorias que luego Bob podrá verificar.

Listing 4.9: Alice

```
def Alice1(v):
    # Calculo V
    gamma = random.randint(0,p-1)
    V = (pow(h, gamma, q) * pow(g, v, q)) % q

    # Calculo A
    #Calculo aL y aR
    a = bin(v)[2:]
    if len(a) < n:
        a = "0"*(n-len(a)) + a
    aL = []
    for i in range(n):
        if a[n-1-i] == '0':
            aL.append(0)
        else:
            aL.append(1)
    aR = [(aL[i]-1) % p for i in range(n)]
    alpha = random.randint(0, p-1)
    A = pow(h, alpha, q)
    for i in range(n):
        A = (A * pow(gi[i], aL[i], q)) % q
        A = (A * pow(hi[i], aR[i], q)) % q

    # Calculo S
    # Calculo sL y sR
    [sL, sR ]= [[],[]]
    for i in range(n):
        sL.append(random.randint(0,p-1))
        sR.append(random.randint(0,p-1))

    rho = random.randint(0,p-1)
    S = pow(h, rho, q)
    for i in range(n):
        S = (S * pow(gi[i], sL[i], q)) % q
        S = (S * pow(hi[i], sR[i], q)) % q

    ym = str(V)+str(A)+str(S)+ str(1)
    ym1 = ym.encode('utf-8')
    y = (int(hashlib.sha1(ym1).hexdigest(), base=16))%p

    zm = str(V)+str(A)+str(S)+ str(2)
    zm1 = zm.encode('utf-8')
    z = (int(hashlib.sha1(zm1).hexdigest(), base=16))%p

    # Calculo t0, t1, t2
    t0 = 0
    for i in range(n):
        t0 += (aL[i]-z) * (pow(y,i,p)*(aR[i]+z)
                           + pow(z,2,p)*pow(2,i,p))
        t0 %= p
```

```
51
52      t1 = 0
53      for i in range(n):
54          t1 += (aL[i]-z)*sR[i]*pow(y,i,p)
55          t1 += sL[i] * (pow(y,i,p)*(aR[i]+z)
56                         + pow(z,2,p)*pow(2,i,p))
57          t1  %= p
58
59      t2 = 0
60      for i in range(n):
61          t2 = (t2 + sL[i]*pow(y,i,p)*sR[i]) % p
62
63      # Calculo tao1, tao2
64      tao1 = random.randint(0, p-1)
65      tao2 = random.randint(0, p-1)
66
67      # Calculo T1, T2
68      T1 = (pow(g, t1, q) * pow(h, tao1, q)) % q
69      T2 = (pow(g, t2, q) * pow(h, tao2, q)) % q
70
71      xm = str(T1)+str(T2)
72      xm1 = xm.encode('utf-8')
73      x = (int(hashlib.sha1(xm1).hexdigest(), base=16))%p
74
75      # 1. Calculo l(x)
76      l = [(aL[i]-z+sL[i]*x)%p for i in range(n)]
77
78      # 2. Calculo r(x)
79      r = [(pow(y,i,p) * (aR[i] + z + sR[i]*x)
80            + z**2 * pow(2, i, p)) % p for i in range(n)]
81
82      # 3. Calculo t
83      t = sum([l[i]*r[i] % p for i in range(n)]) % p
84
85      # 4. taox
86      taox = (tao2 * x**2 + tao1 * x + z**2 * gamma) % p
87
88      # 5. mu
89      mu = (alpha + rho*x) % p
90      return (gamma,V, aL, aR, alpha, A, sL, sR,
91              rho, S, y, z,t0, t1, t2, tao1, tao2,
92              T1,T2,x,l,r,t,taox, mu)
```

Ahora Bob verificará todo el proceso de la misma manera que en la sección anterior.

Listing 4.10: Bob

```
1  def Bob(gamma,V, aL, aR, alpha, A, sL, sR, rho, S, y, z,
2          t0, t1, t2, tao1, tao2, T1,T2,x,l,r,t,taox, mu):
3      #1.Primera igualdad
```

```
    a1 = (pow(g, t, q) * pow(h, taox, q)) % q
    d = ((z-z**2) * sum([pow(y,i,p) for i in range(n)])
         - z**3 * sum([pow(2,i,p) for i in range(n)])) % p

    a2 = pow(V, pow(z,2,p), q) * pow(g, d, q)
    * pow(T1, x, q) * pow(T2, pow(x,2,p), q) % q
    if a1 != a2:
        print("error1")

    # 2. Segunda igualdad
    vh = [pow(hi[i], pow(y, -i, p), q) for i in range(n)]

    b1 = A * pow(S, x, q) % q
    for i in range(n):
        b1 = b1 * pow(gi[i], -z, q) % q
        b1 = b1 * pow(vh[i], (z*pow(y,i,p)
                              + pow(z,2,p)*pow(2,i,p))%p, q) % q

    b2 = pow(h, mu, q)
    for i in range(n):
        b2 = b2 * pow(gi[i], l[i], q) % q
        b2 = b2 * pow(vh[i], r[i], q) % q
    if b1 != b2:
       print("error2")

    # 3. Tercera igualdad
    c1 = t
    c2 = sum([l[i] * r[i] % p for i in range(n)]) % p
    if c1 != c2:
       print("error3")

    # 4. Cuarta Igualdad
    y1m = str(V)+str(A)+str(S)+ str(1)
    y1m1 = y1m.encode('utf-8')
    y1 = (int(hashlib.sha1(y1m1).hexdigest(), base=16))%p
    if y1 != y:
       print("error4")

    z1m = str(V)+str(A)+str(S)+ str(2)
    z1m1 = z1m.encode('utf-8')
    z1 = (int(hashlib.sha1(z1m1).hexdigest(), base=16))%p
    if z1 != z:
       print("error5")

    x1m = str(T1)+str(T2)
    x1m1 = x1m.encode('utf-8')
    x1 = (int(hashlib.sha1(x1m1).hexdigest(), base=16))%p
    if x1 != x:
       print("error6")

    print("ok")
```

4.1.3. Demostración

Proposición 4.1.1. *Se cumple que $v \in [0, 2^n)$ para algún $n \in \mathbb{N}$ es igual a que existan dos vectores $a_R, a_L \in \mathbb{Z}_p^n$ definidos tal que $a_L = (a_0, a_1, \ldots, a_{n-1})$ y $a_R = a_L - (1, \ldots, 1) = (a_0 - 1, \ldots, a_{n-1} - 1)$ y cumplen las siguientes condiciones:*

1. $v = \langle a_L, (1, 2, 2^2, \ldots, 2^{n-1})\rangle$
2. $a_R = a_L - (1, \ldots, 1)$
3. $a_L \cdot a_R = (0, \ldots, 0)$

Demostración. Demostrar que $0 \leq v < 2^n$ es lo mismo que demostrar $v \in [0, 2^{n-1}]$. Si v pertenece al intervalo es lo mismo que decir que v se puede escribir así $v = a_0 + 2a_1 + 2^2 a_2 + \ldots + 2^{n-1} a_{n-1}$, $a_i \in \{0, 1\}$. Por tanto, Alice debe demostrar que $\exists a_i \forall i \in \{1, \ldots, n-1\}$. Si definimos los vectores $a_L = (a_0, a_1, \ldots, a_{n-1}) \in \mathbb{Z}_p^n$ y $a_R = a_L - (1, \ldots, 1) = (a_0 - 1, \ldots, a_{n-1} - 1) \in \mathbb{Z}_p^n$, lo anterior es lo mismo que demostrar la existencia de a_L y a_R. Por construcción, tenemos que si se cumple $0 \leq v < 2^n$, entonces:

1. $v = \langle a_L, (1, 2, 2^2, \ldots, 2^{n-1})\rangle$
2. $a_R = a_L - (1, \ldots, 1)$
3. $a_L \cdot a_R = (0, \ldots, 0)$

□

Proposición 4.1.2. *Existen los vectores a_L y a_R que cumplen la proposición anterior si y solo si, $\forall y \in \mathbb{Z}_p$ se cumplen las siguientes igualdades:*

1. $\langle a_L, (1, 2, 2^2, \ldots, 2^{n-1})\rangle = v$
2. $\langle a_L, a_R \cdot (1, y, \ldots, y^{n-1}\rangle = 0$
3. $\langle a_L - (1, \ldots, 1) - a_R, (1, y, \ldots, y^{n-1}\rangle = 0$

Demostración.

1. $v = \langle a_L, (1, 2, \ldots, 2^{n-1})\rangle$. Es la definición de v.
2. $\langle a_R - a_L + (1, \ldots, 1), (1, y, y^2, \ldots, y^{n-1})\rangle = 0 \Leftrightarrow a_R - a_L + (1, \ldots, 1) = 0 \Leftrightarrow a_R = a_L - (1, \ldots, 1)$ ya que $y \neq 0$ y esta es la definición de a_R
3. $\langle a_L, a_R \cdot (1, y, \ldots, y^{n-1})\rangle = 0 \Leftrightarrow a_{L0} \cdot a_{R0} \cdot 1 + a_{L1} \cdot a_{R1} \cdot y + \ldots + a_{Ln-1} \cdot a_{Rn-1} \cdot y^{n-1} = 0 \Leftrightarrow$

   Hay dos opciones: o bien y es la raíz del polinomio, lo cual tiene una probabilidad de aproximadamente $n - 1/p \approx 0$, o bien, $a_{Li} \cdot a_{Ri} = 0 \ \forall i \in \{0, \ldots, n-1\}$. La probabilidad de la segunda opción es casi 1 por tanto convence a Bob.

□

Proposición 4.1.3. *Dado un $z \in \mathbb{Z}_p$ elegido al azar, que se cumplan las tres ecuaciones anteriores es lo mismo que la siguiente ecuación:*

$$z^2\langle a_L, (1, 2, \ldots, 2^{n-1})\rangle + z\langle a_R - a_L - (1, \ldots, 1), (1, y, y^2, \ldots, y^{n-1})\rangle + \langle a_L, a_R \cdot (1, y, \ldots, y^{n-1})\rangle = z^2 \cdot v$$

Demostración. Trivial

□

Proposición 4.1.4. *La ecuación anterior es equivalente a:*

$$\langle a_L - z(1, \ldots, 1), (1, y, \ldots, y^{n-1}) \cdot (a_R + z(1, \ldots, 1)) + z^2(1, 2, \ldots, 2^{n-1})\rangle = z^2 \cdot v + \delta(y, z)$$

con

$$\delta(y, z) = (z - z^2)\langle (1, \ldots, 1), (1, y, \ldots, y^{n-1})\rangle - z^3\langle (1, \ldots, 1), (1, 2, \ldots, 2^{n-1})\rangle$$

Demostración. Para ver que las dos ecuaciones son equivalentes, se puede expandir y reorganizar los términos en ambas ecuaciones.

□

Proposición 4.1.5. *Dados los siguientes polinomios:*

1. $l(x) = (a_L - z \cdot (1, \ldots, 1)) + s_L \cdot x \in \mathbb{Z}_p^n[x]$
2. $r(x) = (1, y, y^2, \ldots, y^{n-1}) \cdot (a_R + z(1, \ldots, 1) + s_R \cdot x) + z^2(1, 2, \ldots, 2^{n-1}) \in \mathbb{Z}_p^n[x]$
3. $t(x) = \langle l(x), r(x)\rangle = t_o + t_1 \cdot x + t_2 \cdot x^2$ *con* $t_0 = z^2 \cdot v + \delta(y, z)$

y dados $\tau_1, \tau_2, \gamma, \alpha, \rho \in \mathbb{Z}_p$*, elegidos al azar, definimos:*

1. $T_1 = g^{t_1} h^{\tau_1}$
2. $T_2 = g^{t_2} h^{\tau_2}$
3. $\tau_x = \tau_2 x^2 + \tau_1 x + z^2\gamma \in \mathbb{Z}_p$
4. $\mu = \alpha + \rho x$
5. $V = h^\gamma g^v$
6. $A = h^\alpha g^{a_L} \bar{h}^{a_R}$
7. $S = h^\rho g^{s_L} \bar{h}^{s_R}$

Entonces se cumple que la ecuacion de la proposición anterior es equivalente a que a las siguientes igualdades:

1. $t(x) = \langle l(x), r(x)\rangle$
2. $g^{t(x)} h^{\tau_x} = V^{z^2} g^{\delta(y,z)} T_1^x T_2^{x^2}$

3. $g^{l(x)}h'^{r(x)} = h^{-\mu}AS^x g^{-z(1,\dots,1)}h'^{z(1,y,\dots,y^{n-1})+z^2(1,2,\dots,2^{n-1})}$
   con $h' = (h_1^{y^{-1+1}}, h_2^{y^{-2+1}}, \dots, h_n^{y^{-n+1}})$

Demostración. Para ver que las ecuaciones son equivalentes, se puede expandir y reorganizar los términos.

1. Trivial.
2. $g^{t(x)}h^{\tau_x} = V^{z^2}g^{\delta(y,z)}T_1^x T_2^{x^2}$

   $$V^{z^2}g^{\delta(y,z)}T_1^x T_2^{x^2} = g^{vz^2+\delta(y,z)+t_1x+t_2x^2}h^{\gamma z^2+\tau_1 x+\tau_2 x^2}$$

   Es trivial ver que $h^{\gamma z^2+\tau_1 x+\tau_2 x^2} = h^{\tau_x}$ por la definicion de τ_x.
   Ahora solo hay que comprobar que

   $$g^{vz^2+\delta(y,z)+t_1x+t_2x^2} = g^{t(x)}$$

   Para eso comprobamos si se cumple:

   $$vz^2 + \delta(y,z) + t_1x + t_2x^2 = t(x)$$

   Necesitamos que

   $$vz^2 + \delta(y,z) = t_0$$

   Por definición de t_0, esto se cumple.
3. $g^{l(x)}h'^{r(x)} = h^{-\mu}AS^x g^{-z(1,\dots,1)}h'^{z(1,y,\dots,y^{n-1})+z^2(1,2,\dots,2^{n-1})}$
   con $h' = (h_1^{y^{-1+1}}, h_2^{y^{-2+1}}, \dots, h_n^{y^{-n+1}})$
   Sustituimos A y S en la igualdad y la dejamos en funcion de g, h, $\bar{h}$

   $$h^{-\mu}AS^x g^{-z(1,\dots,1)} \cdot h'^{z(1,y,\dots,y^{n-1})+z^2(1,2,\dots,2^{n-1})} =$$

   $$= g^{a_L - z(1,\dots,1)+s_L x}\bar{h}^{a_L+s_R x}h'^{z(1,y,\dots,y^{n-1})+z^2(1,2,\dots,2^{n-1})}$$

   Como $g^{a_L-z(1,\dots,1)+s_L x} = g^{t(x)}$, solo nos queda demostrar que se cumple

   $$\bar{h}^{a_L+s_R x}h'^{z(1,y,\dots,y^{n-1})+z^2(1,2,\dots,2^{n-1})} = h'^{r(x)}$$

   Sustituimos $r(x)$ por $(1, y, y^2, \dots, y^{n-1}) \cdot (a_R + z(1,\dots,1) + s_R$

   $$\bar{h}^{a_L+s_R x}h'^{z(1,y,\dots,y^{n-1})+z^2(1,2,\dots,2^{n-1})} =$$

   $$= h'^{(1,y,y^2,\dots,y^{n-1})\cdot(a_R+z(1,\dots,1)+s_R\cdot x)+z^2(1,2,\dots,2^{n-1})}$$

   Despejando la ecuación llegamos a lo siguiente:

   $$\bar{h}^{a_L+s_R x} = h'^{(1,y,y^2,\dots,y^{n-1})\cdot(a_R+s_R\cdot x)}$$

   Esto se cumple por la definición de h'.

□

Teorema 4.1.1. *Para demostrar que* $v \in [0, 2^n)$ *tenemos que demostrar que se cumplen las siguientes igualdades:*

1. $t(x) = \langle l(x), r(x) \rangle$
2. $g^{t(x)} h^{\tau_x} = V^{z^2} g^{\delta(y,z)} T_1^x T_2^{x^2}$
3. $g^{l(x)} h'^{r(x)} = h^{-\mu} A S^x g^{-z(1,\ldots,1)} h'^{z(1,y,\ldots,y^{n-1}) + z^2(1,2,\ldots,2^{n-1})}$
   con $h' = (h_1^{y^{-1+1}}, h_2^{y^{-2+1}}, \ldots, h_n^{y^{-n+1}})$

La demostración es trivial usando las proposiciones anteriores.

Capítulo 5

Conclusiones

En este Trabajo de Fin de Grado se han explorado diversos protocolos criptográficos y su aplicación en las pruebas de conocimiento cero, con especial atención al protocolo Bulletproof. A lo largo de este estudio, se ha demostrado que las Bulletproofs ofrecen una solución eficaz para mantener la privacidad y seguridad en las transacciones de criptomonedas, como Grin, en comparación con Bitcoin.

El resultado más destacado es la eficiencia en el ahorro de espacio que proporcionan las Bulletproofs. Por ejemplo, el tamaño total del conjunto de UTXOs de Bitcoin podría reducirse significativamente de 160 GB a solo 17 GB si se implementaran las Bulletproofs . Este ahorro de espacio no solo optimizaría el almacenamiento, sino que también mejoraría la escalabilidad de las redes blockchain.

Además, el uso de Bulletproofs en Grin no solo garantiza la confidencialidad de las transacciones, sino que también asegura que los datos almacenados en la blockchain no revelen información sensible sobre las partes involucradas ni las cantidades transferidas. Este nivel de anonimato es crucial en un entorno en el que la privacidad es cada vez más valorada.

Sería interesante que otras criptomonedas consideraran implementar protocolos como Mimblewimble y Bulletproofs para mejorar la privacidad y la eficiencia de sus transacciones. La adopción de estas tecnologías podría representar un avance significativo hacia redes blockchain más seguras y privadas, beneficiando a una comunidad más amplia de usuarios.

En resumen, las Bulletproofs se presentan como una herramienta prometedora para el futuro de las criptomonedas, ofreciendo una combinación de privacidad, seguridad y eficiencia en el almacenamiento de datos que podría transformar la forma en que se realizan y verifican las transacciones en el mundo digital.

Bibliografía

[1] Satoshi Nakamoto: Bitcoin - A peer-to-peer electronic cash system (2008).

[2] Carlos Cilleruelo: ¿Qué es SHA-1? (2024). https://keepcoding.io/blog/que-es-sha-1/

[3] Nik Piepenbreier: Python SHA256 Hashing Algorithm: Explained (2021). https://datagy.io/python-sha256/

[4] Whitfield Diffie and Martin E.Hellman: New Directions in Cryptography. Transactions in information Theory (1976).

[5] Wikipedia: Prueba de conocimiento cero https://es.wikipedia.org/wiki/Prueba_de_conocimiento_cero

[6] Ivan Damgård and Jesper Buus Nielsen: Commitment Schemes and Zero-Knowledge Protocols (2011).

[7] Yehuda Lindell: Simple Three-Round Multiparty Schnorr Signing with Full Simulatability (2020).

[8] H.Ong and C.P.Schnorr: Fast Signature Generation with a Fiat Shamir-Like Scheme (1991).

[9] Thea Peacock, Peter Y. A. Ryan, Steve Schneider and Zhe Xia: Verifiable Voting Systems. Computer and Information Security Handbook, pag. 293–315 (2013).

[10] MimbleWimbleCoin White Paper (2021).

[11] IBM: ¿Qué es el blockchain? (2024). https://www.ibm.com/es-es/topics/blockchain

[12] Guneet Kaur: ¿Qué es la blockchain de Bitcoin? Una guía sobre la tecnología detrás de BTC (2024). https://es.cointelegraph.com/learn/how-does-blockchain-work-a-beginners-guide-to-blockchain-technology

[13] Guneet Kaur: Qué es y cómo funciona la minería de Bitcoin. Todo sobre cómo minar bitcoins (2024). https://es.cointelegraph.com/learn/what-is-mining

[14] Georg Fuchsbauer, Michele Orrù and Yannick Seurin: Aggregate Cash Systems: A Cryptographic Investigation of Mimblewimble (2018).

[15] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille and Greg Maxwell: Bulletproofs: Short Proofs for Confidential Transactions and More (2017).