Criptografía aplicada: Cálculo del hash SHA-256 de un bloque Bitcoin

Javier Domínguez Gómez
jdg@member.fsf.org
Fingerprint: 94AD 19F4 9005 EEB2 3384 C20F 5BDC C668 D664 8E2B

v0.1.07 - Abril 2019

Índice

1. Introducción

1.1. Formatos en el texto

A lo largo de este documento el lector encontrará algunas palabras o bloques de texto con diferentes formatos o tipografía. Se mostrarán en letra *cursiva* las palabras en inglés o que hagan referencia a algún término técnico. Resaltadas o en **negrita** aquellas palabras que describen una propiedad, una variable o un nombre de fichero, y finalmente con tipografía `Courier` aquellos datos que representen un valor hexadecimal o base 16, o fragmentos de código fuente en diferentes lenguajes de programación.

1.2. Objetivo

Este documento presenta en detalle un análisis criptográfico sobre cada una de las partes de las que se compone un bloque de datos en el criptosistema definido por el protocolo *Bitcoin*[1]. Se tratarán algunos aspectos importantes como los diferentes datos, tipos, formatos y la distribución de los mismos en la estructura del bloque, así como las operaciones lógico-matemáticas de la función criptográfica SHA-256[2] que se emplean con el fin de generar un *hash* resultante adecuado, es decir, un resultado que cumpla las características definidas en el protocolo *Bitcoin*. Finalmente este *hash* representará dicho bloque en lo que se conoce como la cadena de bloques, o como el propio creador Satoshi Nakamoto lo llamaba originalmente, *timechain*[3]. Todos los datos utilizados en los ejemplos, así como en los cálculos, son datos reales pertenecientes al bloque número #286819 de la cadena de bloques de *Bitcoin* en la red principal *mainnet*. No existe ninguna razón especial en la elección de este bloque, es uno al azar. El lector puede consultar los datos de este o cualquier otro bloque, bien en la cadena de bloques descargada en su computadora local o bien en un *explorador de bloques online*[4], y realizar los mismos cálculos en todos los casos, obviamente obteniendo resultados diferentes.

1.3. A quién va dirigido

El documento va dirigido a principalmente a lectores familiarizados con la criptología en cualquiera de sus dos ramas: la criptografía y el criptoanálisis. También va dirigido a estudiantes de ciencias de la computación o matemáticas, en general a lectores que todavía no se han adentrado en el campo de del análisis forense criptográfico sobre la información que contiene un bloque de la cadena de bloques de *Bitcoin*, pero que quieren comprender su funcionamiento al más bajo nivel, a nivel de *bits*. El texto trata de facilitar al lector la información sobre los métodos empleados para obtener un *hash* que cumpla una serie de requisitos

[1] https://bitcoin.org/bitcoin.pdf

[2] https://github.com/JavierDominguezGomez/Cryptography/blob/master/SHA-256/Cryptography_SHA-256_es.pdf

[3] https://bitcointalk.org/index.php?topic=382374.0

[4] https://www.blockchain.com/en/btc/block-height/286819

según el criptosistema planteado por el protocolo *Bitcoin*. Para ello será necesario tener de antemano unos conocimientos mínimos sobre matemática discreta, *álgebra de Boole*[5], conversión de datos en diferentes bases como binario, decimal y hexadecimal, y comprender cómo funcionan algunas operaciones lógicas, módulo, rotativas y de desplazamiento de *bits*.

2. Estructura de datos de un bloque

Cuando se hace referencia a un *bloque* en realidad se está haciendo referencia a un conjunto de datos ordenados en base a unas normas, que en el caso que nos atañe se definen cuando se desarrolla el protocolo *Bitcoin*, creando así una estructura de datos ordenada y segmentada de forma que sea sencillo identificar cada una de las partes. Los datos se ordenan en 5 secciones principales. La primera sección se reserva para un dato denominado **magicId**, un dato numérico de 4 *bytes* y la segunda sección para almacenar el dato **size**, al igual que el anterior también es un dato numérico de 4 *bytes*, este representará el tamaño final de el bloque. En los siguientes puntos se explicará en detalle cada uno de los datos. Estos dos segmentos juntos se pueden usar como el delimitador que separa un bloque de otro en la cadena, pues los *bytes* que conforman los datos de los bloques se representan concatenados uno a continuación de otro. La siguiente imagen representa la estructura de un bloque en el que se muestra resaltado en color verde el segmento que alberga el dato *magicId* y en amarillo el segmento que alberga el dato *size*, ambos seguidos por el resto de datos del bloque, la zona de color gris punteada.

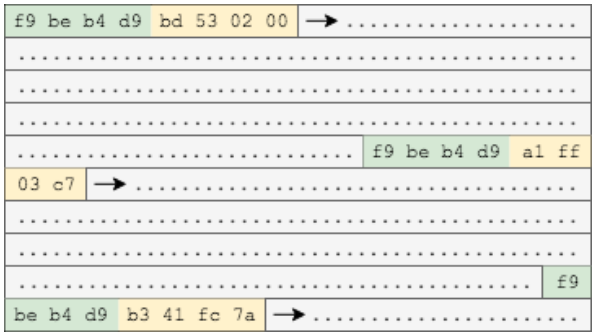


Figura 1: Los dos primeros segmentos de datos concatenados de cada bloque.

El tercer segmento tiene un espacio de memoria reservado de 80 *bytes* para almacenar los datos de la cabecera o **header**. El cuarto segmento almacenará un dato denominado **transactionCount** que representa el número de transacciones que contiene el bloque, será un número de tipo entero sin signo y el tamaño

[5] https://en.wikipedia.org/wiki/Boolean_algebra

puede variar dependiendo del número de transacciones en cada bloque. Por último, el quinto segmento se reserva para almacenar todos los datos de los que se componen cada una de las transacciones, esta sección llamada **transaction data** tendrá un tamaño variable que se corresponde con el tamaño disponible o restante hasta completar el bloque. A continuación una representación gráfica de los 5 segmentos de datos de los que se compone cada bloque.

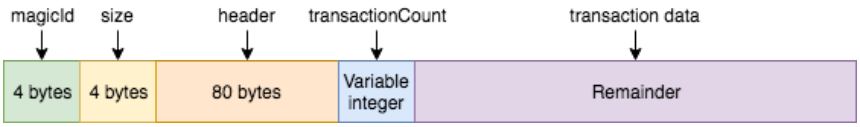


Figura 2: Los 5 bloques de datos principales en cada bloque de la cadena.

El segmento **header** a su vez se compone de 6 datos que son **versionNumber**, **previousBlockHash**, **merkleRoot**, **timeStamp**, **targetDifficulty** y **nonce**. Todos los datos del bloque tienen sus propios atributos, como puede ser el tipo de dato o tipo primitivo, un nombre, el tamaño que ocupa en memoria y el formato en el que este se representará finalmente. En la siguiente tabla se categorizan los datos más significativos y a continuación se explican en detalle.

Tipo de dato	Nombre	Tamaño	Formato
uint32_t	magicID	4 bytes	Little-endian
uint32_t	size	4 bytes	Little-endian
uint32_t	versionNumber	4 bytes	Little-endian
uint8_t[32]	previousBlockHash	32 bytes	Big-endian
uint8_t[32]	merkleRoot	32 bytes	Big-endian
uint32_t	timeStamp	4 bytes	Little-endian
uint32_t	targetDifficulty	4 bytes	Little-endian
uint32_t	nonce	4 bytes	Little-endian
uint8/16/32/64_t	transactionCount	1, 3, 5 o 9 bytes	Big-endian*

Cuadro 1: Principales datos de un bloque en la cadena de bloques de *Bitcoin*.

1. **magicID**: En la red de *Bitcoin* se establecen conexiones entre los nodos con la finalidad de establecer una comunicación para el envío y recepción de datos o mensajes. Los mensajes se envían mediante un canal en el que según van llegando se van concatenando uno detrás de otro. Así pues, cuando se desarrolló el protocolo *Bitcoin* se vio conveniente añadir un prefijo en cada mensaje anteponiendo este dato de 4 *bytes* y así poder identificar fácilmente no solo la red de *Bitcoin* en la que se generan, si no también dónde empieza y termina cada mensaje que circula entre los diferentes nodos de la red. La siguiente tabla muestra los diferentes valores que puede tener la variable *magicID* dependiendo de la red.

Red	magicID
Mainnet	`0xf9beb4d9`
Testnet	`0xfabfb5da`
Testnet3	`0x0b110907`
Namecoin	`0xf9beb4fe`
Regtest	`0xfabfb5da`

Se decidió establecer estos valores tan específicos dada la improbabilidad de que los caracteres ASCII que representan se encuentren en un mensaje estándar, tal y como se indica en el archivo *chainparams.cpp*[6] que lo implementa en el código de *Bitcoin*.

2. **size**: En un dato numérico hexadecimal de 4 *bytes* de valor variable que representa la longitud en *bytes* del bloque actual. Está codificado en formato *Little-endian*. Por ejemplo, un bloque con tamaño 152.509 KB tendrá el siguiente valor:

$$152509_{10} = \texttt{0x000253BD} \Rightarrow \texttt{0xBD530200}$$

3. **versionNumber**: Este dato hexadecimal puede cambiar de valor cuando se actualiza el software y cambia el número de la versión del protocolo. Tiene un tamaño de 4 *bytes* y se codifica en formato *Little-endian*, por ejemplo:

$$2_{10} = \texttt{0x00000002} \Rightarrow \texttt{0x02000000}$$

4. **previousBlockHash**: Se trata del *hash* o *digest* resultante del bloque anterior tras aplicar las funciones criptográficas utilizando los datos de la cabecera de dicho bloque. Tiene una longitud de 256 *bits* o 32 *bytes* codificado en formato *Big-endian*. Tal y como se explica al inicio del este documento el ejemplo en todo caso toma los datos correspondientes al bloque #286819, así pues, el valor de este dato ha de ser el *hash* resultante del bloque anterior, que en este caso es el bloque #286818[7].

```
0000000000000000117c80378b8da0e33559b5997f2ad55e2f7d18ec1975b9717
```

5. **merkleRoot**: Cada vez que una nueva transacción es aceptada el valor de esta variable se modifica. Se trata de un *hash* hexadecimal de 32 *bytes* codificado en formato *Big-endian*. Para obtener el *hash* final, también llamado *hash root* se han de concatenar los datos de las transacciones ubicados en los nodos hoja del árbol, en grupos de dos. De ese modo, por cada dos transacciones se obtiene un *hash* nuevo que será incluido en un nuevo vector que repetirá la acción hasta llegar al *hash root*.

[6] https://github.com/bitcoin/bitcoin/blob/master/src/chainparams.cpp#L99
[7] https://blockstream.info/block/0000000000000000117c80378b8da0e33559b5997f2ad55e2f7d18ec1975b9717

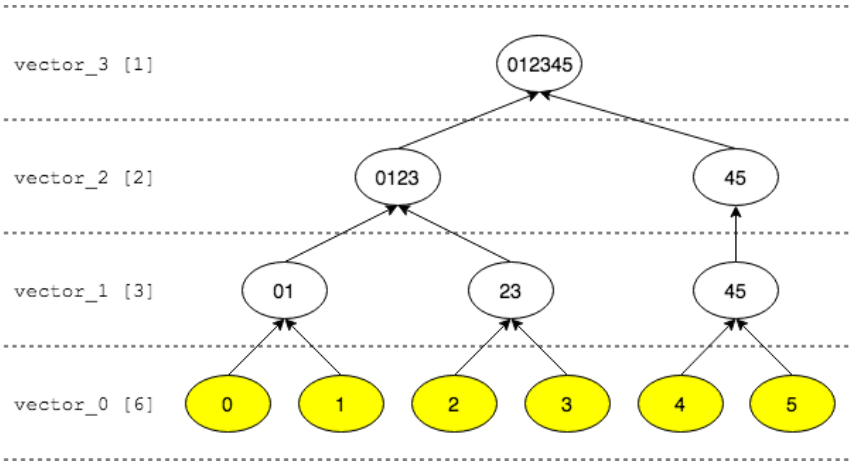


Figura 3: Ejemplo de árbol de Merkle con 5 nodos hoja

Cada uno de los vectores $\mathbf{v} = \langle v_1, v_2, \ldots, v_{n-1}, v_n \rangle$ irá reduciendo el numero de nodos o elementos mediante la siguiente función recursiva por intervalos o algoritmo de complejidad $O(n)$.

$$f(\mathbf{v}) = \sum_{i=0}^{n} \begin{cases} \langle \mathbf{v}', v_i \rangle & \text{si } i = n-1 \\ \langle \mathbf{v}', v_i || v_{i+1} \rangle & \text{si } i < n \\ i+2 \end{cases}$$

A continuación un ejemplo escrito en lenguaje C++ que implementa dicho algoritmo[8]:

```
#include <iostream>
#include <sstream>
#include <vector>
#include "sha256.h"

using namespace std;

void printVector(vector<string>);
string merkleTree(vector<string>);
string merkleTreeRoot;

int main(int argc, char *argv[]) {
    string msg = argv[1];
    istringstream buf(msg);
    vector<string> leafNodesV;

    for (string node; buf >> node;)
```

[8] https://github.com/JavierDominguezGomez/Cryptography/blob/master/MerkleTree/merkleTree.cpp

```
        leafNodesV.push_back(sha256(node));

    printVector(leafNodesV);
    cout << "Root: " << merkleTree(leafNodesV) << endl;

    return 0;
}

string merkleTree(vector<string> v) {
    if (v.size() > 1) {
        vector<string> aux;
        int i;

        for (i = 0; i < v.size(); i += 2) {
            if (i == v.size() - 1) {
                aux.push_back(v[i]);
            } else if (i < v.size()) {
                aux.push_back(sha256(v[i] + v[i + 1]));
            }
        }
        merkleTree(aux);
    } else if(v.size() == 1){
        merkleTreeRoot = v[0];
    }

    return merkleTreeRoot;
}

void printVector(vector<string> v) {
    cout << "v.size() = " << v.size() << endl;
    int i = 0;
    while (i < v.size()) {
        cout << "v[" << i << "]: " << v[i] << endl;
        i++;
    }
}
```

6. **timeStamp**: Se trata de un número entero sin signo de 4 *bytes* también llamado *Epoch* o *Tiempo Unix*. Representa el número de segundos que han transcurrido desde el 1 de enero de 1970 a las 00:00:00. Se codifica en formato *Little-endian*.

   Un ejemplo de cómo se puede calcular este dato es mediante el siguiente código[9] escrito en lenguaje C. Nótese que las horas se procesan como *GMT+1*.

```
#include <stdio.h>
#include <time.h>
```

[9] https://github.com/JavierDominguezGomez/Cryptography/blob/master/tools/date2epoch.c

```c
int main(int argc, char *argv[]) {
    int year, month, day, hour, minute, second;
    struct tm t;
    time_t tod;

    printf("Year: ");
    scanf("%d", &year);
    printf("Month: ");
    scanf("%d", &month);
    printf("Day: ");
    scanf("%d", &day);
    printf("Hour: ");
    scanf("%d", &hour);
    printf("Minute: ");
    scanf("%d", &minute);
    printf("Second: ");
    scanf("%d", &second);

    t.tm_year = year - 1900;
    t.tm_mon = month - 1;   // Values [0-11]
    t.tm_mday = day;
    t.tm_hour = hour + 1;   // GMT+1
    t.tm_min = minute;
    t.tm_sec = second;
    t.tm_isdst = 0;         // DST = 0
    tod = mktime(&t);

    printf("Timestamp epoch: %ld\n", (long) tod);
}
```

7. **targetDifficulty**: A este dato también se le llama simplemente *target* o "*Bits*"si se hace referencia al empaquetado de datos en el bloque. Se trata de un número entero de 256 *bits* representado como un número decimal muy grande, tanto que abarcaría el rango de números existentes entre 0 y $2^{256} - 1$. El siguiente número sería el valor hexadecimal máximo que podría tomar la variable *targetDifficulty*.

$$0x\underbrace{00000000ffffffffffffffffffffffffffffffffffffffffffffffffffffffff}_{256\ bits}$$

En el bloque se almacena como un número decimal de coma flotante truncando, por ejemplo, el valor hexadecimal anterior quedaría representado de la siguiente forma.

$$0x\underbrace{00000000ffff0000000000000000000000000000000000000000000000000000}_{256\ bits}$$

La dificultad es el resultado de dividir el valor máximo entre el valor

actual de la variable *targetDifficulty*, tal y como se muestra en la siguiente fórmula.

$$Dificultad = \frac{target\ maximo}{target\ actual} = \frac{\texttt{0x00000000ffff0000...00}}{\texttt{0x0000000019015f53...00}}$$

A continuación un fragmento de código escrito en lenguaje C++ que realiza el cálculo y que se puede emplear para hallar la dificultad de minado[10].

```
#include <iostream>
#include <cmath>

inline float fast_log(float val)
{
   int * const exp_ptr = reinterpret_cast <int *>(&val);
   int x = *exp_ptr;
   const int log_2 = ((x >> 23) & 255) - 128;
   x &= ~(255 << 23);
   x += 127 << 23;
   *exp_ptr = x;

   val = ((-1.0f/3) * val + 2) * val - 2.0f/3;
   return ((val + log_2) * 0.69314718f);
}

float difficulty(unsigned int bits)
{
    static double max_body = fast_log(0x00ffff),
                              scaland = fast_log(256);
    return exp(max_body - fast_log(bits & 0x00ffffff) +
               scaland * (0x1d - ((bits & 0xff000000) >> 24)));
}

int main()
{
    std::cout << difficulty(0x19015f53) << std::endl;
    return 0;
}
```

Hay que tener en cuenta el valor de la variable *targetDifficulty* en cada caso, por ejemplo en el bloque #286819 dicho valor es `0x19015f53`. Es una de las variables más importantes a tener en cuenta a la hora de obtener el *hash* adecuado del bloque antes de incorporarlo a la cadena como un bloque válido. Así pues la prueba de trabajo o *Proof of Work* tendrá mayor o menor dificultad. En el protocolo de *Bitcoin* se define una regla que dice que el *hash* del bloque ha de ser un número menor o igual al valor de esta variable *targetDifficulty* en ese momento. De modo que si el *hash*

[10] https://en.bitcoin.it/wiki/Difficulty

obtenido como candidato a generar un bloque fuese un número menor o igual al de *targetDifficulty* habría posibilidades para que el *hash* candidato sea válido para generar un nuevo bloque, aunque de forma adicional se han de cumplir otras condiciones. Por el contrario, si el *hash* obtenido como candidato fuera un número mayor que el valor de esta variable *targetDifficulty*, este quedaría desestimado, entonces habría que incrementar el valor de la variable *nonce* y repetir todo el proceso para generar un *hash* nuevo. El siguiente gráfico[11] muestra la variación de la dificultad a lo largo del tiempo.

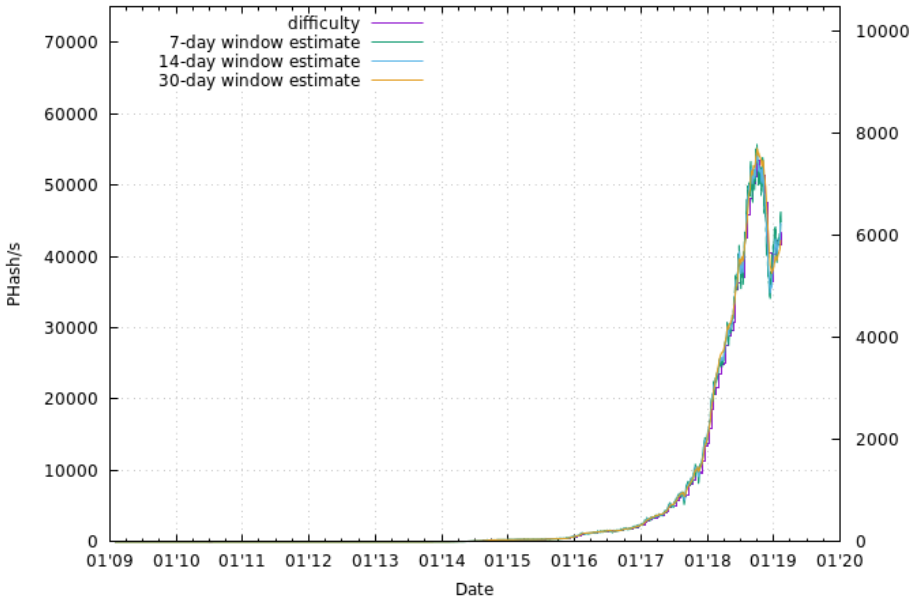


Figura 4: Dificultad de minado desde enero de 2009 hasta hoy.

El valor de la variable *targetDifficulty* se modifica automáticamente una vez se han generado 2016 bloques, esto sucede aproximadamente cada dos semanas. El nuevo valor se obtiene mediante un cálculo que realizan todos los clientes *Bitcoin* de la red en el que toman el tiempo real que ha llevado generar los 2016 bloques y se obtiene la diferencia porcentual respecto al número de bloques que se esperaba haber calculado en el periodo de dos semanas. Cuanto menor sea el valor de la variable *targetDifficulty* más aumentará la dificultad para hallar un *hash* válido para el bloque.

8. **nonce**: Se trata de un número entero sin signo aleatorio con una longitud de 32 *bits* o 4 *bytes* codificado en formato *Little-endian*. Es el dato que ha

[11] http://bitcoin.sipa.be/

de cambiarse tras un intento fallido por encontrar el *hash* del bloque adecuado, de modo que al incrementalo se han de realizar de nuevo todos los cálculos de la función SHA-256 teniendo en cuenta el nuevo valor de esta variable. Siguiendo con el ejemplo del bloque #286819, este se incorporó a la cadena de bloques con un valor decimal o base 10 para la variable *nonce* de 856192328, lo que indica casi con total seguridad que se tuvieron que realizar bastantes intentos.

9. **transactionCount**: En el caso de *transactionCount* el tipo de dato es un entero sin signo de longitud variable. Como el propio nombre indica, dependiendo del número de transacciones que han sido procesadas tendrá un valor numérico u otro.

   *Cuando se le asigna por valor un entero muy grande se codifica en formato *Little-endian*.

2.1. Datos de la cabecera o header

La siguiente imagen es una captura de pantalla de los datos del bloque #286819 visualizados en un explorador de bloques. Se han resaltado los datos que forman parte de la cabecera o *header* del bloque.

Summary	
Height	286819 (Main chain)
Hash	000000000000000e067a478024addfecdc93628978aa52d91fabd4292982a50
Previous Block	0000000000000117c80378b8da0e33559b5997f2ad55e2f7d18ec1975b9717
Next Blocks	000000000000000b0f08ec6a3d1e84994498ecf993a9981f57982cfdb66c443
Time	2014-02-20 04:57:25
Received Time	2014-02-20 04:57:40
Relayed By	GHash.IO
Difficulty	3,129,573,174.52
Bits	419520339
Number Of Transactions	99
Output Total	3,925.7198604 BTC
Estimated Transaction Volume	141.5425474 BTC
Size	152.509 KB
Version	2
Merkle Root	871714dcbae6c8193a2bb9b2a69fe1c0440399f38d94b3a0f1b447275a29978a
Nonce	856192328
Block Reward	25 BTC
Transaction Fees	0.03049675 BTC

Figura 5: Datos empleados para la obtención del *hash* del bloque #286819.

3. Construcción de la cadena de entrada M

Siguiendo con el ejemplo del bloque #286819 de la cadena de bloques de *Bitcoin*, dicho bloque está representado mediante el siguiente *hash*.

```
0000000000000000e067a478024addfecdc93628978aa52d91fabd4292982a50
```

A continuación los datos correspondientes a la cabecera o *header* del bloque:

```
Version:      2
Prev. Block:  00000000000000117c80378b8da0e33559b5997f2ad55e2f7d18ec1975b9717
Merkle root:  871714dcbae6c8193a2bb9b2a69fe1c0440399f38d94b3a0f1b447275a29978a
Timestamp:    2014-02-20 04:57:25 (Epoch timestamp: 1392872245)
Bits:         419520339
Nonce:        856192328
```

En este punto hay que pasar los datos en formato decimal a formato hexadecimal, de modo que quedarían de la siguiente manera:

```
Version:      00000002
Prev. Block:  00000000000000117c80378b8da0e33559b5997f2ad55e2f7d18ec1975b9717
Merkle root:  871714dcbae6c8193a2bb9b2a69fe1c0440399f38d94b3a0f1b447275a29978a
Timestamp:    53058B35
Bits:         19015F53
Nonce:        33087548
```

Ahora hay que reorganizar los datos en formato *Little-Endian*:

```
Version:      02000000
Prev. Block:  17975b97c18ed1f7e255adf297599b55330edab87803c81701000000000000000
Merkle root:  8a97295a2747b4f1a0b3948df3990344c0e19fa6b2b92b3a19c8e6badc141787
Timestamp:    358B0553
Bits:         535F0119
Nonce:        48750833
```

Finalmente se concatenan uno a continuación de otro, empezando por *version*, seguido del *hash* del bloque anterior, *merkle root*, *timestamp*, *bits* y *nonce*, formando una cadena de entrada M de 160 carácteres hexadecimales con con un tamaño total de 640 *bits*.

$$M = \begin{cases} 02000000\,17975b97c18ed1f7e255adf2 \\ 97599b55330edab87803c81701000000 \\ 000000008a97295a2747b4f1a0b3948d \\ f3990344c0e19fa6b2b92b3a19c8e6ba \\ dc141787358b0553535f011948750833 \end{cases}$$

Se segmenta la cadena de entrada M en bloques de 32 *bits*:

$$M = \begin{cases} \texttt{02000000 + 17975b97 + c18ed1f7 + e255adf2} \\ \texttt{97599b55 + 330edab8 + 7803c817 + 01000000} \\ \texttt{00000000 + 8a97295a + 2747b4f1 + a0b3948d} \\ \texttt{f3990344 + c0e19fa6 + b2b92b3a + 19c8e6ba} \\ \texttt{dc141787 + 358b0553 + 535f0119 + 48750833} \end{cases}$$

De este modo se obtiene el mensaje de entrada M. Se ha de reservar este dato para utilizarlo más adelante en la función SHA-256 como *input* en la primera ronda.

3.1. Longitud de la cadena de entrada M

Una vez que se ha obtenido la cadena de entrada M en el punto anterior es necesario calcular la longitud de la misma en formato hexadecimal o base 16, es decir, 640 *bits* que se representan con el valor 280 en hexadecimal.

$$|M| = 280 \ (640 \ bits \ del \ mensaje \ original \ en \ hexadecimal)$$

Este dato se ha de reservar para el siguiente punto, ya que será necesario para completar los registros W_{14} y W_{15} de la variable W_t solo en la segunda y tercera ejecución de la función SHA-256.

3.2. La variable W_t

Tal y como se describe en el punto 3 del documento "*Criptografía aplicada: Función SHA-256*"[12], la variable W_t es un vector de 64 elementos que contiene palabras hexadecimales de 32 *bits*. Tiene un tamaño o longitud de 2048 *bits* (256 *bytes*) y se obtiene mediante la siguiente función recursiva definida por intervalos.

$$W_t = \begin{cases} M_i & si \quad 0 \leq i < 16 \\ \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16} & si \quad 16 \leq i < 64 \end{cases}$$

En la anterior función recursiva definida por intervalos, el primer intervalo es el que abarca los 16 primeros registros, o sea desde W_0 hasta W_{15}. El segundo intervalo es el esquema de los 48 registros restantes, es decir, desde W_{16} hasta W_{63}.

[12] https://github.com/JavierDominguezGomez/Cryptography/blob/master/SHA-256/Cryptography_SHA-256_es.pdf

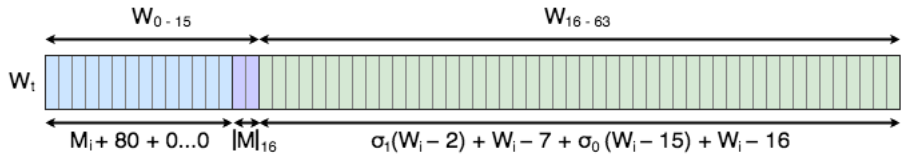


Figura 6: Representación gráfica de W_t

Las funciones σ_0 y σ_1 realizan las siguientes operaciones lógicas de compresión con cada uno de los *bits* de la palabra almacenada en el segmento de W_t que procesan.

$$\sigma_0(x) = ROT\ R^7(x) \oplus ROT\ R^{18}(x) \oplus SHR^3(x)$$
$$\sigma_1(x) = ROT\ R^{17}(x) \oplus ROT\ R^{19}(x) \oplus SHR^{10}(x)$$

Para hallar el valor de cualquiera de los segmentos que van desde W_{16} hasta W_{63} se han de realizar las siguientes operaciones, donde i tiene por valor el valor de t en el segmento que se quiere calcular de W_t. Por ejemplo, para W_{16}:

$$\sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16}$$
$$\Downarrow$$
$$\sigma_1(W_{16-2}) + W_{16-7} + \sigma_0(W_{16-15}) + W_{16-16}$$
$$\Downarrow$$
$$\sigma_1(W_{14}) + W_9 + \sigma_0(W_1) + W_0$$

En este ejemplo los segmentos W_{14}, W_9, W_1 y W_0 de la primera ronda (*Ronda #0*) tienen asignados los siguientes valores:

$$W_{14} = \texttt{0xb2b92b3a} \quad W_9 = \texttt{0x8a97295a}$$
$$W_1 = \texttt{0x17975b97} \quad W_0 = \texttt{0x02000000}$$

A continuación se muestra una representación gráfica la operación lógica de compresión σ_1, que aplica sobre el valor del segmento W_{14}.

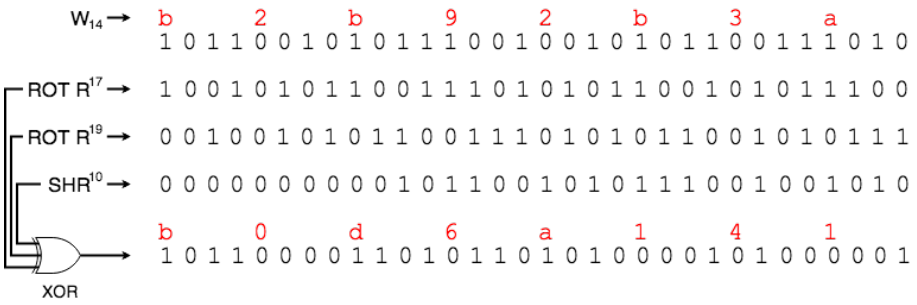


Figura 7: Operación $ROT\ R^{17}(x) \oplus ROT\ R^{19}(x) \oplus SHR^{10}(x)$ con cada *bit*.

Tras el cálculo de $\sigma_1(W_{14})$ se obtiene la palabra hexadecimal de 32 *bits* `0xb0d6a141`. Se ha de reservar este valor para calcular el resultado final. También se muestra la la representación gráfica la operación lógica de compresión σ_0, que aplica sobre el valor del segmento W_1.

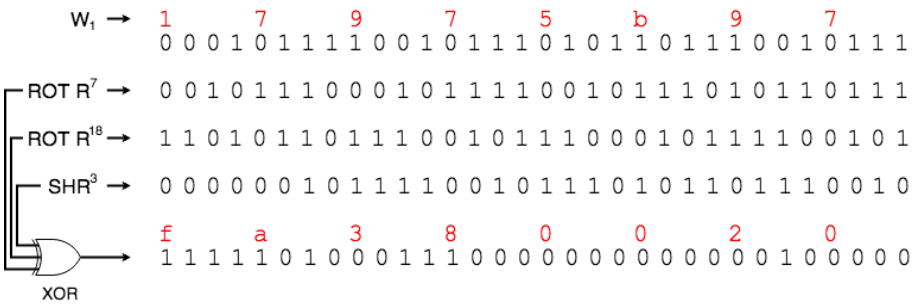


Figura 8: Operación $ROT\ R^7(x) \oplus ROT\ R^{18}(x) \oplus SHR^3(x)$ con cada *bit*.

Tras el cálculo de $\sigma_0(W_1)$ se obtiene la palabra hexadecimal de 32 *bits* `fa380020`. Una vez realizados los cálculos necesarios para obtener el valor de $\sigma_1(W_{14})$ y $\sigma_0(W_1)$ finalmente se ha de realizar la siguiente operación final.

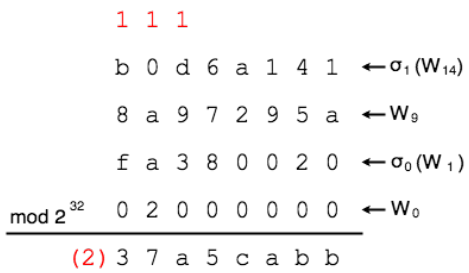


Figura 9: Operación final $mod\ 2^{32}(\sigma_1(W_{14}), W_9, \sigma_0(W_1), W_0)$.

De este modo se obtiene el valor que se almacenará en el segmento W_{16} del vector W_t. Será necesario repetir el mismo proceso por cada uno de los segmentos hasta llegar a W_{63}.

Todos los cálculos anteriores también aplican a las reglas criptográficas de los bloques en el protocolo *Bitcoin*, solo que hay que tener en cuenta que la cadena de entrada M en cada bloque tiene una longitud superior a 512 *bits*, lo que implica utilizar la función *hash* SHA-256 tres veces, una por cada ronda, y el valor de la longitud de M solo se aplicará en el esquema de relleno de la segunda y tercera ronda. En la primera ronda (*Ronda #0*) se rellenan los primeros 16 segmentos de la variable W_t con todos los *bytes* en hexadecimal que quepan, es decir, los primeros 512 *bits* de M, rellenando también los segmentos W_{14} y W_{15}.

Los 128 *bits* restantes de la cadena de entrada M que no caben en esta primera ronda se reservan para los primeros 4 registros de W_t de la segunda ronda. Tras la segunda ronda (*Ronda #1*) se obtiene un *digest* de 256 *bits* que se utiliza para rellenar los primeros 8 segmentos de W_t en la tercera y última ronda (*Ronda #2*). La siguiente figura se detalla el contenido de los primeros 16 registros de la variable W_t en cada una de las tres veces que se ejecutan las 64 iteraciones de la función SHA-256 para obtener el *digest* en cada caso.

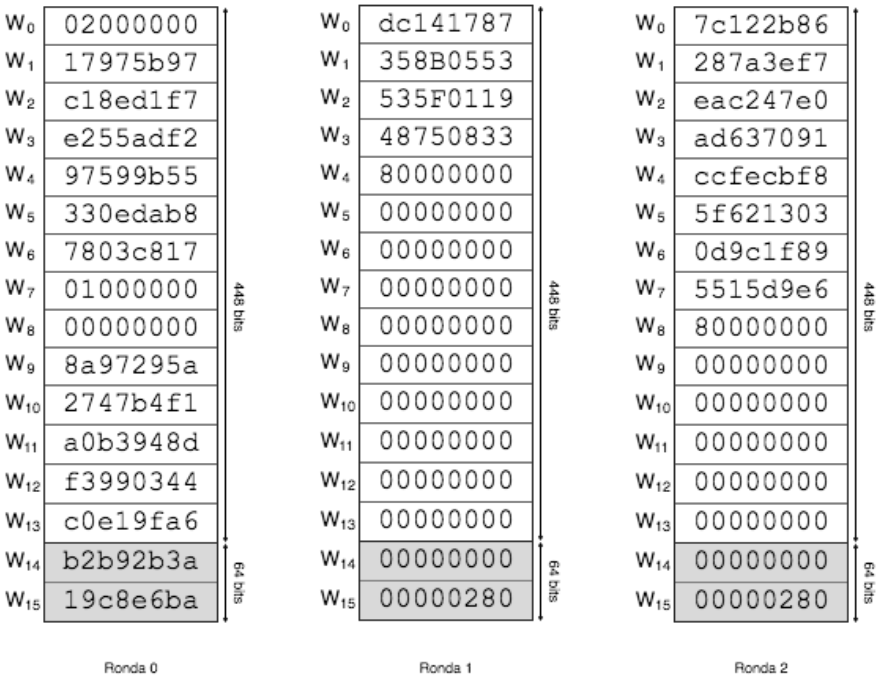
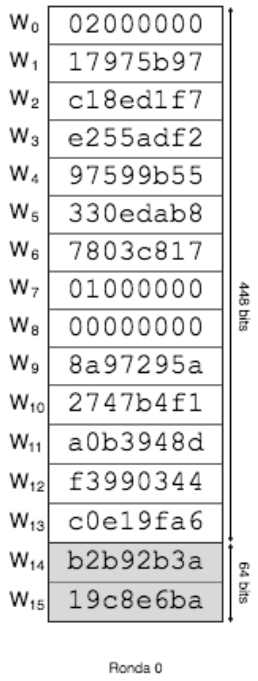


Figura 10: Los primeros 16 registros de la variable W_t en los tres casos en los que se ha de ejecutar la función SHA-256.

En los siguientes puntos se detalla el esquema de relleno al completo en cada una de las rondas.

3.3. Primera ronda SHA-256: Ronda 0

La función *hash* SHA-256 realiza 64 ciclos criptográficos en los que procesará una serie de cálculos en los que tiene en cuenta por un lado las 8 palabras hexadecimales de 32 *bits* asignadas como valor a las variables A, B, C, D, E, F, G y H, y por otro lado la variable W_t, que es un vector de 64 elementos ordenados por intervalos. En la primera ronda criptográfica se han iniciar los



valores de las variables A_0, B_0, C_0, D_0, E_0, F_0, G_0 y H_0 con los valores estándar de la función *hash* SHA-256 en su primera iteración, es decir, los 32 primeros *bits* en hexadecimal o base 16 de la parte fraccionaria de las raíces cuadradas de los primeros 8 números primos.

$$A_0 = \texttt{0x6a09e667} \quad B_0 = \texttt{0xbb67ae85}$$
$$C_0 = \texttt{0x3c6ef372} \quad D_0 = \texttt{0xa54ff53a}$$
$$E_0 = \texttt{0x510e527f} \quad F_0 = \texttt{0x9b05688c}$$
$$G_0 = \texttt{0x1f83d9ab} \quad H_0 = \texttt{0x5be0cd19}$$

En una función SHA-256, cuando el tamaño de la cadena de entrada M es igual o mayor que 512 *bits* se han de rellenar los primeros 16 elementos del vector W_t con los primeros 512 *bits* de la cadena principal de entrada M, incluidos los elementos W_{14} y W_{15}, tal y como se muestra en la imagen. El resto de la cadena M se ha de reservar para la segunda ejecución de la función SHA-256 que se explicará en el siguiente punto. Teniendo la cadena M de entrada, las 8 palabras iniciales, que irán cambiando en cada una de las 64 iteraciones, y los 64 elementos del vector W_t se puede comenzar a realizar las operaciones de la función SHA-256. El objetivo en esta primera ronda es obtener el primer *hash*, necesario en la segunda ronda.

Cuando se hayan realizado las 64 iteraciones SHA-256 con los datos anteriores se ha de realizar una última operación en la que se ha de tomar cada palabra A, B, C, D, E, F, G y H de la primera iteración y realizar una operación *mod* 2^{32} con su homóloga de la última iteración, véase el ejemplo.

$$\begin{array}{llllll}
\texttt{0x6a09e667}\ (A_0) & \textit{mod}\ 2^{32} & \texttt{0x72605526}\ (A_{63}) & = & \texttt{0xdc6a3b8d} \\
\texttt{0xbb67ae85}\ (B_0) & \textit{mod}\ 2^{32} & \texttt{0x51019395}\ (B_{63}) & = & \texttt{0x0c69421a} \\
\texttt{0x3c6ef372}\ (C_0) & \textit{mod}\ 2^{32} & \texttt{0x8eab60c2}\ (C_{63}) & = & \texttt{0xcb1a5434} \\
\texttt{0xa54ff53a}\ (D_0) & \textit{mod}\ 2^{32} & \texttt{0x3fe7029b}\ (D_{63}) & = & \texttt{0xe536f7d5} \\
\texttt{0x510e527f}\ (E_0) & \textit{mod}\ 2^{32} & \texttt{0x72b36765}\ (E_{63}) & = & \texttt{0xc3c1b9e4} \\
\texttt{0x9b05688c}\ (F_0) & \textit{mod}\ 2^{32} & \texttt{0xb1b63303}\ (F_{63}) & = & \texttt{0x4cbb9b8f} \\
\texttt{0x1f83d9ab}\ (G_0) & \textit{mod}\ 2^{32} & \texttt{0x766c3d83}\ (G_{63}) & = & \texttt{0x95f0172e} \\
\texttt{0x5be0cd19}\ (H_0) & \textit{mod}\ 2^{32} & \texttt{0xa06805c6}\ (H_{63}) & = & \texttt{0xfc48d2df}
\end{array}$$

Finalmente se concatenan los valores de los 8 resultados en esta ronda 0 obteniendo el siguiente digest resultante:

```
dc6a3b8d0c69421acb1a5434e536f7d5c3c1b9e44cbb9b8f95f0172efc48d2df
```

Se reserva este dato para realizar la segunda ronda SHA-256 en el siguiente

punto.

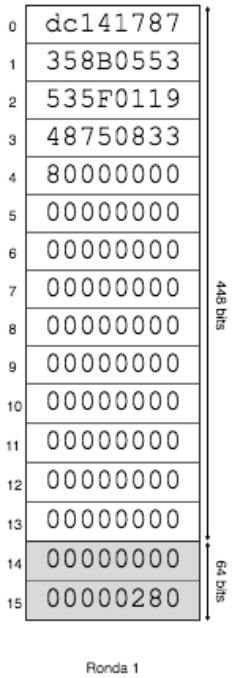
3.4. Segunda ronda SHA-256: Ronda 1

En la segunda ronda se han volver a realizar los 64 ciclos criptográficos que realiza SHA-256 teniendo en cuenta que el valor de las 8 palabras iniciales A, B, C, D, E, F, G y H en la primera iteración tendrán por valor inicial cada uno de los segmentos de 32 *bits* correspondientes al *hash* resultante de la primera ronda.

$$\overbrace{\texttt{dc6a3b8d}}^{A_0}\ \overbrace{\texttt{0c69421a}}^{B_0}\ \overbrace{\texttt{cb1a5434}}^{C_0}\ \overbrace{\texttt{e536f7d5}}^{D_0}\ \overbrace{\texttt{c3c1b9e4}}^{E_0}\ \overbrace{\texttt{4cbb9b8f}}^{F_0}\ \overbrace{\texttt{95f0172e}}^{G_0}\ \overbrace{\texttt{fc48d2df}}^{H_0}$$

$$\begin{aligned} A_0 &= \texttt{0xdc6a3b8d} & B_0 &= \texttt{0xc3c1b9e4} \\ C_0 &= \texttt{0x0c69421a} & D_0 &= \texttt{0x4cbb9b8f} \\ E_0 &= \texttt{0xcb1a5434} & F_0 &= \texttt{0x95f0172e} \\ G_0 &= \texttt{0xe536f7d5} & H_0 &= \texttt{0xfc48d2df} \end{aligned}$$

En cuanto a los primeros 16 registros de la variable W_t en esta segunda ronda se se utilizarán primero los *bits* del mensaje M original de 640 *bits* que no cabían en la ronda anterior.



Ronda 1

$$\texttt{dc141787} + \texttt{358b0553} + \texttt{535f0119} + \texttt{48750833}$$

A continuación hay que tomar un *bit* que represente el número 1 decimal o base 10, es decir 00000001, este se desplaza al *bit* más alto del *byte*, con lo que se obtiene 10000000 y finalmente se calcula el valor hexadecimal, que es 80.

$$10000000_2 = 80_{16}$$

Independientemente de la longitud de cadena hexadecimal de la palabra de entrada, se añade 80 por la derecha.

$$\texttt{486f6c61} + \texttt{206d756e} + \texttt{646f} + \texttt{80}$$

Ahora hay que añadir a la cadena una cantidad de *bits* con valor 0 hasta llegar a 448 *bits* en total, que es la longitud que abarca todos los intervalos que van desde W_0 hasta W_{13}. Para terminar de completar los primeros 16 segmentos de W_t solo queda rellenar los últimos dos bloques de 32 *bits* W_{14} y W_{15} con la longitud del mensaje de entrada $|M|$ en hexadecimal,

con tantos ceros por la izquierda como sean necesario para alcanzar 64 *bits* de longitud, en este caso es 280. Los valores que irán en los segmentos desde W_{16} hasta W_{63} se obtienen siguiendo las pautas explicadas en el punto 6 de este documento. Al igual que en la primera ronda, cuando se hayan realizado las 64 iteraciones SHA-256 con los datos anteriores se ha de realizar una última operación en la que se ha de tomar cada palabra A, B, C, D, E, F, G y H de la primera iteración y realizar una operación *mod* 2^{32} con su homóloga de la última iteración, véase el ejemplo.

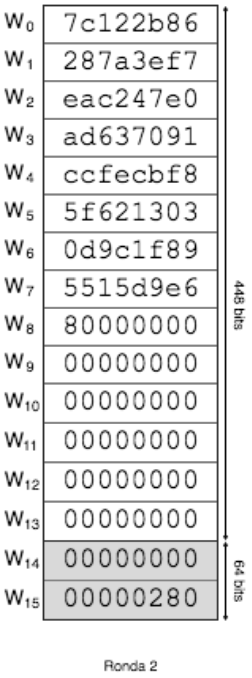
$$
\begin{array}{lcl}
\texttt{0xdc6a3b8d}\ (A_0)\ mod\ 2^{32}\ \texttt{0x9fa7eff9}\ (A_63) & = & \texttt{0x7c122b86} \\
\texttt{0x0c69421a}\ (B_0)\ mod\ 2^{32}\ \texttt{0x1c10fcdd}\ (B_63) & = & \texttt{0x287a3ef7} \\
\texttt{0xcb1a5434}\ (C_0)\ mod\ 2^{32}\ \texttt{0x1fa7f3ac}\ (C_63) & = & \texttt{0xeac247e0} \\
\texttt{0xe536f7d5}\ (D_0)\ mod\ 2^{32}\ \texttt{0xc82c78bc}\ (D_63) & = & \texttt{0xad637091} \\
\texttt{0xc3c1b9e4}\ (E_0)\ mod\ 2^{32}\ \texttt{0x093d1214}\ (E_63) & = & \texttt{0xccfecbf8} \\
\texttt{0x4cbb9b8f}\ (F_0)\ mod\ 2^{32}\ \texttt{0x12a67774}\ (F_63) & = & \texttt{0x5f621303} \\
\texttt{0x95f0172e}\ (G_0)\ mod\ 2^{32}\ \texttt{0x77ac085b}\ (G_63) & = & \texttt{0x0d9c1f89} \\
\texttt{0xfc48d2df}\ (H_0)\ mod\ 2^{32}\ \texttt{0x58cd0707}\ (H_63) & = & \texttt{0x5515d9e6}
\end{array}
$$

Finalmente se concatenan los valores de los 8 resultados en esta ronda 1 obteniendo el siguiente digest resultante:

```
7c122b86287a3ef7eac247e0ad637091ccfecbf85f6213030d9c1f895515d9e6
```

Se reserva este dato para realizar la tercera ronda SHA-256 en el siguiente punto.

3.5. Tercera ronda SHA-256: Ronda 2



Ronda 2

En esta tercera ronda criptográfica (*Ronda #2*) se ha de utilizar como mensaje de entrada M el *hash* obtenido en la ronda anterior (*Ronda #1*). Al igual que en la ronda anterior hay que tomar un *bit* que represente el número 1 decimal o base 10, es decir 00000001, este se desplaza al *bit* más alto del *byte*, con lo que se obtiene 10000000 y finalmente se calcula el valor hexadecimal, que es 80.

$$10000000_2 = 80_{16}$$

Independientemente de la longitud de cadena hexadecimal de la palabra de entrada, se añade 80 por la derecha.

```
7c122b86 + 287a3ef7 + eac247e0 +
ad637091 + ccfecbf8 + 5f621303 +
0d9c1f89 + 5515d9e6 + 80
```

Ahora hay que añadir a la cadena una cantidad de *bits* con valor 0 hasta llegar a 448 *bits* en total, que es la longitud que abarca todos los intervalos que van desde W_0 hasta W_{13}. Cuando se hayan realizado las 64 iteraciones SHA-256 con los datos anteriores se ha de realizar una última operación en la que se ha de tomar cada palabra A, B, C, D, E, F, G y H de la primera iteración y realizar una operación *mod* 2^{32} con su homóloga de la última iteración, véase el ejemplo.

0x6a09e667 (A_0)	*mod* 2^{32}	0xe620b22b (A_{63})	=	0x502a9892
0xbb67ae85 (B_0)	*mod* 2^{32}	0x87564c0c (B_{63})	=	0x42bdfa91
0x3c6ef372 (C_0)	*mod* 2^{32}	0xf1369725 (C_{63})	=	0x2da58a97
0xa54ff53a (D_0)	*mod* 2^{32}	0x82e6d493 (D_{63})	=	0x2836c9cd
0x510e527f (E_0)	*mod* 2^{32}	0xadcef783 (E_{63})	=	0xfedd4a02
0x9b05688c (F_0)	*mod* 2^{32}	0xdd9eff54 (F_{63})	=	0x78a467e0
0x1f83d9ab (G_0)	*mod* 2^{32}	0xe07c2655 (G_{63})	=	0x00000000
0x5be0cd19 (H_0)	*mod* 2^{32}	0xa41f32e7 (H_{63})	=	0x00000000

Finalmente se concatenan los valores de los 8 resultados en esta ronda 0 obteniendo el siguiente digest resultante:

```
502a989242bdfa912da58a972836c9cdfedd4a0278a467e00000000000000000
```

Para terminar, el último paso para obtener el *hash* definitivo de este bloque es convertir el *digest* resultante a formato *Little-Endian*, tal y como se muestra

a continuación.

```
000000000000000e067a478024addfecdc93628978aa52d91fabd4292982a50
```

Se puede comprobar que se cumplen los requisitos del protocolo *Bitcoin* para el cálculo del *hash* del bloque #286819 de la cadena de bloques de *Bitcoin* en la red principal *mainnet*. Los datos empleados en la realización de este documento se pueden comprobar con los de la cadena de bloques de *Bitcoin* desde en cualquier explorador de bloques[13].

4. Almacenamiento en ficheros

Todos los datos de los bloques que ya han sido minados se escriben en ficheros binarios con el nombre **blk*.dat**. Estos archivos se almacenan en diferentes directorios dependiendo del sistema operativo que tenga la computadora donde se esté ejecutando el cliente *Bitcoin Core*, véase la siguiente tabla:

Sistema operativo	Directorio
GNU/Linux y Unix	∼/.bitcoin/blocks/
MacOS	∼/Library/Application Support/Bitcoin/blocks
Windows XP	%APPDATA %\Bitcoin\blocks
Windows Vista/7/8	C:\Users\user\AppData\Roaming\Bitcoin\blocks

4.1. Listado y lectura de archivos

En este ejemplo se ha empleado un sistema operativo 100 % libre[14] GNU/Linux, así pues hay que posicionarse en el directorio ∼/.bitcoin/blocks/. Si se listan los archivos que contiene el directorio aparecerá una lista como la siguiente:

```
∼/.bitcoin/blocks/$ ls -lrt
-rw------- 1 jdg jdg 134211184 oct 7 2018 blk00000.dat
-rw------- 1 jdg jdg 134182654 oct 7 2018 blk00001.dat
-rw------- 1 jdg jdg 134185203 oct 7 2018 blk00002.dat
-rw------- 1 jdg jdg 134205295 oct 7 2018 blk00003.dat
-rw------- 1 jdg jdg 134206001 oct 7 2018 blk00004.dat
-rw------- 1 jdg jdg 134215469 oct 7 2018 blk00005.dat
...
-rw------- 1 jdg jdg  19503795 oct 7 2018 rev00000.dat
-rw------- 1 jdg jdg  16777216 oct 7 2018 rev00002.dat
-rw------- 1 jdg jdg  16965106 oct 7 2018 rev00001.dat
-rw------- 1 jdg jdg  16777216 oct 7 2018 rev00003.dat
...
drwxr-x--- 2 jdg jdg      4096 oct 7 2018 index
```

Entre todos estos archivos los que nos interesan son los que su nombre comienza por **blk** seguido de unos números y terminando con la extensión **.dat**. Al tratarse de archivos binarios no se puede leer su contenido con un editor de

[13] https://www.blockchain.com/en/btc/block-height/286819
[14] https://www.gnu.org/distros/free-distros.es.html

textos tradicional, será necesario utilizar un programa específico para tal caso, como por ejemplo el programa **hexdump**[15]. Una vez abierto cualquiera de los archivos, por ejemplo el primer archivo **blk00000.dat**, así es como se ven los datos que contiene.

```
~/.bitcoin/blocks/$ hexdump -C -s 0 -n 288 blk00000.dat
00000000 f9 be b4 d9 1d 01 00 00 01 00 00 00 00 00 00 00 |................|
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000020 00 00 00 00 00 00 00 00 00 00 00 00 3b a3 ed fd |............;...|
00000030 7a 7b 12 b2 7a c7 2c 3e 67 76 8f 61 7f c8 1b c3 |z{..z.,>gv.a....|
00000040 88 8a 51 32 3a 9f b8 aa 4b 1e 5e 4a 29 ab 5f 49 |..Q2:...K.^J).._I|
00000050 ff ff 00 1d 1d ac 2b 7c 01 01 00 00 00 01 00 00 |......+|........|
00000060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ff ff |................|
00000080 ff ff 4d 04 ff ff 00 1d 01 04 45 54 68 65 20 54 |..M.......EThe T|
00000090 69 6d 65 73 20 30 33 2f 4a 61 6e 2f 32 30 30 39 |imes 03/Jan/2009|
000000a0 20 43 68 61 6e 63 65 6c 6c 6f 72 20 6f 6e 20 62 | Chancellor on b|
000000b0 72 69 6e 6b 20 6f 66 20 73 65 63 6f 6e 64 20 62 |rink of second b|
000000c0 61 69 6c 6f 75 74 20 66 6f 72 20 62 61 6e 6b 73 |ailout for banks|
000000d0 ff ff ff ff 01 00 f2 05 2a 01 00 00 00 43 41 04 |........*....CA.|
000000e0 ...
```

Estos son los primeros *bytes* del primer bloque en el primer archivo de la cadena de bloques de Bitcon, también llamado *bloque génesis*. Este primer archivo no solo contiene los datos del primer bloque, hay muchos más, ordenados cronológicamente según se van minando y añadiendo a la cadena. Tal y como se explica al inicio del punto 2 de este documento, los datos de los bloques se van concatenando uno a continuación de otro.

4.2. Visualización de datos los datos en el bloque

Siguiendo con el ejemplo del bloque #286819, en condiciones normales, es decir, siempre y cuando a la cadena de bloques descargada no se le haya aplicado algún tipo de *prune*[16] o podado, los datos se encuentran en el archivo **blk00116.dat**. Todos los archivos comienzan con los datos de un bloque pero los datos del bloque que se quiere analizar no tienen por qué encontrarse en esa posición, es bastante probable que los datos comiencen muchos *bytes* más adelante o en un *offset* mucho mayor desde el inicio del archivo. En este caso concreto, los datos comienzan a partir del *byte* número 93725126, en el *offset* número `059621c6`. En el siguiente ejemplo se ejecuta el programa **hexdump** con la opción **-C** para obtener una salida *standard* por pantalla, también con la opción **-s** seguido del número de desde el que se quiere empezar a visualizar datos, y la opción **-n** para indicar el número total de *bytes* que se quieren mostrar, en este caso con 288 *bytes* es suficiente.

[15] https://www.freebsd.org/cgi/man.cgi?query=hexdump&sektion=1
[16] https://en.bitcoin.it/wiki/Running_Bitcoin#Command-line_arguments

```
~/.bitcoin/blocks/$ hexdump -C -s 93725126 -n 288 blk00116.dat
059621c6 f9 be b4 d9 bd 53 02 00 02 00 00 00 17 97 5b 97 |.....S........[.|
059621d6 c1 8e d1 f7 e2 55 ad f2 97 59 9b 55 33 0e da b8 |.....U...Y.U3...|
059621e6 78 03 c8 17 01 00 00 00 00 00 00 00 8a 97 29 5a |x.............)Z|
059621f6 27 47 b4 f1 a0 b3 94 8d f3 99 03 44 c0 e1 9f a6 |'G.........D....|
05962206 b2 b9 2b 3a 19 c8 e6 ba dc 14 17 87 35 8b 05 53 |..+:........5..S|
05962216 53 5f 01 19 48 75 08 33 63 01 00 00 00 01 00 00 |S_..Hu.3c.......|
05962226 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
05962236 00 00 00 00 00 00 00 00 00 00 00 00 00 ff ff ff |................|
05962246 ff ff 60 03 63 60 04 06 2f 50 32 53 48 2f 04 35 |..`.c`../P2SH/.5|
05962256 8b 05 53 08 44 04 f2 53 00 00 17 e4 46 52 2c fa |..S.D..S....FR,.|
05962266 be 6d 6d 69 06 88 fb 88 6c 0d f0 c8 7c bc 7e a4 |.mmi....l...|. .|
05962276 f7 f1 b5 c0 05 0b d0 ac 37 51 cf c9 97 d9 d6 97 |........7Q......|
05962286 13 28 de 04 00 00 00 00 00 00 00 48 61 70 70 79 |.(.........Happy|
05962296 20 4e 59 21 20 59 6f 75 72 73 20 47 48 61 73 68 | NY! Yours GHash|
059622a6 2e 49 4f 00 00 00 00 01 cb 81 31 95 00 00 00 00 |.IO.......1.....|
059622b6 19 76 a9 14 80 ad 90 d4 03 58 1f a3 bf 46 08 6a |.v.......X...F.j|
059622c6 91 b2 d9 d4 12 5d b6 c1 88 ac 00 00 00 00 01 00 |.....]..........|
059622d6 00 00 01 7d 67 7c de 17 3f 8c bf 43 31 27 a8 5e |...}g|..?..C1'.^|
059622e6 ...
```

En la columna izquierda se ve el *offset* correspondiente a cada línea, en la parte central los datos del bloque en formato hexadecimal y la columna de la derecha cada uno de los 16 datos representados en caracteres ASCII[17]. Para facilitar la lectura se ha resaltado en diferentes colores los diferentes segmentos de datos. En color rojo aparecen los 4 *bytes* que representan el dato **magicID**, a continuación en color verde los 4 siguientes *bytes* el dato **size** que representa el tamaño final de este bloque. El segmento que sigue en color turquesa son los 80 *bytes* que representan el **header**. En color azul un único *byte* que representa el número de transacciones registradas en el bloque y finalmente en color morado se han resaltado todos los datos correspondientes a las transacciones del bloque. Los datos no terminan en el *offset* `059622e6`, se ha truncado el bloque en ese punto para une mejor visualización en este documento, pues la cantidad de datos del bloque mostrada en este formato podría abarcar demasiado espacio.

[17] https://www.ieee.li/computer/ascii.htm