Sistemas de restricciones algebraicas para pruebas criptográficas aplicado a SHA-256

TRABAJO DE FIN DE GRADO
Curso 2021-2022

MIGUEL MORONA MÍNGUEZ

Facultad de ciencias matemáticas
Grado en ingeniería matemática

29 de junio de 2022

Directores:

Dario Fiore
Ignacio Cascudo Pueyo
Ignacio Luengo Velasco

Resumen

Los sistemas de pruebas criptográficas pretenden demostrar la veracidad de un enunciado. Entre otros propósitos, estas pueden certificar la correcta computación de un cálculo. Muchos sistemas de pruebas criptográficas emplean un método para comprobar dicha corrección, denominado R1CS (*Rank-1 Constraint System*). R1CS es esencialmente una terna de matrices con un vector *witness* que representan las relaciones internas de una función matemática de manera algebraica.

El objetivo de este Trabajo de Fin de Grado es obtener un sistema R1CS óptimo en lo que se refiere a número de relaciones algebraicas y variables para la función hash SHA-256 en el cuerpo binario $\mathbb{F}_2$. Para ello, se construyen sistemas R1CS óptimos para componentes dentro de SHA-256 y se formaliza un método para reunirlas todas en un sistema común. Los resultados finales incluyen la implementación de un algoritmo para generar la terna de matrices, cada una con 23296 filas y 26113 columnas, y su respectivo vector *witness*.

Abstract

Cryptographic proof systems aim to prove a mathematical statement. Among other purposes, they can certify the correctness of a computation. Many cryptographic proof systems employ a method for checking such correctness, called R1CS (*Rank-1 Constraint System*). R1CS is essentially a triplet of matrices and a *witness* vector that represent the internal relationships of a mathematical function in an algebraic manner.

The objective of this Bachelor Thesis is to obtain an optimal R1CS system in terms of number of algebraic relations and variables for the SHA-256 hash function in the binary field $\mathbb{F}_2$. For this purpose, optimal R1CS systems are constructed for components within SHA-256. Then, a framework for gathering all of them into a common system is formalized. The final results include the implementation of an algorithm that generates the three R1CS matrices, each with 23296 rows and 26113 columns, and their respective *witness* vector.

Agradecimientos

Este trabajo es el resultado de 5 maravillosos meses de trabajo en el instituto IMDEA Software. Gracias a Ignacio y a Dario por hacerme disfrutar de esta experiencia y enseñarme todo el trabajo que hacéis, que es increíble. Aprecio mucho el tiempo dedicado y los conocimientos que me habéis compartido.

Me gustaría señalar también el gran esfuerzo de muchas otras personas que han contribuido a que este trabajo saliese adelante, como es el de David Balbás. Gracias por tu dedicación, por tus ganas y por tener tan buena actitud conmigo. Sin ti, este proyecto no habría sido posible. Gracias a Miguel Ambrona, por hablarme sobre IMDEA y mostrarme siempre tanto interés que agradezco enormemente. Dentro del instituto me gustaría nombrar a personas como Arpit o Román, con las que hacía una hermosa convivencia y siempre voy a recordar.

Cómo no nombrar a las personas que están conmigo y que sin duda no sería nada sin ellas. A mis padres: Alicia y Virgilio, por su apoyo y cariño constante. Habéis sido mi pilar fundamental para que mis estudios siguiesen adelante. A mis amigos de Rivas y de la universidad, por haberme acompañado en este gran camino y por ser los responsables de los tantos buenos momentos en mi vida. A todos los que no he podido nombrar y que, en su pequeña aportación, forman parte de mí.

Índice general

Capítulo 1

Introducción

Los sistemas criptográficos han sido utilizados a lo largo de la historia. Uno de los más antiguos es el cifrado César, usado en la época romana, pretendía proteger información militar ante el enemigo. Consistía en desplazar k posiciones todos los caracteres del mensaje para así, hacerlo ilegible. Eligiendo $k = 3$, se puede cifrar el siguiente mensaje:

"Atacaremos al amanecer" $\rightarrow$ "Dxdfduhprv do dpdqhfhu".

Nótese que cada letra corresponde única y exclusivamente a otra, es decir, existe una relación biyectiva entre las cadenas de caracteres. Actualmente este sistema es completamente inseguro pues escasamente protege la información y se puede romper con facilidad. Otro sistema muy conocido es la máquina Enigma, usada durante la Segunda Guerra Mundial y cuyo criptoanálisis facilitó leer cantidades sustanciales de comunicaciones de radio codificadas en Morse de las potencias del Eje.

Actualmente existen procedimientos como RSA, que utiliza clave pública - clave privada ó AES (mucho más eficiente), de clave simétrica. El uso de la criptografía se extiende más allá de métodos de cifrado-descifrado. Desde los años 70, el uso de la criptografía se ha popularizado con el fin de alcanzar distintos objetivos, como asegurar la confidencialidad, integridad, y autenticidad de los datos. Últimamente es más común encontrar sistemas criptográficos que utilicen curvas elípticas como ECDH ó ECDSA (certificados de seguridad SSL y TLS de Internet). Otro campo más novedoso son los retículos, utilizados en criptosistemas post-cuánticos [MR09].

La autenticidad de un mensaje garantiza que el mensaje enviado desde un remitente, verdaderamente pertenece al mismo y no a un posible intermediario malicioso. Para este propósito se utilizan las firmas digitales, que utilizan funciones "unidireccionales" o también llamadas hash.

1.1. Funciones hash

Una función hash $H : \{0,1\}^* \rightarrow \{0,1\}^n$ es una función capaz de transformar un mensaje de longitud arbitraria en una secuencia de bits de longitud fija. La primera función hash data de 1961, cuando Wesley Peterson desarrolló Cyclic Redundancy Check (CRC) [PB61], una

función cuyo principal objetivo era comprobar cómo de correctos eran los datos transmitidos en redes como Internet. Otras funciones hash destacables a lo largo de los años son MD2 (Rivest, 1982), SHA-0 (NSA, 1993), SHA-1 (NSA, 1995) ó SHA-2 (NSA, 2001).

El resultado de diferentes funciones hash puede ser muy distinto, incluso la longitud de su resultado no tiene por qué ser la misma. Se aplicarán las funciones hash MD5, SHA-256, SHA-512 a la frase "Hoy está nublado", obteniendo los siguientes resultados:

MD5: `DB320637493B54C5E983A20668FB1672`
SHA-256: `6AB28237205E2409DE688172F2D4A24DA77AC677AADD50B8FF9E61F805A0300D`
SHA-512: `1D4EAB8EF41B2A05855E18A530B8E925E6FDB4E67C0A99CEDF4BA2E5713AEF9D`
`741884C42EA858546463021B5810E4D40923BF7696158A5CFD044E93F919DA8D`

Nótese que este resultado se expresa en caracteres hexadecimales. Además, las funciones hash deben satisfacer las siguientes propiedades:

1. Eficiencia: Dado un mensaje M, calcular $H(M)$ es poco costoso computacionalmente.
2. Irreversibilidad: Dado Y, es difícil encontrar un M tal que $H(M) = Y$.
3. Efecto avalancha: Cualquier mínimo cambio en el mensaje M, origina un hash $H(M)$ totalmente distinto.
4. Longitud fija: Cualquier mensaje M tiene la misma longitud en la imagen $H(M)$.
5. Resistencia a colisiones: Es decir, que encontrar un par x, y con $x \neq y$ tal que $H(y) = H(x)$ deba ser computacionalmente costoso.
6. Determinismo: $H(M)$ siempre toma el mismo valor si M es invariante.

Como ya se introdujo, las funciones hash son utilizadas, por ejemplo para asegurar la integridad en los datos [RRT17]. Sin embargo, esta seguridad no siempre es infalible. Un ejemplo es la función hash MD5, cuyas expectativas de seguridad eran bastante altas, hasta que en 2004 Xiaoyun Wang y su equipo anunciaron la existencia de colisiones para este hash [WY05]. Por otro lado, para otros hash como SHA-256 no se prevé una brecha a colisiones a corto plazo.

Otros de los usos más visibles de las funciones hash es en las previamente mencionadas firmas digitales, estructuras como el árbol de Merkle o sistemas como Blockchain.

1.1.1. Firmas digitales

Uno de los usos de las funciones hash está en las firmas digitales. Una firma digital es una manera de demostrar la autenticidad e integridad de un mensaje m. Existen muchos tipos de sistemas de firma como son ElGamal [ElG85], que usa la complejidad del cálculo del logaritmo discreto ó ECDSA, que usa curvas elípticas [JMV01].

El papel de la función hash en un sistema de firma es el de mantener la integridad de

los datos. Permite que todos los datos sean codificados a la vez con un valor único comprimido e irrepetible.

Sistema de firma ElGamal

Como ejemplo, se detalla el algoritmo del sistema de firma ElGamal para un mensaje m [ElG85].

1. **Selección de parámetros públicos**:
   - *a*) Una función hash H resistente a colisiones.
   - *b*) Un número primo p grande que dificulte el cómputo del logaritmo discreto $mod\ p$.
   - *c*) Un generador g del grupo multiplicativo $\mathbb{Z}_p^*$.
2. **Generación de claves**.
   - *a*) Se selecciona una clave secreta x tal que $1 < x < p - 1$.
   - *b*) Se calcula $y = g^x\ (mod\ p)$.

   La clave pública es (p, g, y).
   La clave secreta es x.
3. **Generación de una firma**.
   - *a*) Se selecciona un número aleatorio k tal que tal que $1 < k < p-1$ y $gcd(k, p-1) = 1$.
   - *b*) Calcula $r = g^k\ (mod\ p)$.
   - *c*) Calcula $s = (H(m) - xr)k^{-1}\ (mod\ p - 1)$.
   - *d*) Si $s = 0$ se reinicia el proceso.

   El par (r, s) es la firma digital para el mensaje m.
4. **Verificación**. La verificación de la firma de un mensaje en este sistema se consigue de la siguiente forma:
   - *a*) Se tiene que cumplir que $0 < r < p$ y $0 < s < p - 1$.
   - *b*) Se verifica que $g^{H(m)} = y^r r^s\ (mod\ p)$.

La seguridad de este sistema y de muchos otros recae, en parte, sobre la resistencia a colisiones de la función hash. Si existiese una colisión para la función hash *i.e.* $\exists\ m' : H(m') = H(m) = Y$, un adversario podría enviar otro mensaje m' que verifique la firma para m, violando así la integridad y autenticidad del sistema de firma.

1.1.2. Árbol de Merkle

Un árbol de Merkle [Bec08] es una estructura de datos binaria que permite garantizar la integridad de gran cantidad de información. Los datos se dividen en bloques más pequeños y se les aplica una función hash H por pares, siguiendo una estructura de árbol hacia arriba [Figura 1.1.2].

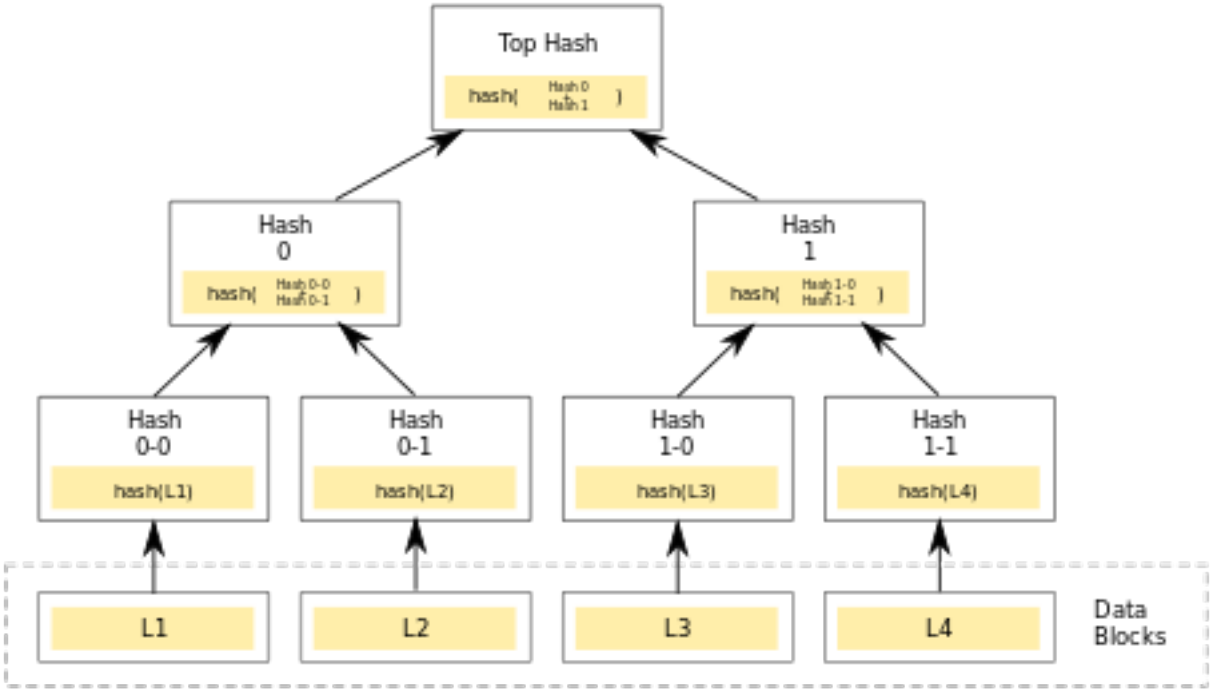

Figura 1.1: *Árbol de Merkle.*

Por ser H una función hash, realizar esta operación a cada par de bloque de datos permite una codificación de los datos única. Esencialmente, la seguridad de este sistema radica en la resistencia a colisiones de H. Cualquier mínima modificación en algún bloque de datos cambiaría radicalmente el hash que lo domina y consecuentemente, por su estructura, cada hash que esté por encima del hash dañado, también resultará alterado. Por tanto, el hash final, asociado con la raíz del árbol (*Top Hash*) estará dañado si y solo si se realiza alguna modificación en los datos originales.

Para una gran cantidad de datos, esta estructura permite localizar el bloque dañado sin necesidad de revisar todo el conjunto de datos hasta encontrar el error. El usuario, que conoce el resultado de los hashes en todos los niveles, los compara con los hashes recibidos y avanza por las ramas por las cuales el hash recibido no coincide con el hash original. Recursivamente, se llega hasta el último nivel del árbol y se escoge el bloque de datos dañado, que es el último hash divergente.

Los árboles de Merkle se usan en Blockchain [Gup17] por su seguridad y eficiencia. Un árbol de Merkle relaciona todas las transacciones de la Blockchain y las agrupa por pares hasta llegar a una dirección raíz. Si esta dirección no fuese la esperada, significaría un cambio en la estructura de los bloques que se podría encontrar fácilmente.

1.1.3. SHA-256

En 2001 el NIST publicó FIPS PUB 180-2, [Bar+02] un documento en el que se presentaba el paquete de funciones hash: SHA-2 (Secure Hashing Algorithm-2), entre las que se encontraba SHA-256. Esta destaca dentro de las hash por su gran seguridad y balance computacional de generación con un *output* fijo de 256 bits, que son 64 caracteres en hexadecimal.

SHA-256 tiene numerosas aplicaciones, como ser parte del proceso de generación de BITCOIN. Si esta función hash se rompiese, se podrían generar BITCOINS al ritmo que quisiera. Además, este hash se puede ver en muchos árboles de Merkle [Dhu+17] y en el estándar de mensaje firmado DKIM. Se hablará más ampliamente de esta función hash y su funcionamiento en capítulos posteriores.

1.2. Estructura del trabajo

Una vez introducidas las funciones hash, este Trabajo de Fin de Grado tratará los sistemas de pruebas criptográficas utilizadas para computación verificable. En el Capítulo 3 se analizará en detalle la función hash SHA-256, previamente introducida en la sección 1.1.3. En el Capítulo 4 se aplicará a esta función hash, un sistema de computación verificable: R1CS. El objetivo de tal capítulo será construir sistemas R1CS óptimos para componentes dentro de SHA-256. Los resultados se arrojan en el capítulo posterior, el 5, donde se reúnen estas componentes y se obtiene un sistema común que permite verificar los resultados de SHA-256 para un *input* arbitrario. En el Capítulo 6 se detallan las conclusiones, y posteriormente se introduce un anexo con la implementación del algoritmo.

Capítulo 2

Computación verificable

En ocasiones se puede delegar la computación de un cálculo a una entidad externa, la cual puede no ser confiable. Verificar que este cálculo sea correcto puede ser complejo. Una primera idea para verificar la corrección del cálculo puede ser la repetición del procedimiento, algo que no resulta nada práctico. Incluso, puede no ser posible ya que a la entidad le puede interesar no revelar parte del *input*, como es el caso en las pruebas de conocimiento cero [GMR89]. Para ello, se construyen los sistema de computación verificable, que permiten demostrar la corrección de una computación de forma eficiente.

Para construir un sistema de computación verificable es necesario caracterizar una computación formalmente. Podemos definir lenguaje como el conjunto de *inputs* $x \in \mathcal{X}$ y *outputs* $y \in \mathcal{Y}$ que son correctos para una función determinista. Es decir:

$$\mathcal{L} = \{(x, y) \in (\mathcal{X}, \mathcal{Y}) : f(x) = y\}$$

De este modo solo los pares $input - output$ correctos forman parte del lenguaje $\mathcal{L}$. Si podemos decidir la pertenencia a $\mathcal{L}$ de (x, y) verificamos la computación. Estamos, por tanto, ante un problema de decisión, que podemos representar mediante la clase NP [AB09].

Definición 1 (NP)**.** *Un lenguaje* $L \subset \{0,1\}^*$ *está en NP si* $\exists$ *máquina de Turing* M *determinista polinomial tal que* $\forall\ x \in \{0,1\}^*$

$$x \in \mathcal{L} \Leftrightarrow \exists w : |w| = poly(|x|) \wedge M(x, w) = 1$$

Es decir, NP engloba los problemas de decisión para los que existe un certificado w que permite resolverlo en tiempo polinomial.

2.1. Sistemas de pruebas criptográficas

Los sistemas de pruebas criptográficas son protocolos que pretenden verificar un enunciado, es decir, probar un problema de decisión en NP. En las pruebas criptográficas actúan dos entidades: probador $\mathcal{P}$ y verificador $\mathcal{V}$, que deben seguir el lenguaje $\mathcal{L}$ que marca el enunciado.

Denotemos por $out(\mathcal{V}, x, r, \mathcal{P}) \in \{0,1\}$ una función que decide bajo un *input* x si el verificador $\mathcal{V}$ está satisfecho, sobre una probabilidad r con la prueba que proporciona $\mathcal{P}$. Para un valor fijo de r, $out(\mathcal{V}, x, r, \mathcal{P})$ es una función determinista. Las pruebas criptográficas deben tener las propiedades de completitud y robustez [Tha21]:

Definición 2 (Completitud)**.** *Todo enunciado x verdadero debe tener una prueba de su validez. $\forall x \in \mathcal{L}$:*

$$Pr_r[out(\mathcal{V}, x, r, \mathcal{P}) = 1] = 1$$

Definición 3 (Robustez)**.** *Todo enunciado x falso no puede tener una prueba de validez. $\forall x \notin \mathcal{L}$:*

$$Pr_r[out(\mathcal{V}, x, r, \mathcal{P}) = 1] \leq \delta_s$$

El sistema de prueba se considera válido si $\delta_s \leq \frac{1}{3}$.

Existen muchos tipos de pruebas criptográficas, algunas de ellas necesitan una interacción entre las entidades, como los protocolos Sigma [Kra03] o el algoritmo de identificación de Schnorr [Sch91]. Sin embargo, otras no necesitan esta interacción para ser válidas. Algunas de las más conocidas son los SNARGs (**S**uccint **N**on-interactive **ARG**ument**s**), que cuando se realizan con conocimiento cero se llaman zk-SNARKs (**z**ero-**k**nowledge **S**uccint **N**on-interactive **AR**gument of **K**nowledge). La ventaja de estos protocolos es la eficiencia y la brevedad de la prueba (*succinctness*). Para construirlas se necesita expresar una función como un circuito aritmético.

Definición 4 (Circuitos aritméticos)**.** *Dada una serie de variables $X_1, \ldots, X_n$, un circuito aritmético $\mathcal{C}$ sobre un cuerpo $\mathbb{F}$ es un grafo dirigido sin ciclos cuyos nodos son puertas (gates) que pueden representar entrada (input) asociadas a una variable X_i, salida (output) asociadas con $c \in \mathbb{F}$ o intermedias, que indican una relación entre ciertas variables (suma o producto modular) [Gen+13] [Figura 2.1].*

Estas puertas dan lugar al *witness* ó w, que representa la totalidad de variables con las que se opera internamente. Se puede entender que dado un enunciado con unas variables X, si conoces los valores de las relaciones entre ellas en el circuito $\mathcal{C}$ que dan lugar al *output out i.e.* $\mathcal{C}(X, w) = out$ debes de conocer el *output*.

Definición 5 (Producto escalar usual)**.** *Dados dos vectores a,$b \in \mathbb{F}^n$,se define su producto escalar, denotado por $\cdot$ como:*

$$\vec{a} = \begin{pmatrix} a_1 & \cdots & a_n \end{pmatrix} \qquad \vec{b} = \begin{pmatrix} b_1 & \cdots & b_n \end{pmatrix}$$

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i \cdot b_i$$

Definición 6 (Producto de Hadamard)**.** *El producto de Hadamard es una operación matemática entre vectores o matrices de igual dimensión, representada con el símbolo $\circ$. Dadas las matrices $A, B \in \mathbb{F}^{m \times n}$:*

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mn} \end{pmatrix} \qquad A \circ B = \begin{pmatrix} a_{11}b_{11} & \cdots & a_{1n}b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & \cdots & a_{mn}b_{mn} \end{pmatrix}$$

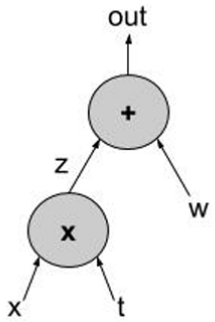

Figura 2.1: *Representación de un circuito aritmético que representa la relación out=x·t+w [Igl18], equivalentemente: out=z+w donde z es una variable intermedia.*

2.2. R1CS

Un método necesario en la construcción de SNARGs y zk-SNARKs para la verificación de un enunciado es construir su sistema R1CS (*Rank-1 Constraint System*) asociado. Este sistema se introdujo recientemente, a principios de la década de 2010 [Ben+13]. R1CS es a grandes rasgos, una equivalencia matricial a partir de un circuito aritmético relacionado con nuestro problema.

Definición 7 (R1CS)**.** *Dada una terna de matrices A,B,C* $\in \mathbb{F}^{m\times n}$*, el protocolo R1CS es un lenguaje que codifica el problema de decisión:* $\exists\ w \in \mathbb{F}^m\ :\ (A\cdot w)\circ(B\cdot w) = C\cdot w$.

Si se tiene un circuito aritmético sobre una función, es posible obtener un sistema R1CS de la misma. Se debe convertir cada relación aritmética de manera que se tenga como máximo una puerta multiplicativa. Se tratará de representar la relación en el siguiente formato:

$$A\cdot B - C = 0$$

De esta forma, si se consiguen todas las relaciones aritméticas hasta llegar al *output* del circuito $\mathcal{C}$, se puede expresar la siguiente relación matricial:

$$(A\cdot w)\circ(B\cdot w) - (C\cdot w) = 0$$

Ejemplo 1. *Se quiere verificar la correcta computación del circuito* $x^3 + x + 5$ *(mod 11) [Buc].*

Para este ejemplo, el circuito es muy corto, generalmente esto no suele ser tan sencillo. El objetivo es conseguir una demostración de dicho resultado, satisfaciendo una serie de relaciones algebraicas que lleguen hasta el *output* del circuito, indicado en la variables *out*.

Lo primero que se debe hacer es representar las relaciones de manera que involucren como mucho una puerta multiplicativa. Es decir, construir las siguientes relaciones:

$x_2 = x_1 \cdot x_1$
$x_3 = x_2 \cdot x_1$

$out = x_3 + x_1 + 5$

Es decir, para representar el enunciado de la manera más sencilla posible, han sido necesarias tres variables extra: x_2, x_3, out. Estas tres ecuaciones algebraicas son equivalentes al circuito del enunciado. Consideramos ahora el vector *witness* w con todas nuestras variables.

$$w = \begin{pmatrix} 1 & x_1 & x_2 & x_3 & out \end{pmatrix}^t$$

Sujeto a las ecuaciones:

$x_1 \cdot x_1 - x_2 = 0$
$x_2 \cdot x_1 - x_3 = 0$
$(x_3 + x_1 + 5) \cdot 1 - out = 0$

El siguiente paso es expresar estas relaciones en formato matricial. Al tener todas las ecuaciones el formato $A \cdot B - C = 0$ la equivalencia matricial es inmediata:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 5 & 1 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ out \end{pmatrix} \circ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ out \end{pmatrix} - \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ out \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

En concreto si consideramos el *witness* $w = \begin{pmatrix} 1 & 3 & 9 & 5 & 2 \end{pmatrix}^t$, se cumplen las 3 restricciones (o *constraints*):

$3 \cdot 3 - 9 = 0 \ (mod\ 11)$
$9 \cdot 3 - 5 = 0 \ (mod\ 11)$
$(5 + 3 + 5) \cdot 1 - 2 = 0 \ (mod\ 11)$

Capítulo 3

SHA-256

Como ya se introdujo anteriormente en el Capítulo 1, SHA-256 es una función hash que convierte cadenas de texto de longitud arbitraria a un cifrado de longitud fija de 256 bits cuyo resultado aparece representado en 64 caracteres hexadecimales. En el presente capítulo se detallará el funcionamiento de este hash.

En las secciones 3.1 y 3.2 se describirán todas las operaciones presentes en la función y en las secciones 3.3 y 3.4 se describirá el algoritmo en conjunto. Es posible visualizar esta función desde el documento oficial del algoritmo [MAI] o de forma interactiva desde internet [Dom22].

3.1. Operadores booleanos

SHA-256 está compuesto ampliamente por operaciones booleanas, que se pueden entender como operaciones binarias en F_2.

3.1.1. AND

Dados dos bits devuelve 1 si y solo si ambos bits son 1. Este operador booleano es representado con el símbolo $\wedge$ y se interpreta como el producto en $\mathbb{F}_2$.

A	B	Salida
0	0	0
0	1	0
1	0	0
1	1	1

Cuadro 3.1: *Puerta lógica AND.*

3.1.2. XOR

Representado por $\oplus$, este operador booleano devuelve 1 si y solo si sus dos entradas son distintas (equivalentemente, si solo una de las entradas es 1). En $\mathbb{F}_2$ se puede entender

como la suma.

A	B	Salida
0	0	0
0	1	1
1	0	1
1	1	0

Cuadro 3.2: *Puerta lógica XOR.*

3.2. Funciones presentes

3.2.1. RotR

Dado una cadena de bits X y un entero m, $RotR(X, m)$ rota de forma circular el conjunto de bits. Todos los bits se desplazan m posiciones "hacia la derecha" y si no es posible, pasan hacia el otro extremo de la cadena.

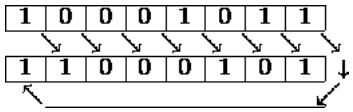

Figura 3.1: *Representación de RotR(X,1).*

3.2.2. ShR

Dado una cadena de bits X y un entero m, $ShR(X, m)$ aplica la función $RotR(X, m)$ y convierte los m-bits que han pasado hacia el otro extremo de la cadena en 0.

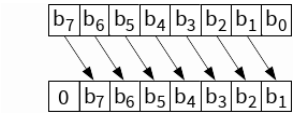

Figura 3.2: *Representación de ShR(X,1).*

Observación 1. *Si m es igual a la longitud de X, entonces ShR(X,m) es igual a la cadena nula.*

3.2.3. Funciones Σ, σ

Dentro de SHA-256 existen operaciones más complejas que utilizan las previamente mencionadas funciones $RotR, ShR$. Las siguientes funciones son concatenaciones de operaciones XOR en las que dada una cadena de 32 bits, devuelve otra cadena de 32 bits de misma longitud. Por cómo están construidas, conseguir una preimagen es una tarea compleja y cuyo resultado puede no ser único.

Función Σ_0

$$\Sigma_0(X) = RotR(X, 2) \oplus RotR(X, 13) \oplus RotR(X, 22)$$

Función Σ_1

$$\Sigma_1(X) = RotR(X, 6) \oplus RotR(X, 11) \oplus RotR(X, 25)$$

Tanto Σ_0 como Σ_1 son concatenaciones XOR de operaciones $RotR$. Las únicas diferencias entre ellas son las posiciones rotadas.

Función σ_0

$$\sigma_0(X) = RotR(X, 7) \oplus RotR(X, 18) \oplus ShR(X, 3)$$

Función σ_1

$$\sigma_1(X) = RotR(X, 17) \oplus RotR(X, 19) \oplus ShR(X, 10)$$

Es posible ver que tanto σ_0 como σ_1 utilizan ambas operaciones $RotR$ y ShR. La únicas diferencias entre sendas funciones son las posiciones de los bits.

3.2.4. Majority

Una de las funciones más recurrentes es Majority, que dados 3 bits x, y, z, devuelve la mayoría de los bits presentes. Se define de la siguiente forma:

$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

x	y	z	Salida
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Cuadro 3.3: *Operador Majority.*

3.2.5. Choice

Otra función que también aparece en SHA-256 es Choice, que dados 3 bits x, y, z, "elige" y ó z dependiendo del valor de x. Si x vale 0, devuelve el valor de z y si x vale 1, devuelve el valor de y. Se define de la siguiente forma:

$$Ch(x, y, z) = (x \wedge y) \oplus (\bar{x} \wedge z)$$

x	y	z	Salida
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Cuadro 3.4: *Operador Choice.*

3.2.6. Suma módulo 2^{32}

La suma de dos números a, b en binario se hace bit a bit. En este caso tenemos 32 bits que se irán sumando de derecha a izquierda. En el primer término, solo se necesita sumar a_1 y b_1 y se hace con una simple operación XOR. Sin embargo a partir del segundo bit se tienen que considerar necesariamente llevadas ó *carry bits* c_i, los cuales serían en total 31.

A diferencia de la suma usual de dos números en binario, la suma módulo 2^{32} mantiene los 32 últimos bits del resultado.

$$\begin{array}{rl} & c_{31} \cdots c_1 \\ & a_{32} \cdots a_2 \; a_1 \\ + & b_{32} \cdots b_2 \;\; b_1 \\ \hline & z_{32} \cdots z_2 \;\; z_1 \end{array}$$

A continuación, se presenta un ejemplo de una operación suma módulo 2^n.

Ejemplo 2. *Representar la suma 29 + 11 (mod 2^5) en binario.*

$$\begin{array}{rl} & 1\ 1\ 1\ 1 \\ & 1\ 1\ 1\ 0\ 1 \\ + & 0\ 1\ 0\ 1\ 1 \\ \hline & 0\ 1\ 0\ 0\ 0 \end{array}$$

Que es igual a 8, lo cual es correcto.

3.2.7. Temp1

Temp1 es un conjunto de operaciones dentro de SHA-256 que pretende producir un *output* de 32 bits, *temp*1, con el cual operar más tarde. Para producirlo, necesita varias variables de entrada como las variables de estado e, f, g, h; las variables W, que serán definidas más tarde y parámetros fijos en el funcionamiento de SHA-256 como son las constantes k.

Dentro de $Temp1$ se van a realizar: una operación intermedia Choice entre e, f, g; una operación Sigma, $\Sigma_1(e)$ y 4 sumas módulo 2^{32}. La fórmula para este conjunto de operaciones (o compuerta):

$$temp1 = h + \Sigma_1(e) + Choice(e, f, g) + k + W$$

El siguiente diagrama ilustra el mecanismo de Temp1:

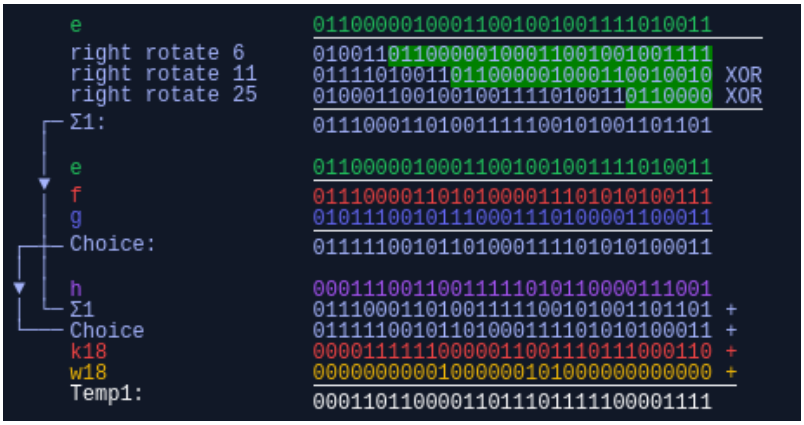

Figura 3.3: *Diagrama de funcionamiento de Temp1, desde https://sha256algorithm.com.*

3.2.8. Temp2

Al igual que Temp1, Temp2 es un conjunto de operaciones que pretenden producir un *output* de 32 bits, *temp*2, con el cual operar más tarde. Para producirlo, necesita 3 variables de estado a, b, c como *input*, cada una de 32 bits de longitud.

Dentro del funcionamiento de $Temp2$, se realizan 3 operaciones: Un Majority entre a, b, c, una operación Sigma, $\Sigma_0(a)$ y finalmente, una suma módulo 2^{32} entre los *output* de $Majority(a, b, c)$ y $\Sigma_0(a)$. La fórmula para esta compuerta:

$$temp2 = \Sigma_0(a) + Majority(a, b, c)$$

Y su diagrama:

Figura 3.4: *Diagrama de funcionamiento de Temp2, desde https://sha256algorithm.com.*

3.3. Algoritmo

Paso 1: Input. SHA-256 depende de un *input*, que es una cadena de caracteres, como ya se vio en el Capítulo 1. Este *input* automáticamente debe ser transformado a lenguaje binario usando UTF-8 y se le añade un 1 al final de la cadena.

Paso 2: Padding. A la cadena binaria se realiza un *padding*. Este paso es necesario para conseguir que el mensaje tenga una longitud múltiplo de 512 bits. El procedimiento añade a la cadena de bits (cuya longitud es $l + 1$), la cantidad k de ceros necesarios tales que $l + 1 + k \equiv 448 \ mod(512)$. Finalmente, se añade al mensaje su longitud l, representado en lenguaje binario de 64 bits. En el caso en el que al terminar el *padding* se obtenga una longitud mayor que 512 bits (1024, 1536, 2048, ...) se dividirán los bits en bloques de 512 bits y se irá haciendo una iteración del algoritmo por bloques.

Paso 3: Elección de las primeras variables W. En la iteración del algoritmo, se escoge el siguiente bloque de 512 bits. Esta cadena binaria, se divide ahora en 16 partes de 32 bits cada una y las partes se asignan a unas variables que llamaremos $W_1, \ldots, W_{16}$.

Paso 4: Completitud de las variables W. A partir de estas variables generadas del mensaje, se generan otras nuevas 48 variables binarias, $W_{17}, \ldots, W_{64}$, que se definen dependiendo del valor que toman sus antecesores. Estas nuevas variables se rigen por la siguiente fórmula de generación:

$$W_i = W_{i-16} + W_{i-7} + \sigma_0(W_{i-15}) + \sigma_1(W_{i-2}) \quad \forall i : 17 \leq i \leq 64$$

Paso 5: Iteración con $a, \ldots, h$. Lo siguiente es calcular las variables $a, \ldots, h$. Al menos para el primer paso y en el primer bloque, se utilizan parámetros conocidos de SHA-256 como $H_1, \ldots, H_8$ [1] y $k_1, \ldots, k_{64}$ [2] . Mediante estos y otras compuertas intermedias como $Temp1, Temp2$ se actualizarán las variables $a, \ldots, h$.

Paso 6: Cálculo de los nuevos parámetros $H_1, \ldots, H_8$. Una vez se haya repetido este proceso 64 veces, tendremos las definitivas variables $a, \ldots, h$. A partir de estas, generaremos unas nuevas variables $H_1, \ldots, H_8$ mediante una suma con las $H_1, \ldots, H_8$ anteriores.

Paso 7: Obtención del resultado. Si el mensaje original excedía la longitud de 512 bits, se realizará este mismo proceso con el siguiente bloque de 512 bits del mensaje y estas últimas variables $H_1, \ldots, H_8$ servirán como los nuevos parámetros $H_1, \ldots, H_8$ de SHA-256 en la siguiente iteración. De lo contrario, si se ha operado ya con el último bloque, el hash final es la concatenación: $H_1 || \cdots || H_8$.

[1] Estos parámetros son arbitrarios. Corresponden a la representación binaria de los primeros 32 bits de la parte fraccional de la raíces cuadradas de los 8 primeros números primos.

[2] Estos parámetros también son arbitrarios. Corresponden a la representación binaria de los primeros 32 bits de la parte fraccional de la raíces cúbicas de los 64 primeros números primos.

3.4. Pseudocódigo

Introducir mensaje M *input* que se descompone en bloques de 512 bits $M_1||\cdots||M_N$

for t = 1 hasta N **do**

$M_t = W_1||\cdots||W_{16}$

for i = 17 hasta 64 **do**
$W_i = W_{i-16} + W_{i-7} + \sigma_0(W_{i-15}) + \sigma_1(W_{i-2})$
end for

(a, b, c, d, e, f, g, h)=$(H_1^{(t-1)}, H_2^{(t-1)}, H_3^{(t-1)}, H_4^{(t-1)}, H_5^{(t-1)}, H_6^{(t-1)}, H_7^{(t-1)}, H_8^{(t-1)})$

for j = 1 hasta 64 **do**

$temp1 = h + \Sigma_1(e) + Choice(e, f, g) + k_j + W_j$
$temp2 = \Sigma_0(a) + Majority(a, b, c)$

$a = temp1 + temp2$
$b = a$
$c = b$
$d = c$
$e = d + temp1$
$f = e$
$g = f$
$h = g$

end for

$H_1^{(t)} = H_1^{(t-1)} + a$
$H_2^{(t)} = H_2^{(t-1)} + b$
$H_3^{(t)} = H_3^{(t-1)} + c$
$H_4^{(t)} = H_4^{(t-1)} + d$
$H_5^{(t)} = H_5^{(t-1)} + e$
$H_6^{(t)} = H_6^{(t-1)} + f$
$H_7^{(t)} = H_7^{(t-1)} + g$
$H_8^{(t)} = H_8^{(t-1)} + h$

end for

El hash final es: $H_1^{(N)}||H_2^{(N)}||H_3^{(N)}||H_4^{(N)}||H_5^{(N)}||H_6^{(N)}||H_7^{(N)}||H_8^{(N)}$

Capítulo 4

Implementación R1CS a SHA-256

Esta sección se centrará en elaborar un sistema R1CS a la función hash SHA-256 en un cuerpo $\mathbb{F}_p$. Concretamente se realizará este sistema sobre el cuerpo binario $\mathbb{F}_2$, la cual es la labor de este Trabajo de Fin de Grado y principal contribución científica. Se elige este cuerpo por la gran cantidad de operaciones binarias presentes en la función, vistas en el capítulo anterior.

El objetivo es obtener este sistema de manera óptima en número de variables y restricciones: cuantas menos, mejor. Se irán desarrollando R1CS lo más óptimos posibles a funciones presentes de manera independiente; se presentará un lema de composición que permita juntar bloques R1CS optimizados para distintas funciones y se aplicará el lema para obtener el R1CS completo.

4.1. Majority

Recordamos cómo se definía la función Majority, para tres input x, y, z, devolvía la mayoría de los bits presentes. Su fórmula era:

$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

Si se quisiese un sistema R1CS de esta función, podemos pensar en una primera instancia en añadir variables intermedias como $a = (x \wedge z)$ y $b = (y \wedge z)$ con lo que se llega al formato $A \cdot B - C = 0$:

$$(x \wedge y) \oplus a \oplus b \oplus Maj = 0$$

Donde Maj representa la variable *output*.

Podemos pensar que añadiendo dos variables, las dimensiones son bajas, pero lo cierto es que la función Majority es bastante recurrente dentro de SHA-256 y de esta manera siempre genera dos variables nuevas para su computación en cada iteración, por lo que optimizarla puede ser primordial.

Analizando en detalle esta operación, podemos concluir una forma de reescribirla en formato R1CS sin que dependa de nuevas variables , es decir, tan solo utilizando las variables

$\{1, x, y, z, Maj\}$:

$$(x \oplus y) \wedge (z \oplus y) \oplus Maj \oplus y = 0$$

De esta forma se obtiene el formato deseado sin necesidad de variables intermedias y solo una ecuación. Si se desarrollase, se llegaría a la expresión original usando que $y^2 = y$ en binario. Consecuentemente, las dimensiones de las matrices de R1CS para esta operación se reducen a $\mathbb{F}_2^{1\times 5}$, con un vector *witness*:

$$w = \begin{pmatrix} 1 & x & y & z & Maj \end{pmatrix}^t$$

Verificando el siguiente sistema:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ x \\ y \\ z \\ Maj \end{pmatrix} \circ \begin{pmatrix} 0 & 0 & 1 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ x \\ y \\ z \\ Maj \end{pmatrix} - \begin{pmatrix} 0 & 0 & 1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ x \\ y \\ z \\ Maj \end{pmatrix} = 0$$

Nótese que al estar en el cuerpo binario, la resta es igual a la suma.

4.2. Choice

Recordamos la definición de esta otra función que dados 3 bits de *input* x, y, z "escogía" y ó z dependiendo del valor de x. Su fórmula era:

$$Ch(x, y, z) = (x \wedge y) \oplus (\bar{x} \wedge z)$$

De la misma manera, inicialmente podríamos pensar en añadir variables intermedias como $a = (1 + x)$ y $b = (a \wedge z)$ que faciliten la comprensión de la fórmula:

$$(x \wedge y) \oplus b \oplus Ch = 0$$

Donde Ch representa la variable *output*.

Esta forma también generaría 2 variables nuevas para la computación de esta operación. Al igual que Majority, Choice es muy recurrente en SHA-256 por lo que una correcta optimización en el número de variables y restricciones es crucial. La manera de optimizar esta operación se detalla a continuación:

$$x \wedge (y \oplus z) \oplus z \oplus Ch = 0$$

De esta forma se obtiene la forma deseada sin necesidad de variables intermedias y solo una ecuación, por lo que finalmente las dimensiones de las matrices del sistema R1CS resultan $\mathbb{F}_2^{1\times 5}$, con un vector *witness*:

$$w = \begin{pmatrix} 1 & x & y & z & Ch \end{pmatrix}^t$$

Verificando el siguiente sistema:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ x \\ y \\ z \\ Ch \end{pmatrix} \circ \begin{pmatrix} 0 & 0 & 1 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ x \\ y \\ z \\ Ch \end{pmatrix} - \begin{pmatrix} 0 & 0 & 0 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ x \\ y \\ z \\ Ch \end{pmatrix} = 0$$

4.3. Suma módulo 2^{32}

Una primera aproximación para el desarrollo de un sistema R1CS a esta operación es tratar de expresar cada sumando secuencialmente, que es posible e involucra las variables de llevada *carry bits*:

$$z_1 = a_1 \oplus b_1$$
$$c_1 = a_1 \wedge b_1$$

$$z_2 = a_2 \oplus b_2 \oplus c_1$$
$$c_2 = (a_2 \wedge b_2) \oplus (a_2 \wedge c_1) \oplus (b_2 \wedge c_1)$$
$$\dots$$
$$z_{32} = a_{32} \oplus b_{32} \oplus c_{31}$$

Lo que supondría un total de 128 variables $\{1, a_1, \dots, a_{32}, b_1, \dots, b_{32}, c_1, \dots, c_{31}, z_1, \dots, z_{32}\}$ y 64 ecuaciones, lo que hace unas dimensiones de las matrices del sistema R1CS de $\mathbb{F}_2^{64\times 128}$.

Para obtener un sistema R1CS óptimo de esta operación será necesaria una implementación sin variables *carry* y una homogeneización de las ecuaciones. Veamos como construir este sistema sin variables extra.

Observación 2. *Determinar el valor de los carry bits en una suma binaria es una operación Majority.*

$$c_i = Maj(a_i, b_i, c_{i-1})$$

Proposición 1 (Carry-bits)**.** *Es posible definir la suma módulo 2^{32} para dos números a,b de 32 bits en formato R1CS sin necesidad de variables carry bits.*

Demostración. Se define cada variable Z_i, c_i de la siguiente forma:

$Z_1 = a_1 \oplus b_1$, que ya está en formato R1CS: $(a_1 \oplus b_1) \wedge 1 \oplus Z_1 = 0$

$c_1 = a_1 \wedge b_1$
$Z_2 = c_1 \oplus a_2 \oplus b_2 = (a_1 \wedge b_1) \oplus a_2 \oplus b_2$, equivalentemente en R1CS: $(a_1 \wedge b_1) \oplus a_2 \oplus b_2 \oplus Z_2 = 0$

$c_2 = Maj(a_2, b_2, c_1) = (a_2 \wedge b_2) \oplus (a_2 \wedge c_1) \oplus (b_2 \wedge c_1)$
$Z_3 = c_2 \oplus a_3 \oplus b_3 = (a_2 \wedge b_2) \oplus (a_2 \wedge c_1) \oplus (b_2 \wedge c_1) \oplus a_3 \oplus b_3$

En general, $\forall i \geq 3$

$c_{i-1} = Maj(a_{i-1}, b_{i-1}, c_{i-2}) = (a_{i-1} \wedge b_{i-1}) \oplus (a_{i-1} \wedge c_{i-2}) \oplus (b_{i-1} \wedge c_{i-2})$
$Z_i = c_{i-1} \oplus a_i \oplus b_i = (a_{i-1} \wedge b_{i-1}) \oplus (a_{i-1} \wedge c_{i-2}) \oplus (b_{i-1} \wedge c_{i-2}) \oplus a_i \oplus b_i$ (No tiene formato R1CS)

Para conseguir un formato R1CS en Z_i, sustituimos c_{i-2} considerando Z_{i-1}

$Z_{i-1} = c_{i-2} \oplus a_{i-1} \oplus b_{i-1}$

Al estar en $\mathbb{F}_2$

$c_{i-2} = Z_{i-1} \oplus a_{i-1} \oplus b_{i-1}$

Por tanto:

$$Z_i = (a_{i-1} \wedge b_{i-1}) \oplus (a_{i-1} \wedge (Z_{i-1} \oplus a_{i-1} \oplus b_{i-1})) \oplus (b_{i-1} \wedge (Z_{i-1} \oplus a_{i-1} \oplus b_{i-1})) \oplus a_i \oplus b_i$$
$$Z_i = (a_{i-1} \wedge b_{i-1}) \oplus (a_{i-1} \wedge Z_{i-1}) \oplus a_{i-1} \oplus (a_{i-1} \wedge b_{i-1}) \oplus (b_{i-1} \wedge Z_{i-1}) \oplus (a_{i-1} \wedge b_{i-1}) \oplus b_{i-1} \oplus a_i \oplus b_i$$
$$Z_i = (a_{i-1} \wedge Z_{i-1}) \oplus a_{i-1} \oplus (b_{i-1} \wedge Z_{i-1}) \oplus (a_{i-1} \wedge b_{i-1}) \oplus b_{i-1} \oplus a_i \oplus b_i$$

Que es equivalente a:

$Z_i = a_i \oplus b_i \oplus a_{i-1} \oplus (Z_{i-1} \oplus b_{i-1}) \wedge (a_{i-1} \oplus b_{i-1})$ (formato R1CS). □

La función consta de 97 variables $\{1, a_1 \dots a_{32}, b_1 \dots b_{32}, Z_1 \dots Z_{32}\}$ y 32 restricciones:

$$a_1 \oplus b_1 \oplus Z_1 = 0$$
$$(a_1 \wedge b_1) \oplus a_2 \oplus b_2 \oplus Z_2 = 0$$
$$a_i \oplus b_i \oplus a_{i-1} \oplus (Z_{i-1} \oplus b_{i-1}) \wedge (a_{i-1} \oplus b_{i-1}) \oplus Z_i = 0 \quad 3 \leq i \leq 32$$

Las dimensiones de las matrices A, B, C del sistema R1CS asociado a la suma módulo 2^{32} son $\mathbb{F}_2^{32 \times 97}$.

Observación 3. *La primera restricción es una concatenación de operaciones XOR por lo que los elementos no nulos en el sistema R1CS se encuentran únicamente en la matriz C.*

Observación 4. *Este ahorro de variables carry de la suma en binario solo es posible si están sumando dos números a,b. Si se quisiese hacer una suma de k números módulo* 2^{32} *se tendrían que hacer* $k-1$ *operaciones suma módulo* 2^{32}*, que conlleva un gasto de* $k-2$ *variables intermedias que almacenen los sumandos más su variable output final Z. En total,* $2 \cdot k - 1$ *variables, de 32 bits cada una, que hacen* $32 \cdot (2 \cdot k - 1)$ *variables totales involucradas.*

4.4. Lema de composición de funciones

Dado un circuito suficientemente complejo, construir su sistema R1CS puede ser difícil. Para ello nos apoyaremos en los siguientes resultados que permitirán definir sistemas R1CS para circuitos que contienen circuitos cuyos sistemas R1CS son ya conocidos.

Lema 1 (Composición de funciones)**.** *Sea* f *un circuito compuesto por* $g_1, \dots, g_m$ *sub-circuitos. Supóngase que cada sub-circuito* g_k *tiene un sistema R1CS asociado con un witness* w_k*. Es posible definir un sistema R1CS para* f *con* $w_f = (w_1 || \cdots || w_m)^t$ *y matrices A,B,C:*

$$A=\begin{pmatrix} \boxed{A_{g_1}} & & & & \\ & \ddots & & & \\ & & \boxed{A_{g_k}} & & \\ & & & \boxed{A_{g_{k+1}}} & \\ & & & & \ddots \\ & & & & & \boxed{A_{g_m}} \end{pmatrix} \quad B=\begin{pmatrix} \boxed{B_{g_1}} & & & & \\ & \ddots & & & \\ & & \boxed{B_{g_k}} & & \\ & & & \boxed{B_{g_{k+1}}} & \\ & & & & \ddots \\ & & & & & \boxed{B_{g_m}} \end{pmatrix}$$

$$C=\begin{pmatrix} \boxed{C_{g_1}} & & & & \\ & \ddots & & & \\ & & \boxed{C_{g_k}} & & \\ & & & \boxed{C_{g_{k+1}}} & \\ & & & & \ddots \\ & & & & & \boxed{C_{g_m}} \end{pmatrix}$$

Donde cada matriz contiene en su diagonal a las sub-matrices de los sistema R1CS asociados a cada sub-circuito $g_1, \ldots, g_m$.

Demostración. Si definimos las matrices como el lema indica:

$$[A \cdot w_f] \circ [B \cdot w_f]$$

Por ser producto escalar, cada w_k se relaciona con su propio sub-circuito:

$$= [(A_{g_1} \cdot w_1, \ldots, A_{g_m} \cdot w_m)^t] \circ [(B_{g_1} \cdot w_1, \ldots, B_{g_m} \cdot w_m)^t] =$$
$$= ((A_{g_1} \cdot w_1) \circ (B_{g_1} \cdot w_1), \ldots, (A_{g_m} \cdot w_m) \circ (B_{g_m} \cdot w_m))^t =$$

Cada sub-circuito g_k tiene que cumplir su sistema R1CS: $[A_{g_k} \cdot w_k] \circ [B_{g_k} \cdot w_k] = C_{g_k} \cdot w_k$:

$$= (C_{g_1} \cdot w_1, \ldots, C_{g_m} \cdot w_m)^t = C \cdot w_f$$

Por tanto, las matrices A, B, C satisfacen la relación R1CS para f. □

En el lema anterior, se ha construido un sistema R1CS para un circuito f formado por sub-circuitos $g_1, \ldots, g_m$, cuyos sistemas R1CS son conocidos. Sin embargo, esta construcción no es óptima. Puede suceder que algunos sub-circuitos $g_1, \ldots, g_m$ utilicen variables comunes. Es decir, que algunas entradas de distintos w_i correspondan a las mismas variables y que por tanto, dichos valores en el vector *witness* sean redundantes.

Lema 2 (Composición optimizada de funciones)**.** *Sea f un circuito compuesto por $g_1, \ldots, g_m$ sub-circuitos y sea $V_1, \ldots, V_m$ el conjunto de variables implicadas en cada uno de ellos. Es posible definir un sistema R1CS optimizado para f donde las variables implicadas en él, recogidas en su vector witness w_f, corresponden con $\cup_{i=1}^{m} V_i$. Finalmente, las matrices A,B,C se construyen de la forma:*

$$M = \begin{pmatrix} \boxed{M_{g_1}} \\ \vdots \\ \boxed{M_{g_k}} \\ \boxed{M_{g_{k+1}}} \\ \vdots \\ \boxed{M_{g_m}} \end{pmatrix}$$

Donde cada sub-matriz M_{g_k} puede tener valores no nulos a lo largo de todas sus filas.

El procedimiento para construir las matrices del sistema optimizado es sencillo. Cada sistema R1CS relacionado con un sub-circuito debe seguir satisfaciéndose. Por lo que cada sub-matriz M_{g_k} tendrá los mismos valores asociados a las variables implicadas en su sub-circuito. Al ser w_f un vector cuyos elementos corresponden con $\cup_{i=1}^{m} V_i$, estos elementos son repartidos por el vector *witness* sin un orden claro. Por tanto, los valores no nulos asociados con dichos elementos pueden estar a lo largo de todas las filas. El resto de variables, las que no son usadas por el sub-circuito g_k, se las asocia con el valor 0 en sendas sub-matrices M_{g_k}.

Como consecuencia, las matrices A, B, C resultan más estrechas (menos columnas) y siguen cumpliendo la relación R1CS. La demostración es equivalente al lema anterior.

4.5. Generación de las variables W

Un sistema R1CS también puede servir, como en este caso, para definir la generación de ciertas variables. Es decir, si se cumplen ciertas restricciones sobre ellas mismas significará que están bien construidas y será posible utilizarlas posteriormente. Para obtener W_i $\forall i : 17 \leq i \leq 64$ es necesario conocer sus antecesores y calcularlos bit a bit. Como ya se introdujo en el capítulo anterior, la generación de estas variables seguía la fórmula:

$$W_i = W_{i-16} + W_{i-7} + \sigma_0(W_{i-15}) + \sigma_1(W_{i-2}) \quad \forall i : 17 \leq i \leq 64$$

Nótese que estas son operaciones de suma módulo 2^{32} y que por tanto, al haber tres sumas serán necesarias dos variables intermedias que almacenen el resultado de las sumas. Esta secuencia se hace 48 veces, por lo que en total se han de generar 96 variables intermedias además de las 64 variables W y se realizarán 144 sumas módulo 2^{32}.

Ahora bien, σ_0 y σ_1 son funciones que utilizan solamente operaciones XOR y que por tanto, en términos de sistema R1CS no necesitan ninguna variable extra que recoja su resultado.

4.6. Temp1

Recordando, dentro de $Temp1$ se realizan 4 sumas módulo 2^{32}, una operación $Choice(e, f, g)$ y una operación sigma, $\Sigma_1(e)$. Como queremos un sistema óptimo en el número de variables y restricciones, observamos que las operaciones presentes en $\Sigma_1(e)$ son sumas XOR, por lo que pueden ser incluidas en la matriz C del sistema R1CS sin necesidad de una variable extra que los recoja ni ecuaciones extra que la defina. Por tanto las variables necesarias para $Temp1$, por el lema de composición de funciones, son e, f, g, h, k_i, W_i,Ch, 3 variables intermedias de suma y el *output* $temp1$, en principio 352 variables.

Para representar las ecuaciones presentes, por el lema de composición de funciones, tenemos que tener en cuenta todas las operaciones. Se consideran las 32 restricciones de

Choice y las 128 restricciones de las 4 sumas módulo 2^{32}. Las ecuaciones para esta compuerta, en formato R1CS:

1. Las ecuaciones de Choice:

$$e_i \wedge (f_i \oplus g_i) = Ch_i \oplus g_i \quad \forall i : 1 \leq i \leq 32$$

2. Las 32 ecuaciones de la primera suma:

$$Sum1_1 = h_1 \oplus [e_7 \oplus e_{12} \oplus e_{26}]$$

$$Sum1_2 = (h_1 \wedge [e_7 \oplus e_{12} \oplus e_{26}]) \oplus h_2 \oplus [e_8 \oplus e_{13} \oplus e_{27}]$$

$$\dots$$

$$Sum1_{32} = h_{32} \oplus [e_6 \oplus e_{11} \oplus e_{25}] \oplus h_{31} \oplus (Sum1_{31} \oplus [e_5 \oplus e_{10} \oplus e_{24}]) \wedge (h_{31} \oplus [e_5 \oplus e_{10} \oplus e_{24}])$$

3. Las 32 ecuaciones de la segunda suma:

$$Sum2_1 = Sum1_1 \oplus Ch_1$$

$$Sum2_2 = (Sum1_1 \wedge Ch_1) \oplus Sum1_2 \oplus Ch_2$$

$$\dots$$

$$Sum2_{32} = Sum1_{32} \oplus Ch_{32} \oplus Sum1_{31} \oplus (Sum2_{31} \oplus Ch_{31}) \wedge (Sum1_{31} \oplus Ch_{31})$$

4. Las 32 ecuaciones de la tercera suma:

$$Sum3_1 = Sum2_1 \oplus k_1$$

$$Sum3_2 = (Sum2_1 \wedge k_1) \oplus Sum2_2 \oplus k_2$$

$$\dots$$

$$Sum_{32} = Sum2_{32} \oplus k_{32} \oplus Sum2_{31} \oplus (Sum3_{31} \oplus k_{31}) \wedge (Sum2_{31} \oplus k_{31})$$

5. Las 32 ecuaciones de la cuarta suma (*output*):

$$temp1_1 = Sum3_1 \oplus W_1$$

$$temp1_2 = (Sum3_1 \wedge W_1) \oplus Sum3_2 \oplus W_2$$

$$\dots$$

$$temp1_{32} = Sum3_{32} \oplus W_{32} \oplus Sum3_{31} \oplus (temp1_{31} \oplus W_{31}) \wedge (Sum3_{31} \oplus W_{31})$$

En total 160 *constraints*.

4.7. Temp2

Esta compuerta necesitaba como *input* 3 variables de estado a, b, c, cada una de 32 bits de longitud y tenía como operaciones internas un Majority, un $\Sigma_0(a)$ (sin necesidad de variables intermedias por ser concatenación de XOR) y una suma módulo 2^{32} (32 variables *output*).

Por el lema de composición de funciones y suponiendo que son variables distintas, serán necesarias $5 \cdot 32 = 160$ variables. Las ecuaciones a seguir en esta compuerta, en forma R1CS, resultan de la forma:

Las 32 ecuaciones para la operación Majority:

$$(a_i \oplus b_i) \wedge (c_i \oplus b_i) \oplus b_i \oplus Maj_i = 0 \quad \forall i : 1 \leq i \leq 64$$

Para las ecuaciones de suma módulo 2^{32}:

$$[a_3 \oplus a_{14} \oplus a_{23}] \oplus Maj_1 \oplus temp2_1 = 0$$

$$(Maj_1 \wedge [a_3 \oplus a_{14} \oplus a_{23}]) \oplus [a_4 \oplus a_{15} \oplus a_{24}] \oplus Maj_2 \oplus temp2_2 = 0$$

$$\dots$$

$$(temp2_{31} \oplus [a_1 \oplus a_{12} \oplus a_{21}]) \wedge (Maj_{31} \oplus [a_1 \oplus a_{12} \oplus a_{21}]) \oplus [a_2 \oplus a_{13} \oplus a_{22}] \oplus Maj_{31} \oplus Maj_{32} \oplus temp2_{32} = 0$$

Lo que hace un total de 64 *constraints*.

4.8. Actualización de las variables de estado

Una vez conseguida la computación de $W_i, Temp1, Temp2$ vamos a intentar realizar una iteración de las variables $a, \dots, h$. Se recuerda la secuencia de actualización de variables:

$$a' = temp1 + temp2 \qquad e' = d + temp1$$

$$b' = a \qquad f' = e$$

$$c' = b \qquad g' = f$$

$$d' = c \qquad h' = g$$

Cabe destacar que esta actualización se realiza 64 veces por lo que el número de variables requeridas en total puede crecer muy rápidamente. Realmente no es así, uno se puede dar cuenta que si se define los valores $a_j, \dots, h_j$ como los valores de las variables de estado en la iteración j y se asigna los valores $a_0, \dots, h_0$ a los valores de las constantes de SHA-256 en el bloque t:

$$b_1 = a_0 \qquad f_1 = e_0$$

$$b_j = a_{j-1} \quad \forall j : 2 \leq j \leq 64 \qquad f_j = e_{j-1} \quad \forall j : 2 \leq j \leq 64$$

$$
\begin{aligned}
c_1 &= b_0 & g_1 &= f_0 \\
c_2 &= a_0 & g_2 &= e_0 \\
c_j &= a_{j-2} \quad \forall j : 3 \leq j \leq 64 & g_j &= e_{j-2} \quad \forall j : 3 \leq j \leq 64
\end{aligned}
$$

$$
\begin{aligned}
d_1 &= c_0 & h_1 &= g_0 \\
d_2 &= b_0 & h_2 &= f_0 \\
d_3 &= a_0 & h_3 &= e_0 \\
d_j &= a_{j-3} \quad \forall j : 4 \leq j \leq 64 & h_j &= e_{j-3} \quad \forall j : 4 \leq j \leq 64
\end{aligned}
$$

Se puede apreciar que solo es necesario actualizar las variables a, e almacenando su resultado en cada iteración. y que por tanto para el sistema R1CS de este bucle de 64 iteraciones solo harán falta la totalidad de las variables a, e y las constantes $a_0, \ldots, h_0$ del bloque t junto con 2 sumas en cada iteración.

4.9. Obtención de las nuevas constantes $H_1, \ldots, H_8$

El resultado del hash final es una cadena de 64 caracteres en hexadecimal. Estos caracteres vienen de 8 cadenas de 32 bits que se generan mediante 8 sumas módulo 2^{32}. Una vez de consiguen las últimas variables $a_{64}, b_{64}, c_{64}, d_{64}, e_{64}, f_{64}, g_{64}, h_{64}$, o lo que es lo mismo, $a_{64}, a_{63}, a_{62}, a_{61}, e_{64}, e_{63}, e_{62}, e_{61}$, se realizan las siguientes sumas:

$$
\begin{aligned}
H_1 &= a_0 + a_{64} & H_5 &= e_0 + e_{64} \\
H_2 &= b_0 + a_{63} & H_6 &= f_0 + e_{63} \\
H_3 &= c_0 + a_{62} & H_7 &= g_0 + e_{62} \\
H_4 &= d_0 + a_{61} & H_8 &= h_0 + e_{61}
\end{aligned}
$$

Por lo que para conseguir su sistema R1CS asociado son necesarias 8 sumas módulo 2^{32} (256 *constraints*) y 24 variables de 32 bits (768 variables).

4.10. Caso N > 1

Para un *input* suficientemente grande, se debe realizar el mismo proceso de SHA-256 por bloques N veces. Como se ha explicado en el capítulo anterior, la nueva iteración toma como nuevas constantes $H_1, \ldots, H_8$ los últimos valores de la la iteración anterior. La diferencia recae en el *witness*. Al cambiar las constantes $H_1, \ldots, H_8$ y el *input* $W_1, \ldots, W_{16}$ el *witness* cambia drásticamente. Sin embargo, las matrices del sistema representan las relaciones entre todas estas variables, por lo que permanecen invariantes.

En términos de R1CS, se debe realizar el mismo sistema de ecuaciones N veces. Se comprueba que cada iteración satisface el sistema matricial y que por tanto, el resultado final está bien computado.

Capítulo 5

Resultados

Una vez visto todo el funcionamiento de SHA-256, se hará una puesta en conjunto. Como se ha visto en el pseudocódigo del Capítulo 3, se pueden diferenciar tres grandes fases dentro de la función hash:

1. Generación de las variables W.
2. Actualización de las variables de estado.
3. Obtención de las nuevas constantes $H_1, \ldots, H_8$.

Aplicando el lema de composición, vamos a calcular el número de variables y *constraints* que serán necesarias finalmente para el sistema R1CS final.

Generación de las variables W

Para la generación de las variables W se necesitaban a parte de las mismas $W_1, \ldots, W_{64}$ también eran necesarias 96 variables que almacenaban los resultados de la suma módulo 2^{32}. Como todas ellas son variables de 32 bits, al final se necesitarán 5120 variables, que representan todos los bits involucrados en este proceso.

Para obtener el número de *constraints*, tenemos que tener en cuenta que las relaciones de suma módulo 2^{32} implican 32 *constraints* cada una. Tenemos 3 sumas módulo 2^{32} para cada uno de los 48 W_i que debemos generar. Esto hace un total de $48 \cdot 3 \cdot 32 = 4608$ *constraints*.

Actualización de las variables de estado

Para la actualización de las variables de estado, son necesarias las compuertas $Temp1, Temp2$ y 2 sumas módulo 2^{32}. En esta fase solo se actualizan las variables de estado $\{a, e\}$ ya que las demás variables $\{b, c, d, f, g, h\}$ dependían de los valores anteriores de las mismas, que son almacenados en el *witness*. Si queremos calcular cuántas variables son estrictamente necesarias, calculamos siguiendo el lema de composición.

Asumiendo que las variables a, e tienen valores distintos a lo largo del bucle, serán necesarias las 128 variables de estado a_i, e_i $1 \leq i \leq 64$ y las 8 constantes $H_1, \ldots, H_8$, correspondientes al bloque vigente y que operan en la primera iteración del bucle.

Para la compuerta $Temp1$, son necesarias las variables e_i, e_{i-1}, e_{i-2}, e_{i-3}, W_i, k_i, Ch_i, 3 variables que almacenen resultados de las sumas módulo 2^{32} y la variable *output* $temp1_i$. Como ya se contaba con todas las variables e_i y W_i, la compuerta $Temp1$ solo añade las variables Ch_i, k_i, las 3 de almacenamiento en la suma módulo 2^{32} y el *output* $temp1$, todas de 32 bits.

Para la compuerta $Temp2$, son necesarias las variables a_i, a_{i-1}, a_{i-2}, Maj_i y el *output* $temp2$. Como ya se habían mencionado las variables a_i, la compuerta $Temp2$ solo añade las variables Maj_i y el *output* $temp2$.

Además, son necesarias 2 sumas módulo 2^{32} que son las que actualizan a_i, e_i pero que no añaden ninguna variable extra, pues todas han sido mencionado previamente:

1. $a_{i+1} = temp1 + temp2$
2. $e_{i+1} = a_{i-3} + temp1$

Por tanto, el número de variables necesarias, todas de 32 bits, son $32 \cdot (128+8+64 \cdot 6+64 \cdot 2) = 20736$ variables.

Para el número de *constraints*, calculamos considerando que en principio, todas las ecuaciones diferentes por ser las variables distintas. La compuerta $Temp1$ necesitaba 160 *constraints* y la compuerta $Temp2$, por otra parte, necesitaba 64. Considerando ahora que se encuadran en un bucle de 64 iteraciones:

1. 64 compuertas $Temp1$ hacen 10240 *constraints*.
2. 64 compuertas $Temp2$ hacen 4096 *constraints*.
3. 64 sumas $temp1 + temp2$ para a_{i+1} hacen 2048 *constraints*.
4. 64 sumas $a_{i-3} + temp1$ para e_{i+1} hacen 2048 *constraints*.

En total 18432 *constraints*.

Obtención de las nuevas constantes $H_1, \ldots, H_8$

Una vez tenemos $a_{64}, a_{63}, a_{62}, a_{61}, e_{64}, e_{63}, e_{62}, e_{61}$, las variables se han de sumar módulo 2^{32} con los parámetros $a_0, b_0, c_0, d_0, e_0, f_0, g_0, h_0$. Es decir, se realizan 8 operaciones suma módulo 2^{32} ($8 \cdot 32 = 256$ nuevas *constraints*), que solo genera 8 nuevas variables *output* $H_1, \ldots, H_8$; de 32 bits cada una, es decir, 256 nuevas variables.

Aplicando el lema de composición, las dimensiones del sistema R1CS final serán:

Filas (*constraints*): 4608 + 18432 + 256 = **23296**.
Columnas (variables): 5120 + 20736 + 256 = **26112**.

Por tanto las dimensiones del sistema R1CS asociado para SHA-256 deberían ser de $\mathbb{F}_2^{23296\times 26113}$, añadiendo la unidad como variable.

Orden del sistema

El orden del *witness* y la especificación de las ecuaciones es muy importante. El orden elegido es tal que se puede inferir un patrón según las *constraints* previamente definidas. Los bits de las variables se ordenan de tal forma que el bit que naturalmente se coloca más "a la derecha" aparece el primero. Como hay muchas variables que se denominan igual, el subíndice indicará la diferencia entre las variables:

$$\begin{aligned} witness = (&1, W_1, W_2, \ldots, W_{16}, W_{17}, \ldots, W_{64}, t1_1, t1_2, \ldots, t1_{48}, t2_1, \ldots, t2_{48},\\ &d_0, c_0, b_0, a_0, a_1, a_2, \ldots, a_{64}, h_0, g_0, f_0, e_0, e_1, e_2, \ldots, e_{64},\\ &Ch_1, \ldots, Ch_{64}, sum1_1, \ldots, sum1_{64}, sum2_1, \ldots, sum3_{64}, temp1_1, \ldots, temp1_{64},\\ &Maj_1, \ldots, Maj_{64}, temp2_1, \ldots, temp2_{64}, k_1, \ldots, k_{64}, H_1, \ldots, H_8) \end{aligned}$$

Donde todas las variables tienen 32 bits, excepto la unidad. El orden pensado para las ecuaciones es el siguiente, aunque una permutación en las filas de todas las matrices también daría un resultado exitoso:

Generación de las variables *W*

1. $t1_1 = W_1 + W_{10}$
...
48. $t1_{48} = W_{48} + W_{57}$
49. $t2_1 = t1_1 + \sigma_1(W_{15})$
...
96. $t2_{48} = t1_{48} + \sigma_1(W_{62})$
97. $W_{17} = t2_1 + \sigma_0(W_2)$
...
144. $W_{64} = t2_{48} + \sigma_0(W_{49})$

Actualización de las variables de estado

$Ch_i(e_{i-1}, e_{i-2}, e_{i-3})$
$Sum1_i = e_{i-4} + \Sigma_1(e_{i-1})$
$Sum2_i = Sum1_i + Ch_i$
$Sum3_i = Sum2_i + k_i$
$temp1_i = Sum3_i + W_i$

$Maj_i(a_{i-1}, a_{i-2}.a_{i-3})$
$temp2_i = Maj_i + \Sigma_0(a_{i-1})$

$a_i = temp1 + temp2$
$e_i = a_{i-4} + temp1_i$

Obtención de las nuevas constantes $H_1, \ldots, H_8$

721. $H_1 = a_0 + a_{64}$
...
728. $H_8 = h_0 + e_{61}$

Como era de esperar, hay 728 operaciones que implican 32 *constraints* cada una. En total hacen 23296 *constraints*.

Estos resultados han sido comprobados empíricamente en un código de Python que se adjunta en los anexos y que ofrece la posibilidad de ver que estas matrices están mayoritariamente formadas por ceros, "*Sparse Matrices*". En las últimas líneas del código adjunto se aprecia que el porcentaje de unos es del 0.01158 % en total y que como mucho, en la matriz C se pueden encontrar 6 unos (y por tanto 26107 ceros) en una fila.

Cabe destacar que este sistema no es el más óptimo posible en cuanto a dimensión matricial. Existe una optimización eliminando la primera ecuación de cada suma módulo 2^{32}. La razón es sencilla, esa ecuación crea una variable que significa solo una suma XOR y el circuito seguiría siendo equivalente. El problema reside en que si la variable se sustituye por su suma XOR, no es tan fácil obtener un patrón para la representación de las matrices.

Por ejemplo, véase la implementación R1CS para la compuerta $Temp1$, que necesita 4 sumas módulo 2^{32}. En la primera suma se elimina esta ecuación y en la segunda suma, en lugar de $Sum1_1$ debería aparecer $h_1 \oplus [e_7 \oplus e_{12} \oplus e_{26}]$. Sucesivamente, en la última suma módulo 2^{32} debería aparecer en lugar de $Sum3_1$ la concatenación de XOR: $h_1 \oplus [e_7 \oplus e_{12} \oplus e_{26}] \oplus Ch_1 \oplus k_1$. Efectivamente, por cada suma módulo 2^{32} se elimina una variable y una ecuación pero no se obtiene ninguna fórmula general.

Si se consiguiese, al haber 600 sumas módulo 2^{32}, supondría un ahorro tanto de 600 variables como 600 *constraints*. Es decir, un 1,64 % de filas y un 2,3 % de columnas menos. Un futuro trabajo puede ser elaborar un método automático para la optimización de la terna de matrices de un R1CS, lo cual puede ser posible.

Capítulo 6

Conclusiones

Este Trabajo de Fin de Grado ha logrado construir un sistema de ecuaciones para el circuito aritmético relacionado a la función SHA-256. Gracias a este sistema, se puede delegar el cálculo de esta función a otra entidad con la certeza de que el *output* recibido es el correcto. SHA-256 es un hash muy utilizado y este resultado puede ser interesante en futuras implementaciones de SNARKs, con aplicaciones por ejemplo en Blockchain.

SHA-256 es una función concreta. Si se quisiera implementar un R1CS a otra función, sería necesario replantear todo el sistema matricial. Por ejemplo, SHA-512 es un algoritmo distinto a SHA-256. Sin embargo, las operaciones dentro de él son muy similares, como son AND, XOR y la suma módulo 2^{64}. Afortunadamente, desarrollando el sistema para SHA-256 se han obtenido numerosos R1CS para determinadas funciones, como es el caso de la suma módulo 2^n por lo que construir un sistema R1CS para SHA-512 puede ser solo una cuestión de aplicar el lema de composición.

Este sistema está construido sobre el cuerpo binario $\mathbb{F}_2$ por su gran cantidad de operaciones binarias, pero para cuerpos $\mathbb{F}_p$ con $p > 2$ las ecuaciones cambian y algunas no se satisfacen. La motivación de construir sistemas R1CS sobre cuerpos reside en las propiedades de evaluación de polinomios cuando se implementan dentro de SNARKs.

Capítulo 7

Anexos

```
global k

k=[]
k.append('01000010100010100010111110011000')
k.append('01110001001101110100010010010001')
k.append('10110101110000001111101111001111')
k.append('11101001101101011101101110100101')
k.append('00111001010101101100001001011011')
k.append('01011001111100010001000111110001')
k.append('10010010001111111000001010100100')
k.append('10101011000111000101111011010101')
k.append('11011000000001111010101010011000')
k.append('00010010100000110101101100000001')
k.append('00100100001100011000010110111110')
k.append('01010101000011000111110111000011')
k.append('01110010101111100101110101110100')
k.append('10000000110111101011000111111110')
k.append('10011011110111000000011010100111')
k.append('11000001100110111111000101110100')
k.append('11100100100110110110100111000001')
k.append('11101111101111100100011110000110')
k.append('00001111110000011001110111000110')
k.append('00100100000011001010000111001100')
k.append('00101101111010010010110001101111')
k.append('01001010011101001000010010101010')
k.append('01011100101100001010100111011100')
k.append('01110110111111001100010011011010')
k.append('10011000001111100101000101010010')
k.append('10101000001100011100011001101101')
k.append('10110000000000011001001111001000')
k.append('10111111010110010111111111000111')
k.append('11000110111000000000101111110011')
k.append('11010101101001111001000101000111')
k.append('00000110110010101001100011010010001')
k.append('00010100001010010010100101100111')
k.append('00100111101101110000101010000101')
k.append('00101110000110110010000100111000')
k.append('01001101001011000110110111111100')
k.append('01010011001110000000110100010011')
```

```
k.append('01100101000010100111001101010100')
k.append('01110110011010100000101010111011')
k.append('10000001110000101100100100101110')
k.append('10010010011100100010110010000101')
k.append('10100010101111111111010001010001')
k.append('10101000000110100110011001001011')
k.append('11000010010010111000101101110000')
k.append('11000111011011000101000110100011')
k.append('11010001100100101111010000011001')
k.append('11010110100110010000011000100100')
k.append('11110100000011100011010110000101')
k.append('00010000011010101010000001110000')
k.append('00011001101001001100000100010110')
k.append('00011110001101110110110000001000')
k.append('00100111010010000111011101001100')
k.append('00110100101100001011110010110101')
k.append('00111001000111000000110010110011')
k.append('01001110110110001010101001001010')
k.append('01011011100111001100101001001111')
k.append('01101000001011100110111111110011')
k.append('01110100100011111100000101110110')
k.append('01111000101001010110001101101111')
k.append('10000100110010000111100000010100')
k.append('10001100110001110000001000001000')
k.append('10010000101111101111111111111010')
k.append('10100100010100000011011001101011')
k.append('10111110111110011010001111110111')
k.append('11000110011100010111100011110010')


A = '01101010000010011110011001100111'
B = '10111011011001111010111010000101'
C = '00111100011011101111001101110010'
D = '10100101010011111111010100111010'
E = '01010001000011100101001001111111'
F = '10011011000001010110100010001100'
G = '00011111100000111101100110101011'
H = '01011011111000001100110100011001'

global A
global B
global C
global D
global E
global F
global G
global H

def RotR(X,m):
    Y=''
    for i in range(len(X)):
        Y += X[(i-m)%len(X)]
    return Y

def ShR(X,m):
```

```
95      Y=''
96      for i in range(len(X)):
97          if(i<m):
98              Y += '0'
99          else:
100             Y += X[(i-m)%len(X)]
101     return Y
102
103 def s0(X):
104     lista=''
105     a = RotR(X,7)
106     b = RotR(X,18)
107     c = ShR(X,3)
108     for i in range(32):
109         lista += str((int(a[i]) + int(b[i]) + int(c[i]))%2)
110     return lista
111
112 def s1(X):
113     lista=''
114     a = RotR(X,17)
115     b = RotR(X,19)
116     c = ShR(X,10)
117     for i in range(32):
118         lista += str((int(a[i]) + int(b[i]) + int(c[i]))%2)
119     return lista
120
121 def S0(X):
122     lista=''
123     a = RotR(X,2)
124     b = RotR(X,13)
125     c = RotR(X,22)
126     for i in range(32):
127         lista += str((int(a[i]) + int(b[i]) + int(c[i]))%2)
128     return lista
129
130 def S1(X):
131     lista=''
132     a = RotR(X,6)
133     b = RotR(X,11)
134     c = RotR(X,25)
135     for i in range(32):
136         lista += str((int(a[i]) + int(b[i]) + int(c[i]))%2)
137     return lista
138
139 def sum1(X,Y):
140     z=''
141     c=0
142     for i in range(31,-1,-1):
143         z += str((c + int(X[i])+int(Y[i]))%2)
144         c = (int(X[i])*int(Y[i]) + c*int(X[i]) + c*int(Y[i]))%2
145     return z[::-1] #outputs z32...z1
146
147 def majority(x,y,z):
148     return ((int(x)+int(y))*(int(y)+int(z)) + int(y))%2
149
```

```
150 def choice(x,y,z):
151     return (int(x)*(int(y)+int(z)) + int(z))%2
152 
153 def padding(M):
154 
155     o=''
156 
157     l = ''.join("{0:08b}".format(int(int.from_bytes(bytes(bytearray(p.
    encode('utf-8'))), byteorder='big')), 'b') for p in M)
158 
159     o = ''.join("{0:064b}".format(len(l), 'b'))  #Original message length
    , represented in binary, 64 bits
160 
161     l+='10000000'                                #Add one 1 and seven 0
    to the original message
162 
163     while((len(l)+64)%512 != 0):
164         l+='0'                                   #Padding
165     l+= o                                        #Add at the very end the
    64 bits that represents the length of
166                                                  #the original message
167 
168     return l
169 
170 def iteration(a,b,c,d,e,f,g,h,l):
171     wit = ''
172     W=[]
173     l1=[]
174     l2=[]
175 
176     Cho=[]
177     Temp11=[]
178     Temp12=[]
179     Temp13=[]
180     Temp1out=[]
181 
182     Maj=[]
183     Temp2out=[]
184 
185     elist=[]
186 
187     for t in range(16):                          #First 16 variables W
188         W.append(l[32*t:32*(t+1)])
189         wit += l[32*t:32*(t+1)][::-1]
190 
191     for i in range(16,64):
192         t1 = sum1(W[i-16],W[i-7])
193         t2 = sum1(t1,s1(W[i-2]))
194         t3 = sum1(t2,s0(W[i-15]))
195         W.append(t3)                                 #All variables W
196         l1.append(t1)
197         l2.append(t2)
198 
199     for t in range(16,64):
```

```
200         wit += W[t][::-1]    #REVERSE STRING, we need to do this in order
    to call the rightest bit as the first
201
202     for t in range(16,64):
203         wit += l1[t-16][::-1]
204
205     for t in range(16,64):
206         wit += l2[t-16][::-1]
207
208     wit += d[::-1]+c[::-1]+b[::-1]+a[::-1] #In order to get a patern, we
    add this variables to the witness now.
209
210     a0 = a
211     b0 = b
212     c0 = c
213     d0 = d
214     e0 = e
215     f0 = f
216     g0 = g
217     h0 = h
218
219     for j in range(64):
220     #--------------------------------------------------------------------
221
222         Ch='' #Temp1
223         for i in range(32):
224             Ch += str(choice(e0[i],f0[i],g0[i]))
225         Cho.append(Ch)
226
227         e1 = S1(e0)
228
229         t1 = sum1(h0,e1)
230         Temp11.append(t1)
231
232         t2 = sum1(t1,Ch)
233         Temp12.append(t2)
234
235         t3 = sum1(t2,k[j])
236         Temp13.append(t3)
237
238         temp1 = sum1(t3,W[j])
239         Temp1out.append(temp1)
240     #--------------------------------------------------------------------
241
242
243         a1 = S0(a0) #Temp2
244         m = ''
245
246         for i in range(32):
247             m += str(majority(a0[i],b0[i],c0[i]))
248         Maj.append(m)
249
250         temp2 = sum1(a1,m)
251         Temp2out.append(temp2)
252     #--------------------------------------------------------------------
```

```
253
254         h0 = g0
255         g0 = f0
256         f0 = e0
257         e0 = sum1(d0,temp1)
258
259         elist.append(e0)
260
261         d0 = c0
262         c0 = b0
263         b0 = a0
264         a0 = sum1(temp1,temp2)
265
266         wit += a0[::-1]
267
268     wit += h[::-1]+g[::-1]+f[::-1]+e[::-1]
269
270     for t in range(64):
271         wit += elist[t][::-1]
272
273     for t in range(64):
274         wit += Cho[t][::-1]
275
276     for t in range(64):
277         wit += Temp11[t][::-1]
278
279     for t in range(64):
280         wit += Temp12[t][::-1]
281
282     for t in range(64):
283         wit += Temp13[t][::-1]
284
285     for t in range(64):
286         wit += Temp1out[t][::-1]
287
288     for t in range(64):
289         wit += Maj[t][::-1]
290
291     for t in range(64):
292         wit += Temp2out[t][::-1]
293
294     for t in range(64):
295         wit += k[t][::-1]
296
297     a = sum1(a0,a)
298     b = sum1(b0,b)
299     c = sum1(c0,c)
300     d = sum1(d0,d)
301     e = sum1(e0,e)
302     f = sum1(f0,f)
303     g = sum1(g0,g)
304     h = sum1(h0,h)
305
306     wit += a[::-1]+b[::-1]+c[::-1]+d[::-1]+e[::-1]+f[::-1]+g[::-1]+h
    [::-1]
```

```
307
308     wit = '1' + wit
309
310     return a,b,c,d,e,f,g,h,wit
311
312 def SHA256(M):
313
314     l = padding(M)
315
316     v = iteration(A,B,C,D,E,F,G,H,l)
317
318     while(len(l)>512):
319         l = l[512:]
320         v = iteration(v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7],l)
321
322     a1 = "{0:0>8x}".format(int(v[0], 2))
323     b1 = "{0:0>8x}".format(int(v[1], 2))
324     c1 = "{0:0>8x}".format(int(v[2], 2))
325     d1 = "{0:0>8x}".format(int(v[3], 2))
326     e1 = "{0:0>8x}".format(int(v[4], 2))
327     f1 = "{0:0>8x}".format(int(v[5], 2))
328     g1 = "{0:0>8x}".format(int(v[6], 2))
329     h1 = "{0:0>8x}".format(int(v[7], 2))
330
331     result = a1+b1+c1+d1+e1+f1+g1+h1
332
333     print(result)
334
335     return v[8]
336
337 #Now, the witness order follows:
338
339 #1,w^{1}_1,...,w^{32}_1,w^{1}_2,...,w^{32}_16,w{1}_17,...,w^{32}_64,t1
    ^{1}_1,...,t1^{1}_2,...,t1^{32}_48,t2^{1}_1...,t2^{32}_48,
340
341 #(now each variable is supposed to have 32 bits)
342
343 #d0,c0,b0,a0,a1,a2,....,a64,h0,g0,f0,e0,e1,e2,...,e64,Ch_1,...,Ch_64,
    sum1_1,...,sum1_64,sum2_1,...,sum3_64
344 #Temp1_1,....,Temp1_64,Maj_1,...,Maj_64,Temp2_1,...,Temp2_64,k_1,...,k_64
    ,H_1,...,H_8
345
346 def Choice_R1CS(m1,m2,m3,row,e,f,g,Ch): #Put the exact witness position
    for e,f,g,Ch
347
348     for i in range(32):
349
350         m1[row+i][e-1+i] = 1
351
352         m2[row+i][f-1+i] = 1
353         m2[row+i][g-1+i] = 1
354
355         m3[row+i][Ch-1+i] = 1
356         m3[row+i][g-1+i] = 1
357
```

```
358     return m1,m2,m3
359 
360 def Majority_R1CS(m1,m2,m3,row,a,b,c,Maj): #Put the exact witness
    position for a,b,c,Maj
361 
362     for i in range(32):
363 
364         m1[row+i][a-1+i] = 1
365         m1[row+i][b-1+i] = 1
366 
367         m2[row+i][c-1+i] = 1
368         m2[row+i][b-1+i] = 1
369 
370         m3[row+i][b-1+i] = 1
371         m3[row+i][Maj-1+i] = 1
372 
373     return m1,m2,m3
374 
375 def Sumod_2_32_R1CS(m1,m2,m3,row,sum1,sum2,total): #Put the exact witness
     position for sum1,sum2,total
376 
377     #scheme for sum modulo 2^32                          [a+b=z]
378 
379          # z1 = a1 + b1,  ||      0*0 +a1+b1+z1 = 0
380          # z2 = a1*b1 + a2 + b2
381          # zi = (zi-1 + bi-1)*(ai-1 + bi-1) + ai + bi + ai-1
382 
383     m3[row][sum1-1] = 1
384     m3[row][sum2-1] = 1
385     m3[row][total-1] = 1
386 #------------------------------
387 
388     m1[row+1][sum1-1] = 1
389 
390     m2[row+1][sum2-1] = 1
391 
392     m3[row+1][sum1] = 1
393     m3[row+1][sum2] = 1
394     m3[row+1][total] = 1
395 
396 
397     for i in range(2,32):
398 
399         m1[row+i][total-2+i] = 1 #total^(i-1)
400         m1[row+i][sum2-2+i] = 1 #sum2^(i-1)
401 
402         m2[row+i][sum1-2+i] = 1 #sum1^(i-1)
403         m2[row+i][sum2-2+i] = 1 #sum2^(i-1)
404 
405         m3[row+i][sum1-1+i] = 1 #sum1^(i)
406         m3[row+i][sum2-1+i] = 1 #sum2^(i)
407         m3[row+i][sum1-2+i] = 1 #sum1^(i-1)
408         m3[row+i][total-1+i] = 1 #total^(i)
409 
410     return m1,m2,m3
```

```
411
412 def Sumod_2_32_R1CS_s(m1,m2,m3,row,sum1,s,total,rot1,rot2,sh,value):
413
414     #Where s is the precise position of the first bit of the variable
    before rotation
415     #value = {0,1} determinates whether a shift rigth rotation is
    required
416
417     m3[row][sum1-1] = 1
418     m3[row][s-1+sh] = 1
419     m3[row][s-1+rot1] = 1
420     m3[row][s-1+rot2] = 1
421     m3[row][total-1] = 1
422 #------------------------------
423
424     m1[row+1][sum1-1] = 1
425
426     m2[row+1][s-1+sh] = 1
427     m2[row+1][s-1+rot1] = 1
428     m2[row+1][s-1+rot2] = 1
429
430     m3[row+1][sum1] = 1
431     m3[row+1][s+sh] = 1
432     m3[row+1][s+rot1] = 1
433     m3[row+1][s+rot2] = 1
434     m3[row+1][total] = 1
435
436
437     for i in range(2,32):
438
439         if(i<32-sh):
440
441             m1[row+i][total-2+i] = 1 #total^(i-1)
442             m1[row+i][s-1+(sh+i-1)%32] = 1 #s1^(i-1)
443             m1[row+i][s-1+(rot1+i-1)%32] = 1
444             m1[row+i][s-1+(rot2+i-1)%32] = 1
445
446             m2[row+i][sum1-2+i] = 1 #sum1^(i-1)
447             m2[row+i][s-1+(sh+i-1)%32] = 1 #s1^(i-1)
448             m2[row+i][s-1+(rot1+i-1)%32] = 1
449             m2[row+i][s-1+(rot2+i-1)%32] = 1
450
451             m3[row+i][sum1-1+i] = 1 #sum1^(i)
452             m3[row+i][s-1+(sh+i)%32] = 1 #s1^(i)
453             m3[row+i][s-1+(rot1+i)%32] = 1
454             m3[row+i][s-1+(rot2+i)%32] = 1
455             m3[row+i][sum1-2+i] = 1 #sum1^(i-1)
456             m3[row+i][total-1+i] = 1 #total^(i)
457
458         elif(i==32-sh):
459
460             m1[row+i][total-2+i] = 1 #total^(i-1)
461             m1[row+i][s-2+sh+i] = 1 #s1^(i-1)
462             m1[row+i][s-1+(rot1+i-1)%32] = 1
463             m1[row+i][s-1+(rot2+i-1)%32] = 1
```

```
464
465             m2[row+i][sum1-2+i] = 1 #sum1^(i-1)
466             m2[row+i][s-2+sh+i] = 1 #s1^(i-1)
467             m2[row+i][s-1+(rot1+i-1)%32] = 1
468             m2[row+i][s-1+(rot2+i-1)%32] = 1
469
470             m3[row+i][sum1-1+i] = 1 #sum1^(i)
471             m3[row+i][s-1+(sh+i)%32] = 1*value #s1^(i)
472             m3[row+i][s-1+(rot1+i)%32] = 1
473             m3[row+i][s-1+(rot2+i)%32] = 1
474             m3[row+i][sum1-2+i] = 1 #sum1^(i-1)
475             m3[row+i][total-1+i] = 1 #total^(i)
476
477         else:
478
479             m1[row+i][total-2+i] = 1 #total^(i-1)
480             m1[row+i][s-1+(sh+i-1)%32] = 1*value #s1^(i-1)
481             m1[row+i][s-1+(rot1+i-1)%32] = 1
482             m1[row+i][s-1+(rot2+i-1)%32] = 1
483
484             m2[row+i][sum1-2+i] = 1 #sum1^(i-1)
485             m2[row+i][s-1+(sh+i-1)%32] = 1*value #s1^(i-1)
486             m2[row+i][s-1+(rot1+i-1)%32] = 1
487             m2[row+i][s-1+(rot2+i-1)%32] = 1
488
489             m3[row+i][sum1-1+i] = 1 #sum1^(i)
490             m3[row+i][s-1+(sh+i)%32] = 1*value #s1^(i)
491             m3[row+i][s-1+(rot1+i)%32] = 1
492             m3[row+i][s-1+(rot2+i)%32] = 1
493             m3[row+i][sum1-2+i] = 1 #sum1^(i-1)
494             m3[row+i][total-1+i] = 1 #total^(i)
495
496
497     return m1,m2,m3
498
499 def R1CS():
500
501     import numpy as np
502
503     A = np.zeros((23296, 26113))
504     B = np.zeros((23296, 26113))
505     C = np.zeros((23296, 26113))
506
507     #FIRST PHASE
508
509     for m in range(48):
510
511         A,B,C = Sumod_2_32_R1CS(A,B,C,m*96,2+m*32,290+m*32,2050+m*32) #m1
    ,m2,m3,row,w1,w10,t1
512
513         A,B,C = Sumod_2_32_R1CS_s(A,B,C,32+m*96,2050+m*32,450+m*32,3586+m
    *32,17,19,10,0) #m1,m2,m3,row,t1,s1,t2 s1=w^{1}_15
514
515         A,B,C = Sumod_2_32_R1CS_s(A,B,C,64+m*96,3586+m*32,34+m*32,514+m
    *32,7,18,3,0) #m1,m2,m3,row,t2,s0,w17 s0=w^{1}_2
```

```
516
517                                 #m1,m2,m3,row,sum1,s,total,rot1,rot2,(sh/
    rot3),value
518                                 # if    value = 0 -> do shift rotate right
519                                 # else value = 1 -> do another rotate
    right
520                                 # s is the first bit position of the
    variable before rotation
521
522     #SECOND PHASE
523
524     for m in range(64):
525
526         A,B,C = Choice_R1CS(A,B,C,4608+m*288,7394+m*32,7362+m*32,7330+m
    *32,9474+m*32) #m1,m2,m3,row,e,f,g,Ch
527         A,B,C = Sumod_2_32_R1CS_s(A,B,C,4608+32+m*288,7298+m*32,7394+m
    *32,11522+m*32,6,11,25,1) #m1,m2,m3,row,h,S1,Sum1
528         A,B,C = Sumod_2_32_R1CS(A,B,C,4608+64+m*288,11522+m*32,9474+m
    *32,13570+m*32) #m1,m2,m3,row,Sum1,Ch,Sum2
529         A,B,C = Sumod_2_32_R1CS(A,B,C,4608+96+m*288,13570+m*32,23810+m
    *32,15618+m*32) #m1,m2,m3,row,Sum2,k,Sum3
530         A,B,C = Sumod_2_32_R1CS(A,B,C,4608+128+m*288,15618+m*32,2+m
    *32,17666+m*32) #m1,m2,m3,row,Sum3,w,Temp1
531
532         A,B,C = Majority_R1CS(A,B,C,4608+160+m*288,5218+m*32,5186+m
    *32,5154+m*32,19714+m*32)
533         A,B,C = Sumod_2_32_R1CS_s(A,B,C,4608+192+m*288,19714+m*32,5218+m
    *32,21762+m*32,2,13,22,1) #m1,m2,m3,row,Maj1,a0,Temp2
534
535         A,B,C = Sumod_2_32_R1CS(A,B,C,4608+224+m*288,17666+m*32,21762+m
    *32,5250+m*32) #m1,m2,m3,row,Temp1,Temp2,a1
536         A,B,C = Sumod_2_32_R1CS(A,B,C,4608+256+m*288,17666+m*32,5122+m
    *32,7426+m*32) #m1,m2,m3,row,Temp1,d0,e1
537
538
539     #THIRD PHASE || H1 = a0 + a64, H2 = b0 + a63, H3 = c0 + a62, H4 = d0
    + a61
540
541     for m in range(4):
542
543         A,B,C = Sumod_2_32_R1CS(A,B,C,23040+m*32,7266-m*32,5218-m
    *32,25858+m*32) #m1,m2,m3,row,(a64,a0,H1)
544
545     #FOURTH PHASE || H5 = e0 + e64, H6 = f0 + e63, H7 = g0 + e62, H8 = h0
     + e61
546
547     for m in range(4):
548
549         A,B,C = Sumod_2_32_R1CS(A,B,C,23168+m*32,9442-m*32,7394-m
    *32,25986+m*32) #m1,m2,m3,row,(e64,e0,H5)
550
551
552     return A,B,C
553
554 def Check(M):
```

```
    import numpy as np

    matrices=R1CS()

    l = padding(M)

    v = iteration(A,B,C,D,E,F,G,H,l)

    wit = list(map(int,v[8]))
    print(sum((np.multiply(np.dot(matrices[0], wit),np.dot(matrices[1],
   wit)) + np.dot(matrices[2],wit))%2 == 0) == 23296)

    while(len(l)>512):

        l = l[512:]
        v = iteration(v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7],l)

        wit = list(map(int,v[8]))
        print(sum((np.multiply(np.dot(matrices[0], wit),np.dot(matrices
   [1], wit)) + np.dot(matrices[2],wit))%2 == 0) == 23296)

    a1 = "{0:0>8x}".format(int(v[0], 2))
    b1 = "{0:0>8x}".format(int(v[1], 2))
    c1 = "{0:0>8x}".format(int(v[2], 2))
    d1 = "{0:0>8x}".format(int(v[3], 2))
    e1 = "{0:0>8x}".format(int(v[4], 2))
    f1 = "{0:0>8x}".format(int(v[5], 2))
    g1 = "{0:0>8x}".format(int(v[6], 2))
    h1 = "{0:0>8x}".format(int(v[7], 2))

    result = a1+b1+c1+d1+e1+f1+g1+h1

    print(result)

#-------------------------------------------------------------

Check('hola esto es una prueba para comprobar si esto funciona')

True
7d55c7b449309f8f6b1768705086e24b18b5ba4e544799304c564c2573c9ae0a

#-------------------------------------------------------------

Check('hola esto es una prueba para comprobar si esto funcionaa')

True
True
e6e628b43309d5e2a37fc3233e109bc20e356f5ff57007d9a746e67cbceaa4a7

#-------------------------------------------------------------

sum(matrices[0][:][:]==1)

55656
```

```
608
609 #------------------------------------------------------------
610
611 sum(matrices[1][:][:]==1)
612
613 58152
614
615 #------------------------------------------------------------
616
617 sum(matrices[2][:][:]==1)
618
619 97504
620
621 #------------------------------------------------------------
622
623 maxim=0
624
625 for i in range(23296):
626
627     maxim = max(maxim,sum(matrices[0][i][:]==1))
628
629 print(maxim)
630
631 4
632
633 #------------------------------------------------------------
634
635 maxim=0
636
637 for i in range(23296):
638
639     maxim = max(maxim,sum(matrices[1][i][:]==1))
640
641 print(maxim)
642
643 4
644
645 #------------------------------------------------------------
646
647 maxim=0
648
649 for i in range(23296):
650
651     maxim = max(maxim,sum(matrices[2][i][:]==1))
652
653 print(maxim)
654
655 6
656
657 #------------------------------------------------------------
658
659 float((55656+58152+97504)/(3*23296*26113))*100
660
661 0.011578832711984783
```

Bibliografía

[AB09] Sanjeev Arora y Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[Bar+02] Elaine B Barker y col. "Secure Hash Standard (SHS)". En: (2002).

[Bec08] Georg Becker. "Merkle signature schemes, merkle trees and their cryptanalysis". En: *Ruhr-University Bochum, Tech. Rep* 12 (2008), pág. 19.

[Ben+13] Eli Ben-Sasson y col. "SNARKs for C: Verifying program executions succinctly and in zero knowledge". En: *Annual cryptology conference*. Springer. 2013, págs. 90-108.

[Buc] William J Buchanan. *https://asecuritysite.com/encryption/go_qap*.

[Dhu+17] Saurabh Dhumwad y col. "A peer to peer money transfer using SHA256 and Merkle tree". En: *2017 23RD Annual International Conference in Advanced Computing and Communications (ADCOM)*. IEEE. 2017, págs. 40-43.

[Dom22] Domingo Martín. *https://sha256algorithm.com*. 2022.

[ElG85] Taher ElGamal. "A public key cryptosystem and a signature scheme based on discrete logarithms". En: *IEEE transactions on information theory* 31.4 (1985), págs. 469-472.

[Gen+13] Rosario Gennaro y col. "Quadratic span programs and succinct NIZKs without PCPs". En: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2013, págs. 626-645.

[GMR89] Shafi Goldwasser, Silvio Micali y Charles Rackoff. "The knowledge complexity of interactive proof systems". En: *SIAM Journal on computing* 18.1 (1989), págs. 186-208.

[Gup17] Sourav Sen Gupta. "Blockchain". En: *IBM Onlone (http://www. IBM. COM)* (2017).

[Igl18] Judit Iglesias Jurado. "zk-SNARK o cómo garantizar la privacidad en las criptomonedas. De la teoría a la práctica." En: (2018).

[JMV01] Don Johnson, Alfred Menezes y Scott Vanstone. "The elliptic curve digital signature algorithm (ECDSA)". En: *International journal of information security* 1.1 (2001), págs. 36-63.

[Kra03] Hugo Krawczyk. "SIGMA: The 'SIGn-and-MAc'approach to authenticated Diffie-Hellman and its use in the IKE protocols". En: *Annual international cryptology conference*. Springer. 2003, págs. 400-425.

[MAI] MAII - FIB. *https://helix.stormhub.org/papers/SHA-256.pdf*.

[MR09] Daniele Micciancio y Oded Regev. "Lattice-based Cryptography". En: *Post-Quantum Cryptography*. Ed. por Daniel J. Bernstein, Johannes Buchmann y Erik Dahmen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, págs. 147-191. ISBN: 978-3-540-88702-7. DOI: 10.1007/978-3-540-88702-7_5. URL: https://doi.org/10.1007/978-3-540-88702-7_5.

[PB61] William Wesley Peterson y Daniel T Brown. "Cyclic codes for error detection". En: *Proceedings of the IRE* 49.1 (1961), págs. 228-235.

[RRT17] Carlos Dolader Retamal, Joan Bel Roig y José Luis Muñoz Tapia. "La blockchain: fundamentos, aplicaciones y relación con otras tecnologías disruptivas". En: *Economía industrial* 405 (2017), págs. 33-40.

[Sch91] Claus-Peter Schnorr. "Efficient signature generation by smart cards". En: *Journal of cryptology* 4.3 (1991), págs. 161-174.

[Tha21] Justin Thaler. *Proofs, arguments, and zero-knowledge*. 2021.

[WY05] Xiaoyun Wang y Hongbo Yu. "How to break MD5 and other hash functions". En: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2005, págs. 19-35.