Treball Fi de Grau - Curs 2022/2023

Bases de dades distribuïdes per cadenes de blocs

Blockchain

Autora: **Irene Barrera Bernad**

Tutors: ENRIC COSME LLÓPEZ
JUAN FRANCISCO CORRECHER VALLS

Facultat de Matemàtiques Grau en Matemàtiques

Índex

Capítol 1

Introducció

La cadena de blocs, o *blockchain* en anglès, és una tecnologia emergent que està revolucionant diverses àrees d'aplicació. Podem pensar-la com un gran llibre de registre digital compartit que registra totes les transaccions i interaccions realitzades en una xarxa descentralitzada. Aquesta tecnologia, que combina conceptes de criptografia, teoria de grups, funcions de resum i altres àrees matemàtiques, com Estructures Algebraiques, Matemàtica Bàsica i Informàtica, ofereix una solució innovadora per a problemes com ara la seguretat, la transparència i la confiança en les transaccions digitals.

Podem imaginar una cadena de blocs com una sèrie de blocs connectats entre ells. Cada bloc conté informació sobre transaccions o registres de dades, i està lligat al bloc anterior. Això fa que la cadena de blocs siga immutable i resistent a l'alteració.

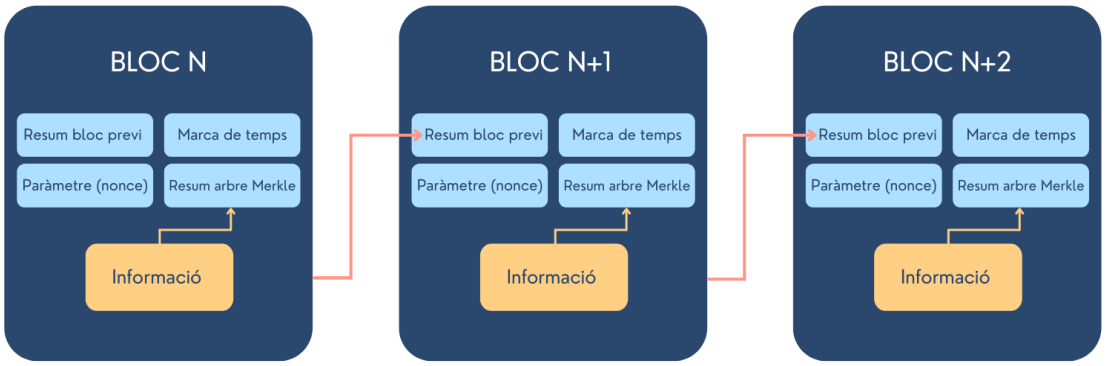

Figura 1.1: Estructura dels blocs de la cadena de blocs de Bitcoin. Font: elaboració pròpia.

En la Figura 1.1, es mostra com s'enllacen els blocs de la cadena i la informació que emmagatzema. Cada bloc conté informació vital, com les dades de transacció, el valor de resum (*hash*) del bloc anterior, un paràmetre aleatori únic (*nonce*) per a la seguretat i una

marca de temps (*timestamp*) per a la referència temporal. El fet de que cada bloc també inclou el valor de resum del bloc anterior en la cadena, crea una connexió seqüencial entre els blocs, ja que el valor de resum del bloc anterior és utilitzat com a part de la informació per generar el valor de resum del bloc següent.

D'aquesta manera, si es realitza alguna modificació en un bloc, aquesta afectarà als valors de resum dels blocs següents, alertant així de possibles manipulacions en la cadena de blocs. Aquest esquema de vinculació mitjançant valors de resum proporciona una integritat i seguretat addicionals a la cadena de blocs, assegurant que els blocs estiguen enllaçats de manera confiable i permetent la verificació de la integritat de la cadena en el seu conjunt.

La gran novetat és que la cadena de blocs no depèn d'una única autoritat central, sinó que és gestionada per una xarxa de nodes o ordinadors distribuïts pel món. Aquests nodes treballen col·lectivament per verificar i validar les transaccions, assegurant-se que siguen vàlides i consensuades per tota la xarxa. Això té implicacions significatives en diferents àmbits. En el cas de les criptomonedes, com Bitcoin [32], la cadena de blocs s'utilitza per a registrar i verificar les transaccions de manera segura i sense necessitat d'intermediaris. També s'aplica en altres àmbits com la logística i la gestió de dades.

La cadena de blocs representa una innovació amb la capacitat de canviar la forma en què ens relacionem i portem a terme operacions en la societat actual.

En aquest treball de fi de grau, explorarem a fons els fonaments matemàtics de la cadena de blocs i la seua relació amb altres àrees de la matemàtica. En el capítol 2, començarem amb una introducció a la teoria de grups, ja que alguns problemes d'aquesta teoria s'utilitzen en criptografia i la cadena de blocs necessita establir comunicacions segures per a la seua estructura i funcionament. En aquesta secció, estudiarem les presentacions de grups, els problemes algorísmics relacionats amb aquesta àrea i la reescriptura de les presentacions de grups per a millorar l'eficiència i la seguretat. A continuació, aprofundirem en la criptografia, que és l'eina fonamental per assegurar la privacitat i la integritat de les transaccions en la cadena de blocs. Explorarem els principis bàsics de la criptografia asimètrica i simètrica, així com els protocols de seguretat més utilitzats en aquest context.

Un altre aspecte fonamental és l'ús de funcions de resum en la cadena de blocs. En el capítol 3, explorarem les propietats de les funcions de resum i com són utilitzades per assegurar la integritat de les dades en la cadena de blocs. Estudiarem les funcions de resum simples, les seues aplicacions a seqüències de caràcters i els algorismes més utilitzats, com l'algorisme SHA-256, el qual analitzarem a fons.

Seguidament, en el capítol 4, explorarem en detall la cadena de blocs. Analitzarem l'origen i el context d'aquesta tecnologia, examinant els factors que han impulsat el seu desenvolupament i la seua ràpida adopció en diverses àrees. Estudiarem els fonaments tècnics de la cadena de blocs, incloent els seus components i els principis en els quals es basa. Dedicarem una secció a l'aplicació principal de la cadena de blocs en les criptomonedes, explorant en profunditat el seu funcionament en aquest àmbit. A més de les criptomonedes, analitzarem altres aplicacions rellevants de la cadena de blocs, com la seua integració amb l'Internet de les Coses (*Internet of Things, IoT*), la seua utilitat per a la

prova d'existència de documents i el seu paper en la seguretat del tractament de dades massives. Per concloure, en l'última secció, abordarem les solucions a alguns dels problemes que pateix la cadena de blocs. Destacarem els desafiaments associats amb aquesta tecnologia, com l'escalabilitat, el consum d'energia, la privacitat, els costos i la seguretat. També explorarem criptomonedes específiques que han implementat solucions per aquests problemes. A través d'aquest capítol, obtindrem una visió completa de la cadena de blocs. Acabarem amb unes breus conclusions en el capítol 5.

A través d'aquest treball esperem proporcionar una visió completa dels fonaments matemàtics de la cadena de blocs i les seues aplicacions, així com destacar la importància d'aquesta tecnologia en el món actual. Aprofundirem en els aspectes matemàtics clau, que són fonamentals per comprendre i explorar el potencial disruptiu de la cadena de blocs en diferents àrees d'aplicació.

Capítol 2

Teoria de grups i criptografia

En aquest capítol, donarem alguns conceptes necessaris de criptografia de clau pública i teoria combinatòria de grups. Les fonts principals on hem consultat aquesta informació són [31] i [45]. Ací podem trobar informació més detallada.

La criptografia de clau pública és una àrea consolidada, que ha estat molt activa des del seu principi oficial en 1976. Actualment, hi ha moltes direccions diferents en les investigacions d'aquest camp, però nosaltres ens fixarem en les àrees més fonamentals i bàsiques, principalment en l'establiment de claus, xifrat i autenticació.

La teoria combinatòria de grups, la tenim present des de fa molts anys i és una àrea ja establerta en les matemàtiques. Com que utilitzarem aquesta teoria connectada a la criptografia, ens centrarem principalment en problemes de teoria de grups, principalment algorísmics, que tenen una aplicació directa en criptografia.

2.1 Presentacions de grups

Definició 2.1.1. Un *grup* és un conjunt G amb una operació binària interna $(\cdot)$ que compleix:

- **G1 (associativa):** $(g_1 \cdot g_2) \cdot g_3 = g_1 \cdot (g_2 \cdot g_3)$ per a tot $g_1, g_2, g_3 \in G$.
- **G2 (neutre):** existeix $1 \in G$ tal que $g \cdot 1 = 1 \cdot g = g$, per a tot $g \in G$.
- **G3 (invers):** per a tot $g \in G$ existeix $h \in G$ tal que $g \cdot h = h \cdot g = 1$.

L'element 1 de $G2$ rep el nom d'*element neutre*. L'element h de $G3$ s'anomena *simètric* de g. És fàcil veure que per a cada g, el seu simètric és únic i sol denotar-se per g^{-1}. Per a simplificar notació evitarem escriure l'operació $\cdot$. Així, escriurem gh en compte de $g \cdot h$.

Un conjunt amb una operació interna que només satisfà la propietat associativa es diu *semigrup*. Quan una operació binària interna és associativa podem ometre els parèntesis

i escriure sense ambigüitat $abc = (ab)c = a(bc)$. Un grup no pot ser mai el conjunt buit, perquè almenys té l'element neutre.

Definició 2.1.2. Un *homomorfisme* entre dos grups G_1 i G_2 és una aplicació $f : G_1 \longrightarrow G_2$ que compleix

$$f(a \cdot b) = f(a) \cdot f(b) \qquad \forall\, a, b \in G_1. \tag{2.1}$$

Nota 2.1.1. *En la fórmula* (2.1)*, el primer producte és el producte en* G_1*, mentre que el segon correspon al producte en* G_2*.*

Definició 2.1.3. Siga $f : G_1 \longrightarrow G_2$ un homomorfisme de grups. Direm que:

1. f és *monomorfisme* si f és injectiva.
2. f és *epimorfisme* si f és suprajectiva.
3. f és *isomorfisme* si f és bijectiva.

Si $G_1 = G_2$ i f és un isomorfisme, direm que f és un *automorfisme*.

Definició 2.1.4. Es diu que dos grups G_1 i G_2 són *isomorfs* si existeix un isomorfisme entre ells. Es denota amb $G_1 \cong G_2$.

Definició 2.1.5. Siga G un grup.

1. Direm que G és *abelià* si per a tot $g_1, g_2 \in G$, es té que $g_1 g_2 = g_2 g_1$ (propietat *commutativa*).
2. Anomenarem *ordre* del grup G, denotat per $|G|$, al cardinal del conjunt G. Direm que un grup G és *finit* si el seu cardinal és finit. Un grup que no siga finit es dirà *infinit*.

Definició 2.1.6. Siga G un grup. Un *subgrup* de G és un subconjunt H de G que satisfà que $1 \in H$ i que té estructura de grup amb la llei de composició interna de G restringida a H. En aquest cas, escrivim $H \leq G$. És clar que $H = \{1\}$ i $H = G$ són subgrups de G.

Direm que H és un *subgrup propi* de G si $H \neq \{1\}$ i $H \neq G$.

Notació 2.1.1. Siga G un grup, $X, Y \subseteq G, x, y \in G$. Denotem:

1. $XY = \{xy \in G \mid x \in X, y \in Y\}$.
2. $X^{-1} = \{x^{-1} \in G \mid x \in X\}$.
3. Si $X = \{x\}, XY = xY = \{xy \mid y \in Y\}$. Si $Y = \{y\}, XY = Xy = \{xy \mid x \in X\}$.

4. Com que l'operació de G és associativa, pot denotar-se sense ambigüitat $XYZ = (XY)Z = X(YZ) = \{xyz \mid x \in X, y \in Y, z \in Z\}$.

   En particular, es farà servir la notació:

$$x^{-1}Yx = \{x^{-1}yx \in G \mid y \in Y\},$$

$$xYx^{-1} = \{xyx^{-1} \in G \mid y \in Y\}.$$

Definició 2.1.7. Siga G un grup. Si $g \in G$ i $x \in G$, aleshores gxg^{-1} es diu *conjugat* de x per g.

En la següent secció, veurem un problema algorísmic anomenat problema de la conjugació.

Definició 2.1.8. Un subgrup N d'un grup G es diu *normal* en G si per a tot $g \in G$, $gN = Ng$. Es denota per $H \trianglelefteq G$. Notem que $gN = Ng$ si, i només si $N = g^{-1}Ng$ si, i només si $N = gNg^{-1}$.

A més, també notem que $G \trianglelefteq G$ ($gG = G = Gg$ per a cada $g \in G$) i $\{1\} \trianglelefteq G$.

Un grup G es diu *simple* si $G \neq \{1\}$ i els únics subgrups normals de G són 1 i G.

Si G és abelià, aleshores tot subgrup de G és normal. El recíproc no és cert: Q_8 no és abelià, però tots les seus subgrups són normals.

Teorema 2.1.1. *Siga $N \trianglelefteq G$. Aleshores el conjunt quocient $G/N = \{xN \mid x \in G\}$ és un grup amb la llei de composició interna donada per*

$$\forall\, x, y \in G, \quad (xN)(yN) = (xy)N.$$

Proposició 2.1.1. *El grup G/N del Teorema* (2.1.1) *té estructura de grup i s'anomena* grup quocient de G per N.

Definició 2.1.9. Si $f : G_1 \longrightarrow G_2$ és un homomorfisme de grups, anomenem:

1. *imatge de f* a $\mathrm{Im}f = f(G_1) \leq G_2$;
2. *nucli de f* a $\mathrm{Ker}f = f^{-1}(1_{G_2}) = \{z \in G_1 \mid f(z) = 1_{G_2}\} \leq G_1$.

Notem que $\mathrm{Ker}f \leq f^{-1}(H)$ per a tot $H \leq G_2$: com que $1_{G_2} \in G$, $\{1_{G_2}\} \subseteq G$, amb la qual cosa $\mathrm{Ker}f = f^{-1}(\{1_{G_2}\}) \subseteq f^{-1}(G)$.

Proposició 2.1.2. *Si $f : G_1 \longrightarrow G_2$ és un homomorfisme de grups, aleshores:*

1. $\mathrm{Ker}f \trianglelefteq G_1$.

2. $\mathrm{Im}f \leq G_2$.

Teorema 2.1.2 (Primer teorema d'isomorfisme de grups)**.** *Siga $f : G_1 \longrightarrow G_2$ un homomorfisme de grups. Aleshores la correspondència:*

$$\begin{array}{rccc} \overline{f}: & G_1/\mathrm{Ker}f & \longrightarrow & \mathrm{Im}f \\ & g\mathrm{Ker}f & \longmapsto & f(g) \end{array}$$

per a cada $g \in G_1$ és un isomorfisme de grups. En particular,

$$G_1/\mathrm{Ker}f \cong \mathrm{Im}f.$$

Definició 2.1.10. Siga $X \subseteq G$, denotem el *subgrup de G generat per X* com $\langle X \rangle$, que és la intersecció de tots els subgrups de G que contenen a X. El subgrup generat per X és el menor subgrup de G que conté X, és a dir, $\langle X \rangle$ queda caracteritzat per les següents propietats:

1. $X \subseteq \langle X \rangle \leq G$,
2. Si $X \subseteq T \leq G$ aleshores $\langle X \rangle \leq T$.

A més, és fàcil veure que:

$$\langle X \rangle = \{x_{i_1}^{\varepsilon_1} \cdots x_{i_n}^{\varepsilon_n} \mid x_{i_j} \in X, \varepsilon_j \in \{-1, 1\}, n \in \mathbb{N}\}.$$

Direm que G està generat per X si $G = \langle X \rangle$.
Direm que G és cíclic si $G = \langle g \rangle$, per a algun $g \in G$.
A continuació, introduïm el concepte de grup lliure sobre un conjunt de generadors.

Definició 2.1.11. Siga X un conjunt arbitrari, una *paraula* w en X, és una seqüència finita d'elements en X (pot ser buida) denotada per $w = x_1 \cdots x_n$, amb $x_i \in X$ per a cada $1 \leq i \leq n$, $x_i \in X$. El nombre n s'anomena longitud de la paraula w, i el denotem per $|w|$. Denotem la paraula buida per λ i tenim que $|\lambda| = 0$.

A més, siga $X^{-1} = \{x^{-1} \in G \mid x \in X\}$, on x^{-1} és només una expressió formal obtinguda de x i -1. Si $x \in X$, els símbols x i x^{-1} s'anomenen *literals en X*. Denotarem per $X^{\pm 1} = X \cup X^{-1}$ al conjunt de tots els literals en X.

Una expressió de la forma

$$w = x_{i_1}^{\varepsilon_1} \cdots x_{i_n}^{\varepsilon_n}$$

on $x_{i_j} \in X$, $\varepsilon_j \in \{1, -1\}$ s'anomena *paraula de grup en X*. Per tant, una paraula de grup en X és només una paraula en l'alfabet $X^{\pm 1}$.

Una paraula de grup en X $w = x_1 \cdots x_n$ es diu reduïda si per a qualsevol $1 \leq i \leq n-1$, es té que $x_i \neq x_{i+1}^{-1}$, és a dir, w no conté cap subparaula de la forma xx^{-1} per qualsevol literal $x \in X^{\pm 1}$. Assumim que la paraula buida és reduïda. Denotem per $F(X)$ al conjunt de paraules reduïdes sobre X. Notem que $F(X)$ té estructura de grup per a la concatenació de paraules i posterior reducció, i té a la paraula buida com a element neutre.

Si $X \subseteq G$ amb G un grup, aleshores cada paraula $w = x_{i_1}^{\varepsilon_1} \cdots x_{i_n}^{\varepsilon_n}$ en X determina un únic element de G que és igual al producte $x_{i_1}^{\varepsilon_1} \cdots x_{i_n}^{\varepsilon_n}$ del elements $x_{i_j}^{\varepsilon_j} \in G$. Per convenció, la paraula buida λ determina l'identitat 1 de G.

Un grup G s'anomena *grup lliure* si hi ha un conjunt generador X de G tal que cada paraula de grup reduïda no buida en X defineix un element no trivial de G. En aquest cas X s'anomena *base lliure* de G i G s'anomena *grup lliure* sobre X o *grup lliurement generat* per X. Se segueix de la definició que cada element de G pot ser definit per una única paraula de grup reduïda en X. A més, paraules reduïdes diferents en X defineixen diferents elements en G. En particular $F(X)$ és un exemple de grup lliure.

Els grups lliures tenen la següent *propietat universal*.

Teorema 2.1.3. *Siga G un grup lliure amb conjunt de generadors $X \subseteq G$ i siga $\varphi : X \longrightarrow H$ una aplicació d'X en un grup H, aleshores existeix un únic homomorfisme de grups $\varphi^{\#} : G \longrightarrow H$ que satisfà l'equació $\varphi^{\#} \circ \mathrm{in}_X = \varphi$, on $\mathrm{in}_X : X \longrightarrow G$ és l'aplicació inclusió d'X en G.*

Corol·lari 2.1.1. *Siga G un grup lliure en X. Aleshores l'aplicació identitat $\mathrm{id}_X : X \longrightarrow X$ es pot extendre a un isomorfisme $G \longrightarrow F(X)$.*

Aquest corol·lari ens permet identificar un grup lliurement generat per X amb el grup $F(X)$ de paraules de grups reduïdes en X. De fet, la construcció d'un grup lliure només depèn del cardinal dels conjunts generadors. A partir d'ara, considerarem $F(X)$ com el grup lliure generat per X.

2.1.1 Presentacions de grups per generadors i relators

La propietat universal dels grups lliures ens permet descriure grups en termes de generadors i relators. Siga G un grup amb un conjunt generador X. Per la propietat universal dels grups lliures, existeix un únic homomorfisme $\psi : F(X) \longrightarrow G$ tal que $\psi(x) = x$ per a tot $x \in X$. Com ψ és sobrejectiva, pel primer teorema d'isomorfisme:

$$G \cong F(X)/\ker(\psi).$$

En aquest cas, $\ker(\psi)$ es veu com al conjunt de relators de G, i la paraula de grup $w \in \ker(\psi)$ s'anomena un relator de G respecte dels generadors X. Si un subconjunt $R \subseteq \ker(\psi)$ genera $\ker(\psi)$ com a un subgrup normal de $F(X)$, aleshores direm que és el conjunt que defineix els relators de G en relació a X. El parell $\langle X|R\rangle$ s'anomena presentació d'un grup G, i determina G de forma única tret d'isomorfisme. La presentació $\langle X|R\rangle$ és finita si ambdós conjunts X i R són finits. Un grup està finitament presentat si té almenys una presentació finita. Les presentacions ens donen un mètode universal per a descriure grups. En particular, els grups finitament presentats admeten descripcions finites, com per exemple:

$$G = \langle x_1, x_2, \cdots, x_n | r_1, r_2, \cdots, r_k \rangle.$$

Per exemple, tots els grups abelians finitament generats són finitaments presentats. Podem trobar més propietats, teoremes i demostracions sobre grups i subgrups en [2].

2.1.2 Problemes algorísmics de presentacions de grups

Els problemes algorísmics de teoria de grups que considerarem en aquesta secció són de dos tipus diferents:

1. Problemes de decisió: són problemes on es dona una propietat $\mathcal{P}$ i un objecte $\mathcal{O}$, i es tracta de comprovar si l'objecte $\mathcal{O}$ té la propietat $\mathcal{P}$ o no.

2. Problemes de recerca: són problemes on es dona una propietat $\mathcal{P}$ i la informació de que hi han objectes amb la propietat $\mathcal{P}$, i es tracta de trobar almenys un objecte particular amb la propietat $\mathcal{P}$.

A continuació, parlarem de diversos algorismes particulars per a problemes de teoria de grups que s'han usat en criptografia.

El problema de la paraula

1. El problema de la paraula *(word problem, WP)* consisteix en que, donada una presentació recursiva d'un grup G i un element $g \in G$, trobar si $g = 1$ o no.

   De la descripció del problema de la paraula, podem veure que consta de dos parts: si $g = 1$ o si no ho és. Anomenem a aquestes parts la "part del sí" i "la part del no" del problema de la paraula, respectivament. Si un grup ve donat per una presentació recursiva en termes de generadors i relators, aleshores la "part del si" té una solució recursiva, gràcies a aquesta proposició.

   Proposició 2.1.3. *[31] Siga $\langle X|R\rangle$ una presentació recursiva d'un grup G. Aleshores, el conjunt de totes les paraules $g \in G$ tal que $g = 1$ és recursivament enumerable.*

2. El problema de la recerca de la paraula *(word search problem, WSP)* consisteix en que, donada una presentació recursiva d'un grup G i un element $g = 1$, trobar una presentació de g com a producte dels conjugats definits pels relators i els seus inversos.

   Notem que el problema de la recerca de la paraula sempre té una solució recursiva ja que podem enumerar recursivament tots els productes dels relators que el defineixen, els seus inversos i els seus conjugats. No obstant això, el nombre de factors que es necessiten en aquest producte per a representar una paraula de longitud n que és igual a 1 en G, pot ser molt elevat comparat amb n. En particular, hi han grups G on el problema de la paraula amb paraules w de longitud n que són iguals a 1 en G són eficientment resolubles, ja que el nombre de factors de qualsevol factorització de w en el producte dels relators que la defineixen, els seus inversos i els seus conjugats,

no està limitat per qualsevol torre d'exponents en n. A més, si en un grup G, el problema de la paraula no és recursivament resoluble, aleshores la longitud d'una prova que verifique que $w = 1$ en G no està limitada per cap funció recursiva de la longitud de w.

El problema de la conjugació

Els següents dos problemes són d'interès per a nosaltres:

1. El problema de la conjugació *(conjugacy problem, CP)* consisteix en que, donada una presentació recursiva d'un grup G i dos elements $g, h \in G$, trobar si hi ha o no un element $x \in G$ tal que $x^{-1}gx = h$.

   De nou, com en el problema de la paraula, el problema de la conjugació consisteix en la part del "sí" i la part del "no", amb la part del "sí" sempre recursiva perquè podem enumerar recursivament tot els conjugats d'un element donat.

2. El problema de la recerca de la conjugació *(conjugacy search problem, CSP)* consisteix en que, donada una presentació recursiva d'un grup G i dos elements conjugats $g, h \in G$, trobar un element particular $x \in G$ tal que $x^{-1}gx = h$.

   Com hem dit, el problema de la recerca de la conjugació té sempre una solució recursiva, perquè podem enumerar recursivament tots els conjugats d'un element donat, però com en el problema de la recerca de la paraula, aquesta solució pot ser extremadament ineficient.

La descomposició i factorització de problemes

Una de les ramificacions naturals del problema de la recerca de la conjugació és la següent: el problema de pertinença. Ara, veurem el següent parell de problemes:

1. El problema de la pertinença (*membership problem, MP*) consisteix en que, donada una presentació recursiva d'un grup G, un subgrup $H \subseteq G$ generat per $h_1, \cdots, h_k$, i un element $g \in G$, trobar si $g \in H$ o no.

   Un altra vegada, el problema de la pertinença consisteix en la part del "sí" i la part del "no", amb la part del "sí" sempre recursiva perquè podem enumerar tots els elements del subgrup (generat per un conjunt finit de generadors) de forma recursiva.

   Notem que el problema de la pertinença també es coneix com "el problema generalitzat de la paraula".

2. El problema de la recerca de la pertinença (*membership search problem, MSP*) consisteix en que, donada una presentació recursiva d'un grup G, un subgrup $H \subseteq G$ generat per $h_1, \cdots, h_k$, i un element $h \in H$, trobar una expressió d'h en termes d'$h_1, \cdots, h_k$.

El problema de l'isomorfisme

1. El problema de l'isomorfisme consisteix en que, donats dos grups finitament presentats G_1 i G_2, trobar si són isomorfs o no.

Notem que el mètode de Tietze, que descriurem més endavant, ens dona una enumeració recursiva dels grups finitament presentats isomorfa a una presentació finita del grup, que implica que la part del "sí"del problema de l'isomorfisme és sempre recursiva.

2.1.3 Reescritura de les presentacions de grups

El mètode de Nielsen [41]

Siga $F(X)$ un grup lliure de rang finit $n \geq 2$, amb un conjunt $X = \{x_1, \cdots, x_n\}$ de generadors lliures. Siga $Y = \{y_1, \cdots, y_m\}$ un conjunt arbitrari finit d'elements del grup $F(X)$. Considerem les següents transformacions elementals, anomenades de Nielsen, que poden ser aplicades a Y:

(N1) y_i és reemplaçat per $y_j y_i$ per a algun $j \neq i$.

(N2) y_i és reemplaçat per y_i^{-1}.

(N3) y_i és reemplaçat per algun y_j, i al mateix temps y_j és reemplaçat per y_i.

(N4) Eliminem els y_i tals que $y_i = 1$.

Se sobreentén que y_j no es canvia si $j \neq i$.

Cada conjunt finit de paraules reduïdes de $F(X)$ pot ser convertit a través d'una seqüència finita de transformacions de Nielsen a un conjunt de Nielsen reduït $U = \{u_1, \cdots, u_k\}$. Per exemple, considerem un conjunt tal que per a v_1, v_2, v_3 de la forma $u_i^{\pm 1}$ es tenen les següents tres condicions:

1. $v_1 \neq 1$;
2. $v_1 v_2 \neq 1$ implica que $|v_1 v_2| \geq |v_1|, |v_2|$;
3. $v_1 v_2 \leq 1$ i $v_2 v_3 \neq 1$ implica que $|v_1 v_2 v_3| > |v_1| - |v_2| + |v_3|$.

És fàcil veure que si $U = \{u_1, u_2, \cdots, u_k\}$ és el conjunt de Nielsen reduït, aleshores el subgrup de $F(X)$ generat per U és lliure amb una base U.

Si ens fixem, algunes de les transformacions $(N1) - (N4)$ són redundants, és a dir, són composicions d'unes altres. La raó que hi ha darrere d'açò l'expliquem a continuació.

Diguem que dos conjunts Y i $\tilde{Y}$ són Nielsen-equivalents si una d'elles es pot obtindre a partir de l'altra aplicant una seqüència de transformacions $(N1) - (N4)$. Està demostrat

per Nielsen [41] que dos conjunts Y i $\tilde{Y}$ generen el mateix subgrup del grup $F(X)$ si i només si són Nielsen-equivalents. Aquest resultat és ara un dels punts centrals de la teoria combinatòria de grups.

Notem, no obstant això, que aquest resultat per si soles no ens dona un algorisme per decidir si Y i $\tilde{Y}$ generen el mateix subgrup de $F(X)$. Per a obtindre l'algorisme, necessitem d'alguna manera definir la complexitat d'un conjunt donat d'elements i a continuació, mostrar que a través d'una seqüència de transformacions de Nielsen $(N1) - (N4)$ podem aconseguir que aquesta complexitat disminuïsca (o al menys que no augmente) en cada pas; i ací és on podríem necessitar les transformacions elementals redundants.

A més, Nielsen també va demostrar que la complexitat d'un conjunt donat $Y = \{y_1, \cdots, y_m\}$ és just la suma de les longituds de les paraules $y_1, \cdots, y_m$. L'algorisme el descriurem a continuació. Primerament, reduïm tots dos conjunts Y i $\tilde{Y}$ a conjunts reduïts de Nielsen. Aquest procediment és finit, ja que la suma de les longituds de les paraules disminueix amb cada pas. Després, resolem el problema de la pertinença per a cada element de Y en el subgrup generat per $\tilde{Y}$ i viceversa.

Per tant, el mètode de Nielsen produeix, en particular, un algorisme per a decidir si, donat un endomorfisme d'un grup lliure de rang finit, és un automorfisme o no.

El mètode de Schreier [31]

Un altre mètode clàssic que tractarem és el de Schreier. Donarem una explicació d'aquest mètode en el cas especial d'un grup lliure; no obstant això, aquest mètode és vàlid per a qualsevol grup arbitrari.

Per a qualsevol subgrup H d'$F(X)$, podem definir una relació d'equivalència $\backsim$ donada per $x \backsim y$ si $x = yh$ per a algun $h \in H$. La equivalència de classes d'aquesta relació d'equivalència són exactament les classes a esquerra mòdul H, i un element x de $F(X)$ està en la classe d'equivalència xH. Per tant, les classes a esquerra mòdul H formen una partició d'$F(X)$.

És també cert que qualsevol dos classes a esquerra mòdul H tenen el mateix cardinal, i en particular, cada classe mòdul H té el mateix cardinal que $eH = H$, on e és l'element identitat. Per això, el cardinal de qualsevol classe a esquerra mòdul H és de l'ordre del cardinal d'H.

Els mateixos resultats són certs també per a les classes a dreta de $F(X)$ i, de fet, podem provar que el conjunt de classes a dreta mòdul H té el mateix cardinal que el conjunt de classes a esquerra mòdul H.

Siga H un subgrup d'$F(X)$. Una funció representant de classe a dreta de $F(X)$ (respecte al generadors x_i) mòdul H és una operació de paraules amb x_i:

$$w(x_1, x_2, \cdots) \longmapsto \bar{w}(x_1, x_2, \cdots),$$

on les $\bar{w}(x_1, x_2, \cdots)$ formen el sistema representant de la classe a dreta de $F(X)$ mòdul

H, el qual conté a la paraula buida, i on $\bar{w}(x_1, x_2, \cdots)$ és la representant de classe de $w(x_1, x_2, \cdots)$. Aleshores tenim:

Teorema 2.1.4. *Si $w \longmapsto \bar{w}$ és una funció de classe a dreta de $F(X)$ mòdul H, aleshores H està generat per les paraules:*

$$ux_i\overline{ux_i}^{-1},$$

on u és un representant arbitrari i x_i és un generador d'$F(X)$.

Açò també implica, per exemple, si $F(X)$ és finitament generat i H és un subgrup amb índex finit, aleshores H és finitament generat.

A més, una funció de Schreier a dreta és una que per a qualsevol segment inicial d'un representant torna a ser un representant. El sistema de representants s'anomena sistema de Schreier. Es pot provar que sempre hi ha algun sistema de Schreier de representants de $F(X)$ mòdul H. També hi ha un sistema de Schreier minimal, és a dir, un sistema de Schreier en el qual cada representant té una longitud que no excedeix la longitud de la paraula que representa. No tots els sistemes de Schreier són minimals.

Exemple 2.1.1. *Siga $F(a, b)$ el grup lliure respecte a i b, i siga H el subgrup normal de $F(a, b)$ generat per a^2, b^2 i $aba^{-1}b^{-1}$. Aleshores $F(a, b)/H$ té quatre classes. El sistema de representants $\{1, a, b, ab\}$ és un sistema de Schreier, ja que el sistema representant és $\{1, a, b, a^{-1}, b^{-1}\}$. El sistema de representants $\{1, a, b, a^{-1}b^{-1}\}$ no és de Schreier, ja que per al segment inicial a^{-1} d'$a^{-1}b^{-1}$ no és representant.*

Ara, utilitzant el procés de reescritura de Reidemeister-Schreier*, podem obtindre una presentació (per generadors i relators) d'H. Açò implica, entre altres coses, el següent resultat important:*

Teorema 2.1.5 (Nielsen-Schreier)**.** *[31]. Cada subgrup no trivial H d'un grup lliure $F(X)$ és lliure.*

Podem obtindre de manera efectiva un conjunt d'elements generadors lliures d'H, és a dir, aquells $ux_i\bar{u}\bar{x}_i{}^{-1}$ tals que ux_i no és lliurement igual a un representant. Schreier va obtenir aquest conjunt de generadors lliures per a H a l'any 1927. En 1921, Nielsen, utilitzant un mètode molt diferent, va construir un conjunt de generadors lliures d'H si $F(X)$ era finitament generat. Els generadors de Nielsen són precisament aquells generadors Schreier que s'obtenen quan s'utilitza un sistema Schreier mínim.

El mètode de Tietze [19]

Intentant resoldre un dels majors problemes de teoria combinatòria de grups, el problema de l'isomorfisme, Tietze va introduir transformacions elementals que conserven l'isomorfisme i aquestes poden ser aplicades a grups presentats per generadors i relators.

Siga $G = \langle X|R\rangle$ una presentació d'un grup G. Les transformacions elementals són dels tipus següents.

(T1) *Introduïm un nou generador*: reemplacem $\langle x_1, x_2 \cdots \mid r_1, r_2 \cdots\rangle$ per $\langle y, x_1, x_2, \cdots \mid ys^{-1}, r_1, r_2, \cdots\rangle$, on $s = s(x_1, x_2, \cdots)$ és un element arbitrari en els generadors $x_1, x_2, \cdots$.

(T2) *Cancel·lem un generador*: (aquest és una conversió de (T1)): si tenim una presentació de la forma $\langle y, x_1, x_2 \cdots \mid q, r_1, r_2 \cdots\rangle$, on q és de la forma de ys^{-1}, i $s, r_1, r_2, \cdots$ són un grup generat per $x_1, x_2, \cdots$, la cancel·lem per $\langle x_1, x_2 \cdots \mid r_1, r_2 \cdots\rangle$.

(T3) *Apliquem un automorfisme*: apliquem un automorfisme del grup lliure generat per $x_1, x_2, \cdots$ a tots els relators $r_1, r_2, \cdots$.

(T4) *Canviant els relators que defineixen al grup*: reemplacem el conjunt $r_1, r_2, \cdots$ que defineixen al grup per un altre conjunt $r'_1, r'_2, \cdots$ amb la mateixa clausura normal. Açò significa, que cadascun dels $r'_1, r'_2, \cdots$ hauria de pertànyer al subgrup normal generat per $r_1, r_2, \cdots$, i viceversa.

Aleshores, tenim el següent resultat com a conseqüència de Tietze:

Teorema 2.1.6. *[31]. Dos grups $\langle x_1, x_2, \cdots \mid r_1, r_2, \cdots\rangle$ i $\langle x_1, x_2, \cdots \mid s_1, s_2, \cdots\rangle$ són isomorfs si i només si un d'ells es pot obtindre de les presentacions de l'altre per una seqüència de transformacions (T1)-(T4).*

Per a la majoria de propòsits pràctics, els grups considerats es presenten de manera finita, i en aquest cas existeix una seqüència finita de transformacions (T1)–(T4) que porta a una de les presentacions a l'altra. Així i tot, el Teorema 2.1.6 no ofereix cap procediment constructiu per decidir en un nombre finit de passos si una presentació finita es pot obtenir d'una altra mitjançant transformacions de Tietze perquè, per exemple, no hi ha cap indicació de com seleccionar l'element s en una transformació (T1). Així, el Teorema 2.1.6 no dóna solució al problema d'isomorfisme.

No obstant això, el mètode de Tietze ofereix una manera fàcil i pràctica de construir exemples exòtics de grups isomorfs que han ajudat a refutar diverses conjectures en la teoria combinatòria de grups. Per una raó similar, les transformacions de Tietze poden ser útils per difondre presentacions de grups en alguns protocols criptogràfics.

2.2 Criptografia de clau pública

En el cor de qualsevol criptografia de clau pública primitiva hi ha alguna pràctica irreversible d'algun procés, usualment coneguda com a *trampa* o *funció de sentit únic*. Per exemple, el sistema criptogràfic RSA (Rivest, Shamir y Adleman) [7] utilitza el fet de que no és difícil computar el producte de dos números enters considerablement grans, però, en canvi,

factoritzar un nombre gran com a producte d'enters pot arribar a ser molt complicat. Per tant, aquest algorisme serà segur mentre no es coneguen formes ràpides de descompondre un nombre gran en productes de números primers.

Un altre exemple, que pot ser siga fins i tot més intuïtiu, és el de la funció $f(x) = x^2$. Aquesta és prou fàcil de tractar en alguns grups o semigrups, però, en canvi, la seua inversa $\sqrt{x}$ és molt menys amigable. Aquest fet va ser explotat en el sistema criptogràfic de Rabin [38].

A dia de hui, la tendència de la criptografia de clau pública és anar ampliant-se, en lloc d'aprofundir, ja que les aplicacions d'aquesta inclouen signatures digitals, autenticació de protocols, etc. Nosaltres, ens centrarem en la criptografia *primitiva*, i en particular, en els camins que hi ha entre dos parts (tradicionalment anomenats Alícia i Bob) per a establir una clau privada comuna sense cap acord previ, i el nostre objectiu serà evitar que, ni les claus ni el missatge que fem arribar d'una part a l'altra, puguen ser interceptats de cap manera (Eva serà l'espia que pretén interceptar açò). Anomenem a aquests processos *protocols d'establiment de claus*. Observem que una vegada hem aconseguit establir la clau privada, Alícia i Bob tenen el comandament d'una criptografia simètrica, que té molts avantatges; en particular, el xifrat i desxifrat pot ser molt eficient, ja que les dos parts tenen un secret compartit.

En la següent secció, mostrarem una manera universal de trobar un xifratge basat en una clau privada comuna. Com qualsevol procés universal, està lluny de ser perfecte, però està bé conèixer-lo.

2.2.1 De l'establiment de la clau al xifrat

Suposem que Alícia i Bob comparteixen una clau privada K, la qual és un element d'un conjunt $\mathcal{S}$ (usualment anomenat *espai de les claus*).

Considerem $H : \mathcal{S} \to \{0,1\}^n$, una funció qualsevol del conjunt $\mathcal{S}$ al conjunt de les cadenes de bits de longitud n. És raonable tindre una n suficientment gran, és a dir, almenys $log_2|\mathcal{S}|$ si $\mathcal{S}$ és finit, o qualsevol n que l'ordinador puga permetre's computar si $\mathcal{S}$ és infinit. Aquestes funcions solen anomenar-se *funcions de resum* (hash function). En unes altres situacions, les funcions de resum s'utilitzen com a representacions compactes, o petjades digitals de dades i per a garantir la integritat del missatge. Deixem la descripció d'alguna d'aquestes funcions de resum per al capítol 3. Continuem ara amb la descripció del protocol.

Xifrat: Bob xifra el seu missatge $m \in \{0,1\}^n$ com:

$$E(m) = m \oplus H(K),$$

on $\oplus$ és la suma en mòdul 2.

Desxifrat: Alícia calcula:

$$(m \oplus H(K)) \oplus H(K) = m \oplus (H(K) \oplus H(K)) = m,$$

i per tant, recupera el missatge m.

Notem que aquest xifrat té un *factor d'expansió* igual a 1, és a dir, el xifrat del missatge és tan llarg com el propi missatge. Açò és molt útil, especialment comparat amb altres tipus de xifrat, on el factor d'expansió és de l'ordre de les centenes, però aquest és el preu que es paga per garantir la seguretat davant de la superioritat computacional de l'adversari.

2.2.2 L'establiment de claus de Diffie-Hellman

L'intercanvi de claus de Diffie-Hellman [25] va ser inventat a 1976 i va ser el primer mètode pràctic per a establir una clau privada compartida davant dels canals de comunicació no protegits. Tot i que aquest procés és conegut com Diffie-Hellman, Merkle [39] assegura que aquest sistema va ser descrit en un article escrit per Diffie i ell mateix, i per tant, hauria d'anomenar-se intercanvi de claus de Diffie-Hellman-Merkle. A més, la patent d'EUA (ja caducada), descriu l'algorisme i acredita al Hellman, Diffie i Merkle com a inventors.

La implementació més simple i original del protocol utilitza el grup multiplicatiu dels enters mòdul p, on p és un nombre primer i g és una arrel primitiva mòdul p. Una descripció més general del protocol utilitza un grup cíclic finit arbitrari. Passem a descriure'l a continuació.

1. Alícia i Bob acorden un grup cíclic finit G i un element generador $g \in G$. Escriurem el grup G de forma multiplicativa.
2. Alícia elegeix un nombre natural aleatori a i li envia g^a a Bob.
3. Bob elegeix un nombre natural aleatori b i li envia g^b a Alícia.
4. Alícia calcula $K_A = (g^b)^a = g^{ba}$.
5. Bob calcula $K_B = (g^a)^b = g^{ab}$.

Com que $ab = ba$ (perquè $\mathbb{Z}$ és commutatiu), Alícia i Bob tenen el mateix element del grup $K = K_A = K_B$, que els pot servir com a clau privada compartida.

El protocol està considerat segur [31] contra els espies si G i g són elegits de forma adequada. L'espia, *Eva*, ha de resoldre el *Problema de Diffie-Hellman* (recuperar g^{ab} a partir de g^a i g^b) per a obtindre la clau privada compartida. Aquest problema es considera difícil de resoldre per a una bona elecció dels paràmetres.

Un algorisme eficient per a resoldre el *problema del logaritme discret* (és a dir recuperar a de g i g^a), ens valdria per a solucionar el problema de Diffie-Hellman, i faria aquest i molts altres sistemes criptogràfics de clau pública insegurs. No obstant això, no se sap amb certesa si el problema del logaritme discret és equivalent al problema de Diffie-Hellman.

Notem que està el mètode de la "força bruta" per a resoldre el problema del logaritme discret: l'espia Eva pot anar agafant els naturals des d'1 fins a n, un a un, i computant g^n, per a veure així si coincideix amb l'element que s'han transmès. Açò requereix $O(|g|)$

multiplicacions, on $|g|$ és l'ordre de g. Donat que en les implementacions pràctiques $|g|$ és normalment al voltant de 10^{300}, aquest mètode és computacionalment inviable.

Açò planteja una qüestió sobre la eficiència computacional per a ambdues parts: aparentment, pareix que les dues parts han d'executar $O(|g|)$ per a calcular g^a o g^b. No obstant això. hi ha una forma més ràpida de calcular g^a per a un a particular, utilitzant l'algorisme d'"exponenciació binària", basat en la forma binària d'a. Per exemple, $g^{22} = (((g^2)^2)^2)^2 \cdot g^2$. Per tant, per a calcular g^a, necessitem $O(log_2 a)$ multiplicacions, la qual cosa és més viable.

2.2.3 El criptosistema de ElGamal

El criptosistema de ElGamal [16] és un criptosistema de clau pública que està basat en l'establiment de claus de Diffie-Hellman [25]. L'algorisme de la signatura digital és una variant de l'esquema de la signatura d'ElGamal, el qual no deu de ser confós amb el protocol del xifrat d'ElGamal, que descriurem a continuació.

1. Alícia i Bob acorden un grup cíclic finit G i un element generador g en G.
2. Alícia (la receptora) elegeix un nombre natural aleatori a i publica $c = g^a$.
3. Bob (l'emissor), que vol enviar un missatge $m \in G$ (anomenat "text simple" en llenguatge criptogràfic) a Alícia, elegeix un nombre natural aleatori b i li envia dos elements, mc^b i g^b a Alícia. Observem que $c^b = g^{ab}$.
4. Alícia recupera $m = (mc^b)((g^b)^a)^{-1}$.

Una característica important del xifrat d'ElGamal és que és *probabilístic*, és a dir, un únic text simple pot ser xifrat en molts possibles textos xifrats.

També cal destacar que el xifrat d'ElGamal té factor d'expansió mitjà de 2. Notem que aquest factor d'expansió és major que el del mecanisme descrit en l'Apartat 2.2.1, i a més, a nivell de computació el nombre d'operacions necessàries no varia molt, per tant, vol dir que aquest criptosistema és més segur que l'altre.

2.2.4 Autenticació

L'autenticació és el procés d'intentar verificar la identitat digital de l'emissor d'una comunicació. D'interès particular en la criptografia pública són les proves **d'identitat de coneixement zero** *(zero-knowledge proofs, ZKP)*. Açò vol dir que si l'identitat és verdadera, cap verificador maliciós aprèn un altra cosa diferent a aquest fet. Per tant, una part (el provador), vol demostrar la seua identitat a una segona part (el verificador) a través d'alguna informació secreta (una clau privada), però no vol que ningú sàpiga el seu secret.

Molts protocols d'establiment de claus poden ser lleugerament modificats per a convertir-se en un protocol d'autenticació. Mostrarem açò en l'exemple del protocol d'establiment de claus de Diffie-Hellman, presentat abans.

Suposem que Alícia és la provadora i Bob és el verificador. Per tant Alícia vol convèncer a Bob de que ella sap un secret sense revelar el propi secret. A continuació, descriurem el procés.

1. Alícia publica un grup cíclic finit G i un element generador $g \in G$. Després elegeix un nombre natural a i publica g^a.
2. Bob elegeix un nombre natural b i envia un *repte* g^b a Alícia.
3. Alícia contesta amb una prova $P = (g^b)^a = g^{ba}$.
4. Bob verifica: $(g^a)^b = P$?

Observem que aquest protocol és quasi idèntic al protocol d'establiment de claus de Diffie-Hellman.

Resum

Així, en aquest capítol hem fet una introducció sobre teoria de grups, tractant conceptes que tenen especial importància en criptografia com els grups lliures i les paraules. Després, hem vist la propietat universal que tenen els grups lliures, la qual ens permet descriure aquest grups en termes de generadors i relators. Basant-nos en aquesta manera de presentar els grups, hem tractat els problemes algorísmics de presentacions de grups i alguns mètodes que ajuden a la seua resolució. Per a concloure el capítol, hem introduït el concepte de criptografia de clau pública i hem descrit alguns protocols per a establir una clau privada, tractant també el concepte d'autenticació.

Capítol 3

Funcions de resum

Les funcions de resum *(hash function)* són les funcions matemàtiques més utilitzades en criptografia per a implementar protocols de seguretat. Una funció de resum converteix el valor d'entrada de qualsevol grandària arbitraria en un valor d'una grandària fixa. Per tant, l'entrada pot ser de qualsevol longitud, però l'eixida és d'una longitud fixa. L'eixida generada sol anomenar-se valor de resum *(hash value)*.

Cal observar que un xifrat és una funció bidireccional, és a dir, les dades xifrades han de desxifrar-se utilitzant una clau, per la qual cosa són reversibles; en canvi, les funcions de resum són funcions unidireccionals, és a dir, no es poden invertir.

Un dels usos més comuns de les funcions de resum, és la comprovació de contrasenyes. Quan l'usuari introdueix la contrasenya, el resum de la contrasenya es genera i es compara amb el resum de la base de dades. Si ambdós són iguals, l'usuari pot iniciar sessió, en cas contrari, l'usuari ha d'introduir la contrasenya de nou.

A més, les funcions de resum s'utilitzen àmpliament en els següents camps: criptomonedes, comprovació de la integritat de les dades i els arxius i en la signatura digital.

En aquest capítol, introduirem les funcions de resum i explicarem unes funcions de resum simples. A continuació, ens centrarem en les funcions de resum més conegudes, i acabarem el capítol analitzant l'algorisme del SHA-256. Podem trobar informació més detallada en les fonts [28] i [21], d'on prové gran part de la informació continguda en aquest capítol.

3.1 Funcions de resum

Una funció de resum és una funció matemàtica que converteix un valor d'entrada numèric en un altre valor numèric comprimit. L'entrada a la funció de resum és de longitud arbitrària, però la d'eixida sempre és de longitud fixa.

Els valors retornats per una funció de resum s'anomenen valors de resum.

Una funció de resum ideal assigna les claus als enters de forma aleatòria, de manera

que els valors de resum es distribueixen uniformement encara que hi hagen regularitats en les dades d'entrada.

Aquest procés es pot dividir en dos passos:

- Assignar la clau a un enter.
- Assignar el nombre enter a un bloc.

Realment, podem aplicar funcions de resum a qualsevol dada, encara que no siga un enter, com poden ser estructures de registre complexes. En el nostre cas, assumirem que les nostres claus són enters, coses que podem tractar com a enters (com caràcters o punters), o seqüències unidimensionals composades per eixe tipus de dades (llista d'enters, cadenes de caràcters, etc.).

3.2 Funcions de resum simples

Les següents funcions assignen una sola clau entera (k) a un valor enter xicotet $h(k)$. Anomenarem m a la grandària de la taula de resum (nombre de valors).

Mètode de divisió (Cormen): [11] S'elegeix un nombre primer que no estiga prop d'una potència de 2. Definim $h(k) = k \ mod \ m$. Aquesta funció de resum és molt simple, però no funciona bé per a molts tipus de patrons en les dades d'entrada.

Implementem aquesta funció en Python:

```
1 def Metdiv(k,m):
2     res=k%m
3 return res
```

Codi 3.1: Mètode de divisió en Python.

I obtenim alguns exemples:

```
1 Metdiv(23456,191)
2 Out[1]: 154
3
4 Metdiv(20,13)
5 Out[2]: 7
6
7 Metdiv(23309,191)
8 Out[3]: 7
```

Codi 3.2: Eixides del mètode de divisió en Python.

Observem que en les eixides 2 i 3 obtenim el mateix resum encara que les dades d'entrada són diferents.

Variant Knuth en la divisió: En aquest cas la funció de resum ve donada per: $h(k) = k(k+3) \ mod \ m$. És una variació de la divisió que funciona millor. Implementem-la:

```
def Knuth(k,m):
    res=(k*(k+3))%m
return res
```

Codi 3.3: Variant Knuth en la divisió en Python.

I obtenim alguns exemples:

```
Knuth(23456,191)
Out[4]: 112

Knuth(23309,191)
Out[5]: 70

Knuth(20,13)
Out[6]: 5
```

Codi 3.4: Exemples de la Variant Knuth en la divisió en Python.

En aquesta és més difícil obtindre la mateixa eixida, però és possible.

Mètode de multiplicació (Cormen): [11] S'elegeix una m que siga potència de 2. Siga A un nombre real d'aspecte aleatori. Knuth suggereix $A = 0.5(\sqrt{5} - 1)$. A continuació, la funció es descriu de la següent manera:

$s = k \cdot A$; $x =$ part fraccionària de s; $h(k) = \text{floor}(m \cdot x)$

Implementant-lo en Python obtenim:

```
import math
def Metmult(k,A,m):
    s=k*A
    x=math.modf(s)
    res=math.floor(m*x[0])
return res
```

Codi 3.5: Mètode de multiplicació en Python.

I ací tenim alguns exemples:

```
A=0.5*(math.sqrt(5)-1)
m=pow(2,8)

Metmult(199234, A, m)
Out[7]: 98

Metmult(123546,A,m)
Out[8]: 160
```

Codi 3.6: Exemples del mètode de multiplicació en Python.

Aquesta és la funció més complexa de les que hem vist fins ara.

3.3 Funcions de resum aplicades a seqüències de caràcters

Les funcions de resum d'aquesta secció prendran una seqüència d'enters $k = (k_1, \ldots, k_n)$ i produiran un valor enter $h(k)$, com als exemples anteriors. D'igual forma, m serà de nou la grandària de la taula de resum, que ha de ser un nombre primer. La seqüència d'enters podria ser una llista d'enters o una matriu de caràcters (una cadena). Quan utilitzem aquests algorismes, les entrades k_i deuen ser sense signe, ja que d'altra manera podem obtindre comportaments estranys.

Per a cada un d'aquests algorismes, siga h el valor d'eixida, establim h en 0, i el fem baixar per la seqüència d'enters, afegint els enters un a un a h. Els algorismes es diferencien exactament en com combinar cada k_i amb h. El valor de retorn final es $h \ mod \ m$.

Variant CRC: [30] Consisteix en fer un desplaçament circular cap a l'esquerra de 5 bits d'h, i a continuació apliquem XOR en $k_i \ \forall 1 \leq i \leq n$, on $k = (k_1, \ldots, k_n)$ és la seqüència d'enters que hem pres. XOR és un operador binari que expliquem en més detall a continuació:

En primer lloc, vegem els operadors que utilitza aquesta funció:

- & : és l'operador AND bit a bit i el que fa és comparar cada bit del primer operand amb el bit corresponent del segon operand. Si ambdós bits són 1, el bit del resultat s'estableix en 1. Pel contrari, el bit del resultat corresponent serà 0.
- << : és un operador de desplaçament bit a bit que mou els bits d'una expressió de tipus enter o d'una enumeració a la dreta.
- >> : és un operador de desplaçament bit a bit que mou els bits d'una expressió de tipus enter o d'una enumeració a l'esquerra.
- ^: és l'operador XOR bit a bit i el que fa és comparar cada bit del primer operand amb el bit corresponent del segon operand. Si el bit d'un dels operands és 0 i el bit de l'altre operand és 1, el bit del resultat corresponent serà 1. En cas contrari, el bit del resultat corresponent serà 0.

Ací tenim l'implementació en Python de la variant CRC:

```
1 highorder = h&0xf8000000
2 h = h<<5
3 h = h^(highorder >> 27)
4 h = h^k(i)
```

Codi 3.7: Variant CRC en Python.

Ara expliquem detalladament en que consisteix el codi de la variant CRC.

En la línia [1], extraguem els 5 bits d'ordre més alt d'h. Açò ho fem perquè en C++, un caràcter és una variable entera de 8 bits. En el codi ASCII només s'usen 7 d'aquests 8

bits i d'aquests 7 bits, els caràcters que solem usar (alfabètics i numèrics), només utilitzen els 6 bits; i el primer d'aquests 6 bits principalment ens diu si és majúscula o minúscula, la qual cosa és relativament insignificant. Per tant, en aquest algorisme i en els següents ens centrarem en conservar la major informació possible dels 5 últims bits de cada nombre, i farem ús almenys dels 3 primers d'aquests 5. Aleshores, per a extraure els 5 bits d'ordre més alt, prenem 0xf8000000, que és la representació hexadecimal del nombre de 32 bits que té un 1 en els primers 5 bits i un 0 en els altres. Per això, en aplicar l'operador & (h & 0xf8000000), el que fem és extraure els 5 bits d'ordre més alt d'h.

En la línia [2], desplacem h 5 bits a la dreta.

En la línia [3], desplacem els 5 bits d'ordre més alt als 5 d'ordre més baix; ja que com estem tractant amb 32 bits, volem que l'últim d'aquests 5 bits ocupe la posició 32, per tant el primer haurà d'ocupar la 27 per a obtindre el que volem. I a continuació, apliquem l'operador ^, que és l'operador XOR bit a bit.

En la línia [4] apliquem l'operador XOR bit a bit a cada k_i $\forall\, 1 \leq i \leq n$, on $k = (k_1, \ldots, k_n)$ és la seqüència d'enters que hem pres.

Funció de resum PJW: [21] Consisteix primerament en un desplaçament circular a l'esquerra de 4 bits d'h. A continuació, se li suma k_i i es mouen els 4 últims bits d'h al principi. És a dir:

```
1 h= (h << 4) + k(i)
2 g= h & 0xf0000000
3 if (g!= 0)
4    h = h^(g >> 24)
5    h = h^g
```

Codi 3.8: Hash PJW en Python.

Els operadors que utilitzem són els mateixos que abans, per tant, expliquem directament el codi.

Suposem que els primers 4 bits d'h són 0.

En la línia [1] fem un desplaçament de 4 bits a l'esquerra d'h i afegim k_i.

En la línia [2], considerem 0xf000000 que és la representació hexadecimal del nombre de 32 bits que té els primers 4 bits igual a 1 i la resta 0, i apliquem l'operador AND, és a dir obtenim els 4 bits d'ordre més alt d'h.

En la línia [3], si la g és diferent a 0, és a dir, si els 4 bits d'ordre més alt d'h no són 0, aleshores fem un desplaçament de 24 bits a l'esquerra i els col·loquem al final d'h. A més, apliquem l'operador XOR, i observem que els 4 primers bits d'h són 0 de nou.

PJW i la variant CRC funcionen bé i en realitat, no hi ha moltes diferències entre elles. Però, la variant CRC és lleugerament millor ja que:

- CRC utilitza 32 bits, mentre que PJW utilitza només 24 bits. Probablement, no siga un problema important, ja que el valor final m serà molt menor que qualsevol dels dos.

- Un desplaçament de 5 bits és millor que un de 4, però així i tot els desplaçaments de 3, 4 i 5 funcionen bé normalment.

- Combinar valors amb XOR és millor que afegir-los. No obstant això, la diferència és molt lleugera.

3.4 Funcions de resum més utilitzades

En els apartats anteriors, hem tractat algunes funcions de resum molt senzilles per a entendre el concepte. En realitat, les funcions que s'utilitzen són molt més complexes. A continuació, esmentem les funcions de resum que més utilitzades en l'actualitat.

MD [22]: Són les sigles de Message Digest. Els més comuns són MD2, MD4, MD5 i MD6. Es tracta d'una funció de resum de 128 bits, i els seu ús principal és l'autenticació de missatges, així com la verificació de contingut i de firmes digitals. El que fa aquesta és verificar que un arxiu enviat coincidisca amb el rebut per la persona destinatària. En aplicar la funció de resum MD5 a la paraula "hola", obtenim:

```
1 Out: 4d186321c1a7f0f354b297e8914ab240
```

Codi 3.9: Exemple de la funció de resum MD5 amb paràmetre d'entrada "hola".

SHA [17]: Són les sigles de Secure Hash Algorithm. S'utilitzen SHA-0, SHA-1, SHA-2 i SHA-3. En particular, SHA-224, SHA-256, SHA-384, i SHA-512 són variants de la família SHA-2. Aquest algorisme és necessari en totes les signatures i certificats digitals relacionats amb les connexions SSL/TLS. A més, s'utilitza en les contrasenyes, ja que d'aquesta manera el servidor únicament ha de recordar resums i no tota la contrasenya. Així, si la base de dades rep un atac, només conté resums, i no hi ha forma d'accedir directament a les contrasenyes. Més endavant, explicarem en detall el codi de l'algorisme SHA-256. En aplicar la funció de resum SHA-256 a la paraula "hola", obtenim:

```
1 Out:b221d9dbb083a7f33428d7c2a3c3198ae925614d70210e28716ccaa7cd4ddb79
```

Codi 3.10: Exemple de la funció SHA-256 amb paràmetre d'entrada "hola".

RIPEMD [49]: Són les sigles de RACE Integrity Primitives Evaluation Message Digest. Les més utilitzades són RIPEMD, RIPEMD-128, i RIPEMD-160. També existeixen les versions de 256 i 320 bits d'aquest algorisme. RIPEMD consta essencialment de dues versions paral·leles d'MD4, amb algunes millores en els canvis i l'ordre de les paraules del missatge; les dues instàncies paral·leles només difereixen en les constants. En aplicar la funció de resum RIPEMD-128 a la paraula "hola", obtenim:

```
Out:9a553a01492a699dc10c7a9d429a2d7d
```

Codi 3.11: Exemple de la funció de resum RIPEMD-128 amb el paràmetre d'entrada "hola".

Whirlpool [4]: És una funció de resum de 512 bits. WHIRLPOOL-0, WHIRLPOOL-T, i WHIRLPOOL són 3 versions d'aquest algorisme. L'algorisme pren com a entrada un missatge amb una longitud màxima inferior a 256 bits i produeix com a sortida un resum de missatges de 512 bits. En aplicar la funció de resum WHIRLPOOL a la paraula "hola", obtenim:

```
Out:1AE2A8A2336D19EDDE472C082CD6B948EE97538851B0EE2FEBC8541240DE711
216CCD46ECB1A603C15D9376A68FCC36BE465EDA54BCAD14BDA3F1C75D239DD42
```

Codi 3.12: Exemple de la funció de resum WHIRPOOL amb paràmetre d'entrada "hola".

3.5 Propietats de les funcions de resum

Ja hem vist algunes funcions de resum, però no sabem quina és millor que altra. Una funció de resum ideal, deuria tindre les següents propietats per a ser efectiva als atacs:

- **Resistència de la pre-imatge:** significa que la funció de resum no pot ser revertida. En paraules simples, si en donar-li el valor d'entrada A a una funció de resum produeix un valor C, deuria ser molt difícil trobar un valor d'entrada B que produïsca el mateix valor C. Aquesta propietat fa impossible que un atacant que té el valor de resum trobe el valor d'entrada.

- **Resistència a la col·lisió:** significa que hauria de ser molt complex trobar dues entrades diferents de qualsevol longitud que produïsquen el mateix resum. En paraules simples, per a una funció de resum donada, és molt complicat trobar dues entrades A i B tals que $h(A) = h(B)$. Aquesta propietat és la que dificulta que l'atacant trobe dos valors d'entrada que generen la mateixa funció de resum.

3.6 Algorisme SHA-256: Codi

SHA-256 és una de les funcions de resum que han substituït a la SHA-1, i és una de les funcions de resum més potents a dia de hui. Per això, anem a analitzar-la de forma més profunda. És una clau de 256 bits.

A continuació, tractarem una implementació amb el llenguatge Python. El script està orientat per a aplicar-lo a missatges de text, i no a dades binàries. La font principal on hem consultat la informació per a aquest capítol és [15].

Primerament, parlarem dels diferents tipus de dades i de les seues representacions, ja que tots els algorismes SHA treballen a nivell de bit, per tant, és molt important entendre els tipus de dades més rellevants.

A continuació, anem a veure com seria el codi d'algunes funcions per a canviar el tipus de dada d'una entrada, per exemple, passar l'entrada de dades de bits a hexadecimal.

3.6.1 Tipus de dades i representacions

De caràcter a bits

Si agafem una cadena de caràcters com a entrada, cada caràcter necessita ser traduït a la seua corresponent representació numèrica (tant ASCII com *Unicode* ens serveixen). Per sort, Python té una funció ja integrada (`ord()`) que fa el que estem buscant. Per exemple, `ord('F')` correspon a l'enter 70.

La funció que té Python incorporada per a convertir un enter a binari també ens servirà. No obstant això, utilitzant aquesta funció alteraríem un poc el resultat, ja que nosaltres necessitem per a la representació de cada caràcter una longitud de bits estandarditzada (que serà de 8 bits, és a dir un byte) i a més, volem utilitzar llistes de zeros i uns en lloc d'una cadena binària de Python. Per això, serà necessari fer-li alguna xicoteta modificació a aquesta funció que té Python integrada.

Observem que obtenim en implementar-la:

```
print(bin(70))
Out: 0b1000110
```

Codi 3.13: Exemple del conversor a binari incorporat en Python.

Ací, '0b' indica a Python que és una cadena binària. Per tant, hem de tallar aquest indicador, i així, ens quedaran 7 bits. Com volem que siga de 8 bits de longitud, haurem d'afegir un 0 al començament, i d'aquesta manera, ja podrem convertir cada caràcter en un enter. La següent funció realitza tot el procediment que hem explicat i si li donem una cadena de caràcters (o un caràcter), ens retorna una llista de zeros i uns.

```
def translate(message):
    #caràcters de la cadena a valors únics
    charcodes = [ord(c) for c in message]
    #valors únics a cadenes de 8 bit (eliminant l'indicador binari)
    bytes = []
    for char in charcodes:
        bytes.append(bin(char)[2:].zfill(8))
    #cadenes de 8 bits a una llista de bits formada per enters
    bits = []
    for byte in bytes:
        for bit in byte:
            bits.append(int(bit))
```

```
13 return bits
```

Codi 3.14: Codi en Python per a convertir caràcters a la seua representació binària en forma de llista.

En la línia [1] definim la funció `translate` que pren un paràmetre `message`.

En la línia [3] creem una llista, la qual anomenem `charchodes`. Aquesta llista itera sobre cada caràcter `c` en el `message` i aplica la funció `ord()` per a obtenir el valor *Unicode* de cada caràcter. Els valors *Unicode* resultants els guardem en la llista `charchodes`.

En les línies [5], [6] i [7] convertim cada valor *Unicode* de `charcodes` en una representació binària de 8 bits de longitud i els emmagatzenem en la variable `bytes`. Per a cada `char` en `charcodes`, apliquem la funció `bin()` per a convertir el enter `char` en la seua representació binària. Per a eliminar els dos primers caràcters de la cadena binària (els quals són l'indicador binari de Python 0b), utilitzarem la instrucció `[2:]`. Amb la funció `zfill(8)`, omplim la cadena binària amb zeros a l'inici per assegurar-nos que té una longitud de 8 caràcters.

De la línia [9] fins la [12], convertim la llista de cadenes binàries de 8 bits en `bytes` en una llista de bits individuals, representats com a nombres enters (0 o 1). Iterem sobre cada `byte` en `bytes` i després iterem sobre cada `bit` de la cadena `byte`. La funció `int()` s'utilitza per convertir cada `bit` d'una cadena a un enter, i els enters resultants s'afegeixen a la llista de `bits`.

En la línia [13], retornem la llista `bits` com a resultat de la funció `translate`.

En resum, aquest codi defineix la funció `translate`, la qual pren `message` com a entrada i converteix cada caràcter de `message` en una representació binària de 8 bits. A continuació converteix cada bit en un enter (0,1). Els bits resultants es tornen en forma de llista, i per tant, aconseguim convertir caràcters a la seua representació binària.

Ara posarem un exemple de l'implementació d'aquest codi. Encara que més endavant ens interessarà una llista de llistes de 8 bits, ara anem a obtindre només una llista unidimensional.

Per exemple:

```
1 print(translate('HOLA'))
2 Out: [0,1,0,0,1,0,0,0,0,1,0,0,1,1,1,1,0,1,0,0,1,1,0,0,0,1,0,0,0,0,0,1]
```

Codi 3.15: Implementació d'un exemple en Python.

I efectivament, en traduir la cadena "HOLA", obtenim 8 bits corresponent a cada caràcter, és a dir, un conjunt de 32 bits, que és el que volíem.

De Bits a Hexadecimal

És molt comú que el valor de resum es presente en notació hexadecimal. Ací està on el SHA-256 obté el seu nom, de la longitud de la seua eixida en bits, que és 256. Més a fons,

realment obtenim 8 valors en aplicar-lo, de 32 bits cadascú. I com que operem a nivell de bit, també necessitem una funció que converteix de Base 2 a Base 16.

Primerament, convertim la llista en una cadena. Per a això, utilitzem la funció `.join` de Python, que intercala allò que indiquem en els elements de la llista (en aquest cas un espai en blanc). Ara, una vegada tenim la cadena, la dividim en blocs de 4 bits i afegim l'indicador 'Ob' (amb el `.append` concatenem aquests blocs de 4 bits encapçalats per l'indicador). Aleshores, ja tenim la forma que volem per a utilitzar la funció `hex` de Python, per tant, l'apliquem i eliminem l'indicador de Python hexadecimal.

```
1  def b2Tob16(value):
2    #prenim una llista de 32 bits i la convertim a una cadena
3    value = ''.join([str(x) for x in value])
4    #creem blocs de 4 bits i afegim l'indicador dels binaris
5    binaries = []
6    for d in range(0, len(value), 4):
7      binaries.append('0b' + value[d:d+4])
8    #transformem a hexadecimal i eliminem l'indicador d'hexadecimals
9    hexes = ''
10   for b in binaries:
11     hexes += hex(int(b ,2))[2:]
12 return hexes
```

Codi 3.16: Conversor de base 2 a base 16 en Python.

Aquesta funció pren una llista de 32 bits com a entrada i la converteix en la seua representació hexadecimal.

En la línia [3], converteix al valor d'entrada `value`, el qual és una llista de 32 bits, en una cadena. Ho aconsegueix iterant sobre cada element `x` de la llista `value`, convertint-lo en una cadena de text utilitzant `str(x)`, i després concatenant totes les cadenes resultants amb una cadena de text buida com a separador utilitzant el mètode `join()`.

En les línies [5],[6] i [7], es crea una llista anomenada `binaries` per emmagatzemar blocs de 4 bits de la representació binària. La funció `range()` s'utilitza per iterar sobre els índexs de la cadena `value` amb un pas de 4. En cada iteració, s'extrau un bloc de 4 bits utilitzant la tècnica de tall de cadena, començant des de l'índex `d` i acabant en `d+4`. El bloc extret s'afegeix a la llista `binaries` després de prefixar-lo amb `'0b'` per indicar que representa un nombre binari.

En les línies [9],[10] i [11], es converteix cada nombre binari de la llista `binaries` a la seua representació hexadecimal. Per a cada nombre binari `b` de la llista `binaries`, primer es converteix a un nombre enter utilitzant `int(b, 2)`, on el segon argument `2` especifica que el nombre està en base 2 (binari). L'enter resultant es converteix a una cadena de text hexadecimal utilitzant `hex()`, i s'eliminen els dos primers caràcters (el prefix `'0x'`) utilitzant una tècnica de tall de cadena amb `[2:]`. Els dígits hexadecimal resultants s'afegeixen a la cadena `hexes`.

En la línia [12], indica que la funció ha de retornar el valor final de la cadena `hexes`.

Per exemple:

```
b2Tob16([0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1,
0, 0, 0, 1, 0, 0, 0, 0, 0, 1])
Out: '484f4c41'
```

Codi 3.17: Implementació d'un conversor de base 2 a base 16 en Python.

Funcions útils

Per a poder ajustar les longituds dels conjunts de bits i ser capaços de omplir amb zeros a la nostra conveniència, utilitzarem la següent funció, la qual ens servirà per a afegir per davant un nombre donat de zeros a una llista.

```
1  def fillZeros(bits, length=8, endian='LE'):
2      l = len(bits)
3      if endian == 'LE':
4          for i in range(l, length):
5              bits.append(0)
6      else:
7          while l < length:
8              bits.insert(0, 0)
9              l = len(bits)
10 return bits
```

Codi 3.18: Funció per a omplir amb zeros en Python.

En la línia [1], es defineix una funció anomenada `fillZeros` que pren tres paràmetres: `bits`, que és una llista de bits; `length`, que és la longitud objectiu de la llista de bits (per defecte, 8); i `endian`, que especifica l'ordre en què s'afegeixen els zeros (per defecte, 'LE' o little-endian). En la línia [2], es guarda la longitud actual de la llista de bits en una variable anomenada `l`.

En les línies [3], [4] i [5], s'afegeixen zeros al final de la llista de bits si l'ordre és 'LE' (little-endian). Si `endian` és igual a 'LE', es fa un bucle `for` que comença des de la longitud actual de la llista `l` i continua fins a `length`. En cada iteració, s'afegeix un zero a la llista de bits utilitzant el mètode `append()`.

De la línia [6] fins la [9], s'afegeixen zeros a l'inici de la llista de bits si l'ordre és diferent de 'LE'. Si `endian` no és igual a 'LE', es fa un bucle `while` que es repeteix mentre la longitud actual de la llista `l` siga menor que `length`. En cada iteració, s'afegeix un zero a l'inici de la llista de bits utilitzant el mètode `insert(0, 0)`. Després de cada inserció, es recalcula la longitud de la llista `l` per comprovar si és igual o superior a `length`.

En la línia [10], s'indica que la funció ha de retornar la llista de bits final, que pot incloure zeros afegits a l'inici o al final, segons l'ordre especificat per `endian`.

Aquesta funció `fillZeros` és útil per assegurar que una llista de bits tinga una longitud determinada, afegint zeros a l'inici o al final segons l'ordre especificat.

Finalment, també necessitarem ser capaços de dividir una llista de bits en fragments de 512 bits i després de 32 bits. Utilitzarem la següent funció:

```
1 def chunker(bits, chunk_length=8):
2     #dividim la llista de bits en els blocs de bytes/paraules desitjats/des
3     #comencem en el LSB (Least Significant Bit, bit menys significatiu)
4     chunked = []
5     for b in range(0, len(bits), chunk_length):
6         chunked.append(bits[b:b+chunk_length])
7 return chunked
```

Codi 3.19: Funció que divideix una llista de bits en blocs de la longitut desitjada de bits en Python.

En la línia [1], es defineix una funció anomenada `chunker` que pren dos paràmetres: `bits`, que és una llista de bits, i `chunk_length`, que és la longitud desitjada per als blocs de bits (per defecte, 8). En la línia [4], es crea una llista buida anomenada `chunked` per emmagatzemar els blocs de bits dividits.

En les línies [5] i [6], s'itera a través dels índexs de la llista de bits amb un pas igual a la longitud del bloc `chunk_length`. En cada iteració, s'extrau un bloc de bits des de l'índex `b` fins a l'índex `b+chunk_length` utilitzant la tècnica de tall de llista `bits[b:b+chunk_length]`. Aquest bloc de bits s'afegeix a la llista `chunked` utilitzant el mètode `append()`.

En la línia [7], s'indica que la funció ha de retornar la llista `chunked` que conté els blocs de bits dividits.

Aquesta funció `chunker` és útil per dividir una llista de bits en blocs més petits, on cada bloc té una longitud especificada pel paràmetre `chunk_length`. Això pot ser útil per a diverses operacions de processament o anàlisi de dades binàries, on és necessari treballar amb blocs de bits de mida fixa.

Per tant, amb açò ja tenim el que necessitem sobre tipus de dades i ara passem a parlar sobre alguns paràmetres fixes i constants que l'algorisme requereix.

3.6.2 Valors de resum inicials i constants enteres

Valors de resum inicials

Els valors de resum inicials, h, són normalment fixes i estan definits per el NIST (National Institute of Standards and Technology) [33].

Aquests valors fixes són els primers 32 bits de les parts fraccionals de les arrels quadrades dels primers 8 números primers en representació hexadecimal. Ara donarem un exemple del càlcul i conversió d'un d'aquests valors.

```
1 import math
2
3 primes = [2, 3, 5, 7, 11, 13, 17, 19]
4 h = []
5
6 for prime in primes:
```

```
    square_root = math.sqrt(prime)
    fractional_part = square_root - math.floor(square_root)
    hex_value = hex(int(fractional_part * (2**32))).zfill(8)
    h.append(hex_value)

print(h)
```

Codi 3.20: Càlcul dels valors de resum inicials h en Python.

Aquest codi utilitza el mòdul `math` per calcular les arrels quadrades dels primers 8 números primers $(2, 3, 5, 7, 11, 13, 17, 19)$. Després, es calcula la part fraccional de cada arrel quadrada restant-li la part entera. A continuació, es multiplica aquesta part fraccional per 2^{32} per obtenir els primers 32 bits de la representació decimal. Finalment, aquests valors es converteixen a la seua representació hexadecimal mitjançant la funció `hex()`, i es garanteix que cada valor tinga una longitud de 8 caràcters mitjançant `zfill(8)`. Els resultats s'emmagatzemen en una llista anomenada `h` i s'imprimeixen per pantalla.

I obtenim aquests valors:

```
Out: h=['0x6a09e667','0xbb67ae85','0x3c6ef372','0xa54ff53a','0x510e527f',
'0x9b05688c','0x1f83d9ab','0x5be0cd19']
```

Codi 3.21: Valors de resum inicials calculats amb Python.

Però, no necessitarem aquest codi, ja que utilitzarem els valors fixes i els inicialitzarem amb una funció que definirem més endavant.

Constants enteres

Un conjunt de constants (k), que utilitzarem per a barrejar-lo durant el procés del resum, són els primers 32 bits de la part fraccional de les arrels cúbiques dels primers 64 números primers. A continuació, tenim el codi per a calcular-los:

```
import math

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

k = []

for prime in primes:
    cube_root = prime ** (1/3)
    fractional_part = cube_root - math.floor(cube_root)
    binary_value = int(fractional_part * (2 ** 32))
    hex_value = hex(binary_value).zfill(8)
    k.append(hex_value)

```

```
print(k)
```

Codi 3.22: Càlcul del conjunt de constants k que utilitzarem durant el procés de resum.

Aquest codi utilitza una llista de longitud 64 anomenada `primes` que conté els primers 64 números primers. A continuació, es recorre aquesta llista i per a cada nombre primer es calcula l'arrel cúbica utilitzant l'operador `**` amb l'exponent 1/3. Després, es calcula la part fraccional restant-li la part entera de l'arrel cúbica.

A continuació, aquesta part fraccional es multiplica per 2^{32} per obtenir els primers 32 bits de la representació decimal. Aquest valor es converteix a una representació hexadecimal utilitzant la funció `hex()`, i s'assegura que tinga una longitud de 8 caràcters utilitzant `zfill(8)`. Cada valor hexadecimal s'emmagatzema en una llista anomenada `k`.

Finalment, es mostra el contingut de la llista `k` imprimint-lo per pantalla. Això mostrarà els primers 32 bits de la part fraccional de les arrels cúbiques dels primers 64 números primers. Aquest codi, ens dóna la següent eixida:

```
Out: k=['0x428a2f98', '0x71374491', '0xb5c0fbcf', '0xe9b5dba5',
'0x3956c25b', '0x59f111f1', '0x923f82a4', '0xab1c5ed5',
'0xd807aa98', '0x12835b01', '0x243185be', '0x550c7dc3',
'0x72be5d74', '0x80deb1fe', '0x9bdc06a7', '0xc19bf174',
'0xe49b69c1', '0xefbe4786', '0xfc19dc6', '0x240ca1cc',
'0x2de92c6f', '0x4a7484aa', '0x5cb0a9dc', '0x76f988da',
'0x983e5152', '0xa831c66d', '0xb00327c8', '0xbf597fc7',
'0xc6e00bf3', '0xd5a79147', '0x6ca6351', '0x14292967',
'0x27b70a85', '0x2e1b2138', '0x4d2c6dfc', '0x53380d13',
'0x650a7354', '0x766a0abb', '0x81c2c92e', '0x92722c85',
'0xa2bfe8a1', '0xa81a664b', '0xc24b8b70', '0xc76c51a3',
'0xd192e819', '0xd6990624', '0xf40e3585', '0x106aa070',
'0x19a4c116', '0x1e376c08', '0x2748774c', '0x34b0bcb5',
'0x391c0cb3', '0x4ed8aa4a', '0x5b9cca4f', '0x682e6ff3',
'0x748f82ee', '0x78a5636f', '0x84c87814', '0x8cc70208',
'0x90befffa', '0xa4506ceb', '0xbef9a3f7', '0xc67178f2']
```

Codi 3.23: Constants enteres calculades en Python.

Aleshores ja tenim els valors hexadecimals corresponents. Aquests també seran fixes com els d'abans.

Inicialitzar els valors

Per a inicialitzar-los en l'algorisme i convertir-los en una llista de bits, utilitzarem el següent codi:

```
def initializer(values):
    #convertim de hexadecimal a la cadena binària de Python (suposem que ja
     estan sense l'indicador binari('0b'))
    binaries = [bin(int(v, 16)) for v in values]
    #convertim de cadena de Python a una llista de llistes de 32 bits
```

```
5       words = []
6       for binary in binaries:
7           word = []
8           for b in binary:
9               word.append(int(b))
10          words.append(fillZeros(word, 32, 'BE'))
11  return words
```

Codi 3.24: Funció que converteix els valors d'una llista de la seua representació hexadecimal a una llista de llistes de 32 bits en forma de binari en Python.

En la línia [1], es defineix una funció anomenada `initializer` la qual pren el paràmetre `values`. La línia [3], converteix els valors hexadecimals de la llista `values` a la seua representació binària en Python. Itera sobre cada valor en `values`, el converteix a un enter utilitzant `int(v, 16)`, i després converteix l'enter a una cadena binària utilitzant `bin()`. Les cadenes binàries resultants s'emmagatzemen en la llista `binaries`.

De la línia [5] fins la [10], converteix les cadenes binàries de la llista `binaries` en una llista de llistes, on cada llista interna representa una paraula de 32 bits. S'inicialitza una llista buida anomenada `words` per emmagatzemar aquestes paraules de 32 bits.

Per a cada cadena binària de la llista `binaries`, es recorre cada caràcter `b` de la cadena binària. A continuació es converteix cada caràcter a un enter utilitzant `int(b)` i s'afegeix a la llista `word`. Després de processar tots els caràcters de la cadena binària, s'afegeix la llista `word` a la llista `words`. La funció `fillZeros()` s'utilitza per garantir que cada llista `word` estiga plena de zeros al davant fins a una longitud de 32 bits, utilitzant el paràmetre `'BE'` (big-endian).

En la línia [11], retorna la llista `words`, que conté les paraules de 32 bits corresponents als valors hexadecimals introduïts a l'inici.

En resum, aquest codi defineix una funció anomenada `initializer` que rep una llista de valors com a paràmetre. El propòsit d'aquesta funció és convertir els valors de la llista de la seua representació hexadecimal a una llista de llistes de 32 bits en forma de binari.

3.6.3 Preparació del missatge ("*padding*")

El primer pas sempre serà preparar o reomplir el missatge. Açò, pot ser resumit en els següents tres passos:

1. Adjuntem un únic '1' al final del missatge.
2. Determinem la longitud de bits del missatge.
3. El partim d'una manera tal que la seua longitud total siga un múltiple de 512.

Notem que, depenent de la longitud del missatge, el tractarem de diferent forma:

- Si és més curt de 448 bits, simplement afegim un '1' i afegim la longitud en bits.

- Si la seua longitud està entre 449 i 512 bits, anem al següent múltiple de 512, que és 1024 bits i seguim el mateix procediment.
- Si la longitud del missatge és més de 512 bits, determinem la longitud total amb un bucle `while`.

```
def preprocessMessage(message):
    #traduïm el missatge a bits
    bits = translate(message)
    #longitud del missatge
    length = len(bits)
    #obtenim la longitud del missatge en bits (en un bloc de 64 bits)
    message_len = [int(b) for b in bin(length)[2:].zfill(64)]
    #si la longitud és més menuda que 448, tractarem el bloc de manera individual, altrament
    #si és exactament de 448, afegirem un únic 1 i ho afegirem a 1024
    #si és més llarga que 448, crearem un múltiple de 512 - 64 bits per a la longitud al final del missatge (big endian)
    if length < 448:
        #afegim un únic 1
        bits.append(1)
        #ho plenem de zeros (de la forma little endian)
        bits = fillZeros(bits, 448, 'LE')
        #afegim els 64 bit representant la longitud del missatge
        bits = bits + message_len
        #ho retornem com a llista
    return [bits]

    elif 448 <= length <= 512:
        bits.append(1)
        #ho menegem al bloc del següent missatge - total de la longitud = 1024
        bits = fillZeros(bits, 1024, 'LE')
        #reemplacem els últims 64 bits pel múltiple de 512 amb la longitud original del missatge
        bits[-64:] = message_len
        #ho retornem en blocs de 512 bits
    return chunker(bits, 512)

    else:
        bits.append(1)
        #fem un bucle fins obtindre un múltiple de 512 + la longitud del missatge escrita en forma de 64 bits si la longitud del missatge és superior a 448 bits
        while (len(bits)+64) % 512 != 0:
            bits.append(0)
        #afegim els 64 bits representant la longitud del missatge
        bits = bits + message_len
        #ho retornem en blocs de 512 bits
```

```
38     return chunker(bits, 512)
```

Codi 3.25: Funció que prepara el missatge per a ser processat en un algorisme de resum en Python.

Aquesta funció, `preprocessMessage`, preprocesa un missatge per a ser utilitzat en un algorisme de resum.

Primerament, es defineix la funció `preprocessMessage` amb un paràmetre `message` que representa el missatge a preprocessar. Es crida a la funció `translate(message)` per convertir el missatge a una seqüència de bits i el resultat s'emmagatzema a la variable `bits`. Es calcula la longitud del missatge en bits i s'emmagatzema a la variable `length`. Es crea una representació de 64 bits de la longitud del missatge, `message_len`, convertint `length` a binari amb `bin(length)`. S'eliminen els dos primers caràcters (indicador binari '0b') i es completa amb zeros a l'esquerra fins a obtenir una longitud de 64 caràcters amb `zfill(64)`. El resultat és una llista de 64 elements amb els bits de la longitud del missatge. S'avalua una condició per comprovar la longitud del missatge.

Si la longitud és inferior a 448 bits, s'afegeix un 1 al final de la llista `bits`. A continuació, s'emplenen de zeros a la dreta fins a una longitud de 448 bits utilitzant la funció `fillZeros(bits, 448, 'LE')`. Finalment, es concatenen els 64 bits de la longitud del missatge amb la llista `bits`. El resultat es retorna com una llista que conté la seqüència de bits preprocessada.

Si la longitud és igual o superior a 448 bits i menor o igual a 512 bits, s'afegeix un 1 al final de la llista `bits`. A continuació, s'emplenen de zeros a la dreta fins a una longitud de 1024 bits utilitzant `fillZeros(bits, 1024, 'LE')`. Els últims 64 bits de `bits` es substitueixen per la llista `message_len`. El resultat es retorna utilitzant la funció `chunker(bits, 512)` per dividir la seqüència de bits en blocs de 512 bits.

Si la longitud és superior a 512 bits, s'afegeix un 1 al final de la llista `bits`. Després, es realitza un bucle mentre la longitud de `bits` més 64 no siga divisible per 512. En cada iteració del bucle, s'afegeix un zero a la llista `bits`. Finalment, es concatenen els 64 bits de la longitud del missatge amb la llista `bits`. El resultat final es retorna utilitzant la funció `chunker(bits, 512)` per dividir la seqüència de bits en blocs de 512 bits.

Aquesta funció és responsable de preparar el missatge per a ser processat en un algorisme de resum, assegurant que tinga una longitud adequada i complisca amb els requeriments específics del protocol de resum utilitzat.

Provem-ho introduint el missatge "hola món" com a valor d'entrada del codi anterior i mostrem el resultat.

```
1 m = preprocessMessage('hola mon')
2 print(m)
3 Out:[[0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0,
    0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0,
    1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 1, 0, 0, 0, 0, 0, 0]]
```

Codi 3.26: Exemple de la funció preprocessMessage introduïnt com a valor d'entrada "hola mon" en Python.

Observem que hem elegit un missatge que està per davall dels 448 bits, aleshores estem en el primer cas. En particular, la cadena "hola món" té una longitud en bits de 64 bits (8 caràcters, cadascun d'ells un byte, 8 bits). En aquest cas, s'afegeix un 1 al final de la llista `bits`, que és el nostre missatge traduït a binari. Com el missatge que hem pres representat de manera binària té 64 bits, afegim l'1 en la posició 65. A continuació, s'emplenen de zeros a la dreta fins a una longitud de 448 bits utilitzant la funció `fillZeros(bits, 448, 'LE')`. Per tant, tindrem en les primeres 64 posicions de la nostra llista la representació binària del missatge, en la posició 65 l'1 que hem afegit, i de la posició 66 a la 448, tindrem tot 0's. Finalment, es concatenen els 64 bits que representen la longitud del missatge amb la llista, la qual cosa implica que de la posició 449 fins la 512 tenim la longitud del missatge expressada en la seua representació binària de 64 bits 'big endian' (és a dir els bits menys significatius al final). Podem comprovar en l'eixida que tot es satisfà correctament.

Açò, és justament el que l'algorisme SHA-256 requereix, i per tant, podem avançar a l'algorisme en qüestió.

3.6.4 Algorisme

Observant l'algorisme de manera general, podem dir que hi ha un bucle principal, que itera sobre els blocs de 512 bits creats per la funció que els preprocessa. A continuació, cada bloc es barreja amb ell mateix i amb una gran quantitat de zeros, i després el passarem per un procés elaborat de mescla. Finalment, els valors inicials de resum s'actualitzen pels corresponents al procés actual. Aquests valors són retornats en format hexadecimal al final de l'última iteració del bucle principal.

Utilitats abans de començar

Abans de començar, necessitem "modificar" algunes funcions integrades de Python, ja que aquestes treballen amb la representació binària de Python i a nosaltres ens interessa treballar amb llistes. Per tant, crearem la nostra pròpia versió tot i que aquestes funcions podem trobar-les integrades en Python. Són funcions d'utilitat i les requerirem més endavant.

```
#la condició de certesa (true) és equivalent a l'enter 1
def isTrue(x): return x == 1

#if simple
def if_(i, y, z): return y if isTrue(i) else z

#and - ambdós arguments han de ser certs
def and_(i, j): return if_(i, j, 0)
def AND(i, j): return [and_(ia, ja) for ia, ja in zip(i,j)]

#simplement nega l'argument
def not_(i): return if_(i, 0, 1)
def NOT(i): return [not_(x) for x in i]

#torna cert (true) si i o j són certs però no els dos al mateix temps
def xor(i, j): return if_(i, not_(j), j)
def XOR(i, j): return [xor(ia, ja) for ia, ja in zip(i, j)]

#si el nombre de valors certs és senar aleshores retorna cert (true)
def xorxor(i, j, l): return xor(i, xor(j, l))
def XORXOR(i, j, l): return [xorxor(ia, ja, la) for ia, ja, la, in zip(i, j
    , l)]

#obté el valor que representa la majoria dels resultats en freqüència, per
    exemple si dos o més de tres valors són el mateix
def maj(i,j,k): return max([i,j,], key=[i,j,k].count)
```

Codi 3.27: Funcions booleanes en Python.

Aquestes són les portes lògiques o funcions booleanes, i les utilitzarem més endavant.

Ara crearem un sumador binari i per a això necessitarem utilitzar operacions de desplaçament i rotació a nivell de bit.

```
#rotació cap a la dreta
def rotr(x, n): return x[-n:] + x[:-n]

#desplaçament cap a la dreta
def shr(x, n): return n * [0] + x[:-n]

#sumador binari complet
def add(i, j):
  #pren una llista de binaris i els suma
```

```
  length = len(i)
  sums = list(range(length))
  #la entrada inicial necessita ser igual a 0
  c = 0
  for x in range(length-1,-1,-1):
    #afegim els bits d'entrada amb una doble funció booleana xor
    sums[x] = xorxor(i[x], j[x], c)
    #el bit que hem d'arrossegar és el bit més representat
    #per exemple si tenim 0,1,0 arrossegarem el 0
    c = maj(i[x], j[x], c)
  #retornem la llista de bits
  return sums
```

Codi 3.28: Codi de tres funcions en Python: rotació cap a la dreta; desplaçament cap a la dreta i sumador binari complet.

Primerament, comencem explicant la rotació cap a la dreta. La funció `rotr` realitza una rotació cap a la dreta de la llista `x` en `n` posicions. Agafa els últims `n` elements de `x` i els afegeix al principi de la llista, mentre que els primers elements d'`x` es desplacen cap al final.

A continuació, explicarem el desplaçament cap a la dreta. La funció `shr` realitza un desplaçament cap a la dreta de la llista `x` en `n` posicions. Afegeix `n` zeros al principi de la llista i elimina els `n` últims elements d'`x`.

Per últim, parlem del sumador binari complet. La funció `add` és un sumador binari complet que pren dues llistes de bits `i` i `j` i realitza una suma bit a bit. Es crea una nova llista `sums` de la mateixa longitud que `i` i `j`. S'inicialitza la variable `c` a 0, que és la que utilitzarem per a emmagatzemar el bit que s'ha de portar. A continuació, es recorre la llista de forma inversa i s'aplica la funció `xorxor` als bits d' `i`, `j` i `c` per obtenir el bit de suma. La funció `xorxor` realitza una operació XOR en cascada entre tres bits consecutius `i`, `j` i `l`, utilitzant la funció `xor` per a les operacions XOR individuals. El bit que s'ha de portar (carry) a la següent suma es calcula utilitzant la funció `maj` (majoritària). La funció `maj` determina el major valor entre tres valors donats, basant-se en la freqüència de cada valor. Finalment, es retorna la llista de bits `sums` amb el resultat de la suma.

Ací tenim un exemple d'una suma:

```
add([1,0,1],[1,1,1])
Out: [1, 0, 0]
```

Codi 3.29: Exemple en Python del sumador binari complet.

La funció `add` rep dues llistes de bits `i` i `j` com a paràmetres. Aquesta funció realitza una suma binària completa entre les dues llistes de bits.

En aquest cas, estem sumant les llistes de bits `[1, 0, 1]` i `[1, 1, 1]`. Comencem amb un transport inicial c igual a zero. A continuació, s'itera des de la darrera posició fins a la primera posició de les llistes de bits, en ordre invers. A cada iteració, es realitzen les següents operacions. Primerament, es calcula la suma de tres bits: `i[x]`, `j[x]` i `c`.

Aquesta operació es realitza utilitzant la funció `xor()`, que retorna el resultat de l'operació XOR entre els seus paràmetres. Això vol dir que es calcula la suma lògica de tres bits. A continuació, es determina el bit de transport `c` per a l'iteració següent. Aquest bit és el resultat de l'operació `maj()`, que rep `i[x]`, `j[x]` i `c` com a paràmetres. La funció `maj()` selecciona el major valor entre `i[x]`, `j[x]` i `c`, basant-se en la freqüència de cada valor.

Després de completar totes les iteracions, es retorna la llista `sums` que conté el resultat final de la suma binària completa entre les dues llistes de bits.

En aquest cas, la suma de `[1, 0, 1]` i `[1, 1, 1]` és `[1, 0, 0]`. En aquest cas, el bit de transport final `c` és 1 i el resultat de la suma es troba a la llista `sums`, que és `[1, 0, 0]`. El sumador binari complet, per defecte arrossega un bit. Si l'últim bit que arrossega és un 1, com al nostre cas, el descartem ja que ens hem de cenyir a les longituds predefinides. De fet, en aquest exemple hem sumat $5 + 7$, que són 12 (en binari $[1, 1, 0, 0]$), però degut a que el bit que deuríem haver arrossegat ha sigut descartat, obtenim $[1, 0, 0]$, que és la representació binària de 4 en base 2.

Preparatius previs al bucle principal

Per a poder controlar el bucle i ser capaços d'actualizar-lo després de cada iteració, inicialitzarem els valors de resum $h = h_0, \cdots, h_7$ i les constants enteres k que hem definit abans. Per a aconseguir açò, la funció d'inicialització que hem definit prèviament juga el seu paper. El propòsit d'aquesta funció és convertir els valors de la llista de la seua representació hexadecimal a una llista de llistes de 32 bits en forma de binari. Per tant, li donem com a entrada h i k que estan en la seua representació hexadecimal, i obtindrem aquestes constants inicialitzades en el format que ens interessa.

```
k=['0x428a2f98', '0x71374491', '0xb5c0fbcf', '0xe9b5dba5',
'0x3956c25b', '0x59f111f1', '0x923f82a4', '0xab1c5ed5',
'0xd807aa98', '0x12835b01', '0x243185be', '0x550c7dc3',
'0x72be5d74', '0x80deb1fe', '0x9bdc06a7', '0xc19bf174',
'0xe49b69c1', '0xefbe4786', '0xfc19dc6', '0x240ca1cc',
'0x2de92c6f', '0x4a7484aa', '0x5cb0a9dc', '0x76f988da',
'0x983e5152', '0xa831c66d', '0xb00327c8', '0xbf597fc7',
'0xc6e00bf3', '0xd5a79147', '0x6ca6351', '0x14292967',
'0x27b70a85', '0x2e1b2138', '0x4d2c6dfc', '0x53380d13',
'0x650a7354', '0x766a0abb', '0x81c2c92e', '0x92722c85',
'0xa2bfe8a1', '0xa81a664b', '0xc24b8b70', '0xc76c51a3',
'0xd192e819', '0xd6990624', '0xf40e3585', '0x106aa070',
'0x19a4c116', '0x1e376c08', '0x2748774c', '0x34b0bcb5',
'0x391c0cb3', '0x4ed8aa4a', '0x5b9cca4f', '0x682e6ff3',
'0x748f82ee', '0x78a5636f', '0x84c87814', '0x8cc70208',
'0x90befffa', '0xa4506ceb', '0xbef9a3f7', '0xc67178f2']

h=['0x6a09e667','0xbb67ae85','0x3c6ef372','0xa54ff53a','0x510e527f',
'0x9b05688c','0x1f83d9ab','0x5be0cd19']

```

```
#inicialitzem les constants
k = initializer(k)
#inicialitzem els valors de resum inicials
h0, h1, h2, h3, h4, h5, h6, h7 = initializer(h)
```

Codi 3.30: Codi on s'inicialitzen les constants en el format desitjat en Python.

Per tant, en el cas de k l'eixida d'aquest codi serà una llista de 64 llistes de 32 bits. I, en el cas d'h, obtindrem una llista de 8 llistes de 32 bits, però ací, separem aquestes 8 llistes en les variables $h_0, \cdots, h_7$ respectivament (per a poder accedir a elles de manera més directa més avant). I així, h_i amb $0 \leq i \leq 7$, és una llista de 32 bits.

El bucle principal

Com hem dit abans, el bucle principal itera sobre blocs de 512 bits, és a dir, el missatge complet. Durant les iteracions, se segueixen els següents passos:

- **Programació del missatge**

  Baix tenim la primera part del bucle principal, on el programa del missatge es crea.

```
#inicialitzem els valors i les constants
k = initializer(k)
h0, h1, h2, h3, h4, h5, h6, h7 = initializer(h)
#preparem el missatge
chunks = preprocessMessage(message)
#bucle principal
for chunk in chunks:
    #programació del missatge
    #creem listes de paraules de 32 bits (512 bits/32 =16 paraules)
    w = chunker(chunk, 32)
    #extenem la longitud de cada bloc a 64 paraules
    #les 48 restants les inicialitzem en 0
    for _ in range(48):
        w.append(32 * [0])
```

Codi 3.31: Codi on s'inicialitzen els valors i es prepara el missatge per al preprocessament i s'executa un bucle principal per a cada bloc de missatge en Python.

A continuació, expliquem el codi:

1. `k = initializer(k)`: Es crida a la funció `initializer(k)` per inicialitzar els valors i les constants de la variable `k`. Aquesta funció converteix els valors de la llista `k` a una representació binària de llistes de 32 bits. Els valors i les constants són assignats a la variable `k`.
2. `h0, h1, h2, h3, h4, h5, h6, h7 = initializer(h)`: Es crida a la funció `initializer(h)` per inicialitzar els valors i les constants de la variable `h`. Aquesta funció converteix els valors de la llista `h` a una representació binària de llistes

de 32 bits. Els valors i les constants són assignats a les variables `h0, h1, h2, h3, h4, h5, h6, h7`.

3. `chunks = preprocessMessage(message)`: Es crida a la funció per preparar el missatge. Aquesta funció realitza el preprocessament del missatge per a ser utilitzat en l'algoritme de resum. El resultat del preprocessament és assignat a la variable `chunks`.
4. `for chunk in chunks`: s'inicia un bucle principal en el qual cada iteració processarà un bloc del missatge. La variable `chunk` contindrà el bloc actual en cada iteració.
5. `w = chunker(chunk, 32)`: es crida a la funció per crear llistes de paraules de 32 bits a partir del bloc actual. Aquesta funció divideix el bloc en paraules de 32 bits i les retorna en una llista. Aquesta llista de paraules de 32 bits és assignada a la variable `w`.
6. `for _ in range(48)`: s'inicia un bucle que s'executarà 48 vegades. A cada iteració, s'afegeixen 32 zeros a la llista `w` per extraure la longitud del bloc a 64 paraules. Això és necessari per a l'algorisme de resum específic que estem implementant (SHA-256). El valor de la variable `_` no s'utilitza, simplement indica que no es necessita accedir al valor de l'índex en aquest bucle.
7. `w.append(32 * [0])`: a cada iteració del bucle anterior, s'afegeix una llista de 32 zeros a la llista `w`. Això extén la longitud de cada bloc a 64 paraules, afegint 48 zeros a la llista.

Aquest codi inicialitza valors i constants, prepara el missatge per al processament i executa un bucle principal per a cada bloc del missatge. En cada iteració del bucle, es creen llistes de paraules de 32 bits a partir del bloc actual i s'extén la longitud del bloc a 64 paraules afegint zeros.

Una vegada tenim el missatge en el format que ens interessa, realitzarem les següents operacions en cada iteració. Primerament, veurem aquestes operacions amb pseudocodi:

```
sigma0 = (w[i-15] rotate right by 7) xor
(w[i-15] rotate right by 18) xor
(w[i- 15] shift right by 3)

sigma1 = (w[i-2] rotate right by 17) xor
(w[i-2] rotate right by 19) xor
(w[i-2] shift right by 10)
```

Codi 3.32: Pseudocodi que descriu el càlcul de les variables sigma0 i sigma1.

Aquest pseudocodi descriu el càlcul de les variables `sigma0` i `sigma1` basades en les paraules `w[i-15]` i `w[i-2]`. Expliquem-ho pas per pas.

Primerament, expliquem `sigma0`:

- Es pren la paraula `w[i-15]` i se li aplica una rotació cap a la dreta de 7 bits.
- Al resultat de la rotació cap a la dreta se li aplica una operació XOR amb una altra rotació cap a la dreta de 18 bits de la mateixa paraula `w[i-15]`.
- Al resultat de la segona rotació cap a la dreta se li aplica una operació XOR amb un desplaçament cap a la dreta de 3 bits de la mateixa paraula `w[i-15]`.
- El resultat d'aquestes operacions XOR consecutives és el valor de la variable `sigma0`.

A continuació, explicarem `sigma1`:

- Es pren la paraula `w[i-2]` i se li aplica una rotació cap a la dreta de 17 bits.
- Al resultat de la rotació cap a la dreta se li aplica una operació XOR amb una altra rotació cap a la dreta de 19 bits de la mateixa paraula `w[i-2]`.
- Al resultat de la segona rotació cap a la dreta se li aplica una operació XOR amb un desplaçament cap a la dreta de 10 bits de la mateixa paraula `w[i-2]`.
- El resultat d'aquestes operacions XOR consecutives és el valor de la variable `sigma1`.

En resum, `sigma0` es calcula realitzant una sèrie de rotacions cap a la dreta i operacions XOR a la paraula `w[i-15]`, mentre que `sigma1` es calcula de manera similar a partir de la paraula `w[i-2]`.

I després, ens quedarà fer aquesta operació: `w[i]` es convertirà en la suma de `w[i-16]`, `sigma0`, `w[i-17]` i `sigma1`, és a dir:

```
w[i] = w[i-16] + sigma0 + w[i-17] + sigma1
```

Codi 3.33: Pseudocodi que descriu el cálcul de w(i).

Aquestes operacions són comunes en algorismes criptogràfics, com SHA-256, per generar la seqüència de paraules `w` basada en les paraules anteriors i altres valors derivats.

A continuació, tenim el codi de Python de les operacions que hem explicat amb el pseudocodi:

```
    w.append(32 * [0])

for i in range(16, 64):
    s0 = XORXOR(rotr(w[i-15], 7), rotr(w[i-15], 18), shr(w[i-15], 3) )
    s1 = XORXOR(rotr(w[i-2], 17), rotr(w[i-2], 19), shr(w[i-2], 10))
    w[i] = add(add(add(w[i-16], s0), w[i-7]), s1)
```

Codi 3.34: Codi on s'apliquen les operacions vistes en el pseudocodi en Python.

Aquest bucle aplica a cada `w` les operacions que hem definit anteriorment per a aconseguir les transformacions descrites en el pseudocodi.

Seguint pas a pas l'algorisme, anem veient com es barregen les paraules de 32 bits. Les rotacions i desplaçaments fan el seu paper, i l'operador XOR també.

Finalment, reemplacem el valor dels índexs de la llista de fragments del missatge. Aleshores, inicialitzarem les variables de pas `a,b,c,d,e,f,g,h` amb els valors de resum inicials `h0,..., h7`, abans de passar al següent bucle.

```
    w[i] = add(add(add(w[i-16], s0), w[i-7]), s1)
a = h0
b = h1
c = h2
d = h3
e = h4
f = h5
g = h6
h = h7
```

Codi 3.35: Codi on s'assignen a les variables a...h el valor de les variables h0...h7 respectivament en Python.

En aquest codi, hem assignat a les variables `a,..., h`, el valor de les variables `h0,..., h7` respectivament.

De moment, estem encara en el bucle principal, és a dir, en el missatge de 512 bits fragmentat. Aquests elements es troben a la mateixa altura (parlant de tabulat): el bucle anterior, les variables d'inicialització que hem vist, el bucle de compressió que veurem a continuació i l'actualització dels valors de resum.

- **Compressió** En aquesta part de l'algorisme és on realment es construeixen els bits i les paraules, i on les constants `k` intervenen barrejant-se.

  Primer, veurem el pseudocodi. Aquest bucle iterarà en el rang 0 a 63. Cada iteració `j` implica les següents operacions:

```
Sigma1=(e rotate right by 6) xor
       (e rotate right by 11) xor
       (e rotate right by 25)
choose=(e and f) xor (not e and g)
temp1=h + Sigma1 + change + k[j] + w[j]
Sigma0=(a rotate right by 2) xor
       (a rotate right by 13) xor
       (a rotate right a by 22)
majority=(a and b) xor (b and c) xor (b and c)
temp2=Sigma0 + majority
```

Codi 3.36: Pseudocodi on es realitzen una sèrie d'operacions per calcular els valors de Sigma1 choose temp1 Sigma0 temp2 i majority.

Ací tenim l'explicació del pseudocodi:

Calculem `Sigma1` fent tres rotacions cap a la dreta de la variable `e` amb diferents quantitats de rotació (6, 11 i 25) i després apliquem l'operació XOR a aquests tres resultats.

Calculem `choose` fent l'operació AND entre les variables `e` i `f`, i després l'operació XOR entre aquest resultat i l'operació AND entre la negació de `e` i `g`.

Calculem `temp1` sumant les variables `h`, `Sigma1`, `choose`, `k[j]` (on `j` és una variable) i `w[j]` (on `j` és una variable).

Calculem `Sigma0` fent tres rotacions cap a la dreta de la variable `a` amb diferents quantitats de rotació (2, 13 i 22) i després aplica l'operació XOR a aquests tres resultats.

Calculem `majority` fent l'operació AND entre les variables `a` i `b`, després l'operació AND entre les variables `b` i `c`, i finalment l'operació XOR entre els dos resultats anteriors i l'operació AND entre `b` i `c`.

Calculem `temp2` sumant les variables `Sigma0` i `majority`.

En resum, realitzem una sèrie d'operacions per calcular els valors de `Sigma1`, `choose`, `temp1`, `Sigma0`, `majority` i `temp2` utilitzant diverses rotacions, operacions lògiques i operacions d'addició.

Seguidament, ve una reassignació de la següent manera:

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

Codi 3.37: Codi on es realitza una reassignació de les variables.

En aquest codi, simplement sobreescrivim les variables que ja teníem definides abans `a, b, c,..., h`, assignant-li un valor diferent al que tenien.

A continuació tenim el codi de Python d'aquests passos:

```
    h = h7

for j in range(64):
    S1 = XORXOR(rotr(e, 6), rotr(e, 11), rotr(e, 25) )
    ch = XOR(AND(e, f), AND(NOT(e), g))
    temp1 = add(add(add(add(h, S1), ch), k[j]), w[j])
    S0 = XORXOR(rotr(a, 2), rotr(a, 13), rotr(a, 22))
    m = XORXOR(AND(a, b), AND(a, c), AND(b, c))
```

```
    temp2 = add(S0, m)
    h = g
    g = f
    f = e
    e = add(d, temp1)
    d = c
    c = b
    b = a
    a = add(temp1, temp2)
```

Codi 3.38: Codi en Python de les transformacions explicades en el pseudocodi.

En aquest codi, simplement hem escrit en llenguatge Python les transformacions que hem explicat abans en el pseudocodi, utilitzant les operacions booleanes que hem definit.

Ja tenim la part més important de l'algorisme i ara veurem com es van actualitzant els valors de resum.

- **Actualització dels valors de resum (valors de *hash*)**

  A continuació ve un altra ronda de reassignar els valors de resum `h0,..., h7` als resultats del bucle d'abans, és a dir l'estat de les variables de la `a` a la `z` quan $j = 63$. Afegim els valors de resum actuals a les variables de pas i convertim els resultats en els nous valors de resum. Ací tenim el codi de Python:

```
    a = add(temp1, temp2)

h0 = add(h0, a)
h1 = add(h1, b)
h2 = add(h2, c)
h3 = add(h3, d)
h4 = add(h4, e)
h5 = add(h5, f)
h6 = add(h6, g)
h7 = add(h7, h)
#...
```

Codi 3.39: Codi en Python on s'actualitzen els valors de resum.

En aquest codi de Python, estem utilitzant l'operació `add` que hem definit anteriorment per a barrejar les variables de resum que tenim i obtindre noves. Ja hem vist que aquests processos són molt freqüents durant el desenvolupament d'una funció de resum.

Aquest procés de programar el missatge, comprimir les dades i reassignar valors es realitza per a cada bloc del missatge.

Per tant, ja tenim el bucle principal i ara ens quedem amb 8 paraules de 32 bits que són bàsicament el valors de resum final. No obstant això, ho convertirem a la representació hexadecimal, ja que visiblement és més fàcil.

- **Conversió a hexadecimal**

  Ja que hem creat la funció per a convertir llistes de 32 bits en un valor hexadecimal, utilitzarem simplement el codi per a afegir els valors de resum `h0,..., h7` entre ells.

```
#...h7 = add(h7, h)
digest = ''
for val in [h0, h1, h2, h3, h4, h5, h6, h7]:
    digest += b2Tob16(val)
return digest
```

Codi 3.40: Codi en Python on es converteix cada valor de la llista en un format hexadecimal i uneix tots aquests valors en una única cadena de caràcters.

  A continuació tenim l'explicació del codi:

  Primerament, es crea una variable buida anomenada `digest`. Aquesta variable s'utilitzarà per emmagatzemar el resultat final del càlcul. S'inicia un bucle `for` que iterarà a través dels valors de la llista `[h0, h1, h2, h3, h4, h5, h6, h7]`. A cada iteració, el valor actual es guardarà en la variable `val`. Dins del bucle, s'afegeix el resultat de cridar a la funció `b2Tob16(val)` a la variable `digest`. La funció `b2Tob16()` converteix el valor binari `val` en una cadena de caràcters hexadecimal. Una vegada finalitza el bucle, es retorna el valor de la variable `digest`, que conté la cadena de caràcters hexadecimal resultat de concatenar els valors convertits de la llista `[h0, h1, h2, h3, h4, h5, h6, h7]`.

  En resum, aquest codi recorre una llista de valors `[h0, h1, h2, h3, h4, h5, h6, h7]`, converteix cada valor de la llista en un format hexadecimal utilitzant la funció `b2Tob16()`, i després uneix tots aquests valors hexadecimals en una única cadena de caràcters. Aquesta cadena resultant és el valor de retorn de la funció.

  Aquest és l'últim pas abans de que acabe el bucle principal, per tant, el valor que ens retorna la conversió a hexadecimal serà el valor que ens retornarà l'algorisme.

- **La funció SHA-256**

```
def sha256(message):
    k=['0x428a2f98', '0x71374491', '0xb5c0fbcf', '0xe9b5dba5',
    '0x3956c25b', '0x59f111f1', '0x923f82a4', '0xab1c5ed5',
    '0xd807aa98', '0x12835b01', '0x243185be', '0x550c7dc3',
    '0x72be5d74', '0x80deb1fe', '0x9bdc06a7', '0xc19bf174',
    '0xe49b69c1', '0xefbe4786', '0xfc19dc6', '0x240ca1cc',
    '0x2de92c6f', '0x4a7484aa', '0x5cb0a9dc', '0x76f988da',
    '0x983e5152', '0xa831c66d', '0xb00327c8', '0xbf597fc7',
    '0xc6e00bf3', '0xd5a79147', '0x6ca6351', '0x14292967',
    '0x27b70a85', '0x2e1b2138', '0x4d2c6dfc', '0x53380d13',
    '0x650a7354', '0x766a0abb', '0x81c2c92e', '0x92722c85',
    '0xa2bfe8a1', '0xa81a664b', '0xc24b8b70', '0xc76c51a3',
    '0xd192e819', '0xd6990624', '0xf40e3585', '0x106aa070',
```

```
    '0x19a4c116', '0x1e376c08', '0x2748774c', '0x34b0bcb5',
    '0x391c0cb3', '0x4ed8aa4a', '0x5b9cca4f', '0x682e6ff3',
    '0x748f82ee', '0x78a5636f', '0x84c87814', '0x8cc70208',
    '0x90befffa', '0xa4506ceb', '0xbef9a3f7', '0xc67178f2']

    h=['0x6a09e667','0xbb67ae85','0x3c6ef372','0xa54ff53a',
    '0x510e527f','0x9b05688c','0x1f83d9ab','0x5be0cd19']
    k = initializer(k)
    h0, h1, h2, h3, h4, h5, h6, h7 = initializer(h)
    chunks = preprocessMessage(message)
    for chunk in chunks:
        w = chunker(chunk, 32)
        for _ in range(48):
            w.append(32 * [0])
        for i in range(16, 64):
            s0 = XORXOR(rotr(w[i-15], 7), rotr(w[i-15], 18),
            shr(w[i-15], 3) )
            s1 = XORXOR(rotr(w[i-2], 17), rotr(w[i-2], 19),
            shr(w[i-2], 10))
            w[i] = add(add(add(w[i-16], s0), w[i-7]), s1)
        a = h0
        b = h1
        c = h2
        d = h3
        e = h4
        f = h5
        g = h6
        h = h7
        for j in range(64):
            S1 = XORXOR(rotr(e, 6), rotr(e, 11), rotr(e, 25) )
            ch = XOR(AND(e, f), AND(NOT(e), g))
            temp1 = add(add(add(add(h, S1), ch), k[j]), w[j])
            S0 = XORXOR(rotr(a, 2), rotr(a, 13), rotr(a, 22))
            m = XORXOR(AND(a, b), AND(a, c), AND(b, c))
            temp2 = add(S0, m)
            h = g
            g = f
            f = e
            e = add(d, temp1)
            d = c
            c = b
            b = a
            a = add(temp1, temp2)
        h0 = add(h0, a)
        h1 = add(h1, b)
        h2 = add(h2, c)
        h3 = add(h3, d)
        h4 = add(h4, e)
        h5 = add(h5, f)
        h6 = add(h6, g)
```

```
64          h7 = add(h7, h)
65      digest = ''
66      for val in [h0, h1, h2, h3, h4, h5, h6, h7]:
67          digest += b2Tob16(val)
68      return digest
```

Codi 3.41: Codi final de la funció SHA-256 on s'agrupen tots els fragments de codi que hem anat explicant.

En aquest codi, simplement agrupem tots els fragments de codi que hem anat explicant en l'última secció, i així obtenim finalment el codi de la funció SHA-256.

Explicarem, per tal de recopilar tota aquesta secció, el que fa aquest algorisme.

Es defineix la funció `sha256` amb un paràmetre `message`, que representa el missatge a resumir.

Es realitza la inicialització dels valors `k`, `h0`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6` i `h7` utilitzant la funció `initializer` amb els paràmetres `k` i `h`, fixats abans. Aquests valors s'utilitzaran en el càlcul del resum.

Es preprocessa el missatge dividint-lo en blocs mitjançant la funció `preprocessMessage`. Els blocs es guarden en la variable `chunks`.

S'inicia un bucle `for` per cada bloc `chunk` en la llista `chunks`. Dins del bucle, es crea una llista `w` de paraules de 32 bits per al bloc actual utilitzant la funció `chunker`. S'afegeixen 48 paraules de 32 bits amb valor 0 a la llista `w`.

S'inicia un altre bucle `for` que va des de 16 fins a 63. A cada iteració, es calculen les variables `s0` i `s1` utilitzant operacions de rotació i desplaçament cap a la dreta. Es realitzen diverses operacions d'addició i XOR per actualitzar els valors de la llista `w` en aquesta iteració. Es realitza una inicialització dels valors `a`, `b`, `c`, `d`, `e`, `f`, `g` i `h` utilitzant els valors inicials `h0`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6` i `h7`.

S'inicia un altre bucle `for` que va des de 0 fins a 63. A cada iteració, es realitzen operacions d'XOR, rotació i desplaçament cap a la dreta per calcular les variables `S1`, `ch`, `temp1`, `S0`, `m` i `temp2`. Es realitzen operacions d'addició per actualitzar els valors `a`, `b`, `c`, `d`, `e`, `f`, `g` i `h` en aquesta iteració.

Una vegada finalitzats els bucles anteriors, es realitzen operacions d'addició per actualitzar els valors finals `h0`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6` i `h7` amb els valors calculats. Es crea una cadena de caràcters buida anomenada `digest`. Es realitza un bucle `for` per cada valor en la llista `[h0, h1, h2, h3, h4, h5, h6, h7]`.

Dins del bucle, s'agafa cada valor `val` de la llista `[h0, h1, h2, h3, h4, h5, h6, h7]`. Es converteix el valor `val` a una representació hexadecimal utilitzant la funció `b2Tob16`. S'afegeix la representació hexadecimal del valor `val` a la cadena de caràcters `digest`. Una vegada finalitzat el bucle, s'ha completat la generació del resum per a tots els blocs del missatge.

Es retorna la cadena de caràcters `digest`, que conté el resum final calculat pel missatge de entrada.

Resumint, aquest codi implementa l'algoritme SHA-256 per generar un resum criptogràfic d'un missatge donat. S'utilitzen diverses operacions matemàtiques i lògiques per processar el missatge en blocs, actualitzar els valors dels registres i generar el resum final. El resultat final és una cadena de caràcters que representa el resum criptogràfic del missatge original.

A continuació, farem un test per a comprovar que funciona de manera correcta.

3.6.5 Test

Anem a comprovar que el que hem programat funciona de forma correcta, utilitzant els vectors de prova que tenim disponibles en [13]. Realitzarem els següents tests:

```
hash_example = sha('hello world')
print('example: ', hash_example)

#longitud en bits: 0
hash_0 = sha('')
vector_hash_0 =
'e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855'
print('bit size 0: ', hash_0 == vector_hash_0)

#longitud en bits: 24
hash_24 = sha('abc')
vector_hash_24 =
'a7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad'
print('bit size 24: ', hash_24 == vector_hash_24)

#longitud en bits: 448
hash_448 = sha('abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq')
vector_hash_448 =
'248d6a61d20638b8e5c026930c3e6039a33ce45964ff2167f6ecedd419db06c1'
print('bit size 448: ', hash_448 == vector_hash_448)

#longitud en bits: 896
hash_896 = sha('abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmnhi
jklmnoijklmnopjklmnopqklmnopqrlmnopqrsmnopqrstnopqrstu')
vector_hash_896 =
'cf5b16a778af8380036ce59e7b0492370b249b11e8f07a51afac45037afee9d1'
print('bit size 896: ', hash_896 == vector_hash_896)
```

Codi 3.42: Tests per a comprovar el funcionament de la nostra funció SHA-256.

I obtenim:

```
hash_example:
b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9

```

```
bit size 0: True
bit size 24: True
bit size 448: True
bit size 896: True
```

Codi 3.43: Resultats dels tests.

Aleshores, hem escrit en Python el script que executa la funció SHA-256 per a entendre completament el funcionament d'aquesta.

Capítol 4

Cadena de blocs: Blockchain

4.1 Origen i context

Des de la seua publicació en el 2008 [32], *Bitcoin* ha passat de ser una proposta de sistema alternatiu de pagaments, a la criptomoneda més popular fins al moment, amb una capitalització que ha arribat als cinc-cents quaranta mil milions de dòlars [9].

Seguint el seu exemple, ha florit un ample ecosistema de monedes electròniques paral·leles i aplicacions que comparteixen part de l'èxit de bitcoin. Totes elles tenen una cosa en comú: la cadena de blocs. La cadena de blocs o *blockchain* va veure la llum en 2008 amb la publicació d'un article, conegut com a *"white paper"* [32], on s'explicava el protocol que s'usava en bitcoin. Aquest nou concepte formava part d'un sistema per a processar transaccions electròniques de forma que no fora necessària una autoritat central o un sistema de fideïcomís (*escrow*). A principis de 2009, es va publicar el primer client bitcoin, de codi obert, amb el què va començar a funcionar la creació de bitcoin i la base de dades pública i immutable amb les transaccions, coneguda com llibre de registres (*ledger*). La tecnologia per a implementar aquest llibre de registres va ser la cadena de blocs.

Encara que originalment la cadena de blocs va ser creada per a emmagatzemar l'historial de transaccions de bitcoin, amb el pas del temps se li ha vist un gran potencial per a ser aplicada en altres àmbits degut a les propietats que ofereix. La cadena de blocs proporciona una base de dades distribuïda immutable basada en una seqüència creixent de blocs. Aquests blocs, en ser públics, conformen un sistema obert que permet la confiança en base a la transparència i a la solidesa de la tècnica de construcció de la cadena de blocs. El sistema, encara que és obert, és també semi-anònim: els usuaris s'identifiquen amb claus públiques i no amb la seua identitat real.

Una vegada introduït el context de la cadena de blocs, descriurem el funcionament d'aquesta tecnologia en major detall i proporcionarem una descripció més amplia sobre possibles aplicacions de la mateixa en diversos àmbits. A més, en aquest capítol, veurem també la relació de les funcions de resum i la criptografia en la cadena de blocs, ja que

aquests són els elements essencials per a comprendre el funcionament de la cadena de blocs.

4.2 Fonaments tècnics de la cadena de blocs

Descripció bàsica

La cadena de blocs és una base de dades que pot ser compartida per una gran quantitat d'usuaris en forma d'igual a igual (*peer-to-peer*, P2P) i que permet emmagatzemar informació de manera immutable i ordenada. El model de xarxa d'igual a igual, és un model en què els nodes tenen la capacitat d'intercanviar informació directament entre ells, sense la necessitat d'un servidor central. En el cas de bitcoin, la informació afegida a la cadena de blocs és pública i pot ser consultada en qualsevol moment per qualsevol usuari de la xarxa. La informació només pot ser afegida a la cadena de blocs si existeix un acord entre la majoria de les parts. Transcorregut un cert temps, es pot assumir que la informació agregada en un bloc ja no podrà ser modificada (immutabilitat). La creació de nous blocs és realitzada per nodes denominats *miners*. Els miners són nodes de la xarxa que participen en el procés d'escriptura de dades en la cadena de blocs a canvi d'una recompensa econòmica. La validesa de l'escriptura d'un bloc per part d'un miner és revisada i acordada tàcitament per la resta de participants.

El procés que permet aconseguir un consens amb garanties entre els miners de la cadena de blocs per a l'ordre d'escriptura de blocs és la denominada prova de treball (*Proof-of-work, PoW*). En concret, perquè un bloc siga acceptat, el miner ha de ser el primer en completar una prova de treball per al següent bloc de la cadena de blocs. La prova de treball és un trencaclosques matemàtic de dificultat ajustable. En particular, la prova de treball consisteix en trobar un paràmetre aleatori únic (*nonce*) que aconseguisca que en fer el resum sobre tot el bloc (inclòs el paràmetre) s'obtinga un valor inferior a la dificultat actual establida per la xarxa. Dit d'una altra forma, es tracta de trobar un paràmetre que obtinga un valor de resum del bloc amb un determinat nombre de zeros a l'inici. A causa de les característiques de la funció de resum, no és possible calcular aquests valors analíticament, és a dir, per a aconseguir un bloc vàlid, el miner ha de recórrer a la força bruta: provar valors del paràmetre fins a trobar-ne un vàlid. El procés de provar valors o força bruta és un procés computacionalment costós, per aquest motiu aquest mecanisme es coneix com a prova de treball.

La prova de treball fa que la creació de blocs amb la intenció de desestabilitzar el consens tinga un cost alt per a l'atacant. D'altra banda, la dificultat d'aquest trencaclosques criptogràfic és fàcilment ajustable: es pot incrementar la dificultat augmentant el nombre de zeros necessaris per a completar la prova de treball o disminuir-la reduint aquest nombre de zeros. En particular, en bitcoin la dificultat es reajusta cada 2016 blocs (que equivalen a catorze dies), amb la condició que la creació de nous blocs tinga una freqüència aproximada

d'un bloc cada deu minuts.

Estructura dels blocs

La cadena de blocs emmagatzema una gran quantitat de dades i a més la seua grandària és creixent amb el temps, ja que en la mateixa només s'afig informació. Per tant, és aconsellable disposar d'algun mecanisme que permeta una consulta a la cadena de blocs eficient, és a dir, que permeta realitzar consultes sense haver de descarregar tota la informació emmagatzemada. Per a aquest propòsit, en la cadena de blocs de bitcoin, es proposa utilitzar un arbre de resum de Merkle [47].

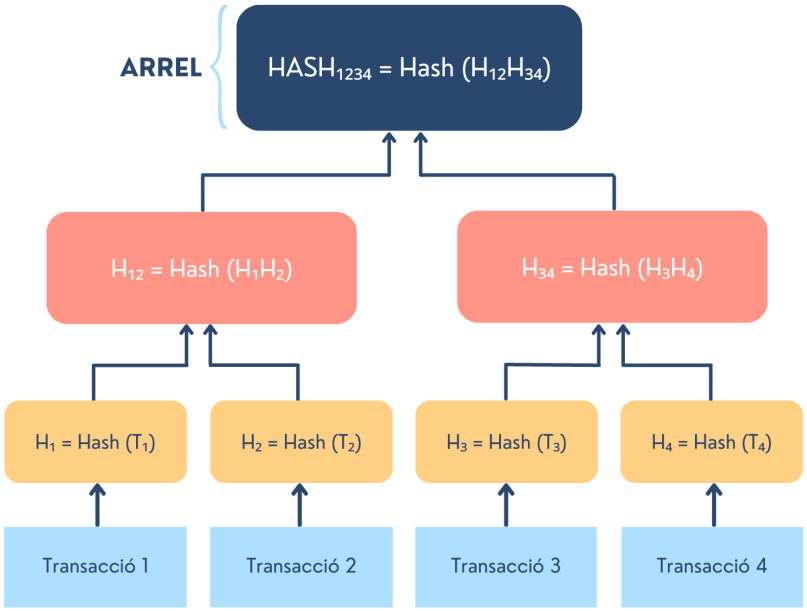

Figura 4.1: Arbre de resum de Merkle. Font: elaboració pròpia.

Com es mostra en la Figura 4.1, l'arbre de resum de Merkle permet emmagatzemar diverses peces d'informació independent (en el cas de bitcoin són transaccions econòmiques) en les fulles d'una estructura en arbre. Per a formar l'arbre, es fa un resum de la informació continguda en cada node fulla. A continuació, per a generar els nodes de cada nivell superior de l'arbre es concatenen diversos valors de resum del nivell inferior (dos valors si l'arbre és binari) i se li aplica la funció de resum a aquesta concatenació (veure Figura 4.1). Repetint aquest procés s'arriba a un nivell on hi ha un node només, denominat l'arrel de l'arbre.

L'avantatge d'aquesta estructura en arbre és que podrem consultar la presència en aquest arbre de les dades d'un cert node/fulla de forma autenticada i sense haver de

disposar de tota la informació que emmagatzema l'arbre. En particular, es pot consultar de forma autenticada qualsevol contingut de l'arbre amb una quantitat de valors de resum proporcional al logaritme del nombre de nodes de l'arbre.

Això és perquè per a validar un contingut únicament cal proporcionar els nodes adjacents en cada nivell i el node arrel (que té contribució de totes les dades emmagatzemades en les fulles) autenticat. Llavors, per a validar un contingut es calcula el valor arrel a partir dels nodes adjacents proporcionats i es comprova que coincideix amb el valor arrel autenticat. L'estructura és segura perquè no es pot generar un conjunt de nodes adjacents a voluntat que done com a resultat el valor del node arrel autenticat.

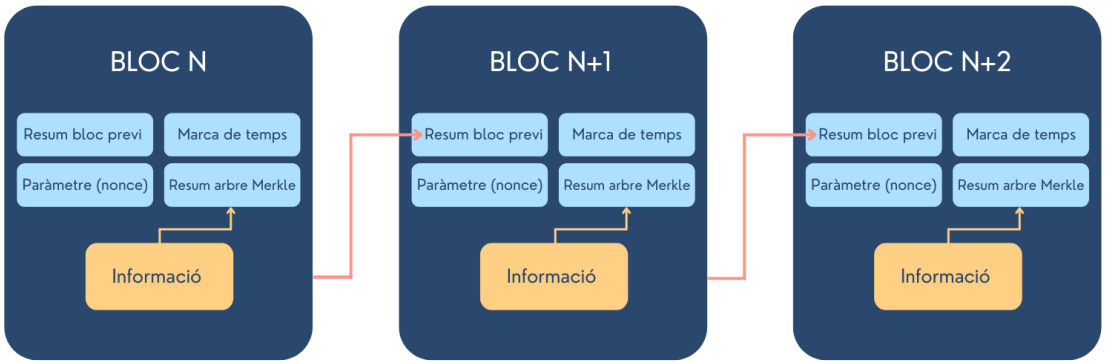

Figura 4.2: Estructura dels blocs de la cadena de blocs de Bitcoin. Font: elaboració pròpia.

D'altra banda, en la Figura 4.2, es mostra la informació que conté cada bloc en la cadena de blocs de bitcoin:

- El valor de resum del bloc previ: aquest valor permet que els blocs queden vinculats seqüencialment formant una cadena.
- Marca de temps (*timestamp*): aquesta marca de temps permet identificar l'instant en el qual va ser creat el bloc.
- El valor del paràmetre (*nonce*): aquest és el valor imposat per força bruta en el procés de minat.
- El valor de l'arrel de l'arbre de Merkle de les transaccions (*root hash*): aquest valor de resum permet referenciar tota la informació del bloc. Com s'ha comentat, amb el valor de l'arrel de l'arbre i certs valors addicionals es poden realitzar consultes sobre la informació continguda en un bloc de manera eficient i segura.
- Informació: finalment, el bloc conté la informació en sí.

En el cas de bitcoin, la informació continguda en els blocs són les transaccions realitzades amb la criptomoneda. En particular, una de les transaccions que ha de ser afegida és la que adjudica al creador del bloc una recompensa (*halving*) per haver-lo minat. La recompensa per la creació d'un bloc es divideix entre dos cada 210.000 blocs que equivalen a quatre anys. Al seu inici en 2009, la recompensa estava fixada en 50 bitcoins i actualment, en 2023 aquesta és de 6,25 bitcoins [50].

Propietats fonamentals de la cadena de blocs

La creació d'una cadena de blocs robusta ha de garantir dues propietats fonamentals:

- Disponibilitat: assegura que una transacció honesta que ha sigut emesa acabe sent afegida a la cadena de blocs, evitant que es produïsca una denegació de servei (*Denial of Service, DoS*) per part de nodes corruptes.

- Persistència: quan un node dona una transacció com a estable, la resta de nodes, si són honestos, validaran aquesta com a estable fent-la immutable.

Per a complir amb la propietat de disponibilitat, la cadena de blocs de bitcoin (i també la de molts altres sistemes) implementa una xarxa de nodes interconnectats on aquests nodes interaccionen com a iguals (xarxa *peer-to-peer*). La xarxa d'igual a igual de bitcoin és descentralitzada, és a dir, qualsevol usuari que desitge pot contribuir. Unes altres cadenes de blocs utilitzen un sistema amb llista blanca (*white-list*) [14].

Una cadena de blocs amb una llista blanca fa referència a un sistema en què s'estableix una llista d'entitats o adreces preaprovades i autoritzades per a participar en la validació de transaccions i la creació de nous blocs en la cadena. En aquest context, la llista blanca actua com a mecanisme de control d'accés, on només els participants inclosos en la llista tenen el permís per a realitzar certes accions dins de la cadena de blocs. Aquestes accions poden incloure la verificació de transaccions, la validació de blocs i la participació en el consens del sistema. La implementació d'una llista blanca en una cadena de blocs pot tenir diferents propòsits, com restringir la participació només a entitats de confiança, garantir la seguretat i la integritat de la xarxa o complir amb regulacions específiques.

En qualsevol cas, els nodes que formen part de la xarxa d'igual a igual disposen, cadascun d'ells, d'una còpia de la cadena de blocs. La gran quantitat de còpies de la cadena de blocs proporciona una gran disponibilitat i robustesa. En particular, en la inserció de blocs en la cadena de blocs mitjançant la xarxa d'igual a igual es distingeixen diferents estats per a la informació de bloc que està sent processada:

- Informació candidata a ser afegida: és informació que els nodes han enviat a la resta de nodes mitjançant la xarxa d'igual a igual però que encara no ha sigut validada en cap bloc.

- Informació confirmada: és informació validada per la xarxa i es procedeix a afegir-la al pròxim bloc.
- Informació estable: és informació que forma part de la cadena de blocs de manera immutable.

En la pràctica, es considera que es compleix la propietat de persistència per a un cert bloc quan existeixen sis blocs minats amb posterioritat al mateix (aproximadament 1 hora d'espera). Mitjançant aquesta senzilla regla s'assegura que una transacció serà immutable amb un risc inferior al 0,1%, suposant que algun atacant tinga més del 10% de la capacitat total del resum de la xarxa [14].

Generació dels blocs en la cadena de blocs

La generació de blocs en la cadena de blocs es realitza de forma descentralitzada. La clau per a aquesta descentralització és que s'arribe a un acord sobre quina informació es guarda en ella. Per a això, és necessari aconseguir un consens distribuït que permeta que els nodes honestos tinguen la capacitat de generar la informació vàlida conjuntament i així evitar que nodes maliciosos puguen guardar informació no desitjada. En el cas del bitcoin, aquest procés permet resoldre el problema de la doble despesa (un usuari gastant dues vegades els mateixos diners) que fins a eixe moment semblava un obstacle insalvable per a dur a terme l'ús de monedes digitals [14].

Anem a explicar el procés que segueix bitcoin per a afegir blocs a la cadena de blocs. En primer lloc, un usuari ha de convertir-se en node dins del sistema per a poder escoltar i emetre noves transaccions (informació). En segon lloc, si l'usuari desitja convertir-se en miner i crear nous blocs ha de competir contra la resta de miners en la xarxa per a resoldre el trencaclosques criptogràfic i així ser qui escriga el nou bloc en la cadena de blocs oficial.

El procés d'autenticació de les transaccions es basa en criptografia asimètrica. Cada compte d'usuari de Bitcoin posseeix dues claus relacionades matemàticament: una pública (identificador de l'usuari en la xarxa, coneguda per tots) i una privada (secreta, coneguda per l'usuari). La clau privada s'usa per a signar les transaccions emeses per l'usuari; aquest especifica les quantitats de moneda a transferir i les claus públiques de destinació. La xarxa i la resta d'usuaris, usant la clau pública de l'emissor, poden obtindre una prova matemàtica que la transacció va ser efectivament signada per aquest usuari i per ningú més, ja que ningú més té la seua clau privada.

Les noves transaccions emeses són validades pels nodes més pròxims a l'emissor, descartant totes les transaccions invàlides i propagant a la resta de nodes les transaccions vàlides, és a dir, aquelles que compleixen amb les especificacions de la xarxa. Posteriorment, es procedeix a afegir les noves transaccions a la cadena de blocs. Aquest procés de confirmació de dades es duu a terme en bitcoin mitjançant el procés de minat de la prova de treball (encara que en l'actualitat estan sorgint altres mecanismes alternatius a la prova de treball com es discuteix en la següent secció).

Finalment, els nodes comproven que, en el nou bloc creat (minat), totes les transaccions són vàlides i que el bloc està correctament vinculat amb el seu predecessor, és a dir, que conté el valor del resum del bloc anterior en la seua capçalera. En cas afirmatiu el bloc és afegit a la cadena de blocs incrementant així la cadena. El procés es repeteix generant una nova ronda de minat amb les noves transaccions emeses que encara no hagen sigut agregades en cap bloc anterior de la cadena de blocs. Si el bloc és invàlid, és descartat, i la resta de nodes segueixen el procés de minat fins a trobar un bloc vàlid.

Alternatives a la prova de treball

Existeixen alternatives a la creació de blocs mitjançant el minat de la prova de treball. La més utilitzada és la prova de participació (*Proof-of-Stake, PoS*) [5]. La seua principal diferència respecte a la prova de treball és que la creació de blocs és duta a terme per nodes que ja posseeixen la criptomoneda subjacent (*stakeholders*), la qual cosa per si mateix aconsegueix que no hi haja interès a corrompre el correcte funcionament del sistema. Aquest protocol assigna una probabilitat de crear blocs proporcional a la quantitat de moneda que posseeix cada node. Així, aquells usuaris que tinguen més quantitat de la criptomoneda seran els que creen blocs amb més freqüència.

El principal avantatge respecte a la prova de treball és que la creació de blocs no requereix potència de computació per al seu correcte funcionament el que, per si mateix, és un gran avantatge. En l'actualitat el minat de bitcoins genera una gran despesa d'energia amb l'únic objectiu de mantindre robust el sistema. Com a referència cal destacar que l'any 2014 la prova de treball de bitcoin ja consumia tanta energia elèctrica com ho feia tota Irlanda [34].

El principal desavantatge és que el sistema podria arribar a no ser descentralitzat. Per a evitar que això passe han sorgit diferents mètodes que solucionen el problema [29]. En ells es pondera la participació de cada propietari juntament amb la duració que fa que aquest el posseïsca amb la condició d'assignar una probabilitat per a la creació del pròxim bloc. Diversos exemples d'implementacions amb la prova de participació són les següents criptomonedes: Ethereum [18], Cardano [8] i Solana [44].

4.3 Criptomonedes: aplicació principal

Una cadena de blocs pot dissenyar-se com una base de dades veritablement descentralitzada i sense una autoritat central. Pot, per tant, servir com a centre d'intercanvis de confiança entre múltiples entitats sense que unes hagen de confiar en les altres, ni tan sols en un intermediari. Això representa una vertadera revolució: en els sistemes d'intercanvi ha existit històricament la necessitat d'un intermediari de confiança de totes les parts. Per exemple, quan algú compra un objecte de segona mà en una plataforma en línia, aquesta s'encarrega de verificar, a canvi d'un percentatge, que la transacció s'ha realitzat amb èxit

i de compensar a les parts en cas de frau. O, anàlogament, quan un banc acorda un préstec amb una empresa, l'acord s'explicita en un contracte amb validesa legal: l'Estat, amb els seus poders coercitius, és el garant del compliment del contracte. Res d'això és necessari en la cadena de blocs.

En el cas de les monedes clàssiques, existeix a més una autoritat monetària: els bancs centrals. Aquests són els únics amb la capacitat d'emetre més unitats monetàries, es troben en la base de la cadena creditícia i tenen certa potestat per a determinar el preu base del crèdit. Aquestes decisions es prenen habitualment responent a criteris de política macroeconòmica com ara: estabilitat a llarg termini, objectius d'inflació, relació importacions/exportacions, etc.

Les criptomonedes basades en les cadenes de blocs eliminen també la necessitat d'una autoritat central. El criteri d'emissió de noves unitats monetàries es troba prefixat. En el cas de Bitcoin, per exemple, s'emet nova moneda cada vegada que es mina un bloc (cada 10 minuts aproximadament) i es posa en possessió del node que l'ha minat. La quantitat decreix aproximadament cada quatre anys i el sistema està dissenyat per a arribar a un total de 21 milions de bitcoins en 2040. Desapareix, per tant, la incertesa associada a aquesta mena de decisions polítiques. S'elimina, a més, la possibilitat de que els Estats usen els bancs centrals per a beneficiar a uns sectors o perjudicar a uns altres.

Existeixen desenes de criptomonedes. Totes elles comparteixen la seua utilitat com a sistema de pagament. Algunes utilitzen una cadena de blocs pròpia i unes altres funcionen damunt de la cadena de blocs de bitcoin. El seu funcionament és bastant heterogeni i totes elles pretenen aportar alguna millora respecte a bitcoin.

Probablement, la més prometedora entre les alternatives és Ethereum, que és la segona criptomoneda amb major capitalització. Ethereum compta amb una cadena de blocs pròpia, és a dir, diferent de bitcoin i a més, basada en la prova de participació. Més enllà del seu funcionament com criptomoneda, l'aportació principal de Ethereum són els contractes intel·ligents (*Smart Contracts*) [6]. Els contractes intel·ligents són xicotets codis (*scripts*) auto-executables que resideixen en la cadena de blocs i que permeten automatitzar gran quantitat de processos comercials d'una forma segura i transparent per a tots els participants. Els exemples del següent apartat representen alguns usos potencials d'aquest tipus de funcionalitat.

4.4 Altres aplicacions

Hui en dia bitcoin és, sense dubte, la realització pràctica de la tecnologia de la cadena de blocs més coneguda. No obstant això, la llista de possibles casos d'ús és molt més llarga i potencialment més revolucionària que la criptomoneda.

Una de les principals aplicacions de la cadena de blocs és el tractament de dades massives, ja que podem trobar una primera relació entre ells: la necessitat d'assegurar un entorn de pagaments legal i lliure de fraus ha dut al desenvolupament de ferramentes d'anàlisi

basades en tècniques de tractament de dades massives per a processar la gran quantitat de dades representades en la cadena de blocs [14]. Per tant, l'anterior, és un possible ús del tractament de dades massives per millorar els processos d'inserció de dades en la cadena de blocs.

A més, un altre cas d'ús per a la cadena de blocs es podria donar en l'àmbit de l'Internet de les coses (*Internet of Things, IoT*). Un exemple és la distribució segura i fiable de *firmware* o dispositius IoT mitjançant un sistema d'arxius d'igual a igual sobre la cadena de blocs. És més, actuen simultàniament com a clients i servidors respecte als demés nodes de la xarxa. En aquest cas d'ús, es podria utilitzar la cadena de blocs per a emmagatzemar actualitzacions de *firmware* d'una forma descentralitzada i segura [14]. A continuació, parlarem de les seues principals aplicacions de forma més detallada.

4.4.1 Blockchain i l'Internet de les coses (IoT)

Sistema de distribució de firmware

A mesura que l'Internet de les coses (*Internet of Things, IoT*) vaja convertint-se en realitat, el nombre de dispositius connectats a la xarxa creixerà exponencialment. Ara, a més de computadors i equips de xarxa, es van connectant a Internet electrodomèstics, dispositius de vigilància i seguretat, sensors de tot tipus, etc. Sens dubte aquest nou ecosistema ofereix una flexibilitat sense precedents. No obstant això, aquest paradigma té també alguns reptes a resoldre. Entre ells, destaca en gran manera l'àmbit de la seguretat. Per exemple, diverses fonts afirmen que l'atac als servidors de DNS del proveïdor DynDNS, que van deixar sense servei durant hores a gegants com Twitter o Facebook, es van realitzar controlant remotament un gran nombre de dispositius que utilitzaven l'Internet de les coses [36].

Un dels primers problemes de seguretat que apareix en l'entorn de l'internet de les coses és el nivell de supervisió dels dispositius. En aquest sentit, en l'entorn de l'Internet de les coses, molts dispositius no estan supervisats amb els nivells usats en el món de la computació (ordinadors personals o telèfons intel·ligents). A més, pel seu baix cost, molts d'aquests dispositius no sempre reben actualitzacions amb una elevada freqüència per part dels fabricants (o fins i tot es queden sense actualitzar en algun punt).

Al mateix temps, per al consumidor, existeix certa desconfiança en dispositius que es comuniquen amb el seu fabricant sense la supervisió de l'usuari final i seria molt millor un enfocament de seguretat més transparent. Aquest escenari, podria solucionar-se amb una cadena de blocs. En aquest cas, s'aprofitaria tant la seua característica de sistema distribuït i la seua transparència, com la seua robustesa i fiabilitat. Els dispositius consultarien la cadena de blocs per a esbrinar si el seu microprogramari està actualitzat. En cas que no ho estiga, demanarien a altres nodes que els manen la nova versió. Una vegada rebuda, podrien usar el codi de la cadena de blocs per a comprovar que el microprogramari no ha

sigut alterat de cap manera, evitant així les intrusions. Aquest enfocament, una vegada implementat, resultaria bastant més barat per al fabricant que només hauria de manar l'actualització a uns pocs nodes i deixar que l'actualització es propague.

Mercat de serveis entre dispositius

Si a una criptomoneda se li afig capacitat per a contractes intel·ligents, l'aplicabilitat en l'Internet de les coses es dispara. En el cas anterior, per exemple, els dispositius que manen el nou microprogramari podria tindre una xicoteta remuneració, ja que consumeixen recursos (electricitat i amplada de banda). La idea és, en general, que cada aparell connectat a la cadena de blocs tinga el seu propi "compte bancari" i que oferisca els seus serveis en l'ecosistema. Filecoin [20], per exemple, permet als seus nodes llogar espai d'emmagatzematge.

En el mercat de l'energia, aquesta arquitectura resulta especialment útil. L'entrada de les renovables com a factor de pes i la proliferació de sistemes d'emmagatzematge (bancs de bateries i bateries a casa) ha tornat l'ecosistema més heterogeni del que ja era. Hi ha productors que produeixen de manera constant (per exemple, una central nuclear), uns altres només durant el dia (com les plaques solars) i uns altres només quan fa vent (com són els generadors eòlics). La demanda té hores molt intensives i unes altres de molt baix consum. Una bateria connectada a la xarxa, per exemple, podria comprar energia en les hores de baix preu per a després vendre-la en les hores puntes, segons les normes definides pel propietari. TransActive Grid [24], per exemple, està experimentant amb aquest concepte de mercat entre aparells. Encara que moltes d'aquestes funcionalitats ja es realitzen actualment, la tecnologia de la cadena de blocs permetria fer-ho en un ecosistema unificat, segur, versàtil, transparent i barat.

Sistema de seguiment de transports

En un enviament internacional de mercaderies participen normalment diverses empreses, ja que s'utilitzen diversos mitjans de transport. Totes elles tenen les seues bases de dades independents, on actualitzen l'estat de l'enviament en funció de la informació proporcionada per les altres companyies o pels seus agents.

La cadena de blocs té ací una clara aplicabilitat que permetria fer el sistema més simple, més transparent i menys costós. Per a començar, la base de dades (la mateixa cadena de blocs) seria compartida per tots els intermediaris i pel remitent i destinatari, reduint els costos. No seria per a això necessària la confiança entre ells en general. En arribar el contenidor en qüestió a un cert port, s'afegiria una actualització a la base de dades. En estar totes les actualitzacions signades amb les claus privades referides al lliurament i a qui el recull; i amb la clau del contenidor, aquesta actualització actuaria com a prova criptogràfica de que el contenidor es troba ara en possessió de l'administrador del node. A més, el sistema inclou marques de temps (*timestamps*) per a fer el seguiment. La

transparència i fiabilitat del concepte ajudaria sense dubte a la resolució de disputes entre els participants.

Si a aquest enfocament innovador se li afig l'Internet de les coses, es pot guanyar encara més eficiència. Els contenidors i els llocs d'intercanvi poden tindre dispositius incorporats que automatitzaren totalment el procés, disminuint els costos encara més i reduint les probabilitats d'error i de frau.

4.4.2 Prova d'existència

Una de les característiques més notables de la cadena de blocs, si no la més notable, és la seua immutabilitat: una vegada la informació s'ha afegit a la base de dades distribuïda, i afegits uns pocs blocs darrere, la probabilitat que siga modificada és, a efectes pràctics, zero.

Aquesta propietat, sense equivalent en el món digital ni en el món real, té òbvies aplicacions. Tradicionalment, si algú vol provar públicament ser l'autor d'una informació, siga un disseny tecnològic o una cançó, acudeix a una oficina de patents o similar. Això és, com passava amb les transaccions, una entitat de confiança de totes les parts que ofereix, bàsicament, la garantia de no modificar res i de no revelar res (a excepció d' allò necessari per a una potencial batalla legal, sempre autoritzada pel propietari). La tecnologia de la cadena de blocs pot substituir també a aquest intermediari.

Per exemple, en la prova d'existència [37] s'ha implementat un servei on qualsevol pot generar una prova de que una informació existia en un moment determinat en el temps i emmagatzemar-la en la cadena de blocs. El funcionament és senzill: es calcula un valor de resum per al document en qüestió. Del valor de resum no es pot deduir el document, però només el posseïdor del document pot haver generat el valor de resum. Després es genera una transacció especial que permet emmagatzemar aquest codi en la cadena de blocs. La transacció s'envia i una vegada és afegida en un bloc, queda ací per sempre. El bloc conté, a més, data i hora, tan immutables com la resta, completant així la prova segura, pública i verificable d'existència. A més, el seu cost és ínfim, sobretot comparat amb els costos de les patents convencionals.

Pel que fa al propietari del document, les seues dades poden trobar-se en el mateix document. A causa de les propietats de la funció de resum, qualsevol canvi, per xicotet que siga, resulta en un valor de resum totalment diferent. El sistema té ací una altra utilitat derivada: comprovació d'integritat de documents. Un usuari pot comprovar mitjançant la mateixa pàgina web que un cert document original emmagatzemat en la cadena de blocs és idèntic al què posseeix.

4.4.3 Seguretat en el tractament de dades massiu

Diverses tècniques de tractament de dades massiu (*big data*) s'usen actualment per a analitzar la cadena de blocs i incrementar els seus nivells de seguretat. Aquestes tècniques

permeten deduir les identitats dels nodes en les criptomonedes, detectar fraus i controlar els fluxos reals de diners.

La relació inversa, no obstant això, és encara més prometedora: utilitzar la tecnologia de la cadena de blocs per a donar seguretat i verificabilitat a entorns empresarials del tractament massiu de dades. Amb l'explosió del tractament massiu de dades, pràcticament tota empresa amb un mínim de clients està interessada a traure el màxim partit a les seues dades per a així mantindre's competitiva. Es tracta de dades que habitualment provenen de diverses fonts, en diversos formats, i són utilitzades en diversos processos per diferents departaments de l'empresa. Els perills d'aquests sistemes resulten bastant evidents: manipulació de les dades per part de treballadors interns, proveïdors maliciosos, corrupció de les dades, errades d'emmagatzematge, ús defectuós, incompliment de legislacions respecte a les dades personals i un llarg etcètera.

En aquest context, la cadena de blocs té molt a aportar: transparència, verificabilitat, portabilitat i escalabilitat. Mitjançant la cadena de blocs, cada nova entrada en les dades, cada canvi, cada extracció per al seu ús o cada visualització es podria realitzar fent servir un registre transparent i segur. A més, les dades podrien anar acompanyades de proves d'integritat de baix nivell o, fins i tot, en el cas de l'extracció, de signatures concretes que possibiliten la seua traçabilitat.

Aquests entorns permeten un grau de seguretat i verificabilitat suficient per a complir amb regulacions bastant restrictives alhora que són intrínsecament distribuïts, escalables i interoperables. Els requisits legals pel que fa a la retenció de dades deixen de ser un problema perquè està en la mateixa naturalesa de la cadena de blocs el poder deduir l'estat de la base de dades en qualsevol punt del temps. L'eina Hadoop Big Data Lakes de la empresa Guardtime és un exemple d'aquesta tipus de tecnologia [3].

Resum

Per fer una conclusió resumida de les aplicacions de la cadena de blocs, diríem que permet implementar una base de dades distribuïda, pública i immutable basada en una seqüència creixent de blocs. Aquesta base de dades proporciona de manera intrínseca tolerància a errades en nodes, robustesa enfront de manipulació, i transparència per ser pública. Els usos d'aquesta tecnologia són potencialment immensos i per això es considera com una de les tecnologies amb més potencial disruptiu dels últims anys.

La possibilitat de tindre una base de dades distribuïda i immutable a posteriori té una infinitat d'utilitats pràctiques que només comencen a sorgir. Les criptomonedes han sigut la seua primera aplicació d'èxit a causa de les necessitats de seguretat i transparència dels sistemes de pagament i a la possibilitat d'eliminar intermediaris. En el futur, no obstant això, és possible que trobem sistemes de cadena de blocs en una infinitat de contextos i sistemes. En aquest sentit, i com a punt de partida, es poden considerar els casos d'ús en escenaris com a l'Internet de les coses i el tractament de dades massives esmentats en aquesta secció.

4.5 Problemes de la cadena de blocs i possibles solucions

La tecnologia de la cadena de blocs ha despertat un gran interès i s'ha posicionat com a una innovació disruptiva en diversos sectors. No obstant això, com qualsevol tecnologia emergent, també enfronta reptes i problemes que han de ser abordats per a la seua adopció i ús generalitzat.

En aquesta secció, explorarem els problemes més destacats associats amb la cadena de blocs i examinarem les possibles solucions i alternatives. A més, abordarem la importància de garantir la seguretat de la cadena de blocs, ja que qualsevol vulnerabilitat pot tenir conseqüències significatives en termes d'integritat i confiança en el sistema. També explorarem els desafiaments relacionats amb la regulació i el marc legal, ja que la naturalesa descentralitzada i global de la cadena de blocs planteja preguntes sobre com establir normes adequades.

Per tant, a continuació parlarem d'alguns dels problemes comuns associats amb la cadena de blocs, junt a possibles alternatives que s'han anat desenvolupant recentment per a cadascun dels problemes que anem a esmentar:

1. **Escalabilitat** [42]: la cadena de blocs pot enfrontar problemes d'escalabilitat a causa de la seua naturalesa descentralitzada i la necessitat que cada node valide i emmagatzeme totes les transaccions. Això pot limitar la quantitat de transaccions que es poden processar en un període de temps donat. S'estan explorant diverses solucions, com l'ús de cadenes laterals (*sidechains*), fragmentació (*sharding*) i algorismes de consens millorats, com la prova de participació (*Proof of Stake, PoS*) [5], que permeten un major rendiment i capacitat d'escalabilitat.

2. **Consum d'energia** [43]: moltes cadenes de blocs, com ara Bitcoin, requereixen una gran quantitat d'energia per al procés de mineria, el que pot tindre un impacte negatiu en el medi ambient. S'estan investigant i desenvolupant algorismes de consens més eficients energèticament, com la prova de participació, que no requereixen l'ús intensiu de recursos computacionals i energia.

3. **Privacitat** [26]: encara que la cadena de blocs garanteix la transparència de les transaccions, també pot plantejar reptes en termes de privacitat. Les transaccions en la cadena de blocs són públicament visibles i poden revelar informació sensible sobre les parts involucrades. S'estan desenvolupant solucions de privacitat millorades, com a tecnologies de signatura d'anell (*ring signatures*) i barreja de transaccions (*transaction mixing*), que permeten mantindre la privacitat de les transaccions mentre es manté la integritat de la cadena de blocs.

4. **Costos i complexitat** [29]: implementar i mantenir una cadena de blocs pot ser costós i requereix coneixements tècnics especialitzats. A més, les actualitzacions i millores en la cadena de blocs poden ser complicades a causa de la necessitat de

consens entre els participants. S'estan creant eines i plataformes més accessibles que simplifiquen el desenvolupament i la implementació d'aplicacions basades en la cadena de blocs, la qual cosa redueix els costos i la complexitat associats.

5. **Seguretat** [26]: encara que la cadena de blocs es considera segura, no està exempta de riscos. Hi ha possibles vulnerabilitats en el codi, així com la possibilitat d'atacs de majoria i atacs del 51%. S'estan duent a terme investigacions i auditories de seguretat exhaustives per a identificar i abordar possibles vulnerabilitats en els protocols i en les implementacions de la cadena de blocs. A més, es promou una major educació i consciència sobre les millors pràctiques de seguretat en el desenvolupament i ús de la cadena de blocs.

6. **Regulació i marc legal** [10]: l'àmbit legal i regulador al voltant de la cadena de blocs encara està en desenvolupament i pot variar segons la jurisdicció. Això pot generar incertesa en termes de compliment normatiu i protecció del consumidor. S'estan realitzant esforços per a desenvolupar marcs reguladors clars i consistents per a la cadena de blocs, la qual cosa brinda major certesa i protecció tant als usuaris com a les empreses. Això implica la col·laboració entre governs, institucions financeres i la comunitat de la cadena de blocs per a establir estàndards i polítiques adequades.

Aquests són només alguns dels problemes associats amb la cadena de blocs. A mesura que la tecnologia avança, és important abordar i resoldre aquests reptes per a maximitzar el potencial de la cadena de blocs en diverses indústries. A més, és important destacar que aquestes solucions i alternatives estan en constant evolució i desenvolupament, ja que la tecnologia de la cadena de blocs continua avançant. Cada problema pot requerir enfocaments específics i adaptats al seu context particular.

4.5.1 Criptomonedes que aborden els problemes de la cadena de blocs

Existeixen diverses criptomonedes que han sorgit amb l'objectiu d'abordar els problemes i desafiaments associats amb la cadena de blocs i oferir solucions innovadores. Algunes d'aquestes monedes que implementen aquestes solucions són:

- **Ethereum (ETH)** [18]: Ethereum és una plataforma de cadena de blocs que permet la creació de contractes intel·ligents i aplicacions descentralitzades. Busca resoldre els problemes d'escalabilitat i privacitat mitjançant l'ús de la tecnologia de capa 2 i l'implementació de millores en el seu protocol.

- **Cardano (ADA)** [8]: Cardano és una plataforma de cadena de blocs que s'enfoca en l'escalabilitat, la seguretat i la sostenibilitat. Utilitza un enfocament de capes i utilitza un algorisme de consens anomenat *Ouroboros*, que busca ser eficient i segur.

- **Ripple (XRP)** [40]: Ripple és una criptomoneda i una xarxa de pagaments que se centra en la velocitat i l'escalabilitat de les transaccions. Utilitza el seu propi protocol de consens anomenat *Ripple Protocol Consensus Algorithm (RPCA)* per a validar i confirmar les transaccions de manera eficient.

- **IOTA (MIOTA)** [27]: IOTA és una criptomoneda dissenyada específicament per a l'Internet de les Coses. Es basa en una arquitectura anomenada *Tangle*, que resol el problema d'escalabilitat en eliminar la necessitat de miners i permetre transaccions paral·leles.

- **Tezos (XTZ**) [46]: Tezos és una plataforma de cadena de blocs que s'autogoverna i permet l'actualització contínua del seu protocol. El seu enfocament en la governança descentralitzada i l'escalabilitat modular cerca solucionar els problemes de governança i actualització en altres cadenes de blocs.

Aquestes són només algunes de les criptomonedes noves que estan implementant solucions innovadores per a abordar els problemes de la cadena de blocs. Cadascuna d'elles té el seu enfocament únic i cerca oferir beneficis específics en termes d'escalabilitat, privacitat i altres aspectes clau de la tecnologia de la cadena de blocs.

Capítol 5

Conclusions

En aquest treball, hem estudiat diversos aspectes sobre la teoria de grups i criptografia, així com la relació que aquests dos presenten entre ells, amb l'objectiu d'endinsar-se en els fonaments tècnics de les funcions de resum. Aquests conceptes ens han donat les bases per a explicar detalladament el funcionament de la cadena de blocs.

A més, per assolir les idees mencionades, hem tractat d'explicar els conceptes des de la seua versió més senzilla per a després poder aprofundir en ells. Sempre que ha sigut possible, l'explicació ha anat acompanyada amb codi de Python, per tal d'il·lustrar d'una manera visual i pràctica com s'implementen les funcions. Gràcies als coneixements adquirits en el grau de Matemàtiques (en particular en les assignatures de Matemàtica bàsica, Estructures algebraiques i Informàtica), hem pogut analitzar i estudiar aquest camp amb la profunditat necessària.

En aquest estudi hem evidenciat que el futur de la cadena de blocs és extremadament prometedor i té una importància creixent en diversos àmbits. Hem vist que la cadena de blocs té la capacitat de transformar les indústries existents, incloent la banca, el comerç, la logística i moltes altres. Amb el seu potencial per aportar transparència, seguretat i eficiència als processos, la cadena de blocs pot canviar la manera en què es gestionen les transaccions i s'intercanvien les dades.

D'altra banda, permet eliminar la necessitat d'intermediaris, ja que les transaccions es verifiquen de forma descentralitzada pels nodes de la xarxa. Això pot afavorir la democratització de l'accés als serveis financers, el control de les dades personals i altres aspectes importants. Per últim, una de les característiques més importants de la cadena de blocs és la seua transparència inherent. Les transaccions registrades a la cadena de blocs són accessibles a tots els participants, el que ajuda a generar confiança i eliminar la necessitat de dependre de tercers.

Concluint, la capacitat de la cadena de blocs per aportar transparència, seguretat i eficiència als processos està transformant les indústries i obrint noves oportunitats per a la innovació. Amb el continu desenvolupament i la investigació, la cadena de blocs té

el potencial de canviar la manera en què interactuem i realitzem transaccions en el món digital.

Bibliografia

[1] Ankan B. (2022). *Merkle Tree*. Recuperat de `https://github.com/akbng/merkle-tree`.

[2] Ballester A., Esteban R., Pérez M.D. (2019). *Estructures Algebraiques*. Apunts de la assignatura de segon curs "Estructures Algebraiques" de l'Universitat de València.

[3] Birender S. (2017). *Hadoop Data Lake - To be or not to be, centralized?* Recuperat de `https://guardtime.com`.

[4] browserling.com (2020). *Whirlpool Hash Generator*. Recuperat de `https://www.browserling.com/tools/whirlpool-hash`.

[5] Buterin V. (2013). *What proof of stake is and why it matters*. Recuperat de `https://bitcoinmagazine.com/culture/what-proof-of-stake-is-and-why-it-matters-1377531463`.

[6] Buterin, V. (2014). *A next-generation smart contract and decentralized application platform*. White paper d'Ethereum. Recuperat de `https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf`.

[7] Calderbank M. (2007). *The RSA Cryptosystem: History, Algorithm, Primes*.

[8] cardano.org (2023). *Discover Cardano*. Recuperat de `https://cardano.org/discover-cardano/`.

[9] coinmarketcap.com (2023). *Bitcoin price*. Recuperat de `https://coinmarketcap.com/currencies/bitcoin/` .

[10] Comisión Europea. (2023). *Marco legal y regulatorio para blockchain*. Recuperat de `https://digital-strategy.ec.europa.eu/es/policies/regulatory-framework-blockchain`.

[11] Cormen, T.H. (2022). *Introduction to Algorithms (4th ed.)*. MIT Press.

[12] Dan's Tools. (2020). *MD5 Hash Generator.* Recuperat de `https://www.md5hashgenerator.com`.

[13] DI Management Services. (2022). *Test vector for SHA-1, SHA-2 i SHA-3.* Recuperat de `https://www.di-mgt.com.au/sha_testvectors.html`.

[14] Dolader C., Bel J., Muñoz J.L. (2017). *La cadena de blocs: Fundamentos, aplicaciones y relación con otras tecnologías disruptivas.* Recuperat de `https://dialnet.unirioja.es/servlet/articulo?codigo=6207510`.

[15] Doms P. (2021). *Python Implementation of SHA-256 from Scratch.* Recuperat de `https://medium.com/@domspaulo/python-implementation-of-sha-256-from-scratch-924f660c5d57`.

[16] ElGamal T. (1985). *A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*(PDF). IEEE Transactions on Information Theory.

[17] Encryption Consulting. (2022). *What is SHA? What is SHA used for?* Recuperat de `https://www.encryptionconsulting.com/education-center/what-is-sha/`.

[18] ethereum.org (2023). *What is ethereum?* Recuperat de `https://ethereum.org/en/what-is-ethereum/`.

[19] Fernández I. (2014). *Introducción a la teoría combinatoria de grupos.*

[20] filecoin.io (2014). *Filecoin: A Cryptocurrency Operated File Storage Network.* Recuperat de `https://filecoin.io/filecoin-jul-2014.pdf`.

[21] Fleck M., Kuenning G. (2000). *Hash Functions.* Recuperat de `https://www.cs.hmc.edu/~geoff/classes/hmc.cs070.200101/homework10/hashfuncs.html`.

[22] Freda A. (2022). *What is the MD5 Hashing Algorithm and how does it work?* Recuperat de `https://www.avast.com/c-md5-hashing-algorithm`.

[23] Goint M., Bertelle C., Duvallet C. (2023). *Secure Access Control to Data in Off-Chain Storage in Blockchain-Based Consent Systems.* Recuperat de `https://www.mdpi.com/2227-7390/11/7/1592`.

[24] Hai Z., Biao L., Xingchen Z., Dawei C. (2022). *Transactive energy system: Concept, configuration, and mechanism.* Recuperat de `https://www.frontiersin.org/articles/10.3389/fenrg.2022.1057106/full`.

[25] Hellman M. (2002). *An Overview of Public Key Cryptography.* IEEE Communications Magazine, pg. 42–49.

[26] IBM (2023). *¿Qué es la seguridad de blockchain?* Recuperat de https://www.ibm.com/es-es/topics/blockchain-security.

[27] iota.org (2023). *What is iota?*. Recuperat de https://www.iota.org/get-started/what-is-iota.

[28] Malviya N. (2020). *Introduction to hash functions.* Recuperat de https://resources.infosecinstitute.com/topic/introduction-to-hash-functions/.

[29] Marr B. (2023). *The 5 Biggest Problems With Blockchain Technology Everyone Must Know About.* Recuperat de https://www.forbes.com/sites/bernardmarr/2023/04/14/the-5-biggest-problems-with-blockchain-technology-everyone-must-know-about/?sh=63fc64ae55d2.

[30] McDaniel B. (2017). *An Algorithm for Error Correcting Cyclic Redundance Checks.*

[31] Myasnikov A., Shpilrain V., Ushakov A. (2007). *Group-based Cryptography.* Basel, Boston, Berlin: Birkhäuser Verlag.

[32] Nakamoto S. (2008). *Bitcoin: A Peer-to-Peer Electronic Cash System.* Recuperat de https://bitcoin.org/bitcoin.pdf.

[33] National Institute of Standards and Technology. (2015). *Secure Hash Standard (SHS).* Recuperat de https://csrc.nist.gov/publications/detail/fips/180/4/final.

[34] O'Dwyer, K. J. (2014). *Bitcoin mining and its energy footprint.* Recuperat de https://proofofexistence.com/.

[35] Online Tools. (2021). *SHA256.* Recuperat de https://emn178.github.io/online-tools/sha256.html.

[36] Perlroth, N. (2016). *Hackers Used New Weapons to Disrupt Major Websites Across U.S.* Recuperat de https://www.nytimes.com/2016/10/22/business/internet-problems-attack.html.

[37] Proof of Existence. (2017). *Proof of Existence.* Recuperat de https://proofofexistence.com.

[38] Rabin M. (1979). *Digitalized Signatures and Public-Key Functions as Intractable as Factorization* MIT Laboratory for Computer Science.

[39] Ralph C.M. (1982). *Secrecy, authentication, and public key systems* UMI Research Press.

[40] ripple.com (2023). *Settlement in seconds, not days*. Recuperat de https://ripple.com/solutions/cross-border-payments/.

[41] Rodney G., Downey D.R., Hirschfeldt S.L., Reed S. (1999). *Reverse Mathematics of the Nielsen-Schreier Theorem*. Recuperat de http://www.math.uchicago.edu/~drh/Papers/Papers/ns.pdf.

[42] Rodriguez A. (2019). *¿Cuál es el problema de la escalabilidad de blockchain?*. Recuperat de https://adr-rod87.medium.com/qu\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{e\global\mathchardef\accent@spacefactor\spacefactor}\let\begingroup\let\typeout\protect\begingroup\def\MessageBreak{Ω(Font)}\let\protect\immediate\write\m@ne{LaTeXFontInfo:\def{}oninputline2075.}\endgroup\endgroup\relax\let\ignorespaces\relax\accent19e\egroup\spacefactor\accent@spacefactor-pasa-con-la-escalabilidad-de-blockchain-47b7a5a91013.

[43] Segura J. (2018). *¿Qué es Prueba de trabajo / Proof of Work (PoW)?* Recuperat de https://academy.bit2me.com/que-es-proof-of-work-pow/.

[44] solana.com (2023). *Blockchain and Solana*. Recuperat de https://solana.com/es/learn/blockchain-basics.

[45] Soto J. (2015). *El qubit y el algoritmo de Shor*. Recuperat de http://pimedios.jesussoto.es/2015/04/23/el-qubit-y-el-algoritmo-de-shor/.

[46] tezos.com (2023). *A blockchain to build, play and collect on*. Recuperat de https://tezos.com/build-play-collect/.

[47] Tomescu A. (2020). *What is a Merkle Tree?*. Recuperat de https://decentralizedthoughts.github.io/2020-12-22-what-is-a-merkle-tree/.

[48] uco.es (2019). *Introducción a la teoria de grupos*. Recuperat de http://www.uco.es/~ma1rurur/grupos%20I.PDF.

[49] unit-conversion.info (2016). *RIPEMD create hash online*. Recuperat de http://www.unit-conversion.info/texttools/ripemd/.

[50] Whittaker M. (2023). *Bitcoin Halving: How It Works and Why It Matters*. Recuperat de https://www.forbes.com/advisor/au/investing/cryptocurrency/bitcoin-halving/.

[51] Wikipedia. (2022). *Peer-to-peer*. Recuperat de https://es.wikipedia.org/wiki/Peer-to-peer.