5. Funcions hash

Les funcions hash són una primitiva criptogràfica cada vegada més important en diferents protocols i aplicacions criptogràfiques. Com veurem, una funció hash és una funció que permet obtenir un valor fixat de mida reduïda a partir d'una entrada arbitràriament gran. Gràcies a les propietats que ofereixen a aquest valor de sortida, els usos de les funcions hash són múltiples, des de la seva utilització per a l'autentificació d'informació sense l'ús de signatures digitals (fent servir criptografia simètrica) fins a la verificació de proves de treball en criptomonedes, passant per la generació de contrasenyes o la reducció de la complexitat de càlcul en un procés de signatura digital.

L'ús de les funcions hash cada vegada en més contextos implica que la seva importància també hagi anat augmentant. Com és sabut, la seguretat que ofereix un sistema criptogràfic és equivalent a la seguretat que ofereix el seu component més feble o insegur. Per tant, a mida que les funcions hash han anat incloent-se en nous sistemes, la robustesa de les funcions hash afecta de ple en la seguretat d'aquests sistemes. Aquest punt és molt rellevant perquè una vulnerabilitat en una funció hash implicaria una vulnerabilitat en tots els sistemes criptogràfics que l'utilitzen. Per exemple, si un atacant pogués predir la sortida d'una funció hash donada una entrada fixada, podria arribar a trencar la seguretat d'algunes criptomonedes.

En aquest capítol definirem què són les funcions hash i quines propietats presenten. Posteriorment, veurem com es poden construir utilitzant com a base un criptosistema de bloc. Repassarem també quines són les funcions hash més utilitzades, funcions hash construïdes específicament per a aquest propòsit i que no es basen en cap criptosistema de bloc. En concret, veurem en detall el funcionament de la funció hash SHA256. Finalment, enumerarem algunes de les múltiples aplicacions que tenen les funcions hash i també algunes propietats addicionals que es poden demanar a les funcions hash que són útils en algunes de les aplicacions esmentades.

5.1 Les funcions hash

Com ja hem avançat, les funcions hash s'utilitzen en múltiples aplicacions i la raó d'aquest fet recau en les seves propietats. En aquest apartat definirem acuradament què són les funcions hash i quina diferència hi ha entre una funció hash i una funció hash criptogràfica.

També descriurem el concepte d'atac a una funció hash i explicarem com n'indiquem el nivell de seguretat.

5.1.1 Definicions

Una **funció hash** de mida n és una funció que pren com a entrada un missatge (o cadena) d'una mida arbitràriament gran i en retorna una cadena de mida fixa n. A més, una funció hash és eficientment calculable i determinista, és a dir, donades dues entrades iguals sempre ens proporcionarà la mateixa sortida.

Mida d'una funció hash	La mida de les funcions hash es determina en bits.

L'eficiència de les funcions hash és un element molt important ja que el seu ús està especialment indicat per a reduir missatges de mida molt gran. Per aquest motiu, la facilitat per tractar aquest tipus de missatges tan grans ha d'estar garantida per tal que la seva utilització no faci augmentar la complexitat del sistema que les utilitza. D'altra banda, malgrat sembli innecessari indicar el caràcter determinista de les funcions hash, és important ressaltar-lo perquè, com veurem més endavant, les funcions hash s'utilitzen de forma similar a un oracle aleatori i això pot induir a pensar que el seu funcionament no és determinista.

Exemple 5.1 Exemple de funció hash

Un exemple de funció hash de mida 3 dígits decimals seria la següent: $h(x) = x \pmod{1000}$

Aquesta funció hash retorna sempre, per a qualsevol mida de l'entrada, un valor fixat de 3 dígits, considerant que representem el nombre amb tres dígits incloent els zeros que calgui davant. Per exemple, $h(8472937003) = 8472937003 \pmod{1000} = 003$.

De la mateixa manera, la funció $h(x) = x \pmod{2^{256}}$ també seria una funció hash, en aquest cas de mida 256 bits.

Si bé les funcions hash tal com les acabem de definir tenen algunes aplicacions, la seva potència s'incrementa quan se li afegeixen un seguit de propietats que conformen el que es coneix com a funció hash criptogràfica.

Una **funció hash criptogràfica** és una funció hash, $h(x)$, amb les següents propietats:

1. Resistent a preimatge (o unidireccional): donat un valor y no és possible calcular una x tal que $h(x) = y$.
2. Resistent a segones preimatges (o resistent a col·lisions febles): donat un valor x tal que $y = h(x)$, no és possible trobar un valor x' tal que $x' \neq x$ i que a més $y = h(x')$.
3. Resistent a col·lisions (o resistent a col·lisions fortes): no és possible trobar dos valors x_1 i x_2 diferents ($x_1 \neq x_2$) tals que $h(x_1) = h(x_2)$.

Un punt important a destacar sobre les funcions hash criptogràfiques és que, tal i com hem vist en la seva definició, no incorporen cap tipus de clau ni d'informació secreta. Donada una entrada, si coneixem de quina funció hash es tracta, en podrem calcular la sortida sense cap problema. És important destacar aquest fet perquè es pot pensar que, tractant-se d'una funció criptogràfica, cal que involucri una clau i en el cas de les funcions hash no és així.

Funcions hash i claus	Tot i que les funcions hash, per definició, no utilitzen cap clau, es poden utilitzar en esquemes en les que se'ls associï una clau, tal i com veurem en l'apartat d'aplicacions d'aquest mateix capítol.

Exemple 5.2 Contraexemple de funció hash criptogràfica

Si ens fixem en les funcions hash que hem definit en l'exemple anterior, veurem que tot i ser funcions hash, no són funcions hash criptogràfiques.

Si analitzem la funció $h(x) = x \pmod{1000}$ veiem que no compleix cap de les tres propietats que hem enumerat. Per exemple, si prenem $y = 345$, és molt simple trobar una imatge x que retorni aquest valor hash, en concret, qualsevol cadena que acabi en 345, com per exemple $x = 642345$. Per tant, la primera propietat ja no es compleix. De fet, és trivial observar que la segona i la tercera tampoc es compleixen, simplement per la simplicitat amb la que s'ha definit la funció. Per exemple, donat $x = 3456$ sabem que $y = h(3456) = 456$ i és trivial trobar un valor $x' \neq x$ tal que $h(x') = h(x)$, per exemple, $x' = 958456$ (o qualsevol cadena acabada en aquests tres nombres).

De fet, malgrat que definir les propietats d'una funció hash criptogràfica és força simple, no és gens fàcil construir una funció que les complexi, com veurem més endavant quan analitzem com es construeixen les funcions hash que s'utilitzen en l'actualitat.

De cara a simplificar tant la redacció com la lectura de la resta del capítol, abusarem del llenguatge i assumirem que totes les funcions hash a les que fem referència a partir d'aquest punt són funcions hash criptogràfiques, excepte quan diguem explícitament el contrari.

5.1.2 Propietats

És important aturar-se a mirar amb deteniment les tres propietats de les funcions hash criptogràfiques ja que l'anàlisi del seu detall permet veure que són més diferents del que aparenten.

En primer lloc, és important remarcar que una funció hash no pot ser una funció bijectiva sinó que únicament és una funció exhaustiva. És a dir, tot element té una imatge però no és cert que donada una imatge només hi hagi una sola antiimatge. Aquest fet és obvi si pensem que el conjunt de sortida pot ser de mida arbitrària (és a dir, tan gran com es vulgui) i el d'arribada té mida fixada n, més petita que la del conjunt de sortida. Per tant, si hem de poder calcular el hash de qualsevol dels elements de sortida, donat que hi ha menys elements al conjunt d'arribada, forçosament se'n repetiran, tal i com es mostra en la Figura 5.1:

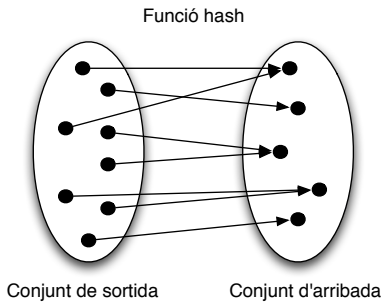


Figura 5.1: L'exhaustivitat de les funcions hash

Un altre punt a analitzar és la diferència entre la segona i la tercera propietat de les funcions hash criptogràfiques, és a dir, la diferència entre col·lisions febles i fortes. Aparentment, les dues propietats poden semblar la mateixa però una anàlisi més acurada ens mostra que ni de bon tros són iguals. La diferència entre aquestes dues propietats s'explica amb el que es coneix com la paradoxa de l'aniversari.

La paradoxa de l'aniversari ens diu que si volem que, amb probabilitat del 50%, almenys dues persones d'un grup tinguin l'aniversari el mateix dia, només cal que el grup tingui 23 persones (suposant uniforme la

distribució dels naixements al llarg dels dies de l'any). Aquests valors contradiuen la nostra intuïció que semblaria que el nombre de persones hagués de ser molt més gran, per exemple, proper o més gran a 183 que és la meitat de dies que té l'any. De fet, la contradicció ve de pensar que aquest problema pot ser equivalent a trobar dues persones que tinguin l'aniversari en un dia concret de l'any. Si fem l'analogia amb les propietats de les funcions hash criptogràfiques, el primer cas correspondria a la tercera propietat (colisions fortes) i el segon cas a la segona (col·lisions febles). Ara bé, si calculem detingudament les dues probabilitats veurem que no s'assemblen gens.

Exemple 5.3 Càlcul de les probabilitats en la paradoxa de l'aniversari.

Donat un grup de $n = 23$ persones, si triem una d'elles a l'atzar, quina és la probabilitat que una de les altres persones del grup tingui l'aniversari el mateix dia (fixeu-vos que això és el cas de les col·lisions febles).

La probabilitat que una persona tingui l'aniversari aquell dia fixat és fàcil de calcular, ja que és $\frac{1}{365}$ si suposem naixements uniformes i anys de no traspàs. Per tant, la probabilitat que l'aniversari d'aquesta persona no sigui el dia triat serà el complementari, és a dir, $1-\frac{1}{365}$. Si ara mirem per a una altra persona del grup, com que el naixement de les dues és independent, veiem que les probabilitats valen el mateix i, per tant, la probabilitat que l'aniversari de dues persones sigui diferent del dia fixat serà $(1-\frac{1}{365})^2$. Si repetim l'argument, les 22 persones restants tindran l'aniversari en un dia diferent al fixat amb probabilitat $(1-\frac{1}{365})^{22} = 0,94$. Així doncs, alguna persona tindrà l'aniversari al dia fixat amb probabilitat $(1-0,94) = 0,06$, és a dir, hi ha un 6% de probabilitat que un d'ells tingui l'aniversari en el mateix dia d'un dels altres membres del grup, un cop el membre ja s'ha fixat prèviament.

Ara bé, quina és la probabilitat que donat un grup de $n = 23$ persones, com a mínim dues d'elles tinguin l'aniversari el mateix dia. Aquest seria el cas de les col·lisions fortes.

Si prenem dues persones, la probabilitat que tinguin l'aniversari en el mateix dia és $\frac{1}{365}$ i per tant, la probabilitat que el tinguin en un dia diferent és $1-\frac{1}{365}$. Ara bé, si afegim una tercera persona, la probabilitat que aquesta nova tingui l'aniversari en un dia diferent de les dues serà de $\frac{2}{365}$, però com que les dues primeres també han de tenir l'aniversari en un dia diferent, ens queda que per a que les tres persones tinguin l'aniversari en un dia diferent la probabilitat és $\left(1-\frac{1}{365}\right)\cdot\left(1-\frac{2}{365}\right)$. Si ho generalitzem a les 23 persones, ens queda que la probabilitat que totes tinguin l'aniversari en un dia diferent és de $\left(1-\frac{1}{365}\right)\cdot\ldots\cdot\left(1-\frac{23-1}{365}\right) = 0,493$. Per tant, la probabilitat que almenys dues tinguin l'aniversari en el mateix dia és de $(1-0,493) = 0,507$.

Així, en aquest cas, amb 23 persones hi ha un 50% de probabilitat que dos d'elles tinguin l'aniversari el mateix dia. Fixeu-vos que això és molt més del que teníem en el primer cas.

Exercici 5.1 Calculeu la probabilitat que en un grup de 50 persones triades a l'atzar, dues d'elles tinguin l'aniversari el mateix dia. Quina és la probabilitat que almenys una d'elles hagi nascut el dia 1 de gener?

5.1.3 Seguretat de les funcions hash

Per parlar de seguretat d'una funció hash ens cal primer definir què s'entén per atac a una funció hash.

Un **atac a una funció hash criptogràfica** és aquell que intenta trencar alguna de les seves propietats: unidireccionalitat o no-existència de colisions (febles o fortes).

Com ja hem comentat en l'apartat anterior, és molt més probable trobar dos elements diferents que proporcionin la mateixa imatge que no pas fixar-ne un i trobar un altre element que retorni la mateixa imatge que

l'element fixat. Per tant, la manera més fàcil d'atacar un funció hash (des del punt de vista probabilístic) és per mitjà de la cerca de col·lisions fortes. Per tant, com a conseqüència de la paradoxa de l'aniversari, podem obtenir la següent fórmula:

$$t \approx 2^{\frac{n+1}{2}} \sqrt{\ln\left(\frac{1}{1-\lambda}\right)}$$

que ens proporciona el nombre de missatges t als quals hem de calcular-los el hash per trobar una col·lisió amb una probabilitat λ, on n és la mida en bits de la funció hash.

En la Taula 5.1 veiem les dades per a diferents mides de funció hash.

Taula 5.1: Nombre de missatges per aconseguir col·lisions.

	Mida de la funció hash (n)				
λ	128 bits	160 bit	256 bits	384 bits	512 bits
0,5	2^{65}	2^{81}	2^{129}	2^{193}	2^{257}
0,9	2^{67}	2^{82}	2^{130}	2^{194}	2^{258}

Així, per exemple, si tenim una funció hash de mida 160 bits, ens caldrà calcular 2^{81} missatges per trobar una col·lisió amb una probabilitat de 0,5. Però només 2^{82} perquè la probabilitat de trobar-la sigui de 0,9. Per tant, com a conclusió, veiem que per a que una funció hash tingui un nivell de seguretat d'x bits necessitarem que la seva mida sigui, com a mínim, de $2x$.

Finalment, és important indicar que malgrat que trobar una única col·lisió en una funció hash és un fet que en posa en entredit la seva seguretat, al tractar-se d'un fet probabilístic, cal analitzar com s'ha trobat la col·lisió ja que una funció hash es considera trencada només quan es pot reduir la complexitat de l'atac a valors més petits dels que determina la Taula 5.1.

5.2 Construcció de funcions hash

Les propietats que es demanen a una funció hash criptogràfica, en particular les propietats que fan referència a col·lisions, ja donen una idea de la complexitat que poden arribar a tenir aquestes funcions. Cal recordar que una funció hash no incorpora cap clau de manera que qualsevol usuari coneix el funcionament exacte i complert de la funció (no hi ha cap paràmetre desconegut) i per tant un atacant pot estudiar la construcció i funcionament per atacar-la. És per aquest motiu, que la complexitat en la definició d'aquest tipus de funcions és molt elevada, com podrem veure al llarg d'aquest capítol.

Tot i la seva complexitat, les funcions hash tenen una estructura general estàndard que s'esquematitza en la Figura 5.2.

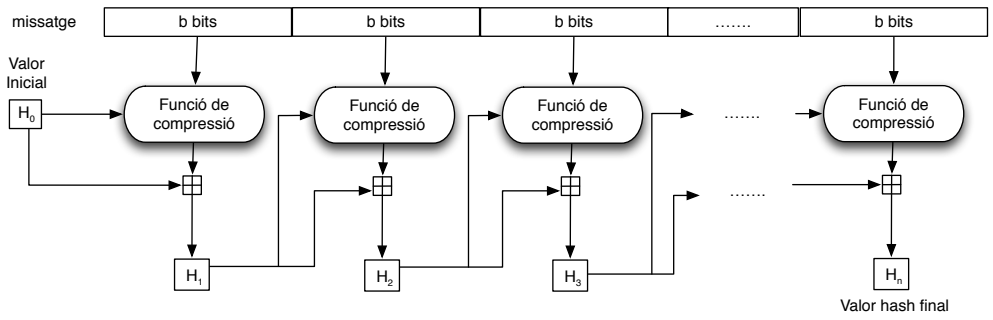


Figura 5.2: Estructura general d'una funció hash

Com es pot veure en la figura, les funcions hash processen els missatges partint-los en blocs (de forma similar a com fan els criptosistemes de bloc), tractant cada bloc de forma específica i combinant les sortides que proporciona la funció per cada bloc amb la resta de sortides dels altres blocs (també de forma semblant als modes de xifrat de bloc).

La base de les funcions hash és una funció interna que s'identifica com a funció de compressió. Aquesta funció processa cada bloc del missatge a tractar proporcionant-ne una sortida de mida igual o més petita que el propi bloc, d'aquí la seva denominació de compressió. La mida de la sortida d'aquesta funció de compressió serà la mida de la pròpia funció hash.

Depenent de com es dissenyi aquesta funció de compressió, les funcions hash es poden dividir en dos grups: funcions hash basades en criptosistemes de bloc i funcions hash de disseny específic.

5.2.1 Funcions hash basades en criptosistemes de bloc

Una manera de construir una funció hash és partint d'un criptosistema de bloc. Per a fer-ho, trobem diferents tècniques que poden combinar de diferent manera les sortides del criptosistema i el mateix bloc que s'està tractant.

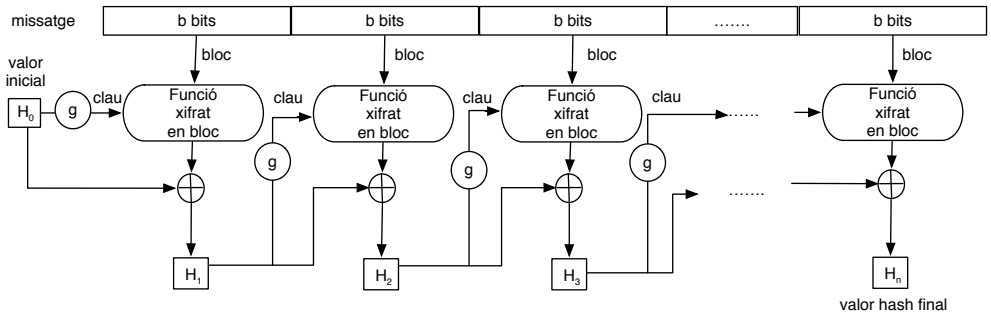


Figura 5.3: Funció hash a partir de criptosistema de bloc

En la Figura 5.3 es pot veure l'esquema d'una funció hash a partir d'un criptosistema de bloc. En la figura, el valor b indica la mida dels blocs amb el que es partirà el missatge a tractar. Per tant, com que cada bloc serà l'entrada del criptosistema de bloc, el criptosistema de bloc ha de poder treballar amb blocs de mida b. D'altra banda, el criptosistema de bloc també treballarà amb una clau. Aquesta clau té una mida l que pot coincidir, o no, amb la mida b del bloc. En el cas que les dues mides no coincideixin, com que s'utilitza la sortida d'un bloc com a clau del següent bloc ens caldrà una funció g que converteixi cadenes de b-bits a cadenes d'l-bits. En el cas que la mida del bloc sigui igual a la mida de la clau, podem prescindir de la funció g, simplement suposant que és la funció identitat. Finalment, el signe $\oplus$ del gràfic representa una operació XOR, fet que no representa un problema perquè la mida dels dos blocs que arriben a cada XOR sempre és la mateixa. Per últim, el gràfic també mostra que la mida de la funció hash és justament b, que és la mida del valor final de la sortida de l'esquema i que també coincideix amb la mida del criptosistema de bloc que estem fent servir.

Padding	En el cas que la mida del missatge no sigui múltiple del bloc, caldrà fer el *padding* del missatge per forçar-ho.

De forma més analítica, podem expressar l'esquema de la Figura 5.3 de la següent manera. Partint d'un missatge d'entrada m, el dividim en blocs de b bits obtenint $m_1, m_2, \cdots, m_n$ i per a cada bloc m_i, per a $i = 1, \cdots, n$, apliquem la següent funció definida de forma recursiva com:

$$h_i = E_{g(h_{i-1})}(m_i) \oplus h_{i-1}$$

on $E_k(\cdot)$ és la funció de xifrat en bloc amb la clau k i $h_0 = VI$, on VI és un vector inicial públicament especificat per a la funció hash en qüestió.

Exemple 5.4 Exemple de funció hash basada en un criptosistema de bloc

Definició del criptosistema de bloc:
Definim un criptosistema en bloc que treballa sobre blocs de 4 bits i denotem per $m_0m_1m_2m_3$ un bloc de text en clar. La mida de la clau d'aquest criptosistema serà de 2 bits, que denotarem per $k = k_0k_1$. La nostra funció de xifrat serà una XOR del text en clar i la clau de la següent manera:

$$c = E_k(m) = c_0c_1c_2c_3 = (m_0m_1m_2m_3) \oplus k_0k_1k_1k_0$$

D'aquesta manera, per exemple, si tenim $m = 0111$ i $k = 01$ el valor xifrat correspondrà a $c = E_k(m) = 0111 \oplus 0110 = 0001$.

Definició de la funció hash:
Definirem la nostra funció hash, $h(\cdot)$ de mida $n = 4$ bits, utilitzant el criptosistema de bloc definit anteriorment i el vector inicial $VI = 0111$. La funció $g(\cdot)$ rebrà 4 bits d'entrada i en retornarà 2 de la següent manera $g(x_0x_1x_2x_3) = (x_0 \oplus x_1)(x_2 \oplus x_3)$.

En base a aquests paràmetres, veiem com es calcularia el valor hash del missatge $m = 11001110$, és a dir $h(11001110)$.

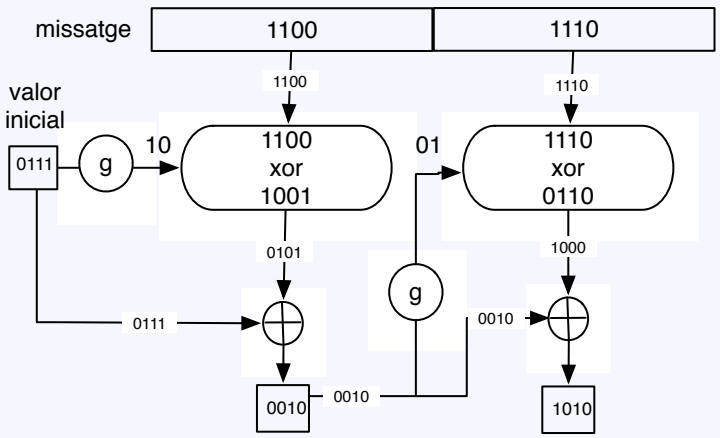
En primer lloc, partirem el missatge en blocs de 4 bits. En aquest cas tenim dos blocs $m_1 = 1100$ i $m_2 = 1110$.

Apliquem la funció de xifrat sobre m_1 amb la clau $g(VI)$. En aquest cas, $g(VI) = g(0111) = (0 \oplus 1)(1 \oplus 1) = 10$ per tant la clau que utilitzarem per al primer bloc serà $k = 10$ i el resultat del xifrat del primer bloc serà $c = E_k(m_1) = E_{10}(1100) = (1 \oplus 1)(1 \oplus 0)(0 \oplus 0)(0 \oplus 1) = 0101$. Si ara fem la XOR amb h_0 tenim $h_1 = 0101 \oplus 0111 = 0010$.

Un cop processat el primer bloc podem processar el següent utilitzant, en aquest cas, l'expressió $E_{g(h_1)}(m_2) \oplus h_1 = E_{g(0010)}(1110) \oplus 0010 = E_{01}(1110) \oplus 0010 = 1000 \oplus 0010 = 1010$.

Com que ja hem processat tots els blocs, ja hem obtingut el resultat final: $h(11001110) = 1010$.

En el gràfic de la figura següent es pot veure la versió gràfica dels càlculs:



Amb aquests tipus de construccions i utilitzant un criptosistema en bloc prou robust, podem crear funcions hash. Per exemple, utilitzant un AES amb una mida de block de 256 bits podem obtenir una funció hash amb una seguretat de 128 bits. De tota manera, a la pràctica i de forma general, s'utilitzen funcions hash de disseny específic. com les que passem a descriure en el següent apartat.

Exercici 5.2 Tenim un criptosistema de bloc que actua sobre blocs de 4 bits de longitud amb una mida de clau, k, de 3 bits, és a dir $k = k_1k_2k_3$ on $k_i \in \{0,1\}$. La funció de xifrat queda definida per la següent expressió:

$$E_k(m) = m \oplus k_{ext}$$

on k_{ext} s'obté a partir de la clau k amb la següent expressió:

$$k_{ext} = k_1k_2k_3(k_1 \oplus k_3)$$

Construïu una funció hash amb aquest criptosistema de bloc utilitzant la construcció de la Figura 5.3 prenent com a $IV = 1111$ i com a funció $g(x_1x_2x_3x_4) = x_2x_3(x_1 \oplus x_4)$.

Dibuixeu-ne l'esquema i calculeu el resultat $h(01010101)$.

5.2.2 Funcions hash de disseny específic

Més enllà de poder construir una funció hash a partir d'un criptosistema de bloc, hi ha moltes funcions hash que s'han definit específicament per a aquest propòsit. A continuació, llistem les més rellevants, donant una breu informació sobre cada una d'elles.

Les funcions MD4 i MD5 (acrònim de *Message Digest*) són funcions criptogràfiques creades per Ronald Rivest els anys 1990 i 1992, respectivament. Ambdues tenen una mida de 128 bits i processen blocs de dades de 512 bits. Les primeres vulnerabilitats de l'MD4, definida en l'RFC 1320, van ser provades ja el 1991 i en el 1995 ja es podien realitzar atacs de col·lisions en pocs segons fet que posteriorment va propiciar la retirada de la funció, explicitada en l'RFC 6150. La funció MD5, definida en l'RFC 1321, va ser desenvolupada per pal·liar les vulnerabilitats de l'MD4. De tota manera, en l'actualitat, l'MD5 es considera també insegura i el seu ús està totalment desaconsellat ja que és fàcil trobar-ne col·lisions i, fins hi tot, generar certificats digitals amb claus públiques diferents que tinguin el mateix valor hash MD5.

Certificats digitals.	La descripció i ús dels certificats digitals s'inclou en el capítol: "Infraestructures de clau pública".

RIPEMD, acrònim de *RACE Integrity Primitives Evaluation Message Digest*, és una família de funcions hash creades pels criptògrafs belgues Hans Dobbertin, Antoon Bosselaers i Bart Preneel l'any 1996 basades en la funció MD4, incorporant un seguit de millores en base a les anàlisis de seguretat i atacs realitzats sobre l'MD4. De les funcions de la família, la més coneguda i utilitzada és la RIPEMD-160, una funció hash de mida 160 bits, tot i que el conjunt de la família inclou funcions de mida 128, 256 i 320 bits. Totes elles processen el missatge amb blocs de 512 bits. L'ús d'aquesta funció, malgrat no conèixer-se'n cap atac, està poc estès ja que té una mida igual que altres funcions estandarditzades, com ara el SHA-1.

WHIRLPOOL és una altra funció hash criptogràfica creada pels criptògrafs Vincent Rijmen i Paulo S. L. M. Barreto l'any 2000. La mida d'aquesta funció és de 512 bits, la mateixa mida dels blocs que processa i la seva estructura està basada en un criptosistema semblant a l'AES. Aquesta funció ha estat estandarditzada per la International Organization for Standardization (ISO) i la International Electrotechnical Commission (IEC) sota l'estàndard ISO/IEC 10118-3.

La família de funcions SHA

Els *Secure Hash Algorithms* són un conjunt de funcions hash que estan estandarditzades pel *National Institute of Standards and Technology* (NIST) dels Estats Units. Aquestes funcions s'agrupen bàsicament en tres grans grups: SHA-1, SHA-2 i SHA-3.

En el grup SHA-1 s'hi inclou una única funció hash de mida 160 bits. Aquesta funció va ser dissenyada per la *National Security Agency* (NSA) per a la seva utilització en signatures digitals amb l'estàndard DSA. Des

del 2010, però, a causa de les seves debilitats, no se'n recomana el seu ús.

El grup SHA-2, també dissenyat per la NSA, el formen essencialment dues funcions: la SHA-256 i la SHA-512. A partir de la definició d'aquestes dues funcions, que tenen una mida de 256 i 512 bits respectivament, l'estàndard de NIST també defineix un seguit de variants de diferents mides (SHA-224, SHA-384, SHA-512/224, SHA-512/256) que s'aconsegueixen truncant els resultats del SHA-256 o del SHA-512 a més d'utilitzar uns vectors inicials diferents.

Aquests dos primers grups, SHA-1 i SHA-2, estan detalladament descrits en l'estàndard FIPS 180-4: *Secure Hash Standard* publicat pel NIST al març del 2012.

L'últim grup de funcions hash, el SHA-3, és el més nou i és un conjunt de funcions hash definides en l'estàndard FIPS 202, publicat a l'agost del 2015. Està format per quatre funcions hash, SHA3-224, SHA3-256, SHA3-384 i SHA3-512, on la numeració indica la mida en bits de cada funció. A diferència de les funcions dels dos grups anteriors, la tria del SHA-3 es va realitzar a través d'una selecció pública i oberta en la que van participar investigadors de tot el món, de forma semblant a la que es va realitzar per a la tria de l'AES. En aquest cas, l'algorisme seleccionat va ser el KECCAK, proposat per Guido Bertoni, Joan Daemen, Michaël Peeters, i Gilles Van Assche que és el que s'ha estandarditzat sota les sigles SHA-3.

5.3 L'estàndard SHA-256

En aquest apartat estudiarem en detall una de les funcions hash més utilitzades en l'actualitat: el SHA-256. Veurem quines són les seves característiques i descriurem amb detall tot el seu funcionament.

Com ja hem comentat, el SHA-256 és una de les funcions hash definides en l'estàndard FIPS-180-4 publicat pel NIST i que va ser desenvolupat al 2001 per la NSA. Com a característiques generals, el SHA-256 és una funció hash de mida 256 bits que processa els missatges d'entrada en blocs de 512 bits. Pot processar missatges de fins a 2^{64} bits i utilitza un sistema de càlcul iteratiu amb un total de 64 iteracions.

L'estructura del SHA-256 segueix l'esquema mostrat en la Figura 5.2 amb una funció de compressió que s'executa sobre cada bloc del missatge d'entrada, el resultat de la qual es combina amb el resultat de la mateixa funció del bloc anterior.

Prèviament al processat de cada un dels blocs del missatge, el SHA-256 processa el missatge a tractar per tal d'assegurar-se que la mida del missatge coincideix amb un nombre enter de blocs. Aquest procés és coneix com a *padding* i es descriu en el següent apartat.

5.3.1 Padding del missatge

Quan s'utilitzen funcions que processen els missatges en blocs, pot succeir que la mida del missatge a tractar no sigui un múltiple de la mida del bloc, és a dir, que quan dividim el missatge en blocs ens quedi un últim bloc més petit que la mida del bloc amb el que treballa la funció. En aquests casos el que es fa és un procés de farcit (en anglès *padding*). L'estàndard SHA-256 defineix, de la següent manera, com s'ha de realitzar aquest procés.

Padding	Tingueu en compte que tot i que el padding és una tècnica molt utilitzada per a funcions que tracten els missatges en blocs, la manera com es fa aquest padding pot diferir en cada cas. Per exemple, el padding que utilitza el SHA-256 és diferent del que utilitza el SHA-512 i també diferent del que utilitza el criptosistema de bloc AES.

Suposem un missatge M de mida l bits, on $l \neq 0 \mod 512$. En aquest cas procedirem a:

1. afegir el bit 1 al final del missatge,
2. seguit per k bits a zero, on $k = 448 - (l+1) \mod 512$, prenent la solució més petita i no negativa,
3. afegir un bloc de 64 bits que sigui igual al nombre l expressat en binari.

Exemple 5.5 Exemple de càlcul de padding en el SHA-256

Suposem qeu volem processar el missatge `abc` amb la funció hash SHA-256.

Prenem el missatge `abc`, que expressat en codi ASCII de 8 bits és:

$$\underbrace{01100001}_{\texttt{a}}\ \underbrace{01100010}_{\texttt{b}}\ \underbrace{01100011}_{\texttt{c}}$$

Per tant, tenim que la mida del missatge l val $l = 3 \cdot 8 = 24$ i com que no és 512 ens cal fer padding.

En primer lloc afegim el bit 1:

$$\underbrace{01100001}_{\texttt{a}}\ \underbrace{01100010}_{\texttt{b}}\ \underbrace{01100011}_{\texttt{c}}\ 1$$

Ara, calculem el valor k com $k = 448 - (24 + 1) \bmod 512 = 423$. És a dir, caldrà afegir 423 zeros al missatge:

$$\underbrace{01100001}_{\texttt{a}}\ \underbrace{01100010}_{\texttt{b}}\ \underbrace{01100011}_{\texttt{c}}\ 1\ \underbrace{00\cdots0}_{\text{423 zeros}}$$

i finalment, caldrà afegir els 64 bits que falten fins a completar els 512 que necessitem. Aquests 64 bits seran el valor de l en binari, és a dir $l = 24_{(10} = 00\cdots011000_{(2}$, de manera que el missatge final amb el padding serà:

$$\underbrace{01100001}_{\texttt{a}}\ \underbrace{01100010}_{\texttt{b}}\ \underbrace{01100011}_{\texttt{c}}\ 1\ \underbrace{00\cdots0}_{\text{423 zeros}}\ \underbrace{00\cdots011000}_{\text{64 bits}}$$

Mida màxima dels missatges

Fixeu-vos que aquest mecanisme de *padding* implica que la mida màxima dels missatges que pot tractar el SHA-256 és de 2^{64} bits, ja que és el màxim valor que es pot representar en la última part de la cadena de *padding*.

Exercici 5.3 Calculeu la cadena de bits que processarà la funció SHA256 una vegada s'ha realitzat el padding al missatge d'entrada $m = \texttt{SALA}$, on els caràcters s'han codificat en ASCII amb 8 bits.

5.3.2 Funció de compressió del SHA-256

Tal com ja hem comentat, el SHA-256 treballa amb blocs de 512 bits els quals processa a través de la funció de compressió. En aquesta funció de compressió podem identificar tres fases:

1. Expansió del bloc (*block schedule*).
2. Inicialització del buffer.
3. Procés de compressió.

En l'expansió del bloc és processen els 512 bits del bloc per tal d'obtenir-ne una cadena molt més llarga de 2048 bits (64 paraules de 32 bits). En la inicialització del buffer es carreguen en memòria els valors d'inicialització de la funció recurrent, valors definits en l'estàndard. Posteriorment, s'aplica el procés de compressió a la cadena de 2048 bits.

Prèviament a detallar cada un d'aquests passos, definirem algunes funcions internes que s'utilitzen en cada un dels processos que acabem d'enumerar.

Definició de les funcions internes del SHA-256

En aquest apartat definirem un seguit de funcions que s'utilitzaran tant en la part d'expansió del bloc com en el procés de compressió del SHA-256.

La funció $ROTR^n(x)$ efectua una rotació circular a la dreta d'n bits.
La funció $SHR^n(x)$ és un operador lògic de desplaçament a la dreta: $SHR^n(x) = x \ggg n$, és a dir, mou n bits a la dreta omplint els nous bits amb zeros.

Evidentment, ambdues funcions treballen a nivell de bit.

Exemple 5.6 Exemple de càlcul de la funció $ROTR^n(x)$

Sigui $x = abcdefgh$ una cadena de 8 bits i $n = 3$, aleshores
$ROTR^3(abcdefgh) = fghabcde$

Exemple de càlcul de la funció $SHR^n(x)$

Sigui $x = abcdefgh$ una cadena de 8 bits i $n = 3$, aleshores
$SHR^3(abcdefgh) = 000abcde$

Expansió del bloc

Per a processar un bloc de 512 bits, en primer lloc, la funció SHA-256 l'expandeix a un total de 2048 bits. Per fer-ho, parteix el bloc de 512 bits en 16 paraules de 32 bits. Sigui M el bloc de dades de 512 bits, el podem expressar com $M = M_0||M_1||\cdots||M_{15}$ on cada M_i té una mida de 32 bits. A partir d'aquests blocs, generarem 64 blocs de 32 bits, denotats per $W_0, W_1, \cdots, W_{63}$ que formaran el total de 2048 bits que necessitarem. Per fer-ho utilitzarem la següent expressió:

$$W_t = \begin{cases} M_t & 0 \leq t \leq 15 \\ \sigma_1(W_{t-2}) \boxplus W_{t-7} \boxplus \sigma_0(W_{t-15}) \boxplus W_{t-16} & 16 \leq t \leq 63 \end{cases}$$

on les funcions σ_0 i σ_1 estan definides de la següent manera:

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

i l'operació $\boxplus$ és una suma mòdul 2^{32}.

Exemple 5.7 Exemple d'expansió d'un bloc

Suposem que volem fer l'expansió del bloc que hem obtingut en l'exemple del padding del missatge `abc` expressant en codi ASCII de 8 bits. Hem vist que el bloc de 512 bits, expressat en hexadecimal amb paraules de 32 bits és:

$M = M_0||M_1||\cdots||M_{15} =$ 0x61626380 0x00000000 0x00000000 0x00000000

0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000

0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000018

Els primers 16 valors $W_0, \cdots, W_{15}$ de la cadena expandida seran aquests mateixos valors del bloc, és a dir:

$W = W_0||W_1||\cdots||W_{15} =$ 0x61626380 0x00000000 0x00000000 0x00000000

0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000

0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000018

Passem ara a calcular la següent paraula de 32 bit, W_{16}.

$$
\begin{aligned}
W_{16} &= \sigma_1(W_{14}) \boxplus W_9 \boxplus \sigma_0(W_1) \boxplus W_0 = \\
&= \sigma_1(\text{0x00000000}) \boxplus \text{0x00000000} \boxplus \sigma_0(\text{0x00000000}) \boxplus \text{0x61626380} = \\
&= (ROTR^{17}(\text{0x00000000}) \oplus ROTR^{19}(\text{0x00000000}) \oplus SHR^{10}(\text{0x00000000})) \boxplus \\
&\quad \boxplus \text{0x00000000} \boxplus \\
&\quad \boxplus (ROTR^{7}(\text{0x00000000}) \oplus ROTR^{18}(\text{0x00000000}) \oplus SHR^{3}(\text{0x00000000})) \boxplus \\
&\quad \boxplus \text{0x61626380} = \\
&= \text{0x00000000} \boxplus \text{0x00000000} \boxplus \text{0x00000000} \boxplus \text{0x61626380} = \\
&= \text{0x61626380}
\end{aligned}
$$

Per tant, $W_{16} =$ 0x61626380.

De la mateixa manera podem calcular el següent element W_{17}:

$$
\begin{aligned}
W_{17} &= \sigma_1(W_{15}) \boxplus W_{10} \boxplus \sigma_0(W_2) \boxplus W_1 = \\
&= \sigma_1(\text{0x00000018}) \boxplus \text{0x00000000} \boxplus \sigma_0(\text{0x00000000}) \boxplus \text{0x00000000} = \\
&= (ROTR^{17}(\text{0x00000018}) \oplus ROTR^{19}(\text{0x00000018}) \oplus SHR^{10}(\text{0x00000018})) \boxplus \\
&\quad \boxplus \text{0x00000000} \boxplus \\
&\quad \boxplus (ROTR^{7}(\text{0x00000000}) \oplus ROTR^{18}(\text{0x00000000}) \oplus SHR^{3}(\text{0x00000000})) \boxplus \\
&\quad \boxplus \text{0x00000000} = \\
&= \text{0x000}f\text{0000} \boxplus \text{0x00000000} \boxplus \text{0x00000000} \boxplus \text{0x00000000} = \\
&= \text{0x000}f\text{0000}
\end{aligned}
$$

Així, $W_{17} =$ 0x000f0000.

De la mateixa manera es calculen la resta de paraules fins a completar els 2048 bits.

Inicialització del buffer

Tal com veurem en el següent apartat, el procés de compressió és un procés recursiu. Per aquest motiu, ens caldrà definir uns valors als quals s'inicialitzaran les variables de la funció hash. Aquests valors, que es detallen a continuació, estan definits en el propi estàndard:

$H_0^{(0)} =$ `0x6a09e667`
$H_1^{(0)} =$ `0xbb67ae85`
$H_2^{(0)} =$ `0x3c6ef372`
$H_3^{(0)} =$ `0xa54ff53a`
$H_4^{(0)} =$ `0x510e527f`
$H_5^{(0)} =$ `0x9b05688c`

$H_6^{(0)} = \texttt{0x1f83d9ab}$
$H_7^{(0)} = \texttt{0x5be0cd19}$

A més d'aquests valors d'inicialització, l'estàndard també defineix 64 constants que s'utilitzen en cada una de les iteracions de la funció de compressió. Aquests constants són les següents:

```
K = [ 0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1,
      0x923f82a4, 0xab1c5ed5, 0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
      0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174, 0xe49b69c1, 0xefbe4786,
      0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
      0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147,
      0x06ca6351, 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 ]
```

Funció de compressió

La funció de compressió és l'encarregada de prendre la cadena estesa de 64 paraules de 32 bits (és a dir, 2048 bits) i reduir-la a una cadena de 256 bits, que és justament la mida de la funció hash. Aquesta funció de compressió és un procés iteratiu en el qual s'executen 64 rondes. En la Figura 5.4 es pot veure el diagrama de la funció de compressió del SHA-256 aplicada a un únic bloc.

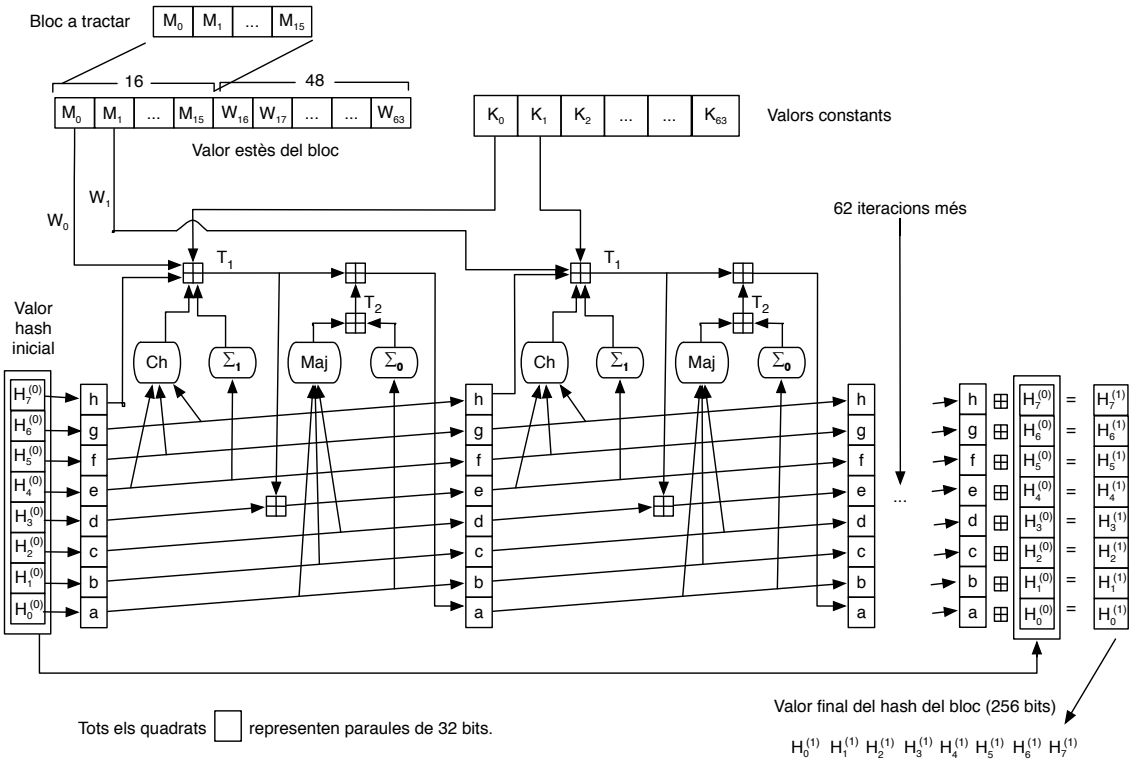


Figura 5.4: Esquema de compressió d'un bloc de la funció SHA-256

En l'esquema de la Figura 5.4 es pot veure com en cada una de les 64 iteracions es fa servir tant una de les paraules de 32 bits, W_i, com una de les constants k_i del vector K, també de 32 bits, definides en l'estàndard.

També veiem que els valors del bloc a comprimir es combinen amb els valors inicials del hash, definits també en l'estàndard com $H^{(0)}$. El gràfic també mostra com aquestes combinacions estan formades per quatre funcions, Ch, Maj, $\sum_0$ i $\sum_1$, que es descriuen a continuació:

- $\text{Ch}(e,f,g) = (e \wedge f) \oplus (\neg e \wedge g)$
- $\text{Maj}(a,b,c) = (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$
- $\sum_0(a) = ROTR^2(a) \oplus ROTR^{13}(a) \oplus ROTR^{22}(a)$
- $\sum_1(e) = ROTR^6(e) \oplus ROTR^{11}(e) \oplus ROTR^{25}(e)$

Noteu que les funcions $\text{Ch}(e,f,g)$ i $\text{Maj}(a,b,c)$ malgrat la complicació de la seva formulació tenen una interpretació força simple. La funció $\text{Ch}(e,f,g)$ és una funció de tria (Ch- *choose*). Si el bit del valor e és un 1, la sortida de la funció és el bit del valor f i si el bit del valor e és un 0, la sortida és el bit del valor g. La funció $\text{Maj}(a,b,c)$ és una funció de majoria. El bit de sortida de la funció és el bit que, en cada posició, apareix més vegades quan comparem les tres cadenes a,b i c.

Recordeu

Una XOR es pot pensar com una suma mòdul 2 (component a compoment) $000101 \oplus 000111 = 000010$ i un AND com un producte mòdul 2 (component a compoment) $000101 \wedge 000111 = 000101$. Recordeu també l'operant de negació $\neg 0100 = 1011$. A més, en la nostra notació, l'operació $\boxplus$ és una suma mòdul 2^{32}.

La funció Ch actua sobre tres paraules de 32 bits amb operacions lògiques bàsiques.

Exemple 5.8 Exemple de càlcul de la funció Ch.

Càlcul de la funció $\text{Ch}(e,f,g)$ per als valors:
$e = 0\text{x}510e527f, f = 0\text{x}9b05688c, g = 0\text{x}1f83d9ab$.

e	01010001000011100101001001111111
f	10011011000001010110100010001100
$e \wedge f$	00010001000001000100000000001100

$\neg e$	10101110111100011010110110000000
g	00011111100000111101100110101011
$\neg e \wedge g$	00001110100000011000100110000000

$e \wedge f$	00010001000001000100000000001100
$\neg e \wedge g$	00001110100000011000100110000000
$(e \wedge f) \oplus (\neg e \wedge g)$	00011111100001011100100110001100

Per tant, el resultat en hexadecimal serà $\text{Ch}(e,f,g) = 0\text{x}1f85c98c$

La funció Maj també actua sobre 3 paraules de 32 bits i, al igual que la funció Ch, també opera utilitzant operacions lògiques bàsiques.

Exemple 5.9 Exemple de càlcul de la funció Maj.

Càlcul de la funció $\text{Maj}(a,b,c)$ per als valors:
$a = 0\text{x}6a09e667, b = 0\text{x}bb67ae85, c = 0\text{x}3c6ef372$.

a	01101010000010011110011001100111
b	10111011011001111010111010000101
$a \wedge b$	00101010000000011010011000000101

a	01101010000010011110011001100111
c	00111100011011101111001101110010
$a \wedge c$	00101000000010001110001001100010

b	10111011011001111010111010000101
c	00111100011011101111001101110010
$b \wedge c$	00111000011001101010001000000000

$a \wedge b$	00101010000000011010011000000101
$a \wedge c$	00101000000010001110001001100010
$b \wedge c$	00111000011001101010001000000000
$(a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$	00111010011011111110011001100111

Per tant, el resultat en hexadecimal serà $\text{Maj}(a,b,c) = 0\text{x}3a6fe667$

La funció $\sum_0$ actua únicament sobre una sola paraula de 32 bits generant tres paraules a partir de diferents rotacions dels seus bits i realitzant una XOR d'aquestes tres paraules.

Exemple 5.10 Exemple de càlcul de la funció $\sum_0$.

Càlcul de la funció $\sum_0(a)$ per al valor: $a = 0\text{x}6a09e667$

a	01101010000010011110011001100111
$ROTR^2(a)$	11011010100000100111100110011001

a	01101010000010011110011001100111
$ROTR^{13}(a)$	00110011001110110101000001001111

a	01101010000010011110011001100111
$ROTR^{22}(a)$	00100111100110011001110110101000

$ROTR^2(a)$	11011010100000100111100110011001
$ROTR^{13}(a)$	00110011001110110101000001001111
$ROTR^{22}(a)$	00100111100110011001110110101000
$ROTR^2(a) \oplus ROTR^{13}(a) \oplus ROTR^{22}(a)$	11001110001000001011010001111110

Per tant, el resultat en hexadecimal serà $\sum_0(a) = 0\text{x}ce20b47e$.

La funció $\sum_1$ és molt similar a la funció $\sum_0$ i únicament es diferencia en el número de bits que es roten per derivar les tres paraules.

Exemple 5.11 Exemple de càlcul de la funció $\sum_1$.

Càlcul de la funció $\sum_1(e)$ per al valor: $e = 0\text{x}510e527f$

e	01010001000011100101001001111111
$ROTR^6(e)$	11111101010001000011100101001001

e	01010001000011100101001001111111
$ROTR^{11}(e)$	01001111111010100010000111001010

e	01010001000011100101001001111111
$ROTR^{25}(e)$	10000111001010010011111110101000

$ROTR^{6}(e)$	11111101010001000011100101001001
$ROTR^{11}(e)$	01001111111010100010000111001010
$ROTR^{25}(e)$	10000111001010010011111110101000
$ROTR^{6}(e) \oplus ROTR^{11}(e) \oplus ROTR^{25}(e)$	00110101100001110010011100101011

Per tant, el resultat en hexadecimal serà $\sum_1(e) = 0x3587272b$

Exercici 5.4 Realitzeu els següents càlculs de les funcions internes del SHA256 tenint en compte els valors de les cadenes:
$m_1 = 00000000000000000111111111111111$
$m_2 = 11110000000000000111111111110000$
$m_3 = 11111111000000001111111100000000$

1. Calculeu el resultat de la funció $ROTR^7(m_1)$.
2. Calculeu el resultat de la funció $SHR^{10}(m_1)$.
3. Calculeu el resultat de la funció $\sigma_0(m_1)$.
4. Calculeu el resultat de la funció $\sigma_1(m_1)$.
5. Calculeu el resultat de la funció $\sum_0(m_1)$.
6. Calculeu el resultat de la funció $\sum_1(m_1)$.
7. Calculeu el resultat de la funció $\mathrm{Ch}(m_1, m_2, m_3)$.
8. Calculeu el resultat de la funció $\mathrm{Maj}(m_1, m_2, m_3)$.

5.3.3 SHA-256 sobre múltiples blocs

En els apartat anteriors hem proporcionat el detall de la funció SHA-256 quan aquesta s'aplica a un únic bloc de dades de 512 bits, que és la mida del bloc amb el que treballa la funció. Ara bé, si el missatge del qual volem calcular el hash conté més d'un bloc, aleshores cal aplicar la funció de compressió de forma recursiva sobre cada bloc, encadenant la sortida de cada bloc amb l'entrada del següent. En la Figura 5.5 es pot veure l'esquema complert per al càlcul d'un hash sobre un missatge amb tres blocs, és a dir un missatge de 1536 bits de longitud.

Com es pot apreciar en la figura, per a cada bloc es realitza l'expansió per obtenir la cadena W de 2048 bits. Les paraules de 32 bits que formen aquesta cadena són utilitzades en cada una de les 64 iteracions de la funció de compressió juntament amb els valors constants K definits en l'estàndard. Fixeu-vos que en cada bloc s'utilitzen els corresponents valors W_i obtinguts del propi bloc però en canvi, els valors K_i utilitzats en el processat de cada bloc són sempre els mateixos. Per últim, cal notar que el resultat del hash de cada bloc s'utilitza com a valor inicial per al càlcul del hash del següent bloc.

5.4 Aplicacions de les funcions hash

Un cop estudiades les característiques i propietats de les funcions hash i després de veure com es poden construir passem al darrer punt d'aquest capítol en el que veurem les múltiples aplicacions on s'utilitzen les funcions hash.

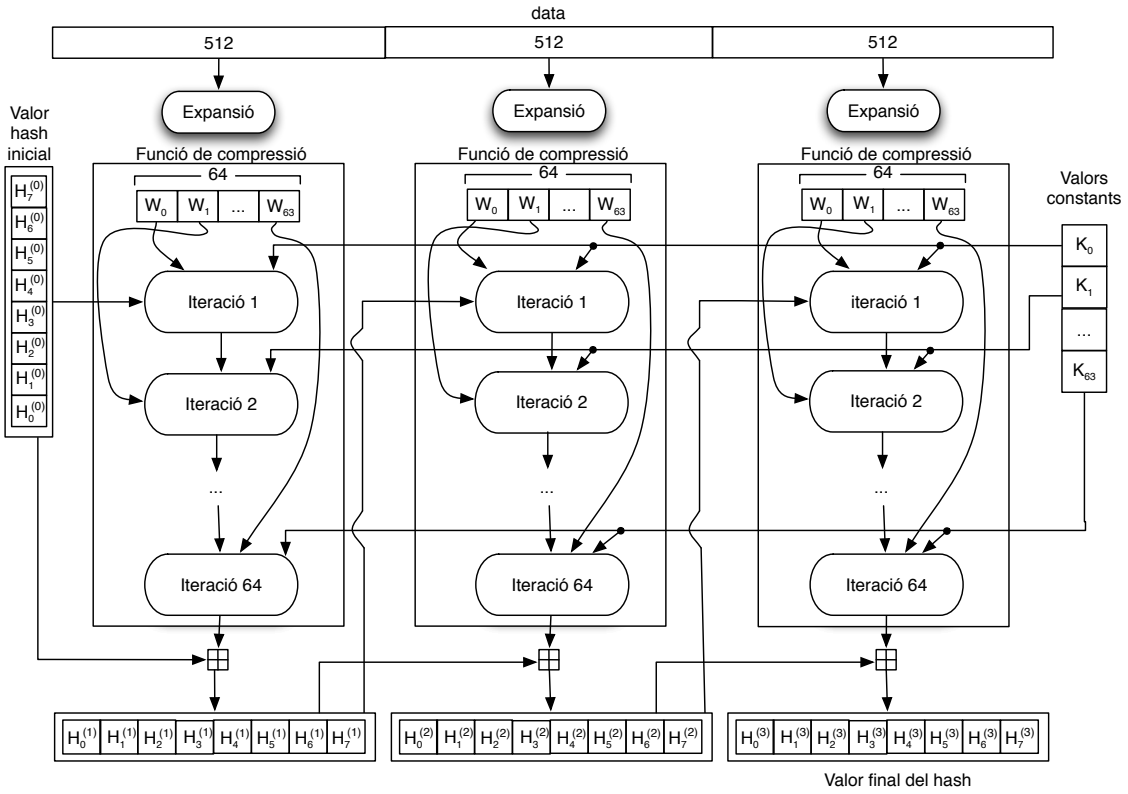


Figura 5.5: SHA-256 mostrant el processat de 3 blocs.

5.4.1 Codis d'autenticació de missatges

Un **codi d'autentificació de missatge**, en anglès *Message Authentication Code* (MAC), és una cadena curta d'informació relacionada amb el propi missatge a través d'una clau de manera que permet la seva autenticació.

Altres denominacions

Els *message authentication codes* també s'acostumen a conèixer com a *cryptographic checksums* o *keyed hash functions*.

Donat que els codis d'autentificació permeten autentificar missatges, comparteixen algunes propietats amb les signatures digitals com ara la pròpia autentificació així com la integritat del missatge. Tot i això, els codis d'autenticació no ofereixen la propietat de no repudi, propietat que sí que ofereixen les signatures digitals. Ara bé, els codis d'autenticació són molt més ràpids i eficients de calcular i és per aquest motiu que s'utilitzen en entorns on la propietat de no repudi no és essencial.

Signatures digitals

La definició i funcionament de les signatures digitals la trobareu en el capítol "Criptografia de clau pública".

Com veurem a continuació, els MAC es poden implementar de forma molt simple utilitzant conjuntament funcions hash i una clau. Aquest tipus de funcions MAC s'acostumen a denominar HMAC, justament per la utilització de la funció hash. Aquesta idea d'utilitzar una clau pot semblar contradictòria amb el que hem comentat anteriorment, indicant que les funcions hash no incorporen cap clau ni cap element secret. La manera, però, com s'utilitza la clau és simplement per variar d'alguna manera la forma del missatge que es vol autenticar. Per exemple, donada una funció hash $h(\cdot)$ podem derivar-ne dos MACs de la següent manera:

$$HMAC1_k(m) = h(k \parallel m)$$
$$HMAC2_k(m) = h(m \parallel k)$$

on el símbol $\parallel$ representa la concatenació de cadenes. La primera expressió es coneix com a *secret prefix HMAC* i la segona com a *scret suffix HMAC*.

Exemple 5.12 Exemple de càlcul d'un HMAC

Veiem un exemple de com utilitzar la funció hash definida en l'Exemple 5.4 per a calcular un *secret prefix* HMAC.

El missatge sobre el que calcularem l'HMAC serà el següent $m = 11101010$ i utilitzarem la clau $k = 1100$.

Per tant,

$$HMAC_k(m) = h(k \parallel m) = HMAC_{1100}(11101010) = h(1100 \parallel 11101010) = h(110011101010)$$

En aquest cas tenim tres blocs: $m_1 = 1100$, $m_2 = 1110$ i $m_3 = 1010$. Si ens hi fixem, els dos primers blocs són els mateixos que els de l'exemple de la funció hash, per tant tenim que $h_2 = 1010$. Per tant, el valor final de sortida de la funció dels tres blocs serà $h = h_3 = E_{g(h_2)}(m_3) \oplus h_2 = E_{11}(1010) \oplus 1010 = 0101 \oplus 1010 = 1111$

Per tant, $HMAC_k(m) = HMAC_{1100}(11101010) = 1111$.

Des del punt de vista pràctic, la manera com els MAC s'utilitzen per autentificar missatges es troba esquematitzat en la Figura 5.6.

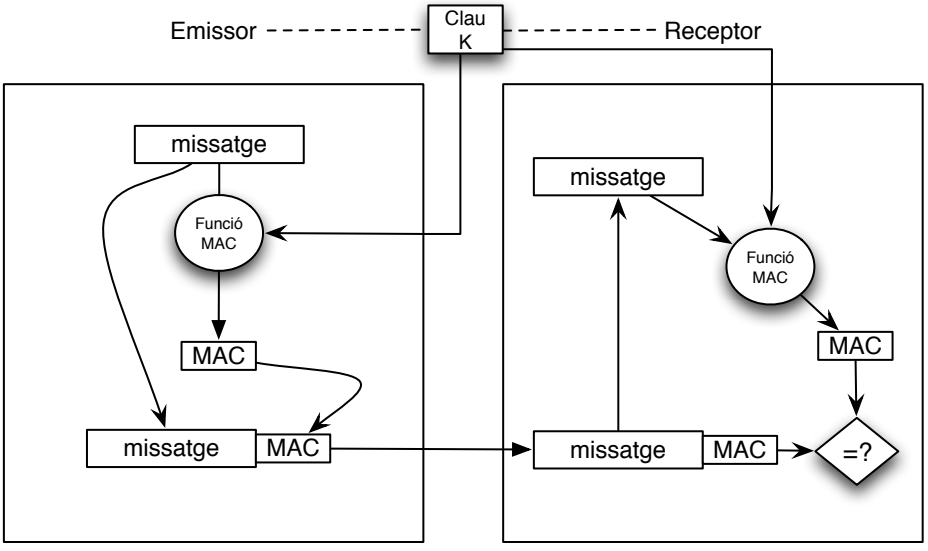


Figura 5.6: Utilització d'un HMAC per autentificar missatges.

Com es pot veure en la figura, emissor i receptor comparteixen una clau secreta. Cada vegada que l'emissor vol enviar un missatge al receptor, en calcula el seu valor HMAC utilitzant la clau secreta que comparteix

amb el receptor i annexa al missatge el valor resultant. Quan el receptor rep el missatge pot utilitzar la clau i la funció hash establerta per tornar a calcular-ne el valor HMAC i comprovar que efectivament coincideix. Fixeu-vos que un atacant que canviï el missatge que emissor i receptor s'intercanvien, ha de canviar també el valor HMAC del missatge ja que si no ho fa, la comprovació del receptor no serà correcta. Ara bé, l'atacant no coneix el valor de la clau que intercanvien i per tant no pot calcular el valor correcte de l'HMAC per al missatge modificat.

Autenticació vs confidencialitat

Fixeu-vos que els codis HMAC s'utilitzen per assegurar-se que ningú no pot canviar el missatge sense que el receptor se n'adoni. Ara bé, donat que no xifrem el missatge, la comunicació no oferirà confidencialitat i un atacant pot conèixer-ne el contingut.

Exercici 5.5 Els usuaris A i B s'intercanvien missatges. Com que A i B ja comparteixen una clau simètrica, han decidit que calcularan un HMAC dels missatges per assegurar-ne la seva integritat, és a dir, per assegurar-se que ningú que intercepti la informació pugui modificar-la. Calculant un HMAC del missatge a partir de la clau simètrica que comparteixen volen evitar que si algú modifica el missatge no pugui modificar l'HMAC de forma correcta, ja que l'atacant desconeix la clau. Fan servir una funció HMAC basada en la funció hash $h(\cdot)$ definida en l'Exemple 5.4, concretament utilitzant la clau k com un *secret prefix*, és a dir $\text{HMAC}_k(m) = h(k \,\|\, m)$. D'aquesta manera, A envia el missatge $m = 0111$ a B seguit de l'$\text{HMAC}_k = 0111$, on k és la clau simètrica que fan servir i només ells dos coneixen.

Malauradament, no saben que la tècnica del *secret prefix* no és segura i nosaltres, com a atacants, podem afegir la cadena que vulguem al missatge original i calcular l'HMAC sense conèixer k. Podríeu calcular l'HMAC corresponent al missatge $m' = 01111111$?

5.4.2 Resum de missatges

Una altra de les aplicacions en les que s'utilitzen les funcions hash és per obtenir una representació compacta d'un missatge més gran. Gràcies a que el valor hash d'un missatge pot permetre identificar-lo de forma pràcticament unívoca, aquest resum es pot utilitzar en diferents contexts. Per exemple, quan es volen emmagatzemar fitxers molt grans, sovint en format multimèdia, en una base de dades, s'acostuma a guardar-ne només el seu valor hash en la pròpia base de dades i una localització externa. D'aquesta manera, es pot referenciar el contingut i fer-ne cerques fins hi tot partint del propi contingut, també utilitzant-ne el valor hash, per tal d'obtenir-ne informació associada.

Tenir un resum d'un missatge també és molt rellevant quan les operacions que s'han de realitzar sobre el missatge són molt costoses i ens és suficient realitzar-les sobre un resum. Aquest és el cas de les signatures digitals. Tal com veurem més endavant, les signatures digitals són computacionalment poc eficients i per aquest motiu, en comptes de realitzar-les sobre el missatge sencer s'apliquen sobre un resum d'aquest. La mida reduïda i fixa que s'obté amb una funció hash permet augmentar molt l'eficiència de les signatures digitals.

5.4.3 Emmagatzematge de contrasenyes

Una altra de les aplicacions de les funcions hash és la seva utilització en l'emmagatzematge d'algunes dades sensibles, com ara les contrasenyes d'accés a un sistema informàtic. La protecció de les contrasenyes és altament necessària per assegurar que cap usuari maliciós se'n pugui apoderar i pugui accedir al sistema suplantant altres usuaris. Per aquest motiu les contrasenyes mai es guarden en clar.

L'emmagatzematge de les contrasenyes serveix per poder-les comparar amb les que els usuaris introdueixen en el procés d'autenticació. Si la contrasenya proporcionada per l'usuari coincideix amb la que el sistema emmagatzema, l'autenticació es considera vàlida. Ara bé, com ja hem dit, les contrasenyes no s'emmagatzemen en clar en el sistema sinó que s'emmagatzema la imatge de la contrasenya per una funció hash.

Contrasenyes no-xifrades

Tot i que col·loquialment sovint es parla que les contrasenyes en els sistemes es guarden xifrades, aquesta denominació no és correcta ja que una informació xifrada s'ha de poder desxifrar i la imatge d'una funció hash no permet "desxifrar-ne" el seu valor, perquè voldria dir invertir la funció hash, cosa que no és possible.

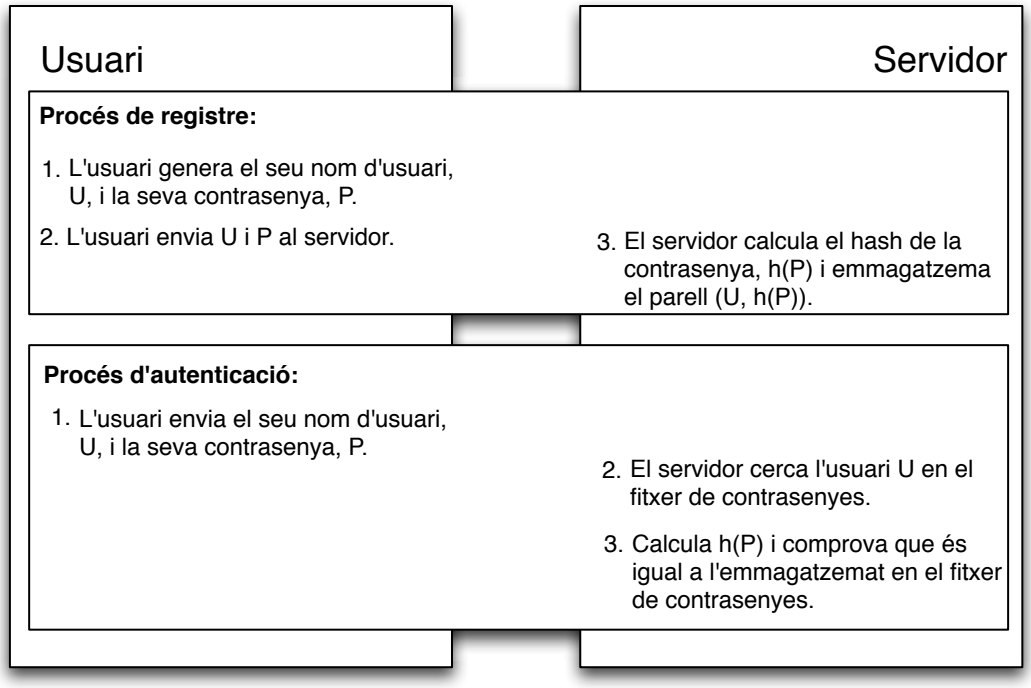


Figura 5.7: Esquema d'autenticació amb contrasenya

El procés d'autenticació amb contrasenyes guardades com a imatge d'una funció hash es mostra en la Figura 5.7. Quan un usuari vol accedir al sistema, proporciona el seu usuari i la seva contrasenya. El sistema, a partir de la contrasenya que li ha fet arribar l'usuari, en calcula el seu hash i el compara amb el valor que té emmagatzemat. En el cas que els dos valors coincideixin, l'usuari queda autenticat correctament.

Recuperació de contrasenya

En un sistema d'accés amb contrasenya ben implementat, ni tan sols l'administrador del sistema us pot dir la vostra contrasenya en cas que l'hàgiu oblidat, perquè ell no la coneix i només en té la imatge per una funció hash. Per aquest motiu, quan oblidem la contrasenya el sistema ens demana que en generem una de nova.

En realitat, el sistema descrit anteriorment és una simplificació del sistema que realment es fa servir per emmagatzemar contrasenyes, ja que les contrasenyes emmagatzemades únicament amb el seu hash permeten atacs eficients com ara el següent. Suposem un sistema que té les contrasenyes emmagatzemades utilitzant el hash SHA256 de la contrasenya. En aquest cas, el fitxer de contrasenyes tindrà un seguit de valors de 256 bits cada un d'ells vinculat a un usuari. En el cas que un atacant pogués aconseguir aquest fitxer, podria realitzar el següent atac: Prenent un diccionari de contrasenyes habituals, pot anar calculant el valor SHA256 d'aquestes contrasenyes i anar-lo comparant amb cada un dels valors del fitxer. Fixeu-vos que només que algun dels usuaris del sistema tingui una de les contrasenyes del diccionari, a l'atacant només li caldrà calcular un sol hash i comparar-lo amb cada un dels valors del fitxer fins a trobar el correcte. A més, en el cas que diferents sistemes utilitzessin la mateixa funció hash, l'atacant també podria tenir un diccionari dels hashos en comptes del diccionari de les contrasenyes.

Per evitar aquests tipus d'atacs, abans de calcular el hash de la contrasenya per a emmagatzemar-la, el que es fa és afegir a la contrasenya un valor fixat, que s'anomena *salt*, i que és diferent per cada usuari, de manera que encara que dos usuaris tinguin la mateixa contrasenya el hash que s'emmagatzemi sigui diferent. Evidentment, aquest valor també s'haurà d'afegir en el procés d'autenticació quan s'està validant la correcció de la contrasenya proporcionada per l'usuari. Fixeu-vos que un atacant que s'enfronta a un fitxer de contrasenyes amb *salt*, tot i conèixer el *salt* de cada usuari, ha de calcular el hash de cada un dels valors del diccionari de contrasenyes per cada un dels usuaris del sistema al que està atacant.

Rainbow tables

Les *rainbow tables* són unes taules construïdes per optimitzar la informació que s'emmagatzema i la que calcula un atacant que fa un atac sobre un fitxer de contrasenyes a les quals no se'ls ha afegit un *salt*. Tot i això, les *rainbow tables* no són útils amb contrasenyes desades amb *salt*.

Les *salts* acostumen a emmagatzemar-se juntament amb el hash de la contrasenya, ja que la seva funció no és pas impedir el càlcul del hash sinó evitar que l'atacant pugui reaprofitar càlculs. Quan es vol afegir un nivell més de protecció, es poden fer servir *salts* secretes (també conegudes com a *pepper*) que es desen en un altre dispositiu, diferent del que emmagatzema les contrasenyes. Així, un atacant que només té accés al fitxer de contrasenyes no pot fer l'atac, ja que no coneix les *salts* que s'han fet servir per a calcular cadascun dels hashos.

5.4.4 Derivació de claus

En criptografia és habitual l'ús de claus criptogràfiques en diferents contextos, com per exemple per a xifrar informació. Ara bé, la capacitat de les persones per a generar i recordar cadenes de zeros i uns és més aviat limitada, sobretot si aquestes cadenes són molt llargues, com podria ser una simple clau de l'AES de 128 bits. Per aconseguir que les persones puguin generar i recordar claus de forma simple, es fan servir les contrasenyes de sempre, que els usuaris estan acostumats a utilitzar, combinades amb funcions hash. Així, donada una contrasenya se li aplica una funció hash per derivar-ne una clau. Si sempre s'utilitza la mateixa funció hash, donat que aquesta és determinista, per a la mateixa contrasenya d'entrada generarà la mateixa clau. Aquesta idea simple presenta algunes debilitats de seguretat i per aquest motiu s'han dissenyat funcions específiques de derivació de claus que, això sí, és basen en una funció hash.

La funció PBKDF2

La funció ***Password-Based Key Derivation Function* (PBKDF2)** és una funció definida en l'RFC2898 que proporciona un mecanisme segur per obtenir una clau a partir d'una contrasenya.

Aquesta funció és una funció força utilitzada en diferents aplicacions, com ara per a les claus dels accessos a les xarxes WIFI (amb els protocols WPA i WPA2), en el xifrat amb AES en el WinZip i en múltiples aplicacions de programari que permeten xifrar el disc dur de l'ordinador.

La funció PBKDF2 rep com a entrada cinc paràmetres i retorna la clau que n'ha derivat. En la següent expressió s'inclouen els paràmetres que requereix la funció:

$$K = \texttt{PBKDF2(PRF, Contrasenya, Salt, c, dkLen)}$$

D'aquests paràmetres, el més evident és la contrasenya (codificada en UTF-8), que serà el valor que l'usuari proporcionarà per tal d'obtenir-ne la clau. La resta de valors són interns de cada implementació i estaran fixats per tal que cada contrasenya només pugui derivar una única clau. El valor `PRF` indica una funció pseudoaleatòria que utilitza dos paràmetres, una clau i un valor. Aquesta funció proporcionarà una sortida de mida `hLen`. Si ens hi fixem, aquesta definició de funció coincideix amb el d'una funció HMAC com la que hem definit en l'Apartat 5.4.1 i és en aquest punt on les funcions hash queden lligades a la funció de

derivació de claus. D'altra banda, el valor `Salt` és una seqüència de bits utilitzada per afegir aleatorietat al procés, com s'acostuma a fer amb els valors de salt en altres processos de seguretat. El valor `c` és un valor que determina el nombre d'iteracions que realitzarà la funció de derivació abans de proporcionar la clau. Com més gran sigui aquest valor més robusta serà la clau generada però també més trigarà la funció a calcular-la. Es recomana que aquest valor sigui, com a mínim, 1000. Finalment, el valor `dkLen` és la mida de la clau K que es vol generar.

Una vegada definits cada un dels paràmetres que utilitza la funció de generació de claus, passem a detallar-ne el seu funcionament. La següent expressió proporciona el sistema per a calcular la clau utilitzant la funció PBKDF2:

$$K = \mathtt{T}_1 \parallel \mathtt{T}_2 \parallel \cdots \parallel \mathtt{T}_{\lceil \mathtt{dkLen}/\mathtt{hLen} \rceil}$$

on el símbol $\parallel$ indica la concatenació i els valors $\mathtt{T}_i$ es descriuen a continuació.

Cada valor $\mathtt{T}_i$ el definim com $\mathtt{T}_i$ `= F(Contrasenya, Salt, c,` i`)` on la funció `F` queda explicitada en la següent expressió:

$$\mathtt{F(Password,\ Salt,\ c,}i\mathtt{)} = \mathtt{U}_1 \oplus \mathtt{U}_2 \oplus \cdots \oplus \mathtt{U}_\mathtt{c}$$

on els valors $\mathtt{U}_i$ són els següents:

$\mathtt{U}_1$ `= PRF(Contrasenya, Salt` $\parallel$ `INT_32_BE(i))`
$\mathtt{U}_2$ `= PRF(Contrasenya,` $\mathtt{U}_1$`)`
⋮
$\mathtt{U}_\mathtt{c}$ `= PRF(Contrasenya,` $\mathtt{U}_{\mathtt{c}-1}$`)`

on `INT_32_BE(i)` és l'índex i codificat com un enter de 32 bits en notació *big-endian*.

Com hem comentat anteriorment, el valor `c` és el que determina el nombre d'iteracions que es realitzaran. Fixeu-vos a més, que es pot donar el cas en que el valor `dkLen/hLen` no sigui un enter, i per tant la part entera superior de la divisió, és a dir $\lceil$`dkLen/hLen`$\rceil$, proporcionarà un nombre total de bits de la clau superior al que s'havia indicat en el valor `dkLen`. En aquest cas, la última paraula de la clau, $\mathtt{T}_{\lceil \mathtt{dkLen}/\mathtt{hLen} \rceil}$ es truncarà per la dreta per tal que la clau tingui exactament `dkLen` bits.

5.4.5 Pseudonimització de dades

Actualment les dades tenen un paper clau en molts dels sectors de la societat. L'ús de dades massives ha permès progressar en àmbits tant diversos com la medicina, les telecomunicacions o les finances, però l'ús indiscriminat de dades personals comporta problemes de privadesa que cal adreçar.

Les funcions hash s'utilitzen sovint per a pseudonimitzar identificadors en conjunts de dades, tot i que, com veurem a continuació, aquesta no sempre és una bona alternativa.

Informalment, la pseudonimització permet dissassociar la identitat d'un subjecte de les dades d'aquest. Normalment aquest procés es duu a terme substituint un o diversos identificadors per un pseudònim, per exemple, una cadena generada pseudoaleatòriament. Així, les dades queden associades a aquest pseudònim, i desvinculades de la identitat del seu propietari.

Una primera aproximació a la pseudonimització amb funcions hash consistira a substituir els identificadors d'un conjunt de dades pel resultat d'avaluar una funció hash sobre aquests.

Exemple 5.13 Exemple de pseudonimització trivial amb funcions hash

Suposem que disposem d'un conjunt de dades amb notes d'estudiants que conté els atributs DNI (sense lletra) i nota de l'estudiant en una assignatura.

La pseudonimització trivial d'aquest conjunt de dades substituiria el DNI dels estudiants pel resultat d'aplicar una funció hash al DNI. A priori, això desvincularia la identitat de l'estudiant de la seva nota.

A continuació veurem dos dels problemes d'aquesta tècnica. D'una banda, un atacant que coneix el

DNI d'un estudiant de l'assignatura, podria reidentificar el registre i aconseguir saber la nota d'aquest estudiant. Per fer-ho, simplement hauria de calcular el hash del DNI, i consultar la nota al conjunt de dades pseudonimitzat. D'altra banda, si l'atacant no coneix el DNI de cap estudiant, podria llançar un atac de força bruta per trobar els DNIs dels estudiants, aprofitant el fet que els DNIs tenen un format concret per a reduir l'espai de cerca. Els DNIs estan formats per 8 dígits, de manera que caldria calcular 10^8 hashos per a comprovar tots els DNIs possibles. Si assumim que l'atacant pot calcular un 10^6 hashos per segon (cosa que es podria fer amb qualsevol ordinador sense hardware especialitzat), l'atac tardaria menys de dos segons.

Dues alternatives més robustes per a pseudonimitzar identificadors consisteixen en l'ús de MACs (*Message Authentication Code*) o la incorporació de *salts* secretes. En ambdues alternatives, a cada identificador li corresponen diversos pseudònims, depenent de la clau o la *salt* utilitzada. La clau o la *salt* secreta es mantenen separats de les dades, de manera que un atacant que només diposa de les dades no pot reidentificar-ne els registres reproduint el procés que s'ha dut a terme per calcular els pseudònims.

Exemple 5.14 Exemple de pseudonimització amb *salt* secreta o *pepper*

Suposem que disposem del mateix conjunt de dades que a l'exemple anterior, i que s'aplica un procés de pseudonimització substituint el DNI pel hash d'una *salt* secreta (generada pseudoaleatòriament) concatenada al DNI.

La Figura següent mostra un exemple del procediment, fent servir SHA-256 com a funció hash. Les dades originals són dividides en dos conjunts de dades diferents, que seran desades separadament. D'una banda, es desarà la *salt* de cada registre, que s'haurà generat pseudoaleatòriament (Taula 1.3). D'altra banda, es desaran les notes, associades al pseudònim, que s'haurà calculat com el SHA-256 de la concatenació de la *salt* i el DNI.

Table 1.2: Conjunt de dades original

DNI	Nota
50705923	5
88046921	9.8
20091322	7
64802452	2

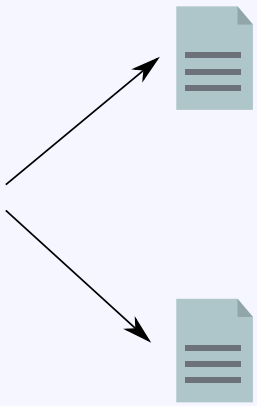


Table 1.4: Conjunt de dades pseudonimitzat

Pseudoidentificador	Nota
`0x340fff ...`	5
`0x1648da ...`	9.8
`0x40e616 ...`	7
`0x9d480f ...`	2

Table 1.3: Salts secretes

Salt
56714
78910
28285
92670

Aquesta tècnica permet adreçar les limitacions de la tècnica trivial. Ara, si un atacant accedeix al conjunt de dades pseudonimitzades, i coneix un dels DNIs dels estudiants de l'assignatura, no podrà reidentificar el registre que correspon a aquest estudiant, ja que no podrà recrear el hash. Anàlogament, un atacant tampoc podrà calcular els pseudònims de tots els DNIs existents, encara que sàpiga que aquests estan formats per 8 dígits.

Ara bé, cal anar molt en compte a l'aplicar aquest tipus de tècniques per pseudonimitzar dades. De fet, en podríem fer una assignatura sencera només explicant com fer-ho! Per exemple, suposem que el conjunt de dades conté les notes de diverses activitats:

Pseudoidentificador	Act. 1	Act. 2	Act. 3
`0x340fff...`	5	6	4
`0x1648da...`	10	9	7
`0x40e616...`	7	7	3
`0x9d480f...`	2	5	3

i que l'atacant sap quin estudiant és l'únic que ha aprovat la tercera activitat de l'assignatura (per exemple, pot ser que sigui l'únic que va sortir content de classe després de saber-se les notes d'aquesta activitat). En aquest cas, l'atacant serà capaç de reindentificar el registre de l'estudiant, i obtenir així informació addicional sobre aquest (en particular, les notes exactes de totes les activitats).

5.4.6 Generació de cadenes de bits pseudoaleatòries

La criptografia requereix sovint de l'ús de cadenes de bits generades aleatòriament, per exemple, en la generació de claus, de vectors d'inicialització o de valors per a reptes en protocols interactius. Per tal de generar cadenes de bits aleatòries, es disposa principalment de dues estratègies: o bé es fa servir una font d'aleatorietat a partir d'algun procés físic que no sigui predictible, o bé es calculen els bits de manera determinista amb un algorisme a partir d'una llavor. Les tècniques que fan servir aquesta segona estratègia s'engloben sota el nom de generadors de nombres pseudoaleatoris o PRNG (per les seves inicials en anglès, *PseudoRandom Number Generator*) i de les cadenes de bits que generen diem que són pseudoaleatòries.

Els PRNG són algorismes deterministes que produeixen una seqüències de bits a partir d'una llavor, que s'ha d'obtenir d'una font aleatòria. Quan la llavor és secreta, els bits que genera un PRNG no són predictibles.

Es poden construir PRNG a partir de funcions hash. Així, per exemple, el NIST defineix un PRNG basat en funcions hash anomenat Hash_DRBG (DRBG són les inicials de *Deterministic Random Bit Generator*, un altre terme per referir-se als PRNG). L'algorisme Hash_DRBG emmagatzema un estat format per una variable i una constant (V i C, respectivament) i un comptador. Inicialment, V i C es deriven de la llavor aleatòria. Després, el valor de la variable V es fa servir per derivar els bits pseudoaleatoris, i s'actualitza el valor d'aquesta variable (en aquesta actualització es fa servir la constant C). Cada vegada que es generen nous bits, s'incrementa el comptador de l'estat intern. Quan aquest comptador arriba a un llindar prestablert, cal tornar a introduir aleatorietat a l'algorisme per tal de seguir generant bits pseudoaleatoris, procés que es coneix com a ressembrat (de l'anglès, *reseeding*).

5.4.7 Compromís de bit

Hi ha situacions quotidianes en les que estem acostumats a fer servir alguns mecanismes molt simples que funcionen sense cap dificultat d'execució. Un d'aquests casos és el de 'tirar una moneda a l'aire' per, per exemple, decidir quin dels dos jugadors d'una partida d'escacs tindrà les fitxes blanques. Ara bé, quan les dues parts que duen a terme aquest petit protocol no es troben físicament al mateix lloc, la simplicitat de tirar una moneda a l'aire no ens serveix en el cas que hi hagi certa desconfiança entre els dos participants.

Si analitzem el procés de tirar una moneda a l'aire veiem que, normalment, un dels dos usuaris tria cara o creu i l'altre, una vegada s'ha decidit qui guanyarà segons el revers de la moneda, tira la moneda a l'aire. En aquest simple esquema, l'usuari que tria cara o creu ho fa de forma pública, de manera que després (quan cau la moneda) no pot dir que ha triat una altra cosa. I l'usuari que tira la moneda no pot fer trampa (assumint que la moneda no està trucada!) perquè tira la moneda davant de l'altre usuari i els dos veuen el resultat que en surt, de manera que qui tira la moneda no pot canviar-ne el resultat.

Per emular aquest protocol de forma remota (o digital) es fa servir un esquema de compromís de bit.

Un **esquema de compromís de bit** (en anglès, *bit commitment*) és una tècnica per la qual un usuari A es compromet, davant d'un usuari B, a un valor m per mitjà d'un valor $C(m)$, que serà el compromís. Aquest compromís ha de tenir les següents propietats:

1. Donat el compromís $C(m)$, B no pot obtenir informació del valor compromès m.
2. A ha de poder obrir el compromís $C(m)$ mostrant el valor compromès m.
3. A no pot obrir el compromís $C(m)$ mostrant un valor diferent al valor m compromès inicialment.

Amb un esquema de compromís de bit com el que acabem de descriure, el protocol de tirar una moneda a l'aire es pot definir amb els següents passos.

1. L'usuari A tria cara o creu i codifica la seva tria en el missatge m. Posteriorment, calcula el compromís d'm, $C(m)$, i l'envia a B.
2. B genera aleatòriament un bit, on 1 correspondrà al valor cara i 0 correspondrà a creu. B enviarà a A el valor aleatori generat.
3. A obrirà el compromís $C(m)$ mostrant a B quin valor (cara o creu) havia triat, de manera que es veurà qui ha guanyat en el protocol de tirar una moneda a l'aire.

Fixeu-vos que en el pas 2 del protocol, l'usuari A ja ha triat cara o creu però l'usuari B, tot i tenir el compromís $C(m)$, no pot saber quin valor ha triat (gràcies a la primera propietat de l'esquema de compromís de bit). En el pas 2, tot i que l'usuari B no generés el bit de forma aleatòria (per intentar alterar el protocol) el fet que no coneix si A ha triat cara o creu fa que la tria d'aquest valor aleatori sigui intrascendent. D'altra banda, en el pas 3, A ja sap quin valor ha obtingut B i per tant B no pot desdir-se'n. A més, A obre el seu compromís i, tot i conèixer el valor obtingut per B, no pot obrir-lo mostrant un altre valor diferent al que s'ha compromès, gràcies a la tercera propietat de l'esquema de compromís de bit.

Els protocols de compromís de bit es descriuen per mitjà de dues fases: fase de generació del compromís i fase d'obertura del compromís i en els següents apartats veurem dues tècniques diferents que implementen un esquema de compromís de bit.

Compromís de bit utilitzant funcions hash

Una de les tècniques més utilitzades per implementar un esquema de compromís de bit és mitjançant una funció hash.

Sigui m el missatge al qual l'usuari es vol comprometre, en la **fase de generació del compromís** l'usuari A selecciona un valor aleatori r i calcula $C(m) = h(r \parallel m)$ on h és una funció hash criptogràfica.

En la **fase d'obertura del compromís** $C(m)$, l'usuari A revela els valors r i m. A partir d'aquests valors, l'usuari B pot calcular $h(r \parallel m)$ i comprovar que efectivament coincideix amb el valor $C(m)$ al qual A s'havia compromès.

Comprovem que aquest esquema compleix amb les tres propietats d'un esquema de compromís de bit.

1. B no pot obtenir el valor compromès m a partir el compromís $C(m)$ ja que $h(\cdot)$ és una funció hash criptogràfica i per tant no es pot invertir. Fixeu-vos que el valor aleatori r s'utilitza en cas que el missatge m se seleccioni d'un conjunt petit de missatges, per tal d'evitar que B pugui calcular totes les imatges de la funció hash per a tots els possibles valors diferents d'm i descobrir-ne el valor compromès.
2. A pot obrir el compromís $C(m)$ fent públics els valors r i m.
3. A no pot obrir el compromís, $C(m)$, obtenint un valor $m' \neq m$ perquè això voldria dir que A pot trobar $(r \parallel m) \neq (r' \parallel m')$ tal que $h(r \parallel m) = h(r' \parallel m')$ i això no és possible per les propietats que hem enumerat de la funció hash criptogràfica que s'utilitza.

5.4.8 Prova de treball

En l'execució d'alguns protocols criptogràfics, en ocasions, és necessari assegurar que un participant realitza un cert esforç de càlcul abans de poder realitzar una operació per tal que l'operació en qüestió no sigui fàcil de realitzar de forma automàtica i repetitiva. Aquests tipus de mecanismes s'anomenen prova de treball.

> Una **prova de treball** (en anglès *proof-of-work*), és un mecanisme que permet a l'usuari d'un sistema demostrar a la resta d'usuaris de forma fidedigna que ha realitzat una certa quantitat de feina, normalment, una certa quantitat de càlculs.

El concepte de prova de treball el van proposar Cyntia Dwrok i Moni Naor en un article publicat al congrés *Crypto* l'any 1992 però no va ser fins més tard, l'any 1999, que M. Jakobsson i A. Juels van formalitza-lo i van proposar-ne el terme *proof-of-work*.

Les aplicacions de les proves de treball són variades i van des de la prevenció de correu brossa fins al manteniment d'integritat en els sistemes de criptomonedes.

La propietat més important d'una prova de treball és la seva asimetria, en el sentit que el cost de realització de la prova de treball s'ha de poder prefixar de forma arbitrària, però la verificació de la prova de treball, independentment de la dificultat fixada en el cost, ha de ser extremadament eficient i, per tant, no ha de requerir tornar a realitzar els càlculs que s'han de realitzar per produir-la. És per aquest motiu que les funcions unidireccionals utilitzades en criptografia, com ara les funcions hash, són una bona base per a la creació de proves de treball.

Una de les proves de treball més utilitzades en l'actualitat, ja que moltes de les criptomonedes existents la fan servir, és el Hashcash, una prova de treball proposada per A. Back l'any 1997 per tal de limitar el correu brossa i, en general, altres atacs de denegació de servei. Aquesta prova de treball consisteix a calcular el valor hash d'una certa informació i aconseguir que la imatge resultant sigui un valor inferior a un cert llindar. Per a fer-ho, cal habilitar un camp aleatori en la informació en qüestió per tal de poder-lo variar per obtenir-ne diferent valors hash.

Per exemple, una simplificació del sistema anti-correu brossa basat en aquesta prova de treball seria el següent. Quan l'usuari A vol enviar un correu a l'usuari B, un cop generat tot el missatge, inclosa l'adreça del destinatari, l'usuari A afegeix a la capçalera un nou camp, que podrà contenir qualsevol valor aleatori. Amb tota aquesta informació, A en calcularà la imatge per una funció hash determinada, que haurà consensuat amb B. Prèviament, A i B hauran també fixat quin és l'esforç (en la prova de treball) que A ha de fer per enviar-li un correu a B. Aquest esforç s'explicitarà triant un **valor objectiu** concret d'entre totes les imatges possibles de la funció hash. Abans de processar el correu, l'usuari B comprovarà si el hash del missatge que ha rebut d'A és inferior al valor objectiu. En cas afirmatiu, processarà el correu, en cas negatiu el descartarà. Fixeu-vos que una vegada A ha redactat el correu, si al realitzar el càlcul del hash n'obté un valor superior al valor objectiu, no pot enviar el missatge (ja que B el descartaria). Abans de fer-ho ha de modificar el nou camp que ha afegit a la capçalera amb un valor aleatori i tornar a calcular-ne el hash. Si és menor al valor objectiu, ja podrà enviar-lo, però si no ho és haurà de tornar a modificar el valor del camp, tornar a calcular el hash i anar repetint aquesta operació fins que el hash del correu sigui més petit que el valor objectiu. Fixeu-vos que la mida del valor objectiu fixarà la dificultat de la prova de treball, com més petit sigui el valor objectiu, més feina haurà de fer A per enviar el missatge a B.

Fixeu-vos que la necessitat que té l'emissor del missatge per enviar-lo fa que si aquest emissor és un generador de correu brossa, per enviar cada correu brossa li sigui necessari realitzar un cert volum de càlcul per a cada correu (ja que el destinatari del correu forma part de la informació que s'inclou en el hash i per tant no pot reaprofitar els càlculs d'un altre correu) i per tant es desincentiva aquest pràctica.

Exemple 5.15 La dificultat de la prova de treball i les probabilitats

Les propietats estadístiques de les funcions hash criptogràfiques fan que la seva sortida es pugui considerar un generador pseudoaleatori en el sentit que donada una entrada no se'n pot predir la sortida i una mínima modificació de l'entrada provoca una modificació significativa del valor de la sortida. Amb aquesta premissa, suposem una funció hash de mida 3 dígits, és a dir, el resultat d'aplicar aquesta funció hash a un missatge ens pot donar un valor entre el 0 i el 999, és a dir 1000 valors. Així, la probabilitat que donada una entrada m el seu valor hash siguin un nombre menor que 1000 serà 1, ja que qualsevol sortida ens donarà un d'aquests valors. Ara bé, si fixem el valor objectiu de la nostra prova de treball en 500, la probabilitat que l'entrada d'aquesta funció sigui menor que 500 és de $\frac{1}{2}$. I si el valor objectiu és 100, la probabilitat és de $\frac{1}{10}$. En aquest últim cas, fixeu-vos que un emissor del sistema Hashcash que vulgui enviar un correu, haurà de regenerar el valor aleatori i recalcular el hash 10 vegades, en mitjana, fins a obtenir una sortida inferior al valor objectiu i per tant un correu que sigui acceptat pel receptor. Per tant, com més petit és el valor objectiu, més dura és la prova de treball.

Aquest mateix mecanisme de prova de treball es el que fan servir moltes criptomonedes, com ara els Bitcoins, per assegurar que un usuari no pot gastar de nou uns diners que ja havia gastat prèviament.

Exercici 5.6 Tenim un sistema que utilitza una prova de treball a través d'una funció hash. Aquesta funció hash té una mida de 64 bits i la potència de càlcul de la xarxa que l'utilitza està fixada en 100.000 hashos per segon. Fixeu un valor objectiu de la funció hash per tal que, amb la potència de càlcul que s'indica, es trobi una imatge del hash menor que valor objectiu, en mitjana, cada 10 minuts.

5.4.9 Taules hash

Les funcions hash també s'utilitzen molt sovint com a primitives en la creació d'estructures de dades. Aquestes aplicacions es fan servir a vegades en el context de la seguretat de la informació, però també trascendeixen a altres contextos de les ciències de la computació. A continuació presentarem tres estructures de dades basades en funcions hash: les taules hash, els arbres de Merkle, i els filtres de Bloom.

Les taules hash són una estructura de dades utilitzada per a implementar diccionaris (també coneguts com a arrays associatius), és a dir, estructures que emmagatzemen parells no ordenats de clau-valor, on les claus són úniques. Les taules hash permeten inserir, buscar i eliminar elements de manera eficient.

Una **taula hash** és una estructura de dades que implementa un diccionari o array associatiu.

Exemple 5.16 Exemple de diccionari o array associatiu

Els diccionaris són estructures que es fan servir sovint en programació. A continuació es llisten un parell d'exemples de dades que es poden desar en un diccionari:

- Un diccionari pot utilitzar-se per a desar el hash de la contrasenya d'un conjunt d'usuaris d'un sistema. Les claus del diccionari contindran els identificadors dels usuaris (de manera que no hi ha claus repetides) i el valor associat a cada clau serà el hash de la seva contrasenya.
- Un diccionari pot emmagatzemar els prefixos telefònics de cada província. Les claus del diccionari contindran el nom de la província (que és únic) i el valor associat a cada clau serà el prefix telefònic d'aquella província.

Les taules hash fan servir una funció hash per calcular l'índex d'un element a partir de la seva clau. Aquest

índex indica a on està desat l'element.

Així, els procesos d'**afegir**, **eliminar** o **buscar** un element a la taula hash consten d'una primera fase comuna, que consisteix en calcular l'índex de l'element. La segona fase és específica per a cada procés i consisteix a fer l'acció especificada, és a dir, escriure un nou element a la taula, eliminar-ne un d'existent, o bé comprovar si un element hi és.

Exemple 5.17 Exemple de taula hash

Seguint amb l'exemple de les contrasenyes, suposem que disposem del següent diccionari:

```
{
  "morpheus":  0x4c9a82ce72ca2519f38d0af0abbb4cecb9fceca9,
  "neo":       0x356a192b7913b04c54574d18c28d46e6395428ab,
  "trinity":   0x7110eda4d09e062aa5e4a390b0a572ac0d2c0220
}
```

i que el volem implementar amb una taula hash que fa servir com a índex els 4 primers bits del SHA-256 de la clau de cada element. Procedim a calcular l'índex de cada element:

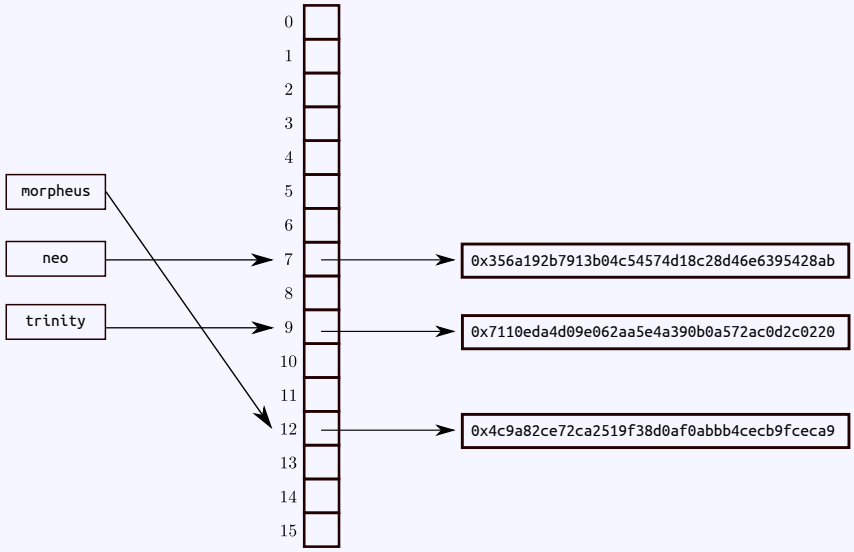
$$\begin{aligned} SHA256(\texttt{morpheus}) &= \texttt{0xc1a1e4aa ...} \\ i(\texttt{morpheus}) &= SHA256(\texttt{morpheus})_{0..3} = \texttt{0xc} = 12 \end{aligned}$$

$$\begin{aligned} SHA256(\texttt{neo}) &= \texttt{73ef176d ...} \\ i(\texttt{neo}) &= SHA256(\texttt{neo})_{0..3} = \texttt{0x7} = 7 \end{aligned}$$

$$\begin{aligned} SHA256(\texttt{trinity}) &= \texttt{0x934a11e6 ...} \\ i(\texttt{trinity}) &= SHA256(\texttt{trinity})_{0..3} = \texttt{0x9} = 9 \end{aligned}$$

on l'expressió $X_{i..j}$ denota els bits des de la posició i a la posició j del valor X.

Aleshores, la taula hash quedaria de la manera següent:



La versió de la taula hash que acabem d'explicar té però un problema evident: no és capaç de gestionar les col·lisions d'índexos. És a dir, si dos elements tenen el mateix índex, no es podran emmagatzemar tots dos, ja que en cada posició només s'hi desa un element. Això és un problema important, que es pot adreçar de diverses maneres.

Una alternativa per a adreçar les col·lisions en taules hash és l'ús de llistes enllaçades. Així, cada posició de la taula hash apunta al primer element d'una llista enllaçada, que contindrà tots els elements que comparteixin el mateix índex. En aquesta variant, els processos de cerca, inserció i eliminació fan ús, per tant, de dues estructures de dades. D'una banda, calculen l'índex de l'element a la taula hash i, d'altra banda, operen sobre la llista enllaçada per tal de cercar, inserir o eliminar elements.

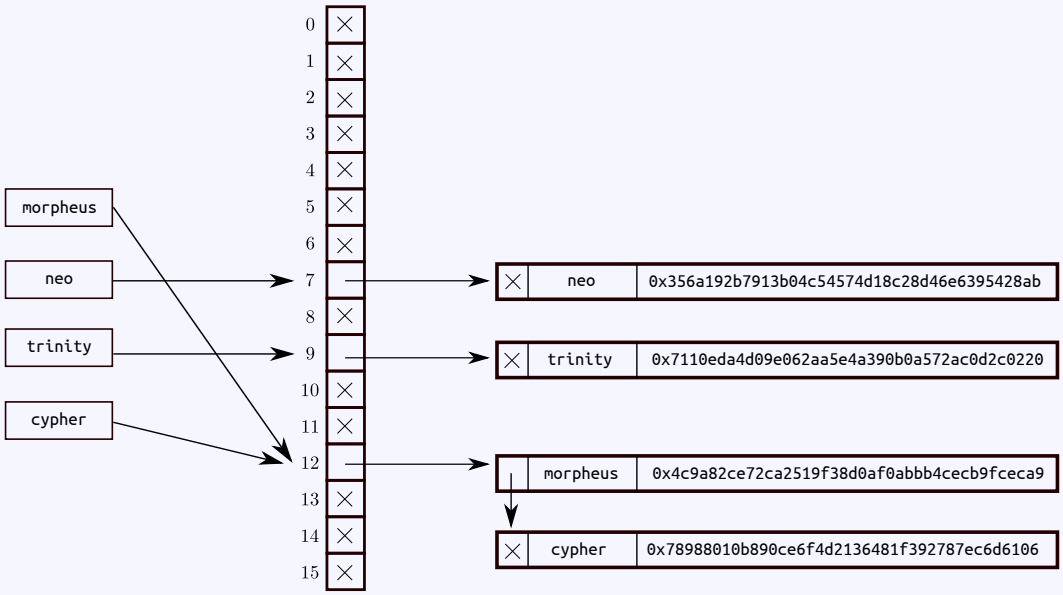
Exemple 5.18 Exemple de taula hash amb llista enllaçada

Suposem que fem servir una taula hash amb llista enllaçada per emmagatzemar els mateixos elements que a l'exemple anterior, més la contrasenya d'un usuari nou, en *cypher*.

En primer lloc, calculem l'índex del nou element:

$$SHA256(\texttt{cypher}) = \texttt{0xc9d22bd2 ...}$$
$$i(\texttt{cypher}) = SHA256(\texttt{cypher})_{0..3} = \texttt{0xc} = 12$$

Per tant, la taula hash quedaria ara:



Fixeu-vos que ara hem de desar tant la clau com el valor de cada element, ja que hem de poder distingir les contrasenyes de diferents usuaris que comparteixen índexos.

Triar la funció hash adequada per a implementar una taula hash és una tasca complicada i, alhora, crítica. Cal tenir en compte tant la distribució de valors com el rendiment de l'estructura de dades. Sovint es fan servir funcions hash no criptogràfiques per a implementar taules hash, ja que la seva avaluació és molt més ràpida. Així, la funció hash a utilitzar dependrà dels requeriments de l'escenari en què es desplegui la taula hash.

5.4.10 Arbres de Merkle

Els arbres de Merkle van ser proposats l'any 1979 per Ralph Merkle, de qui en deuen el nom, i permeten desar un resum d'un conjunt de dades, de tal manera que es pugui demostrar que una dada pertany a aquest conjunt eficientment.

> Una **arbre de Merkle** és un arbre en el qual cada fulla conté el hash d'un bloc de dades, i els nodes interns contenen el hash de la concatenació dels valors dels seus fills.

Habitualment, els arbres de Merkle són binaris, és a dir, cada node intern té com a molt dos fills.

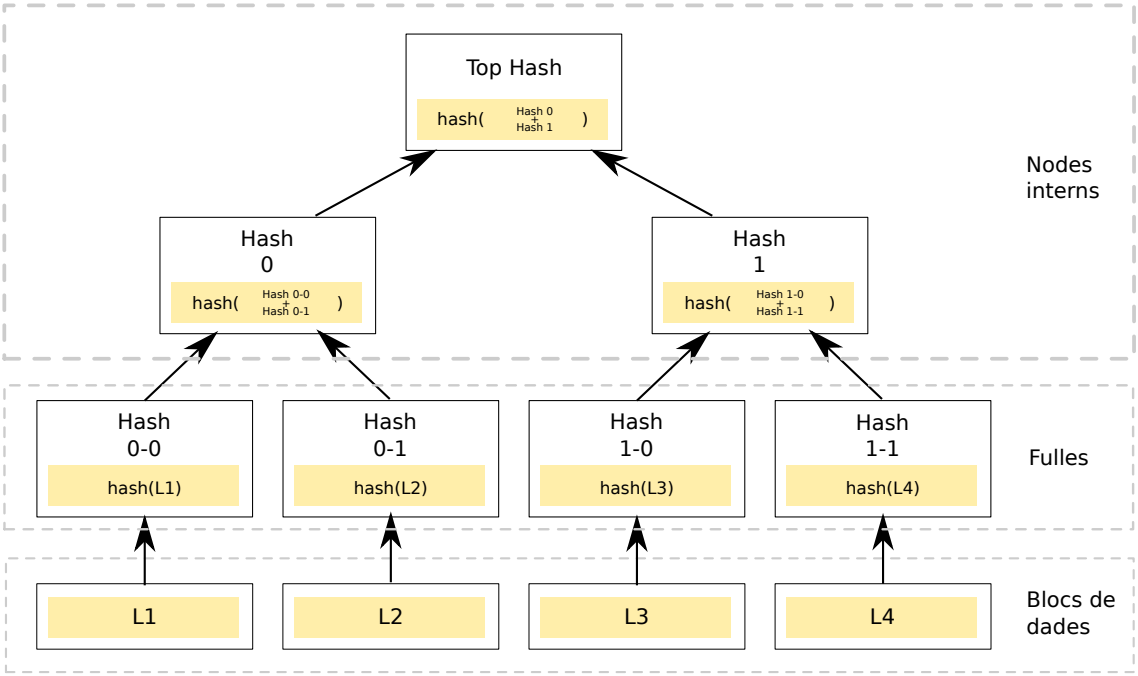


Figura 5.8: Exemple d'un arbre de Merkle. Il·lustració original de David Göthberg, sota llicència CC0. 1.0.

Exemple 5.19 Exemple d'arbre de Merkle

La Figura 5.8 mostra un arbre de Merkle per a quatre blocs de dades $(L_1, \cdots, L_4)$.

L'arbre té quatre fulles (els nodes 0-0, 0-1, 1-0 i 1-1), que contenen el hash de cadascun dels blocs de dades, és a dir, $h_{00} = H(L_1)$, $h_{01} = H(L_2)$, $h_{10} = H(L_3)$ i $h_{11} = H(L_4)$.

El segon nivell de l'arbre té dos nodes: el node 0 conté el hash de la concatenació dels nodes 0-0 i 0-1 ($h_0 = H(h_{00}||h_{01})$) i el node 1 conté el hash de la contatenació dels nodes 1-0 i 1-1 ($h_1 = H(h_{10}||h_{11})$).

El primer nivell conté l'arrel de l'arbre, un únic node que desa el hash de la concatenació dels nodes 0 i 1 ($h_r = H(h_0||h_1)$).

Exercici 5.7 Calculeu el hash de l'arrel de l'arbre de Merkle per al següent conjunt de blocs de dades fent servir SHA-256 com a funció hash.

$L_1 =$ `GREAT PYRAMID OF GIZA`
$L_2 =$ `COLOSSUS OF RHODES`
$L_3 =$ `HANGING GARDENS OF BABYLON`
$L_4 =$ `LIGHTHOUSE OF ALEXANDRIA`
$L_5 =$ `MAUSOLEUM AT HALICARNASSUS`
$L_6 =$ `STATUE OF ZEUS AT OLYMPIA`
$L_7 =$ `TEMPLE OF ARTEMIS AT EPHESUS`
$L_8 =$ `MILFORD SOUND`

Els arbres de Merkle són utilitzats per realitzar **proves de pertinença** a un conjunt de manera eficient. Suposem que tenim un conjunt d'n blocs de dades, $\mathscr{L} = \{L_1, \cdots, L_n\}$, i que volem generar-ne un resum, de manera que posteriorment puguem demostrar que els elements L_i (amb $i = 1 \cdots n$) pertanyen al conjunt $\mathscr{L}$ eficientment. En primer lloc, calcularíem el resum h_r, que correspondria al hash de l'arrel de l'arbre de Merkle amb els blocs de dades d'$\mathscr{L}$ a les fulles. Aquest resum h_r seria l'únic valor que caldria que el verificador desés per tal de poder comprovar, posteriorment, que qualsevol dels blocs L_i pertany a $\mathscr{L}$. Per tal de demostrar que un bloc L_i pertany al conjunt $\mathscr{L}$, el provador genera una prova Π que conté el bloc L_i i els hashos de tots els nodes germans que hi ha en el camí des del node L_i a l'arrel de l'arbre h_r. El verificador pot comprovar que la prova és correcta calculant el hash del bloc L_i i reconstruint l'arbre de Merkle amb els hashos dels germans proporcionats a la prova. Si l'arrel de l'arbre de Merkle calculat és igual a l'arrel h_r que havia emmagatzemat, la prova és correcta, i el verificador queda convençut que $L_i \in \mathscr{L}$.

Exemple 5.20 Exemple de prova de pertinença amb arbre de Merkle

Seguint amb l'exemple de la figura 5.8, suposem que h_r és el valor del hash de l'arrel de l'arbre, i que volem crear una prova de pertinença per al bloc L_3. La prova de pertinença per a L_3 seria $\Pi = (L_3, h_{11}, h_0)$.

Per tal de verificar la prova de pertinença, el verificador procediria a calcular:

$$h_{10} = H(L3)$$

$$h_1 = H(h_{10}||h_{11})$$

$$h_{r'} = H(h_0||h_1)$$

Si $h_{r'} = h_r$, aleshores la prova de pertinença seria satisfactòria. En cas contrari, es rebutjaria la prova.

Un detall a notar és que per tal de verificar la prova de pertinença, el verificador ha de saber en quin ordre concatenar els hashos a cada nivell. Aquesta informació es pot incloure a la prova de pertinença, afegint un únic bit per a cada element que indiqui si és el fill dret o l'esquerra, o bé indicant explícitament la posició del node dins de l'abre.

Exercici 5.8 Genereu una prova de pertinença del bloc de dades `MILFORD SOUND` per a l'arbre de Merkle de l'Exercici 5.7. Valideu la prova generada.

D'una banda, la prova de pertinença que acabem de descriure és eficient i concisa. Independentment del número de blocs de dades n i de la seva mida, el resum h_r a guardar per tal de poder verificar les proves de

pertinença és petit, i té una mida constant (que correspondrà a la mida de sortida de la funció hash que es faci servir). Addicionalment, la prova de pertinença conté únicament $\log_2 n$ elements de mida constant (la mida de sortida del hash), més el bloc de dades a verificar. Finalment, el còmput a realitzar per fer la verificació és també de $\log_2 n + 1$ hashos (el hash del bloc de dades i un hash per cada element de la prova).

D'altra banda, si la funció hash que es fa servir en la construcció de l'arbre de Merkle és una funció hash criptogràfica, aleshores un atacant no podrà construir una prova de pertinença falsa, és a dir, no podrà convèncer al verificador que un bloc $L_j \notin \mathscr{L}$ sí que pertany a $\mathscr{L}$. Fixeu-vos que, per a crear una prova de pertinença falsa, l'atacant hauria de ser capaç de crear un nou bloc $L_j \notin \mathscr{L}$ que tingués el mateix hash que algun dels blocs $L_i \in \mathscr{L}$, és a dir, trobar un $L_j \notin \mathscr{L}$ tal que $H(L_j) = H(L_i)$ per a algun $i \in 1 \cdots n$. Això no és possible ja que una funció hash criptogràfica és resistent a segones preimatges. Una altra estratègia que podria seguir l'atacant és modificar els valors dels hashos germans que conformen la prova de pertinença, per tal d'intentar que el hash de l'arrel de l'arbre de Merkle calculat coincideixi amb l'emmagatzemat, h_r. De nou, això no és possible si la funció hash és criptogràfica, ja que suposaria crear segones preimatges (amb restriccions addicionals sobre el seu contingut).

Les proves de pertinença en arbres de Merkle que hem presentat permeten a un provador demostrar a un verificador que un determinat bloc de dades pertany a un conjunt. Ara bé, tal com les hem presentat, aquestes proves no serveixen per a demostrar el contrari, és a dir, que un bloc de dades no pertany al conjunt. Fixeu-vos que si la prova de pertinença falla, el verificador no pot assegurar que el bloc no es troba present (el bloc pot ser-hi però en una altra posició, o bé els hashos germans presentats poden ser erronis). Una petita variant dels arbres de Merkle permet provar que un element no està en un conjunt.

Una **arbre de Merkle ordenat** és un arbre de Merkle en el qual els blocs de dades de les fulles es troben ordenats, de manera que $L_1 < L_2 < \cdots < L_n$.

Els arbres de Merkle ordenats es poden fer servir per a fer proves de no pertinença. Com en el cas de les proves de pertinença, calcularem el hash de l'arrel de l'arbre, h_r, que serà l'únic valor que el verificador haurà de desar per poder verificar les proves. Ara, per tal de crear una prova que demostri que un bloc de dades L_j no pertany a $\mathscr{L}$, en primer lloc cal localitzar els blocs L_i i L_{i+1} tals que $L_i < L_j < L_{i+1}$. La prova de no pertinença $\bar{\Pi}$ consistirà en els dos blocs de dades, L_i i L_{i+1}, juntament amb les proves de pertinença de cadascun d'ells.

A partir d'aquesta prova $\bar{\Pi}$, es pot verificar que un bloc L_j no pertany a $\mathscr{L}$ de la següent manera. En primer lloc, es comprova que efectivament $L_i < L_j < L_{i+1}$. A continuació, es calculen els hashos dels blocs de dades L_i i L_{i+1}, i es validen les proves de pertinença de casdascun d'aquests blocs. Finalment, es comprova que els blocs de dades L_i i L_{i+1} són blocs consecutius, és a dir, que es troben un immediatament a continuació de l'altre en les fulles de l'arbre de Merkle. Aquesta última comprovació es fa validant la posició que ocupen els blocs de dades en l'arbre, que es pot derivar de l'ordre en què cal concatenar els hashos per tal d'aconseguir obtenir el hash de l'arrel esperat.

També es poden generar proves de no pertinença per a valors L_j inferior a L_1 o superiors a L_n, amb una petita variant del protocol que acabem de presentar.

Exemple 5.21 Exemple de prova de no pertinença amb arbre de Merkle

Seguint amb l'exemple de la Figura 5.8, suposem que h_r és el valor del hash de l'arrel de l'arbre i que els blocs de dades L_i emmagatzemen els següents enters: $L_1 = 31, L_2 = 37, L_3 = 41, L_4 = 43$. L'arbre de Merkle és un arbre ordenat, ja que $L_1 < L_2 < \cdots < L_n$. En aquest exemple crearem una prova de no pertinença per al bloc 42.

En primer lloc, es localitzen els dos blocs de dades consecutius entre els quals es trobaria el bloc de dades 42, que són L_3 i L_4 (ja que $L_3 < 42 < L_4$ i 3 i 4 són consecutius).

La prova de no pertinença seria $\bar{\Pi} = (42, (L_3, h_{11}, h_0), (L_4, h_{10}, h_0))$.

Per tal de verificar la prova de no pertinença, en primer lloc el verificador comprovaria que $41 < 42 < 43$.

Després, procediria a calcular:

$$h_{10} = H(L3)$$
$$h_1 = H(h_{10}||h_{11})$$
$$h_{r'} = H(h_0||h_1)$$

$$h_{11} = H(L4)$$
$$h_1 = H(h_{10}||h_{11})$$
$$h_{r'} = H(h_0||h_1)$$

i validaria que els valors h'_r obtinguts coincideixen amb l'h_r que té emmagatzemat.

Finalment, comprovaria que les fulles en posicions L_3 i L_4 són consecutives.

Si les tres verificacions són satisfactòries, aleshores el provador pot estar segur que 42 no pertany a $\mathscr{L}$.

Els arbres de Merkle es fan servir per a fer proves de pertinença o no pertinença en diversos contextos. Per exemple, es fan servir en la criptomoneda Bitcoin per a que clients lleugers (com podrien ser els que s'executen en un dispositiu mòbil) puguin validar la inclusió de transaccions en els blocs que formen la cadena de blocs (la *blockchain*), sense haver d'emmagatzemar la cadena de blocs sencera (que ocupa diversos centenars de gigabytes). Cada bloc de la cadena conté l'arrel de l'arbre de Merkle de totes les transaccions que s'hi emmagatzemen. Quan un client lleuger necessita comprovar si una transacció s'ha inclòs en un bloc (per exemple, per saber si ha rebut un pagament), el client demana una prova de pertinença de la transacció a la cadena de blocs. Aleshores, un servidor que sí que disposa de totes les dades, genera la prova de pertinença per a la transacció i l'envia al client, que la valida reconstruint l'arbre de Merkle. D'aquesta manera, el client pot estar segur que la transacció s'ha inclòs a la cadena de blocs, ja que el servidor no pot falsificar la prova.

5.4.11 Filtres de Bloom

Els filtres de Bloom van ser proposats l'any 1970 per Burton Howard Bloom. Són estructures de dades que permeten fer testos de pertinença fent servir molt poc espai d'emmagatzemament però, a diferència dels arbres de Merkle, són estructures probabilístiques, que poden retornar resultats erronis (amb una certa probabilitat).

> Un **filtre de Bloom** és una estructura de dades probabilística que permet fer testos de pertinença aproximats fent un ús eficient de l'espai d'emmagatzemament.

Un filtre de Bloom pot retornar falsos positius però mai falsos negatius. És a dir, la resposta a una consulta de pertinença amb un filtre de Bloom serà o bé que l'element no es troba en el filtre o bé que probablement sí que hi és.

Un filtre de Bloom f està format per:

1. un vector binari V d'n bits,
2. i un conjunt de k funcions hash independents $h_1, h_2, \cdots, h_k$ que tenen rang $[0, n-1]$.

El procés de **creació del filtre** consisteix a seleccionar els paràmetres del filtre (la mida del vector n, el nombre de funcions hash k i les k funcions hash a utilitzar) i a inicialitzar el vector, assignant 0 a totes les posicions.

Per tal d'**afegir un element al filtre** es procedeix de la manera següent. Primer, s'apliquen les k funcions hash a l'element, obtenint k valors entre 0 i $n-1$ (un valor per a cada funció hash). A continuació, s'assignen a 1 les k posicions del vector indicades per les sortides de les funcions hash. Definirem doncs la funció d'afegir un element e al filtre de Bloom com:

$$V[h_i(e)] = 1 \quad \forall i \in [1,k]$$

on $V[j]$ és la posició j del vector V.

Aquest procediment es repeteix per a tots els elements a afegir al filtre, procés en el qual es poden generar col·lisions. És a dir, es pot haver d'assignar un 1 a una posició que ja havia estat fixada a 1 per un altre element. La freqüència de les col·lisions vindrà determinada per la mida del filtre i el nombre de funcions hash utilitzades. En aquest cas, si un dels bits a assignar a 1 ja és 1, no caldrà modificar-lo, i seguirà sent 1.

Per tal de **comprovar si un element e es troba en el filtre**, s'apliquen de nou les k funcions hash a l'element, i es comprova si totes les posicions del vector binari indicades per les sortides de les funcions hash són 1. Si alguna de les posicions indicades conté un 0, aleshores direm amb tota seguretat que l'element no pertany al filtre. En canvi, si totes les posicions contenen un 1, aleshores direm que l'element pertany al filtre, tot i que en aquest cas només podrem afirmar-ho amb certa probabilitat. Així, doncs, definim la funció p que retorna 1 si l'element e es troba en el filtre f i 0 en cas contrari com a:

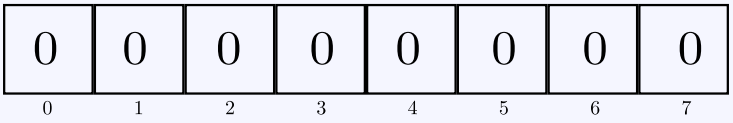
$$p(e,f) = \prod_{i=1}^{k} V[h_i(e)]$$

És interessant notar perquè un filtre de Bloom mai no dóna falsos negatius. La funció p retornarà 0 si alguna de les posicions indicades per les funcions hash són 0. En aquest cas, tenint en compte que quan s'afegeixen els elements les posicions es marquen amb 1, podem estar segurs que l'element no hi és. En canvi, la funció p retornarà 1 si totes les posicions indicades per les funcions hash són 1. En aquest cas, podria ser que les posicions estessin a 1 perquè s'han modificat a l'afegir l'element, però també podria ser que s'haguessin marcat a 1 afegint d'altres elements, generant aleshores un fals positiu.

Exemple 5.22 Exemple de filtre de Bloom

En aquest exemple tenim un filtre de Bloom f que consta d'un vector binari de mida $n = 8$ bits i $k = 3$ funcions hash.

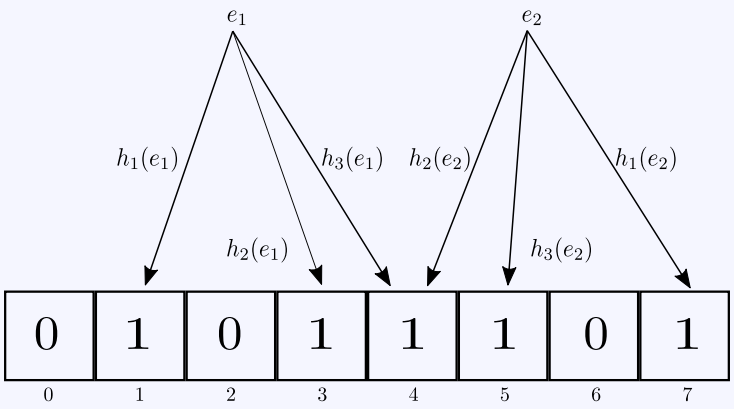
Inicialment, el filtre es troba buit i, per tant, totes les posicions del vector es troben a 0:



A continuació s'afegeixen dos elements, e_1 i e_2, al filtre. Per fer-ho, s'apliquen les tres funcions hash a cadascun dels elements, i s'estableixen a 1 els bits indicats. Suposem que els resultats de les funcions hash són els següents:

$$h_1(e_1) = 1 \qquad h_1(e_2) = 7$$
$$h_2(e_1) = 3 \qquad h_2(e_2) = 4$$
$$h_3(e_1) = 4 \qquad h_3(e_2) = 5$$

La següent figura mostra el filtre de Bloom després d'afegir els elements e_1 i e_2.



A partir del vector binari i les tres funcions hash, es poden fer testos de pertinença sobre el filtre. Així, per exemple, per a comprovar si l'element e_1 pertany al filtre, calcularíem:

$$p(e_1, f) = \prod_{i=1}^{k} V[h_i(e_1)] = V[1] \times V[3] \times V[4] = 1 \times 1 \times 1 = 1$$

i diríem, per tant, que l'element e_1 es troba en el filtre.

Suposem ara que disposem de dos elements addicionals, e_3 i e_4 per als quals també volem comprovar si es troben al filtre, i que els resultats d'aplicar les funcions hash a aquests elements són els següents:

$$h_1(e_3) = 1 \qquad h_1(e_4) = 7$$
$$h_2(e_3) = 0 \qquad h_2(e_4) = 3$$
$$h_3(e_3) = 7 \qquad h_3(e_4) = 1$$

Calculem doncs si els elements es troben al filtre:

$$p(e_3, f) = \prod_{i=1}^{k} V[h_i(e_3)] = V[1] \times V[0] \times V[7] = 1 \times 0 \times 1 = 0$$

$$p(e_4, f) = \prod_{i=1}^{k} V[h_i(e_4)] = V[7] \times V[3] \times V[1] = 1 \times 1 \times 1 = 1$$

Per a e_3 obtenim una resposta correcta, indicant que l'element no es troba al filtre quan, efectivament, no hi és. En canvi, per a e_4 obtenim una resposta errònia: el filtre ens indica que l'element hi és quan, en realitat, aquest no ha estat afegit. El filtre genera un fals positiu per a l'element e_4, produït per les col·lisions que es generen en afegir els elements e_1 i e_2.

Exercici 5.9 Sigui f un filtre de Bloom amb el vector binari següent:

1	1	1	1	0	1	1	1	1	1	1	1	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

i les 5 funcions h_i definides de la manera següent:

$$\begin{aligned} h_1(e) &= e \mod 16 \\ h_2(e) &= e+1 \mod 16 \\ h_3(e) &= e+2 \mod 16 \\ h_4(e) &= e+3 \mod 16 \\ h_5(e) &= e+4 \mod 16 \end{aligned}$$

on els elements e a afegir al filtre sempre són enters.

1. Digueu si els elements següents es troben al filtre:
   $e_1 = 0$, $e_2 = 1429$, $e_3 = 117$ i $e_4 = 15839$.
2. Justifiqueu si l'elecció de les funcions h_i és adient per al seu ús en filtres de Bloom.

Exercici 5.10 Siguin f_1 i f_2 dos filtres de Bloom de la mateixa mida n i que fan servir les mateixes k funcions h_i. Expliqueu com construiríeu un únic filtre que contingui tots els elements que hi ha en els dos filtres.

Pel que fa a l'eficiència de l'estructura de dades, els filtres de Bloom permeten tant afegir elements com consultar si hi pertanyen amb complexitat temporal $\mathcal{O}(k)$, ja que les dues operacions impliquen calcular k hashos. És a dir, afegir elements i comprovar si hi són no depèn de la mida del filtre ni del nombre d'elements que hi pertanyen! Aquesta característica dels filtres de Bloom els fa adequats per a tractar certs tipus de problemes com els que presentem a continuació.

Exemple 5.23 L'ús de filtres de Bloom en aplicacions reals

Un dels usos més habituals dels filtres de Bloom és com a part d'un sistema de *cache*. Per exemple, a l'estudiar el problema del disseny de sistemes de *cache* per a pàgines web, es va observar que la gran majoria de pàgines només són descarregades una única vegada, mentre que un conjunt petit de pàgines es descarreguen molt sovint. A partir d'aquesta observació, els proveïdors intenten crear sistemes de *cache* que incloguin aquest conjunt petit de pàgines que es fan servir sovint, ja que això permet optimitzar la descàrrega sense consumir innecessàriament recursos de *cache* per a pàgines que no tornaran a descarregar-se més.

En aquest cas, es pot servir un filtre de Bloom per emmagatzemar les pàgines que han estat visitades alguna vegada. Quan un client fa una petició d'una pàgina, es consulta el filtre per saber si aquesta pàgina ja ha estat buscada anteriorment.

- Si la pàgina no es troba al filtre, vol dir que no ha estat buscada en el passat. Aleshores, s'afegeix la pàgina al filtre i es recupera de l'emmagatzemament principal. Com que la pàgina només ha estat buscada una vegada, aquesta no s'afegeix a la *cache*, ja que potencialment no és d'interés per a altres usuaris.
- Si la pàgina es troba ja al filtre, vol dir que aquesta ja havia estat consultada en el passat. Aleshores, s'intenta recuperar la pàgina de la *cache*. Si hi és, se serveix al client aquesta versió, guanyant velocitat de descàrrega. En canvi, si la pàgina no es troba a la *cache*, voldrà dir que és el segon cop que es busca, i aleshores s'afegirà a la *cache*.

D'aquesta manera, la cache contindrà totes les pàgines que s'han buscat com a mínim dues vegades.

Facebook i Akamai fan servir aquest tipus d'estratègies en les seves plataformes.

Un filtre de Bloom és una tècnica eficient per a un sistema de *cache* com el que acabem de presentar: el

nombre de possibles pàgines a descarregar és immens (de manera que mantenir una llista completa amb el nombre de descàrregues de cada pàgina seria costós) i un fals positiu no provoca un error en el sistema (sinó que simplement implica afegir una pàgina addicional a la *cache*).

Probabilitat de generar falsos positius

El disseny del filtre de Bloom presenta un compromís entre l'espai que es vol destinar al filtre i la probabilitat de generar falsos positius que es vol acceptar.

La probabilitat de generar un fals positiu en un filtre de Bloom (FPP o *False Positive Probability*) ve determinada per la mida del vector binari (n), el nombre funcions hash (k) i el nombre d'elements que conté (m):

$$FPP(n,k,m) = \left(1 - \left(1 - \frac{1}{n}\right)^{km}\right)^k$$

Vegem pas per pas d'on sorgeix aquesta expressió. La probabilitat que un bit específic del vector segueixi a 0 després d'haver afegit un element al filtre és $(1-1/n)^k$, ja que amb probabilitat $1/n$ el bit es fixarà a 1 per cadascuna de les k funcions hash. Després d'haver afegit els m elements, la probabilitat que un bit segueixi a 0 és doncs $(1-1/n)^{km}$ (repetim m vegades el procés d'afegir un element). Finalment, la probabilitat de generar un fals positiu és la probabilitat que les k posicions consultades per a l'element siguin 1.

Així doncs, donat un filtre d'una mida i nombre de funcions hash determinats, la FPP augmenta conforme es van afegint elements al filtre. La Figura 5.9 mostra com varia la probabilitat d'un fals positiu per a un filtre de 64 bits que fes servir dues, tres o quatre funcions hash.

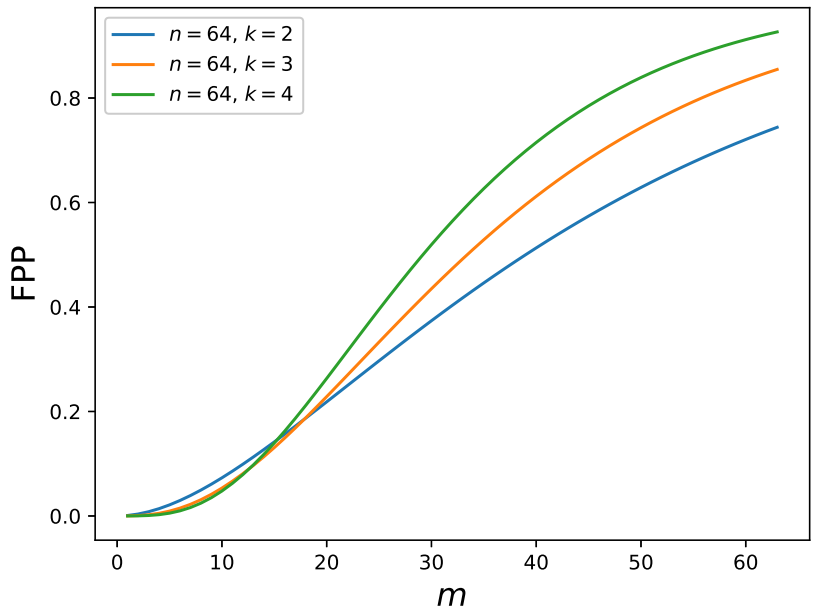


Figura 5.9: Probabilitat de fals positiu (FPP) segons el nombre d'elements del filtre (m), per a $k = 2$ (blau), $k = 3$ (taronja) i $k = 4$ (verd) funcions hash.

Donat un filtre de mida n amb m elements, ens podem preguntar quin és el nombre de funcions hash k òptim per tal de minimitzar la FPP del filtre. La resposta no és immediata, doncs d'una banda, augmentar k permet

comprovar més bits per cada element que es vulgui testejar, minimitzant així la FPP però, d'altra banda, disminuir k permet augmentar la probabilitat de trobar un bit a 0, que és el que ens permet evitar un fals positiu.

El valor de k òptim per minimitzar la FPP ve donat per l'expressió següent:

$$k_{opt} \approx \frac{n}{m} \ln 2$$

Nombre òptim de funcions hash

El lector interessat pot consultar el capítol 5 del llibre Mitzenmacher, Michael, and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis.* Cambridge university press, 2017, per aprendre com deduir l'expressió que permet calcular el nombre òptim de funcions hash a partir de l'expressió de la FPP.

Així, el nombre òptim de funcions hash ve determinat pel factor n/m, que representa el nombre de bits per element emmagatzemat al filtre. La Figura 5.10 mostra l'evolució de la FPP en base al nombre de funcions hash que s'utilitzen per a filtres amb diferents bits per element (n/m). També s'hi mostra el nombre òptim de funcions hash a fer servir en cadascun dels casos.

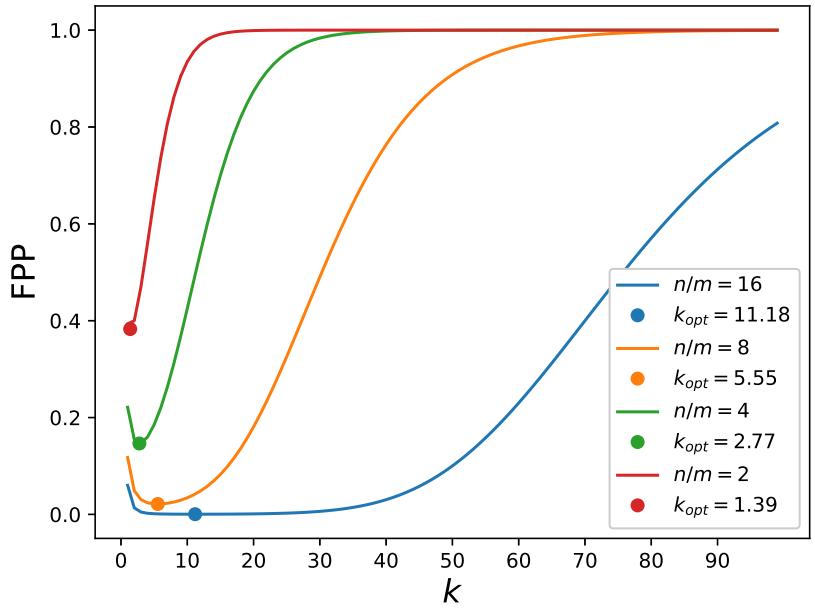


Figura 5.10: Probabilitat de fals positiu (FPP) segons el nombre de funcions hash (k).

Cal tenir en compte que l'expressió que permet calcular k_{opt} pot retornar un nombre real, però el nombre de funcions hash d'un filtre sempre serà un enter, que caldrà triar en el moment del disseny.

Les funcions hash dels filtres de Bloom

Els filtres de Bloom fan ús de diverses funcions hash, que han de ser independents entre elles i han de tenir una distribució de sortida uniforme.

Es fan servir diferents tècniques per tal de poder implementar aquestes funcions sense fer ús de funcions hash diferents, cosa que sovint seria molt costosa. Així, per exemple, es poden agafar diferents parts de la

sortida d'una mateixa funció hash per a cadascuna de les h_i; es pot concatenar un valor inicial, diferent per a cada h_i, a l'entrada de la funció hash; o bé es poden combinar les sortides de dues funcions hash per a recrear-ne les k necessàries.

Exemple 5.24 Exemples de definicions per a les h_i

Suposem que construim un filtre per a una *cache* d'$n = 256$ bits i $k = 4$ funcions hash, i que hi volem afegir la pàgina `uoc.edu`. Per a indexar les 256 posicions del filtre necessitem 8 bits ($2^8 = 256$), de manera que cadascuna de les h_i haurà de tenir una sortida de 8 bits.

Particionar la sortida d'una funció hash

Una manera d'implementar les quatre h_i que necessitem és fer servir una única funció hash que tingui una sortida de 32 bits com a mínim, i prendre blocs de 8 bits d'aquesta sortida per a cadascuna de les k funcions hash.

Per exemple, prenem el SHA-1, que té una sortida de 160 bits, com a funció hash base, i calculem els h_i de la manera següent:

$$\begin{aligned} SHA1(\texttt{uoc.edu}) &= \texttt{0xe6a62a58a28f94d745d3ea9a47163c846a065a3c} \\ h_1(\texttt{uoc.edu}) &= SHA1(\texttt{uoc.edu})_{0..7} = \texttt{0xe6} = 230 \\ h_2(\texttt{uoc.edu}) &= SHA1(\texttt{uoc.edu})_{8..15} = \texttt{0xa6} = 166 \\ h_3(\texttt{uoc.edu}) &= SHA1(\texttt{uoc.edu})_{16..23} = \texttt{0x2a} = 42 \\ h_4(\texttt{uoc.edu}) &= SHA1(\texttt{uoc.edu})_{24..31} = \texttt{0x58} = 88 \end{aligned}$$

on l'expressió $X_{i..j}$ denota els bits des de la posició i a la posició j del valor X.

L'avantatge d'aquest mètode és que només requereix el càlcul d'una única funció hash. En canvi, però, el nombre de bits que s'obtenen queda limitat per la mida de la sortida de la funció hash, de manera que no serveix per a filtres molt grans o que utilitzin moltes funcions hash.

Exercici 5.11 Sigui f un filtre de Bloom amb un vector $n = 65536$ posicions i $k = 10$. Justifiqueu quina de les següents tres funcions hash seria més adient per a utilitzar per definir les 10 h_i amb la tècnica de particionar la sortida i proposeu una possible definició de les h_i.

1. MD5
2. SHA1
3. SHA256

Ús d'una llavor

Una alternativa és fer servir una llavor concatenada amb l'element com a entrada d'una única funció hash, i variar el valor de la llavor per a cada h_i. Per exemple, si prenem de nou el SHA-1 com a funció hash base i un comptador com a llavor, procediríem a calcular:

$$\begin{aligned} h_1(\texttt{uoc.edu}) &= SHA1(\texttt{1uoc.edu})_{0..7} = \texttt{0x81} = 129 \\ h_2(\texttt{uoc.edu}) &= SHA1(\texttt{2uoc.edu})_{0..7} = \texttt{0x87} = 135 \\ h_3(\texttt{uoc.edu}) &= SHA1(\texttt{3uoc.edu})_{0..7} = \texttt{0xe9} = 233 \\ h_4(\texttt{uoc.edu}) &= SHA1(\texttt{4uoc.edu})_{0..7} = \texttt{0x16} = 22 \end{aligned}$$

En aquest cas, com que només necessitem 8 bits per cada h_i, hem conservat els primers 8 bits de la sortida i hem descartat la resta de bits.

L'avantatge d'aquest mètode respecte a l'anterior és que no té límit en relació al nombre de funcions hash k a implementar ni el nombre de bits de sortida de cada h_i individual (si se'n necessiten més dels que ofereix la sortida de la funció hash base, es poden fer servir diverses llavors per a cada h_i). No obstant això, aquest mètode és molt més costós computacionalment que el mètode de particionar la sortida, ja que per cada element a afegir o comprovar caldrà calcular diversos hashos.

Alternatives eficients per al càlcul de les funcions hash

Per a conèixer una alternativa eficient per a derivar les k funcions hash recomanem la lectura de l'article Kirsch, Adam, and Michael Mitzenmacher. *Less hashing, same performance: building a better bloom filter."* European Symposium on Algorithms. Springer, Berlin, Heidelberg, 2006.

Més enllà de com aconseguir k funcions hash per a construir filtres de Bloom, ens podem preguntar també quin tipus de funció hash és adient com a funció hash base en aquestes construccions. Hem esmentat que és necessari que les diferents h_i siguin independents entre elles, i també que generin una sortida uniforme. Les funcions hash criptogràfiques poden complir aquests requisits. Ara bé, fer servir com a funció base una funció hash criptogràfica (com ara el SHA1) és costós computacionalment. Serien útils, per a aquesta aplicació, l'ús de funcions hash no criptogràfiques, que tot i que no compleixen certs requisits de seguretat, són molt més ràpides de calcular? La resposta no és absoluta, i dependrà de l'entorn en el qual preveiem desplegar el filtre de Bloom. Si l'entorn no té adversaris, potser podem fer servir funcions no criptogràfiques, sempre que es repecti la uniformitat de les sortides i la independència entre les h_i que en derivem. Alguns exemples de funcions hash no criptogràfiques que es fan servir en implementacions de filtres de Bloom són la funció hash Murmur3 o la funció Fowler-Noll-Vo (FNV). En canvi, si l'entorn en el qual despleguem el filtre pot tenir adversaris, que tinguin un interés en fer fallar els testos de pertinença, aleshores en general serà preferible l'ús de funcions criptogràfiques, ja que les seves propietats faran el filtre més robust a atacs. En qualsevol cas, cal estudiar amb detall l'escenari i els possibles adversaris que s'hi poden trobar, per tal de decidir quin tipus de funció hash cal implementar.

Murmur3

La funció hash no criptogràfica Murmur3 deu el seu nom a les operacions en què basa el seu funcionament: **mu**ltiplicar-**r**odar-**mu**ltiplicar-**r**odar. La primera versió d'aquesta funció hash (coneguda com a Murmur1) va fer-se pública el 2008, i la versió actual té dues variants: Murmur3A, que genera una sortida de 32 bits, i Murmur3F, que té una sortida de 128 bits.

Fowler-Noll-Vo (FNV)

La funció hash no criptogràfica Fowler-Noll-Vo deu el seu nom als autors que la van dissenyar. La primera versió es va començar a gestar al 1991. La versió actual d'aquesta funció ofereix variants amb sortides de 32, 64, 128, 256, 512 i 1024 bits.

Variants de filtres de Bloom

Els filtres de Bloom que acabem de descriure corresponen a la variant bàsica d'aquesta estructura de dades. Ara bé, existeixen una gran diversitat de variants dels filtres de Bloom, cadascuna de les quals aporta alguna nova característica en relació a la versió bàsica.

Així, per exemple, la variant bàsica del filtre de Bloom no permet eliminar elements del filtre. Una vegada s'ha afegit un element ja no es pot esborrar, ja que si fixéssim a 0 totes les posicions indicades per les funcions hash per a aquell element, podríem estar afectant altres elements que també haguessin modificat aquelles posicions. Els filtres de Bloom amb comptadors (en anglès, es coneixen com a *Counting Bloom filters*) són una variant que permet eliminar elements.

Els **filtres de Bloom amb comptadors** canvien el vector binari per un vector d'enters, que s'inicialitza també a 0. Per a afegir un element, s'incrementa el comptador de les posicions indicades per les funcions hash. D'aquesta manera, es pot definir una operació d'esborrat, que consisteix simplement en decrementar el comptador de les posicions afectades.

Exercici 5.12 Sigui f un filtre de Bloom amb comptadors d'$n = 16$ posicions i $k = 3$ funcions hash, on $h_i = SHA1_{4i..4i+3}$.

1. Mostreu el contingut del filtre després d'afegir els tres elements següents: `uoc.edu`, `cv.uoc.edu`, `biblioteca.uoc.edu`.

2. Elimineu l'element `uoc.edu` del filtre i mostreu com queda el vector després d'aquesta operació.

Un altre dels problemes que presenta la variant bàsica en el seu ús en aplicacions reals és que cal decidir la mida del filtre abans de començar a treballar-hi, en base al nombre d'elements que s'hi preveuen emmagatzemar i la probabilitat de falsos positius que l'aplicació pot tolerar. Ara bé, estimar el nombre d'elements que s'hi emmagatzemaran abans de desplegar l'aplicació pot no ser fàcil i les conseqüències d'una mala estimació afectaran al rendiment. D'una banda, si l'estimació és superior als elements que realment s'hi emmagatzemen, estarem desaprofitant espai de disc. D'altra banda, si l'estimació és inferior, la probabilitat de falsos positius augmentarà per sobre del llindar que l'aplicació pot tolerar. Els filtres de Bloom escalables permeten afrontar aquest problema, oferint la possibilitat d'augmentar la mida dels filtres a mesura que aquests es van omplint.

Els **filtres de Bloom escalables** (SBF) estan formats per un o més filtres de Bloom bàsics. Quan els filtres existents en un moment donat s'omplen, aleshores s'afegeix un nou filtre bàsic a l'SBF. Cada nou filtre es dissenya de manera que la probabilitat de fals positiu en l'estructura completa (l'SBF) sigui l'especificada en el moment del disseny. D'aquesta manera, es pot desplegar una aplicació amb un filtre de Bloom petit, i anar-lo ampliant conforme creixen les necessitats de l'aplicació sens que augmenti la probabilitat de falsos positius.

5.5 Funcions hash amb propietats addicionals

Algunes de les aplicacions que acabem de presentar es poden beneficiar de l'ús de funcions hash amb algunes propietats addicionals, més enllà de les necessàries per a funcions hash criptogràfiques que s'han presentat a l'inici del capítol. Una d'aquestes propietats és que siguin computacionalment costoses de calcular i/o difícilment optimitzables en hardware específic. Aquesta propietat pot ser d'interès en les funcions hash utilitzades per emmagatzemar contrasenyes, per derivar claus o en proves de treball.

En el cas de les contrasenyes i la derivació de claus, augmentar el temps de còmput de la funció hash té poc impacte en l'ús legítim de les aplicacions, doncs l'usuari legítim que s'ha d'autenticar només necessita calcular un únic resultat. En canvi, aquest augment dificulta els atacs contra aquests sistemes, ja que els atacants necessiten calcular moltes vegades la funció hash (per exemple, per fer atacs de diccionari o de força bruta).

En el cas de les proves de treball, la situació és similar en el seu ús com a protecció per a correu brossa. El cas de les criptomonedes és una mica diferent, i té a veure amb la descentralització del minat: certs tipus de funcions hash són fàcilment implementables en dispositius hardware específics (ASICs o FPGAs) per al càlcul de la funció hash, però la creació i adquisició d'aquests dispositius no es troba a l'abast de tothom. Això fa que hi hagi un interès en evitar utilitzar funcions hash optimitzables per hardware, ja que aquestes porten a dificultar l'accés al minat per al públic en general.

ASICs

Un ASIC (de l'anglès, *application-specific integrated circuit*) és un circuit integrat d'aplicació específica, és a dir, un circuit dissenyat i fabricat per a dur a terme una funció específica. Això es contraposa amb circuits d'ús genèric, com ara les CPUs, que estan pensades per a poder executar diverses aplicacions diferents. Els ASICs es dissenyen fent servir llenguatges de descripció de harware, que després se sintetizen per produir una descripció a nivell de portes lògiques de l'aplicació. El disseny i creació d'un ASIC té uns costos fixos molt elevats, però en canvi són circuits molt eficients (en quant a velocitat i consum energètic) per a fer la tasca per la qual estan dissenyats. A més, els costos variables són petits, de manera que són adients per a fer grans produccions.

FPGAs

Una FPGA (de l'anglès, *Field-programmable gate array*) és un circuit dissenyat per a ser configurat després de la seva producció. Les FPGAs contenen arrays de blocs lògics, que poden implementar portes lògiques o altres funcions més compexes, i permeten configurar les interconnexions entre aquests blocs. Com els ASICs, es configuren fent servir també llenguatges de descripció de hardware. Les FPGAs ofereixen un rendiment menor que els ASICs, però el cost d'una implementació és molt més barat que el d'un ASIC, ja que són configurables després d'haver sortit de la fàbrica de producció.

Així, hi ha un interès creixent en el disseny de funcions hash resistents a ASICs, és a dir, funcions hash que no donin un gran avanatge al ser implementades en ASICs. Una de les tècniques que s'utilitza és la creació de funcions amb un ús intensiu de memòria (en anglès, es coneixen com a *memory-hard functions*). Com que la memòria té un cost similar, tant si es fa servir en un ASIC com des d'un dispositiu de propòsit general, les funcions que requereixen d'un ús intensiu de la memòria no es beneficien molt de la seva implementació en ASICs.

Una funció hash amb ús intensiu de memòria (en anglès, en diem ***memory-hard hash function***) és una funció hash que requereix d'un ús intensiu de memòria per a avaluar-la de manera ràpida.

Noteu que l'ús intensiu de memòria no és un requisit indispensable per a poder avaluar la funció hash: la memòria és necessària per a avaluar-la de manera ràpida. Si no es disposa de la memòria, aleshores la funció es pot avaluar però aquesta avaluació és molt més lenta.

La funció hash **scrypt** és una funció amb ús intensiu de memòria feta presentada l'any 2009 i publicada com a RFC el 2016 (RFC 7914). Per aconseguir l'ús intensiu de memòria, la funció fa ús d'un vector pseudoaleatori molt gran. Durant l'execució de la funció, cal accedir a diversos elements d'aquest vector, en un ordre també pseudoaleatori, i recuperant una mateixa posició diverses vegades. La generació d'aquest vector és un procés computacionalment costós. Així, una implementació que emmagatzemi el vector sencer serà ràpida a executar-se, ja que només calcularà el vector una única vegada i recuperarà els elements que vagi necessitant del vector durant l'execució. En canvi, una implementació que no desi aquest vector, necessitarà calcular els elements cada vegada que els necessiti, un procés costós per disseny.

5.6 Resum

Com hem pogut veure en aquest capítol, les funcions hash són una eina criptogràfica extremadament versàtil que s'utilitza cada vegada més en diferents aplicacions i protocols criptogràfics. La seva principal característica és la impossibilitat de predir-ne la sortida, malgrat conèixer-ne l'entrada coneguda tot i assumint que la definició de la funció hash queda totalment determinada de forma pública. A més, aquesta predicció no es pot realitzar tot i conèixer la sortida d'altres valors propers a l'entrada, ja que les propietats d'aquestes funcions impliquen que un petit canvi en l'entrada provoqui un canvi significativament gran en la sortida.

Per aconseguir aquestes característiques, hem vist que les funcions hash estan formades per un seguit de subfuncions que incorporen un alt grau de no-linealitat justament per tal de fer imprevisible la seva sortida. A més, les operacions internes d'una funció hash s'iteren varies vegades perquè encara sigui més complicat realitzar-ne una anàlisi. Per aquest motiu, la simple definició d'una funció hash, com ara el SHA-256, ja implica una complexitat força elevada.

Gràcies a aquestes propietats, les funcions hash es poden utilitzar per realitzar autenticació de missatges amb criptografia de clau simètrica, per obtenir resums quasi unívocs de missatges, per a generar valors pseudoaleatoris, en protocols de compromís, per a l'emmagatzemament de contrasenyes o bé per a la derivació de claus. A més, les funcions hash són també un dels pilars de les noves criptomonedes, gràcies a la seva utilització en les proves de treball, els arbres de Merkle o els filtres de Bloom.

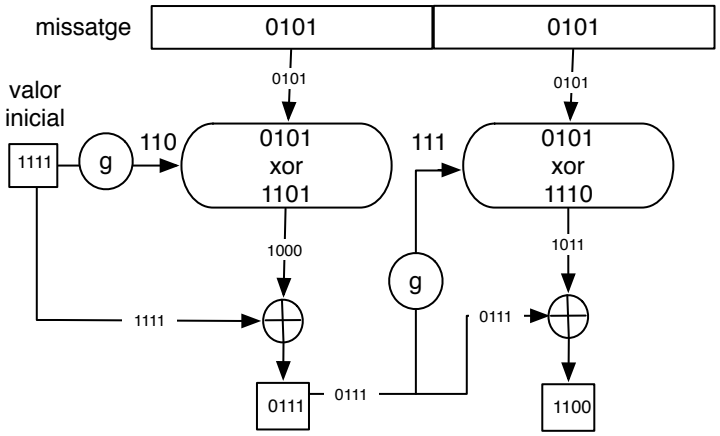
5.7 Solucions dels exercicis

Exercici 5.1:

Utilitzant les expressions de l'Apartat 1.2, la probabilitat que en un grup de 50 persones triades a l'atzar, dues d'elles tinguin l'aniversari el mateix dia és $1-[(1-\frac{1}{365})\cdot(1-\frac{2}{365})\cdots(1-\frac{49}{365})]=0,97$. D'altra banda, la probabilitat que almenys una d'elles hagi nascut el dia 1 de gener és de $1-(1-\frac{1}{365}^{49})=0,12$

Exercici 5.2:

El resultat de la funció hash és el valor 1100 i el procés de càlcul es mostra en el següent gràfic:



Exercici 5.3:

En primer lloc, cal passar el valor 'SALA' a una cadena de bits, obtenint:

01010011010000010100110001000001.

A continuació afegim un '1' a la cadena, obtenint 01010011010000010100110001000011.

Després, hi afegim $448-(32+1)=415$ zeros i els últims 64 bits són la representació en binari de la mida del missatge 'SALA' en bits. Com que la mida del missatge era de 32 bits, tenim que els 64 bits finals del missatge són: $000\cdots00100000$. Per tant els 512 bits de padding són:

$$01010011010000010100110001000001100\underbrace{00\cdots0}_{415\text{ zeros}}\underbrace{00\cdots00100000}_{64\text{ bits}}$$

Exercici 5.4:

Els resultats de les funcions són els següents:

$ROTR^7(m_1)=11111111000000000000000111111111$
$SHR^{10}(m_1)=00000000000000000000000000111111$
$\sigma_0(m_1)=11000001111111111101111000000000$
$\sigma_1(m_1)=01100000000000000110000000111111$
$\sum_0(m_1)=00111100000001111100001111111000$
$\sum_1(m_1)=00000011100111111111110001100000$
$\mathrm{Ch}(m_1,m_2,m_3)=11111111000000001111111111110000$
$\mathrm{Maj}(m_1,m_2,m_3)=11110000000000001111111111110000$

Exercici 5.5:

La utilització d'una HMAC amb *secret prefix* no és segura perquè la informació secreta s'afegeix a l'inici i a continuació es calcula el hash. Això fa que en un disseny estàndard de funció hash, afegir un bloc al final

d'un missatge únicament requereixi continuar iterant la funció hash amb el nou bloc sense necessitat de conèixer el valor de la clau secreta utilitzada per calcular l'HMAC. Així, en aquest exercici, com que volem calcular l'HMAC de la cadena 01111111 però coneixem l'HMAC de la cadena més curta 0111, només caldrà utilitzar la funció hash una iteració més amb el valor 0111 com a vector inicial i el fragment nou, 1111, com a missatge a calcular el hash. Si realitzem els càlculs trobarem que l'HMAC per al missatge 01111111 serà 0001.

Exercici 5.6:

Una funció hash de 64 bits té un nombre total de possibles imatges de 2^{64}. En 10 minuts, si podem provar 100.000 hashos per segon, haurem provat $100.000 \cdot 60 \cdot 10 = 60.000.000$ valors. Per tant, el valor objectiu per a la nostra prova de treball serà $\frac{2^{64}}{60.000.000} \approx 307445734561$ que si ho expressem com una cadena de 64 bits tenim

0000000000000000000000000100011110010101001100011001110010100001

Exercici 5.7:

En primer lloc es calculen els hashos de cadascun dels blocs de dades:

$$h_{000} = H(L_1) = \texttt{ea64f9d949b9508d847c85de6a03a0db71258ee7d7b01d135c5b7c794bbb9848}$$
$$h_{001} = H(L_2) = \texttt{9689fc250b6c60ec0c5c6b6f9bc7e621c69df7febb4ad1e641cc4e7792b26146}$$
$$h_{010} = H(L_3) = \texttt{876cb92390cf4b52cb3e58a48b7f221eccd99f01c1925eefc9aa1da5d4c88901}$$
$$h_{011} = H(L_4) = \texttt{bb063417e4eddab0426c3cdfef9da597af388a60690ceaa01fb15f49ce30c1e4}$$
$$h_{100} = H(L_5) = \texttt{ce9b1b67eb23b708147ac7a35c5a77ecdedf9ac991e0f084d1248521918795ab}$$
$$h_{101} = H(L_6) = \texttt{94335c418b09a0223a6f90757209ca69f96a26f5705d6c14d132b33cdea84c4e}$$
$$h_{110} = H(L_7) = \texttt{032cf476fae7facbb766e98e7dac817e75cd29434b15108e2b10b3d9b3a68967}$$
$$h_{111} = H(L_8) = \texttt{d7409ed2339ca2803e527d63ec1e4b08ae3705ebf97b8de4646911f91ee1294c}$$

Després, es procedeix a calcular els hashos del tercer nivell de l'arbre:

$$h_{00} = H(h_{000}||h_{001}) = H(\texttt{ea64f9d9}\cdots\texttt{9689fc25}\cdots) =$$
$$= \texttt{f4aceecdaec0b6e7702331a7d5c4d2dfd708dcda5becc9e6ae2ebcb55b891771}$$
$$h_{01} = H(h_{010}||h_{011}) = \texttt{344bdb1d3370f4e61c676421c048a4c9284ea5538dd678b4d2bb7ee9756c5337}$$
$$h_{10} = H(h_{100}||h_{101}) = \texttt{3dbbcce400af55fc8acdca4013550a0d76169ce584a82b802ddf2b4a2761c897}$$
$$h_{11} = H(h_{110}||h_{111}) = \texttt{2988ac56ddbfed1dfae8966ef1dfa1d8fe31bbd52f0002afd2d4cbbe1148f8e8}$$

A continuació es calculen els hashos del segon nivell:

$$h_0 = H(h_{00}||h_{01}) = H(\texttt{f4aceecd}\cdots\texttt{344bdb1d}\cdots) =$$
$$= \texttt{04d15a9750de7caa93411a12a1d53a9672cf1fd5213e31d4241650f4e7f34a59}$$
$$h_1 = H(h_{10}||h_{11}) = \texttt{e2f528e5024516fdaaeaeb2b92fcde9b8228908f780141e936a89b6cd6dc6c0d}$$

I finalment es calcula l'arrel de l'arbre:

$$h_r = H(h_0||h_1) = H(\texttt{04d15a97}\cdots\texttt{e2f528e5}\cdots) =$$
$$= \texttt{3cb579c97652a53b9997027665390bf927943a1cc09b69e5a5c2518862811d37}$$

Exercici 5.8:

La prova de pertinença per al bloc de dades `MILFORD SOUND` és:

$$\begin{aligned}\Pi &= (\texttt{MILFORD SOUND}, h_{110}, h_{10}, h_0) = \\ &= (\texttt{MILFORD SOUND}, \texttt{032cf476}\cdots, \texttt{3dbbcce4}\cdots, \texttt{04d15a97}\cdots)\end{aligned}$$

El verificador, que coneix $h_r = \texttt{3cb579c9}\cdots$ procedirà a validar la prova calculant:

$$\begin{aligned}h_{111} &= H(\texttt{MILFORD SOUND}) = \texttt{d7409ed2}\cdots \\ h_{11} &= H(h_{110}||h_{111}) = H(\texttt{032cf476}\cdots\texttt{d7409ed2}\cdots) = \texttt{2988ac56}\cdots \\ h_1 &= H(h_{10}||h_{11}) = H(\texttt{3dbbcce4}\cdots\texttt{2988ac56}\cdots) = \texttt{e2f528e5}\cdots \\ h'_r &= H(h_0||h_1) = H(\texttt{04d15a97}\cdots\texttt{e2f528e5}\cdots) = \texttt{3cb579c9}\cdots\end{aligned}$$

Com que h'_r és efectivament igual a h_r, el verificador donarà la prova per vàlida.

Exercici 5.9:

1. L'element $e_1 = 0$ no es troba al filtre, ja que el bit $h_5(0) = 4$ és 0. En canvi, els elements e_2, e_3 i e_4 sí que es troben al filtre, ja que totes les posicions indicades pels h_i són 1 al vector:

$$\begin{aligned}&h_1(e_2) = h_1(1429) = 5;\ h_2(e_2) = h_2(1429) = 6;\ h_3(e_2) = h_3(1429) = 7; \\ &\quad h_4(e_2) = h_4(1429) = 8;\ h_5(e_2) = h_5(1429) = 9; \\ &h_1(e_3) = h_1(117) = 5;\ h_2(e_3) = h_2(117) = 6;\ h_3(e_3) = h_3(117) = 7; \\ &\quad h_4(e_3) = h_4(117) = 8;\ h_5(e_3) = h_5(117) = 9; \\ &h_1(e_4) = h_1(15839) = 15;\ h_2(e_4) = h_2(15839) = 0;\ h_3(e_4) = h_3(15839) = 1; \\ &\quad h_4(e_4) = h_4(15839) = 2;\ h_5(e_4) = h_5(15839) = 3;\end{aligned}$$

2. La tria de les funcions h_i és nesfasta ja que les funcions h_i no només no són independents entre elles, sinó que el resultat de qualsevol d'elles determina de forma única el resultat de la resta. Això fa que l'ús de diverses funcions h_i sigui contraproduent i augmenta els errors.

Exercici 5.10: Podem crear un filtre f_3 de la mateixa mida que f_1 i f_2 que contingui la unió dels elements que hi ha a f_1 i f_2 fent una OR lògica de cadascuna de les posicions dels dos filtres f_1 i f_2. Així, a la posició i del filtre f_3 hi posaríem el resultat d'una OR entre el valor de la posició i del filtre f_1 i el valor de la posició i del filtre f_2, per a $i = 1 \ldots n$.

Així, tot element que es troba a f_1 o a f_2 es trobaria també al filtre f_3, ja que les posicions que aquest element ha fixat a 1 seguirien sent 1 al nou filtre. No obtant això, la probabilitat de fals positiu seria superior a f_3 (ja que hi hauria més elements per a un filtre amb la mateixa mida i mateix nombre de funcions hash).

Exercici 5.11:

Per a indexar les $n = 65536$ posicions es necessiten 16 bits ($2^{16} = 65536$). Com que $k = 10$, la sortida de la funció hash haurà de tenir $10 \cdot 16 = 160$ bits com a mínim. Això descarta l'ús d'MD5, que té una sortida de 128 bits. Per tant, pel que fa a la mida de la sortida, tant SHA1 (160 bits) com SHA256 (256 bits) serien bones candidates.

Tot i que els resultats específics de velocitat de càlcul de SHA1 i SHA256 depenen del maquinari que es faci servir, en general SHA1 és més ràpid. Per tant, si volem prioritzar la velocitat d'afegir i consultar elements, preferirem utilitzar SHA1.

L'eina openssl permet executar testos de rendiment de les primitives criptogràfiques que implementa. Per comparar les tres funcions hash, podem executar:

```
openssl speed md5 sha1 sha256
```

El resultat d'executar la instrucció anterior en un Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz és:

```
The numbers are in 1000s of bytes per second processed.
type     16 bytes     64 bytes     256 bytes    1024 bytes   8192 bytes
md5      112326.65k   262123.46k   462351.10k   571945.30k   613750.10k
sha1     123068.22k   284170.87k   593228.37k   787015.58k   865129.81k
sha256   70635.03k    158946.75k   292461.14k   359245.23k   389903.70k
```

En aquest cas, SHA1 és prop del doble de ràpid que SHA256 a l'hora de calcular el hash.

Exercici 5.12:

Calculem h_1, h_2 i h_3 per a cada element a afegir al filtre:

$$\begin{aligned}
SHA1(\texttt{uoc.edu}) &= \texttt{0xe6a62a58a28f94d745d3ea9a47163c846a065a3c}\\
h_1(\texttt{uoc.edu}) &= SHA1(\texttt{uoc.edu})_{0..3} = \texttt{0xe} = 15\\
h_2(\texttt{uoc.edu}) &= SHA1(\texttt{uoc.edu})_{4..7} = \texttt{0x6} = 6\\
h_3(\texttt{uoc.edu}) &= SHA1(\texttt{uoc.edu})_{8..11} = \texttt{0xa} = 10
\end{aligned}$$

$$\begin{aligned}
SHA1(\texttt{cv.uoc.edu}) &= \texttt{0x061053c6a11e8cd254a35edca6d8ab0a29765bb2b}\\
h_1(\texttt{cv.uoc.edu}) &= SHA1(\texttt{cv.uoc.edu})_{0..3} = \texttt{0x0} = 0\\
h_2(\texttt{cv.uoc.edu}) &= SHA1(\texttt{cv.uoc.edu})_{4..7} = \texttt{0x6} = 6\\
h_3(\texttt{cv.uoc.edu}) &= SHA1(\texttt{cv.uoc.edu})_{8..11} = \texttt{0x1} = 1
\end{aligned}$$

$$\begin{aligned}
SHA1(\texttt{biblioteca.uoc.edu}) &= \texttt{0x37a00421948415623523179cc7d97877302d98d0}\\
h_1(\texttt{biblioteca.uoc.edu}) &= SHA1(\texttt{biblioteca.uoc.edu})_{0..3} = \texttt{0x3} = 3\\
h_2(\texttt{biblioteca.uoc.edu}) &= SHA1(\texttt{biblioteca.uoc.edu})_{4..7} = \texttt{0x7} = 7\\
h_3(\texttt{biblioteca.uoc.edu}) &= SHA1(\texttt{biblioteca.uoc.edu})_{8..11} = \texttt{0xa} = 10
\end{aligned}$$

Per tant, el contingut del filtre una vegada s'han afegit els elements `uoc.edu`, `cv.uoc.edu` i `biblioteca.uoc.edu` és:

1	1	0	1	0	0	2	1	0	0	2	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

El contingut del filtre després d'eliminar l'element `uoc.edu` és:

1	1	0	1	0	0	1	1	0	0	1	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

5.8 Bibliografia

Christof Paar, Jan Pelzl (2010). *Understanding Cryptography. Capítol 11*. Springer-Verlag Berlin Heidelberg. DOI: 10.1007/978-3-642-04101-3

R. Rivest (1992). "The MD4 Message-Digest Algorithm" *Request for Comments (núm: 1320) RFC 1320*. Internet Engineering Task Force.

R. Rivest (2011). "MD4 to Historic Status" *Request for Comments (núm: 6150) RFC 6150*. Internet Engineering Task Force.

R. Rivest (1992). "The MD5 Message-Digest Algorithm" *Request for Comments (núm: 1321) RFC 1321*. Internet Engineering Task Force.

A. Lenstra, X. Wang, B. Weger (2005). *Colliding X.509 Certificates*. Cryptology ePrint Archive Report 2005/067.

H. Dobbertin, A. Bosselaers, B. Preneel (1996). *RIPEMD-160: A Strengthened Version of RIPEMD*. Proceedings of FSE, LNCS 1039, pp. 71-82, Springer-Verlag

V. Rijmen, P. Barreto (2000). *The WHIRLPOOL Hash Function*.

National Institute of Standard and Technology (NIST) (2015). "Secure Hash Standard (SHS)". *Federal information processing standards (num. 180-4) FIPS 180-4*. Washington: NIST. DOI: 10.6028/NIST.FIPS.180-4

National Institute of Standard and Technology (NIST) (2015). "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions". *Federal information processing standards (num. 202) FIPS 202*. Washington: NIST. DOI: 10.6028/NIST.FIPS.202

R. Rivest (1992). "PKCS #5: Password-Based Cryptography Specification Version 2.0" *Request for Comments (núm: 2898) RFC 2898*. Internet Engineering Task Force.

A. Narayanan et al. (2016). "Bitcoin and cryptocurrency technologies: a comprehensive introduction". *Princeton University Press*

D. Boneh, V. Shoup (2015). "A graduate course in applied cryptography".

E. Barker, L. Feldman, G. Witte (2015). "Recommendation for random number generation using deterministic random bit generators". *National Institute of Standards and Technology*