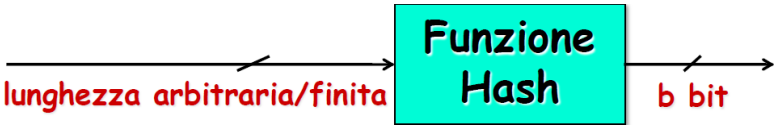


La caratteristica della funzioni Hash è che, preso un messaggio in input che ha una lunghezza arbitraria, la funzione Hash produce una sequenza di b bit fissati, qualunque sia la lunghezza della stringa presa in input. La funzione Hash non prende in input chiavi e valori segreti ed è per questo una funzione deterministica, cioè, se applico per 2 volte lo stesso input, ottengo per 2 volte lo stesso risultato. L'idea di base della funzione Hash è che è come se fosse un'impronta digitale del documento, quindi, preso un valore Hash, questo identifica un particolare documento e no un altro. Teoricamente questo è un problema perché dato che in input si ha una stringa di lunghezza arbitraria, e b bit fissati in output alla funzione Hash, è chiaro che ci sono delle collisioni: cioè ci sono più valori che hanno lo stesso valore Hash .

Un'altra caratteristica importante della funzione Hash è che è velocissima da calcolare, infatti, tra tutte le primitive viste (cifratura, Stream Cipher) è la più veloce a computare.

Ha diverse applicazioni, quali le firme digitali, l'integrità dei dati (per esempio si utilizza l'Hash di un hard disk per poter fare un'analisi dei dati, oppure posso usarlo per poter verificare se qualcuno ha manomesso qualche informazione e il controllo da fare è: se l'Hash attuale è uguale a quello precedente allora non c'è stato alcun attacco. Naturalmente una modifica fatta non si ripristina ma posso solo sapere che c'è stata una manomissione dei dati) e la certificazione nel tempo (stabilisce in che data è stato creato un certo documento D).

Chiaramente delle funzioni Hash non è importante solo la facilità di calcolo, ma interessa anche il livello di sicurezza, ed è il seguente:

Se un attaccante prepara 2 versioni di un contratto M ed M' di cui M è favorevole ad Alice e M' è sfavorevole ad Alice. Dal momento che viene verificato l'Hash del messaggio, l'idea è di modificare i 2 documenti in maniera tale da trovare una modifica di M che è uguale ad una modifica di M' rispetto al valore Hash, ad esempio, se modifica M' a caso, aggiungendo degli spazi, con 32 possibilità riuscirebbe ad ottenere 2^{32} messaggi. Se riesce a trovare messaggi che hanno lo stesso valore Hash, quindi il corpo del messaggio diventa lo stesso, e Alice firma il messaggio M → $Firma_{kpriv}(h(M))$ e l'attaccante ha quindi la firma di M' $Firma_{kpriv}(h(M'))$ quindi, l'attaccante potrebbe affermare che Alice ha firmato un contratto diverso da quello che lei crede realmente di aver firmato. Questo genera un grande problema di sicurezza, l'obiettivo di una funzione Hash è quindi che una situazione del genere non debba accadere.

Le condizioni di sicurezza della funzioni Hash sono:

- **Sicurezza debole**: dato un messaggio M è computazionalmente difficile trovare un altro M' tale che h(M)= h(M').
- **Sicurezza forte**: computazionalmente difficile trovare 2 diversi messaggi con lo stesso valore hash (in questo caso non ho l'input M).
- **One-way**: dato y è computazionalmente difficile trovare M tale che y=h(M).

Sicurezza forte → Sicurezza debole è un'implicazione vera, ma non viceversa.

Sicurezza forte → One-way, ma non viceversa, si dimostra per contraddizione (solo l'idea del concetto, niente dettagli): assumo che l'implicazione non sia vera, e dunque la sicurezza è forte ma non è one-way. Poiché One-way=F, allora esiste un algoritmo di inversione h e quindi esiste un algoritmo che rompe la sicurezza forte e quindi calcola le collisioni con probabilità $\geq 1/2$. L'algoritmo è il seguente:

1. Scegli a caso x in X
2. $z \leftarrow h(x)$
3. $x' \leftarrow$ **ALG**(z)
4. **If** $x' \neq x$ **then** (x', x) è una collisione
   else fallito

Non si forniscono ulteriori informazioni riguardo quest'algoritmo, e con questo si conclude l'accenno alla dimostrazione.

Quant'è la lunghezza della funzione Hash? Supponiamo che la funzione Hash che si vuole è di 3 bit, quindi tutti i possibili valori sono 2^3 bit, quanti valori si devono scegliere per essere certi di trovare una collisione? Sono 9, perché se ne scelgo 8 si può essere così sfortunati da trovare tutti valori diversi. In generale quindi, se |Z|= numero di bit che voglio che la funzione Hash restituisca, per essere sicuro di trovare una collisione, devo generare |Z| + 1 diversi valori di M.

Ovviamente poiché questa è una prova certa, un consiglio è di generare valori Hash di almeno 160 bit in quanto le possibili combinazioni sono 2^{160} che sono cifre molto alte e quindi difficilmente calcolabili.

Si sta parlando di un concetto di certezza, ossia avere una buona probabilità (anche 50%) di trovare una collisione, la certezza non è richiesta.

Quanti tentativi servono per avere una certa probabilità ε di trovare una collisione?

Per rispondere a questa domanda, ci agganciamo al seguente problema, quello del **paradosso del compleanno.**

PARADOSSO DEL COMPLEANNO:

"Quante persone scegliere a caso affinché, con probabilità ≥ 0.5 ci siano almeno due con lo stesso compleanno?"

Supponiamo di avere 50 persone. La probabilità di avere due persone che festeggiano il compleanno allo stesso giorno è ovviamente minore a 0.5.

Si potrebbe pensare, intuitivamente, di pensare ad un numero di persone che sia almeno grande quanto la metà dei giorni che ci sono in un anno (182), ma ciò è sbagliato, ed è per questo che si parla di paradosso.

La risposta infatti, è che bastano 23 persone. Scelte 23 persone a caso, ho una probabilità maggiore del 50% che trovo almeno 2 persone con lo stesso compleanno. Se si pensa che bastano solo 23 persone, tenendo conto di 365 giorni in un anno, è veramente un numero molto piccolo e rappresenta un'idea totalmente opposta a quella di considerare almeno 182 persone. Questo è il motivo del paradosso. Vediamo la matematica che c'è dietro a queste considerazioni:

La probabilità che tra t valori scelti a caso ed indipendentemente non ci siano valori uguali, la probabilità è:

$$\frac{365 \cdot 364 \cdot \ldots \cdot (365 - t + 1)}{365^t}$$

← casi favorevoli

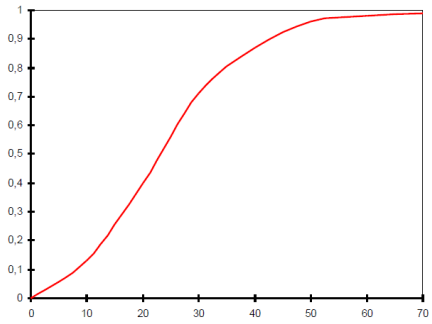
← totale casi

365 perché la prima persona ha 365 giorni possibili per avere una data di compleanno, la seconda 364 perché non può scegliere la data della persona precedente, e così via…

La probabilità che tra t valori scelti a caso ed indipendentemente almeno 2 valori sono uguali è il complemento a 1 di questo valore e cioè:

$$1-\frac{365\cdot 364\cdot \ldots \cdot (365-t+1)}{365^t}$$

Questa funzione è disegnabile nel seguente modo:



Se andiamo a rappresentare la t=23, allora si ottiene che ε= 0.5073. Se t=100, ε=0.9999997.

Il collegamento tra funzioni Hash e il paradosso del compleanno è che così come si possono trovare 2 persone che festeggiano lo stesso compleanno, si può trovare una collisione tra 2 valori Hash uguali. Posso quindi considerare il giorno del compleanno come il valore Hash. È come se si facesse un Hash di tutte le date di nascita delle persone. Quindi questo valore Hash è il compleanno e ognuno ha la sua data di complenno.

Se faccio un analisi più approfondita, il numero di elementi da scegliere se si vuole che la probabilità che ci siano almeno due elementi sia uguale a ε è più o meno:

$$t \approx \sqrt{n \cdot 2\ln\left(\frac{1}{1-\varepsilon}\right)}$$

che mi permette di ottenere una certa probabilità che ci sia una collisione.

Se ε=0.5 allora t~$1.17\sqrt{n}$

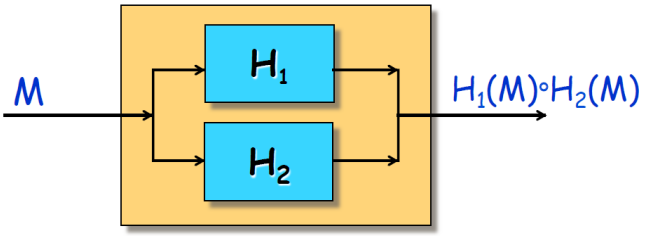
Applicazione sul paradosso del compleanno: n=365 e ε=0.5 allora t= 22.3 quindi come detto in precedenza ma ora visto con un'altra formula matematica, sono necessarie t=22-23 persone affinché la probabilità di avere 2 persone che festeggiano il compleanno lo stesso giorno sia maggiore del 50%.

In relazione alle funzioni Hash, tornando al discorso in cui abbiamo detto teoricamente che se si generano valori Hash di 160 bit, allora si ha una certa tranquillità sul fatto di non ricevere attacchi:

se n= 2^{160}, allora t~ 2^{80}, quindi trovare una collisione, almeno pochi anni fa era veramente difficile, al giorno d'oggi questa probabilità è diminuita.

se n= 2^{80} allora t~ 2^{40}, implica che 2^{40} tentativi trovo una collisione ed è un numero basso quindi non va bene.

Un'altra tecnica utilizzata riguardo le funzioni Hash, è quella della composizione, cioè, se ho due funzioni diverse e sono indeciso su quale utilizzare per avere il valore Hash, si potrebbe pensare di comporle: la nuova funzione Hash è la composizione delle 2 funzioni Hash. Ciò significa che con due funzioni diverse, si ha un vantaggio per la sicurezza poiché trovare un inverso o una collisione per questa nuova funzione Hash rappresenta il trovare 2 messaggi che collidono contemporaneamente per la prima funzione H_1 e la seconda H_2. Lo svantaggio di questa operazione è la lunghezza dell'output della nuova funzione Hash definita. A seconda dell'applicazione, questa tecnica può essere utile o meno.



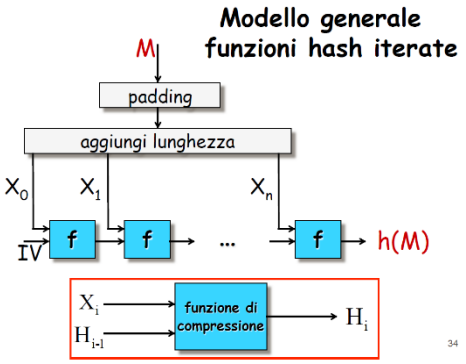
Il modello generale per funzioni hash iterate è il seguente:

L'input è di taglia arbitraria, e generalmente, tutte le funzioni hash si basano sulla funzione di compressione, che viene utilizzata iterativamente per più volte. La procedura è la seguente:

viene preso il messaggio M, a cui aggiungo un eventuale padding nel caso in cui il messaggio non ha una lunghezza opportuna. Una volta generato un messaggio di lunghezza opportuna lo divido in blocchi, ed ogni blocco va in input alla funzione **f**. Ad esempio, osservando la prima funzione **f**, l'output di **f** entra in input alla **f** successiva, insieme al blocco X_1, e così via.

La funzione **f** è quindi una funzione definita per blocchi fissati e la utilizzo più volte in base al numero di blocchi del messaggio, per ottenere alla fine il valore Hash h(M).

Qualche esempio concreto di funzione Hash è:



11.0 MD5

In cui MD sta per **M**essage **D**igeste, 5 indica la versione. La lunghezza dell'output è fissata, ed è di 128 bit. È una funzione Hash che è ottimizzata su architetture a 32 bit little-endian (presa una parola di 32 bit, quindi formata da 4 byte $a_1a_2a_3a_4$, questa parola rappresenta l'intero $a_4 2^{24}+a_3 2^{16}+a_2 2^8+a_1$).

Il funzionamento è il seguente: MD5 processa il messaggio in blocchi di 512 bit, in cui ogni blocco consta di 16 parole di 32 bit (16*32=512 bit), se il messaggio originario M che viene dato in input non arriva ad un multiplo di 512 bit allora bisogna aggiungere un padding, in modo da ricavare il messaggio M':

$M' = M\ 100....0\ b$ in cui metto un 1 e tanti 0, e negli ultimi 64 bit metto la lunghezza del messaggio originario.

L'idea dell'MD5, è quello di utilizzare delle funzioni, di volta in volta, in 4 round che sono facili da calcolare

Funzioni definite su parole di 32 bit:

round 1:	$F(X,Y,Z) = (X \wedge Y) \vee ((\neg X) \wedge Z)$	(if X then Y else Z)
round 2:	$G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge (\neg Z))$	(if Z then X else Y)
round 3:	$H(X,Y,Z) = X \oplus Y \oplus Z$	(bit di parità)
round 4:	$I(X,Y,Z) = Y \oplus (X \vee (\neg Z))$	(nuova funzione)

X	Y	Z	F	G	H	I
0	0	0	0	0	0	1
0	0	1	1	0	1	0
0	1	0	0	1	1	0
0	1	1	1	0	0	1
1	0	0	0	0	1	1
1	0	1	0	1	0	1
1	1	0	1	1	0	0
1	1	1	1	1	1	0

Il primo e il secondo round sono in realtà la costruzione di un costrutto if-then-else. Una volta espresso questi costrutti in operazioni logiche, le operazioni vengono fatte bit per bit per le parole di 32 bit.

La terza funzione di round è lo XOR tra le 3 parole che equivale al bit di parità.

La funzione del round 4 è una funzione nuova creata appositamente per il MD5.

Ogni round consiste di 16 operazioni fatte in questo modo: la notazione concisa è [ABCD.k.s.i]

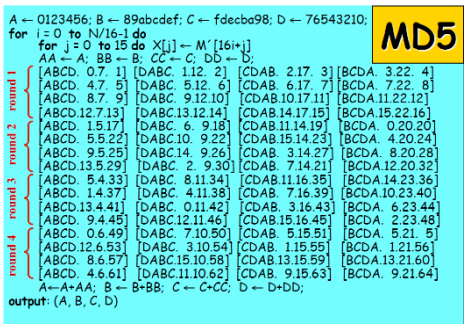
$A \leftarrow B + ((A + W(B,C,D) + X[k] + T[i])) \lll s$ in cui W rappresenta una delle funzioni espresse sopra a seconda del round in cui ci si trova (può quindi essere F-G-H-I), k è l'indice della parola, s indica lo shift ciclico, i è l'indice dell'iterazione

La costante T[i] che viene aggiunta, viene presa dai primi 32 bit della seguente funzione di espansione (leggere solo): $|\sin(i)| = \lfloor 2^{32} \cdot |\sin(i)| \rfloor$ (i in radianti)

, è la tabella delle varie costanti è la seguente:

1	d76aa478	17	f61e2562	33	fffa3942	49	f4292244
2	e8c7b756	18	c040b340	34	8771f681	50	432aff97
3	242070db	19	265e5a51	35	6d9d6122	51	ab9423a7
4	c1bdceee	20	e9b6c7aa	36	fde5380c	52	fc93a039
5	f57c0faf	21	d62f105d	37	a4beea44	53	655b59c3
6	4787c62a	22	02441453	38	4bdecfa9	54	8f0ccc92
7	a8304613	23	d8a1e681	39	f6bb4b60	55	ffeff47d
8	fd469501	24	e7d3fbc8	40	bebfbc70	56	85845dd1
9	698098d8	25	21e1cde6	41	289b7ec6	57	6fa87e4f
10	8b44f7af	26	c33707d6	42	eaa127fa	58	fe2ce6e0
11	ffff5bb1	27	f4d50d87	43	d4ef3085	59	a3014314
12	895cd7be	28	455a14ed	44	04881d05	60	4e0811a1
13	6b901122	29	a9e3e905	45	d9d4d039	61	f7537e82
14	fd987193	30	fcefa3f8	46	e6db99e5	62	bd3af235
15	a679438e	31	676f02d9	47	1fa27cf8	63	2ad7d2bb
16	49b40821	32	8d2a4c8a	48	c4ac5665	64	eb86d391

L'algoritmo totale è il seguente (vedi solo):

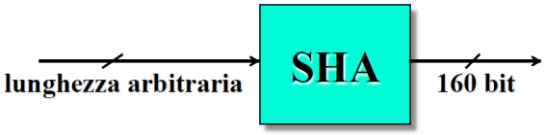


In cui vengono usate più volte le varie funzioni dei 4 round, applicate con vari parametri.

L'MD5 è stata oggetto di attacchi. Il più conosciuto ha stabilito che si possono trovare collisioni in meno di un minuto. Il grosso motivo per cui si trovano delle collisioni, è che essendo di 128 bit, l'attacco, se utilizzo il paradosso del compleanno, ha essenzialmente una complessità dell'ordine di 2^{65} e non è quindi molto sicuro.

Data questa criticità si è scelto di andare verso funzioni più sicure:

11.1 SHS/SHA

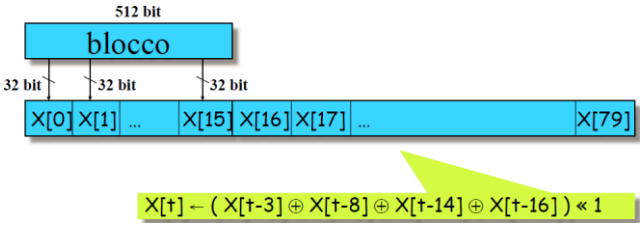


SHS sta per **S**ecure **H**ash **S**tandard, mentre SHA sta per **S**ecure **H**ash **A**lgorithm, in genere viene più utilizzato SHA. Ha lo stesso principio di funzionamento di MD5, ma, in questa funzione Hash, l'output è di 160 bit (se facessi un attacco che si basa sul paradosso del compleanno, la complessità è nell'ordine di 2^{80} che è un po' più sicuro) . È una funzione Hash che è ottimizzata su architetture a 32 bit big-endian (presa una parola di 32 bit, quindi formata da 4 byte $a_1a_2a_3a_4$, questa parola rappresenta l'intero $a_1 2^{24}+a_2 2^{16}+a_3 2^8+a_4$).

Le caratteristiche del messaggio M sono identiche a quelle di MD5 (blocchi 512 bit, eventuale padding) e per questo non vengono riportate.

Per quanto riguarda invece l'idea tecnica, c'è qualche variazione:

Preso il messaggio M di 512 bit, lo divido nei vari blocchi, parole di 32 bit e ad un certo punto applico un'espansione che mi consente di ottenere 80 parole di 32 bit. Ogni parola in più che aggiungo è data dal calcolo X[t], in cui uso parole provenienti da blocchi precedenti, come si vede in foto.



Le funzioni logiche di SHA sono simili a quelle di MD5:

```
Funzione  F(t,X,Y,Z)
round t=  0,...,19:  F(t,X,Y,Z) = (X ∧ Y) ∨ ((¬ X) ∧ Z)        (if X then Y else Z)
round t=20,...,39:  F(t,X,Y,Z) = X ⊕ Y⊕ Z                     (bit di parità)
round t=40,...,59:  F(t,X,Y,Z) = (X ∧ Z) ∨ (Y ∧ Z) ∨ (X ∧ Y)   (2 su 3)
round t=60,...,79:  F(t,X,Y,Z) = Y ⊕ X ⊕ Z                     (bit di parità)
```

X	Y	Z	F(0,..)	F(20,..)	F(40,..)	F(60,..)
0	0	0	0	0	0	0
0	0	1	1	1	0	1
0	1	0	0	1	0	1
0	1	1	1	0	1	0
1	0	0	0	1	0	1
1	0	1	0	0	1	0
1	1	0	1	0	1	0
1	1	1	1	1	1	1

Così come in MD5, anche qui la costante additiva K[t] viene generata con il seguente criterio:

```
round  t =   0,...,19:  5a827999
round  t = 20,...,39:  6ed9eba1
round  t = 40,...,59:  8f1bbcdc
round  t = 60,...,79:  ca62c1d1
```

L'algoritmo totale è il seguente (vedi solo):

```
A=67452310; B=efcdab89; C=98badcfe; D=10325476; E=c3d2e1f0;
for  i = 0  to  N/16-1 do
    for  j = 0  to 15 do
        X[j] ← M´[16i+j]
    for  t = 16  to 79 do
        X[t] ← ( X[t-3] ⊕ X[t-8] ⊕ X[t-14] ⊕ X[t-16] ) « 1
    AA ← A;  BB ← B; CC ← C;  DD ← D;  EE ← E;
    for t=0 to 79 do
        TEMP ← (A«5) + F(t,B,C,D) + E + X[t] + K[t]
        E ← D
        D ← C
        C ← (B«30)
        B ← A
        A ← TEMP
    A ← A + AA;  B ← B + BB;  C ← C + CC;  D ← D + DD;  E ← E + EE;
output: (A, B, C, D, E)
```

SHA-1

espansione
da 16 ad 80 parole
"«1" non c'era in SHA

Date in input A,B,C,D,E di 32 bit (5*32= 160 bit di output), questi valori vengono modificati in base all'input, e vengono poi restituiti da questa funzione di compressione o Hash.

Il miglior attacco effettuato con un sistema di sicurezza SHA-1, stima di poter trovare una collisione in un tempo di ordine $2^{57,5}$, a causa di quest'attacco mai realizzato ma mai implementato, diverse aziende quale Microsoft ha deciso di disincentivare l'utilizzo di SHA-1.

La caratteristica di questo attacco, è che si trova una collisione, ma potrebbe non avere alcun effetto pratico, cioè vengono trovate semplicemente delle stringhe che collidono ma nient'altro.

Il NIST ha deciso di costruire degli algoritmi SHA-1, SHA-2 e SHA-3 che hanno come obiettivo maggiore sicurezza a discapito degli attacchi precedenti, in particolare SHA-2 è formato da: SHA-224, SHA-256, SHA-384, SHA-512, e la sigla finale indica il numero di bit rilasciati dall'algoritmo. Si vede come la lunghezza dei bit restituiti è molto più lunga. Quello più usato è SHA-256, se si fa un attacco sulla base del paradosso del compleanno, il tempo per trovare una collisione è 2^{128} che è un tempo molto alto e quindi da la dovuta tranquillità al fronte di attacchi.

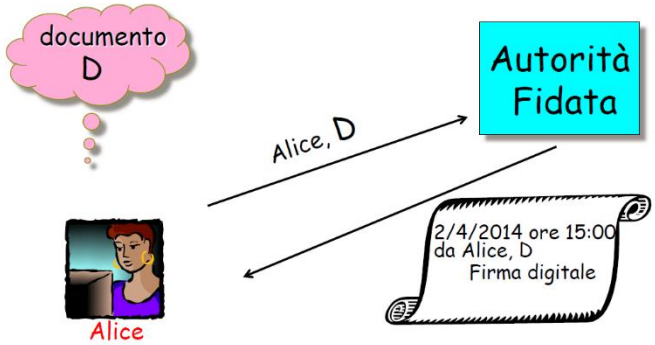
Contemporaneamente alla costruzione di SHA-2, in anni successivi è stato rilasciato SHA-3 che si compone di: SHA3-224, SHA-3-256, SHA3-384, SHA3-512. L'utilizzo di SHA-2 o SHA-3 è indifferente in quanto al giorno d'oggi non è stato rilasciato alcun attacco. In SHA-3 vengono aggiunte 2 extandable-output functions (XOFs): SHAKE128 e SHAKE256 (nessun dettaglio, solo informativo).

SHA-2 ha gli stessi principi di MD4, MD5 e SHA-1.

SHA-3 si basa su funzioni spugna (Sponge Construction, nessun dettaglio).

All'inizio della prima pagina delle Funzioni Hash, si è parlato di un utilizzo di marcatura temporale, cioè si vuole sapere quando un documento digitale è stato creato. In generale, la marca temporale di un documento è qualcosa aggiunto ad esso che prova che il documento è stato "prodotto" prima, dopo, oppure ad un fissato momento. Questo concetto teorico si può mostrare praticamente partendo da una soluzione naive (ingenua):

Se Alice ha un documento e c'è un'Autorità Fidata, allora Alice invia il documento D all'Autorità Fidata con il suo identificativo e l'Autorità rilascia un comunicato che stabilisce che al giorno gg/mm/aaaa alle ore hh:mm Alice ha inviato il documento D con la firma digitale. L'autorità sancisce quindi un determinato evento con tutti i dettagli temporali. All'interno di questo documento che viene generato dall'Autorità Fidata, ci sta anche una marca temporale che viene descritta come "evidenza informatica che consente di rendere opponibile a terzi un riferimento temporale". Il tempo preciso che viene inserito nel documento viene calcolato in base alla scala temporale UTC (Tempo Universale Coordinato) ed in particolare, la differenza temporale sul



documento, in relazione a quando effettivamente viene emesso, non deve essere superiore ad un minuto, questo perché sulla rete viaggiano tantissime informazioni, tra cui l'indicazione del tempo.

Questa idea, descritta come soluzione naive, dunque ingenua, fa nascere 3 problematiche:

1. Dimensione del documento D, per la comunicazione e per la memorizzazione dell'Autorità Fidata.
2. Privatezza del contenuto di D.
3. Quanto è fidata l'Autorità Fidata?

Per risolvere questi problemi, gli step da fare sono i seguenti:

Per risolvere i primi 2 problemi, mi basta mandare l'Hash del messaggio D, perché la dimensione sarà fissata a seconda dell'algoritmo Hash utilizzato, indipendentemente da quanto possa essere grande il messaggio, e l'autorità non potrà mai risalire al testo originale del messaggio. In questo modo sulla documentazione che rilascia l'Autorità Fidata, non ci sarà scritto che Alice ha inviato il documento D, bensì il valore Hash del documento D. Questo crea una sicurezza elevata, poiché non si possono calcolare collisioni.

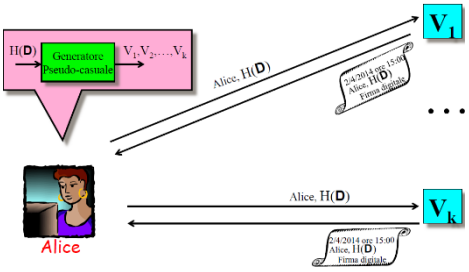
Per risolvere il terzo problema dell'Autorità fidata, Alice manda il valore Hash del messaggio D, ad un Certificatore accreditato, chiamato così perché è un certificatore riconosciuto dai vari decreti normativi. Questo garantisce la totale sicurezza.

Nel pratico, due possibili soluzioni al problema dell'Autorità Fidata sono mediante l'utilizzo di Protocolli:

- **Protocolli distribuiti** (senza Autorità Fidata): avere più "testimonianze" del tempo.
- **Protocolli con "link"** (con Autorità Fidata): collegare tra loro le marche dei documenti

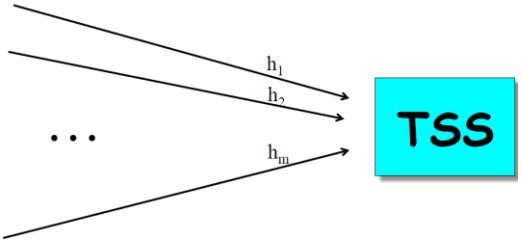
Un approccio di protocollo distribuito è il seguente:

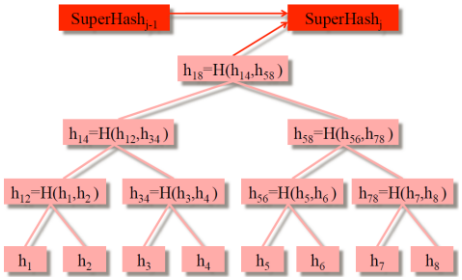
Se Alice deve dare una marca ad un certo documento, genera un insieme di utenti V_1, V_2,..., V_k a seconda dell'Hash del documento che deve calcolare grazie ad un generatore. Ad ogni utente V_1, V_2,..., V_k, Alice manda le richieste di avere una marca temporale sul proprio documento. Ogni utente invia ad Alice la propria marca temporale cosicché la marca temporale è fatta da un insieme di marche temporali. La generazione degli utenti è ovviamente casuale in modo tale che Alice non possa decidere chi contattare per generare delle marche false.



Un approccio con protocollo distribuito crea dei problemi (non spiegati), per cui l'idea più ragionevole è di utilizzare un protocollo con "link", in cui tutte le richieste vengono collegate tra di loro. L'idea è la seguente:

L'Autorità Fidata TSS riceve h_1,...,h_m richieste in un certo intervallo di tempo. Queste m richieste devono essere organizzate in maniera tale da non poter più cambiare idea. Per fare questa operazione, decide di costruire un albero di Hash:



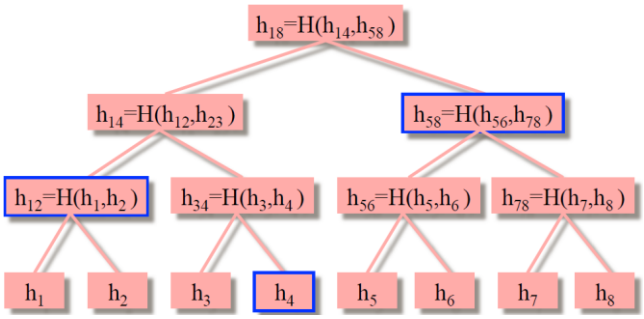


Si assuma che le richieste che arrivano in un determinato istante di tempo sono 8 e le inserisco come foglie dell'albero. Per ogni coppia di Hash, calcolo il valore Hash della concatenazione della coppia presa (ad esempio, dati h_1 e h_2, ottengo h_{12} che è uguale all'Hash della concatenazione di h_1,h_2). Questo processo viene ripetuto per le successive coppie fino ad ottenere h_{18}.

Poiché il tempo di computazione che genera h_{18} potrebbe essere di qualche minuto, si andrebbe a perdere il tempo preciso in cui un utente effettua la richiesta, falsando quindi il documento. Per colmare questa lacuna, e quindi recuperare questo intervallo di tempo, esiste un SuperHash che viene calcolato con il SuperHash dell'intervallo precedente (il tempo effettivo in cui mando la richiesta) e quello corrente. L'output di questo SuperHash rappresenta il tempo preciso in cui viene effettuata la richiesta. Viene generata quindi la marca temporale:

ID utente della richiesta
h_i
data ed ora
h_{1m} (valore hash della radice dell'albero)
info necessarie per verificare che h_i è stato utilizzato per costruire l'albero con radice h_{1m}
$SuperHash_{j-1}$ e $SuperHash_j$
Firma del TSS

Quali sono le informazioni necessarie per verificare che h_i è stato utilizzato per costruire l'albero con radice h_{1m}? Viene illustrato con un esempio:



Si supponga di aver chiesto una marca per h_3 e l'Autorità di certificazione da il valore h_{18}. Se ho h_3 e h_{18}, come faccio a sapere che, quando è stato calcolato h_{18}, è stato realmente utilizzato h_3? Per questa verifica, si potrebbe pensare di fornire tutto l'albero e quindi mostrare effettivamente che h_3 è una foglia dell'albero e che viene quindi utilizzata nei calcoli, questa informazione è lineare al numero di Hash.

Un'altra idea sarebbe quella di fornire un'informazione che è logaritmica in base al numero di Hash, cioè dipende dall'altezza dell'albero, e quindi si risparmierebbe molto tempo. Quindi, se devo fornire informazioni su h_3, posso dare h_4 in modo che l'utente può calcolare h_{34}, devo poi fornire h_{12} in modo che l'utente può calcolare h_{14}, e come ultimo step devo fornire h_{58} per poter calcolare h_{18}. In figura i valori in blu. In generale, devo quindi fornire i nodi fratelli che si trovano nel cammino che va da h_3 alla root h_{18}.

Fissato il valore Hash della radice, non è possibile inserire un nuovo valore nell'albero di Hash e cambiare anche un solo valore nell'albero di Hash, altrimenti si determinerebbe una collisione per la funzione Hash, e questo spiega il perché del concetto definito precedentemente come "non poter cambiare più idea".

Una volta prodotti SuperHash, questi possono essere distribuiti agli utenti su Internet, quotidiani, mail.