Hash Functions

Federico Casu

February 7, 2024

1 Introduzione

Una funzione hash è una funzione matematica avente la seguente proprietà (informale):

> Una funzione hash si definisce tale quando, per qualsiasi input, indipendentemente dalla sua lunghezza, la funzione restituisce un output di dimensione costante.

Formalizzare la proprietà che abbiamo appena citato è molto semplice:

$$\mathsf{H} : \{0,1\}^* \to \{0,1\}^n$$

In un contesto in cui la sicurezza è una priorità, le funzioni hash necessitano ulteriori proprietà. Facciamo un esempio:

Esempio 1. Supponiamo di voler velocizzare il processo di firma digitale di un documento. Come ben sappiamo, la crittografia a chiave pubblica è più lenta di 2 (talvolta 3) ordini di grandezza rispetto alla crittografia a chiave simmetrica. Per diminuire il costo della firma digitale possiamo firmare il **digest** del documento, ovvero la firma digitale non viene apposta sull'intero documento ma su una stringa di bit, quest'ultima di dimensione molto minori rispetto al documento, prodotta da una funzione hash.

Da questo esempio possiamo dedurre alcune proprietà legate alla sicurezza:

1. La funzione hash deve essere *one-way*, ovvero $\mathsf{H}(\mathsf{x}) = \mathsf{y}$ deve essere semplice da calcolare ma difficile da invertire ($\mathsf{H}^{-1}(\mathsf{y}) = \mathsf{x}$ deve essere computazionalmente infattibile).
2. L'output della funzione, $\mathsf{H}(\mathsf{x}) = \mathsf{y}$, deve rappresentare univocamente l'input x.
3. La funzione hash non deve richiedere una chiave.
4. Il digest prodotto dalla funzione hash deve essere sensibile ad ogni bit di input. In altre parole, si vorrebe che se due input differiscono di un solo bit allora i rispettivi output (in media) differiscono in almeno la metà dei bits.

Purtroppo, tra le proprietà sopra elencate, c'è ne una che matematicamente non è possibile ottenere: per definizione, una funzione hash è una funzione *many-to-one*. Visto che il dominio della funzione ha una cardinalità maggiore del codominio, $\exists\ \mathsf{x}_1 \neq \mathsf{x}_2$ tale che $\mathsf{H}(\mathsf{x}_1) = \mathsf{H}(\mathsf{x}_2)$.
Al solito, siamo ingegneri, e ci accontentiamo di un po' meno rigore matematico a fronte di un approssimazione che nella pratica è verosimile.

2 Funzioni Hash Sicure

Prima di poter elencare le proprietà che rendono *sicura* una funzione hash, dobbiamo definire il concetto di collisione.

Definizione 1. (Collisione) Si ha una *collisione* quando esistono $\mathsf{x}_1 \neq \mathsf{x}_2$ tali che $\mathsf{H}(\mathsf{x}_1) = \mathsf{H}(\mathsf{x}_2)$.

Perchè le collisioni sono nostre nemiche? Facciamo un esempio.

Esempio 1. Supponiamo che Alice, per firmare un certo contratto di pagamento, abbia dovuto firmare il digest del documento. Bob, che è venuto a sapere ciò, vuole fare un dispetto ad Alice e scopre che il digest $\mathsf{y} = \mathsf{H}(\mathsf{x})$ del contratto collide con un altro documento $\mathsf{x}' \neq \mathsf{x}$, avente lo stesso

contenuto del documento originale ma accuratamente modificato nella somma di denaro ed in altre posizioni non visibili. Bob potrebbe sostituire il contratto firmato da Alice con il contratto che lui, in mala fede, ha modificato: nessuno se ne accorgerebbe perchè

$$\mathsf{H}(\mathsf{x}) = \mathsf{H}(\mathsf{x}') \rightarrow \mathsf{Sign}(\mathsf{K}^{\mathsf{A}}_{\mathsf{priv}}, \mathsf{H}(\mathsf{x})) = \mathsf{Sign}(\mathsf{K}^{\mathsf{A}}_{\mathsf{priv}}, \mathsf{H}(\mathsf{x}'))$$

Definizione 2. (Pre-image resistance) Una funzione hash gode della proprietà di *pre-image resistance* se, dato un digest y qualsiasi, è computazionalmente infattibile calcolare x tale che $\mathsf{y} = \mathsf{H}(\mathsf{x})$.

Definizione 3. (Second pre-image resistance) Una funzione hash gode della proprietà di *second pre-image resistance* se, dato x_0 qualsiasi, è computazionalmente infattibile trovare x_1 tale che $\mathsf{H}(\mathsf{x}_0) = \mathsf{H}(\mathsf{x}_1)$. Tale proprietà talvolta è detta *weak collision resistance*.

Definizione 4. (Collision resistance) Una funzione hash gode della proprietà di *collision resistance* se è computazionalmente infattibile trovare una coppia $(\mathsf{x}_0, \mathsf{x}_1)$, $\mathsf{x}_0 \neq \mathsf{x}_1$, tale che $\mathsf{H}(\mathsf{x}_0) = \mathsf{H}(\mathsf{x}_1)$. Tale proprietà talvolta è detta *strong collision resistance*.

Definizione 5. (One-way hash function) Una funzione hash è detta *one-way hash function* se gode della proprietà di *pre-image resistance* e della proprietà di *second pre-image resitance*.

Definizione 6. (Cryptographically Secure hash function) Una funzione hash è detta *cryptographically secure hash function* se gode delle proprietà di *second pre-image resitance* e *collision resistance*.

Fact 1. Collision resistance → 2nd pre-image resistance Se una funzione hash gode della proprietà di *collision resistance* allora gode anche della proprietà di *second pre-image resistance* (strong collision resistance implica weak collision resistance, non è vero il contrario).

Fact 2. Se una funzione hash gode della proprietà di *collision resistance* non è detto che goda anche della proprietà di *pre-image resistance* (strong collision resistance non implica pre-image resistance).

3 Attacchi di tipo *black box*

Sia nella crittografia a chiave simmetrica, sia nella crittografia a chiave pubblica, abbiamo potuto notare il fatto che è sempre possibile eseguire un attacco a *forza bruta*. Questa tipologia di attacchi consistono nel trattare l' *encryption scheme* come una scatola nera che prende in ingresso un messaggio in chiaro e restituisce il messaggio cifrato corrispondente diverso a seconda della chiave di cifratura utilizzata. Le funzioni hash possono essere attaccate: l'obiettivo dell'attacante è ottenere una collisione.

Supponiamo di voler invertire una funzione hash, ovvero vorremo ottenere l'input x che ha prodotto il digest y. L'attacco è il seguente:

Algorithm 1 Guessing Attack

```
repeat
    x ← random()
until H(x) == y
```

Qual'è la complessità dell'algoritmo? Dipende dalla dimensione dell'output: visto che il codominio ha cardinalità $\mathsf{Card}\left(\{0,1\}^{\mathsf{n}}\right) = 2^{\mathsf{n}}$, al più la funzione hash è in grado di generare 2^{n} output distinti. Ciò significa che per trovare x tale che $\mathsf{y} = \mathsf{H}(\mathsf{x})$ è necessario, in media, eseguire $\mathsf{O}(2^{\mathsf{n}})$ iterazioni.
Si presti attenzione al fatto che i black box attacks rappresentano solamente un limite superiore. In altre parole, i black box attacks ci dicono che peggio di così non possiamo fare. Purtroppo non possiamo assumere che la complessità di un algoritmo in grado di rompere una funzione hash sia $\mathsf{O}(2^{\mathsf{n}})$ perchè non possiamo affermare con certezza che non esistano attacchi più efficienti.

Possiamo suddividere la classe degli attacchi alle funzioni hash in due sottoclassi:

1. **Existential forgery**: l'attacante non ha controllo sull'input x.
2. **Selective forgery**: l'attacante ha completo (o parziale) controllo sull'input x.

3.1 Birthday attack

Le funzioni hash, indipendentemente dal loro design, possono essere soggette ad una tipologia di attacco la cui complessità è $O(2^{n/2})$!
Prendiamo in esame i seguenti semplici problemi di probabilità, apparantemente scollegati dal mondo delle funzioni hash, ma che in seguito si riveleranno fondamentali.

Esempio 1. Si consideri un insieme di $t = 23$ persone. Qual'è la probabilità che almeno una persona sia nata il 25 Dicembre?
Soluzione: l'evento E = "almeno una persona del gruppo è nata il 25 Dicembre" è dato dall'unione degli eventi "persona j nata il 25 Dicembre", $j = 1, ..., 23$. Ciascun evento è indipendentemente e disgiunto dagli altri. Segue che la probabilità può essere calcolata come:

$$P(E) = \sum_{j=1} 23 \frac{1}{365} = \frac{23}{365} = 0.063$$

Esempio 2. (Birthday Paradox) Si consideri un insieme di $t = 23$ persone. Qual'è la probabilità che almeno due persone siano nate nello stesso giorno?
Soluzione: possiamo calcolare la probabilità $P(E)$ come $P(E) = 1 - P(Q)$, dove Q è l'evento "nessun individuo è nato lo stesso giorno di un altro individuo".

$$P(e) = 1 - \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right) \cdot \cdots \cdot \left(1 - \frac{22}{365}\right) = 0.507$$

Notiamo che la probabilità dell'evento $Q = \exists$ 2 persone nate lo stesso giorno è, circa, di un ordine di grandezza maggiore rispetto alla probabilità del primo problema. Questi due semplici problemi ci permettono di capire che esiste una seconda tipologia di attacco la cui complessità è nell'ordine di $O(2^{n/2})$.

Fact 1. Il problema Birthday Paradox è equivalente al seguente problema: qual'è la probabilità di trovare una coppia $\langle x_0, x_1 \rangle$, $x_0, \neq x_1$, $H(x_0) = H(x_1)$.

Di fatto, possiamo paragonare il numero di persone presenti nella sala in cui si svolge la narrazione del Birthday Paradox al numero di tentativi compiuti per trovare una collisione a partire da una coppia di input qualsiasi. Il passo successivo è il calcolo del numero di tentativi necessari per forgiare una collisione:

$$\begin{aligned} P(Q) &= \left(1 - \frac{1}{2^n}\right) \cdot \left(1 - \frac{2}{2^n}\right) \cdot \cdots \cdot \left(1 - \frac{t-1}{2^n}\right) = \\ &= \prod_{j=1}^{t-1} \left(1 - \frac{j}{2^n}\right) \end{aligned}$$

Ora applichiamo una approssimazione: $1 - x \approx e^{-x}$ se $x << 1$. Segue:

$$\begin{aligned} P(Q) &= \prod_{j=1}^{t-1} \left(1 - \frac{j}{2^n}\right) \approx \\ &\approx \prod_{j=1}^{t-1} e^{-\frac{j}{2^n}} = \\ &= e^{-\left(\frac{1}{2^n} + \frac{2}{2^n} + \cdots + \frac{t-1}{2^n}\right)} = \\ &= e^{-\frac{1+2+\cdots+t-1}{2^n}} = \\ &= e^{-\frac{t(t-1)}{2 \cdot 2^n}} \approx \\ &\approx e^{-\frac{t^2}{2 \cdot 2^n}} \end{aligned}$$

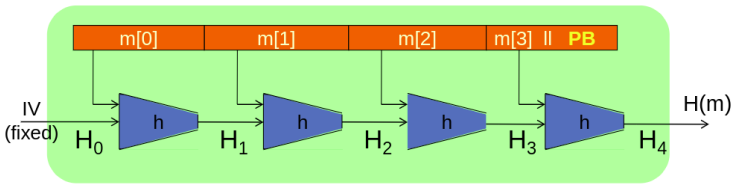

Figure 1: Merkle-Damgard design.

Adesso vogliamo dare un valore all'incognita t: quanti tentativi è necessario compiere per ottenere una collisione con una certa probabilità λ?

$$
\begin{aligned}
&P(Q) \approx e^{-\frac{t^2}{2\cdot 2^n}} \\
&P(E) = 1 - P(Q) = 1 - e^{-\frac{t^2}{2\cdot 2^n}} \\
&\quad \text{supponiamo } P(E) = \lambda \\
&\quad 1 - e^{-\frac{t^2}{2\cdot 2^n}} = \lambda \\
&\quad e^{-\frac{t^2}{2\cdot 2^n}} = 1 - \lambda \\
&\quad -\frac{t^2}{2\cdot 2^n} = \log(1-\lambda) \\
&\quad \frac{t^2}{2\cdot 2^n} = \log(\frac{1}{1-\lambda}) \\
&\quad t^2 = 2^{n+1}\log(\frac{1}{1-\lambda}) \\
&\quad t = \sqrt{2^{n+1}\log(\frac{1}{1-\lambda})} \\
&\quad t = 2^{\frac{n+1}{2}}\sqrt{\log(\frac{1}{1-\lambda})}
\end{aligned}
$$

Cosa possiamo notare? Il parametro λ è pressochè ininfluente: l'ordine del numero di operazioni è dato da $2^{\frac{n+1}{2}}$. Il numero di messaggi che dobbiamo hashare per trovare una collisione è dell'ordine della radice quadrata della cardinalità del codominio, ovvero $O(2^{n/2})$.

4 Come costruire una funzione hash

Finora abbiamo discusso i requisiti delle funzioni di hash. Ora introduciamo come costruirle effettivamente. Esistono due possibili approcci:

1. **Funzioni hash dedicate**. Questi sono algoritmi appositamente progettati per funzionare come funzioni hash.
2. **Funzioni hash basate su cifrari a blocchi**. Si possono utilizzare cifrari a blocchi (esempio: AES) per costruire funzioni hash.

Come abbiamo visto nelle sezioni precedenti, le funzioni hash possono elaborare un messaggio di lunghezza arbitraria e produrre un output di lunghezza fissa. Nella pratica, ciò è ottenuto suddividendo l'input in una serie di blocchi di dimensioni uguali. Questi blocchi vengono elaborati sequenzialmente dalla funzione hash: ciascun blocco diventa l'input di una funzione, detta funzione di compressione, insiem al digest ottenuto dal blocco precedente. Questo design è noto come costruzione di Merkle-Damgard. L' hash del messaggio in ingresso è quindi definito come l'output dell'ultima iterazione della funzione di compressione.