Clase 11: seguridad informal y uso de funciones de hash

Fernando Virdia, versión: 0.0.1, junio 2024

6 Funciones de hash

Las funciones de hash son el primitivo criptográfico probablemente mas conocido. Si alguna vez intentarlo programar el login de una pagina web, probablemente las hayan encontrado (MD5, SHA1–3 las mas conocidas). Son también un primitivo a menudo usado incorrectamente, poniendo cuentas online en riesgo.

Definición 16 (Hash, informal) *Una función* $H\colon \{0,1\}^* \to \{0,1\}^\ell$ *se dice de "hash criptografico" si satisface las siguientes propiedades:*

- *Puede ser implementada con un algoritmo determinista eficiente.*
- *"Preimage resistance":* dado $h \in \{0,1\}^\ell$, es difícil encontrar x tal que $H(x) = h$.
- *"Second preimage resistance":* dado $(x, H(x))$, es difícil encontrar $x' \neq x$ tal que $H(x') = H(x)$.
- *"Collision-resistance" (CR):* es difícil encontrar $x \neq x'$ tal que $H(x) = H(x')$.

Comentario 34 *Por que definiciones informales? Por que para cada propiedad,* existen adversarios eficientes! *Por ejemplo:*

- *CR: dado que* $|\{0,1\}^*| > \left|\{0,1\}^\ell\right|$*, colisiones existen necesariamente. Como* H *es una función' fija, supongamos que* x, x' *sean una colisión.* $\mathcal{A} \coloneqq$ **return** (x, x') *es un algoritmo eficiente que retorna una colisión con probabilidad* 1*. Simplemente, no lo conocemos.*
- *Preimage resistance: es común que funciones de hash sean evaluadas en inputs de baja entropía, predecibles. Por lo tanto es plausible que en una aplicación practica, dado* h*, un algoritmo pueda retornar* x *tal que* $H(x) = h$*. Esta falta de entropía en entrada diferencia "preimage resistance" de la definición de "seguridad PRG".*
- *La falta de entropía en las entradas también impide una definición de second preimage resistance.*

Sin definiciones formales, demostrar seguridad de construcciones que las usan no es completamente posible. A menudo usamos el "modelo del oráculo aleatorio" (ROM, "random oracle model"), que asume que H *es una función completamente aleatoria. Alternativamente, algunos textos asumen que el hash toma en input una clave, aunque en realidad no es así.*

Lemma 11 (informal) *Sea* H *una función de hash.*

1. *Collision resistance* $\Rightarrow$ *second-preimage resistance.*
2. *Second-preimage resistance* $\Rightarrow$ *preimage resistance.*
3. *Collision resistance* $\Rightarrow$ *preimage resistance.*
4. *Collision resistance* $\not\Rightarrow$ *"partial" preimage resistance.*

Demostración. (informal) Por casos:

1. Sea $\mathcal{A}$ un adversario que rompe second-preimage resistance: dado $(x, H(x))$ retorna $x' \neq x$ tal que $H(x') = H(x)$. Podemos elegir un $x \in \{0,1\}^*$ cualquiera, calcular $x' \leftarrow \mathcal{A}(x, H(x))$. Siendo el espacio de input infinito, es improbable que $x = x'$. Por lo tanto podemos retornar una colisión (x, x').
2. Sea $\mathcal{B}$ un adversario que rompe preimage resistance: dado h retorna x tal que $H(x) = h$. En input $(x, H(x))$ podemos calcular $x' \leftarrow \mathcal{B}(H(x))$. Siendo el dominio de H infinito, probablemente $x \neq x'$, y pudimos encontrar una second preimage.

3. Sea $\mathcal{B}$ como arriba. El mismo razonamiento da una colisión, si elegimos nosotros x en primer lugar.

4. Mas allá del punto 3., es posible construir funciones que son collision resistant y por lo tanto "preimage resistant", pero para las cuales si existe un adversario que rompe preimage resistance sobre un subconjunto bien definido del espacio de input.
   Sea G una función de hash CR. Definimos $H(x) \coloneqq \begin{cases} 0||x & \text{si } |x| = n, \\ 1||G(x) & \text{si } |x| \neq n. \end{cases}$ Por inspección, H es CR dado que colisiones donde $H(x) = 0\ldots$ son imposibles y colisiones con $H(x) = 1\ldots$ implican colisiones en G. Sin embargo, si $h = H(x) = 0\ldots$, $|h| = n + 1$, x puede ser recuperado a partir de h.

□

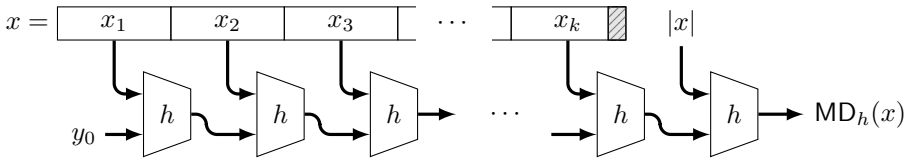

Figure 7: Una manera común de construir funciones de hash es usando PRPs o PRFs iteradas según el diseño de Merkle-Damgard (MD). Imagen tomada de [Ros21, Construction 11.2], usada con permiso del autor.

Lemma 12 (fixed-length MAC + CR hash ⇒ arbitrary-length MAC, hash-then-MAC) *Sea* $\Pi = (\mathsf{Gen}, \mathsf{Tag}, \mathsf{Ver})$ *un fixed-length* (ε', t', q')*-MAC con* $\mathcal{M} = \{0,1\}^\ell$*, y* $H\colon \{0,1\}^* \to \{0,1\}^\ell$ *un CR-hash. Definimos un MAC* $\Pi' = (\mathsf{Gen}', \mathsf{Tag}', \mathsf{Ver}')$*,*

$\mathsf{Gen}'()$	$\mathsf{Tag}'(k, m)$	$\mathsf{Ver}'(k, m, \tau)$
1 : $k \xleftarrow{\$} \mathsf{Gen}$	1 : $h \leftarrow H(m)$	1 : $h \leftarrow H(m)$
2 : **return** k	2 : $\tau \leftarrow \mathsf{Tag}(k, h)$	2 : $b \leftarrow \mathsf{Ver}(k, h, \tau)$
	3 : **return** τ	3 : **return** b

Π' *ofrece* (ε', t', q')*-seguridad MAC con* $t' \approx t$*,* $q' = q$*,* $\varepsilon' \leq \varepsilon + \Pr[\mathsf{coll}]$*, donde* coll *es la probabilidad que un adversario contra* Π *produzca una colisión en* H*.*

Demostración. Usamos un adversario $\mathcal{A}$ contra Π' para definir un adversario $\mathcal{B}$ contra Π.

$\mathcal{B}^T$	$\mathcal{O}(m)$
1 : $Q \leftarrow \{\,\}, S \leftarrow \{\,\}$	1 : $h \leftarrow H(m)$
2 : $(m^*, \tau^*) \leftarrow \mathcal{A}^{\mathcal{O}}()$	2 : $\tau \leftarrow T(h)$
3 : $h^* \leftarrow H(m^*)$	3 : $Q \leftarrow Q \cup \{(h, \tau)\}$
4 : **if** $(h^*, \tau^*) \in Q$:	4 : $S \leftarrow S \cup \{(m, \tau)\}$
5 : **return** $(\bot, \bot)$	5 : **return** τ
6 : **else return** (h^*, τ^*)	

NB: El conjunto S (en gris) simplifica el análisis, pero no es requerido por $\mathcal{B}$.
Empezamos expandiendo la ventaja de $\mathcal{A}$ contra Π' en términos de Π.

$$\begin{aligned}
\mathsf{Adv}(\mathsf{Exp}^{\mathrm{MAC}}, \mathcal{A}) &= \Pr[\mathsf{Exp}^{\mathrm{MAC}}(\mathcal{A}) \Rightarrow 1] = \Pr[\mathsf{Ver}'(k, m^*, \tau^*) \Rightarrow 1] \\
&= \Pr[\mathsf{Ver}(k, H(m^*), \tau^*) \Rightarrow 1] = \Pr[\mathsf{Ver}(k, h^*, \tau^*) \Rightarrow 1] \\
&= \cancel{\Pr[\mathsf{Ver}(k, h^*, \tau^*) \Rightarrow 1 \mid (h^*, \tau^*) \in Q] \cdot \Pr[(h^*, \tau^*) \in Q]} \\
&+ \cancel{\Pr[\mathsf{Ver}(k, h^*, \tau^*) \Rightarrow 1 \mid (h^*, \tau^*) \notin Q] \cdot \Pr[(h^*, \tau^*) \notin Q]} \\
&\leq \Pr[(h^*, \tau^*) \in Q] + \Pr[\mathsf{Ver}(k, h^*, \tau^*) \Rightarrow 1 \mid (h^*, \tau^*) \notin Q]
\end{aligned}$$

Luego notamos que si $(h^*, \tau^*) \notin Q$, entonces $\mathcal{B}$ es un adversario valido para $\mathsf{Exp}^{\mathrm{MAC}}$ respecto a Π, dado que retorna una nueva falsificación. Por lo tanto,

$$\Pr[\mathsf{Ver}(k, h^*, \tau^*) \Rightarrow 1 \mid (h^*, \tau^*) \notin Q] = \Pr[\mathsf{Exp}^{\mathrm{MAC}}(\mathcal{B}) \Rightarrow 1] = \mathsf{Adv}(\mathsf{Exp}^{\mathrm{MAC}}, \mathcal{B}).$$

De otro modo, supongamos que $(h^*, \tau^*) \in Q$. Por suposición de que $\mathcal{A}$ juega $\mathsf{Exp}^{\mathrm{MAC}}$ contra Π', sabemos que retorna una nueva falsificación $(m^*, \tau^*) \notin S = \{(m_i, \tau_i)\}_i$. Aplicando H a los mensajes retornados y "quieriados" por $\mathcal{A}$, sigue que

$$(H(m^*), \tau^*) = (h^*, \tau^*) \in Q = \{(H(m_i), \tau_i)\}_i \implies \exists m_i \neq m^* \text{ tal que } H(m_i) = H(m^*).$$

Lo que quiere decir que $\mathcal{B}$ encuentra una colisión en H, usando m^*. $\implies \Pr[(h^*, \tau^*) \in Q] \leq \Pr[\mathsf{coll}]$. Juntando las tres desigualdades,

$$\begin{aligned}\mathsf{Adv}(\mathsf{Exp}^{\mathrm{MAC}}, \mathcal{A}) &\leq \Pr[(h^*, \tau^*) \in Q] + \Pr[\mathsf{Ver}(k, h^*, \tau^*) \Rightarrow 1 \mid (h^*, \tau^*) \notin Q] \\ &\leq \Pr[\mathsf{coll}] + \mathsf{Adv}(\mathsf{Exp}^{\mathrm{MAC}}, \mathcal{B}).\end{aligned}$$

Por inspección de $\mathcal{A}$ y $\mathcal{B}$, $t' \approx t$, $q' = q$ y $\varepsilon' = \varepsilon + \Pr[\mathsf{coll}]$.

□

Comentario 35 *Requerir un MAC y un hash construidos de manera independiente puede ser costoso en hardware. Construcciones tipo HMAC se especializan en construir hash-then-MAC compactos a partir de funciones de hash de tipo Merkle-Damgard.*

6.1 Side quest: hashing de contraseñas

Imaginen tener que desarrollar un sistema de login para una pagina web.

Idealmente: quieren investigar como se usan las librerías de "2-factor autentication", y protocolos tipo FIDO.

Tradicionalmente, se usan contraseñas. Los siguientes son dos escenarios comunes e inseguros.

- El servidor guarda una base de datos con una tabla

usuario	clave
maria@gmail.com	"messi10"
⋮	⋮

  Cual es el problema? Probablemente Maria utiliza esa clave en otras paginas. Si el server viene vulnerado y el DB es leakeado, $\mathcal{A}$ podría obtener acceso a cuentas mas importantes.

- (Solución que se encuentra en muchos, pésimos, tutoriales):

usuario	clave
jorge@gmail.com	H("03/14/1994")
⋮	⋮

  SHA-256("03/14/1994") = `ac144fd70bcfcc4a2aa2cd97c4221b82ad03c5fe05c8347fa3e6f29b1e0a36ca`, y H es preimage- y collision-resistant, por lo cual recuperar la clave (o una equivalente) es difícil. Cual es el problema?

  1. H es una función fija
  2. La clave tiene "poca entropia".

  Si $\mathcal{A}$ leakea el DB, puede comparar los hashes con valores de input comunes para los cuales precomputó el output. Tablas de tales valores se llaman comúnmente "rainbow tables". Software tipo "John the Ripper"[6] puede ser usado junto a rainbow tables para recuperar inputs. Si $\mathcal{A}$ conoce H("01/01/1900"), H("02/01/1900"), ..., H("31/12/2024"), va a poder reconocer la clave de Jorge en el DB.

Ahora dos soluciones validas:

[6] https://www.openwall.com/john/

- "salt and hash",

usuario	**salt**	**clave**
alicia@protonmail.com	$s \xleftarrow{\$} \{0,1\}^\ell$	$H(s\|\|\text{“mafalda”})$
mateo@protonmail.com	$s' \xleftarrow{\$} \{0,1\}^\ell$	$H(s'\|\|\text{“dragonball”})$
$\vdots$	$\vdots$	$\vdots$

  Al registrar cada usuario, una nueva cadena aleatoria (salt) $s \xleftarrow{\$} \{0,1\}^\ell$ es generada. El hash es hecho sobre "$s||$clave", y el resultado es guardado junto al salt. De esta manera el login puede ser verificado, pero cada usuario recibe una función de hash "diferente". Es improbable que el adversario haya precomputado $H(s||x)$, para el s especifico del usuario, volviendo el ataque mas improbable (aunque x tenga poca entropía, $s||x$ tiene mucha).
  Nota: esta es una estrategia "folclórica". No hemos demostrado seguridad, en cuanto no tenemos una definición de seguridad. Formalizaciones y demostraciones sobre este escenario son muy recientes (Farshim & Tessaro [FT21], EUROCRYPT 2021).

- Aun mejor: "salt and hash", donde el hash es una función diseñada para guardar contraseñas: Argon2 (IETF RFC9106).

Comentario 36 *No hay salt and hash bueno lo suficiente para protegerte del usar una clave previsible como "dragonball". De otra parte, las soluciones de password managing pueden ser complicadas para el usuario común. Por esa razón, soluciones "multi-factor" son recomendables.*