

Clase 10: seguridad informal y funciones de hash

Fernando Virdia, versión: 0.1, mayo 2025

5 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 18 (Seguridad de funciones de hash, informal). *Una función $H: \{0, 1\}^* \rightarrow \{0, 1\}^\ell$ se dice de “hash criptografico” si satisface las siguientes propiedades:*

- *Puede ser implementada con un algoritmo determinista eficiente.*
- *“Preimage resistance”:* dado $h \in H(\{0, 1\}^*) \subset \{0, 1\}^\ell$, 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 33. *Por que definiciones informales? La propiedad generalmente requerida de una función de hash es la de collision-resistance. En la practica, las funciones de hash son fijas, no requiriendo ninguna clave. Para una tal funcion, existe un adversario eficiente que recupera con probabilidad maxima una colisión!*

- *Dado que $|\{0, 1\}^*| > |\{0, 1\}^\ell|$, colisiones existen necesariamente. Como H es una función fija, supongamos que x, x' sean una colisión. $\boxed{\mathcal{A} := \text{return } (x, x')}$ es un algoritmo eficiente que retorna una colisión con probabilidad 1. Simplemente, no lo conocemos.*

Es posible obtener definiciones formales de (second-)preimage resistance parametrizando x' y h como variables aleatorias. De todos modos, en la practica las funciones de hash a menudo son utilizadas en entornos de baja entropia, por ejemplo para processar contraseñas. Esto limita la seguridad teorica otorgable dado que generalmente las contraseñas son muestradas de distribuciones con muy baja entropia.

Sin definiciones formales, demostrar seguridad de construcciones que las usan no es posible de manera estrictamente formal. Estrategias comunes incluyen:

- Usar el “modelo del oráculo aleatorio” (ROM, “random oracle model”), donde suponemos que H es una función completamente aleatoria.
- Suponer que la funcion de hash toma en input una clave (publica), aunque en realidad no es así.
- Construir un adversario generico (respecto a H) que encuentra colisiones como parte de la demostración, y suponer que en la practica este adversario efectivamente no contiene suficiente informacion sobre H para resultar en colisiones.

Lemma 10 (informal). *Sea H una función de hash.*

1. *Collision resistance $\Rightarrow$ second-preimage resistance.*
2. *Second-preimage resistance $\Rightarrow$ preimage resistance.*
3. *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 x' una second-preimage, $x' \neq x$. Por lo tanto podemos retornar la 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 por lo tanto x' es una second preimage.
3. Aunque collision-resistance $\implies$ second-preimage resistance, y esta $\implies$ preimage resistance, es posible construir funciones que son collision resistant pero para las cuales existe un adversario que rompe preimage resistance sobre un subconjunto bien definido del espacio de input.

Sea G una función de hash CR. Definimos $H(x) := \begin{cases} 0||x & \text{si } |x| = n, \\ 1||G(x) & \text{si } |x| \neq n. \end{cases}$

Por inspección, H es CR dado que colisiones donde $H(x) = 0 \dots$ son imposibles y colisiones con $H(x) = 1 \dots$ implican colisiones en G . Sin embargo, si $h = H(x) = 0 \dots$, x puede ser recuperado dado que $h = 0||x$.

□

Una construcción clásica: Merkle-Damgard.

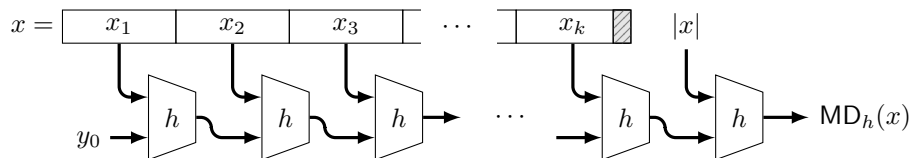


Figure 7: Una manera común de construir funciones de hash es usando PRPs o PRFs iteradas según el diseño de Merkle-Damgard (MD). Imagen tomada de [Ros21, Construction 11.2], usada con permiso del autor.

En la practica, es comun que funciones de hash sean construidas a partir de permutaciones de funciones sobre conjuntos de input finito. Por ejemplo, un paradigma comun de construcción es el de Merkle-Damgard (Figure 7), donde una “funcion de compresión” h comprime iteradametne bloques de bits. Es posible demostrar que si h es collision-resistant, así mismo lo es H .

5.1 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 authentication”, 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 leakado, $\mathcal{A}$ podría obtener acceso a cuentas mas importantes.

- Solución que se encuentra en demaciados tutoriales:

usuario	clave
jorge@gmail.com	$H(\text{“03/14/1994”})$
⋮	⋮

$\text{SHA-256}(\text{“03/14/1994”}) = \text{ac144fd70bcfcc4} \dots \text{c83b1e0a36ca}$, 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 entropía”.

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 $H(\text{“01/01/1900”})$, $H(\text{“02/01/1900”})$, $\dots$, $H(\text{“31/12/2024”})$, va a poder reconocer la clave de Jorge en el DB.

Ahora una solución valida:

- “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 “ $s||\text{clave}$ ”, 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 específico 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 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 34. *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.*

⁶<https://www.openwall.com/john/>