NOMBRE: Lucas Aguilera Gómez SIGLA: IIC3253

N° ALUMNO: 20643039

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC3253 — Criptografía y seguridad computacional — 1' 2025

Tarea 2 – Respuesta Pregunta 1

(a) EstablecerClave(1^n) falla en dos casos, se vera la probabilidad de que falle en ambos casos. Como sabemos que no pueden ocurrir los dos fallos al mismo tiempo, sabemos que la probabilidad de que ocurran estos dos casos es: $P(A \cup B) = P(A) + P(B)$, entonces si demostramos que cada fallo tiene probabilidad despreciable (menor a una función despreciable), entonces la unión también será despreciable.

- Fallo 1: No se encuentra ningún par de indices (i,j) para los cuales los hashes calculados a_i y b_j sean iguales. Por tanto $I = \varnothing$.

  Sabemos que si usuario A y B escogieran al menos una palabra igual, entonces $I \neq \varnothing$. Esto significa que el caso opuesto (la contra-positiva) sería: Si $I = \varnothing$, entonces A y B no escogieron ninguna palabra igual.

  Para simplificar lenguaje, definamos que A elige $U = \{u_1, ..., u_m\}$ del conjunto P, y B elige $V = \{v_1, ..., v_m\}$ del conjunto P.

  Ojo, la contra-positiva nos demostró que $U \cap V = \varnothing$ siempre debe ocurrir cuando $I = \varnothing$, por tanto, $P(I = \varnothing) \leq P(U \cap V = \varnothing)$. (Nota: podemos fijarnos que la $P(I = \varnothing)$ es $P((U \cap V = \varnothing)$ **y (que no ocurra colisión entre las palabras diferentes)**) y esto demuestra lo dicho anteriormente).

  Ahora intentemos calcular $P(U \cap V = \varnothing)$.

  - Sabemos que P es de tamaño $Np = n^2$.
  - Sabemos que A elige m palabras distintas de P. $\binom{Np}{m}$ formas de hacerlo.
  - Sabemos que B elige m palabras distintas de P. $\binom{Np}{m}$ formas de hacerlo.

  La probabilidad de que B no saque ningún elemento igual que A, es la de que, una vez elegido U, B solo pueda elegir sus m palabras del conjunto $P \setminus U$, el cual tiene Np - m palabras.
  Entonces tenemos: $P(U \cap V = \varnothing) = \binom{Np-m}{m} / \binom{Np}{m}$. Esto se simplifica a:

$$\frac{\binom{Np-m}{m}}{\binom{Np}{m}} = \frac{\frac{(Np-m)!}{m!\,(Np-2m)!}}{\frac{(Np)!}{m!\,(Np-m)!}} = \frac{(Np-m)! \cdot (Np-m)!}{(Np-2m)! \cdot (Np)!}$$

  Simplificando lo anterior, tenemos:

$$= \frac{(Np-m)(Np-m-1)\cdots(Np-2m+1)}{Np(Np-1)\cdots(Np-m+1)}$$

Luego, lo anterior se puede escribir como:

$$P(U \cap V = \emptyset) = \prod_{s=0}^{m-1} \frac{Np - m - s}{Np - s} = \prod_{s=0}^{m-1} \left(1 - \frac{m}{Np - s}\right).$$

Ahora, es obvio que $\frac{m}{Np-s} \geq \frac{m}{Np}$ y por tanto: $1 - \frac{m}{Np-s} \leq 1 - \frac{m}{Np}$, esto significa que: $P(U \cap V = \emptyset) \leq (1 - \frac{m}{Np})^m$.
Recordemos que la serie de Taylor para ln(1-x) es:

$$\ln(1-x) = -\sum_{n=1}^{\infty} \frac{x^n}{n} \quad \text{para } |x| < 1$$

$$\ln(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \frac{x^4}{4} - \cdots$$

Esto significa que para x en (0,1):

$$k \cdot \ln(1-x) \leq -xk$$
$$\ln\left((1-x)^k\right) \leq \ln\left(e^{-xk}\right)$$
$$(1-x)^k \leq e^{-xk}$$

Esta inecuación se puede aplicar a $(1 - \frac{m}{Np})^m$:

$$P(U \cap V = \emptyset) \leq \exp\left(-\frac{\lceil \log n \rceil}{n} \cdot n\lceil \log n \rceil\right) = \exp\left(-\left(\lceil \log n \rceil\right)^2\right)$$

Ahora demostremos que: $f(n) = \exp\left(-(\lceil \log n \rceil)^2\right)$ es despreciable.
Queremos probar que f es despreciable, es decir:

$$\forall \, \text{polinomio } p(n) \in \mathbb{N}[n],\ \exists\, n_0 \in \mathbb{N},\ \forall\, n \geq n_0,\ f(n) < \frac{1}{p(n)}$$

Imaginemos entonces un polinomio $p(n) = n^c$ que para cualquier constante positiva c, $f(n) < n^{-c}$ cuando n es suficientemente grande ($n \geq n_0$).
Esto es equivalente a demostrar que:

$$\exp\left(-\left(\lceil \log n \rceil\right)^2\right) < \frac{1}{n^c}$$

Tomando logaritmos naturales en ambos lados:

$$-\left(\lceil \log n \rceil\right)^2 < -c \ln n$$

Multiplicando por -1:

$$\left(\lceil \log n \rceil\right)^2 > c \ln n$$

Como $\lceil \log n \rceil \geq \log n = \frac{\ln n}{\ln 2}$, por tanto $(\lceil \log n \rceil)^2 \geq (\frac{\ln n}{\ln 2})^2$, luego para n suficientemente grande notemos que $(\frac{\ln n}{\ln 2})^2 > c \ln n$, esto se prueba al dividir por ln: $(\ln n) > c(\ln 2)^2$. Por tanto, siempre va a existir un $n \geq n_0$, tal que se cumpla lo anterior y por tanto, $f(n) = \exp\left(-(\lceil \log n \rceil)^2\right)$ es despreciable, lo que hace que $P(Fallo1) = P(I = \varnothing) \leq P(U \cap V = \emptyset) \leq f(n)$ también sea despreciable.

- Fallo 2: Se encuentra un par de indices (k, l) (el menor par según orden lexicográfico del enunciado) tal que $a_k = b_l$, pero $x_k \neq y_l$.

  Notemos que este fallo curre solo en el caso donde ($I \neq \varnothing$) y además, hay al menos una colisión indebida, esto significa que dos palabras distintas dieron hashes iguales. Esto en términos de n es; $a_k = b_l$ y $x_k \neq y_l$ por tanto, $h_n(s||x_k) = h_n(s||y_l)$ esto es por definición, una colisión para la función h_n.

  Ahora bien, sabemos que $\{h_n\}$ es resistente a colisiones, esto significa que es computacionalmente inviable para cualquier algoritmo que se ejecute en tiempo razonable (dependiente de un tiempo polinomial en n) el encontrar una colisión.

  Luego, como establecerClave actúa como un algoritmo, el cual sigue una serie de pasos donde cada paso se ejecuta en tiempo polinomial en n, entonces la probabilidad de que establecerClave (siendo un algoritmo de tiempo polinomial en n) encuentre una colisión es despreciable, por la misma definición de la resistencia a colisiones, y por tanto, $P(fallo2)$ es despreciable.

Finalmente, podemos decir que $P(fallo1 \cup fallo2) = P(fallo1) + P(fallo2)$, y por lo tanto, por propiedades de funciones despreciables, sabemos que la suma de dos funciones despreciables es despreciable. Por tanto, $P(fallo1 \cup fallo2)$ es despreciable.

(b) Analicemos cada paso del algoritmo y la complejidad asociada de cada uno.

- Etapa 1.0: A escoge $s \in \{0,1\}^n$ y se lo manda a B. O(n).
- Etapa 1.1: Tanto A como B escogen $m \approx n \log n$. Asumiendo que escoger implica "crear" o "copiar" la palabra de largo n, entones esto toma O(n) tiempo por palabra. Como A y B escogen m palabras en total, tenemos que O(2nm) = O(nm).
- Etapa 1.2: A y B tienen que calcular el hash de cada palabra, esto tarda exactamente O(n). En total A tiene que transformar m palabras, por tanto, la complejidad de este proceso es O(nm). Lo mismo sucede con B.
- Etapa 1.3; El costo de "enviar" y "recibir" tuplas de números y hash, deberían rondar O(nm) (la cantidad de letras multiplicado por su largo en bits), lo cual no cambia el resultado que llevamos hasta el momento.
- Vieja Etapa 2: La etapa donde A y B comparan sus hashes guardados con los enviados. Acá se presenta un problema: si A y B hacen una comparación directa entre listas, terminamos con una complejidad $O(nm^2) = O(n^3 \log^2 n)$. Dado que esto no es lo que queremos, necesitamos que se manejen las listas de otra forma.
- Nueva Etapa 2, Veamos el caso de A pues B es homologo:
  - A recibe la lista de B: $L_B = [(1, b_1), (2, b_2), (3, b_3), ..., (m, b_m)]$.
  - A ordena la lista recibida L_B con sus elementos (j, b_j). Esta ordenación primero ordena las tuplas en orden lexicográfico por el valor del hash b_j, sin embargo, en caso de que experimentemos dos hash iguales, se ordenan por indice j. Merge Sort es un algoritmo de ordenación eficiente que realiza este ordenamiento en O(m log m) (ademas, merge sort es estable, por tanto facilita el mantener el segundo ordenamiento por indice). El coste de ordenación sería: $O(nm \log m)$ (se añade factor n, por el costo de cada comparación de n bits). Sustituyendo m = $O(n \log n)$: Costo = $O(n(n \log n) \log(n \log n)) = O(n(n \log n)(\log n + \log(\log n))) = O(n^2 \log^2 n + n^2 \log n \log(\log n)) = O(n^2 \log^2 n)$, puesto que el primer termino domina al segundo.
  - A busca el par (k, l): Para cada a_k realiza una binary search en el L_B ordenado. Las búsquedas binarias tienen un coste de O(log m) comparaciones. Cada comparación (entre a_k y un b_j de L_B ordenado) compara bits a bits, por tanto es O(n). Esto quiere decir, que una búsqueda binaria de un solo elemento cuesta O(n log m). Dado el ordenamiento de L_B y el orden predefinido de L_A, entonces el primer par encontrado será el aceptado. Notemos finalmente,

que en el peor caso posible, A debe realizar un búsqueda binaria por todos sus elementos: m. Por tanto el coste sería: $O(mn \log m) = O(n^2 \log^2 n)$, como vimos anteriormente.

- A selecciona la llave encontrada, esto debería tomar O(n) para leer la palabra de n bits.
- B hace lo mismo que A, pero esto no afecta el coste total (constantes son ignoradas).

Notemos que la peor complejidad es $O(n^2 \log^2 n)$, por tanto es el termino dominante en el tiempo total que se tarda en establecer una clave.

(c) A partir del algoritmo A con las propiedades dadas, construiremos un A_{pf} (algoritmo que no cumpla puzzle friendly).

A_{pf} toma como entrada (u, v) donde $u \in \{0,1\}^n$ y $v \in \{0,1\}^n$. Recordemos que por enunciado la probabilidad despreciable sería la de solucionar el puzzle dado un u al azar y un v que maximice la probabilidad (es decir, el mejor caso), luego A_{pf} debe encontrar $x \in \{0,1\}^n$ tal que $h_n(u||x) = v$.

Esto hace A_{pf}:

(a) Dividir el espacio de búsqueda de x: Sabemos que se busca un x que tiene tamaño n. Por tanto, podemos decir que $\lceil 2^n/n^2 \rceil$ es una división aproximada de bloques (cada uno de tamaño n^2), donde cada bloque tiene palabras de tamaño n (para conveniencia, ordenados de manera lexicográfica). Puede que el ultimo bloque sea $\leq n^2$. Como ahora tenemos estos bloques $B_1 ... B_{N_B}$, podemos identificar sus limites inferiores y superiores, es decir, para bloque B_i van a existir u_{1i} y u_{2i}, donde todos los elementos del bloque cumplen que $u_{1i} \leq b_{ii} \leq u_{2i}$.

(b) Iterar sobre los bloques: Para i desde 1 hasta N_B Sea el bloque actual $B_i = [u_{1i}, u_{2i}]$. Por cada bloque, llamar a $A(s = u, u_1 = u_{1i}, u_2 = u_{2i}, v = v)$. Si A retorna $x \neq \perp$, por la definición de A sabemos que encontró solución $\in B_i$ tal que $h_n(u||x = v)$, en caso de que esto pase, A_{pf} se detiene y retorna x. Por otro lado, si A retorna $x = \perp$ hasta el final, sabemos que no se encontró solución alguna, por tanto A_{pf} retorna $\perp$, y termina.

De A_{pf} anterior hay un caso borde: El ultimo B_i puede ser de tamaño menor que n^2 (mencionado en paso a). Podemos solucionar esto llenando B_{N_B} con las palabras binarias que se encuentran lexicograficamente menores al menor elemento de B_{N_B} hasta que este cumpla el tamaño adecuado. Este caso borde debería añadir una complejidad (en el peor de los casos) de $O(n^2)$, como veremos a continuación, esto no aumenta la complejidad total del problema.

Analizando complejidad:

- Crear todas las posibles x ordenadas tarda: $O(n2^n)$ (si consideramos que crear una letra de tamaño n tarda n).
- Por lógica sabemos que dividirlas en bloques debería tardar menos. Lo mismo ocurre para encontrar limites.
- Luego para los N_B bloques sabemos que existen aproximadamente $N_B = 2^n/n^2$. Y ademas sabemos que cada bloque llama a A que cuesta $O(n^3)$. Por tanto, el tiempo total de la iteración es de $O(2^n n^3/n^2) = O(n2^n)$.
- En total, la complejidad es $O(n2^n)$ esto significa que la complejidad del algoritmo es igual a la establecida en el enunciado (es competidor de romper puzzle friendly), en otras palabras, ahora solo necesitamos demostrar que este algoritmo NO genera una probabilidad despreciable.

Analizando probabilidad de éxito:

- Imaginemos que $A_{pf}(u, v)$ soluciona el puzzle. En este caso, sabemos que la solución va a pertenecer a algún bloque B_i. Luego, como A_{pf} procesa todos los bloques usando A, al procesar B_i el algoritmo A tiene garantizado encontrar la solución.

- En el caso opuesto, si A_{pf} no encuentra solución, esto significa que se procesaron todos los bloques posibles y A no encontró solución en ninguno (esto también es considerado "solución" según el enunciado).
- Los puntos anteriores nos llevan a la siguiente conclusión: $A_{pf}(u, v)$ encuentra solución ssi el puzzle (u, v) tiene solución.
- La probabilidad que se espera despreciable es: $\max_{v \in \{0,1\}^n} \Pr_{u \sim \mathcal{U}(\{0,1\}^n)} [A_{\text{pf}}$ soluciona el puzzle $(u, v)]$ pero demostramos que esto es igual a: $\max_{v \in \{0,1\}^n} \Pr_{u \sim \mathcal{U}(\{0,1\}^n)}$ [El puzzle (u, v) tiene solución].
- Pero notemos que el enunciado considera "resolver" el problema como: "A soluciona el puzzle (u,v) si $A(u, v) \in \{0, 1\}^n$ y $h(u||A(u, v)) = v$ en caso de que el puzzle tenga solución, y $A(u, v) = \perp$ en caso de que el puzzle (u,v) no tenga solución". Pero notamos que, si ese es el caso, entonces nuestro algoritmo A_{pf} es de probabilidad 1, pues siempre identifica un x' especifico que sirva, o en caso de no existir uno, retorna $\perp$ y por tanto igual resuelve el problema.

Como sabemos que 1 no es despreciable, y es una constante, entonces se demostró por contradicción que existiría A_{pf} que rompe la definición de puzzle-friendly, y por tanto, A no puede existir.