Hashing

Today

- Hashing Definition
- Desirable Properties
  - One-Way
  - Collision Resistant
- Finding Collisions
  - Birthday Attack
  - Floyd’s Two-Finger Algorithm
- Inverting $H$
  - Rainbow Tables

Definition

- a hash function $H$ maps a universe $U$ to a finite set $S$
- more concretely: $H : \{0, 1\}^* \rightarrow \{0, 1\}^\lambda$

Some Desirable Properties (more to come next lecture)

The definition is extremely loose. For example, a function that just truncates or is constant is technically a ‘valid’ hash function. Thus, we define some desirable properties. Each use case of hash functions will require a certain subset of these criteria.

- One-Way (non-invertible)
  - $x \leftarrow U, y = H(x)$
  - given $y$, infeasible to find $x'$ s.t. $H(x') = y$
  - necessary for password storage
- Collision Resistant
  - difficult to find $x \neq x'$ s.t. $H(x) = H(x')$
  - necessary for hash tables, Bitcoin (digital signatures)
- There are more! Save for lecture on Monday

Finding Collisions

- Goal: break CR of $H$ with $x \neq x'$, s.t. $H(x) = H(x')$
- Idea 1: store random $(x, H(x))$ pairs until two collide

Birthday Attack

- try random pairs until one collides, or you run out of resources
- succeeds with a relatively high constant probability in $O(\sqrt{|S|})$ time and memory (since you are checking $\Theta(n^2)$ pairs), but this is prohibitively large for $|S| \geq$ say $2^{128}$.
- see Katz and Lindell Lemma 10.2 for proof.
• Idea 2: treat repeated applications of $H : S \rightarrow S$ as a directed graph, look for a cycle. Once found, last element on tail = $x$, last element on cycle = $x'$

• How do we know cycles exist? If we assume $H$ is a random oracle (to be covered next lecture), then we can expect to “loop back” to some previously visited node after $\approx \sqrt{|S|}$ traversals (same intuition as birthday attack). Then, with probability $\approx 1 - \frac{1}{\sqrt{|S|}}$ (very close to 1), we loop back to a node that is not the first, and there is a tail of length $> 0$. Now let’s see how to use this...

Floyd’s Two-Finger Cycle Detection Algorithm

• We set two pointers $a, b$ to a random node $x$
• We then advance $b$ twice as fast as $a$ until they meet again
  – Set $a = H(a)$, $b = H(H(b))$ until $a = b$

Informal Proof
  – If $a$ and $b$ begin on a node which leads to a cycle, they will eventually meet.
  * More formally: Thm: let $x$ be a node on a tail of length $t$ to a cycle of length $n$. Then after $i$ iterations, $i \geq t$, the position of $a$ and $b$ are as follows:
    \begin{align*}
    a &= x_{(i-t) \mod n} \\
    b &= x_{(2i-t) \mod n}
    \end{align*}
  * Note that $\forall i \geq t$ s.t. $i$ is a multiple of $n$, $a = b = x - t \mod n$
  * ; after $\max(t + (-t \mod n), n)$ iterations, $a$ and $b$ will meet at node $x - t \mod n$

• Suppose $a = b = x'$ after $d$ iterations (we detected a cycle). How do we use this to find a collision?
  – We know $x' = x - t \mod n$
  – Set $a = x = x - t, b = x - t \mod n$, step each one edge at a time, remembering last element visited for each
  – After $t$ steps, $a$ and $b$ will meet at $x_0$. Return $x_{-1}, x_{-1} \mod n$ as colliding pre-images

Analysis
  – Time:
    * Phase 1: $3 \max(t + (-t \mod n), n)$ hashes
    * Phase 2: $2t$ hashes
    * Overall: $\Theta(n + t)$ hashes
  – Memory:
    * 4 pointers, $O(1)$

Inverting Hash Functions

Rainbow Tables

• Goal: create a space/time tradeoff by storing head and tail of hash chains of length $k$
• First attempt:
  – Precomputation: assume we want to store hashes of $n$ pre-images
    * choose $\frac{n}{k}$ random pre-images $x_i$
    * store $(x_i, H^{(k)}(x_i))$ for each $x_i$
  – Query: target hash $y$, want to find $x$ s.t. $H(x) = y$
    * let $y_i = H^{(i)}(y)$
    * compute $y_i$ for $i \in \{1 \ldots k\}$
• check if any \( y_i \) equals tail of any chain
  
  - if so, start at head of chain, hash until \( y \) reached, last pre-image inverts \( y \)

• Problem: only works for pre-images that are also images of \( H \), but most passwords people use don’t look like pseudorandom bits
  
  - Instead, create a reduction function \( R \) which maps images of \( H \) back into a target set \( P \), i.e. 10 letters followed by 2 digits
  
  - example of \( R \): treat input as 10 base 26 digits followed by 2 base 10 digits, and truncate the rest

• Modified Algorithm:
  
  - Precomputation:
    * choose \( \frac{n}{k} \) random pre-images \( p_i \in P \)
    * chain function is now \( C = R \circ H \)
    * store \( (p_i, C^{(k)}(p_i)) \) for each \( p_i \)
  
  - Query: target hash \( y \), want to find \( p \in P \) s.t. \( H(p) = y \)
    * compute \( C^{(i)}(R(y)) \) for \( i \in [1, k] \)
    * proceed same as first version, but we risk false positives since \( R \) maps to a smaller set \( P \)
    * i.e. even if \( C^{(i)}(p) = R(y) \), it is possible that \( H(p) \neq y \), in which case we just skip this false positive and continue searching

• Analysis for querying \( n \) preimages:
  
  - Time:
    * Precomputation: \( \Theta(n) \)
    * Query: \( O(k) \)
  
  - Memory: \( \Theta(\frac{n}{k}) \)

• Combating Rainbow Tables:
  
  - Salt your passwords! Storing \( H(p)||r \) where \( r \) is a long random bit string makes precomputing a rainbow table infeasible