1 Hash Functions

Properties of hash functions:
- Pre-Image Resistant (PR)
- Collision Resistant (CR)
- Weak Collision Resistance/Second Pre-Image Resistance (WCR/2nd PR)

1.1 Pre-Image Resistant

The ideal hash function: It is a uniform probability distribution that, for any “ball” thrown, it is equally likely for that ball to be thrown into any “bin”

\[ E(\text{number of pre-images}) = f(n) \rightarrow \text{If you throw n balls, how many will end up in a specified bin?} \]

So:

\[ E(\text{number of pre-images}) = f(n) = \sum_{n} \frac{1}{2^d} \]

1 = \frac{n}{2^d}

Note that in the above equation: \( d \) is the output of the ahash in bits, \( 2^d \) is the output space, \( \frac{1}{2^d} \) is the probability that a ball ends up in a specific bin, and \( \sum_n \) is the number of balls thrown (i.e. the amount of work done, which is \( O(n) \))

It is NIST recommended that \( d = 112 \), however we typically use \( d = 128 \) as it is a nicer number.

1.2 Collision Resistance

It is infeasible to find two values \( x_1 \) and \( x_2 \) such that \( H(x_1) = H(x_2) \) (i.e. two values whose hashes give the same value)
Figure 2: A representation of Collision Resistance and a collision

\[
E(\text{number of collisions}) = f(n) \\
= \binom{n}{2} \Pr(H(x_i) = H(x_j)) \rightarrow \Pr(2 \text{ balls land in the same bin}) \\
= \binom{n}{2} \cdot \frac{1}{2^d} \\
= 2^{-d}(n - 1)n \\
= 2^{-d}n^2 \\
1 = 2^{-d}n^2 \\
n^2 = 2^d \\
n = \sqrt{2^d} \\
n = 2^{d/2} \rightarrow n = d^128 = 2^d/2 : d = 256
\]

To find expectation 1 (i.e. to find either one pre-image or one collision):

\[
\begin{array}{c|c|c}
\text{Work (balls thrown)} & \text{Pre-image} & \text{Collision} \\
\hline
\text{d} & \theta(2^d) & \theta(2^{d/2}) = \theta(\sqrt{2^d}) \\
\end{array}
\]

1.3 Second Pre-Image Resistance/Weak Collision Resistance

It is infeasible, given some value of \( x \), to find a second value, \( y \), not equal to the first value (\( x \)), such that the hash value of the first equals the hash value of the second (i.e. it is infeasible given \( x = \{0,1\}^* \) to find \( x \neq y \) such that \( H(x) = H(y) \)).

Note that CR \( \rightarrow \) WCR but not WCR \( \rightarrow \) CR. Indeed:

Example:

\[
H'(x) = \begin{cases} 
0||x, & |x| = n \\
1||H'(x), & \text{otherwise}
\end{cases}
\]
This gives us $y = H(x)$ and $|y| = 2^n$, as well as $y' = H'(x)$ with $|y'| = 2^{n+1}$. It should be noted that this is collision resistant by default (as all inputs and the appending onto these inputs are all unique). However, this is not pre-image resistant as values are merely being appended onto the input.

Example: An algorithm to find collisions, Floyd’s cycle-finding algorithm, is no faster than exhaustive search in that it takes $\theta(2^{d/2})$ time, however, it is better in the sense that it requires less memory.

It should be noted that:

1. This algorithm can’t go on forever as you would run out of $x$ values (there are $2^d$ values)
2. This algorithm can’t terminate (as you can always take the hash of a value)

Intuition of Floyd’s cycle-finding algorithm (i.e. to find a collision):
2 registers:

1. Start at $x_1$, move sequentially
2. Start at $x_1$, move twice as fast

Eventually both registers will reach the “same spot” $\varphi$. It should be noted that $\varphi$’s distance is half the distance of going through the cycle. From this point, you do the following:
2 registers:

1. Start at $x_1$, move sequentially
2. Start at $x_1$, move twice as fast
1. Start at $x_1$, move sequentially
2. Start at $\phi$, move sequentially

This time, when both registers reach the “same spot”, they will have found a collision.

![Figure 6: Finding the “same spot” (phi)](image)

Example: Password Storage
Alice sends Google her password ($pwd$), Google has the following options for storage:

- Store $pwd$ (cleartext) → horrible
- $H(pwd) \rightarrow$ better
- $H(pwd + salt) \rightarrow$ even better

Note that for this type of case (password storage), the PR property is required. However, you don’t really need CR as it doesn’t give you anything extra and doesn’t contribute anything important. However, some attacks are still possible. An example would be that, given $H(x)$, you could try every possible password to reproduce $H(x)$ (i.e. a brute-force attack). This type of attack is strengthen resistant but is note prevent complete (i.e. you can strengthen against this type of attack and make it very costly and highly unlikely that an attacker will find the original password, but you can’t prevent it entirely).

A way to make these brute-force attacks more feasible are the use of rainbow tables (a lookup table of password values and hashes of those password values (i.e. $H(x)$ where $H$ is a common hash function)).

A way to help prevent this is using a salt when password hashing: $H(pwd||salt)$ where the salt is a random value that is not secret.

Another way to help prevent these types of attacks is iterative hashing which is where you hash repeatedly, making it a lot of work for the attacker to gain the original password (i.e. $H^{1000}(pwd||salt)$).
Example: Signed Messages

**Later we will talk about digital signatures, which is secure due to CR**

Signed messages use the “hash then sign” paradigm - you have a message \( m \) which you hash (in order to get it down to a good size) and then sign it with a signature \( S = \text{sig}(H(m)) \)

Example: Signed Messages

**Later we will talk about digital signatures, which is secure due to CR**

Signed messages use the “hash then sign” paradigm - you have a message \( m \) which you hash (in order to get it down to a good size) and then sign it with a signature \( S = \text{sig}(H(m)) \)

You can attack this by coming up with two values:

\[
H(m) = H(m') \begin{cases} m, & \text{Looks like a valid value} \\ m', & \text{The value that you want} \end{cases}
\]

However, it should be noted that you need to find valid, meaningful collisions (i.e. not just random noise that hashes to the correct value).
Example: Hash Tree (Merkle Tree) → Used with bit coin
This is a space saving storage mechanism that also has security properties.

\[ y \rightarrow 256 \text{ bits} \]

\[
\begin{array}{c}
H_3() \\
H_1() \\
\quad x_1 \quad x_2 \\
H_2() \\
\quad x_3 \quad x_4
\end{array}
\]

\[
\begin{array}{c}
H_3() \\
\quad x_3 \quad x_4'
\end{array}
\]

\[ H_2() \]

Same value means a collision

Figure 8: Merkle Tree visualization

So: You are sent \( y \) (the root) and you are unable to change any \( x_i \) value now (as it will alter the value \( y \) to another value \( y' \)) without finding a collision.

Notes on common hash functions:

<table>
<thead>
<tr>
<th>Common hash functions</th>
<th>Output Size</th>
<th>Collision Resistance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD5</td>
<td>128</td>
<td>56</td>
</tr>
<tr>
<td>SHA1</td>
<td>160</td>
<td>80</td>
</tr>
<tr>
<td>SHA2 (SHA256/SHA512)</td>
<td>256/512</td>
<td>128/256</td>
</tr>
<tr>
<td>SHA3</td>
<td>256/512</td>
<td>128/256</td>
</tr>
</tbody>
</table>

NIST recomends anything SHA1 (i.e. SHA2 (SHA256/SHA512) and SHA3) as it provides at least 212 bit security.