CMSC 28400 Introduction to Cryptography **Autumn 2021**

Notes #12: Collision Finding

Instructor: David Cash

These notes cover the latter part of Lecture 14. For definitions of collision resistance and constructions of hash functions, please refer to the slides.

12.1 Birthday Attack

Let's look at the simplest attack that comes to mind when trying to find a collision: Just try computing $H(k, x)$ for different values of x until you find two inputs that collide. A bit more precisely, the attack would look like this:

```
Alg $\mathcal{A}(k)$
Initialize a hash table $Y$
For $x = 1, \ldots, q$ :
    $y \leftarrow H(k, x)$
    If $Y[y] \neq \perp$:
        $x' \leftarrow Y[y]$
        Output $x, x'$
    $Y[y] \leftarrow x$
Output "fail"
```

For what value of q do we expect this attack to generate a collision with good probability? Certainly $q = |\mathcal{R}| + 1$ is enough, since by the pigeonhole principle we will get a repeated value by then. But in practice it turns out that this attack performs much better, finding a collision when q is on the order of the *square root* of $|\mathcal{R}|$. So for $|\mathcal{R}| = 2^{128}$, we could set $q = \sqrt{|\mathcal{R}|} = 2^{64}$, which is feasible for some adversaries, while $q = 2^{128}$ is not. It is because of this attack that hash function outputs need to have length $2n$ to have any chance of resisting 2^n-time attacks. (So, for example, resisting 2^{128}-time attacks requires a hash with 256 bit output.) The algorithm is often called a "birthday attack" because it exploits the birthday paradox to succeed surprisingly quickly.

We now explain why the attack succeeds with good probability with $q \approx \sqrt{|\mathcal{R}|}$. Heuristically, we can think of the q values for y generated in the loop as uniform and independent samples from $\mathcal{R}$. When modeled this way, the algorithm succeeds if any of these q samples are equal. The notes on probability analyzes this event (in the last section), which happens with probability $\mathsf{Col}(|\mathcal{R}|, q)$, a value which satisfies

$$0.3\frac{q(q-1)}{|\mathcal{R}|} \leq \mathsf{Col}(|\mathcal{R}|, q) \leq 0.5\frac{q(q-1)}{|\mathcal{R}|}$$

when $q \leq \sqrt{2|\mathcal{R}|}$. Here we only need the lower bound. Setting $q = \sqrt{|\mathcal{R}|}$ gives

$$\mathsf{Col}(|\mathcal{R}|, q) \geq 0.3\frac{\sqrt{|\mathcal{R}|}(\sqrt{|\mathcal{R}|} - 1)}{|\mathcal{R}|} = 0.3 - 0.3/\sqrt{|\mathcal{R}|} \approx 0.3.$$

For our purposes, this is a very high advantage! And as q goes above $\sqrt{|\mathcal{R}|}$ this probability will get very close to 1.

12.2 Small-Space Collision-Finding

One might object to the previous attack because of the amount of space it consumes. The space required to store the hash table Y will grow with q, and quickly become impractical. For instance, it is possible that adversaries can actually run an attack taking $q = 2^{80}$ hash evaluations, but can't store a hash table with 2^{80} entries.

It turns out that there is a different attack that also finds a collision in about $\sqrt{|\mathcal{R}|}$ time, but runs with very little space. In fact, it only needs to store two hash values and a counter! The algorithm is called *Floyd's tortoise and hare algorithm.* Here's the pseudocode:

```
Alg $\mathcal{A}_{\text{floyd}}(k)$
    $x_{\text{tort}} \leftarrow 0$;  $x_{\text{hare}} \leftarrow 0$
    Do:
        $x_{\text{tort}} \leftarrow H(k, x_{\text{tort}})$
        $x_{\text{hare}} \leftarrow H(k, H(k, x_{\text{hare}}))$
    Until $x_{\text{tort}} = x_{\text{hare}}$
    $x_{\text{tort}} \leftarrow 0$
    While $H(k, x_{\text{tort}}) \neq H(k, x_{\text{hare}})$:
        $x_{\text{tort}} \leftarrow H(k, x_{\text{tort}})$
        $x_{\text{hare}} \leftarrow H(k, x_{\text{hare}})$
    Output $x_{\text{tort}}, x_{\text{hare}}$
```

The algorithm maintains two values $x_{\text{tort}}, x_{\text{hare}} \in \mathcal{R}$. In the first loop, each iteration updates x_{tort} by evaluating H on the current value of x_{tort}; Think of this as a "single step" (it's the slow tortoise). The loop also updates x_{hare} to $H(k, H(k, x_{\text{hare}}))$, which you can think of as a "double step" (it's the fast hare). The loop terminates when the values are equal (a condition which does not obviously happen, but we will argue that it does!).

Once the values are made equal, x_{tort} is reset to the starting value 0, and the second loop now updates both values in single steps until their next hash values are equal. Once this happens, a collision is found, since $x_{\text{tort}} \neq x_{\text{hare}}$ (unless we got very unlucky and had $H(k, 0) = 0$, but if this is true then the hash function is broken for other reasons).

12.2.1 Analyzing Floyd's Algorithm

We will explain why this algorithm works in two steps: First, assuming it works, why it succeeds in time $\approx \sqrt{|\mathcal{R}|}$, and then why it works at all.

Define $a_0 = 0$ and for $i \geq 1$ define $a_i = H(k, a_{i-1})$. Then the values of a_i are the hash values computed by the tortoise as it walks. When the tortoise is at a_i, the hare will be at a_{2i}, since it takes twice as many steps.

By the pigeonhole principle, we know that the a_i must eventually repeat. Once this happens, the values will loop indefinitely. (See Figure 12.1. See also the slides and video for another drawing; The rest of this paragraph is much easier to understand with a picture.) In fact, the sequence $a_0, a_1, \ldots$ must have the following structure: After some initial "tail" $a_0, a_1, \ldots, a_{\ell_{\text{tail}}}$ of $\ell_{\text{tail}} + 1$ distinct values, the subsequent values of a_i loop, repeating the values $a_{\ell_{\text{tail}}+1}, a_{\ell_{\text{tail}}+2}, \ldots, a_{\ell_{\text{tail}}+\ell_{\text{loop}}}$, where ℓ_{loop} is the size of "loop". Here, $a_{\ell_{\text{tail}}} = a_{\ell_{\text{tail}}+\ell_{\text{loop}}} = a_{\ell_{\text{tail}}+2\ell_{\text{loop}}} = \cdots$ as the walk loops indefinitely.

$$\begin{tikzcd}
 & & & a_7 \arrow[dd] & a_6 \arrow[l] & \\
 & & & & & a_5 \arrow[ul] \\
a_0 \arrow[r] & a_1 \arrow[r] & a_2 \arrow[r] & a_3 \arrow[r] & a_4 \arrow[ur] &
\end{tikzcd}$$

Figure 12.1: A visualization of the values computed in Floyd's algorithm. Here, $\ell_{\text{tail}} = 3$ and $\ell_{\text{loop}} = 5$. In the execution of the algorithm, the first loop will exit after 5 iterations, since the hare will catch the tortoise at node $a_5 = a_{10}$. Afterwards the walks from a_0 and a_5 will reach a_2 and a_7 respectively, at which point the algorithm will exit, since they collide at a_3.

We first claim that the total combined size of the tail and loop will be about $\sqrt{|\mathcal{R}|}$, heuristically. This follows by the same reasoning as above: We can think of the values $a_0, a_1, \ldots$ as uniform and independent samples, so we expect a repeated value after about $\sqrt{|\mathcal{R}|}$ steps.

Assuming that a collision happens, we will now bound the running time of the algorithm by showing that the first loop executes at most $\ell_{\text{tail}} + \ell_{\text{loop}}$ times and the second loop executes exactly $\ell_{\text{tail}} - 1$ times. This gives a total run time of $2\ell_{\text{tail}} + \ell_{\text{loop}}$ iterations, which is still on the other of $\sqrt{|\mathcal{R}|}$, as desired.

The main insight is the following claim:

Claim 1. *For any $m \geq \ell_{\text{tail}}$ and any $n \geq 0$, $a_m = a_{m+n}$ if any only if n is a multiple of ℓ_{loop}.*

This claim is true because once the walk has left the tail (i.e. $m \geq \ell_{\text{tail}}$) the only way a repeat can happen is if one makes some number of complete trips around the loop. (A formal proof could use modular arithmetic, but we won't insist on doing this.)

We now prove three small corollaries, and afterwards relate them to the algorithm.

Corollary 1. *For any i, $a_i = a_{2i}$ if and only if i is a multiple of ℓ_{loop} greater than ℓ_{tail}.*

Proof. Suppose first that i is multiple of ℓ_{loop} greater than ℓ_{tail}. Now apply Claim 1 with $m = i$ and $n = i$. Since we have $m \geq \ell_{\text{tail}}$ and n is a positive multiple of ℓ_{loop}, Claim 1 tells us that $a_m = a_{m+n}$. But this is just $a_i = a_{2i}$ since we took $m = i$ and $m + n = 2i$.

Now suppose i satisfies $a_i = a_{2i}$. We must have $i \geq \ell_{\text{tail}}$ since the values on the tail are never repeated. Now apply the other direction of Claim 1, again with $m = i$ and $n = i$. Since $a_m = a_{m+n}$, it must be that n is a multiple of ℓ_{loop}, as desired. □

Corollary 2. *For some $1 \leq i < \ell_{\text{tail}} + \ell_{\text{loop}}$, $a_i = a_{2i}$.*

Proof. Let i the smallest multiple of ℓ_{loop} that is at least ℓ_{tail}. Then $i < \ell_{\text{tail}} + \ell_{\text{loop}}$, since there is always a multiple of ℓ_{tail} in any interval ℓ_{loop} numbers, and in particular in $\{\ell_{\text{tail}}, \ell_{\text{tail}} + 1, \ldots, \ell_{\text{tail}} + \ell_{\text{loop}} - 1\}$. Finally, since our chosen i is a multiple of ℓ_{loop} greater than ℓ_{tail}, the previous corollary implies $a_i = a_{2i}$.

□

Corollary 3. *If i is a multiple of ℓ_{loop}, then $a_{i+\ell_{\text{tail}}} = a_{\ell_{\text{tail}}}$.*

Proof. Apply the "$\Longleftarrow$" direction of Claim 1 with $m = \ell_{\text{tail}}$ and $n = i$. □

Now let's wrap up the analysis of the algorithm. The first loop will exit at the first i such that $a_i = a_{2i}$; By the second corollary, this happens in less than $\ell_{\text{tail}} + \ell_{\text{loop}}$ iterations. By the first corollary, we know that this value of i must be a multiple of ℓ_{loop}. Finally, we can see that the second loop finds the collision after exactly $\ell_{\text{tail}} - 1$ steps, since it will walk x_{tort} from a_0 to $a_{\ell_{\text{tail}}-1}$ and x_{hare} from a_i to $a_{i+\ell_{\text{tail}}-1}$, and their next steps collide by the third corollary.