

Security Properties of Hash Functions

Let $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$ be a hash function that we suppose collision resistant. Let h' be the following function:

$$h' : \begin{cases} \{0, 1\}^* & \rightarrow & \{0, 1\}^{n+1} \\ x & \mapsto & \begin{cases} 0\|x & \text{if } |x| = n \\ 1\|h(x) & \text{otherwise} \end{cases} \end{cases}$$

Question 1. Show that h' is collision resistant.

Solution. Recall the definition of a collision attack: “can we write an algorithm faster than the birthday bound that outputs a collision?”

Here, the only assumption we have is that h is collision resistant. So we will reduce the collision resistance to that of h . We do the proof by contraposition: we show that if h' is not collision resistant, then h is not collision resistant either.

Suppose that h' is not collision resistant. There exists a collision attack, i.e., we have an algorithm \mathcal{A} that outputs a collision faster than the generic birthday bound. Suppose that \mathcal{A} outputs a pair (x, y) .

We notice that $h'(x)$ cannot start with 0, because this would mean $h'(x) = 0\|x = h'(y) = 0\|y$ leading to a contradiction (we’re supposed to have $x \neq y$). Thus $h'(x)$ starts with 1 and $h'(x) = 1\|h(x) = h'(y) = 1\|h(y)$ implying $h(x) = h(y)$. We can use the algorithm \mathcal{A} as a collision attack on h . This concludes the proof.

Question 2. Show that h' is not preimage resistant.

Solution. Notice that h is not assumed to be preimage resistant, but it is also not assumed to be broken. We simply cannot say anything on its preimage resistance.

Recall the definition of a preimage attack: “can we write an algorithm faster than the exhaustive search, and which succeeds with constant probability?”. The last part is important, because it’s OK if the attack does not succeed on all possible targets. You can think of this as follows: when you attack the hash function, the target preimage will typically be chosen at random, and if the attack fails, you simply re-run it.

So, let’s design such an algorithm. Input a target t taken uniformly at random. If t starts with 0 then $t = 0\|t'$ and t' is a valid preimage of t , following the definition of h' . If t starts with 1, we just fail. Then we succeed with probability $1/2$ on random inputs, which is enough to claim an attack.

Collisions

During this exercise, I suggest to generate random integers using the function `randrange` from the package `random`: `randrange(N)` generates a uniform random integer from $\{0, 1, \dots, N - 1\}$. You may start from the file `tp1_code.py`.

We use the library `hashlib` from Python’s standard library, which implements many hash functions (MD5, SHA-1, SHA-2, SHA-3). These constructions take as inputs objects of type `bytes` and not `str`. In particular, you need to use the `.encode()` function or directly construct a byte string with the prefix “b”. Read the documentation¹ for more information. Here is an example:

¹<https://docs.python.org/3/library/hashlib.html>

```

1 import hashlib
2
3 sha2 = hashlib.sha256
4 print(sha2(b"Hello world").hexdigest())
5 print(sha2("Hello world".encode()).hexdigest())

```

Collision on Truncated Hash Function

Question 3. Choose a prefix (for example your first name) and implement the generic collision search algorithm using the birthday paradox, to find two strings s_1 and s_2 both starting with this prefix, so that the 32 first bits (i.e., 8 first characters in the hex string) of $\text{SHA2}(s_1)$ and $\text{SHA2}(s_2)$ are equal. For example:

Collision found !

Input 1: maxime15857573905157511205

Input 2: maxime13871373172309900626

sha2 (input1) =

a4de129026e4f1b46270dc73772a14c26d90c3df19d2a040d347cc154d38c4f8

sha2 (input2) =

a4de1290d4324581554e4804b53f01f95211371a4241386372502d571fc1e06c

SHA256 prefix (first 32 bits) : a4de1290

Question 4. Estimate roughly the time and memory complexities of your algorithm. Does it depend on the length of your first name?

Solution. We expect that (up to a small constant) it will take 2^{16} evaluations of the hash function, and 2^{16} memory. Both are quite small, so your code should run instantly. (You should use a dictionary to store the hash outputs, as it allows fast access.)

This complexity only depends on the length of the output, not on the chosen prefix.

Collisions with Small Memory

In the following, we identify an n -bit truncated hexadecimal hash to an integer between 0 and $2^n - 1$, and use the function `sha2Trunc` defined in `tp1_code.py`.

Let $H : \{0, 1, \dots, 2^n - 1\} \rightarrow \{0, 1, \dots, 2^n - 1\}$ be a function (for example the `sha2Trunc` function). Starting from a message X_0 , we define the sequence $X_{i+1} := H(X_i)$. Since it takes values in a finite set, it's necessarily periodic after some point.

We denote by c the length of the pre-period X_0, \dots, X_{c-1} (the length of the tail of the ρ) and ℓ the length of the cycle, so that $X_0, \dots, X_{c+\ell-1}$ are all distinct.

Floyd's cycle-finding algorithm is given in ???. It defines another sequence $Y_i = X_{2i}$, i.e., $Y_0 = X_0$ and $Y_{i+1} = H(H(Y_i))$, and outputs an element x that belongs to the cycle.

Question 5. Implement Floyd's algorithm with a uniformly random value for X_0 .

Question 6. Write an algorithm to find the length ℓ of the cycle.

Algorithm 1 Floyd's cycle-finding algorithm.

```
1:  $x \leftarrow H(X_0)$ 
2:  $y \leftarrow H(H(X_0))$ 
3: while  $x \neq y$  do
4:    $x \leftarrow H(x)$ 
5:    $y \leftarrow H(H(y))$ 
6: end while
```

Solution. We simply restart a chain from the x obtained at the previous question. We know that x belongs to the cycle, so we compute $H^i(x)$ until $H^i(x) = x$: the first such i is the length of the cycle ℓ .

For this question, you can compare the result with your neighbours. Indeed, despite the fact that we started from random elements, with high probability we fall on the same cycle (the one of the large component), so we will obtain the same value.

If $c > 0$ and $\ell > 1$ one has:

$$H(X_{c-1}) = X_c = X_{c+\ell} = H(X_{c+\ell-1})$$

and by definition $X_{c-1} \neq X_{c+\ell-1}$. We will use this property to find a collision.

Question 7. Write two new chains that start from X_0 and X_ℓ and compute X_i and $X_{i+\ell}$ until equality. During the computation of the chains, you should remember the previous element (thus 4 variables are required), and output it when $X_i = X_{i+\ell}$. Check that you indeed obtained a collision.

Question 8. Deduce examples of collisions on the n first bits of SHA256 for $n = 16, 32, 40, 48$.