Assignment 6–Hash Functions

CSCI3381-Cryptography

Due Wednesday, April 5

There's really only one problem here, but at three different levels: Use the birthday attack to generate a collision in a reduced version of SHA-1 (just a 40-bit hash). To find the hash value of a string `s`, you can execute the following Python code:

```
import hashlib
hashval=hashlib.sha1(s).hexdigest()
```

We will use the high-order 40 bits (ten hex digits) as the 'hash value', so as to make this problem do-able.

Problem 2 is harder than Problem 1, and Problem 3 is harder than Problem 2. You get 60 points for a completely successful solution of Problem 1, 80 for Problem 2 (in which case you don't have to hand in Problem 1), 100 for Problem 3 (in which case you don't have to hand in 1 or 2).

1 Basic birthday attack on a 40-bit hash

Write a function `birthday1()` that returns a tuple (s, t, n), where s and t are different ASCII strings whose SHA-1 hashes have the same high-order 40 bits (same 10 initial hex digits). The last component n of the return value is the number of calls to SHA-1. Again, you can generate random ASCII strings by converting random integers to hex. By the theory of these birthday attacks, you will need to compute somewhat more than 1 million hashes to find this collision with probability greater than $1/2$. The simplest way to do it is to repeatedly generate random strings `s` and enter the pair

```
SHA-1(s):s
```

in a Python dictionary structure. When you find a hash value that's already in the dictionary, you're done. Include two different colliding pairs of strings in your writeup.

2 Low-memory birthday attack on a 40-bit hash

The birthday attack in the preceding problem required a dictionary with 1 million+ items. Implement the Floyd cycle-finding algorithm described in the notes to generate the same kind of collision using almost no memory. Call the function `birthday2()`. The return value should have the same format. Note that you will require a larger number of calls to the hash function, because each step of the initial phase of the algorithm requires the computation of three hashes:

$$(x_i, x_{2i}) \mapsto (h(x_i), h(h(x_{2i})) = (x_{i+1}, x_{2(i+1)}),$$

and each step of the second phase requires the computation of two hashes. Using different starting points will give you different collisions. Include two different colliding pairs of strings in your writeup.

3 Low-memory birthday attack with a meaningful collision.

Produce a pair of *meaningful* ASCII texts that will permit you to cheat at the coin-tossing-by-telephone game. Announce that you are writing out your guess of the outcome of the coin toss, and storing it in a text file. You then provide a hash of the text to your friend. After he announces the outcome of the toss, you send him the text file, and he uses the hash function applied to the text to verify that the guess you provided was correct. Of course you have stored *two* different text files, and wait until the toss is announced to decide which one to send him.

The function you write, `birthday3()`, should again return a triple (s, t, n) with the SHA-1 hashes of s and t agreeing in the high-order 40 bits, but this time `s` and `t` will be something like,

```
I <your name> have a prediction.
I predict that the coin you are tossing
will come up heads.

I, <your name>  believe
that that penny you are
about to toss will show tails.
```

Here is the trick: Implement a function that takes as input a 40-bit string b and produces as output an ASCII string $f(b)$ having one of the formats above. If the rightmost bit of b is 0, the last word of the message is 'tails', if the rightmost bit is 1, the last word of the message is 'heads'. Use other bits of the message to guide a choice between two words that will not change the meaning of the message. For example, you could construct the messages as

```
I <,|> Howard Straubing <, |>
<do |> <hereby|> <guess|predict>
<the following outcome|this result> ...
```

If you put 30 or so such choice points within the message then more than one billion different messages are possible. Now, use the Floyd cycle-finding algorithm to find a collision, not in the reduced hash function H, but in the function

$$K(x) = H(f(x)).$$

A collision means that you will have two ASCII messages x_1, x_2 such that $H(f(x_1)) = H(f(x_2))$. It is possible that x_1, x_2 both predict heads, or both predict tails. In that case, you should run the algorithm again. In about half of all trials, you will find messages predicting opposite results that collide. Include the two text files with your submission. (Executing the two lines

```
hashlib.sha1(open('file1.txt','r').read()).hexdigest[10]
hashlib.sha1(open('file2.txt','r').read()).hexdigest[10]
```

should give identical values.)