Problem Set 7–Hash Functions

CSCI3381-Cryptography

Due November 21, 2014

All these problems use reduced versions of the hash function SHA-1. To find the hash value of a string `s,` you can execute the following Python code:

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

The result is a string of 40 hex digits. If you want to extract the first n bits, where $n = 4m$, then you can just write `hashval[:m]`. We will consider reduced versions of SHA-1 in which $n = 20$ and $n = 40$.

Your finished homework should contain the following elements:

- A file `hw7.py` containing all the functions described in the problems, with exactly the same parameter lists. It is perfectly all right to have additional functions in this file, especially for the last problem, but the end result should be computed by a function of the prescribed form.
- A text-processed document containing the solutions you found for each of the problems. All solutions should very *briefly* discuss how the complexity of the solution (the number of hashes) compares with the statistical predicitions.

1 Hashcash tags

(a) Write a function:

```
compute_tag(s)
```

where the argument `s` is a string, and the return value is a tuple `(t,m)`. The component `t` should be a string formed by appending up to 16 hex digits to `s`. For example, if `s` is:

```
Boston College
```

then `t` might look like

```
Boston College70878594372e29fa
```

or

```
Boston College4d6082044a0f587a
```

The string `t` should also satisfy the following property: The SHA-1 hash of `t` should have its high-order 20 bits (*i.e.,* its 5 leading hex digits) all equal to 0. The component m should be the number of calls to the hash function. Incidentally, both the strings shown above in the examples have the required property.

A reasonable strategy for this problem is to generate random integers in the range 0 to $2^{64} - 1$, convert them to hex using the `hex` function, and strip away the leading `'0x'` and the trailing `'L'`. Eventually you'll find a hex string that works.

(b) Use your result in *(a)* to compute *two different* legitimate hashcash headers associated with today's date and YOUR e-mail address. The tag should have the format

```
0:dddddd:youraddress@bc.edu:xxxxxxxxxxxxxxxx
```

where `dddddd` is the date in format [2-digit year,month, day] (for instance 141112 for November 12, 2014), and the `x` are hex digits.

2 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.

3 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.

4 Birthday attack with a meaningful collision.

Produce a pair of *meaningful* ASCII texts that will permit Alice to cheat at the coin-tossing-by-telephone game. Your function `birthday3()` should again return a triple (s, t, n) with the SHA-1 hashes of s and t agreeing in the first 40 bits, but this time `s` and `t` will be something like,

```
I swear on a stack of bibles that the penny I just tossed came up heads.
I, Alice, do affirm that the shiny penny I just tossed landed TAILS!
```

This version of the birthday attack is described in Section 9.4, in the chapter on digital signatures, but you don't need to know about signatures to understand it. The slightly different math behind it is explained very briefly at the top of page 231. The idea is that by introducing about 20 choice points in the formation of the first message (*e.g.,* Alice or no name, affirm or swear, comma or no comma, *etc.*) you can create 2^{20} variants of the same message, all saying that the coin came up heads. (Be creative!) Enter these strings, along with their hashes, into a dictionary. You can similarly create 2^{20} versions of the tails message and check if the hash of the message is in the dictionary.

The cheat is that Alice tells the common hashed value to Bob before he guesses, and then gives him the tails message if he guesses heads, and vice-versa.