
Problem Set 2

Submit this problem set in PostScript, PDF, or MS Word format to `6857-submit@csail.mit.edu` before lecture on the due date. We have provided templates for L^AT_EX and Microsoft Word on the course website. Each solution must appear on separate sheets of paper. Mark the top of each sheet with your name(s), the course number (6.857), the problem set number and question, and the date.

You are to work in groups of three or four people and should submit a single set of solutions for all problems parts designated [**Group**]. You should turn in a separate, individual solution to any problems designated [**Individual**].

We may distribute our favorite solution to each problem as the “official” solution. If you do not wish for your homework to be used as an official solution, or if you wish that it only be used anonymously, please note this on your homework.

Problem 2-1. Revoke Your Fake Keys from PS1-1 [Individual]

Please write and sign the following statement:

“I have revoked any PGP keys I may have created in anyone else’s name.”

Problem 2-2. Finding Hash Collisions (Without Wasting Memory) [Group]

The standard birthday attack mentioned in lecture is fast, but requires an extremely large amount of memory. For example, if the hash function outputs mere 64-bit strings, an attacker would have to remember approximately 2^{32} strings of 64 bits each, requiring at least one terabyte of active memory. For this problem, you will implement a more memory-efficient method for finding hash collisions.

Such algorithms typically work by repeatedly applying a hash function in order to find a *cycle*.¹ That is, starting with some x_0 , the algorithm will compute $x_1 = h(x_0)$, $x_2 = h(x_1)$, ... $x_i = h(x_{i-1})$. Note that all of the values x_i (except possibly x_0) come from the range of h , which is finite. Thus, eventually this technique will find an i and j such that $x_i = x_j$. Then repeated application of the hash function will simply go around a closed cycle:

$$x_{j+1} = h(x_j) = h(x_i) = x_{i+1}$$

$$x_{j+2} = x_{i+2}$$

...

$$x_{2j-i} = x_j$$

The *cycle length* c is therefore $j - i$. Once a cycle is detected, one must go back to where one first entered the cycle. Here, there are two values—the last value before entering the cycle, and the last value within the cycle before it starts to repeat—that hash to the same value. Thus, a hash collision has been found. Figure 1 illustrates such a hash chain:

Of course, keeping track of all x_i values in the chain would be just as memory-inefficient as keeping a full table as with a naive birthday attack. However, there are several different methods designed to detect cycles while remembering only *some* of the x_i values.

Based on the references below, implement a memory-efficient algorithm that, on input k , will find a hash collision in the k th truncated SHA1 function $h^{(k)}$. Define $h^{(k)}(x)$ simply as the first k bits of $\text{SHA1}(x)$. For example, since $\text{SHA1}(\text{“6.857 is fun”}) = \text{ad7ce0c6f95397fdabde8eeab91633df82aad4a2}$, $h^{(20)}(\text{“6.857 is fun”})$ would simply be `ad7ce`. Note that each hexadecimal symbol denotes four bits. For your initial value x_0 , use the hash of the name of one of the members of your group, e.g. $h^{(k)}(\text{“Ben Bitdiddle”})$. Run your algorithm for various lengths k .

For this section, turn in:

¹This technique can actually be applied much more broadly; for example, Pollard’s rho method (see references for this problem) uses this technique for factoring numbers.

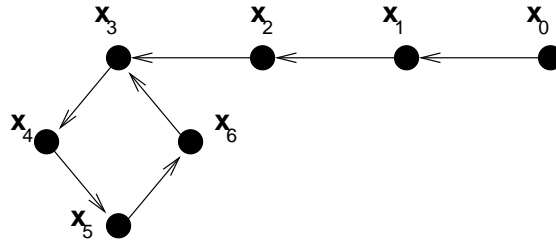


Figure 1: A hash chain of length 7 with a cycle length of 4. The hash $h(x_2) = h(x_6)$.

- The largest value of k for which you were able to find a collision.
- The name of the group member, and the resulting initial value $x_0 = h^{(k)}(\text{name})$ for your largest value of k .
- The collision itself; i.e. the two values that hash to the same value.
- The length c of the cycle you found.
- The approximate amount of computation time it took (e.g. “This took four hours on a 1.2 GHz Athlon machine”). Please be considerate to other users if you run your code on a shared computing resource.

Please express any hashed values in hexadecimal notation.

Algorithmic references.

These references, while not necessarily mentioning hash functions, give memory-efficient algorithms that can be adapted for the problem above. **Note:** These references will give you pointers on how to find a cycle. However, you still have to figure out how to detect where you *first* enter a cycle (since that is where you can find a collision), and how to do so in a memory-efficient manner.

Nivasch’s stack-based cycle-detection algorithm uses (on average) logarithmic storage space, and can be adapted with tradeoffs for storage space versus speed. A formal description of the algorithm can be found at <http://www.wisdom.weizmann.ac.il/~gabrieln/papers/cycleDet.pdf>, and an informal summary at <http://yucs.org/~gnivasch/stackalg/>.

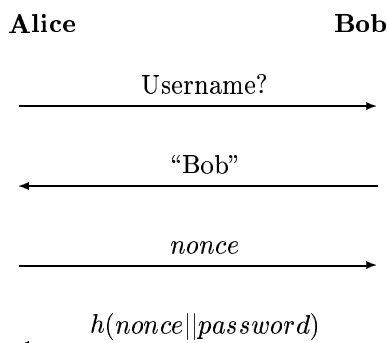
Floyd’s cycle-finding algorithm is the basis of several different techniques, including the Pollard rho algorithm used for factoring. It uses constant memory space. There are discussions of the Pollard rho algorithm in section 3.2.2 of *Handbook of Applied Cryptography* by Menezes et al., and in section 31.9 (pp. 896-901) of *Introduction to Algorithms: Second Edition* by Cormen, Leiserson, Rivest, and Stein. Also, Brent has a statement of Floyd’s algorithm, along with his own improvements, at <http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pd/rpb051i.pdf>

Coding resources.

There are various cryptographic libraries available for many different programming languages. For computing SHA1, Python has the `sha` module. For C, there are numerous libraries available, including `libgcrypt`. C++ has the `Crypto++` library, Java has class `MessageDigest`, and Perl has `Digest::SHA1`. We don’t expect you to implement SHA1 yourself!

Problem 2-3. Hash Login Scheme [Individual]

Consider the following login scheme:



In this system, Alice keeps a local record of Bob’s password. For each login session, Alice will challenge Bob with a random “nonce”. Bob will hash this nonce concatenated with his password using the function h . Alice will allow Bob to login only if the value $h(\textit{nonce}||\textit{password})$ matches what she computes locally. Note that Bob’s password is never sent in the clear.

Consider an adversary, Eve, who monitors several valid login sessions by Bob, then tries to log in pretending to be him. Eve can compute h and perform any polynomial-time computations she wants. Eve can only talk to Alice and cannot send any messages to Bob in an attempt to play (wo)man-in-the-middle.

Suppose that h has both the one-way and weak collision resistance properties. In other words, h is a “one-way hash function” by Definition 9.3 from the Handbook of Applied Cryptography. Is h sufficient to protect this system from Eve?

Problem 2-4. One-way and Collision-resistant Hash Functions [Group]

Let h be a one-way collision-resistant hash function mapping some domain D to itself, i.e. $h : D \rightarrow D$.

Extend h to a mapping from sequences (a_1, a_2, \dots, a_n) to elements of D as follows. We call the resulting hash algorithm h^* :

1. $h^*(()) = h(d_0)$, where d_0 is some fixed element of D .
2. $h^*((a_1, a_2, \dots, a_n)) = h(a_1 || h^*(a_2, \dots, a_n))$

Argue that h^* is both OW and CR.

Problem 2-5. One-Time Pads [Group]

The one-time pad works as follows:

- The message M is divided into a sequence of “elements” (in the usual case, these elements are bits). Let S denote the set of possible elements. (For example, $S = \{0, 1\}$ if the elements are bits.) Suppose that $M = m_1 m_2 \dots m_n$.
- A “pad” $P = p_1 p_2 \dots p_n$ of the same length is chosen by choosing each element p_i uniformly at random from S .
- The ciphertext $C = c_1 c_2 \dots c_n$ is formed by combining each message element with the corresponding element of the pad, using some encryption function e :

$$c_i = e(p_i, m_i)$$

- Of course, it must be decryptable to someone who knows the pad, so there is a corresponding decryption function d that operates element by element, using each element of pad and each element of the ciphertext to produce the corresponding element of the message:

$$m_i = d(p_i, c_i)$$

- In the case where the elements are bits, the operations e and d are usually taken as addition modulo 2 (i.e. xor or \oplus). What is another choice for e and d that provides unconditional security? (Proof not needed.)

- (a) The key to the proof of unconditional security is that

$$\forall c, m \Pr[(c_i = c) | (m_i = m)] = \frac{1}{|S|} \quad (*)$$

That is, for each element of the ciphertext, any of the $|S|$ possible message elements could have produced it with equal probability. This will hold whenever:

1. The elements of C are also elements of the set S .
2. There is some function $r(., .)$ that permits recovering the pad element p_i from the corresponding elements m_i and c_i :

$$r(m_i, c_i) = p_i$$

Argue that (*) follows from (1) and (2).

- (b) A finite *group* consists of a finite nonempty set S together with a binary operator \circ (“composition”) mapping S^2 into S , such that:

1. \circ is associative (i.e. $\forall x, y, z \in S : (x \circ y) \circ z = x \circ (y \circ z)$)
2. S contains an *identity element* denoted I such that for all $x \in S$: $x = I \circ x = x \circ I$
3. For every element $x \in S$ there is another unique element in S , denoted x^{-1} , such that $x \circ x^{-1} = x^{-1} \circ x = I$

(It is not necessary that $x \circ y = y \circ x$; if this holds we have a special kind of group known as a commutative group or an abelian group.)

Show that an arbitrary finite group can be used as a basis for a one-time pad, by defining how the encryption operator e and the decryption operator d should work, and by defining how the recovery operator r works, all in terms of the group operations of composition \circ and inverse $^{-1}$.

- (c) Suppose that your message was naturally composed of elements that are the 27 symbols A, B, ..., Z and “space”. What group could you use to make a one-time pad for this set of symbols?