Massachusetts Institute of Technology
6.857: Network and Computer Security
Professor Ronald L. Rivest

Handout 3
September 18, 2006
Due: October 2, 2006

Problem Set 2

This problem set is due, *via email,* to `6.857-submit@theory.csail.mit.edu` by *Monday, October 2* at the beginning of class.

You are to work on this problem set in groups of three or four people. Problems turned in by individuals, pairs, pentuples, etc. will not be accepted. Be sure that all group members can explain the solutions. See Handout 1 (*Course Information*) for our policy on collaboration. If you do not have a group, let us know.

Homework must be submitted electronically! Each problem answer must appear on a separate page. Mark the top of each page with your group member names, the course number (6.857), the problem set number and question, and the date. We have provided templates for LaTeX and Microsoft Word on the course website (see the *Resources* page).

Grading and Late Policy: Each problem is worth 10 points. Late homework will not be accepted without prior approval. Homework should not be submitted by email except with prior approval. (*Somebody* from your group should be in class on the day that the homework is due.)

With the authors' permission, we will distribute our favorite solution to each problem as the "official" solution—this is your chance to become famous! 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. Finding Hash Collisions using Cycle Detection

Let h be a hash function that maps some finite domain D onto itself (e.g., D is the domain describing all unique 128-bit binary values). Given initial element $x_0 \in D$, consider the infinite sequence: $x_1 = h(x_0), x_2 = h(x_1), x_3 = h(x_2), ...$

Because D is finite, this sequence must eventually repeat a value. This will lead the sequence to subsequently cycle through the same sub-sequence from then on. Such a cycle can be used to find a collision for h. Specifically, let x_i be the first value encountered in the cycle. Let x_j be the last value in the cycle before we return to x_i. It follows that $h(x_{i-1}) = h(x_j) = x_i$. Therefore, if we can detect where a cycle begins in a such a hash sequence we can identify a collision for the hash function.

There exist quite a few algorithms for detecting these cycles without using a prohibitive amount of memory. Consider the following, stack-based cycle detection algorithm, described by Nivasch at `http://yucs.org/~gnivasch/stackalg/` (under the heading **The basic stack algorithm**):

> [The] basic algorithm is as follows: Keep a stack of pairs (x_i, i), where, at all times, both the i's and the x_i's in the stack form strictly increasing sequences from bottom to top. The stack is initially empty. At each step j, pop from the stack all entries (x_i, i) where $x_i > x_j$. If an $x_i = x_j$ is found in the stack, we are done; then the cycle length is equal to $j - i$. Otherwise, push (x_j, j) on top of the stack and continue.

It is not too difficult to show that the size of this stack is bounded by $O(\log |D|)$, with high probability. This is a big improvement over an approach such as the standard birthday attack, which would require, on average, for the attacker to store up to $2^{\frac{|D|}{2}}$ values.

Your Task In this problem, we ask you to implement this stack-based cycle-detection algorithm using the Python programming language. We then ask you to use it to find SHA-256 collisions. Specifically:

(a) Implement Nivasch's cycle-detection algorithm as described above. Notice, this algorithm detects a cycle and calculates its length, but it **does not** determine where the cycle begins—which is necessary to identify a specific collision. You must add a second phase to the algorithm that makes use of the

cycle detection and length information reported from Nivach's algorithm to find the beginning of the cycle and subsequently print out the colliding values.
We expect you to implement the algorithm using Python. See the *Resources* page of the course web site for a Python tutorial as well as some sample code to help you implement stacks and calculate SHA-256 values using the language. We expect your algorithm to work for the k^{th} truncated SHA-256 hash function, $h^{(k)}(x)$, where we define $h^{(k)}(x)$ simply as the first k bits of $SHA256(x)$. For example, h("I hash therefore I am") = ef57b2301c4539bdfe24bf96f14a91e215cc827b, therefore $h^{(20)}$("I hash therefore I am") = ef57b (each hex symbol is 4 bits). As mentioned, the SHA-256 function is readily available using Python's `hashlib` module. Reference the sample code on the *Resources* page to learn how to use this built-in function—don't try to implement SHA-256 on your own! **Note:** To use SHA-256 you need to be running Python version 2.5 (the first verison to include the hashlib library). Python 2.5 is easily installed on Windows, Mac, and linux machines—see the *Resources* page for the appropriate link.

(b) Pick an arbitrary initial value and use your algorithm to discover k^{th} truncated SHA-256 collisions for various values of k. At least one of your k values should be large enough that a birthday attack would have been unreasonable (say, $k \geq 24$).

(c) Turn in a concise explanation of the second-phase of your algorithm (i.e., how it finds where the cycle begins), your source code, and your largest collision (i.e., the two colliding values for the largest value of k you tried, as well as k itself). Describe the colliding values as hex strings in your write-up. If possible, report how long it took you to find the collision, and describe the key specs of the machine you ran your algorithm on.

Problem 2-2. Security of Hash Functions

Ned Nerdle understands that if a hash function has n bits of output and also n bits of internal state (e.g. as does MD5 and SHA-1, for $n = 128$ and $n = 160$, respectively), then the time to find a collision is at most $2^{n/2}$, and the time to invert is at most 2^n.

Ned decides to double the internal state size and output size to $2n$ bits, as follows, hoping to improve collision-resistance to work 2^n and the time to invert to 2^{2n}. His idea is keep the same compression function $f(s, m)$, but to change its usage. Here s is the n-bit input state variable, m is a 512-bit message block input, and $f(s, m)$ is an n-bit compression function output.

The state variable is now a pair (x, y) of n-bit variables. The initial state is called (x_0, y_0). The state after processing k message blocks is called (x_k, y_k). After all message blocks have been processed, the final state is the hash function output.

The state is updated as follows. In iteration i, message block m_i is combined with input state (x_{i-1}, y_{i-1}) via:

$$\begin{aligned} x_i &= y_{i-1} \\ y_i &= x_{i-1} \oplus f(y_{i-1}, m_i) \end{aligned}$$

[This structure is often called a "Feistel" structure.]

(a) Argue that Ned has not improved collision resistance; an adversary can still find collisions in time $2^{n/2}$.

(b) Argue that Ned has not improved inversion difficulty; an adversary should still be able to invert in time 2^n.

Problem 2-3. Time-stamping

A "time-stamping service" accepts a digital document m, and produces a "time-stamp" (certificate) of the form $sign((t, m))$—the time-stamping authority's (TSA's) digital signature on the pair (t, m), where t is the time that m was submitted for time-stamping.

Assume that the TSA uses the usual "hash-and-sign" model for producing digital signatures.

In this case, there is a risk that the TSA's hash function might be compromised (so that it is no longer collision-resistant). Then the time stamps become meaningless.

Perry Noid has had an important document m time-stamped by TSA, producing time-stamp $c = sign_{TSA}((time, m))$. Now Perry is worried that the hash function used by TSA is looking vulnerable to newly proposed attacks.

Perry wishes to make use of another time-stamping service TSA', which signs using a hash function h' which is deemed to be much stronger than the hash function h used by TSA.

Argue that Perry can now use the services of TSA' to obtain a time-stamp for m that will *be effective for the original time-stamp time* t, even if the hash function h of TSA is later compromised.