Massachusetts Institute of Technology
6.857: Network and Computer Security
Professor Ron Rivest

Handout 9
April 1, 2012
Due: April 13, 2012

Problem Set 4

This problem set is due as an email to the course staff, at `6.857-staff@mit.edu`, on *Friday, April 13* by **11:59 PM**. Please note that no late submissions will be accepted.

You are to work on this problem in a group of 4. (Or possibly 3– only one group should only 3 members, and so you will need permission from the TA to have a small group.) Please email the TAs with your group composition once you have it! If you need help finding a group, please try emailing the class list, `6.857-students@mit.edu`. If you need further help finding a group, please email the TAs at `6.857-tas@mit.edu`. Be sure that all group members can explain the solutions. See Handout 1 (*Course Information*) for our policy on collaboration

Homework must be submitted electronically! Submissions should be in pdf form, with the document named by each team member's last name appended. 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: All problems are worth 10 points.

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 in the email used to turn in the submission.

Problem 4-1. MACs For Sets

To save time, some MACs can be used authenticate *sets* of values with a single MAC, instead of having a distinct MAC for each value. Note that since sets are equivalent for any ordering of elements, for a set X, $\text{MAC}(K, X)$ should not depend on the order of presentation of the elements in the set X.

For this problem, you'll design a MAC for sets for the specific application of authenticating a computer's filesystem. You may for this problem conveniently view the file system as a set of (key,value) pairs where the keys are the file pathnames, the values are the file contents, and no two pairs have the same key (but they may have the same value).

One design goal is to have an authentication method that may be implemented to run efficiently in parallel.

To allow this to work, your scheme needs to support the operation of taking unions for (disjoint) sets. For any X and Y that are disjoint subsets the (key,value) pairs for collections of files, your method should be able to compute $\text{MAC}(K, X \cup Y)$ from K, $\text{MAC}(K, X)$ and $\text{MAC}(K, Y)$ *without* using X or Y.

Additionally, given K and a MAC of the filesystem, this should be efficiently updatable upon a delete request for a set of files.

To make this problem nontrivial, the size of $\text{MAC}(K, X)$ should be *independent* of $|X|$– in other words, the size of the MAC should not grow with the number of elements in the set it authenticates. (Otherwise, a trivial solution is obtained by including X in $\text{MAC}(K, X)$.)

For your construction, you may use hash functions (modelled as random oracles), symmetric encryption schemes, and simple operations such as addition or xor as components. (If you'd like to use other components, please ask the TA.)

While supporting this new operation, your MAC scheme should maintain security against a chosen message attack with the additional capabilities for an adversary to request MACs for the union of the (disjoint) sets represented by two MACs, and can request a MAC with some items removed if it can give a set that the MAC is valid for. As usual, a method is insecure if the adversary can forge a MAC for a set it hasn't requested a MAC for.

Describe your construction, and argue as best you can that your method is secure by relating the security of your scheme to the assumed security of its components.

Problem 4-2. Generating Random Numbers, with Factors!

Adam Kalai has a nice (and short!) paper describing how to generate random numbers along with their factors. You can find it at

`http://people.cs.uchicago.edu/~kalai/papers/old_papers/factorcryptology.pdf`

(a) The Proof of Claim 2 would be correct if a small typo/mistake was fixed - what is it?

(b) Based on the analysis in Kalai's paper, what fraction of the time is the process aborted because the product of the prime s_i's exceeds n?

(c) In Lecture 13, Professor Rivest explained how to create a random generator for $\mathbb{Z}_p^*$ where $p = 2q + 1$ is a safe prime, i.e., q is also prime. Outline how you could use the results of Kalai's paper to find, given an integer n, both a prime p and a generator of $\mathbb{Z}_p^*$, where p is a prime selected uniformly at random between 1 and the given integer n.

Your procedure should be efficient even when $p - 1$ has many prime factors.

Describe your procedure, and argue that your procedure operates in time polynomial in $\log(n)$, on the average.

Problem 4-3. RSA Keys with Overlapping Primes

A recent paper, "Ron was wrong, Whit is right" at `http://eprint.iacr.org/2012/064`, discusses the generation of RSA keys with weak randomness; this may cause one of the two prime factors of an RSA modulus n_i to be the same as the prime factor in a different RSA modulus n_j. Since there are efficient algorithms to find the greatest common divisor, such unintentional "sharing" of prime factors between RSA keys can make it relatively easy to factor both keys and thus break their security!

The course staff will post a list of one million 256-bit RSA moduli, each the product of two 128-bit primes, some of which (about 1%) will share prime factors. For example, if we number the moduli from n_0 to n_{999999}, then the first overlap occurs between n_{58} and n_{51}. If you work on only the first k moduli, then between them there should be about $k/100$ pairs of moduli that share a prime factor.

Your assignment: factor as many of the moduli as possible. Moreover, you will also be graded on the efficiency of your algorithm.

Please include the code you use in your writeup of the problem, and an analysis of its efficiency. Additionally, please attach a txt file with the keys you are able to factor, with the format of one n p q per line, where n is a public key from the list, and p and q are its prime factors.

Since the list we're providing is very long, your search will be significantly easier if you write an efficient algorithm.

Nadia Heninger has a recent blog on this subject at `https://freedom-to-tinker.com/blog/nadiah/new-research-theres-no-need-panic-over-factorable-keys-just-mind-your-ps-and-qs`.

She recommends Daniel Bernstein's paper "How to Find Smooth Parts of Integers" as a reference, available at `http://cr.yp.to/factorization/smoothparts-20040510.pdf`.

Sage (`http://www.sagemath.org`) is an excellent open source mathematics system built on top of python. The implementation of "gcd" that it contains is *much* faster than the one built-in to Python! This could be helpful for you, or you can try to use a faster language or library (perhaps NTL?).