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

Handout 12
September 26, 2002

Problem Set 3

This problem set is due on *Thursday, October 3, 2002* at the beginning of class. Late homeworks will *not* be accepted. This entire problem set is a group problem set. You must, however, continue to cite your sources properly.

At the end of each problem, tell us how each person contributed (writing, designing, coding, proof reading, never showed up, etc).

Problem 3-1. All your PRIMES is belong to us[1]

You are a new technical lead at PRIMES Computer, Inc. Your job is to manufacture groovy primes. Your boss, Austin M. Powers[2], has generated a special task for your group.

In a 6.857 lecture secretly simulcast at PRIMES Computer, Inc., you learned about primality testing and generators of a group Z_p^* where p is prime. Austin has assigned you the task of generating such values.

Austin says you may use any programming language you like, though we recommend Java because of its BigInteger package. The numbers you will be working with are quite large – plain old `long` or `integer` types won't be large enough. If you use C or C++, you could try the GNU mp package. If you use this package, you are on your own though.

In this problem, you may not use any built-in procedures for primality testing, prime number generation, or generator discovery. You can assume you have a source of randomness. If you borrow any code, cite the appropriate authors. Moreover, you must explain how and why any borrowed code works with inline comments.

We recommend you read Chapter 31 of CLRS [1] and perhaps pages 160–164 of HAC [3].

(a) Generators of Z_p^*

Write a function to generate a b-bit prime given parameters b and a source of randomness. Your function should produce a b-bit number that has an overwhelming probability of being prime. Write a second function that given g, p, and the prime factorization of $p-1$ tests whether the element g is a generator of Z_p^* where p is prime.

Using these functions, write a program that given a list of names, a number b, and a source of randomness, produces a b-bit prime number p for such that each name (treated as an element of Z_p) is a generator of Z_p^*.

A potentially comforting fact is that the number of generators of Z_p^* is $\phi(\phi(p))$ where ϕ is Euler's phi function. In the special case where your prime $p = 2q+1$ and q is prime, then $\phi(\phi(p)) = q-1$. If you generate a prime p with this form[3], then you already know the factors of $p-1$.

To encode names as elements, take the hexadecimal ASCII representation. Then combine the digits into a single number. The first letter of a name is the high order byte, the last letter of the name

[1] Somebody set us up the factors

[2] Modular is his middle name.

[3] You have no chance to factor, make your time.

is the low order byte, etc. If your group consisted of members Kevin, Thien-Loc, and Ron, your names and elements would be:

Name	ASCII Hex	Element in decimal
Kevin	0x4B 65 76 69 6E	323824806254
Thien-Loc	0x54 68 69 65 6E 2D 4C 6F 63	1557050158367982251875
Ron	0x52 6F 6E	5402478

Print out and submit your code. Please make sure there are no line-wrap problems.

(b) Expected amount of work to find primes and generators

For a random prime number p, what is the approximate probability of a random element $g \in Z_p$ being a generator of Z_p^*? You can assume p is large.

How many trials do you expect to make for your group to find a suitable prime p such that each of your names is a generator of Z_p^*? Give your estimate in terms of b (the size of a b-bit prime p) and the number of students in your group s.

(c) Find a prime

Using your program from part (a), find a prime number p where your first names are generators of Z_p^*. (1) Tell us this prime. You can try for any size prime. We recommend trying small bit sizes (e.g., 64) for testing. If you finish early, try to generate a 1024-bit prime (this could take a lot of time especially in Java, so only do this for fun if you have time).

(2) In your final run of the working program, how many trials did it take to find a suitable prime? That is, how many random numbers did you try before finding a suitable prime?

(d) Finding co-Sophie-Germain primes

We alluded to *Sophie-Germain* primes in part (a). A number q is a Sophie-Germain prime if $2q+1$ is also prime. We then call the value $2q+1$ a co-Sophie-Germain prime (or sometimes a *safe prime*).

While we know there are an infinite number of primes, it's an open problem whether there are an infinite number of Sophie-Germain primes. It turns out that in practice, however, Sophie-Germain primes are not too rare.

Assuming that the likelihood of q being prime and $2q+1$ being prime is independent, approximately how many 1024-bit Sophie-Germain primes would you expect to exist?

Problem 3-2. A MAC based on a block cipher

Alice missed Professor Rivest's lecture recommending that to obtain both confidentiality and authentication, one should first encrypt the message, then append a MAC of the ciphertext to the end of the ciphertext.

So, for her project she implements the following procedure. In her project all messages are a sequence of 128-bit blocks exactly (no partial blocks). She first appends a checksum block to the end of the message, where the checksum block is the XOR of the message blocks. Then she encrypts the message/checksum sequence using AES in CBC mode using a randomly-chosen IV. The unencrypted IV, as usual, becomes the first block of ciphertext.

The recipient Bob first decrypts the message/checksum sequence, and then checks that the checksum is correct. If the checksum is not correct, Bob concludes that the ciphertext was forged or altered in some manner; it is not authentic. Of course, Alice and Bob share a secret key for the encryption operation.

Alice figures that since the message and checksum are encrypted, an adversary who doesn't know the encryption key should not be able to forge a ciphertext that Bob will accept as authentic.

Show that Alice's scheme is insecure against a *chosen plaintext attack*, in which an adversary can arbitrarily choose a number of plaintext queries, and obtain Alice's encryption of each of them (with checksum). The adversary is judged to have succeeded if he can then produce a new ciphertext, not identical to any that Alice returned to him, that Bob would nonetheless accept as authentic. The attack may or may not need to be *adaptive*, in which later queries depend on the responses obtained to earlier queries.

Problem 3-3. Block ciphers need to be non-affine

Eager to become a zillionaire soon, Kevin Nguyen decides to design a secure telephone protocol. In order to convince the investors to fund his start-up, he needs to present a live demo. As usual, he devotes the task of designing the crypto component to Thien-Loc Fu.

In need of a fast but secure cipher for real-time encryption/decryption, Thien-Loc opts for a hardware implementation of 128-bit AES. Unfortunately, Thien-Loc's prototype is too slow and too costly for Kevin's needs. Instead of further optimizing the hardware circuits, Thien-Loc decides to modify AES to improve performance and reduce costs. He gets rid of the S-box (that is, replaces it with the identity function).

You can refer to Handout 9, and especially [2] for a detailed description of AES.

Definitions: Let E and F be $\mathcal{GF}(2^8)$-vector spaces of dimension m and n.

1. A function/transformation $f : E \to F$ is linear if it can be written as $f(x) = A * x$, where A is an n x m matrix with coefficients in $\mathcal{GF}(2^8)$
2. A function/transformation $g : E \to F$ is affine if it can be written as $g(x) = A * x + b$, where A is an n x m matrix with coefficients in $\mathcal{GF}(2^8)$ and b is a constant vector in F.

I - TAES is an affine transformation

In this part, we prove that TAES is an affine transformation.

(I-a) Without the S-box, TAES is a composition of the following operations: *ShiftRow*, *MixColumn*, and *AddRoundKey*.

What are the domains and codomains of each of the aforementioned operations? What are the domain and codomain of TAES?

Prove briefly that each of the aforementioned operations is an affine transformation.

(I-b) Prove that the composition of affine transformations is also affine. Deduce from part (a) that TAES is affine.

(I-c) TAES is invertible. Is TAES^{-1} affine? Justify your answer.

II - An affine transformation is easily breakable

In this part, we show how to break an affine transformation.

Let g be an affine transformation: $g(x) = A * x + b$, where A is an n x n matrix with coefficients in $\mathcal{GF}(2^8)$, and b is a column vector of dimension n with coefficients also in $\mathcal{GF}(2^8)$.

(II-a) Explain how you can reduce the resolution of g to that of the linear transformation $f(x) = A * x$. What is the minimum number of pairs (x, f(x)) that you need to know in order to achieve that reduction? What properties do they need to satisfy in order to reach that minimum?

(II-b) What is the minimum number of pairs (x, f(x)) that you need to know in order to completely break f? What properties do they need to satisfy in order to reach that minimum?

III - How to break TAES

We regroup here the results of parts I and II.

Describe a *chosen plaintext attack*, in which an adversary can arbitrarily choose a number of plaintext queries, and obtain the TAES encryption of each of them.

What is the minimum number of messages do you need to query? Give a sequence of actual plaintexts and explain how you use them (along with the corresponding ciphertexts) to break TAES.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 2nd Edition.* MIT Press/McGraw-Hill, 2001.

[2] Joan Daemen and Vincent Rijmen. Rijndael: the Advanced Encryption Standard. *Dr. Dobb's Journal*, pages 137–139, March 2001.

[3] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.