Programming Project, CS378, Spring 2015
Implementing ElGamal Encryption

Andrew Klapper[*]

1 Overview

The purpose of this project is to implement the ElGamal cryptosystem. This is a widely used public key cryptosystem. The components you will need are the following. They are described in more detail later in this document.

1. The facility for doing large integer arithmetic. See Section 7. You may opt to
   (a) For up to 100 points: implement your own large integer arithmetic package. See Section 8 for more details.
   (b) For up to 90 points: use an existing large integer package (such as PARI/GP).
   (c) For up to 80 points: use a language with built in large integers (such as Python).
2. A fast modular exponentiator. See Section 2.
3. A module that finds a prime number p and a primitive root g modulo p. You should base prime number finding on prime number testing by either the Miller-Rabin Primality Test or the Solovay-Strassen Primality Test. See Section 3.
4. A module that converts a sequence of bytes $m_0, m_1, \cdots$ into a sequence of integers $z_0, z_1, \cdots$ with $0 \le z_i < p$. See Section 4.
5. A module that converts a sequence of integers $z_0, z_1, \cdots$ with $0 \le z_i < p$ into a sequence of bytes $m_0, m_1, \cdots$. See Section 4.
6. A module that generates a public/private key pair for ElGamal. See Section 5.
7. ElGamal components: Key setup using prime number and primitive root finding, encryption and decryption using fast modular exponentiation. See Section 6.

[*]Dept. of Computer Science, 307 Marksbury Building, University of Kentucky, Lexington, KY, 40506-0633. www.cs.uky.edu/∼klapper.

2 Modular Exponentiation

For component (2), you should write a module that inputs integers a, k, and m and returns $a^k \pmod{m}$. Use the method described on page 79 of the text.

You will use this module in component (3) if you implement Solovay-Strassen and in components (6) and (7).

3 Prime Number/Primitive Root Finding

You need large prime number p and a primitive root g as global parameters for ElGamal. Write a module that inputs integers n (the desired size of the prime p) and t (the desired confidence that p is prime) and finds a prime number p in the range $2^n \leq p < 2^{n+1}$ together with a primitive root g modulo p. The algorithm should operate by randomly choosing an odd integer x, with $2^{n-1} \leq x < 2^n$, and testing whether x is prime with probability at least $1 - 2^{-t}$. If it is not, then try another x. If it is, then test whether, $p = 2x + 1$ is prime with probability at least $1 - 2^{-t}$. If it is not, then try another x. If it is, then randomly choose an integer g with $2 \leq g < p$ and test whether the multiplicative order of g is $p - 1$ (by a fast test for primitivity using the prime factorization of $p - 1 = 2x$). If it is not, then try another g. If it is, return p and g.

The primality testing should be done by executing $t/2$ trials of the Miller-Rabin Primality Test or t trials of the Solovay-Strassen Primality Test (see Section 6.3, page 176 of the text). Decide that an integer is prime if and only if all trials report that it is prime.

4 Message Encoding

The message to be encrypted will appear as a sequence of bytes. Somehow we must encode this as a sequence of integers z_i with $0 \leq z_i < p$. Here is one way to do this. Suppose the message is $m = m_0, m_1, \cdots$. Each m_i is a byte, which we can interpret as an integer with $0 \leq m_i < 2^8$. We can interpret the first k message bytes as an integer z_0 by letting

$$z_0 = \sum_{i=0}^{k-1} m_i 2^{8i} < 2^{8k}$$

(i.e., interpret the m_is as the coefficients in the base 2^8 expansion of an integer z_0). Thus we want to choose k as large as possible so that $2^{8k} \leq 2^n$. (What does this say about the choice of n?) Then use the next k message bytes to construct z_1, and so on. Of course the result will be very large integers that may not be directly representable in your favorite programming language (see Section 7).

You will also have to reverse this process to decode the message: given an integer z_i with $0 \leq z_i < 2^{8k}$ (how do you know this holds for the decrypted cyphertext?), find integers m_i, $i = 0, \cdots, k-1$ so that $0 \leq m_i < 2^8$ and $z_i = \sum_{i=0}^{k-1} m_i 2^{8i}$.

5 Public and Private Key Generation

Write a module that inputs an integer n and:

1. Finds p and g, where p is an n bit prime and g is a primitive root modulo p (use the module from Section 3);
2. Randomly chooses x with $1 \leq x < p$ and computes $h = g^x \pmod{p}$;
3. Returns the public key $k_1 = (p, g, h)$ and the private key $k_2 = (p, g, x)$.

6 ElGamal

Implement 3 modules that make up ElGamal. The modules will communicate with each other by writing to files.

Key setup: Generate a public/private key pair as in Section 5. Write the public key to a file named $K1$ and write the private key to a file named $K2$.

Encryption: Read $K1$ to obtain p, g, and h. The message will be a series of bytes. Convert it into a series of integers $z_0, z_1, \cdots$ with $0 \leq z_i < p$ (see Section 4). Encrypt each z_i as follows.

1. Choose a random $y \in \{0, 1, \cdots, p-1\}$.
2. Compute $(c_i, d_i) = (g^y \pmod{p}, z_i h^y \pmod{p})$.

Write the result to a file named Cipher. Cipher will contain a list of pairs of integers, each integer in $\{0, 1, \cdots, p-1\}$. I suggest formatting the file with one integer pair per line so that it is readable.

Decryption: Read $K2$ to obtain p, g, and x. Read Cipher to obtain $(c_0, d_0), (c_1, d_1), \cdots$. Decrypt each cipher pair (c_i, d_i) as follows.

1. Compute $s = c_i^x \pmod{p}$.
2. Compute $z_i = d_i s^{-1} \pmod{p}$. Recall that if $\gcd(s, p) = 1$, then $s^{p-1} \equiv 1 \pmod{p}$, so $s^{-1} \equiv s^{p-2} \pmod{p}$.

Reconstruct the message from the z_i. Write the result to a file named Plaintext.

7 Programming Language

You may use any standard general purpose programming language, such as C, C++, Java, Python, etc. You may **not** use Matlab. You may **not** use any library routines to implement the various operations described above (except as described in items 1b and 1c of Section 1). You may **not** look up source code for any components of this project on the web or elsewhere. Also, if you are unable to make your program work, then it will be harder for me to give you partial credit if you choose an obscure programming language with which I am unfamiliar.

With ElGamal it is important to think about the size of integers. If we only use integers that can be maintained in 32 bit words, then we must limit ourselves to 16 bit unsigned integers so that we can multiply without overflow. Then the keys are too small — it's trivial to search for x with $h \equiv g^x \pmod{p}$. To make ElGamal practical, we need a way to do modular arithmetic with large integers, at least as large as 2^{500}. But most programming languages do not have large precision integers as a data type. One exception is Python, which does have arbitrarily large integers. However, python is a poor choice because it is interpreted (rather than compiled), so it's not very efficient. I ran a test of an arithmetic-intensive program implemented in Python and C: on my Mac Air the Python version took about 50 times as long to execute. Other languages have libraries that implement arbitrary precision integer arithmetic. For example, PARI/GP, found at http://pari.math.u-bordeaux.fr/, is a C library. GMP is an open source C library that runs on many Unix-like systems, available at http://gmplib.org/ Descriptions of a few other packages can be found at http://orion.math.iastate.edu/cbergman/crypto/bignums.html.

You have three choices:

(a) Pick a language that does not have arbitrary precision integers, and implement arbitrary precision integers (maximum grade 100 out of 100).

(b) Pick a language that does not have arbitrary precision integers, and learn to use a large integer library (maximum grade 90 out of 100).

(c) Pick a programming language that has arbitrary precision integers, such as Python (maximum grade 80 out of 100).

8 Implementing Big Integers

If you choose to implement your own package of arbitrary precision integers (called *bigints* here), here are some suggestions.

A bigint should be a linked list (singly or doubly linked – think carefully about this) of ordinary integers which I will call "coefficients". The idea is to represent an integer x in base N for some N by storing a list $(a_0, a_1, \cdots, a_{m-1})$ where $x = \sum_{i=0}^{m-1} a_i N^i$. You probably want the list length m stored in the header node. Assuming a 32 bit architecture, you would like the coefficients to be 32 bits (i.e., $N = 2^{32}$), this causes a problem with overflow when you add or multiply bigints. One solution is to sacrifice memory and use only 16 bits for a coefficient. I.e., let $N = 2^n$ where $n = 16$, so $0 \leq$ coefficient $< 2^{16}$.

Note that since you will only be doing modular arithmetic, you can assume all bigints are nonnegative.

You will need to implement at least the following operations.

badd(x, y, z)**:** Compute $x = y + z$.

bsub(x, y, z)**:** Compute $x = y - z$. You might be able to assume $y \geq z$.

bmul(x, y, z)**:** Compute $x = y * z$.

bdiv(q, r, y, z)**:** Compute q, r so that $y = qz + r$ and $0 \leq r < z$.

bequal(x, y)**:** True iff $x = y$.

bless(x, y)**:** True iff $x < y$.

bshift(x, k)**:** Shift the coefficient sequence k places. Ie, integer divide by $N^{|k|}$ if $k < 0$. Multiply by N^k if $k > 0$. You should not just use bdiv( ) and bmul( ).

binit(x, a)**:** Create a new big integer x with value a, where $0 <= a < N$.

binitstring(x, s)**:** Create a new big integer x whose value is given by the string s of digits.

bprintf(f, x)**:** Print bigint x to file f. The easiest way to do this is to print in hexadecimal, since each 16 bit coefficient yields 4 hexadecimal digits.

brand(x, m)**:** Generate a pseudorandom bigint x with m bits.

You will also need modular versions of badd, bsub, and bmul. E.g., bmodadd(x, y, z, m) computes $x = y + z \pmod{m}$. You may also want the following utilities:

bexpand(x, m)**:** Increase the number of coefficients in x to at least m by adding leading zeros if needed.

bshrink(x)**:** Delete all leading zeros from x.

bfree(x)**:** Release all memory used by x.

bcopy(x, y)**:** Set $x = y$.

9 Documentation

Your program must be sufficiently documented that I can make sense out of your code. If I can't make sense of it, it will not receive a good grade. If your initial submission is not adequately documented, I will notify you that it is rejected. You will then have one week to add documentation and resubmit the project, but your maximum possible grade will be lowered by 10 points.

Documentation must include:

1. A narrative describing the overall purpose of the project with a list of all the major components and how they fit together. This should include an explanation of any problems you encountered and how you overcame them. Describe what you did to verify the correctness of your program. If you failed to get the program to work properly, explain what parts failed and how they failed and describe what parts you believe are working correctly.

2. In each module, an explanation of the functionality of the module (what is the input, how is the output related to the input).

3. A description in comments of all variables (e.g., "int i; /* loop counter */").

4. A description in comments of anything complicated (e.g., how you chose to encode bytes as integers).

10 What to Hand In

Projects should be turned in by sending me a single tar file. This file should contain:

- The narrative documentation described in Section 9.
- The source code for all modules implemented, thoroughly documented.
- The results of a test run. Use the message:

  > You can trust some of the people all of the time. You can trust all of the people some of the time. But you can't trust all of the people all of the time.

  as plaintext. Use $n = 200$ and $t = 100$ as parameters for the global setup.

  The results of the test run should be demonstrated by including the four files $K1$, $K2$, Cipher, and Plaintext, as described in Section 6.

You tar file must be e-mailed to me by 2:00 PM, April 24.