
Problem Set 3

This problem set is due on *Wednesday, March 26, 2008*. You must submit your homework to your edlab CS591d course directory. Make sure that your homework files have proper permissions set such that the course staff can read the files, but not other students. Do not make any modifications to your homework files after the due date, unless you notify the TA that you are using a late pass. You may write your solution in any program (Word, L^AT_EX, etc.), but you must convert your document to PDF for submission. We will accept only PDFs. Please contact the TA well ahead of the deadline if you have a question about these procedures.

This is an individual problem set. See Handout 1 (*Course Information*) for our policy on collaboration and late penalties. Mark the top of each page with your name, cs591d, the problem set number and question, and the date. Each solution must begin on a new page. Points may be deducted if your TA has problems understanding your solution. In mathematical problems, **show all your work**.

Each team member should turn in the individual problem on his or her own. For the group problems, have your team submit a *single solution*.

In this problem set we will cover both (1) the internals of implementing number theoretic routines (e.g., primality testing) with low-level programming, and (2) the methods of implementing higher-level semantics with a programming language containing built-in number theoretic primitives.

Some of the problems require extensive programming. We advise you to start early on these problems! The numbers you will be working with are quite large – plain old `long` or `integer` types won't be large enough. Java, Python, C++, C, and Scheme offer reasonable libraries for cryptography. A popular C/C++ software library supporting multi-precision arithmetic is GMP (GNU Multi-Precision Package) from www.swox.com/gmp. Java has built-in support for large numbers in its `BigInteger` class. Python at www.python.org has built-in support for large numbers too. For the die hard fans of first-class languages, Scheme from www.swiss.ai.mit.edu/projects/scheme/ has built-in large number support. We leave the choice of the language to your personal preferences. We suggest Java as the simplest language if you are unable to decide in bounded time. But you will be responsible for debugging your own software.

Some problems also involve a number theoretic package called SAGE. You can either use SAGE directly on the Edlab machines (type `sage`) or you could install it on your own computer. See <http://www.sagemath.org/> for more information.

Each team member should turn in the individual problem on his or her own. For the group problems, have your team submit a *single solution*.

Problem 3-1. RSA and ElGamal

This is an individual problem.

1. Do exercise 5.14 from Stinson
2. Do exercise 5.16 from Stinson

For the following exercises assume that $E_e(x)$ is the RSA encryption of message x via the public key e .

1. Show that RSA is a commutative encryption scheme i.e. show that $E_{e_1}(E_{e_2}(x)) = E_{e_2}(E_{e_1}(x))$.
2. Show that RSA is a multiplicative homomorphic encryption scheme i.e. show that $E_e(x) * E_e(y) = E_e(x * y)$.
3. Is ElGamal a commutative encryption scheme? Is it multiplicative homomorphic? When answering yes, provide a proof. When answering no, provide a clear counter example.

Problem 3-2. Primes (4 pts)

This is a group problem. Divide up the work amongst your team in an efficient manner. But **all** team members must contribute in some way. Each team member should list exactly what role you played. A couple sentences per team member is sufficient. If you receive any key insights from another person or resource, cite that person or resource.

(a)

Write an efficient program (i.e. probabilistic polynomial time) that takes as input a string s and produces a number likely to be a prime and whose high order bytes are s . The input s is a byte string, and the output number should be printed in decimal. Use the primality testing techniques discussed in lecture (FLT and non-trivial square roots of 1).

For example, the string “kevinfu+rlychev” is

```
0x6B 65 76 69 6E 66 75 2B 72 6C 79 63 68 65 76 00 00
```

when each ASCII character is converted to its hexadecimal representation¹ and two zero bytes are appended. The two extra zero bytes provide the flexibility needed to modify this number into a prime without having to change the original message. In decimal form, our starting number is 36545080117543814583894181840954890518528. By flipping some of the bits in the last two bytes of this number, we can generate the following number likely to be prime:

```
0x6B 65 76 69 6E 66 75 2B 72 6C 79 63 68 65 76 00 21
```

which is 36545080117543814583894181840954890518561 in decimal.

You may use any programming language to implement this program. However, you CANNOT use any built-in methods for primality testing (e.g., `java.math.BigInteger.isProbablePrime()`). You must implement your own primality testing code. We recommend that you either use Java or C. Java comes with a package for arbitrarily large integers (`java.math.BigInteger`). You might find the GNU Multi-Precision (GMP) library useful for programming in C². Submit the source code to your program. We have provided a minimal framework in Java to get you started:

¹Type “man ascii” for help on Unix

²<http://www.swox.com/gmp/>

```
import java.math.*;
import java.io.*;
import java.util.*;

public class FindPrime
{
    public static void main(String args[])
    {
        if (args.length != 1) {
            System.out.println("Usage: java FindPrime <msg>");
            System.exit(1);
        }

        String msg = args[0];
        byte[] m = msg.getBytes();
        BigInteger big = FindPrime.findPrime(m);
        System.out.println(big.toString());
    }

    public static BigInteger findPrime(byte[] m)
    { /* your code here */ }

    public static boolean isPrime(BigInteger b)
    { /* your code here */ }
}
```

(b) Use your program to encode the email addresses of your group in a number that is likely prime. In your submitted solution, include the output so that we can verify your number is correct.

If you want to have some fun on your own time, write a program that outputs a prime that when decoded is the source code of the program that creates the prime. This extra program is completely optional and just for fun, not to be graded.

Problem 3-3. ElGamal (3 pts)

This is a group problem. Divide up the work amongst your team in an efficient manner. But **all** team members must contribute in some way. Each team member should list exactly what role you played. A couple sentences per team member is sufficient. If you receive any key insights from another person or resource, cite that person or resource.

Below is one method to sign a numeric message m with ElGamal. See page 282 of Stinson for a general explanation of the properties of cryptographic signatures.

Key generation:

1. Generate prime p and generator g of Z_p^*
2. Randomly select an exponent x s.t. $x \in_R Z_p^*$
3. Let $y = g^x \bmod p$
4. Public key is (g, p, y)
5. Private key is (x)

Sign (m, x, g, p, y) :

1. Pick a random k s.t. $\gcd(k, p-1) = 1$. That is, $k \in_R Z_{p-1}^* = Z_{p'}^*$ where $p = 2p' + 1$
2. Let $r = g^k \bmod p$
3. Let $s = k^{-1}(m - rx) \bmod (p-1)$
4. Signature of m is (r, s)

Verify (m, r, s, g, p, y)

1. Verify $y^r r^s \equiv g^m \bmod p$

Why is this correct? Expand in terms of g .

$$y^r = g^{rx}, r^s = g^{ks}$$

So the equation becomes

$$g^{rx} g^{ks} \equiv g^m \bmod p$$

Since exponents have the same base g , we can add them up

$$g^{rx+ks} \equiv g^m \bmod p$$

We know that this is true iff

$$rx + ks \equiv m \bmod \phi(p)$$

Solve for s

$$s \equiv k^{-1}(m - rx) \bmod (p-1)$$

This is precisely what the signature algorithm defined for s , so the verification is correct.

Below is SAGE code that implements rudimentary ElGamal encryption and decryption with hard-coded numbers:

```
p = 7919
P = IntegerModRing (p)

alpha = 686
secretkey = 845
beta = P(alpha^secretkey)
randomk = 32

message = 55

y1 = P(alpha^randomk)
y2 = P(message*beta^randomk)

y2, P(y2/(y1^secretkey))

(1544, 55)
```

(a) ElGamal signatures

Now that you have seen example SAGE code, write the appropriate SAGE code to (1) create an ElGamal key pair, (2) create an ElGamal signature, and (3) verify an ElGamal signature. Submit your pretty-printed code along with a sample execution.

(b) Misuse of ElGamal

Reusing the random k value for multiple signatures is very dangerous. If an adversary receives two messages with the same k reused, then the adversary can compute a signature on a new message.

We will email you two ElGamal signatures on distinct messages with a reused k . Create an ElGamal signature on a creative new message in decimal, and explain how the attack works.

Problem 3-4. Hashing (3 pts)

This is an individual problem. As customary, if you receive any key insights from someone else or some other resource, you must cite that person or resource.

Do problem 4.6 from Stinson on page 157.