Zheng Ma *April 2, 2005*

Solutions to Problem Set 4

Problem 11, 12, 13 count 10 points each. Problem 14 counts 20 points.

Problem 11: Primitive roots

Does the modulus $n = pq$ in RSA have primitive roots? Explain why or why not.

Solution: The answer is NO. Here are two proofs:

Proof 1: Trivially, by Gauss's theorem, the numbers having primitive roots are $1, 2, 4, p^k, 2p^k$, where p is an odd prime and $k \geq 1$. However, in RSA, we need to choose two large distinct primes p and q, so the resulting number $n = pq$ is not in above set and hence does not have primitive roots.

Proof 2: Here we can prove our result by showing a contradiction. (Avoid using previous method which is kind of so called "proof by intimidation"[1])

Suppose we have $n = pq$, where p and q are large prime numbers. Suppose $\mathbf{Z}_n^*$ has a primitive root r. Then $\gcd(r, n) = 1$ and $\mathrm{ord}(r) = \phi(n)$ by definition. Since $\gcd(r, n) = 1$ then $\gcd(r, p) = 1$, and $\gcd(r, q) = 1$. By Euler's theorem, we know that $r^{\phi(p)} \equiv 1 \pmod{p}$ and $r^{\phi(q)} \equiv 1 \pmod{q}$. Let u be the Least Common Multiple of $\phi(p)$ and $\phi(q)$. Because $\phi(p) = p - 1$ and $\phi(q) = q - 1$ are even numbers, they share a common factor of 2, so $u < \phi(p) \cdot \phi(q)$. Also, since $\phi(p) \mid u$ and $\phi(q) \mid u$, we have $r^u \equiv 1 \pmod{p}$ and $r^u \equiv 1 \pmod{q}$, which means $r^u \equiv 1 \pmod{n}$. So $\mathrm{ord}(r) \leq u < \phi(p) \cdot \phi(q) = \phi(n)$, contradicting the assumption that r is a primitive root. Hence, there is no such r in $\mathbf{Z}_n^*$.

Problem 12: Quadratic Residues

Problem 6.13 in the textbook:

> "Let $n = pq$ with p and q being distinct primes. Under what condition $-1 \in \mathrm{QR}_n$? Under what condition $\left(\frac{-1}{n}\right) = -1$?"

Solution: First, we should notice that one of p and q can be 2. That's what a lot of people missed in their answers. Second, the perfect answers we are looking for are the conditions on n, i.e., the properties of p and q that make the statements true.

(a) $-1 \in \mathrm{QR}_n$: For distinct primes p and q, a number is a quadratic residue modulo $n = pq$ iff it is a quadratic residue modulo both p and q. Hence, $-1 \in \mathrm{QR}_n$ iff both $-1 \in \mathrm{QR}_p$ and $-1 \in \mathrm{QR}_q$.

We now characterize those primes p for which $-1 \in \mathrm{QR}_p$.

Case 1: $p = 2$. Then $-1 \in \mathrm{QR}_p$ since $-1 \equiv 1 \equiv 1^2 \pmod{2}$.

Case 2: p is odd. By the Euler Criterion, $-1 \in \mathrm{QR}_p$ iff $(-1)^{\frac{p-1}{2}} \equiv 1 \pmod{p}$. This holds iff $\frac{p-1}{2}$ is even. Hence, $-1 \in \mathrm{QR}_p$ iff $p \equiv 1 \pmod{4}$.

[1] http://www.ling.ed.ac.uk/ heycock/proof.html

It follows from these two cases that $-1 \in \mathrm{QR}_p$ iff $p \not\equiv 3 \pmod 4$. Using similar reasoning, $-1 \in \mathrm{QR}_q$ iff $q \not\equiv 3 \pmod 4$. We conclude that $-1 \in \mathrm{QR}_n$ iff $p \not\equiv 3 \pmod 4$ and $q \not\equiv 3 \pmod 4$.

(b) $\left(\frac{-1}{n}\right) = -1$: The Jacobi symbol $\left(\frac{-1}{n}\right)$ is only defined for odd n; hence, $n = pq$ must be odd, so both p and q must be odd. In that case, $\left(\frac{-1}{n}\right) = \left(\frac{-1}{p}\right) \cdot \left(\frac{-1}{q}\right)$ by definition. Since

$$\left(\frac{-1}{p}\right), \left(\frac{-1}{q}\right) \in \{-1, 1\},$$

it follows that $\left(\frac{-1}{n}\right) = -1$ iff one of $\left(\frac{-1}{p}\right), \left(\frac{-1}{q}\right)$ is 1 and the other is -1. From the definition of the Legendre symbol and part (a) above, we see for odd primes p that

$$\left(\frac{-1}{p}\right) = 1 \text{ iff } -1 \in \mathrm{QR}_p \text{ iff } p \equiv 1 \pmod 4,$$

and

$$\left(\frac{-1}{p}\right) = -1 \text{ iff } p \equiv 3 \pmod 4.$$

Hence, $\left(\frac{-1}{n}\right) = -1$ iff one of p, q is congruent to 1 modulo 4 and the other is congruent to 3 modulo 4.

Problem 13: Generators over $\mathbf{Z}_p^*$

Prove or disprove the following: Let g, h be generators for $\mathbf{Z}_p^*$, where p is an odd prime. Suppose $x \equiv g^{2u} \equiv h^{2v} \pmod p$. Then $g^u \equiv h^v \pmod p$.

Solution: The statement is false. We can see it from the following counter-example: When $p = 5$, then $g = 2$ and $h = 3$ are generators. Let $v = 3$. Then $g^{2v} \equiv h^{2v} \equiv 4 \pmod p$. However, we can see that $g^v \equiv 3 \pmod p$ and $h^v = 2 \pmod p$.

The following argument gives some insight into what is going on for arbitrary odd primes p: If $g^{2v} \equiv h^{2v} \pmod p$ then

$$g^{2v} - h^{2v} \equiv (g^v - h^v)(g^v + h^v) \equiv 0 \pmod p.$$

So we can see $g^v - h^v \equiv 0 \pmod p$ or $g^v + h^v \equiv 0 \pmod p$. The first of these possibilities implies the desired conclusion, $g^v \equiv h^v \pmod p$, but the other does not. It can happen (as shown above) that $g^v + h^v \equiv 0 \pmod p$ but $g^v - h^v \not\equiv 0 \pmod p$. That's why the original statement fails.

Problem 14: DSA Signature Scheme Implementation

In this problem, you will implement the key generation, signature, and verification functions of the DSA Signature Scheme. (See lecture notes week 8.) In particular, you should write the following three programs that take the indicated command line arguments:

dsa_key <*keyfile*>

This program generates random DSA parameters p, g, and x. The primes p and q should be exactly 1024 bits long and 160 bits long, respectively. Next it computes a. It then creates two files named `<keyfile>.pub` and `<keyfile>.prv`, where <*keyfile*> is the command line argument. It writes p, g, and a to `<keyfile>.pub`, in decimal, one number per line. It writes p, g, and x to `<keyfile>.prv`, also one decimal number per line. The lines should contain only digits (no labels) so that the key file may be easily read back by another program.

`dsa_sign <keyfile> <msgfile>`
Reads p, g, and x from the private key file, `<keyfile>.prv`. Reads a decimal integer $m \in Z_p$ (the message) from `<msgfile>`. Signs m using the DSA signature scheme to produce a signature $s = (b, c)$, which it writes to `<msgifle>.sig` as two decimal numbers, one per line.

`dsa_verify <keyfile> <msgfile>`
Reads p, g, and a from the public key file, `<keyfile>.pub`. Reads m from `<msgfile>`. Reads the signature $s = (b, c)$ from `<msgifle>.sig`. Checks the signature of m using DSA signature verification and writes ”good” or ”bad” to standard output accordingly.

(continued on next page)

You may use either the GMP or ln3 bignum package, as well as any of the functions that they provide.

The hardest part of this assignment is probably the generation of large random primes of the desired length. The GMP functions `mpz_random()` and `mpz_random2()` are obsolete and should not be used. Use `mpz_urandomb()` or `mpz_urandomn()` instead. In ln3, you might use either `Random()` or `RandomPrime()`, but note that the desired size arguments to both of them is in terms of decimal digits rather than bits. If you'd prefer, you can call the standard Unix random number generator `random()` repeatedly and shift the bits in to form a large number. Note that `random()` returns 31 bits, not 32.

Of course, code for DSA signatures is undoubtedly available from various places on the web. You're welcome to read it but not to copy it. In any case, I expect you to use the routines in the GMP or ln3 packages rather than solve the problem in some other way. Start early on this assignment, and feel free to ask me for help if there are things you don't understand. This is not a difficult assignment, but whenever you program, there are bound to be hidden pitfalls.

Solution:

I use the following criteria in grading the programming assignment:

```
-------------------------------------------------------------
| I used the following criteria in grading this assignment: |
| Grade of 20 is broken down as:                            |
|                                                           |
| 1. Key generation (8 points)                              |
| 2. Signing (6 points)                                     |
| 3. Verification (6 points)                                |
|                                                           |
| Things I looked for:                                      |
| - large random primes are generated correctly             |
| - signing/verification produces consistent answers        |
| - msgfile and keyfile are in correct format               |
| - source code is written in a coherent way and documented |
| - DSA signature is implemented correctly                  |
-------------------------------------------------------------
```

The following programs are due to Melody Chan:

dsa_key.cc

```
/* dsa_key.cc: Generating random DSA keys */

#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <ctype.h>
#include <string.h>

#include <lnv3/lnv3.h>
#include <nttl/isPrime.h>

/* Generate a random long integer of n bits */
```

```
ln random_ln (int n) {

  int i;
  ln r, bit;

  if (n <=0) exit(1);

  /* The leading bit must be 1. */
  r = "1";

  /* Generate rightmost n-1 bits */
  for (i = 0; i < n-1; i++) {
    /* A hack to get a random bit */
    bit = ln().Random(1) % 2;
    r = 2 * r + bit;
  }

  return r;

}

/* Generate a random long number mod m */
ln random_mod (ln m) {

  ln r;
  r = ln().Random(m.NumBits()) % m;
  /* the random number is therefore chosen from a range that
   is a constant factor bigger than m.  this is what i want */
  return r;

}


int main (int argc, char* argv[]) {

  ln p, q, g, x, a;

  ln lower, upper;
  ln z, zmin, zmax, zrange, rem;
  ln h;

  char* pubkey;
  char* prvkey;
  char buffer[2000];

  /* Check command line */
  if (argc != 2) {
    printf("Please specify a filename in the command line\n");
```

```
    exit(1);
  }

  pubkey = new char[strlen(argv[1])+5];
  prvkey = new char[strlen(argv[1])+5];
  strcpy(pubkey, argv[1]);
  strcpy(prvkey, argv[1]);
  strcat(pubkey, ".pub");
  strcat(prvkey, ".prv");

  ofstream pubfile;
  pubfile.open (pubkey, ios::out);
  ofstream prvfile;
  prvfile.open (prvkey, ios::out);

  /* lower = 2^1023 - 1, upper = 2^1024 - 1. a hack */
  lower = "8988465674311579538646525953945123668089884894711532
  8636715040578866337902750481566354238661203768010560056939935
  6966788293948844072083112464237153197370621888839467124327426
  3815110980062304705972654147604250288441907534117123144073695
  6555270413618581675255342293149119973622969239858152417678164
  812112068607";
  upper = 2 * lower + 1;

  /* q is a random 160-bit prime */
  while (!IsPrime(q = random_ln(160), 20));


  /* p is a 1024-bit prime in the form zq+1 */
  /* compute bounds for z */
  lower.Divide (q, &zmin, &rem);
  if (rem != 0) zmin++;
  upper.Divide (q, &zmax, &rem);
  if (rem == 0) zmax--;
  /* find p */
  zrange = zmax-zmin;
  do {
    z = zmin + random_mod(zrange+1);
    } while (!IsPrime((p = z * q + 1), 20));
  /* now z = (p-1)/q. */


  /* Find random h in Z*_p and compute g = h^z, */
  /* such that g is a z^th root of unity. */
  do {
    h = random_mod(p);
  } while ((g = h.FastExp(z,p)) <= 1);
```

```
  /* x in Z*_q */
  while ((x = random_mod(q)) == 0);


  /* a is g^x mod p */
  a = g.FastExp(x,p);

  /* Write to files */
  pubfile<<p<<endl<<q<<endl<<g<<endl<<a<<endl;
  prvfile<<p<<endl<<q<<endl<<g<<endl<<x<<endl;
  pubfile.close();
  prvfile.close();

  return 0;

}
```

dsa_sign.cc

```
/* dsa_sign.cc Sign a message */

#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <ctype.h>
#include <string.h>

#include <lnv3/lnv3.h>
#include <nttl/isPrime.h>
#include <nttl/inverse.h>


int main (int argc, char* argv[]) {

  ln p, q, g, x, m, b, c, y;

  if (argc != 3) {
    cout<<"Specify a keyfile and message file"<<endl;
    exit(1);
  }

  /* Read keyfile */
  char *keyfilename = new char[strlen(argv[1])+5];
  strcpy(keyfilename, argv[1]);
  strcat(keyfilename, ".prv");

  ifstream keyfile;
```

```
  keyfile.open(keyfilename, ios::in);
  if (!keyfile.is_open()) {
    cout<<"Invalid keyfile"<<endl;
    exit(1);
  }
  keyfile >> p >> q >> g >> x;
  keyfile.close();


  /* Read message file */
  char* msgfilename = new char[strlen(argv[2])+5];
  strcpy(msgfilename, argv[2]);

  ifstream msgfile;
  msgfile.open(msgfilename, ios::in);
  if (!msgfile.is_open()) {
    cout<<"Invalid message file" << endl;
    exit(1);
  }
  msgfile >> m;
  msgfile.close();

  /* Choose y in Z^*_q */
  while ((y = ln().Random(q.NumBits()) % q) == 0);

  /* Compute b and c */
  b = g.FastExp(y,p) % q;
  c = (m + x * b)*Inverse(y,q) % q;

  /* Write to msfile.sig */
  char* msignedname = new char[strlen(msgfilename)+1];
  ofstream msigned;
  strcpy(msignedname, msgfilename);
  strcat(msignedname, ".sig");
  msigned.open(msignedname, ios::out);
  if (!msigned.is_open()) {
    cout<<"Could not open" << msignedname <<endl;
    exit(1);
  }

  msigned << b << endl << c << endl;
  msigned.close();

  return 0;
}
```

dsa_verify.cc:

```
/* dsa_verify.cc Verify a signature */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <ctype.h>
#include <string.h>

#include <lnv3/lnv3.h>
#include <nttl/isPrime.h>
#include <nttl/inverse.h>


/* If bad */
void bad (void) {

  cout<<"bad";
  exit (0);

}


int main (int argc, char* argv[]) {

  ln p, q, g, a, m, b, c, u1, u2, v;

  if (argc != 3) {
    cout<<"Specify a keyfile and message file"<<endl;
    exit(1);
  }

  /* Read keyfile */
  char *keyfilename = new char[strlen(argv[1])+5];
  strcpy(keyfilename, argv[1]);
  strcat(keyfilename, ".pub");

  ifstream keyfile;
  keyfile.open(keyfilename, ios::in);
  if (!keyfile.is_open()) {
    cout<<"Invalid keyfile"<<endl;
    exit(1);
  }
  keyfile >> p >> q >> g >> a;
  keyfile.close();


  /* Read message file */
  char* msgfilename = new char[strlen(argv[2])+5];
```

```
  strcpy(msgfilename, argv[2]);

  ifstream msgfile;
  msgfile.open(msgfilename, ios::in);
  if (!msgfile.is_open()) {
    cout<<"Invalid message file" << endl;
    exit(1);
  }
  msgfile >> m;
  msgfile.close();

  /* Read signature */
  char* msignedname = new char[strlen(msgfilename)+1];
  ifstream msigned;
  strcpy(msignedname, msgfilename);
  strcat(msignedname, ".sig");
  msigned.open(msignedname, ios::in);
  if (!msigned.is_open()) {
    cout<<"Invalid signature file"<<endl;
    exit(1);
  }
  msigned >> b >> c;
  msigned.close();

  /* Verify! */
  if (b <=0 || b >= q) bad();
  if (c <=0 || c >= q) bad();

  u1 = m * Inverse(c,q) % q;
  u2 = b * Inverse(c,q) % q;
  v = (g.FastExp(u1, p) * a.FastExp(u2, p) % p) % q;
  if (v != b) bad();

  cout<<"good";
  return 0;
}
```