YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CPSC 467b: Cryptography and Computer Security Handout #23
Zheng Ma *May 2, 2005*

Solutions to Problem Set 7

Problem 21: (Oblivious Transfer)

Oblivious Transfer is a two party protocol $(A(b), B)$ such that at the end of this protocol one of the following two events occurs, each with probability $1/2$:

(a) B learns the value b.

(b) B gains no information about b beyond what, if anything, B knew about b before the protocol.

At the end of the protocol, B knows which of the two events occurred, and A has no idea which event occurred.

One-out-of-two Oblivious Transfer is a two party protocol $(A(b_0, b_1), B(s))$, such that at the end of this protocol, all of the following three conditions hold:

(a) B learns the value b_s.

(b) B gains no information about b_t beyond what, if anything, B knew about b_t before the protocol, where $t = 1 - s$.

(c) A learns nothing about s.

Here is an implementation of one-out-of-two oblivious transfer using a basic oblivious transfer primitive as a black box. (This protocol is adapted from one in lecture notes by Rafail Ostrovsky, http://www.cs.ucla.edu/~rafail/TEACHING/WINTER-2005/L10/L10.pdf.)

1. Let n be a security parameter, and set $M = 3n$. A chooses a random bit string $r = r_1 r_2 \ldots r_M$ of length M. A uses the basic oblivious transfer protocol M times to transfer r to B, one bit at a time. B learns approximately 1/2 of the bits r_i. Let $I \subseteq \{1, \ldots, M\}$ be the set of indices i for which B does learn r_i.

2. B's input bit is s. B wants to learn A's secret b_s. B chooses a random subset I_s of I of size n and a random subset I_{1-s} of $\{1, \ldots, M\} - I$, also of size n, and sends the sets I_0 and I_1 to A.

3. A checks that I_0, I_1 are disjoint subsets of the right form. A then computes $c_i = b_i \oplus \left(\bigoplus_{j \in I_i} r_j\right)$, for $i = 0, 1$, and sends c_0, c_1 to B.

4. B computes $b_s = c_s \oplus \left(\bigoplus_{j \in I_s} r_j\right)$.

Questions:

(a) This protocol can sometimes fail. Explain how.
Solution: If $|I| < n$ or $|I| > 2n$ in the first step. Thus B can't choose I_s or I_{1-S} in step 2. So the protocol will fail.

(b) The above definition of one-out-of-two oblivious transfer does not allow for failure. Make a minor change to the definition so that it matches what this protocol is actually able to achieve.
Solution: Change the definition to "With the high probability, that the following three condition hold: ..."

(c) Describe why B learns the desired value b_s. Is this always true or only true with high probability?
Solution: Since B learns all elements in I_s, he can calculate b_s as below:

$$b_s = c_s \oplus (\oplus_{j \in I_s} r_j) = b_s \oplus (\oplus_{j \in I_s} r_j) \oplus (\oplus_{j \in I_s} r_j) = b_s \oplus (\oplus_{j \in I_s} (r_j \oplus r_j)) = b_s \qquad (1)$$

Because the protocol might fail with a small probability, the above statement is true with high probability.

(d) Describe why B gains no information about b_{1-s}. Is this always true or only true with high probability?
Solution: B gains no information about any of the elements in I_{1-s}, so in particular, B gains no information about $(\oplus_{j \in I_{1-s}} r_j)$. Hence, B gains no information about b_{1-s}, even given $c_{1-s} = b_{1-s} \oplus (\oplus_{j \in I_{1-s}} r_j)$. So for an honest B, he gains no information no matter whether the protocol succeeds or fails. However, for a cheating B, with small probability that $|I| \geq 2n$, B can learn both b_s and b_{1-s}. So the above statement is true with high probability.

(e) Describe why A learns nothing about s. Is this always true or only true with high probability?
Solution: A learns nothing about s because of the fact that I_0 and I_1 are the same size and both contain only the information of the indices. With the definition of basic oblivious transfer, A has no idea whether B learns the bit value or not. This is always true.

(f) Describe why a cheating B cannot learn both b_0 and b_1. Is this always true or only true with high probability?
Solution: If B knows more than $2n$ bits, he can cheat and send both sets as disjoint subsets of the bits he knows. Then he can learn both bits. This happens with small probability that $|I| > 2n$. If B doesn't know more than $2n$ bits he can't send any two disjoint sets because he will not know enough bits to do so. So the above statement is true with high probability.

(g) Why does Alice need to check I_0 and I_1 in step (3)? Explain how B could cheat if she failed to do so.
Solution: Otherwise, B could send $I_s = I_{1-s} \subseteq I$ and learn both bits.

(h) Does the protocol still work if M is defined to be $2n$ instead of $3n$? Defined to be $5n$ instead of $3n$? Explain.
Solution: If M is defined as $2n$, then the protocol will fail when $|I| \neq n$. Because the probability that $|I| \neq n$ is very high when $M = 2n$, the protocol can't work in this case.

If M is defined as $5n$, the protocol may still work. However, the probability of B being able to cheat is big, more than $1/2$. This is B learns more than $2n$ bits from the oblivious transfer with high probability. So from the modified definition of the One-out-of-two Oblivious Transfer, this is still a failed solution.

The next two problems concern the Blum-Blum-Shub pseudorandom sequence generator. See Handout 18 for the exact definitions assumed by these problems.

Problem 22: (BBS Pseudorandom Sequence Generator)

Write a C function to implement the Blum-Blum-Shub pseudorandom sequence generator. You can assume the inputs to your programs are numbers at most 15 bits long (so they are short enough to fit into a variable of type `short int`).

Your function should have the prototype

```
short int bbs_random( short int len,
                      short int buf[],
                      short int seed,
                      short int n );
```

`buf` is assumed to be a buffer of length `len`, `seed` is the seed (starting value) for the BBS generator, and `n` is the modulus for the BBS generator. You may assume that `seed` is in $\mathbf{Z}_n^*$ and that `n` is a Blum integer. A call to `bbs_random()` generates `len` pseudorandom bits and places them in `buf[0]`, ..., `buf[len-1]`, one bit per array element. The new seed is returned.

To test your function, write a command `bbs` that calls `bbs_random()`. The command line "`bbs len seed n`" generates `len` bits starting from seed `seed` and modulus `n` and prints three lines of output. The first line echos the command line arguments. The second contains the pseudorandom bit sequence, printed as a sequence of 0's and 1's with no intervening spaces. The third contains the new seed, printed in decimal.

Run your command on the arguments `80 3 13589`. (Note that $13589 = 107 \times 127$ is a Blum integer.) Write your answers to a file called `bbsout.txt` and submit both the program and the answers file.

Solution: Your output should be the following.

```
80 3 13589
11101110111100111011111110000100011110110100100011110100000010101101001110000101
7955
```

A possible [slightly corny] implementation is the following.

```
#include <stdio.h>
#include <stdlib.h>

#define a for (i=0; i<len;i++){printf ("%1d",buf[i]);}
#define putarg(i,into) sscanf(argv[i],"%d",into)
#define P putarg(1, &len);
#define p putarg(2, &seed);
#define Y putarg(3, &n);
#define usage "bbs_15 <len> <seed> <n>"
#define L printf("\n%d\n",seed);
#define A if (argc != 4){printf("Usage:%s\n",usage);exit(1);}
#define H int len,seed,n,*buf,i;
#define F buf = (int*)malloc(len * sizeof(int));
#define S free(buf); exit(0);
```

```
#define N printf("%s %s %s\n", argv[1],argv[2],argv[3]);
#define I seed =  bbs_random(len,buf,seed,n);

int bbs_random(int len, int buf[], int seed, int n){
 int i;

 for (i=0; i<len; i++){
   seed=(seed*seed)%n; buf[i] = seed%2 ;
 }
 return seed;
}

int main( int argc , char *argv[]){
  H A P p Y   F I N a L S
}
```

Problem 23: (Cycle Lengths)

The purpose of this problem is to explore the cycle lengths of the various possible seeds in the BBS generator of problem 22. For any seed $s_0 \in \mathbf{Z}_n^*$, define the *cycle length* of s_0 to be $k-1$, where k is the least integer > 1 such that $s_k = s_1$ in the BBS-generated sequence $s_1, s_2, s_3, \ldots$, where $s_i = s_{i-1}^2 \bmod n$, for $i = 1, 2, 3, \ldots$.

Questions:

(a) Why is the cycle length well defined for every $s_0 \in \mathbf{Z}_n^*$? That is, why does s_1 occur in the sequence $s_2, s_3, s_4, \ldots$?
Solution: Let's prove it by showing a contradiction. Suppose s_1 doesn't occur in the sequence of $s_2, s_3, s_4, \ldots$. Then there must be some other i $(i \neq 1)$ such that s_i repeats in the sequence. This follows from the fact that the sequence is infinite while the number of quadratic residues in $\mathbf{Z}_n^*$ is finite. Suppose that the first repeated number is s_k $(k \neq 1)$, so the sequence has the form $s_1, \ldots, s_{k-1}, s_k, \ldots, s_l, s_k, \ldots$. Then both s_{k-1} and s_l are the principal square root of s_k since for a Blum integer n, each quadratic residue in Z_n^* has exactly one principal square root. This shows s_{k-1} equals to s_l. So s_k is not the first repeated number in the sequence, a contradiction. So s_1 occurs in the sequence $s_2, s_3, s_4, \ldots$.

(b) What is the expected cycle length when s_0 is chosen uniformly at random from $\mathbf{Z}_n^*$, where $n = 13589 = 107 \times 127$.

For part (b), you should write a program to build a table of quadratic residues and the cycles they lie in. Then compute a table of cycles and their lengths. Finally, compute the expected cycle length. For example, for $n = 33 = 3 \times 11$, there are 5 quadratic residues, so the table of quadratic residues and the table of cycles might look as follows:

x	$(x^2 \bmod 33)$	cycle #
1	1	1
4	16	2
16	25	2
25	31	2
31	4	2

cycle #	length
1	1
2	4

From this table, we see that there are only two cycles: (1) and $(4, 16, 25, 31)$, Of the 20 possible seeds in $\mathbf{Z}_{33}^*$, 4 lead to the first cycle and 16 lead to the second cycle. Hence, the expected cycle length is

$$\frac{4}{20} \times 1 + \frac{16}{20} \times 4 = \frac{68}{20} = 3.4$$

Solution: The expected cycle length is 148.3. Here is one of the implementations. (Thank Melody Chan for letting me use her solution).

```
#include <stdio.h>
#include <stdlib.h>

/* GCD */
int gcd(int a, int b) {

  if (b==0) return a;
  return gcd(b, a%b);

}


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

  int n, i, j, last_cycle, num_qrs;
  int **table;
  int *table2;

  if (argc != 2) {
    printf("usage: cycle n\n");
    exit(1);
  }

  sscanf(argv[1], "%d", &n);

  table = malloc(sizeof(int *) * n);
  /* Row i contains
      i*i (or 0 if i is not relatively prime to n)
      a flag indicating whether i is a QR mod n
      cycle # of i (or 0 if i is not a QR) */

  for (i = 0; i < n; i++) {
    table[i] = malloc(sizeof(int) * 3);

    if (gcd(i, n) == 1) table[i][0] = i * i % n;
    else table[i][0] = 0;

    table[i][1] = 0; /* initially set QR flag to 0 */
```

```
    table[i][2] = 0; /* initially is not part of a cycle */

  }

  /* Mark QRs and count them */
  num_qrs = 0;
  for (i = 0; i < n; i++) {
    if (table[i][0] && (table[table[i][0]][1] == 0)) {
      table[table[i][0]][1] = 1;
      num_qrs++;
    }
  }

  /* Number cycles */
  last_cycle = 0;
  for (i = 0; i < n; i++) {
    if (table[i][1] && (table[i][2] == 0)) {

      table[i][2] = ++last_cycle;
      for (j = table[i][0]; j != i; j = table[j][0])
table[j][2] = last_cycle;

    }
  }

  table2 = malloc(sizeof(int) * (last_cycle + 1));
  /* initialize */
  for (i = 0; i < last_cycle + 1; i++) table2[i] = 0;
  /* count up how many in each cycle */
  /* we will count those in "cycle 0" and then discard that info */
  for (i = 0; i < n; i++) table2[table[i][2]]++;

  /* The expected cycle length is the sum of the squares of the
  lengths divided by the number of quadratic residues */
  j = 0;
  for (i = 1; i < last_cycle + 1; i++) j += table2[i] * table2[i];
  printf("%f\n", (float) j / num_qrs);

  /* Free */
  for (i = 0; i < n; i++) free(table[i]);
  free(table);
  free(table2);

  return 0;

}
```