YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Solution to Problem Set 6

In the problems below, "textbook" refers to *Introduction to Cryptography with Coding Theory: Second Edition* by Trappe and Washington..

Problem 27: Primality Testing

(a) Implement the Miller-Rabin primality testing algorithm so as to handle numbers at least 256 bits long. As usual, your program should be written in C, C++, or Java and should use one of the suggested big number libraries. If your library already contains a probabilistic primality tester, *do not use it* (except for checking) – implement your own instead. But it's okay to use built-in implementations of modular exponentiation, random number generators, and the other arithmetic functions and predicates.

(b) Twin primes are pairs of primes of the form $(p, p+2)$, e.g., $(11, 13)$. (See http://mathworld .wolfram.com/TwinPrimes.html.) Use your program from part (a) to find the smallest $p > 2^{255} + 100$ such that $(p, p+2)$ is a twin prime.

Solution:

part a:

Program P27.java

```
package cs467;

import java.math.BigInteger;
import java.util.Random;

public class P27 {
    /**
     * @param n Number to test compositness on
     * @return true if n is composite
     *            false if not sure
     * This function is written using the same notation
     * as the class book in page 178
     */
    static boolean MillerRabinTest_isComposite(BigInteger n){
        Random r = new Random();
        BigInteger m1 = n.subtract(BigInteger.ONE);
        int k=0;
        while(!m1.testBit(0)){
            k++;
            m1=m1.shiftRight(1);
        }
        BigInteger m = m1;

        BigInteger a = new BigInteger(n.bitLength(),r);
        BigInteger b0 = a.modPow(m,n);
        if (b0.equals(BigInteger.ONE) || b0.equals(n.subtract(BigInteger.ONE)))
            return false;
```

```
        for (int i=0;i<k;i++){
            b0=b0.multiply(b0).mod(n);
            if (b0.equals(BigInteger.ONE)){ // b0 == 1
                System.out.println("We factored n!!!");
                System.out.println(n.gcd(b0.subtract(BigInteger.ONE)));
                return true;
            }else if (b0.equals(n.subtract(BigInteger.ONE))){ // b0==-1
                return false;
            }
        }
        return true;

    }

    public static boolean MR_PrimTest(BigInteger n){
        return !MillerRabinTest_isComposite(n);
    }
    static void solvePart2(){
        BigInteger start = BigInteger.ONE.shiftLeft(255).add(new BigInteger("101"));
        BigInteger two = new BigInteger("2");
        for (int i=1;;i++){
            BigInteger plustwo = start.add(two);
            if (MR_PrimTest(start)){
                if (MR_PrimTest(plustwo)){
                    System.out.println("Found("+i+") "+start);
                    return;
                }else{
                    // I can skip 4 numbers because start+2 is not prime
                    start = plustwo.add(two);
                }
            }else
                start= plustwo;
        }
    }

    public static void main(String[] args) {
        solvePart2();
    }

}
```

part b:

$p = 57896044618658097711785492504343953926634992332820282019728792003956564821237$

Problem 28: ElGamal Variants

Textbook, problem 9.6.4.

Solution:

part a:

Given $s \equiv a^{-1}(m - rk) \pmod{p-1}$ then

$$(\alpha^a)^s r^r \equiv \alpha^{a(a^{-1}(m-rk))} r^r \pmod{p}.$$

Because $aa^{-1} \equiv 1 \pmod{p-1}$ we get

$$\equiv \alpha^{m-kr} r^r \pmod{p}.$$

By definition, r is α^k. Therefore

$$\equiv \alpha^m(\alpha^k)^{-r}r^r \equiv \alpha^m r^{-r}r^r \equiv \alpha^m \pmod{p}.$$

part b:

$$(\alpha^a)^m r^r \equiv (\alpha^a)^m(\alpha^k)^r \equiv \alpha^{am+rk} \equiv \alpha^s \pmod{p}$$

part c:

$$(\alpha^a)^r r^m \equiv \alpha^{ar}\alpha^{mk} \equiv \alpha^{ar+mk} \equiv \alpha^s \pmod{p}$$

Problem 29: Existential Forgery of ElGamal Signatures

Textbook, problem 9.6.5.

Solution:

part a:

$r \equiv \beta^v\alpha^u \pmod{p}$ and $s \equiv -rv^{-1} \pmod{p-1}$. The ElGamal verification tests if $\beta^r r^s \equiv \alpha^m \pmod{p}$. In our case $m = su \pmod{p-1}$. Starting on the left side we get

$$\begin{aligned}
\beta^r r^s &\equiv \beta^r(\beta^v\alpha^u)^s \pmod{p} \\
&\equiv \beta^r\beta^{vs}\alpha^{su} \pmod{p} \\
&\equiv \beta^{r+vs}\alpha^{su} \pmod{p} \\
&\equiv \beta^{r-vrv^{-1}}\alpha^{su} \pmod{p} \\
&\equiv \alpha^{su} \pmod{p} \\
&\equiv \alpha^m \pmod{p}.
\end{aligned}$$

Thus the verification holds.

part b:

To make the forgery work using a hash function we would need to force $h(m) = su \pmod{p-1}$. So forging the signature the implies obtaining m from $h(m)$ that is hard.

Problem 30: Hash Function Based on Squaring

Textbook, problem 8.8.2.

Solution:

part a:

We have seen that finding square roots $\pmod{pq}$ is a hard problem equivalent to factoring pq. In this case finding a pre-image is finding a square root therefore it is supposed to be hard.

part b:

Because x and $-x$ have the same square they map to the same $h(m)$. Thus it is easy to find a colliding pair.

Problem 31: Hash Function Based on Matrices

Textbook, problem 8.8.10.

Solution:

part a:

In both cases we can do the same thing. A pre-image of y will be $y \cdot 0000000000000000000\ldots$ where $\cdot$ is concatenation and $|y| = n$. In the first case this works because the first row in the matrix will be y and all the others will be all zeros. When doing the XOR the result will be y. For the second case this works because the first row is not rotated, and since all other rows are all zeros, rotation does nothing, getting as result the same matrix as in the first case.

part b:

Let x and y be both bit strings of length n. Then $h(x \cdot y) = x \oplus y = h(y \cdot x)$ where $\oplus$ is the bit-by-bit XOR operation. For the second method we need to get smarter. Let x and y be the same as before. Now let $z = y \hookrightarrow 1$. Then $h_2(x \cdot y \cdot z) = x$ so we can generate a second pair y', z' s.t. $z' = y' \hookrightarrow 1$. Now $h_2(x \cdot y' \cdot z') = x$; hence we found a collision.