MCS 360 Project One : Public Key Cryptography with Elliptic Curves due Friday 14 February 2003, at 2PM.

This goal of this project is to explore a public key cryptosystem, based on elliptic curves. This document is made from a Maple worksheet. A good way to start this project is to download the worksheet from the course web site.

1. Elliptic Curves

In this section we explore calculation with points on elliptic curves. We consider all points $(x, y) \in \mathbb{Z}_p \times \mathbb{Z}_p$ which satisfy $y^2 = x^3 + ax + b \bmod p$. The number p must be a prime higher than three and the numbers a and b must have a nonzero discriminant: $4a^3 + 27b^2$ is nonzero modulo p.

```
> a := 3: b := 5: p := 53:
> isprime(p);
                                  true

> discriminant := 4*a^3 + 27*b^2 mod p;

                           discriminant := 41
```

So these values are safe to use.

We define two Maple functions. The first function takes on input a point P$[x, y]$ and returns true if the point belongs to the curve, or false otherwise. For the elliptic curve $y^2 = x^3 + 3x + 5 \bmod 53$, the definition is

```
> f := P -> evalb(P[2]^2 mod 53 = P[1]^3 + 3*P[1] + 5 mod 53):
```

Here is an example on how to use this function

```
> f([1,3]);
                                  true
```

The second function is a bit more involved. For any two points on the curve, we can define the addition

```
> addf := proc(Q,R)
>   description `adds Q and R on y^2 = x^3 + a*x + b mod p`:
>   local a,p,sigma,x3,y3:
>   a := 3:   # coefficient with x of elliptic curve
>   p := 53:  # prime number
>   if Q = O
>    then return R;
>   end if;
>   if R = 0
>    then return Q;
>   end if;
>   if evalb(Q[1] mod p = R[1] mod p)
>    then if evalb(R[2] mod p = -Q[2] mod p)
>          then return(0):
>         end if;
>   end if;
>   if evalb(Q[1] mod p = R[1] mod p)
>     and evalb(Q[2] mod p = R[2] mod p)
>    then sigma := (3*Q[1]^2+a)/(2*Q[2]) mod p;
>    else sigma := (R[2]-Q[2])/(R[1]-Q[1]) mod p;
>   end if;
>   x3 := (sigma^2 - Q[1] - R[1]) mod p:
>   y3 := (sigma*(Q[1] - x3) - Q[2]) mod p:
>   return ([x3,y3]);
> end proc:
```

In the following we show how to generate the group from any point P on the curve:

```
> P := [1,3]; f(P);
                             P := [1, 3]
                                 true
> Q0 := 0; Q1 := P;
> for i from 2 to 52 do
>   Q||i := addf(P,Q||(i-1));
> end do;
                             Q0 := 0
                             Q1 := [1, 3]
                             Q2 := [52, 52]
                             Q3 := [4, 44]
                             Q4 := [11, 37]
                                 .
                                 .
                                 .
                             Q51 := [1, 50]
                             Q52 := 0
```

We have created a “group”. Any group satisfies the following properties:

1. The sum of any two elements in the group is again a member of the group;

2. The sum is associative: (P+Q)+R = P+(Q+R);
3. The origin O is the neutral element, i.e.: P+O=P=O+P
   (observe that O does not correspond to [0,0]);
4. Every element P in the group has a unique inverse Q: P+Q = O = Q + P;
5. We can reverse the order: P+Q=Q+P.

The last property makes the group into a commutative or abelian group.

2. Encrypting and decripting messages

In this section we show how to use elliptic codes in public key cryptography. Take a point P on the elliptic curve and a number n. Calculate Q = n*P. The public key is (P,Q). The secret key is n.

Here is how public key cryptography works. Every one who wants to communicate produces a public key (P,Q) and keeps the secret key n private. The public keys are published like in a phone book. To send a message one uses the public key of the destination. Since only the person who produced the public key (P,Q) knows n, only the receiver (not even the sender!) can decrypt the message.

A message is a point on the curve M = [x,y]. But first we need another auxiliary procedure to multiply a point with a number:

```
> mulf := proc(n,P)
>   description `returns n*P`:
>   local i,Q:
>   Q := P:
>   for i from 1 to n-1 do
>     Q := addf(Q,P):
>   end do;
>   return Q:
> end proc:
```

For example:

```
> mulf(61,[1,3]); f(%);
                                  [32, 50]
                                    true
```

The procedure to encrypt takes the public key and the message:

```
> encrypt := proc(P,Q,M)
>   description `encription of the message M`:
```

```
>   local k,v1,v2:
>   k := 31:  # some random number
>   v1 := mulf(k,P):
>   v2 := mulf(k,Q):
>   v2 := addf(M,v2):
>   return ([v1,v2]):
> end proc:
```

For example (the secret key is n):

```
> P := [1,3]: n := 7: Q := mulf(n,P): M := Q3:
> M; f(M); # message must belong to the curve!

                                  [4, 44]
                                    true

> eM := encrypt(P,Q,M); # encrypted message

                         eM := [[45, 23], [33, 52]]
```

The routine to decrypt uses the secret key:

```
> decrypt := proc(eM)
>   description 'returns an encrypted message':
>   local n,p,inv_n,v1,v2:
>   n := 7:   # secret key
>   p := 53:  # working over Z_p
>   v1 := eM[1]:
>   v1[2] := -v1[2]:
>   v1 := mulf(n,v1):
>   v2 := addf(v1,eM[2]):
>   return v2:
> end proc:
> dM := decrypt(eM); # decrypted message

                              dM := [4, 44]

> evalb(M = dM);     # check if decrypted message is original

                                    true
```

To use the code to encrypt real mesages, we need some utilities. First we transform a string into a list of points on the elliptic curve:

```
> encode := proc(message::string)
>   description ‘returns a numerical encoding of a string‘:
>   local result, i, str2num:
>   str2num := table([" "=Q1,"a"=Q2,"b"=Q3,"c"=Q4,"d"=Q5,"e"=Q6,"f"=Q7,
"g"=Q8,"h"=Q9,"i"=Q10,"j"=Q11,"k"=Q12,"l"=Q13,"m"=Q14,"n"=Q15,"o"=Q16,
"p"=Q17,"q"=Q18,"r"=Q19,"s"=Q20,"t"=Q21,"u"=Q22,"v"=Q23,"w"=Q24,"x"=Q25,
"y"=Q26,"z"=Q27]):
>    result := []:
>    for i from 1 to length(message) do
>      result := [op(result),str2num[message[i]]]:
>   end do:
>   return (result):
> end proc:
```

For example:

```
> cm := encode("hello world"); # coded message

  cm := [[32, 50], [28, 24], [48, 36], [48, 36], [26, 13], [1, 3],
         [51, 16], [26, 13], [24, 42], [48, 36], [25, 32]]
```

The inverse to the encode procedure takes on input a list of points and returns the corresponding string:

```
> decode := proc(l::list)
>   description ‘returns a string encoding of the list of points‘:
>   local result, i, num2str:
>   num2str := table([Q1=" ",Q2="a",Q3="b",Q4="c",Q5="d",Q6="e",Q7="f",
Q8="g",Q9="h",Q10="i",Q11="j",Q12="k",Q13="l",Q14="m",Q15="n",Q16="o",
Q17="p",Q18="q",Q19="r",Q20="s",Q21="t",Q22="u",Q23="v",Q24="w",Q25="x",
Q26="y",Q27="z"]):
>   result := "":
>   for i from 1 to nops(l) do
>     result := cat(result,num2str[l[i]]):
>   end do:
>   return result:
> end proc:
> decode(cm); # decoded message

                              "hello world"
```

3. Properties of the cryptosystem

While it is straightforward to generate the entire group from one point P by adding P to itself, it is very difficult to find n when P and n*P are given, especially when the prime is large.

4. Assignments

The solution to this project consists of a well documented worksheet, organized in three subsections, containing the answers to the three questions below.

4.1 Make your own elliptic curve

Generate a prime applying the nextprime command to your social security number. For the numbers a and b you apply the nextprime command on the respective first three and last four digits of your social security number. In the unlikely event that the discriminant becomes zero, apply nextprime again. In any case, no two students should have the same elliptic code.

Set up a way to verify the five group laws. As testing the laws on all elements may not be feasible, verify *all five* laws for random points on the curve.

4.2 Encrypt and decrypt your own messages

Modify the encode and decode procedures so that you can also use capital letters, punctuation, and newline symbols. The purpose of this question is that you demonstrate that you can encrypt and decrypt any normal paragraph of text.

4.3 Test the properties of the cryptosystem

To secure the system, primes of size 2^{160} are recommended. As our primes have far less than 49 decimal places, try to see if you can crack the cryptosystem. In particular, given the public key (P,Q), estimates how long it will take to get to the secret key.

Explain why for a secure and efficient cryptosystem, you must have a more efficient implementation of `mulf`. Give the modification of `mulf` to make it more efficient and show that, for larger numbers, the security does not come at the expense of efficiency.

4.4 The deadline is Friday 14 February 2003, at 2PM.

Bring to the lecture the print out of your worksheet. In addition also e-mail me your worksheet.

If you have questions, comments, or difficulties, feel free to come to my office for help.