Two Cryptanalysis Challenges based on the Discrete Log Problem

I had the good fortune to write two Crypto Challenges based on the Discrete Log Problem and I thank Nequ Marba for trying them out and solving it correctly.

Question One

You are an expert cryptanalyst at the missile launch control center belonging to the axis of good. The axis of evil is to unleash their secret weapon in a few hours. The axis of evil launches a preemptive strike on your launch control centre and you are the sole survivor. It is up to you crack the launch codes and save the world.

The launch code preparation data sheet provides the following useful information:

Primitive polynomial P(X) over GF(3)
(1)*X^99+(1)*X^98+(2)*X^96+(2)*X^92+(1)*X^91+(2)*X^90+(2)*X^89+(1)*X^87+
(1)*X^86+(1)*X^85+(2)*X^84+(2)*X^83+(2)*X^81+(2)*X^80+(2)*X^79+(2)*X^78+
(1)*X^75+(2)*X^74+(2)*X^73+(2)*X^72+(2)*X^71+(2)*X^70+(1)*X^69+(2)*X^68+
(2)*X^67+(2)*X^66+(2)*X^65+(1)*X^62+(1)*X^60+(2)*X^59+(2)*X^56+(2)*X^55+
(2)*X^54+(1)*X^52+(2)*X^51+(1)*X^50+(1)*X^48+(1)*X^47+(1)*X^45+(1)*X^43+
(2)*X^42+(2)*X^41+(1)*X^40+(1)*X^39+(1)*X^38+(1)*X^36+(2)*X^28+(2)*X^26+
(2)*X^23+(2)*X^22+(1)*X^21+(2)*X^20+(1)*X^19+(1)*X^18+(1)*X^15+(2)*X^14+
(2)*X^13+(1)*X^12+(2)*X^10+(1)*X^9+(2)*X^7+(1)*X^6+(2)*X^4+(1)*X^2+(2)*X+
(1).

and

X ^ launch_code = 2 mod P(x). (^ should be read as 'to the power of')

which is a case of solving discrete logarithms involving polynomials over
GF(3).

[Hint: There is a much better solution than brute force.]

Solution to Question One

Another exciting challenge has been solved. It is our pleasure to congratulate nequ.marba for his splendid cryptanalysis!

nequ.marba wrote:
no time 2 talk..
launch_code == 858962534553352218394101882942702121170179203333

ticker *ticker* *confirm*
--silence--
launch

END OF MOVE

If P(X) is primitive then every polynomial other than A=(0) should have a multiplicative inverse in mod P(X). Therefore the orbits w.r.t f(B) = XB mod P(X) are cycles of equal length and (0) as a fix. therefore 3^99-1 = #{polynomials mod P(X) that are not (0) } is a multiple of the cycle-length.

=> X^(3^99-1) == (1) mod P(X)

=> X^((3^99-1)/2) == (1) or (2) mod P(X)
note: (1) and (2) are the square-roots of (1) in mod P(X)

giving: X^((3^99-1)/2) a try..

```
from array import *

def poly():
        a=array('i',100*[0]);
        a[99]=1;
        a[98]=1;
        .... #result of some sed activities
        a[0]=1;
        return a;

def norm(a):
        p=poly();
        while len(a)>=len(p):
          d=len(a)-len(p)
          m=a.pop()
          for i in range(len(p)-1):
            a[d+i] = (a[d+i]-m*p[i])%3

def add(a,b):
        norm(a)
        norm(b)
        r=array('i',100*[0]);
        for i in range(99):
          r[i]=a[i]+b[i]
        return r

def mul(a,b):
        norm(a);
```

```
        norm(b);
        r=array('i',200*[0]);
        for i in range(99):
          for j in range(99):
            r[i+j] = r[i+j] + a[i]*b[j]
        norm(r);
        return r

def expo(b,e):
    r=array('i',[1,0,0]+100*[0]); # this is "(1)"
    while e>0:
      if e % 2 == 1:
        r = mul(r,b)
        e = e - 1
      b= mul(b,b)
      e=e/2
    return r

p = [0,1]+ 100*[0]; # this is "X"
ex = (3**99-1) / 2
print ex
print expo(p, ex) # will give (2,0,0,0,0,,...)
```

End of nequ.marba's comment.

Let us now formally write down Nequ's careful observation. We know that a primitive polynomial P(x) (element of) Fp[x] of degree m>=1 has order p^m -1. Hence the order of the polynomial in the challenge is 3^99 -1. Now the following result enables us to determine the launch_code.

Let P(x) (element of) Fp[x] be a polynomial of positive degree where f(0) is nonzero. Let 'r' be the least positive integer for which x^r==a modulo P(x), for some element 'a' in Fp*. Then order(P(x))=h.r where 'h' is the order of 'a' in the multiplicative group Fp* .

Therefore we know
3^99-1 = h*r (our task is to find 'r' which is the launch_code.)

We know h=order(a) in Fp*.
h=order(2) in F3*, since p=3.
Since 2^2==1 mod 3, the order(2) is 2. Hence 'h' is 2.
Therefore,
r=launch_code=(3^99 -1)/2.

This challenge is based on a problem in [5] and the primitive polynomial over GF(3) was computed with the help of [6].

Question Two

Tell the color of the sky and win exciting prizes, not any more. Your favorite T.V channel 'accidentally' hires a mathematician for channel promotion. If you answer his question correctly, the channel promoters provides you with a two day, three night stay package with your partner at an exotic island of your choice!.

Here is his question

Let
y=g^x (modulo p)
where

'p' is an odd prime.
'g' is a generator(modulo p)
'x' is a large positive integer.

We know that
P=
55043966783761716278659701487250086579117559740038746969702132145286149026123635393680445274643016477310977

g = 5

and

y=
39484558935413027037560973830889937944814968273298433695677167854170604782536404097409028160938915426071678

Find 'x', which is a discrete log problem.

[Hint: p-1= (3*2^353) and once the least significant 8 bits of 'x' is found, a pattern emerges that would help you find the remaining bits of x. There is a much faster solution than brute force, though the calculations may be annoyingly large. There may be more than one approach to the problem.]

Disclaimer: The mathematician and the prize announcement are hypothetical. We will not bear any expense or responsibility towards the island trip :-)

Solution to Question Two

Nequ.Marba has done it again, with some clever cryptanalysis!
Nequ.Marba
wrote:1884636869096943284888850667459113083549316156667326888663046614712127761L
send me to the island! :)

Let a^b = a**b
just found that y ** (2 ** 350) == g ** (2 ** 350) [mod p]
so tried to do sqrt in mod p
with knowledge that g ** (3 * 2 ** 352) == -1 [mod p]
you can get from
y ** (2 ** n) == g ** foo
to
y ** (2 ** (n-1)) == g ** bar

so finally you get to
y ** (2 ** 0) == y == g ** x

well, lets see code:

```
def pow(b,e,m):
  r=1
  while e>0:
    if e%2 == 1:
      r=(r*b)%m
      e=e-1
    e=e/2
    b=(b*b)%m
  return r

e1=e2=2**350;

while True:
  if pow(y,e1/2,p) == pow(5,e2/2,p):
    e1=e1/2; e2=e2/2; print"E"; continue
  if pow(y,e1/2,p) == pow(5,e2/2+3*2**352,p):
    e1=e1/2; e2=e2/2+3*2**352; print"O"; continue
  break

e1
e2
```

p-1 has the factor 2 many times..
therefore the operation sq: x -> x*x has a high collision ratio.

so started with doing sq 500 times on y and on g. collision found!
tried with 400. still a collision!
so reducing the number of times to apply sq-operator i ended with
y ** (2 ** 350) == g ** (2 ** 350) [mod p]
btw: this is luck.. there are pairs that never collide when applying sq over and over again.. but in crypto-things playing around often helps [if you know what you are doing].. and voila: this time it got me to the island :)

that's a general problem with crypto. knowing which keys are weak is impossible. because playing with a single key a mathematician can have very good guesses what way to roll the dice. sometimes if he breaks a single key he finds an algo to break a class of keys. knowing that these are weak does not tell which will be weak in future.

End of nequ.marba's comment.

So, the trick is to find such collisions.
Our task is to find y ^ (2 ^ n) == g ^ (foo) (mod p).

For us, p-1= (3*2^353). Now, if we had p to be a power of 2, then for any z=k mod (p=2^r), z is just the r lower order bits of k. The remaining bits are neglected.
e.g. let p = 65536= 2^16= (10000000000000000) to the base 2 and z=80000= (1 0011100010000000) to the base 2. Then 80000 mod 65536= (0011100010000000) to the base 2. Hence, now we can try to fix the lower significant 'r' bits and hope to find collisions or at least be sure that there is a good chance for collisions.

Now consider p to be an odd prime. Then, when we compute k mod p, if we do this operation by binary division, we see that the binary subtracted carry of p, propagates until the numerator bit from which '1' was borrowed.
e.g.80000 mod 65537= (2^16+1)= (111000 01111111) to the base 2. If we compare this to 80000 mod 65536= (111000 10000000) to the base 2, we see that the 6 higher order bits are same. At least, we are still able to fix some of the higher order bits which increases our chances of collision or at least be guaranteed that there is a fair chance of finding a collision

In our case, p is prime and p=3*(2^353+1)=(110...01) to the base 2. That is basically (11) to the base 2 left shifted 353 times and 1 added. We still can fix some of the higher order bits as in the previous case (p being an odd prime) and hope for collisions or be assured that there are some collisions.

Since the factorization of p-1 is smooth in our case, any DL algorithm that makes advantage of this knowledge may be used. Let us now look at a slightly different algorithm that suits our problem at hand. This algorithm is described as in [1].

We define,
The legendre symbol (x/p), for any integer with gcd(x,p)=1 is defined as (x/p)

= 1 if x^[(p-1)/2]== 1 mod p.
= -1 if x^[(p-1)/2]== -1 mod p.

Let y=g^x mod p, --->(1)
where g is a generator mod p, then
(y/p) = ((g^[(p-1)/2]) ^ x = (-1) ^ x

Let x = x_0 + 2*x_1 + 2^2 * x_2 + ...+ 2^j * x_j (Binary expansion)

If (y/n) = 1, x_i=0 (even), for the given bit i of x.
If (y/n) = -1, x_i=1 (odd), for the given bit i of x.

If y is a quadratic residue mod p, then the square roots of y are y^(1/2) and -y^(1/2).
y^(1/2) = g^(x/2) mod p and

-y^(1/2) = -g^(x/2) mod p = -1 * g^(x/2) = g^[(p-1)/2] * g^(x/2) = g^ [(p-1)/2 + (x/2)].

If p==1 mod 4, the least significant bit (lsb) of [(p-1)/2]=0. Hence the lsb of (x/2) = lsb of [(p-1)/2 + (x/2)]. Therefore, the legendre symbol of y^(1/2) or -y^(1/2) will correctly determine the value of the x_i unambiguously. Note that p==1 mod 4, for the challenge.

In general, if p-1 = 2^r * s, where s is odd and r>1, the least significant r-bits of the discrete log, 'x', can be identified uniquely. The ambiguity starts from the (r+1) th significant bit.

Let us now look at a small example.
Find x, such that
7=3^x (mod 17).

We note 17==1 mod 4.
(7/17) = 7^[(17-1)/2] mod 17 = -1. Therefore x_0=1.

Let b_0=y=7

Note:
If x_i = 1, we compute b_i=(b_(i-1) * g ^ -1) mod p, for the bit i of x.
If x_i = 0, we compute b_i=(b_(i-1)) mod p, for the bit i of x.

b_1 = 7*(3 ^ -1) mod 17 =8

(b_1)^ 1/2 =5 and -(b_1)^ 1/2 =12.

Choose any of the roots, lets choose 5 and set b_1=5

(5/17) = 5^8 mod 17 = -1. Therefore x_1=1.

b_2 = 5*(3 ^ -1) mod 17 = 13
(b_2)^ 1/2 =8 and -(b_2)^ 1/2 =9.

Choose any of the roots, lets choose 8 and set b_2=8

(8/17) = 8^8 mod 17 = 1. Therefore x_2=0.

b_3 = 8 mod 17 =8.
(b_3)^ 1/2 =5 and -(b_2)^ 1/2 =12.

Choose any of the roots, lets choose 12 and set b_3=12

(12/17) = 12^8 mod 17 = -1. Therefore x_3= 1

Hence x=(1011) to the base 2, i.e x =11 to the base 10.
We verify that 7 =3^11 mod 17, which is true.

Thanks to Hans Riesel's extended list for primes of the form k*2^n + 1 in [2].

On a closing note, here is one more challenge to solve.

Let f(x) be a primitive polynomial modulo 2(but not a trinomial). Let 'alpha' be a primitive root of f(x).
Let 'j' be a positive integer close to 2^40 and f(x) be of degree 80.

Find a primitive polynomial of degree 80 with 'j' close to 2^40 such that
alpha^j==alpha+1 mod f(x)

Finding such a desired 'j'(close to a given positive integer) for a primitive polynomial f(x) of given degree, is of use in irregular clocking of linear feedback shift registers without stepping through all the internal states. We are now able to 'jump' to the desired state. For more information see [3] and for a solution to this problem see [4]

Hope you all enjoyed it!

Bibliography

[1.] "What is the Inverse of Repeated Square and Multiply Algorithm?",Gadiyar,Maini and Padma. Available for download at:
http://arxiv.org/abs/math.NT/0602154

[2.] Hans Riesel's extended list for primes of the form k*2^n + 1, k<300. Website:
http://www.prothsearch.net/riesel.html

[3.] "Cascade Jump Controlled Sequence Generator (CJCSG)", Cees Jansen and Alexander Kolosha . Available for download at:
http://www.ecrypt.eu.org/stream/pomaranch.html

[4]. "Finding Characteristic Polynomials with Jump Indices", Steve Babbage and Matthew Dodd. Available for download at:
http://eprint.iacr.org/2006/010

[5.] " Introduction to Finite Fields and their Applications", Lidl and Niederreiter.

[6.] The Primitive and Irreducible Polynomial Server (PIPS). Website:
http://zenfact.sourceforge.net/PIPS/polyformind.html

-Sarad A.V