

University College London
Department of Computer Science

Cryptanalysis Lab 4

J. P. Bootle

Generating Discrete Logarithm Instances

Recall that a prime p is called a ‘strong prime’ if $p = 2q + 1$, where q is also prime.

The following function generates random discrete logarithm instances. On input n , the function first finds the smallest strong prime p that is greater than n . Thus, \mathbb{Z}_p^* has a subgroup of order q . Finally, the function generates two random elements g, h of the subgroup, and outputs $[p, q, g, h]$.

```
def dlog_gen(n):
    p = next_prime(n)
    while not is_prime( floor((p-1)/2 ) ):
        p = next_prime(p)
    x = randint(1,p-1)
    y = randint(1,p-1)
    g = x*x % p
    h = y*y % p
    return [p,floor( (p-1)/2 ),g,h]
```



Copy and paste the code into SAGE. This function will be used to generate discrete logarithm instances for the following questions.

Modular Exponentiation

The following function performs modular exponentiation. It computes $a^k \bmod n$ and outputs the answer.

```
def MyPower(a,k,n):
    K = bin(k)[2:]
    A = a % n
    c = 1
    if int(K[0])==1:
        c = (c*A) % n
    for j in range(1,len(K)):
        c = (c2) % n
        if int(K[j])==1:
            c = (c*A) % n
    return c
```



Back

Copy and paste the code into SAGE. You may use this function to check your solutions to discrete logarithm instances.

Implementing the Baby-Step Giant-Step Algorithm

Click on the green letter before each question to get a full solution. Click on the green square to go back to the questions.

EXERCISE 1.

- (a) Write a function ‘BSGS’ which implements the Baby-Step Giant-Step algorithm. The function should take inputs p, q, g, h as produced by the DLOG instance generator, and output $k \in \{0, 1, \dots, q-1\}$ such that $g^k = h \pmod p$.

You may wish to structure your code as follows.

- Calculate the sizes of the baby steps and the giant steps.
 - Compute all of the baby steps, and store them in a list.
 - Compute giant steps until you get an item in the list.
 - Return the correct value of k .
- (b) Generate DLOG instances with `dlog_gen(n)` for a range of large n . Using the `timeit` command, test how long your program takes



Back

to solve these instances. Plot a graph of the time taken to solve each instance against the size of the group in the instance. What shape graph do you expect to see?

Implementing the Pollard-Rho Algorithm

Click on the green letter before each question to get a full solution. Click on the green square to go back to the questions.

EXERCISE 2.

- (a) Write a function `iterator` which implements the iterative function required for the Pollard-Rho algorithm for the discrete logarithm problem. This is part of the algorithm given on page 18 of the slides on DLOG and Factoring. The function should take inputs $[a, b, G]$ and p, q, g, h , where $G = g^a h^b$, and output the new values $[a', b', G']$ according to the iterative function.
- (b) Write a function `pollard_rho` which implements the low-memory version of the Pollard-Rho algorithm. The function should take inputs p, q, g, h as produced by the DLOG instance generator, and



Back

output $k \in \{0, 1, \dots, q - 1\}$ such that $g^k = h \pmod p$. Run the algorithm for a fixed number of iterations. You may wish to structure your code as follows.

- Definition of the initial $[a, b, G]$ for the iteration.
 - Set the number of iterations to do.
 - Main loop using the iterative function.
 - At each step of the main loop, check for collisions.
 - Return the correct value of k or output ‘Fail’.
- (c) According to the analysis of the running time of the Pollard-Rho algorithm, how many iterations should we expect to use before the algorithm succeeds in finding a collision?
- (d) Generate DLOG instances with `dlog_gen(n)` for a range of large n . Using the `timeit` command, test how long your program takes to solve these instances. Plot a graph of the time taken to solve each instance against the size of the group in the instance. Compare with your results from the Baby-Step Giant-Step algorithm. Which of your implementations is faster?
- (e) (Bonus Question) The running time of the Pollard-Rho algorithm



depends on the iteration function behaving like a random function. Modifying the iteration function can improve the running time of the algorithm in practice. Modify your functions `iterator` and `pollard_rho` to work as follows. Does this improve the running time of the algorithm?

- `pollard_rho` generates $g' = g^{a'} h^{b'}$ and $g'' = g^{a''} h^{b''}$, where a', a'', b', b'' are chosen uniformly at random from $\{0, 1, \dots, q-1\}$.
- `iterator` takes $g', g'', a', a'', b', b''$ as additional inputs.
- For $0 \leq G < p/5$, `iterator` maps $[a, b, G]$ to $[a + 1, b, G * g]$.
- For $p/5 \leq G < 2p/5$, `iterator` maps $[a, b, G]$ to $[a, b + 1, G * h]$.
- For $2p/5 \leq G < 3p/5$, `iterator` maps $[a, b, G]$ to $[2a, 2b, G^2]$.
- For $3p/5 \leq G < 4p/5$, `iterator` maps $[a, b, G]$ to $[a + a', b + b', G * g']$.
- For $4p/5 \leq G < p$, `iterator` maps $[a, b, G]$ to $[a + a'', b + b'', G * g'']$.



Elliptic Curve Diffie-Hellman

Click on the green letter in front of each sub-question (e.g. (a)) to see a solution. Click on the green square at the end of the solution to go back to the questions.

EXERCISE 3. In this exercise, you will use Sage and share a Diffie-Hellman Key with a partner, using points on an elliptic curve. To create an elliptic curve E defined by $y^2 = x^3 + ax + b$ over \mathbb{F}_p , use `E = EllipticCurve(GF(p), [a, b])`.

- (a) Create an elliptic curve E defined by $y^2 = x^3 + 70x + 355$, over the finite field of size 1031.
- (b) The command `n = E.cardinality()` sets n to be the number of points on the curve. What is the value of n ? What properties should n have in order to be suitable for Diffie-Hellman?
- (c) Typing `E.gens()` gives a set of points which generate all the points on the elliptic curve. In this case, there is only one generator, and `P = E.gens()[0]` sets P to be a group generator for this curve. If $P = (x : y : z)$, then your partner can get P by typing `P = E(x, y, z)`.



Back

- (d) Choose a random integer a such that $0 \leq a < n$. Your partner should choose b similarly.
- (e) Use Sage to find the elliptic curve point $A = a^*P$, and give this to your partner. For example, if $A = (x : y : z)$ then your partner can type $\mathbf{A} = \mathbf{E}(x, y, z)$ to get A .

Your partner should compute $B = b^*P$ and give this to you in the same way.

- (f) Use Sage to find $a^*B = (ab)^*P$. Your partner will also find $(ab)^*P$ via b^*A . The point $(ab)^*P$ is your shared secret key. Check that you and your partner computed the same answer.



Solutions to Exercises

Exercise 1(a) The following code implements the Baby-Step Giant-Step algorithm.

```
def BSGS(p,q,g,h):
    n = floor(sqrt(q))
    baby_steps = [1]
    for j in range(0,n):
        baby_steps = baby_steps + [(baby_steps[-1]*g) % p]
    v = (baby_steps[-1]*g) % p
    G = inverse_mod(v,p)
    H = h % p
    if H in baby_steps:
        return baby_steps.index(H)
    for i in range(1,n):
        H = (H*G) % p
        if H in baby_steps:
            return i*(n+1)+baby_steps.index(H)
    return 'fail'
```





Exercise 1(b) Experimenting with the following code should allow you to plot graphs with curves of best fit. The line of code `'model(t)=a*(thatb)'` finds the best power of the input size and constant multiplier to match the points on the graph.

```
x = [(1,3),(2,5),(3,7),(4,9)]
```

```
var('a,b,t')
```

```
model(t)=a*(thatb)
```



```
fit=find_fit(x,model,solution_dict=True)
```

```
plot(model.subs(fit),(t,0,5))+points(x,size=20,color='red')
```



Exercise 2(a) The following code implements the iterative function.

```
def iterator(triple,p,q,g,h):
```

```
    [a,b,G] = triple
```

```
    if G  $\leq$  p/3:
```

```
        return [(a+1) % q , b,(G*g) % p]
```

```
    elif G  $\leq$  2*p/3:
```

```
        return [a,(b+1) % q,(G*h) % p]
```

```
    else:
```

```
        return [(2*a)%q,(2*b)%q,(G*G) % p]
```



Back

Exercise 2(b) The following code implements the Pollard-Rho algorithm.

```
def pollard_rho(p,q,g,h):
    n = floor(sqrt(q))
    ai = 1
    bi = 0
    Gi = g % p
    a2i = 1
    b2i = 0
    G2i = g % p
    for k in range(1,n):
        [ai,bi,Gi] = iterator([ai,bi,Gi],p,q,g,h)
        [a2i,b2i,G2i] = iterator([a2i,b2i,G2i],p,q,g,h)
        [a2i,b2i,G2i] = iterator([a2i,b2i,G2i],p,q,g,h)
        if Gi == G2i:
            if ((bi-b2i) % q) == 0:
                return 'fail1'
            return (a2i-ai)*inverse_mod(bi-b2i,q) % q
    return 'fail'
```

[Back](#)



Exercise 2(d) Experimenting with the following code should allow you to plot graphs with curves of best fit. The line of code ‘`model(t)=a*(thatb)`’ finds the best power of the input size and constant

```
x = [(1,3),(2,5),(3,7),(4,9)]
```

```
var('a,b,t')
```

multiplier to match the points on the graph.

```
model(t)=a*(thatb)
```

```
fit=find_fit(x,model,solution)
```

```
plot(model.subs(fit),(t,0,5))-
```



Exercise 3(a) Use $E = \text{EllipticCurve}(\text{GF}(1031), [70, 355])$ to produce the correct elliptic curve. □

[Back](#)

Exercise 3(b) You should get $n = 1009$. For secure Diffie-Hellman key exchange, we ideally want n to be large and prime so that the Discrete Logarithm problem is hard in the elliptic curve group. \square



Exercise 3(c) An example generator is the point $P = (5 : 393 : 1)$. It doesn't matter which generator you use, as long as you and your partner are using the same generator. \square



Exercise 3(d) You can use `a = randint(0,1009)` to get a .



Back

Exercise 3(e) The point a^*P is computed in Sage exactly as written here: `a*P`. □



Exercise 3(f) Get b^*P from your partner.



Back