Implementing Discrete Logarithm based Digital Signature Schemes

[1]A. B. Nimbalkar, [2]Dr. C. G. Desai
[1]A.M.College, Hadapsar, Maharashtra, Pune, India
[2]H.O.D. MIT Aurangabad, Maharashtra, Pune, India

Abstract

A digital signature is a cryptographic method for verifying the identity of an individual, a process, computer system, or any other entity, in much the same way as a handwritten signature verifies the identity of a person. Digital signatures use the properties of public-key cryptography to produce pieces of information that verify the origin of data. Several digital schemes have been proposed as on date based on factorization, discrete logarithm and elliptical curve. However, the Pollard rho and the baby-step giant-step Algorithm digital scheme based on discrete logarithm gained wide acceptance. Many schemes followed there by with little changes in it. Some of the schemes evolved by combing factorization and discrete logarithm together making it difficult for solving two hard problems from the hackers point of view. This paper presents the implementation of Pollard rho and the baby-step giant-step algorithm, with the help different tools and further analyzed them for different perceptions.

Keywords

Cryptography, Discrete Logarithm, Digital Signature

I. Introduction

Cryptography is the Art or Science of Keeping Secrets Secret Cryptography is About secure communication through insecure channels [1]. Cryptographic techniques, such as digital signature, key agreement and secrete sharing schemes, are important building blocks in implementing any security services for confidential communication.

A digital signature is typically created by computing a message from the original document and concatenating it with the information about the signer, such as time stamp. The resulting string is then encrypted using the private key of the signer. The encrypted block of bits is known as digital signature. Digital signatures are used to verify that the message really come s from the sender the receipting supposes sent the message.

Public key cryptography is an asymmetric scheme that uses a pair of keys: a public key, which encrypts data, a corresponding private key, or secrete key for decryption. A major benefit of public key cryptography is that it provides a method for employing digital signatures. Digital signatures enable the recipient of the information to verify the authenticity of information origin, and also verify that the information is intact [1].

In order to verify the digital signature, the recipient must decide whether it trusts that the key used to encrypt the message actually belongs to the person it is supposed to belong to. A digital signature is very small amount of data created using some secrete key. Typically there is a public key that can be used to verify that the signature was really generated using the corresponding private key. The algorithm used to generate the signature is of sufficient cipher strength that, knowing the secrete key, it would be impossible to create a counterfeit signature that would verify it as valid. Once the recipient has decrypted the signature using public key of the sender, the recipient compares the information to see if it matches that of the message. Only then is the signature accepted valid.

II. Discrete Logarithm based Algorithms [1]

Many of the most commonly used cryptography systems are based on the assumption that the discrete log is extremely difficult to compute; the more difficult it is, the more security it provides a data transfer. One way to increase the difficulty of the discrete log problem is to base the cryptosystem on a larger group. The discrete log problem is of fundamental importance to the area of public key cryptography. The two important algorithms for discrete logarithm are as follows.

A. The Baby-Step Giant-Step Algorithm [1]

This algorithm is a meet-in-the-middle algorithm computing the discrete logarithm. The baby-step giant-step algorithm is a generic algorithm. It works for every finite cyclic group.

The algorithm was originally developed by Daniel Shanks [4]. Let $m = \lceil \sqrt{n} \rceil$ where n is the order of α. The baby-step giant-step algorithm is a time memory trade-off of the method of exhaustive search and is based on the following observation.

If $\beta = \alpha^x$, then one can write $x = im+j$, where $0 \le i, j < m$. Hence, $\alpha^x = \alpha^{im}\alpha^j$, which implies $\beta(\alpha^{-m})i = \alpha^j$. This suggests the following algorithm for computing x.

INPUT: a generator α of a cyclic group G of order n, and an element $\beta \in G$.

OUTPUT: the discrete logarithm $x = \log\alpha\ \beta$.

1. Set $m \leftarrow \lceil \sqrt{n} \rceil$
2. Construct a table with entries (j, α^j) for $0 \le j < m$. Sort this table by second component. (Alternatively, use conventional hashing on the second component to store the entries in a hash table; placing an entry, and searching for an entry in the table takes constant time.)
3. Compute $\alpha{-}m$ and set $\gamma \leftarrow \beta$.
4. For i from 0 to m^{-1} do the following:

4.1 Check if γ is the second component of some entry in the table.

4.2 If $\gamma = \alpha^j$ then return($x = im + j$).

4.3 Set $\gamma \leftarrow \gamma * \alpha^{-m}$.

The running time of the baby-step giant-step algorithm is O ($\sqrt{n}$) group multiplications.

Example (baby-step giant-step algorithm for logarithms in $\mathbb{Z}^*_{113}$)

Let p = 113. The element

α = 3 is a generator of $\mathbb{Z}^*_{113}$ of order n = 112. Consider β = 57.

Then log3 57 is 100.

- The baby-step giant-step algorithm is a generic algorithm. It works for every finite cyclic group.
- It is not necessary to know the order of the group G in advance. The algorithm still works if n is merely an upper bound on the group order.
- The algorithm is used for groups whose order is prime.
- The algorithm requires O(m) memory. It is possible to use less memory by choosing a smaller m in the first step of the algorithm.

 ISSN : 0976-8491 (Online) | ISSN : 2229-4333 (Print)

The implementation of baby-step giant-step Algorithm is done in MATELAB 7.1

The Code is:

```
p=113;
 Alpha=3;
 n=112;
 Beta=57;

 % Step 1
 m=ceil(sqrt(n));
 disp(m);
%Step 2
j=0;
   for k = 1:m
      Tab(1,k)=j;
      z= power(3,j);
      rm=mod(z,p); %
      Tab(2,k)=rm;
    j=j+1;
   end
  % Sorting of Array
  disp(Tab);
   disp('Sort');
   for k = 1:m
      for tmp = 1 : m
          if(Tab(2,k) < Tab(2,tmp))
             swp=Tab(2,k);
             Tab(2,k)=Tab(2,tmp);
             Tab(2,tmp)=swp;

             % First Row
             swp=Tab(1,k);
             Tab(1,k)=Tab(1,tmp);
             Tab(1,tmp)=swp;
          end
      end
   end
   disp(Tab);

%   Step 3
   newAlpha= Mult_Inverse(Alpha,p);
   disp(newAlpha);
   mdExpo= modexpo(newAlpha,m,p);
  disp(mdExpo);

% Step 4
i=0;
 k=1;
   while(k>0)
      Tab2(1,k)=i;
       rm=modexpo_baby(Beta,mdExpo,i,p);
       Tab2(2,k)=rm;
      if(rm==Alpha)
          k=0;
          break;
      end
    i=i+1;
    k=k+1;
   end
  disp(Tab2);

  % x= im + j;
 %search j in Tab1 for same value like  rm
   for k = 1:m
     if(Tab(2,k)==rm)
        j=Tab(1,k);
        break;
     end
  end
 x= i*m +j;
 disp(x);
```

B. Pollard's rho Algorithm for Logarithms [1]

Pollard's rho algorithm for computing discrete logarithms is a randomized algorithm with the expected same running time as the baby-step giant-step algorithm, but only a small memory requirement [1]. For simplicity, it is assumed in this subsection that G is a cyclic group whose order n is prime.

The group G is partitioned into three sets S1, S2, and S3 of roughly equal size based on some easily testable property. Some care must be exercised in selecting the partition; for example, $1 \notin S2$. Define a sequence of group elements $x_0, x_1, x_2, \ldots$ by $x_0 = 1$ and

$$x_{i+1} = f(x_i) \stackrel{\text{def}}{=} \begin{cases} \beta \cdot x_i, & \text{if } x_i \in S_1, \\ x_i^2, & \text{if } x_i \in S_2, \\ \alpha \cdot x_i, & \text{if } x_i \in S_3, \end{cases} \quad (1)$$

for $i \geq 0$. This sequence of group elements in turn defines two sequences of integers $a_0, a_1, a_2, \ldots$ and $b_0, b_1, b_2, \ldots$ satisfying $xi = \alpha^{ai}\beta^{bi}$ for $i \geq 0$: a0 = 0, b0 = 0, and for $i \geq 0$,

$$a_{i+1} = \begin{cases} a_i, & \text{if } x_i \in S_1, \\ 2a_i \bmod n, & \text{if } x_i \in S_2, \\ a_i + 1 \bmod n, & \text{if } x_i \in S_3, \end{cases} \quad (2)$$

$$b_{i+1} = \begin{cases} b_i + 1 \bmod n, & \text{if } x_i \in S_1, \\ 2b_i \bmod n, & \text{if } x_i \in S_2, \\ b_i, & \text{if } x_i \in S_3, \end{cases} \quad (3)$$

Floyd's cycle-finding algorithm [1] can then be utilized to find two group elements

x_i and x_{2i} such that $x_i = x_{2i}$. Hence $\alpha^{ai}\beta^{bi} = \alpha^{a2i}\beta^{b2i}$, and so $\beta^{bi-b2i} = \alpha^{a2i-ai}$.

Taking logarithms to the base α of both sides of this last equation yields $(b_i - b_{2i}) * \log_\alpha \beta \equiv (a_{2i} - a_i) \pmod n$.

Provided $bi \not\equiv b2i \pmod n$ (note: $b_i \equiv b_{2i}$ occurs with negligible probability), this equation can be then be efficiently solved to determine $\log\alpha\ \beta$.

INPUT: a generator α of a cyclic group G of prime order n, and an element $\beta \in G$.

OUTPUT: the discrete logarithm $x = \log_\alpha \beta$.

1. Set $x_0 \leftarrow 1$, $a_0 \leftarrow 0$, $b_0 \leftarrow 0$.
2. For i = 1, 2, . . . do the following:

2.1 Using the quantities $x_{i-1}, a_{i-1}, b_{i-1}$, and $x_{2i-2}, a_{2i-2}, b_{2i-2}$ computed previously, compute x_i, a_i, b_i and x_{2i}, a_{2i}, b_{2i} using equations 1,2, and 3.

2.2 If $x_i = x_{2i}$, then do the following:

Set $r \leftarrow b_i - b_{2i} \bmod n$.

If r = 0 then terminate the algorithm with failure; otherwise, compute
$x = r^{-1}(a_{2i} - a_i) \bmod n$ and return(x).
In the rare case that if this algorithm terminates with failure, the procedure can be repeated by selecting random integers a_0, b_0 in the interval [1, n−1], and starting with $x_0 = \alpha^{a0}\beta^{b0}$.
Example: The logarithms in a subgroup of $\mathbb{Z}^*_{383}$. The element $\alpha = 2$ is a generator of the subgroup G of $\mathbb{Z}^*_{383}$ of order n = 191. Suppose $\beta = 228$. Partition the elements of G into three subsets according to the rule x S1 if $x \equiv 1 \pmod 3$, $x \in S2$ if $x \equiv 0 \pmod 3$, and $x \in S3$ if $x \equiv 2 \pmod 3$. The values of xi, ai, bi are calculated. The x_{2i}, a_{2i}, and b_{2i} at the end of each iteration of step 2 of the Algorithm. The values of $x_{14} = x_{28} = 144$. Finally, compute $r = b_{14} - b_{28} \bmod 191 = 125$, $r^{-1} = 125^{-1} \bmod 191 = 136$, and $r^{-1}(a_{28} - a_{14}) \bmod 191 = 110$. Hence, $\log_2 228 = 110$.
Then the expected running time of Pollard's rho algorithm for discrete logarithms in G is $O(\sqrt{n})$ group operations. Moreover, the algorithm requires negligible storage.
2.2.1 The implementation of Pollard's rho algorithm for logarithms in MATLAB 7.1. The code for this is as follows :

```
% Input : A Generator α of Cyclic group G of prime order n and an element B β belongs to G
% element B Beta belongs to G
% Output : The discrete Logarithm x=log α^β

% Step 1
n=191;
x=1;
 a=0;
 b=0;
 XX=x;
AA=a;
 BB=b;

%  Beta=228;   Declared in Function Def_group
%  Alpha=2;
%  Z=383;

 % Log in subgroup of Z*
 % Step 2
for i=1:n
[x,a,b]=Def_group(x,a,b,n);
[XX,AA,BB]=Def_group(XX,AA,BB,n);
[XX,AA,BB]=Def_group(XX,AA,BB,n);

%Step 2.1
% if xi == x2i then stop calculating xi and x2i's
if(x==XX)
   disp(x);
   break;
end
end;
% now we get xi , ai, bi and x2i(ie XX) ,a2i(AA),b2i(BB)
disp('OutPut');
disp(i);
disp(x);
disp(a);
disp(b);
disp(XX);
disp(AA);
disp(BB);
% Step 2.2
 % compute  r = bi - b2i (ie b and BB) mod with n;
 r= mod( (b-BB) , n);
 disp(r);

 % To find  r inverse : Find the number that multiply to r such that
 % remainder is  1 , when devide  by n
 r_inv=1;
while(1)
   ans=mod((r*r_inv),n);
   if(ans ==1)
      break;
   end
   r_inv = r_inv+1;
end;
disp(r_inv);

%The discrete Logarithm x=log alpha B
% r inverse(a2i - ai) mod n
log= mod(((AA-a)*r_inv) , n);
disp('Log is : ');
disp(log);
```

The supporting function Def_group

```
function [x,a,b]=Def_group(x,a,b,n)
  Beta=228
   Alpha=2;
    Z=383;
%   n=191;
  % According to the rule
  %x belongs to S1 if x= 1(mod 3)
  %x belongs to S2 if x= 0(mod 3)
  %x belongs to S3 if x= 2(mod 3)

switch(mod(x,3))
   case 0
         x=x*x;      %S2
         a=mod(2*a,n);
         b=mod(2*b,n)
   case 1
         x=Beta * x;  %S1
         a=a;
         b=mod(b+1,n);
   case 2
         x=Alpha*x;   %S3
         a=mod(a+1,n);
         b=b;
end

x=mod(x,Z);
 return;
end
```

2.2.2 The implementation of Pollard's rho algorithm for logarithms in Maple15. The code for this is as follows :
The supporting function Def :

 ISSN : 0976-8491 (Online) | ISSN : 2229-4333 (Print)

```
Def := proc(x :: integer, a :: integer, b :: integer, n
  local Beta := 228;
 local Alpha := 2;
 local Z := 383;
 local case1;
local A;
 case1 := x mod 3;
 x1 := x;
 a1 := a;
b1 := b;
 if case1 = 0 then
   x1 := x·x;
   a1 := (2·a ) mod n;
   b1 := (2·b ) mod n;
 elif case1 = 1 then
     x1 := Beta·x;
    a1 := a;
    b1 := (b + 1) mod n ;
  else
       x1 := Alpha·x;
       a1 := (a + 1) mod n;
       b1 := b;
 fi;
 x1 := x1 mod Z;
 A := Array([x1, a1, b1]);
 end proc ;
```

```
# Main prg

n := 191;
x := 1;
a := 0;
b := 0;
XX := x;
AA := a;
BB := b;
 # STEP 2
for i from 1 by 1to n do
 Ar := Def(x, a, b, n);
x := Ar[1];
a := Ar[2];
 b := Ar[3];
 print('Ar'); print(Ar);
 Ar2 := Def(XX, AA, BB, n);
  XX := Ar2[1];
  AA := Ar2[2];
  BB := Ar2[3];
 Ar2 := Def(XX, AA, BB, n);
  XX := Ar2[1];
  AA := Ar2[2];
  BB := Ar2[3];
 print('Ar2');
 print(Ar2);
  if x = XX then
   print(x);
   break;
  end if;
  end do;

# Step 2.2
 r := (b-BB) mod n ;
 rInv := 1;
 while rInv > 0 do
 ans := (r·rInv) mod n;
  if ans = 1 then
   break;
   fi;
  rInv := rInv + 1;
 end do;

log1 := ((AA-a)·rInv) mod n;
#print('Log is : ');
print('Log');
print(log1);
```

III. Conclusion

For developing the integral factoring algorithm, the large prime numbers and large digit numbers are used. There are some difficulties for implementation of these algorithms in Matlab7.1. As per the method adopted for implementing these algorithms, the Matlab7.1 does not support for large digit number, it support maximum 16 digit number. So there is limitation for secure key generation.

Due to number limitation in Matlab7.1, the Maple15 is used. It is product of Maple soft. Basically it is designing tool. It also support for large computing. It support for large integer number. We tried for large prime number, also checked for the square of large number, We could generate various operations on max 160 digit number , to give successful results.

Looking at the advantage supported by Maple15, the Pollard-rho algorithm was again implemented in Maple15. However, it was observed that programming controls in Matlab is easier as compared to Maple. It was also observed that function calling in Maple15 was more time consuming as compared to Matlab7.1.

The program written in Matlab we can edit any where but program in Maple 15, we cannot edit outside Maple 15.

The Maple use interpreter, It execute line by line and output show on same programming window. But in Matlab the programming window is 3 different and output Window is different and output windows is different so it is easy to change or debug the program.

The Maple 15 does not show the detail errors, also not gives the line numbers, where the syntactical error occur, only it gives error messages, so it is very difficult to find out or fix the error.

In Matlab it shows the error details with line numbers so it is easy to fix the error.

References

[1] Menezes, Alfred J; van Oorschot, Paul C.; Vanstone, Scott A. (2001),"Handbook of Applied Cryptography", [Online] Available: http://www.cacr.math.uwaterloo.ca/hac/about/chap3.pdf

[2] Pollard, J. M. (1978),"Monte Carlo methods for index computation (mod p)", Mathematics of Computation 32 (143): 918–924. JSTOR 2006496.Richard Crandall; Carl Pomerance. Chapter 5, Prime Numbers: A computational perspective, 2nd ed., Springer

[3] Stinson, Douglas Robert (2006),"Cryptography: Theory and Practice (3rd ed.)", London: CRC Press.

[4] Cohen,"A course in computational algebraic number theory", Springer, 1996. // baby step

[5] Matlab 7.1 and Maple15 Software's.

Mr. A.B.Nimbalkar received his B.C.S.(Bachelor of Computer Science) degree from Pune University, in 1996, M.C.S. degree from Pune University, Pune, India, in 2000. The M.Phil. (Computer science) degree from Algappa University in 2008. He is working as a Assistant professor, with Department of Computer Science in Annasaheb Magar Mahavidyalaya, Hadapsar, Pune University, from 2000. His research interests include Soft Computing, Cryptography, Digital Signature. He has presented two papers in State level and National level Conferences. At present, He is engaged in Digital Signature Algorithms using Discrete Logarithm and Integer Factoring.