

# Projets de TP de Cryptographie : Échange de clé de Diffie-Hellman

Robert Rolland

19 Avril 2004

e-mail : rolland@iml.univ-mrs.fr

**Avertissement :** Ceci est un exemple de projet de Travaux Pratiques de cryptographie. Bien sûr il ne s'agit pas là de la description du travail à faire, donnée aux étudiants. Il s'agit de l'étude de faisabilité du projet, ainsi que du relevé rapide des outils théoriques et des algorithmes permettant de comprendre le sujet et de le traiter. Il s'agit aussi de prévoir une démarche, qui va des réflexions théoriques jusqu'aux applications concrètes en passant par la mise en place d'algorithmes, leur programmation et les réflexions sur les résultats obtenus. Cette démarche est, me semble-t-il, apte éclairer le contexte des outils introduits, et à motiver les débutants dans l'apprentissage du sujet. La programmation et les tests sont faits avec le logiciel xcas [2]. J'attire l'attention des enseignants sur le grand intérêt de l'utilisation de ce logiciel dans les classes de divers niveaux.

## 1 Présentation du problème

Nous présentons ici le **principe de l'échange de clés de Diffie-Hellman**. L'échange de clés s'insère dans un protocole complexe. Il s'agit en général d'établir un canal de communication sûr. Pour cela on utilise le plus souvent, un circuit de chiffrement à clé secrète, beaucoup plus rapide pour chiffrer un flux de données. L'échange de la clé secrète se fait, soit avec un chiffrement à clé publique comme par exemple RSA, soit par un échange de type Diffie-Hellman (basé sur la résistance du problème du logarithme discret et ses variantes : problème de Diffie-Hellman, problème décisionnel de Diffie-Hellman). Cette dernière méthode présente l'avantage d'assurer la sécurité rétroactive, c'est-à-dire qu'en cas de divulgation d'un élément secret ou privé du système, seul l'échange en cours est affecté,

mais pas les échanges précédents. Autrement dit un ennemi qui aurait enregistré pendant une certaine période les chiffrés échangés, ne peut pas remonter dans le temps. En revanche elle est très vulnérable à l'attaque de l'homme au milieu. Couplée avec un procédé d'authentification mutuelle des deux parties, elle est très efficace.

Le principe est le suivant : On dispose d'un grand nombre premier  $p$  et d'un élément primitif  $\alpha$  du corps  $\mathbb{Z}/p\mathbb{Z}$  (générateur du groupe multiplicatif  $(\mathbb{Z}/p\mathbb{Z})^*$ ). Chacun des deux interlocuteurs  $A$  et  $B$  choisit un entier inférieur à l'ordre du groupe. Nous noterons  $n$  et  $m$  ces deux entiers choisis respectivement par  $A$  et par  $B$ .  $A$  transmet à  $B$  le nombre  $\alpha^n \pmod p$ , tandis que  $B$  transmet à  $A$  le nombre  $\alpha^m \pmod p$ . Chacun des deux interlocuteurs est alors en mesure de calculer  $L = \alpha^{mn} \pmod p$ . En général, ce nombre  $L$  est, d'une part, trop long pour servir de clé pour un système à clé secrète, d'autre part, certains bits sont faciles à calculer. Si bien qu'on le hache pour obtenir la clé commune  $K = h(L)$  (bien entendu cette clé  $K$  est reconstruite à chaque nouvelle session avec des nouveaux paramètres  $n$  et  $m$ ) (cf. [1]).

## 2 Mise en place du système

Nous allons simuler un tel système en utilisant un logiciel de calcul. Le logiciel utilisé est **xcas** [2].

La mise en place du système passe par la construction d'un grand nombre premier  $p$  et la détermination d'un élément primitif  $\alpha$ . Rappelons (cf. [1]) que pour tester si un élément  $a$  est primitif de manière efficace, on ne dispose guère que de l'algorithme qui suit : Supposons que :

$$p - 1 = \prod_{i=1}^k q_i^{s_i}$$

soit la décomposition de  $p - 1$  en facteurs premiers. Alors pour que  $a$  soit primitif il faut et il suffit que les  $k$  nombres

$$a^{\frac{p-1}{q_i}} \pmod p$$

soient tous différents de 1. Pour réaliser ce test il faut disposer de la factorisation de  $p - 1$ , ce qui n'est pas le cas si on prend  $p$  au hasard. On va donc commencer par construire un grand nombre premier  $q$ , puis chercher un nombre premier  $p$  de

la forme  $2kq + 1$  (n'oublions pas que  $p$  sera nécessairement impair), de telle sorte que  $k$  soit suffisamment petit pour pouvoir être factorisé. On peut se demander si lorsque  $q$  est déterminé, on va pouvoir facilement trouver un  $p$  qui convienne. Le théorème de Dirichlet sur la densité des nombres premiers d'une progression arithmétique répond à cette question (cf. [1]). On doit trouver un tel  $p$  en temps moyen de l'ordre  $\ln(p)$ , c'est-à-dire de la taille prévue pour  $p$ .

### 3 Programmation des fonctions utiles

```
#####
## alea(x)
#####

#### entree : nombre de bits x
#### sortie : nombre aleatoire s de x bits

alea:=proc(x)
  local n,s;
  s:=1;
  for n from 1 to x-1
    do
      s:=rand(2)+2*s;
    od;
  RETURN(s);
end;

#####
## nbp(x)
#####

#### entree : nombre de bits x
#### sortie : nombre premier aleatoire p de x bits

nbp:=proc(x)
  local p,a;
```

```

p:=0;
while (p=0)
  do
    a:=alea(x);
    p:=nextprime(a);
    if p>=2^x
      then p:=0;
    fi;
  od;
RETURN(p);
end;

```

```

#####
## trouvepqk(x,y)
#####

```

```

#### entree : nombre de bits x d'un nombre premier q
           nombre de bits y d'un nombre premier p
           ces deux nombres sont tels que q divise p-1
#### sortie : p,q, k tel que p=2kq+1,
####         compteur est le nombre d'iteratons pour
####         trouver p, une fois q construit

```

```

trouvepqk:=proc(x,y)
  local z,k,q,deuxq,p,compteur,u;
  compteur:=1;
  z:=y-x-1;
  q:=nbp(x);
  deuxq:=2*q;
  k:=alea(z);
  p:=deuxq*k+1;
  while (is_prime(p)=0)
    do
      p:=p+deuxq;
      compteur:=compteur+1;
      k:=k+1;
    od;
  end;

```

```

        od;
    u:=[p,q,k,compteur];
    RETURN(u);
end;

#####
## prim(p,q,k)
#####

#### entree : p,q,k de la sortie de la procedure precedente
####          p=2kq+1
#### sortie : Un element primitif alpha de Z/pZ

prim:=proc(p,q,k)
    local f,j,deuxk,t,alpha,y;
    deuxk:=2*k;
    f:=ifactors(deuxk);
    t:=size(f);
    y:=0;
    alpha:=1;
    while (y=0)
        do
            y:=1;
            alpha:=alpha+1;
            if powmod(alpha,deuxk,p)=1
                then
                    y:=0;
                else
                    j:=1;
                    while ((y=1) and (j < t))
                        do
                            if powmod(alpha,iquo(p-1,f[j]),p)=1
                                then y:=0;
                            fi;
                            j:=j+2;
                        od;
                    fi;
                od;
            od;
        od;
end;

```

```
    RETURN(alpha);  
end;
```

## 4 Une session de calcul

Voici une session de calcul avec xcas. On choisit pour  $q$  une taille de 874 bits, et pour  $p$  une taille de 1024 bits. La taille de  $2 * k$  doit donc être de 150 bits, et celle de  $k$  de 75 bits, ce qui fait que  $k$  peut être facilement factorisé.

On cherche  $p$ ,  $q$  et  $k$  tels que  $p = 2kq + 1$ .

```
u := of(trouvepqk, [874, 1024]) =  
[847434702647920618132328639463206504746782160  
61533006449870240203007744627180184397285191  
19069422827000558182905336922721273179395395  
24136266739740570992112117523932497386716123  
70155380821091982967631603450407844018120905  
64132507687818541979694030068100685669863137  
5177540711869789363140337168543688828000521,  
  
10857317176242920389243014706907901175279977  
96868165818524726105268662626627106241984579  
91455823882171774875468182025447927560260986  
85387527048184593342486350918852755182557587  
71344141222851954766466239096853132949745483  
22024312541321124535144134739562182930750467,  
  
390259715587109698032584610171005489694708780,
```

490]

On remarque qu'on a fait 490 itérations avant de trouver  $p$  à partir de  $q$ .  
 On extrait du tableau les paramètres qui vont servir par la suite :

$$p := \text{at}(u, 0) =$$

847434702647920618132328639463206504746782160  
 61533006449870240203007744627180184397285191  
 19069422827000558182905336922721273179395395  
 24136266739740570992112117523932497386716123  
 70155380821091982967631603450407844018120905  
 64132507687818541979694030068100685669863137  
 5177540711869789363140337168543688828000521

$$q := \text{at}(u, 1) =$$

10857317176242920389243014706907901175279977  
 96868165818524726105268662626627106241984579  
 91455823882171774875468182025447927560260986  
 85387527048184593342486350918852755182557587  
 71344141222851954766466239096853132949745483  
 22024312541321124535144134739562182930750467

$$k := \text{at}(u, 2) =$$

390259715587109698032584610171005489694708780

La fonction `prim()` calcule la factorisation de  $k$ . On dispose alors à ce stade de la factorisation complète de  $p - 1$ . La fonction `prim()` calcule alors un élément primitif  $\alpha$  :

$$\alpha := \text{of}(\text{prim}, [p, q, k]) = 58.$$

Le système est mis en place.

## Références

- [1] **Pierre Barthélemy, Robert Rolland, Pascal Véron.** *Cours de Cryptographie*. En cours de rédaction
- [2] **Bernard Parisse, Renée De Graeve.** *Paquetage logiciel Xcas*. [http ://www-fourier.ujf-grenoble.fr/~parisse/giac\\_fr.html](http://www-fourier.ujf-grenoble.fr/~parisse/giac_fr.html)