Université de Limoges

Développement Logiciel Cryptographique

Master 2 - Cryptis

CHABROULIN Sylvain
sylvain.chabroulin@etu.unilim.fr

ANOMAN Don
don.anoman@etu.unilim.fr

Table des matières

Introduction

Le logarithme discret est un outil mathématique utilisé notamment en cryptographie. Le logarithme discret s'utilise principalement pour la cryptographie à clé publique comme chiffrement El Gamal mais aussi pour l'échange de clés Diffie-Hellman.

Definition 1. *(Logarithme Discret) Étant donné un groupe cylclique G d'ordre n, α un générateur de G, et $\beta \in G$ un élément quelconque de ce groupe, on appelle logarithme discret de β en base α l'unique entier x, $0 \leq x \leq n-1$, vérifiant $\beta = \alpha^x$. On note $x = log_\alpha \beta$.*

Pour tout premier p, le groupe $\mathbb{Z}_p^*$ est cyclique d'ordre $p-1$. Le problème du logarithme discret dans $\mathbb{Z}_p^*$, dont $\alpha \in \mathbb{Z}_p^*$ est un générateur, consiste donc pour un élément $\beta \in \mathbb{Z}_p^*$ quelconque, à trouver x, $0 \leq x \leq p-2$, tel que $\beta \equiv \alpha^x (mod p)$.

Lors de ce projet nous étudions et implémentons trois méthodes de détermination du Logarithme discret.

1 Génération aléatoire d'un groupe cyclique

Dans le cadre du projet, il était intéressant de pouvoir exécuter les méthodes sur des exemples de groupes cycliques générés aléatoirement. La question est donc de savoir si on peut calculer un générateur de $\mathbb{Z}_p^*$. Cependant on ne connaît de méthode efficace pour trouver un générateur de $\mathbb{Z}_p^*$ que si on connaît la factorisation de $p-1$. L'idée naïve est de prendre un nombre premier p et de factoriser $p-1$, cependant $p-1$ peut être difficile à factoriser. L'idée utilisé pour le programme est donc de générer une factorisation que l'on nommera $p-1$ jusqu'à ce que p soit premier. Pour cela on utilisera l'algorithme 2 RFN pour générer une liste de nombres premiers $(p_1, ..., p_r)$ extraite d'une liste de nombres aléatoires $(n_1, ..., n_k)$ obtenue par l'algorithme 1 RS. On répète ceci jusqu'à ce que l'on ait $p = 1 + \prod_{i=1}^{r} p_i$ premier. À partir de cette factorisation et de p on calcul un générateur avec l'algorithme 3 "Gegen".
Pour la suite on veut que le nombre premier généré ait k chiffres, on pose donc $m = 2^k$ pour le programme.

1.1 Algorithme RS

Étant donné un $m \in \mathbb{N}$, génère une liste d'entiers aléatoires tels que $n_i \leq n_{i-1}$ jusqu'à que $n_k = 1$.

Algorithm 1 RS

Require: $m \in \mathbb{N}$, $m \geq 2$
Ensure: $(n_1,...,n_k)$ une liste de nombres, tels que $1 \leq n_i \leq n_{i-1}$, avec $i \in [1..k]$ et $n_0 = m$

1: $n_0 \leftarrow m$
2: $k \leftarrow 0$
3: **repeat**
4: $\quad k \leftarrow k + 1$
5: $\quad n_k \leftarrow^R [1, ..., n_{k-1}]$
6: **until** $n_k = 1$
7: renvoyer $(n_1, ..., n_k)$

1.2 Algorithme RFN

L'algorithme RFN génère la factorisation de $p-1$. On commence par obtenir une liste de nombres aléatoires par l'algorithme RS dont on garde les éléments premiers déterminés par un test de primalité. Par la suite on vérifie que $p-1 = \prod_{i=1}^{r} p_i$ a k chiffre. À la sortie de l'algorithme on teste si p est premier, dans le cas échéant on recommence.

Algorithm 2 RFN

Require: $m \in \mathbb{N}$, $m \geq 2$
Ensure: $(p_1, ..., p_r)$ une liste de nombres premiers
1: **repeat**
2: on obtient $(n_1, ..., n_k)$ par RS(m)
3: Soit $(p_1, ..., p_r)$ la sous-liste de premier de $(n_1, ..., n_k)$
4: $y \leftarrow \prod_{i=1}^{r} p_i$
5: **if** $y \leq m$ **then**
6: $x \leftarrow^{R} [1, ..., m]$
7: **if** $x \leq y$ **then**
8: renvoyer $(p_1, ..., p_r)$ et stop
9: **end if**
10: **end if**
11: **until** Sans fin

1.3 Algorithme "Gegen"

Cette méthode a pour but, étant donnée la factorisation $(q_1^{e_1}, ..., q_r^{e_r})$ de $p-1$ avec p premier , de donner un générateur de $\mathbb{Z}_p^*$ et cela en se basant sur la remarque suivante :

Remarque 1.
Un élément g d'un groupe d'ordre h est un générateur de ce Groupe si et seulement si

$$\forall q_i \text{ diviseur de } h \ , \ g^{\frac{h}{q_i}} \neq 1$$

Ainsi pour chaque diviseur q_i de $p-1$: - on génère un certain α tel que $\alpha^{\frac{(p-1)}{q_i}}) \neq 1$, il est exactement d'ordre q_i.
- On définit $\gamma_i = \alpha^{\frac{(p-1)}{q_i^{e_i}}}$ qui est un élément d'ordre exactement $q_i^{e_i}$.
Ainsi on pose $\gamma = \prod \gamma_i$ qui est exactement d'ordre le $PPCM(q_1^{e_1}, ..., q_2^{e_r}) = p-1$

Algorithm 3 Gegen(génération d'un générateur du groupe $\mathbb{Z}_p^*$)

Require: p un nombre premier impair, $(q_1^{e_1}, ..., q_r^{e_r})$ la décomposition en facteurs premiers de $p-1$
Ensure: γ un générateur de $\mathbb{Z}_p^*$
1: **for** $i \leftarrow 1$ to r **do**
2: **repeat**
3: on choisit aléatoirement $\alpha \in \mathbb{Z}_p^*$
4: on calcul $\beta \leftarrow \alpha^{(p-1)/q_i}$
5: **until** $\beta \neq 1$
6: $\gamma_i \leftarrow \alpha^{(p-1)/q_i^{e_i}}$
7: **end for**
8: $\gamma \leftarrow \prod_{t=1}^{r} \gamma_i$
9: renvoyer γ

2 Méthode Pas de Bébé - Pas de Géant de Shanks

La méthode dite des pas de bébé-pas de géant(baby-step giant-step) de Daniel SHANKS permet de calculer le logarithme discret dans n'importe quel groupe cyclique arbitraire G d'ordre n avec une complexité en temps $\mathbb{O}(\sqrt{n})$, et une complexité en mémoire en mémoire en $\mathbb{O}(\sqrt{n})$ également.

2.1 Principe

L'objectif est de calculer le logarithme discret d'un élément β dans un groupe $\mathbb{G}$ d'ordre n, avec α un générateur du groupe $\mathbb{G}$. L'idée est que x (où $\beta = \alpha^x$) peut s'écrire $x = im + j$, avec $m = \sqrt{n}$, $0 \leq i, j < m$, d'où $\beta = \alpha^x = \alpha^{im} \times \alpha^j$ ce qui implique que $\beta(\alpha^{-m})^i = \alpha^j$. On crée donc une table avec l'ensemble des valeurs (j, α^j) triées suivant α^j. Par la suite on va calculer α^{-m} et chercher si $\beta(\alpha^{-m})^i$ est dans la table pour $0 \leq i < m$. Lorsque qu'une correspondance est trouvée, c'est à dire $\beta(\alpha^{-m})^i = \alpha^j$ pour un certain i et j, alors on a trouvé le logarithme de β en calculant $x = im + j$.

2.2 Excercices

Calculer le logarithme discret de $\beta = 2010$ en base α dans $\mathbb{Z}_p^*$ pour chacun des cas suivants :

k	p	α	$log_\alpha \beta$
8	89671807	63009164	76946976
10	7109837807	1104985548	3899193059
12	948699031289	767019334801	137038884108
14	30928521775237	462746514589	

Algorithm 4 Baby-step giant-step algorithm

Require: A group $\mathbb{G}$ of order n, $\alpha \in \mathbb{G}$ a generator, $\beta \in \mathbb{G}$
Ensure: $x = log_\alpha \beta$
1: $m \leftarrow \lceil \sqrt{n} \rceil$
2: For all $0 \leq j < m$, compute α^j and store (α^j, j) in a table
3: Compute $\alpha^{-m}, \gamma \leftarrow \beta$
4: **for** $i = 0$ to $m - 1$ **do**
5: **if** (γ, j) appears in the table for some j **then**
6: return $x = im + j$
7: **end if**
8: $\gamma \leftarrow \gamma.\alpha^{-m}$
9: **end for**

3 Méthode Rho de Pollard

Soient g un générateur du groupe multiplicatif $\mathbb{Z}_p^*$ et $\alpha \in \mathbb{Z}_p^*$.
Il existe $m \in [\![0, p-2]\!]$ vérifiant $\alpha = g^m \ mod\ p$. Notre but est de déterminer ce m .

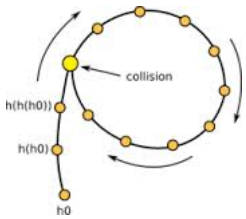
3.1 Idée de base

On se donne une application aléatoire (uniformément distribuée)
$f : \mathbb{Z}_p^* \to \mathbb{Z}_p^*$ et on définit une suite récurrente $(x_i)_{i \in \mathbb{N}}$:

$$\begin{cases} x_0 & = & a \\ x_{i+1} & = & f(x_i) \quad = \quad f^{(i+1)}(a) \end{cases}$$

avec $f^{(i+1)}$ la fonction f itérée i fois.

Du fait que $\mathbb{Z}_p^*$ soit fini il existe u et v tels que $0 \leq u < v$ vérifiant $x_u = x_v$.
On obtient ainsi le rho suivant



Pour un couple (α , β) d' éléments de $\mathbb{Z}_p^*$, la probabilité d'avoir une collision (les éléments sont égaux) est de $\frac{1}{p-1} \simeq \frac{1}{p}$. Ainsi le nombre moyen de couple qu'il faut pour obtenir une collision est p. Or pour avoir p couples, il faut de l'ordre

de $\sqrt{p}$ éléments de $\mathbb{Z}_p^*$. D'où au bout de $O(\sqrt{p})$ applications de la fonction f l'on doit trouver une collision. Ce résultat aurait directement donné le coût de la méthode si on s'autorisait un aussi grand stockage . Or c'est ce que l'on veut éviter. Pour cela , l'on utilise des méthodes de détermination de collisions avec un coût mémoire très négligeable.
Pour appliquer ces méthodes,dans notre cas, nous utilisons la fonction f suivante :

$$f(x) := \begin{cases} \alpha x & si\ x \in S_1 =]0, \lfloor \frac{p}{3} \rfloor] \\ x^2 & si\ x \in S_2 = [\lfloor \frac{p}{3} \rfloor + 1\,,\, \lfloor \frac{2p}{3} \rfloor] \\ g\,x & si\ x \in\ S_3[\lfloor \frac{2p}{3} \rfloor + 1\,,\, p-1] \end{cases}$$

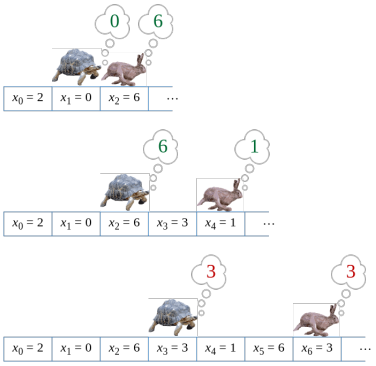
Quant à la suite réccurente, elle est définie comme suit :
On choisit $b_0 \in [\![0, p-2]\!]$ aléatoire et on pose $x_0 := g^{b_0}$

$$x_i = \alpha^{a_i} g^{b_i}; \quad x_{i+1} = \alpha^{a_{i+1}} g^{b_{i+1}} \text{ avec}$$

$$a_{i+1} = \begin{cases} a_i + 1 \mod (p-1) & si\ x_i \in\ S_1 \\ 2\,a_i \mod (p-1) & si\ x_i \in\ S_2 \\ a_i & si\ x_i\ \in\ S_3 \end{cases}$$

$$b_{i+1} = \begin{cases} b_i \mod (p-1) & si\ x \in\ S_1 \\ 2\,b_i \mod (p-1) & si\ x \in\ S_2 \\ b_i + 1 & si\ x \in\ S_3 \end{cases}$$

3.2 Méthode de Floyd



Aussi appelée " méthode du lièvre et de la tortue ", elle consiste à partir du point de départ $x_0 = y_0$, à parcourir le "Rho" avec deux indices : les indices " i" (les x_i) et les indices "2i" (les x_{2i} notés y_i) : $x_{i+1} = f(x_i)$; $y_{i+1} = f(f(y_i))$

Exploitation de la collision

Si une collision est trouvée , cela signifie que :

$$\begin{aligned} & x_i = y_i \quad \Rightarrow \quad x_i = x_{2i} \\ \Rightarrow\ & \alpha^{a_i} g^{b_i} = \alpha^{a_{2i}} g^{b_{2i}} \\ \Rightarrow\ & m(a_i - a_{2i}) = b_{2i} - b_i \ mod\,(p-1) \end{aligned}$$

Posons

$$diffa := (a_i - a_{2i})\ ;\ diffb := (b_{2i} - b_i)\ ,\ et\ d := gcd(diffa, (p-1))$$

Ainsi

$$m \equiv m_0 = \frac{diffb}{diffa} \ \ mod\,[\frac{p-1}{d}]$$

La valeur de m est

$$m = m_0 + k\,\frac{p-1}{d} \text{ avec } 0 \leq k \leq d-1 \text{ et } g^{m_0 + k\frac{p-1}{d}} = \alpha \ \ mod\,p$$

Si la valeur d est trop grande, le test ci-dessus pour les valeurs de k ne pourra être effectué. La méthode aura alors échoué. Pour permettre à nos tests de se dérouler sans limitation nous n'avons pas tenu compte de cette restriction. Une autre méthode , légèrement plus efficace existe :

3.3 Méthode de Brent

Elle est aussi appelée " method of hare and teleporting turtle ". L'idée à retenir est que pour chaque puissance (2^i) de 2 , on positionne la tortue en x_{2^i} et le lièvre parcourt les indices $2^i + 1$ à $2^{i+1} - 1$ et compare les différentes valeurs trouvées avec la tortue. Si une collision est trouvée on l'exploite. Sinon, la tortue se téléporte à la position 2^{i+1}.

Cette méthode nous donne en plus la plus petite valeur de la période.
L'exploitation de la collision reste la même que dans la méthode précédente en remplaçant x_i par la tortue et y_i par le lièvre.

3.4 Exercice

On cherche le logarithme de $\beta = 2010$.

p	α	$log_\alpha \beta$
689740383853	394319546118	453953871584
12263445054821	6959266850417	3570021118442
7100573378083121	4195130626214895	1637143506945755
239108134213568687	111966532448230429	86136716009103701

Ces Différentes méthodes sont en $h \in O(\sqrt(n))$ avec n le nombre premier manipulé. Vu que le nombre de calcul maximal en temps raisonnable est $T \leq 2^{80}$, il faut grosso modo que

$$sqrt(n) \leq 2^{80} \Leftrightarrow$$
$$n \leq 2^{160}$$

Il faut donc que les nombres premiers aient une taille inférieur ou égale à 160 bits.
Les temps des méthodes de Floyd et Brent pour le logarithme de $\beta = 2016$:

p	temps Floyd	temps Brent	Ratio Floyd/Brent
19583585617	0s	0s	1
4340113247891	12s	10s	1.2
830968927049689	27s	51s	0.53
2627342534044607	191s	73s	2.62
27051905839874933	160s	85s	1.88
99697115851186411	639s	422s	1.51
748315331603105447	1436s	2543	0.56

4 Méthode de Pohlig-Hellman

4.1 Idée Générale

Cette méthode consiste, étant donné un groupe G cyclique d'ordre n , à factoriser son ordre en "petits " facteurs premiers si possible. Vu que pour chaque diviseur de l'ordre il existe un unique sous groupe de G d'ordre ce diviseur, on résous le logarithme discret dans chacun de ces sous groupe d'ordre premier en appliquant la méthode rho. Par une recombinaison CRT on retrouve le logarithme dans G :
Dans nôtre cas , le groupe G est $\mathbb{Z}^*_{module}$. Il est paramétré par son le nombre premier ($module$), le générateur g et l'ordre du groupe $n = module - 1$.
Pour chaque diviseur premier p de l'ordre, le sous groupe G_p d'ordre p est paramétré par :
- Le module qui défini les congruences des multiplications
- Le générateur h - L'ordre du sous groupe $(p-1)$ qui défini les congruences

sur les exposants. Cette paramétrisation permet de réutiliser telles quelles les fonctions que nous avons écrites.

4.2 Explication Concrète

Étant donné un nombre premier $module$, g un générateur du groupe cyclique $G = \mathbb{Z}^*_{module}$ et $\alpha \in G$ on cherche $m \in [\![0, module - 1]\!]$ tel que $\alpha = g^m$.
Si $module - 1 := \prod_{i=1..l} P_i^{e_i}$ alors pour chaque i on calcule

$$m_i = m \; mod \; p_i^{e_i}$$

Pour cela on considère la représentation de m_i suivant les puissances de P_i :

$$m_i = k_{e_i-1} P_i^{e_i-1} + ... + k_1 \, P_i + l_0$$

Pour déterminer les k_i : Pour chaque $P = P_i$ Si on a $\alpha_0 = \alpha^{\frac{n}{p}}$ qui est un élément de G_p

$$\begin{aligned} \alpha_0 &= g^{(k_0 + k_1 p + ... + k_{e-1} p^{e-1}) \frac{n}{p}} && (1) \\ &= h^{m_0} * g^{(k_1 + ... + k_{e-1} p^{e-2}) n} && (2) \\ &= h^{k_0} && (3) \end{aligned}$$

Ainsi , dans G_p on calcule le $log_h(\alpha_0) = k_0$. De même si on a $\alpha_1 = (\alpha \times g^{-m_0})^{\frac{n}{p^2}}$ on obtient k_1. Dans nôtre cas nous utilisons un accumulateur pour les calculer au fur et à mesure.
On obtient de cette manière chaque m_i et en utilisant la formule de recombinaison CRT on obtient m.

Coût de la méthode :
Pour chaque sous groupe G_{p_i} on a : e_i calculs de Log $\rightarrow e_i O(\sqrt{p_i})$
$2 * e_i$ exponentiations $\rightarrow e_i * O(log_2 n)$
En tout on a

$$O(\sum_{i=1}^{l} e_i (log(n) + \sqrt{p_i})$$

4.3 Exercice

Pour ce faire on a d'abord factorisé $p-1$. On a constaté qu'il se décompose en petit facteurs.
Ensuite nous avons appliqué la méthode de Pohlig-Hellman pour déterminer l'exposant d'Alice .
Enfin le secret partagé est le message envoyé par Bob à la puissance le secret d'Alice :$y_{Bob}^{s_A}$ qui est :

144340898251692544493573305970760
987470711762256287931885896723250
514475179339966970893441035938649
463908786361867819292133791577007
768702657694761195853684066623421
354387533796843050118773096139622
919716560350080168319976189056144
675387531707482602178213279169497
692571335691533864258936560432045
35060829378

Conclusion

Au terme de ce projet nous avons pu étudier différents aspects du problème du logarithme discret, notamment dans des cas que l'on peut considérer assez facile. Cela nous emmène donc à faire preuve d'une certaine attention lors du choix des paramètres utilisés dans les cryptosystèmes dont la sécurité est basée sur le problème du logarithme discret.

Références

M.Rybowicz *Corps Finis, Master 1 Cryptis - Université de Limoges*
A.Menezes P.van Oorshot S.Vanstone *Chapitre 3, Handbook of Applied Cryptography,CRC Press,1996*
V.Shoup *A Computational Introduction to Nulber Theory and Algebra*
A.Bruasse-Bac *Logarithme discret*
Christophe Clavier *The Discrete Logarithm Problem*
https://en.wikipedia.org/wiki/Cycle_detection