Introduction à la cryptographie
Devoir Maison – Implémentation d'El Gamal

A rendre pour le 6 Mars 2023
M1, Université de Lorraine, 2022/2023
Marine Minier, `marine.minier@loria.fr`

1 Introduction

Ce devoir est à réaliser par groupe de deux étudiant-e-s. Au plus, un seul trinôme sera accepté.

Le rendu se fera sur ARCHE et est en deux parties :

- Un rapport dactylographié au format PDF contenant les réponses aux questions posées dans le sujet. Il vous est demandé de rédiger correctement ces réponses, de détailler votre raisonnement et de justifier vos affirmations.
- Les programmes que vous avez développés. Le `main` de votre programme devra suivre le format de celui décrit en Annexe B. La sortie de votre programme devra être assemblées dans un unique fichier appelé `test.txt`.

L'ensemble des fonctions sera développé dans le langage de votre choix.

Les objectifs de ce DM sont les suivants :

- Comprendre le fonctionnement du système de chiffrement El Gamal.
- Implémenter le chiffrement El Gamal avec des paramètres imposés.
- Tester votre implémentation.
- Tester une propriété particulière d'El Gamal.

2 L'algorithme de chiffrement El Gamal

Le premier but de ce DM est d'implémenter l'algorithme de chiffrement El Gamal.

La sécurité de cet algorithme repose sur la difficulté de résoudre le problème du logarithme discret : étant donné p un grand nombre premier, g un générateur du groupe $(\mathbb{Z}/p\mathbb{Z})^*$ et x un entier tiré au hasard entre 2 et $p-2$, on calcule $X \equiv g^x \bmod p$. Retrouver x étant donnés p, g et X est un problème difficile.

Le système de chiffrement El Gamal fonctionne de la manière suivante, en supposant qu'Alice veut envoyer un message à Bob. Bob commence par construire son couple de clés publique/secrète : (K_p, K_s).

- `KeyGen()` : Bob choisit p un grand nombre premier, g un générateur du groupe $(\mathbb{Z}/p\mathbb{Z})^*$, x un entier pris au hasard entre 2 et $p-2$. Il calcule $X \equiv g^x \bmod p$. Il publie sa clé publique $K_p = (p, g, X)$ et garde secret $K_s = x$.
- `Encrypt()` : Lorsque Alice veut envoyer un message $m \in (\mathbb{Z}/p\mathbb{Z})^*$ à Bob à l'aide de la clé publique de Bob K_p, elle utilise la procédure suivante :
  - Alice tire au hasard un nombre r (différent à chaque fois) compris entre 2 et $p-2$.
  - Elle calcule $y \equiv X^r \bmod p$.
  - Elle transmet à Bob le couple de valeurs $(C \equiv m \times y \bmod p, B \equiv g^r \bmod p)$.
- `Decrypt()` : Bob reçoit ce couple de valeurs (C, B). Pour retrouver le message m, il calcule :
  - $D = B^x \bmod p$.
  - Et ensuite : $C \times (D)^{-1} \bmod p$.
  - Bob retrouve ainsi m car
    - $D = B^x \equiv g^{xr} \equiv (g^x)^r \equiv X^r \equiv y \bmod p$ et
    - $C \times (D)^{-1} \equiv m \times y \times (y)^{-1} \equiv m \bmod p$

Remarques :

- Lors de tous les calculs faits durant les trois étapes, il faut effectuer la réduction mod$p$ après chaque opération élémentaire. En particulier, lors de l'exponentiation modulaire, il faut réduire mod$p$ après chaque étape de l'exponentiation binaire (voir la decription de cette méthode à l'Annexe A).
- Il est à noter également que durant le déchiffrement, il est nécessaire de faire un calcul d'inverse modulaire pour obtenir la valeur $(D)^{-1} \bmod p$.

Le nombre r tiré aléatoirement doit également être différent à chaque chiffrement. Le chiffrement El Gamal n'est considéré comme sûr aujourd'hui que pour des très grands nombres : p doit faire au moins 1024 bits.

3 Avant d'implémenter le chiffrement El Gamal

Le but de ce devoir maison est donc d'implémenter le chiffrement El Gamal.

3.1 Implémentation de fonctions élémentaires

Nous allons y aller pas par pas. Il faut que vous commenciez par choisir un langage de programmation. Une fois, ce langage choisi, comme nous allons travailler avec des nombres de très grande taille, bien plus grande que les longueurs standards utilisées (64 ou 128 bits), il faut trouver la bibliothèque de grands nombres adaptée au langage choisi. Par exemple, si vous décidez de programmer en C, la bibliothèque GMP[1] permet de travailler en multi-précision sur des nombres entiers de grande taille.

[1] voir : `https://gmplib.org/`.

Question 1. Quel langage de programmation avez-vous choisi ? Quelle bibliothèque permettant de gérer des nombres entiers de grande taille allez-vous utiliser ? Quelles sont les opérations implémentées dans cette bibliothèque (multiplication, addition,...) ?

De plus, pour générer les valeurs (très grandes elles aussi), x et r, il faut être capable de générer des nombres aléatoires de très bonne qualité et dits cryptographiquement sûrs.

Question 2. En vous aidant d'internet, donnez la définition d'un nombre aléatoire cryptographiquement sûr. Selon le langage de programmation choisi, donnez le nom de la bibliothèque qui va vous permettre de générer ces nombres aléatoires.

Il est à noter que nous aurons besoin de nombre de 1024 bits, il faut donc s'assurer que le générateur de nombres aléatoires utilisé produit une chaîne binaire aléatoire suffisament longue. Soit il le fait par défaut, soit vous avez besoin d'implémenter une fonction intermédiaire `BigRandom` qui prend en entrée plusieurs petits nombres aléatoires et produit en sortie un nombre aléatoire de 1024 bits.

3.2 Choix de p et g

Le choix d'une partie de la clé publique de Bob, à savoir p et g, est donné dans beaucoup de standards comme par exemple la RFC 2409[2]. En effet, beaucoup de personnes peuvent utiliser le même p et g sans remettre en cause la sécurité du schéma qui repose sur le secret x et le problème du logarithme discret. Nous allons donc utiliser ici les p et g fournis dans la RFC 2409 et appelés dans celle-ci Second Oakley Group. On aura ainsi $g = 2$ et :

$$p = 2^{1024} - 2^{960} - 1 + 2^{64} * ([2^{894}\pi] + 129093)$$

où $[2^{894}\pi]$ est la partie entière du nombre $2^{894}\pi$. Sa valeur en hexadécimal est :

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE65381
FFFFFFFF FFFFFFFF
```

3.3 Inversion d'un élément modulo p

Avant de pouvoir implémenter le chiffrement El Gamal en entier, il faut implémenter le calcul de l'inversion d'un élément $a \in \mathbb{Z}/p\mathbb{Z}$ modulo p. Même si cette fonction existe déjà dans la bibliothèque que vous utilisez, il vous est demandé de l'implémenter vous-même. Comme vous le savez déjà, cela se fait en retrouvant les coefficients de Bézout dans la formule $a \cdot u + p \cdot v = 1$ et via l'algorithme d'Euclide étendu. Il vous faudra donc implémenter une fonction `Euclide()` travaillant sur les grands entiers qui prend en entrée les nombres a et p et renvoie en sortie les coefficients u et v.

Question 3. Implémentez la fonction `Euclide()`. Testez là en vérifiant sur 10000 valeurs différentes de a que $a \cdot u + p \cdot v = 1$. Vous pouvez également vérifier que si une fonction existe déjà dans votre librairie, vous retournez bien les mêmes valeurs.

[2] voir : `https://tools.ietf.org/html/rfc2409#section-6.2`.

3.4 Implémentation de l'exponentiation binaire

Cette méthode rapide d'exponentiation modulaire est décrite dans l'Annexe A. Il s'agit donc d'implémenter ici cette méthode sous la forme d'une fonction `ExpMod()` qui prend en entrée p, g et a et qui renvoie en sortie $A \equiv g^a \bmod p$.

Question 4. Implémentez la fonction `ExpMod()`. Testez-la sur 10000 valeurs différentes.

4 Implémenter le chiffrement El Gamal

Normalement à cette étape vous avez tous les éléments pour implémenter un chiffrement El Gamal complet comme décrit précédemment. Pour cela nous allons implé-menter les trois fonctions décrites précédemment : `KeyGen()`, `Encrypt()` et `Decrypt()`.

- `KeyGen()` est la fonction qui génère les clés de Bob, elle prend en entrée/sortie sans les modifier p et g, elle tire au hasard un x, la clé secrète de Bob et calcule sa clé publique correspondante $X \equiv g^x \bmod p$. Elle renvoie en sortie x et X.
- `Encrypt()` est la fonction qui produit en sortie le couple chiffré $(C \equiv m \times y \bmod p, B \equiv g^r \bmod p)$ en prenant en entrée la clé publique de Bob $K_p = (p, g, X)$ et un message m.
- `Decrypt()` est la fonction de déchiffrement qui prend en entrée (C, B) et la clé secrète de Bob $K_s = x$ et produit en sortie le message m.

Question 5. Implémentez ces trois fonctions. Testez-la en vérifiant que 100 valeurs différentes de m donne bien les déchiffrés attendus (c'est-à-dire de nouveau les m!). Vérifiez également que les 100 r tirés au hasard sont bien différents.

Ca y est, normalement, vous avez un chiffrement El Gamal qui fonctionne !

5 Implémenter la propriété homomorphique du chiffrement El Gamal

Nous allons à présent nous intéresser à vérifier une propriété particulière d'El Gamal, à savoir que le produit de deux chiffrés, si on le déchiffre correctement, donne le produit des deux clairs. Si on écrit cela, il s'agit donc de vérifier la chose suivante :

- Grâce à la fonction `Encrypt()`, on chiffre deux messages m_1 et m_2 pour obtenir les couples de chiffrés correspondants $(C_1 \equiv m_1 X^{r_1} \bmod p, B_1 \equiv g^{r_1} \bmod p)$ et $(C_2 \equiv m_2 X^{r_2} \bmod p, B_2 \equiv g^{r_2} \bmod p)$. Puis on calcule $C \equiv C_1 \times C_2 \bmod p$ et $B \equiv B_1 \times B_2 \bmod p$.
- Grâce à la fonction `Decrypt()`, on déchiffre le couple (C, B) ainsi obtenu pour calculer $m \equiv m_1 \times m_2 \bmod p$. En effet, $B \equiv B_1 \times B_2 \equiv g^{r_1} \times g^{r_2} \equiv g^{r_1+r_2} \bmod p$ et $C = C_1 \times C_2 \equiv m_1 m_2 X^{r_1+r_2} \equiv m_1 m_2 g^{x(r_1+r_2)} \bmod p$ et donc $\frac{C}{B^x} \equiv m \equiv m_1 m_2 \bmod p$.

Question 6. Vérifiez cette propriété en chiffrant deux messages différents m_1 et m_2 avec la fonction `Encrypt()`, puis en calculant $C = C_1 \times C_2 \bmod p$ et $B = B_1 \times B_2 \bmod p$ et enfin en déchiffrant via la fonction `Decrypt()` le couple (C, B) pour récupérer m. Vérifier ensuite que $m = m_1 m_2 \bmod p$.

Ce type de propriété s'appelle une propriété homomorphique (c'est-à-dire qu'on est capable de faire des opérations sur les chiffrés). Elle peut être un avantage dans certains contextes comme le vote électronique où l'on cherche à connaître une somme de votes et non chaque vote individuellement mais cela peut également poser des problèmes notamment en signature quand ce type de propriété peut être utilisée pour forger de fausses signatures.

A Calculer une exponentiation modulaire à l'aide de la méthode dite d'exponentiation binaire

Lorsque l'on cherche à calculer g^a, la méthode naïve consiste à multiplier g par lui-même a fois. Ce n'est pourtant pas la méthode la plus rapide sur les ordinateurs, la meilleure méthode consiste à décomposer a en écriture binaire $a = \sum_{i=0}^{d} a_i 2^i$ où d est la taille en bits de a et où $a_i \in \{0, 1\}$. On a alors : $g^a = g^{a_0}(g^2)^{a_1} \cdots (g^{2^d})^{a_d}$. Il faut ainsi d opérations pour calculer les g^{2^i} et d opérations pour former le produit des $(g^{2^i})^{a_i}$, soit, au total, $2 \cdot d$ opérations. Ainsi l'algorithme le plus rapide pour calculer une exponentiation est le suivant dans sa version récursive qui calcule g^a pour a strictement positif :

$$\texttt{puissance}(g, a) = \begin{cases} g, & \text{si } a{=}1 \\ \texttt{puissance}(g^2, a/2), & \text{si } a \text{ est pair} \\ g \times \texttt{puissance}(g^2, (a-1)/2), & \text{si } a > 2 \text{ est impair} \end{cases}$$

Cet algorithme correspond à une élévation à la puissance classique. Comme nous souhaitons ici travailler modulo p pour calculer $g^a \bmod p$, tous les calculs de l'algorithme récursif doivent être faits modulo p.

B Description du `main()` attendu et des sorties dans `test.txt`

Le `main()` attendu implémentera les 5 premières occurences des tests demandés dans les différentes questions et les résultats de ces tests seront affichés dans le fichier `test.txt`.