Préambule

Le but de ce TP est d'écrire une implémentation simple des algorithmes de chiffrement et signature en utilisant :

— RSA

— ElGamal (repose sur Diffie-Hellman/logarithme discret)

Le choix du langage est Python (un squelette des fonctions est fourni). Il est fortement recommandé de tester unitairement vos fonctions sur des entrées judicieusement choisies au fur et à mesure que vous les écrivez. **Ne modifiez pas les prototypes des fonctions.**

Le squelette de code vous fournit des fonctions utilisées pour l'implémentation.

Modalités : à rendre par binôme par email à la fin de la séance. Selon l'encadrant :

— geraud.senizergues@u-bordeaux.fr

— abel.calluaud@protonmail.com

— pierre.ramet@u-bordeaux.fr

— mihail.popov@inria.fr

1. Arithmétique

1.1 pgcd, inverse modulaire

Q1 Écrire une fonction *pgcd(a, b)* qui renvoie le *pgcd* de a et b, où a et b sont deux entiers > 0. Mettez en œuvre l'algorithme d'Euclide étendu dans une fonction euclide_ext(a, b). Cette fonction retourne un couple de Bezout (u, v) tel que $ua + vb = pgcd(a, b)$.

Q2 Déduire de la question précédente une fonction *inverse_ modulaire(a, N)* calculant un inverse de a modulo N . Pour mémoire, b est un inverse de a modulo N s'il existe un entier k tel que $ab = 1 + kN$, i.e., $ab \equiv 1[N]$.

1.2 Exponentiation modulaire

L'exponentiation modulaire pour un exposant e, une base b et un module N consiste à calculer c tel que $c \in [0, N-1]$ et $c \equiv b^e[N]$.

Q3 Écrire une fonction d'exponentiation modulaire *expo_ modulaire(e,b,n)* en e étapes par le produit des congruences. Exemple : $111^3\ [\,13] = 111 \times 111\ [\,13] \times 111\ [\,13]$. Attention à bien calculer les résidus modulaires à chaque étape pour éviter des calculs trop énormes. Afficher pour chaque exécution de la fonction le nombre d'opérations "×" et modulo effectuées par votre algorithme.

Q4 Écrire une fonction d'exponentiation modulaire rapide utilisant la fonction précédente. Afficher le nombre d'opérations ("×" et modulo) effectuées à chaque appel de la fonction. Pour cela, utilisez le calcul de la représentation binaire de l'exposant. Supposons que l'on veuille calculer b^{17} pour un b donné. Avec la version naïve de la question précédente, on effectue 17 multiplications par b. Maintenant, écrivons 17 en binaire :

$$17_{10} = 10001_2 = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$17_{10} = 2^4 + 2^0$$

On a donc

$$b^{17} = b^{2^4+1} = b^{2^4} \times b$$

Ainsi, en effectuant des élévations au carré successives ($b^2 = b \times b, b^4 = b^2 \times b^2 \ldots$), on peut calculer b^{17} avec seulement 5 multiplications.

2. Tests de primalité

2.1 Crible d'Ératosthène

Une première idée pour obtenir des nombres premiers est de tous les générer jusqu'à une borne n fixée. Le principe du crible d'Erathostène est très simple : on part de la liste d'entiers de 2 à n. A chaque étape, soit a le plus petit entier de la liste. On conserve a comme nombre premier puis on rappelle le procédé sur la liste initiale privée de tous les multiples de a. On s'arrête lorsque que le premier élément de la liste au carré est supérieur à n.

Q5 Implémentez cet algorithme. Quelle est la complexité en temps et en espace de du crible d'Erathostène ?

Une autre stratégie consiste à tirer un entier aléatoirement et tester ensuite s'il est premier. Pour cela, il est nécessaire de déterminer rapidement si un entier donné est premier.

2.2 Test de Fermat

Une première méthode pour tester la primalité d'un nombre est appelée test de Fermat et est fondée sur le petit théorème de Fermat :

Théorème Si p est un nombre premier alors $\forall a \in [1, p-1], a^{p-1} \equiv 1[p]$.

Si un entier n satisfait ce test pour de « nombreux » a, alors n a de fortes chances d'être premier. On choisit donc des a au hasard, on les élève à la puissance $n-1$ et on vérifie la congruence à 1 modulo n.

Q6 Écrire un test de primalité *test_fermat(n,t)* utilisant ce principe. Reprenez votre fonction d'exponentiation rapide. Votre fonction prendra en paramètre un entier n à tester et un entier t représentant le nombre de tests à faire.

Toutefois, cette méthode possède une faiblesse pour les nombres composés dits « de Carmichael » qui ne sont pas premiers mais pour lesquels tous les entiers plus petits passent le test de Fermat.

2.3 Test de Miller-Rabin

Le test de Miller-Rabin (Rabin, en abrégé) permet de contourner le problème des nombres de Carmichael. Il tire parti d'une propriété de l'entier n, qui dépend d'un entier auxiliaire, le témoin. Pour un entier n impair, notons $n-1 = 2^r \times u$ où r et u sont les entiers tels que $r \geq 1$ et u impair. En d'autres termes, r est le nombre maximum de fois que l'on peut mettre 2 en facteur dans $n-1$. Soit $a \in Z_n^{*2}$. Le nombre $a \in [1, n-1]$ est un témoin que n est composé si et seulement si

1. $a^u \not\equiv 1[n]$
2. $\forall i, 0 \leq i \leq r-1 \Rightarrow a^{2^i u} \not\equiv -1[n]$

Si n est premier, il n'existe pas de témoin pour n. Réciproquement, s'il n'existe pas de témoin pour n, alors n est un nombre premier. De plus, lorsque n est impair composé, il admet beaucoup de témoins :

Proposition Pour tout nombre impair composé n, au moins 3/4 des entiers $a \in [1, n-1]$ sont des témoins de Rabin pour n.

Le principe du test est de vérifier qu'un nombre suffisamment grand d'entiers $a \in Z_n^*$ ne sont pas témoins pour n. Le test se déroule comme suit :

1. Éliminer le cas où n est pair
2. Écrire $n-1$ sous la forme $n-1 = 2^r \times u$ avec u impair
3. Choisir aléatoirement $a \in [1, n-2]$
4. Si $pgcd(a, n) \neq 1$ alors n est composé
5. Si a est un témoin alors n est composé, sinon le test est concluant

L'efficacité de ce test découle de la Proposition.

Q7 Écrire une fonction, *find_ru(n)*, qui pour un entier n pair, renvoie un couple (r, u) tel que $n = 2^r \times u$ avec u impair.

Q8 Écrire une fonction réalisant le test de Rabin. Il est fortement conseillé de décomposer le test en sous-fonctions afin de faciliter l'implémentation. Par exemple, écrire une fonction *temoin_rabin(a,n)* qui pour un a et un n donnés, renvoie vrai si a est un témoin de Rabin que n est composé, faux sinon. Ensuite, créer une fonction *test_rabin(n,t)* où n est l'entier à tester et t le nombre de tests à effectuer (ie, le nombre d'appels à la fonction ci-avant avec un a tiré aléatoirement). Il n'est pas interdit de reprendre les fonctions des questions précédentes.

3. Système cryptographique RSA

Le but de cette partie est d'implémenter le protocole RSA.

Q9 Écrire une première fonction *gen_rsa(n)* qui génère une paire de clés RSA. La fonction prendra en paramètre la taille (en nombre de bits) souhaitée pour le "module" N.

Q10 En réutilisant votre fonction d'exponentiation modulaire, écrire les fonctions *enc_rsa(m,e,N)* et *dec_rsa(c,d,N)* de chiffrement et de déchiffrement. On souhaiterait chiffrer des messages et non des nombres. Pour cela, on va considérer un caractère comme un chiffre en base 256 (taille de la table ASCII) qui peut être obtenu par une simple conversion Python. Plus de détails sont donnés dans l'archive de code.

Q11 Créer une fonction qui, à une chaîne de caractère, associe un codage de cette chaîne en entier, puis le chiffre. Créer de même la fonction réciproque.

Q12 A l'aide de votre implémentation, générer des clés dont le module est de taille 512 bits et chiffrer le message «ceci est le message de la question». Pour permettre son déchiffrement, vous fournirez le message chiffré ainsi que la clé secrète.

Q13 Implémentez 2 fonctions *RSAsignature(d,N,m)* et *RSAverification(e,N,m,sig)* qui permettent de signer et vérifier la signature d'un message. Vérifiez votre implémentation sur au moins 2 messages : un premier message correctement signé et un second qui a une signature invalide. N'hésitez pas à ajouter des fonctions intermédiaires.

4. Système cryptographique ElGamal

Le but de cette partie est d'implémenter une version basique du protocole de chiffrement ElGamal.

Q14 Écrire une première fonction *gen_ElGamal_pg(n)* qui génère une paire de valeurs p et g. La fonction prendra en paramètre la taille (en nombre de bits) souhaitée pour le nombre premier p. *Optionnel : Considérez des valeurs autres que* 3 *pour g !*

Q15 Écrivez la fonction *gen_ElGamal_sk_pk(p,g)* qui génère une clé prive et une clé publique a partir de p et g.

Q16 Écrivez la fonction *gen_ElGamal_get_secret(pk_a,sk_a,pk_b,sk_b,p)* qui calcule le secret partagé entre 2 personnes à partir de leurs clés publiques et prives. Assurez-vous que le secret retourné est le même, indépendamment de la clé publique utilisée.

Q17 Écrivez les fonctions *enc_ElGamal(m,secret,p)* et *dec_ElGamal(c,secret,p)* qui respectivement chiffrent et déchiffrent un texte en clair à partir du secret partagé. (*détails dans CM 9 slide 50*).

Q18 En utilisant des clés éphémèrales et vos fonctions de chiffrement/déchiffrement, envoyez 2 messages entre 2 personnes.

Q19 Implémentez 2 fonctions *elgamalsignature(g,p,sk,m)* et *elgamalverification(g,p,r,s,m,pk)* qui permettent de signer et vérifier la signature d'un message. Vérifiez votre implémentation sur au moins 2 messages : un premier message correctement signé et un second qui a une signature invalide. N'hésitez pas à ajouter des fonctions intermédiaires.

5. Des attaques

Dans les questions qui suivent, le chiffrement RSA utilisé est celui que vous avez programmé dans les sections précédentes.

Q20 Déchiffrez le message c dans le code. **Indice : N est un nombre petit.**

Q21 Déchiffrez le message c dans le code. **Indice : $N1$ et $N2$ partagent un facteur.**

Q22 Le même message m a été chiffré trois fois avec trois clés différentes. Retrouvez ce message. **Indice : le même exposant a été utilisé pour les 3 chiffrements. De plus $N1$, $N2$ et $N3$ sont premiers entre eux !**

Q23 - bonus / difficile Retrouver le message $m1$, qui a été chiffré avec la clé $e1$ modulo $N1$. Vous connaissez aussi ($e2$, $d2$, $N2$). Aide : chercher une racine carrée non triviale de $1[N1]$. On pourra utiliser :

Théorème : Soit N un entier composé impair. Soit a un élément aléatoire de Z_N^*, et soit $w = 2^r * u$, u impair tel que $a^w \equiv 1[N]$. La séquence

$$a^u, a^{2*u}, a^{2^i*u}, ..., a^{2^r*u}$$

contient avec probabilité $\geq 1/2$ une racine carrée non triviale (c'est-à-dire non congrue à 1 ou -1 $[N]$) de $1[N]$.