RAPPORT

Question 1:

Pour le chiffrement AES nous utilisons la librairie javax.crypto. Nous avons tout d'abord une fonction GenerateKey qui va générer une clé de 128 bits de manière aléatoire en utilisant un KeyGenerator fourni par la librairie javax.crypto. La méthode generateIV produit un vecteur d'initialisation (IV) de 16 octets. Qui va permettre que le chiffrement de chaque message soit unique. La méthode encrypt va prendre en paramètre un message, une clé en un vecteur. On va chiffrer le message par blocs de 16 bits. Pour chaque blocs on va :

- Nous utilisons AES pour chiffrer le compteur initialisé avec le IV.
- XOR du bloc avec le bloc chiffré du compteur.
- Incrémentation du Compteur

Pour le déchiffrement, nous effectuons les mêmes étapes.

Question 2:

Pour le système RSA nous avons utilisé le type Biginteger pour le chiffrement des grands nombres. D'abord deux grands nombres premiers aleatoire (partKey_p, partKey_q) ont été généré, ses premialité ont été vérifié, et aussi la clef publique défini par key = partKey_p*partKey_q. La méthode 'encryptMsg' prendre en paramètre une message en Biginteger et renvoie le message crypté, pour ça la valeur ϕ(key) a été calculé ϕ(key)=(p-1)*(q-1) et un exposant de chiffrement 'e' a été choisi aléatoirement tel que (i) 1 < e < ϕ(key) et (ii) e et ϕ(key) sont premiers entre eux, cette dernière condition est vérifiée avec la méthode auxiliaire 'euclideEntendu', finalement le calcul de chiffrement est fait en utilisant la formule (message)^e mod key. La méthode 'decryptMsg' réalise l'opération contraire à 'encryptMsg', en prenant un exposant de déchiffrement 'd' calculé à partir de 'euclideEntendu', et en effectuant la même formule d'avant.

Question 3:

Nous avons décidé d'utiliser la méthode de Pollard pour factoriser de grands nombres premiers. Comme le protocole RSA est basé sur des objets de type `BigInteger`, la fonction `pollardFactor` prend en argument un `BigInteger n`, puis initialise deux variables, `x` et `y`, avec des valeurs de départ, ainsi qu'une variable `d` qui vérifie la primalité des valeurs de `x` et `y`.
Le programme utilise une boucle `while` dans laquelle la variable `y` avance deux fois plus vite que la variable `x`, avec la fonction f(x2−1)mod n. Cela permet de détecter un cycle. À chaque tour, la variable `d` est mise à jour, et on vérifie si le PGCD de `(x - y, n)` est différent de 1. Si c'est le cas, alors nous avons trouvé un facteur non trivial de `n`.

Question 4 :

Le protocole de transfert inconscient repose sur le schéma RSA. Dans ce protocole, Alice doit générer des messages via la fonction `aliceGenereMessage([x0, x1, ..., xn], int n)`, qui prend un paramètre `n` représentant le nombre de messages à générer. Une alternative consiste à ce qu'Alice reçoive un tableau de chaînes de caractères

contenant les messages, puis elle les convertit en objets `BigInteger` afin de les chiffrer avec RSA via la fonction `aliceChiffreMessage([c0, c1, ..., cn])`.

Une fois les messages préparés par Alice, Bob doit choisir un nombre entre 0 et `n-1` avec la fonction `bobChoisieB()`. Ensuite, Bob doit calculer `v = cb + k^e` à l'aide de la fonction `bobCalculeV()`. Dans cette étape, Bob génère une clé privée (key) et la sauvegarde. La variable `cb` représente le message chiffré d'Alice à l'indice `b`, `key` est la clé privée de Bob, et `e` est l'exposant du protocole RSA.

Alice, ayant reçu `v`, peut calculer les valeurs `ki = (v - ci)^d` pour chaque `i` entre 0 et `n`. Elle utilise la fonction `aliceCalculeKi()` pour ce calcul, qui comprend une boucle de 1 à `n` dans laquelle elle soustrait `v` au message chiffré `ci` et le déchiffre avec RSA. Elle peut ensuite calculer les valeurs déchiffrées `m0 = x0 + k0` et `m1 = x1 + k1`, et ainsi de suite, où `xi` correspond aux messages originaux d'Alice, et `ki` sont les valeurs calculées précédemment.

Après avoir calculé le tableau `m`, Bob n'a qu'à récupérer le message `m[b]` et en soustraire sa clé privée pour obtenir le message déchiffré qu'il recherchait.

Question 5 :

Les circuits sont représentés par une hashmap. Chaque nœud est représenté par un nom (INA, OUTB… etc) et par un type de nœud (InputNode…etc). La fonction minCircuit génère le circuit min avec la taille des valeurs d'entrée sur n bits. Le circuit généré par la fonction prend les bits d'entrées de A et de B, les compare 1 à 1 puis définit lequel est le plus petit ou égal à l'autre. La fonction generateCircuit génère le circuit pour calculer a<=b. Puis on continue de générer le circuit en définissant le minimum strict entre 2 bits grâce à l'information calculée juste avant et grâce aux bits d'entrée.

Question 6 :

La fonction topologicalSort effectue un tri topologique du circuit et la fonction evaluateCircuit renvoie les valeurs pour chaque nœud en fonction des valeurs données en entrée. La fonction evaluateCircuit prend un circuit et des valeurs d'entrées. Puis pour chaque nœud, prend comme valeur d'entrée la valeur du nœud qui le précède et renvoie comme valeur de sortie, le calcul effectué sur la valeur (ou les valeurs) qu'il a récupéré (AND, NOT… etc). La fonction testExhaustif prend en entrée un circuit et la taille des valeurs d'entrée et fait des tests exhaustifs pour chaque valeur possible des valeurs d'entrée. Dans le main du fichier Tests, on crée un circuit min8 et on effectue une batterie de tests exhaustifs pour des entrées de 8 bits. Pour effectuer des tests sur des entrées de tailles différentes, il suffit de générer un circuit min de la taille voulue et d'appeler la fonction de test avec le circuit et la taille des entrées.

Question 7 :

La machine virtuelle est représentée par la classe VirtualMachine. Elle s'initialise en prenant le nombre de noeud n d'un circuit et initialise les variables nécessaires à savoir, n+4 pour Alice, n+2 pour Bob et 2 buffer un de Alice vers Bob et l'autre dans l'autre sens. Dans cette même classe, on a un jeu d'instruction pour la machine virtuelle (push, pop… etc) ainsi que 3 autres fonctions, une qui permet d'exécuter les instructions d'Alice et l'autre celles de Bob. La fonction execute quant à elle teste la VM sur un circuit donné. Il est possible de tester la VM pour un autre circuit en donnant à la fonction execute un autre circuit et des valeurs d'entrées correspondant au circuit. Dans le main de la classe Tests 2 tests sont effectués, un sur le circuit min et un sur le circuit <=.
La classe CircuitCompiler possède une unique fonction qui selon un circuit donné, génère une liste d'instruction pour Alice et une pour Bob. En fonction d'un nœud donné, les instructions correspondantes sont ajoutées aux listes. Cette fonction est utilisée dans la fonction execute de la VM.
Les instructions mises dans les listes sont représentées par des classes java (principalement Expression et Instruction) permettant à la VM de reconnaître facilement une instruction à réaliser.

Question 8 :

Pour préparer le circuit, Alice doit d'abord parcourir le circuit et ajouter des clés AES à chaque nœud, sauf pour ceux ayant une sortie OUTA ou OUTB, où elle doit chiffrer les deux valeurs de vérité possibles, soit "0" soit "1". Si le nœud parcouru est INA, Alice doit sauvegarder l'une des clés AES pour l'envoyer à Bob via l'attribut `shareKey`.

Une fois toutes les clés initialisées, il faut préparer les tables brouillées pour chaque nœud du circuit, sauf INA, INB, OUTA, et OUTB. Une table brouillée contient une `ArrayList<String>`, qui représente les éléments du circuit brouillé, ainsi qu'un nœud. La fonction `genereTB` est utilisée pour générer ces tables et elle effectue plusieurs vérifications :

- Si la sortie du nœud est OUTB, un indicateur `bool` est défini comme `true`.
- La fonction vérifie ensuite le type de nœud (par exemple, NOT, AND, XOR).
- Si la sortie est OUTB, alors Alice doit chiffrer directement les valeurs de vérité "0" ou "1".
- Sinon, elle doit chiffrer la clé AES du nœud en tenant compte de toutes les possibilités de la table de vérité.

Si le nœud est de type NOT, Alice chiffre sa clé AES avec l'opposé de la valeur de vérité de son entrée (car un NOT n'a qu'une seule entrée).

Si le nœud est de type AND ou XOR, Alice doit effectuer un double chiffrement :

- Elle chiffre d'abord la clé ou la valeur de vérité avec le premier input.
- Ensuite, elle rechiffre avec le second input, en respectant la table de vérité des deux portes.

Ainsi, Alice peut envoyer à Bob la clé `shareKey` associée à INA, ainsi que la liste des tables brouillées.

Question 9 :

Bob récupère la `shareKey` et la liste des tables brouillées. Il doit maintenant obtenir sa clé `k` sur le nœud INB en utilisant le protocole de transfert inconscient, via la fonction `OT`. Il ajoute ses clés, la clé partagée d'Alice (`shareKey`), et la nouvelle clé `k` à une liste de clés, `keyList`.

Ensuite, Bob doit évaluer les circuits brouillés (qui sont organisés dans l'ordre d'un tri topologique) avec la fonction `eval_circuit_brouiller`. Cette fonction parcourt tous les nœuds du circuit, sauf INA, INB, OUTA, et OUTB. Pour chaque circuit brouillé, il appelle la fonction `decryptCircuit`. Cette dernière teste chaque message chiffré du circuit brouillé avec toutes les clés disponibles dans `keyList`. Elle déchiffre selon la table de vérité du nœud, en fonction de son type : NOT, AND, ou XOR.

Si le déchiffrement réussit, le message obtenu est soit "0" soit "1". Dans ce cas, Bob peut ajouter la clé de déchiffrement à `keyList` pour l'utiliser dans les étapes suivantes du circuit. Il faut faire attention lors du déchiffrement des nœuds AND et XOR, car il faut effectuer un double déchiffrement pour récupérer la clé AES ou le message correspondant.

Lin Pascal

Houard Romain

Xia Didier

Fonseca Rodrigues

Loiseau Poilpré Zoé