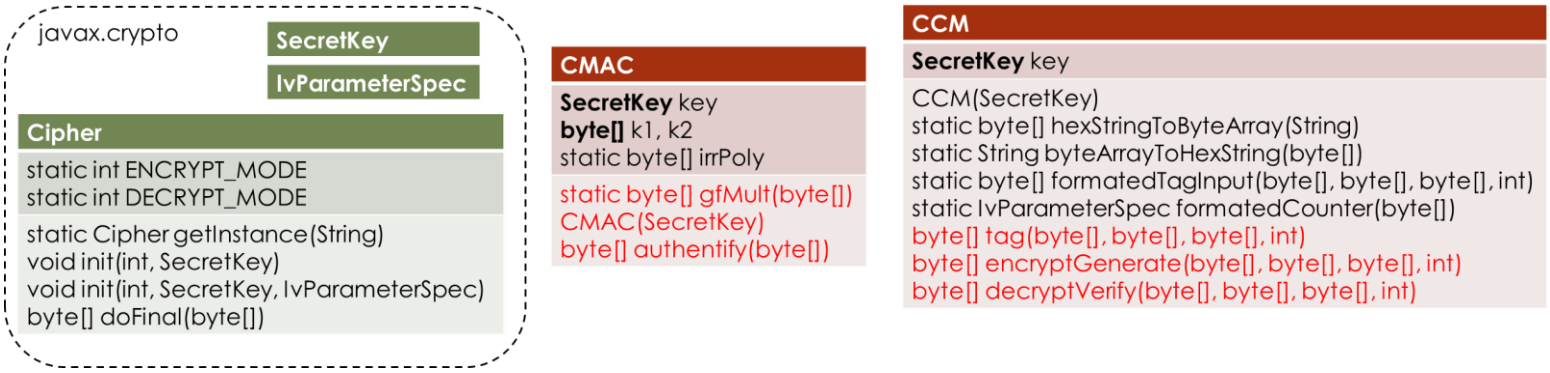
Sécurité des réseaux TP4

Le TP consiste à implémenter les fonctions d'authentification CMAC et le mode de chiffrement-authentification CCM. Ces fonctions sont décrites dans les slides du cours 8, que vous pouvez consulter sur mon site web. Il existe également une documentation plus exhaustive sur le site de la NIST (CMAC, CCM).

Voici le diagramme UML des classes du TP :



I) Classes de javax.crypto

Pour effectuer ce TP, vous allez utiliser des classes de la bibliothèque standard *javax.crypto*, qui implémentent un certain nombre d'algorithmes de chiffrement standard, ainsi que d'autres fonctions cryptographiques comme des fonctions de hachages, … Les classes de cette bibliothèque font les calculs directement en binaire sur des tableaux d'octets *byte[]*. Les données à authentifier/chiffrer seront donc des tableaux d'octets, et il faudra traiter les données octets par octets. Les blocs considérés dans CMAC et CCM devront avoir des tailles qui sont des multiples de 8 pour rentrer dans ces tableaux.

Avant de discuter des différentes classes que vous allez utiliser dans cette bibliothèque, je vais présenter les opérations standards qui peuvent être effectuées sur ces octets de type *byte*. Il est possible d'initialiser un *byte* par un entier entre 0 et 255, dont la valeur en binaire sera stockée dans l'octet. Par exemple, si a est un byte, et l'opération $a = 17$ est effectuée, alors la valeur stockée dans a sera 00010001. Il est possible également d'utiliser la représentation en hexadécimale, en faisant précéder la valeur par $0x$. Ainsi l'opération $a = 0xA5$ va affecter la valeur 10100101 au *byte* a.

Le OU exclusif bit à bit correspond à l'opérateur ^. Ainsi, si a et b sont des *bytes*, alors $a\hat{}b$ sera un *byte* formé à partir du OU exclusif bit à bit entre a et b. Le ET logique bit à bit correspond à l'opérateur &, et le OU logique bit à bit correspond à l'opérateur |. Ces deux opérateurs peuvent permettre de changer un des 8 bits d'un octet. Par exemple pour mettre à 1 le dernier bit (le bit de poids faible, ou autrement dit le bit le plus à droite) d'un *byte* a, il est possible d'utiliser l'opérateur | en faisant l'opération $a \mid 0x01$ (où $0x01$ correspond à l'octet 00000001). Autre exemple, pour mettre à 0 le premier bit (le bit de poids fort, ou autrement dit le bit le plus à gauche) d'un *byte* a, il

est possible d'utiliser l'opérateur & en faisant l'opération $a \ \& \ 0x7F$ (où $0x7F$ correspond à l'octet 01111111).

Enfin l'opérateur << permet de faire un certain nombre de décalages à gauche (donné en paramètre). Ainsi, si a est un *byte*, l'opération $a \ll 2$ va renvoyer un *byte* où les bits de a ont été décalés de deux positions à gauche. Les valeurs des deux bits de poids forts de a sont perdues ici, et les bits insérés à droite sont des 0. Cette opération est équivalente à une multiplication par 2^i modulo 256, où i est le nombre de décalage désiré.

A) Classe SecretKey

Il s'agit d'une classe permettant de stocker des clés de chiffrement. Etant donné que je vais vous donner dans le TP ces clés de chiffrement en paramètre de votre fonction, vous n'avez pas besoin de connaitre la façon de créer ces clés, mais si vous êtes curieux, vous pouvez consulter la documentation https://docs.oracle.com/javase/8/docs/api/javax/crypto/spec/SecretKeySpec.html. Vous pouvez également regarder des tutoriaux expliquant l'utilisation des classes de la bibliothèque javax.crypto, comme par exemple https://www.baeldung.com/java-cipher-class.

B) Classe IvParameterSpec

La classe *IvParamSpec* permet de créer des vecteurs initiaux (IV) utilisés, entre autres, par certains modes de chiffrement (comme CTR, même s'il s'agit d'un nonce). Etant donné que je vous donne dans ce TP une fonction qui permet d'obtenir cet objet à partir d'un tableau de bytes représentant le nonce, vous n'avez pas besoin de connaitre la façon de créer ces objets, mais si vous êtes curieux vous pouvez consulter la documentation https://docs.oracle.com/javase%2F7%2Fdocs%2Fapi%2F%2F/javax/crypto/spec/IvParameterSpec.html.

C) Classe Cipher

La classe *Cipher* permet de créer des objets capables d'effectuer le chiffrement et déchiffrement, en utilisant des algorithmes de chiffrement symétrique (comme AES) ou asymétrique (comme RSA), et en utilisant différent modes de chiffrement (comme ECB ou CTR). Si vous souhaitez la consulter, vous pouvez trouver une description exhaustive des fonctions de cette classe dans la javadoc présente à l'adresse https://docs.oracle.com/javase%2F7%2Fdocs%2Fapi%2F%2F/javax/crypto/Cipher.html.

Un objet de cette classe peut être créé à l'aide de la fonction statique *getInstance*, qui prend en paramètre une chaine de caractères décrivant l'algorithme de chiffrement à utiliser, le mode de fonctionnement utilisé, et le type de « padding » à utiliser lorsque le bloc final n'est pas complet. Dans le cadre de ce TP, vous aurez à utiliser les chaines de caractères « AES/ECB/NoPadding » et « AES/CTR/NoPadding », pour l'utilisation de l'algorithme de chiffrement AES avec les modes de fonctionnement ECB et CTR.

La fonction *init* permet d'initialiser un certain nombre de paramètres de l'objet de type *Cipher*, et en particulier la clé de chiffrement utilisée, le vecteur initial utilisé par le mode de chiffrement, ainsi que le type d'action effectuée (par exemple chiffrement ou déchiffrement). La première fonction que je vais vous présenter prend en paramètre un entier, qui indique quel type d'action est effectuée. Sa

valeur pourra être, dans le cadre de ce TP, soit la constante *Cipher.ENCRYPT_MODE*, lorsque c'est le chiffrement qui devra être effectué, soit la constante *Cipher.DECRYPT_MODE*, lorsque c'est le déchiffrement qui sera effectué. Le second paramètre est la clé de chiffrement (ou déchiffrement, en fonction du premier paramètre) utilisée. La seconde fonction *init* possède un paramètre supplémentaire, qui est un vecteur initial (IV), objet de la classe *IvParameterSpec*. Cet IV sera utilisé, entre autres, comme valeur de compteur initial lorsque le mode de fonctionnement est CTR.

La dernière fonction utile dans la cadre de ce TP est la fonction *doFinal*, qui va effectuer l'action associée à l'objet *Cipher*, c'est-à-dire le chiffrement ou le déchiffrement, en mode ECB ou CTR, en fonction des paramètres donnés à la fonction *init* auparavant. Cette action de chiffrement ou déchiffrement est effectuée sur la paramètre pris en entré de cette fonction, qui est un tableau d'octets (*byte*) représentant la chaine de bits à chiffrer ou déchiffrer. Cette fonction renvoie le résultat du chiffrement/déchiffrement, sous la forme d'un tableau d'octets.

II) Classe CMAC

Il s'agit de la classe qui va calculer le Cipher-based Message Authentication Code (CMAC) d'une séquence d'octets. Cette classe possède un attribut de type *SecretKey*, qui est la clé de chiffrement utilisée par l'algorithme de chiffrement AES. Il y a également deux autres attributs, *k1* et *k2*, qui sont des tableaux d'octets (*byte*). Ces deux éléments correspondent aux deux clés K_1 et K_2, qui sont générées à partir de la clé de chiffrement, et qui vont être mélangées au dernier bloc de données à l'aide de l'opérateur OU exclusif. La clé K_1 est utilisée lorsque la chaine de bits à coder n'a pas besoin de padding (taille facteur de 128 bits), et la clé K_2 est utilisée lorsque le padding est nécessaire.

La classe possède un attribut statique *irrPoly*, qui est un tableau d'octets (*byte*) contenant le polynôme irréductible utilisé lors des multiplications dans $GF(2^{128})$. Pour être plus précis, le bit de poids fort, qui est le 129ème bit (correspondant au monôme de degré 128), a été exclu, à la fois pour faire tenir le nombre sur 128 bits (et donc 16 octets), mais aussi parce que ce bit n'a pour rôle, dans la multiplication polynomial modulo le polynôme irréductible, que d'annuler le 129ème bit qui serait passé à 1. Or lorsque vous allez effectuer la multiplication par x, qui consiste à un décalage à gauche (opérateur $\ll$ décrit plus haut), le bit le plus à gauche de chaque octet sera perdu, et va donc tout simplement disparaitre. Il faudra néanmoins ajouter (opération de OU exclusif effectué par l'opérateur ^) les autres bits de ce polynôme irréductible lorsque le bits à gauche de l'opérande de la multiplication était à 1 (voir cours 6 slide 7 sur les multiplications dans l'arithmétique polynomiale modulo un certain polynôme irréductible).

La classe *CMAC* possède une fonction statique *gfMult* (à implémenter), qui va effectuer une multiplication polynomiale dans $GF(2^{128})$ en utilisant comme modulo le polynôme irréductible $m(x) = x^{128} + x^7 + x^2 + x + 1$. Comme indiqué dans le paragraphe précédent, les coefficients de ce polynôme sont stockés dans la variable statique *irrPoly*. La fonction prend en paramètre un tableau d'octets (*byte*) contenant la chaine de bits à multiplier, et va retourner un tableau d'octets contenant le résultat cette multiplication de la chaine par x modulo $m(x)$. Vous pouvez utiliser la règle de calcul du décalage à gauche conditionnel (décrit dans le slide 7 du cours 6) pour effectuer cette opération.

La classe *CMAC* possède un unique constructeur (à implémenter), qui prend en paramètre la clé secrète utilisée par l'algorithme de chiffrement. Ce constructeur va enregistrer la clé prise en paramètre dans l'attribut *key*, et va créer les deux sous-clés K_1 et K_2 et les stocker dans les attributs

K1 et *K2*. Notez que vous aurez besoin de l'algorithme AES, ainsi que de la multiplication polynomiale ici, et donc de la fonction *gfMult* décrite au paragraphe précédent.

Enfin, la classe *CMAC* possède une fonction *authentify* (à implémenter), qui va prendre en paramètre la chaine de bits à authentifier (sous la forme d'un tableau de *bytes*), et qui va retourner le CMAC associé à cette chaine, en utilisant la clé contenue dans l'attribut *key*. Le CMAC est retourné sous la forme d'un tableau de 16 octets (*byte*) contenant sa valeur.

III) Classe CCM

Il s'agit de la classe qui va implémenter l'algorithme de chiffrement-authentification Counter with Cipher Block Chaining-Message Authentication Code (CCM). Cette classe possède un unique attribut *key*, qui est la clé de chiffrement utilisée, à la fois par l'algorithme de chiffrement, mais aussi par l'algorithme d'authentification. Cet attribut est un objet de la classe *SecretKey* qui pourra être utilisé par l'algorithme de chiffrement AES.

La classe *CCM* possède un unique constructeur, qui prend en paramètre la clé de chiffrement utilisée, et initialise l'attribut *key* avec cette valeur. La classe possède également deux fonctions statiques qui permettent de transformer une chaine de bits en chaine de caractères contenant sa représentation en hexadécimal (*byteArrayToHexString*), ainsi qu'une autre fonction qui fait l'opération inverse (*hexStringToByteArray*), c'est-à-dire transformer une chaine de caractères contenant la représentation en hexadécimal d'une chaine de bits en tableau d'octets. Vous pourrez utiliser, entre autres, ces fonctions pour afficher les valeurs intermédiaires des blocs de données. Notez que ces fonctions peuvent être également utilisées lors de l'implémentation de CMAC.

La classe *CCM* possède une fonction *formatedTagInput*, qui va prendre en paramètre le nonce, les données associées ainsi que le texte à chiffrer (tous trois sous la forme de tableaux de bytes), ainsi que la taille du tag (sous la forme d'un entier qui représente le nombre d'octets). Cette fonction va renvoyer l'ensemble des blocs utilisés pour construire le tag servant à l'authentification. Le formatage effectué par cette fonction, qui n'a pas été donné en cours, est décrit dans https://csrc.nist.gov/pubs/sp/800/38/c/upd1/final. Vous n'avez pas besoin de connaitre les détails de ce formatage pour effectuer ce TP, mais ce formatage est nécessaire pour que les exemples marchent (donc vous devez utiliser cette fonction même si vous ne savez pas ce qu'il y a dedans).

La classe *CCM* possède également une fonction *formatedCounter*, qui prend en paramètre un tableau d'octets (*byte*) contenant la valeur du nonce, et retourne la valeur initiale du conteur utilisé dans le mode CTR sous la forme d'un objet de type *IvParameterSpec*. Le compteur retourné respecte un formatage qui n'a pas été décrit dans le cours, mais que vous pouvez trouver dans la documentation du NIST https://csrc.nist.gov/pubs/sp/800/38/c/upd1/final. Vous n'avez pas besoin de connaitre les détails de ce formatage pour effectuer ce TP, mais ce formatage est nécessaire pour que les exemples marchent (donc vous devez utiliser cette fonction même si vous ignorez son contenu).

La classe *CCM* possède une fonction *tag* (à implémenter), qui va calculer la valeur du tag ajouté à la suite des données avant l'étape de chiffrement (voir slide 18 du cours 8). Cette fonction va prendre en paramètre le nonce, les données associées ainsi que le texte à chiffrer (tous trois sous la forme de tableaux de *bytes*), ainsi que la taille du tag (sous la forme d'un entier qui représente le nombre d'octets). Cette fonction renvoie le tag sous la forme d'un tableau d'octets (dont la taille correspond au paramètre entier donné en paramètre de la fonction). Cette fonction va utiliser la fonction

formatedTagInput pour transformer les inputs en une succession de bloc, et appliquer l'algorithme DAA sur ces blocs pour calculer le tag.

La classe *CCM* possède une fonction *encryptGenerate* (à implémenter), qui va appliquer l'algorithme de chiffrement-authentification. Cette fonction prend en paramètre le nonce, les données associées ainsi que le texte à chiffrer (tous trois sous la forme de tableaux de *bytes*), ainsi que la taille du tag (sous la forme d'un entier qui représente le nombre d'octets). Elle renvoie le message chiffré accompagné d'un tag (lui aussi chiffré), le tout sous la forme d'un tableau d'octets (*byte*). Cette fonction va utiliser la fonction *tag* pour calculer le tag à ajouter après les données. Elle va également faire appel à la fonction *formatedCounter* pour obtenir la valeur de conteur initiale, qui pourra être utilisée avec un objet de type *Cipher* permettant le chiffrement en mode CTR.

Enfin, la classe *CCM* possède une fonction *decryptVerify* (à implémenter), qui va effectuer le déchiffrement et vérifier l'intégrité du message à partir du tag. Cette fonction prend en paramètre le nonce, les données associées ainsi que le texte chiffré (tous trois sous la forme de tableaux de *bytes*), ainsi que la taille du tag (sous la forme d'un entier qui représente le nombre d'octets). Cette fonction va utiliser la fonction *formatedCounter* pour obtenir la valeur de conteur initiale, qui pourra être utilisée avec un objet de type *Cipher* permettant le déchiffrement en mode CTR. Le tag pourra être recalculé à l'aide de la fonction *tag*. Si le tag recalculé correspond au tag déchiffré, alors le message en clair sera retourné sous la forme d'un tableau d'octets (*byte*). Sinon, un tableau d'octets vide (0 octet) sera renvoyé pour signifier que l'authentification a échouée.

IV) Travail à effectuer

Vous devez implémenter l'ensemble des fonctions indiquées en rouge dans le diagramme UML. Vous pourrez utiliser la fonction *main* de la classe *CCM* pour tester votre code. Les quatre messages utilisés pour tester la classe *CMAC* correspondent à des exemples donnés dans la documentation du NIST https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/examples/aes_cmac.pdf (voir les exemples #1, #2, #3 et #4). Vous pourrez afficher les blocs de données (qui sont des tableaux d'octets) après chaque opération en utilisant la fonction *byteArrayToHexString* de la classe *CCM*. Cette fonction va renvoyer une chaine de caractères décrivant les valeurs des octets en hexadécimal. Les quatre messages utilisés pour tester la classe *CCM* correspondent à des exemples donnés dans la documentation du NIST https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/examples/aes_ccm.pdf (voir les exemples #1, #2 et #3).