Université de Rennes

Projet de complément en cryptographie

Implémentation de l'AES dans le language C

Auteur :
Moyse Waly

Contents

1 Introduction **2**

2 Notice d'utilisation **3**
2.1 Liste des options disponibles 3
2.2 Tests de performance des modes de chiffrement 4
2.3 Valeurs par défaut et incompatibilité des options 5
2.4 Exemples de commandes 6

3 Description de l'implémentation **7**
3.1 File parser 7
3.2 Expansion de la clé 7
3.3 Algorithme de l'AES 7
3.4 Les modes de chiffrements 8

4 Conclusions **9**

1 Introduction

Le projet présenté ici a été réalisé dans le cadre de ma première année en Master de *Mathématiques de l'information, cryptographie*. En effet, il nous a été demandé, pour la fin de l'année scolaire, d'implémenter l'AES dans le langage de programmation C.

Cette implémentation devait à minima être capable de chiffrer un texte (contenu dans un fichier) en mode ECB. Néanmoins, nous étions invités à faire une version plus complète comprenant l'AES 192 et 256, ainsi que différents modes de chiffrement par blocs (CBC, GCM...).

Déterminé à ne pas faire le "minimum syndical", j'ai entrepris de commencer la programmation de ce projet le plus tôt possible, afin de me laisser plus de temps pour implémenter le maximum d'options possibles. Cela m'a également permis de me réconcilier avec le langage de programmation C. Celui-ci m'avait fait subir beaucoup de difficultés lors du premier semestre de la même année.

Cette implémentation comprend donc l'AES 128, 192 et 256 ainsi que les modes de chiffrements ECB, CBC, CFB et GCM. Tous les algorithmes de chiffrements ont été réalisés à partir des descriptions présentes dans les standards de ces protocoles. Il y a cependant un léger doute sur le fonctionnement de certaines parties du mode GCM qui seront mentionnées plus tard.

Ce document comporte donc une notice d'utilisation comportant une description plus complète de toutes les options disponibles lors de l' exécution du code. De plus, il contient des exemples de commandes réalisables. La partie suivante est une description du fonctionnement de mon implémentation. Enfin, ce rapport mentionne les difficultés que j'ai rencontrées avec cette dernière, en revenant également sur les points améliorables du code.

2 Notice d'utilisation

2.1 Liste des options disponibles

Depuis un terminal et une fois dans le bon répertoire, faites la commande 'make' ou 'make all' afin de générer tous les fichiers binaires. Il est ensuite possible d'exécuter le programme en tapant './AES', suivit des options souhaitées. Les options disponibles sont les suivantes :

- -h ou - -help : cette option affiche une liste des options, ainsi qu'une courte description de chacune d'entre elle. L'aide de -h est moins complète que la notice de ce document. Mettre cette option dans une ligne de commande ne créera aucun chiffrement et rendra inutile toutes les autres options.

- -k KEY ou - -key KEY : cette option permet à l'utilisateur de fournir sa clé privée pour le chiffrement ou le déchiffrement du texte. L'argument KEY doit être la clé en format hexadécimal sans espace. Le format binaire n'est pas supporté. L'argument peut être sous la forme 0xKEY ou KEY. Il n'est pas nécessaire de préciser la taille de la clé, car le programme le déduira de lui-même. Néanmoins, il renverra une erreur si la taille n'est pas de 128, 192 ou 256 bits. Cette option ne peut pas être utilisée plusieurs fois dans la même commande.

- -a AUTH ou - -authdata AUTH : cette option permet à l'utilisateur de fournir une 'authentification data' lors d'un déchiffrement ou un chiffrement en mode GCM. Le format de l'argument AUTH est le même que pour KEY, à la différence qu'il peut prendre une taille quelconque tant qu'il ne dépasse pas 128 bits. Cette option ne peut pas être utilisée plusieurs fois dans la même commande.

- -g TAG ou - -authtag TAG : cette option permet à l'utilisateur de fournir le tag d'authentification nécessaire pour un déchiffrement en mode GCM. Le format de l'argument TAG est le même que pour KEY, à la différence qu'il doit faire exactement 128 bits. Cette option ne peut pas être utilisée plusieurs fois dans la même commande.

- -v IV ou - -init_vector IV : cette option permet à l'utilisateur de fournir un vecteur d'initialisation utilisé pour un déchiffrement ou un chiffrement en mode CBC, CFB ou GCM. Le format de l'argument IV est le même que pour AUTH (la taille doit être exactement de 128 bits.). Cette option ne peut pas être utilisée plusieurs fois dans la même commande.

- -f NAME ou - -filename NAME : cette option permet à l'utilisateur de donner le nom du fichier qu'il souhaite utiliser pour un chiffrement ou un déchiffrement. L'argument NAME doit être le nom du fichier (par exemple fichier.txt).

- -m MODE ou - -mode MODE : cette option permet à l'utilisateur d'indiquer le mode qu'il souhaite utiliser pour le chiffrement ou le déchiffrement. MODE doit donc être une chaîne de caractère entre EBC, CBC, CFB ou GCM. Une erreur sera renvoyée si l'argument n'est pas un élément de cette liste. Cette option ne peut pas être utilisée plusieurs fois dans la même commande.

- -c ou - -cipher : cette option permet à l'utilisateur d'indiquer qu'il veut effectuer un chiffrement.
- -d ou - -d decipher : cette option permet à l'utilisateur d'indiquer qu'il veut effectuer un déchiffrement.
- -t ou - -test : cette option permet à l'utilisateur qu'il souhaite effectuer un test de performance (plus d'information en prochaine section).
- -i ou - -hexa_input : cette option permet à l'utilisateur d'indiquer que le contenu du fichier qu'il souhaite chiffrer ou déchiffrer est en Hexadécimal. Les écritures hexadécimales commençant par 0x ne sont pas supportées par cette implémentation. Le fichier doit contenir uniquement des espaces, des sauts de lignes, des tabulations, des chiffres ou les lettres 'a', 'b', 'c', 'd', 'e', 'f', 'A', 'B', 'C', 'D', 'E', 'F'. Si cette option n'est pas utilisée, le programme convertira chaque caractère du fichier en sa valeur dans le code ASCII.
- -o ou - -hexa_output : cette option permet à l'utilisateur d'indiquer qu'il souhaite que le texte du fichier renvoyé par le programme (la version chiffrée ou déchiffrée du texte) soit en hexadécimal. Si cette option n'est pas utilisée, le programme écrira le fichier selon le code ASCII.
- -b SIZE ou - -block_size SIZE : cette option n'est utile que si l'utilisateur souhaite utiliser le mode CFB. Elle permet d'indiquer la taille des blocs à utiliser pour chiffrer ou déchiffrer le message. SIZE doit être écrit en décimale et indiquer la taille des blocs en bits. Cette taille ne doit pas dépasser 128 et doit être un multiple de 8. Cette option ne peut pas être utilisée plusieurs fois dans la même commande.

2.2 Tests de performance des modes de chiffrement

Comme indiqué précédemment, il existe une option (-t) permettant de tester la vitesse de chiffrement. Ce test est disponible pour chacun des 4 modes, il est d'ailleurs nécessaire d'en indiquer un. Il consiste en une mesure du temps pris par le programme pour chiffrer un texte 100 fois avec les paramètres donnés. Ce test ne compte pas le temps pris par le programme à lire le fichier, mais seulement la phase de chiffrement. Le test ne chiffre de plus pas vraiment 100 fois le même texte : le premier chiffrement est fait sur le texte d'origine, le second chiffrement est fait sur le premier chiffré et ainsi de suite. Il est possible de choisir la plupart des paramètres pour ce test (la clé, le vecteur initial..) mais il n'est pas possible de choisir le fichier sur lequel on effectue le test (option -f). Ce test sera fait sur le fichier alice.txt présent dans le dossier. Une utilisation de l'option -f dans ce cas ne sera pas prise en compte.

2.3 Valeurs par défaut et incompatibilité des options

Il est possible que l'utilisateur fasse une commande sans préciser certains paramètres nécessaires au chiffrement. Dans certains cas, il y a des paramètres par défauts. Ci-dessous, les paramètres par défaut utilisés s'ils ne sont pas précisés par l'utilisateur :

- La clé par défaut est 0x000102030405060708090A0B0C0D0E0F et correspond donc a de lAES 128.
- Le vecteur initial et l'authentification data par défaut sont tous les deux 0x00000000000000000000000000000000.
- Le fichier utilisé pour le chiffrement ou déchiffrement est alice.txt par défaut.
- La taille des blocs pour le mode CFB est 128 par défaut.

Chaque fois que le programme utilise une valeur par défaut, il préviendra l'utilisateur depuis le terminal dans le cas où cela serait un oubli de sa part. Le programme renverra cependant une erreur si l'utilisateur souhaite effectuer un déchiffrement en mode GCM sans fournir de tag d'authentification.

Il est également possible qu'une commande comporte des paramètres non-nécessaires au chiffrement ou au déchiffrement (par exemple un vecteur d'initialisation pour le mode ECB). Dans ces situations, le programme ne renverra pas d'erreur, mais il préviendra l'utilisateur depuis le terminal les informations fournies n'ayant pas été utilisées.

Ce programme ne comporte pas de mode par défaut. Il est donc nécessaire de préciser le mode souhaité dans une commande.

Il est également nécessaire d'utiliser une des options '-t', '-c' ou '-d' pour préciser quelle opération est voulue par l'utilisateur. (L'option -c ne s'activera pas par défaut). Les trois options '-t', '-c' ou '-d' ne sont pas compatibles entre elles. Utiliser les option '-c' et '-d' ou les options '-t' et '-d' dans la même commande renverra une erreur. Il n'est pas nécessaire d'utiliser l'option '-c' lorsque l'option '-t' est utilisée. Néanmoins, utiliser les deux ensemble ne renverra pas d'erreur.

Utiliser l'option '-b' pour un autre mode que le CFB ne renverra pas d'erreur, mais l'argument de l'option ne sera pas pris en compte.

Il n'est pas possible de choisir le nom du fichier que renverra le programme. Ce fichier s'appellera 'ciphered_text.txt' en cas d'utilisation de l'option '-c', et 'deciphered_text.txt pour l'option '-d'.

2.4 Exemples de commandes

Voici quelques commandes valides et une description de leur signification :

1) ./AES -t -m ECB

Le programme va effectuer un test de 100 chiffrements du fichier alice.txt en mode ECB avec la clé par défaut.

2) ./AES -c -m CBC -k 0x3101A20304C506076F094ACA0C0D0E0F -v 0x1234567 -f montext.txt -o

Le programme va effectuer un chiffrement en mode CBC du texte contenue dans le fichier montxt.txt (si existant) en utilisant la clé privée 3101A20304C506076F094ACA0C0D0E0F et le vecteur initial 0x1234567 (qui sera paddé avec des 0). Le fichier rendu en sortie sera le résultat du chiffrement en format hexadécimal.

3) ./AES -d -m GCM -k 3101A20304C506076F094ACA0C0D0E0F -v 1234567 -a 7777777 -g 31AAA203BBC50CC76F544ACF2C0D650F -i -f montext.txt

Le programme va tenter de déchiffrer le contenu du fichier montext.txt, qui est en hexadécimal. Avec la clé 0x3101A20304C506076F094ACA0C0D0E0F, le vecteur initial 0x1234567, l'authtag 0x7777777 (ces deux derniers sont paddés avec des 0) et le tag d'authentification 0x31AAA203BBC50CC76F544ACF2C0D650F.

Si le tag fourni est incorrect, la fonction renverra une erreur.

3 Description de l'implémentation

Cette partie est une explication globale du fonctionnement de l'implémentation.

Elle ne parlera pas du parser d'argument présent dans la fonction main, car son fonctionnement est celui décrit par la notice d'utilisation. La seule chose qu'il est nécessaire de savoir est que celui-ci permet de réunir tous les paramètres nécessaires au bon fonctionnement du code.

3.1 File parser

Les fonctions file parser ont pour but de stocker le texte présent dans le fichier voulu dans un tableau d'unsigned char 2D.

Il y a deux file parser : un pour lire un fichier en hexa et l'autre pour lire un fichier en ASCII.

La fonction s'occupe de renvoyer une erreur si le fichier n'existe pas. Dans les deux cas, les fonctions commencent par compter le nombre de caractères du texte afin de savoir la mémoire à allouer pour tout stocker. Dans le cas d'un fichier en hexa, chaque pair de caractère numérique va correspondre à un octet qui sera stocké sous forme d'un unsigned char (en effet, ce type à une taille d'un octet). Dans l'autre cas, chaque caractère sera stocké sous la forme de l'unsigned char correspondant. Ce code part donc du principe qu'il n'y a pas de caractère "bizarre" ou unicode dans le document. Il est probable que le résultat obtenu dans le cas contraire soit quelque peu hasardeux.

Chaque partie du texte correspondant à 16 unsigned char est stockée dans un tableau. Celui-ci sera lui-même stocké dans un tableau qui comportera tous les blocs de 16 unsigned char du texte dans l'ordre.

Des pointeurs en paramètre des fonctions permettront de stocker certaines informations en mémoire comme le nombre de blocs, ainsi, crées ou la longueur totale du chiffré nécessaire pour le mode GCM.

3.2 Expansion de la clé

La clé utilisée doit être agrandie afin de créer des clés de rondes. La taille de cette clé dépend de celle de la clé d'origine. La fonction ExpandKey renvoie donc la clé agrandie correspondante en suivant à la lettre le standard de l'AES. Le tout est stocké dans un tableau d'unsigned char. La fonction s'occupe également d'allouer la mémoire pour cette clé.

3.3 Algorithme de l'AES

Quel que soit le mode utilisé, il est nécessaire de pouvoir chiffrer un bloc de taille 128 bits en suivant l'algorithme de l'AES. La fonction cipher_block s'en occupe donc en prenant un bloc et la clé agrandie et en modifiant le block en lui appliquant l'algorithme utilisé pour l'AES. Chaque opération de l'AES est représenté par une fonction qui applique l'opération a un état. L'algorithme prend également la taille de la clé en compte, car l'algorithme change entre les 3 tailles de clés possibles.

La fonction decipher_block suis le même principe, mais en utilisant les opérations inverses également décrites dans le standard de l'AES.

Il n'y a pas de remarque particulière à faire sur cette partie mise à part que la multiplication utilisée pour l'opération MixColumns a été faite en utilisant une table de correspondance calculé depuis sage maths.

3.4 Les modes de chiffrements

Comme dit précédemment, cette implémentation comprend 4 modes de chiffrements : ECB, CBC, CFB et GCM.

Chaque mode possède son opération de chiffrement et de déchiffrement. Les modes ECB et CBC sont particulièrement basiques, car ils consistent simplement en des chiffrements de blocs, et des XoR pour le mode CBC.

Les modes CFB et GCM ont cependant été plus complexes à implémenter.

En effet, le mode CFB nécessite une taille de bloc différente de 128, ce qui nécessitait donc de rajouter un paramètre dans le file parser afin de faire des blocs de la bonne taille. J'ai décidé pour simplifier d'empêcher les blocs d'une taille qui n'est pas un multiple de 8. En effet, dans le cas inverse, il aurait fallu que je travaille avec des portions de caractères ce qui aurait rendu la chose beaucoup plus compliquée. Ce choix m'a permis de faire une fonction efficace que ce soit en temps d'exécution, mais également en taille.

Le principal challenge a été d'implémenter le mode GCM. En effet, il s'agit d'un mode permettant d'authentifié une personne voulant déchiffrer un message.

La difficulté du mode GCM réside dans la multiplication qui y est faite dans un corps particulier. Ce corps étant trop gros, il n'est pas envisageable de définir les multiplications avec des tables de correspondance comme pour MixColumn. Il faut donc coder la multiplication en utilisant des opérations logiques. Pour simplifier ce processus, j'ai converti les blocs de 128 bits que je multiplie entre eux en une liste de 2 long long int (faisant chacun 128 bits). Cette conversion a été définie dans les deux sens afin de pouvoir passer de mes blocs d'unsigned char à ces long long int et inversement. J'ai ensuite pu définir la multiplication avec moins de difficulté. J'ai également implémenté une méthode plus rapide se basant sur un pré calcul. En effet, le mode GCM utilise la multiplication en ayant un terme H qui est toujours le même pour un texte donné. Le pré calcul fait sur H consiste à calculer la multiplication de H par toutes les suites de 128 bits comportant exactement un seul bit non nul les résultats sont ainsi stockés dans un tableau. La multiplication entre H et un autre terme K consistent alors en des XoR des éléments de notre liste résultante en fonction des bits non nul de H.

Une fois cette multiplication définie, il suffisait de suivre l'algorithme du standard GCM. J'ai cependant quelques doutes quant à mon utilisation de certains paramètres notamment lenA lenC et le vecteur initial. Il est possible que mon implémentation ne soit pas totalement fidèle au standard et donne donc un résultat différent de celui d'une implémentation professionnelle de GCM.

Le mode GCM dans le cas d'un chiffrement donnera le tag d'authentification calculé dans le terminal de l'utilisateur. Le mode GCM dans le cas d'un déchiffrement renverra une erreur si le tag donné par l'utilisateur n'est pas valide.

4 Conclusions

Ce projet m'a permis de nettement m'améliorer en C et m'a également fait comprendre l'efficacité de ce langage, notamment dans le cadre de la cryptographie. Mon code n'est sans doute pas encore très professionnel, mais j'ai remarqué une nette amélioration par rapport au semestre dernier. Certains aspects de mon implémentation sont sans doute encore améliorables. Après plusieurs tests, je pense avoir corrigé la plupart des erreurs graves et j'espère qu'il ne comporte aucun bug majeur. Je reste cependant satisfait d'avoir réussi à faire une implémentation comportant toutes les options que je souhaitais implémenter.