Projet PX222 maths/info:
Implémentations d'AES en C (Extension)

Grenoble INP – Esisar – année 2022-23

On rappelle que vous devez avoir réalisé l'AES 128, 192 et 256. Vous devez également avoir une API claire qui permettent d'utiliser directement l'AES dans un autre programme. Vous devez donc avoir 2 fonctions définies comme suit :

```
int aes_encrypt(char* data, int size, char* key, int keysize);
int aes_decrypt(char* data, int size, char* key, int keysize);
```

Le chiffrement et déchiffrement se font donc en place sur le pointeur `data` de taille `size` qui doit être multiple de 16. La fonction renvoie 0 si tout c'est bien passé et 1 sinon (par exemple si l'allocation mémoire pour l'expansion de la clé à échoué). Lorsque vous avez terminé l'AES avec les 3 tailles de clé, apposez une étiquette dans votre répertoire git avec :

```
git add <vos fichiers .c .h>
git commit -m "V1 fonctionnelle" # ou adaptez
git tag V1
git push --tags
```

Cela permettra d'y revenir plus facilement par la suite.

Modes de fonctionnement de l'AES

Il existe plusieurs mode de fonctionnement de l'algorithme AES. Ici, nous ne nous intéresserons qu'aux modes *Electronic CodeBook* (ECB) et *Cipher Block Chaining* (CBC). Il en existe bien d'autres que vous pouvez trouver en ligne (https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation) et qui ont également leurs propres spécifications. Implémentez le mode CBC et rajouter à votre API un paramètre `mode` de type `int` qui prendra comme valeur 0 pour ECB et 1 pour CBC.

Pour bien saisir la différence entre les 2 de manière visuelle, vous pouvez chiffrer une image au format bitmap en conservant l'entête de fichier non chiffré. Le format bitmap contient la taille du fichier à l'offset 0x2 sur 4 octets en petit boutiste et l'offset des données à l'offset (du fichier) 0xA sur 4 octets petit boutiste[1]. Pour les personnes sous Windows, assurez-vous d'ouvrir les fichiers en mode binaire et non en mode texte.

Pour mesurer objectivement l'aléa dans un fichier, on utilise l'entropie, concept emprunté à la physique statistique. L'entropie en base 2 se calcule comme suit :

$$H_b(X) = \sum_{i=0}^{2^n} P_i \log_2(\frac{1}{P_i}) = -\sum_{i=0}^{2^n} P_i \log_2(P_i)$$

avec P_i la probabilité de l'octet ayant la valeur i et X la variable aléatoire correspondant à notre fichier. Pour calculer l'entropie en considérant les octets, vous pouvez réaliser l'histogramme des octets du fichier et en déduire la probabilité de chaque octet.

[1] Si vous manquez de temps, le code permettant de ne chiffrer que les données peut vous être fourni. Il faudra cependant l'adapter à votre API.

Ici, on travaille sur des octets, donc $n = 8$ bits. L'entropie est donc comprise entre 0 et n. Plus l'entropie est élevée, plus chaque mot transmet de l'information et plus le fichier est "aléatoire". Une entropie de n correspond à une suite aléatoire, une entropie nulle correspond à la répétition des mêmes données.

Constatez visuellement la différence entre ECB et CBC sur la même image qui utilise un fond à peu près uniforme et avec des bords d'objets nets. Calculez l'entropie de l'image source et des 2 images chiffrées. Conclure sur les 2 modes de chiffrement.

Code C conforme

Le C est un langage bas niveau comparé à Haskell. Il offre beaucoup de latitude au programmeur et permet même d'insérer du code assembleur directement. La contrepartie est que le C a de nombreux comportements indéfinis et éventuellement des fuites mémoires. Ces comportements indéfinis (*UB* pour *Undefined Behavior*) sont de potentielles failles de sécurité et les fuites mémoire peuvent également être exploitées par un attaquant.

Pour trouver les comportements indéfinis dans votre code, vous pouvez utiliser UBSAN (Undefined Behavior SANitizer). De même pour les fuites mémoire, vous pouvez utiliser ASAN (Address SANitizer). Pour cela, compilez votre programme avec les flags suivants (à ajouter dans votre makefile) :

```
gcc -g -fsanitize=address -fsanitize=undefined -fsanitize=shift
   -fsanitize=shift-exponent -fsanitize=shift-base
   -fsanitize=integer-divide-by-zero -fsanitize=unreachable
   -fsanitize=vla-bound -fsanitize=null -fsanitize=return
   -fsanitize=signed-integer-overflow -fsanitize=bounds
   -fsanitize=bounds-strict -fsanitize=bool -fsanitize=enum ... ##
   compilation
gcc ... -lasan -lubsan ## link
```

À l'exécution, le sanitizer vous informe des comportements dangereux qu'il détecte. Notez dans votre carnet de bord les erreurs et remédiez y. Il est conseillé de toujours utiliser ces flags lors du développement et de les supprimer uniquement lors de la livraison finale au client.

Mesure de performances

De nombreuses façons de mesurer la performance existent. En général, on ne s'intéresse qu'au temps d'exécution. Un code permettant la mesure du temps d'exécution vous est fourni. Dans cette partie, il faut retirer les éventuels flags de debug tels qu'ASAN. Le plus important est le profilage qui permet de savoir où le programme passe le plus de temps. Pour cela, on utilise l'outil valgrind :

```
valgrind --tool=callgrind --callgrind-out-file=aes.out
   --dump-instr=yes --collect-jumps=yes monprogramme mes arguments
kcachegrind aes.out
```

L'outil kcachegrind permet de visualiser le graphe d'appel et notamment de voir où le programme est le moins efficace. Par exemple, il est possible que vous ayez réalisés l'expansion de la clé à chaque chiffrement d'un bloc alors qu'on peut ne la faire qu'une seule et unique fois. Pour estimer le débit, vous pouvez utiliser la fonction `clock_t clock(void);` définie dans l'entête `time.h`.

Notez dans votre carnet de bord chaque étape avec par exemple le graphe d'appel et le débit obtenu pour chaque version du code qui doit correspondre à un commit ou à une étiquette

du git.

Optimisation via le compilateur

Le compilateur propose différents niveaux d'optimisation. Par défaut, c'est le niveau -O2, mais on peut utiliser -O3 voire -Ofast. Cependant, le flag -Ofast peut casser le respect de la conformité au standard C, il est donc déconseillé. Vous pouvez comparer le temps d'exécution entre chaque niveau d'optimisation, de -O0 à -O3. De plus, vous pouvez aussi jeter un oeil au code généré :

```
objdump -M intel -dS monprogramme > monprogramme.dump
```

Le flag `-M intel` indique la syntaxe à utiliser, `-d` de ne vidanger que les sections exécutables et `-S` d'entrelacer le code assembleur avec votre code C (ne fonctionne que si le programme a été compilé avec les symboles de debug `-g`).

Réagencement du code

Le compilateur est capable d'optimiser à travers les appels de fonction sous réserve qu'il a accès au code source de la fonction appelée dans le même fichier. Vous avez peut-être organiser votre code comme en Haskell avec une vue plus mathématique. En C, cela a moins de sens et empêche même le compilateur d'optimiser. En particulier, si vous avez dans un fichier les fonctions qui manipulent des octets, dans un autre fichier les fonctions qui manipulent des mots et dans un troisième fichier, les fonctions qui manipulent les états de 16 octets, alors le compilateur ne peut qu'optimiser les fonctions indépendamment.

Pour aider un peu le compilateur, vous pouvez regrouper dans un premier temps les fonctions octet et les fonctions mot ensembles. Mesurez les performances obtenues et observez le code généré. Puis dans un second temps, regroupez les fonctions mot et état, mesurez les performances obtenues et observez le code généré.

Réécriture du code

Lorsque vous manipulez des octets, vous n'utilisez que 8 bits sur des registres de 32 ou 64 bits. Il est possible de réaliser certaines fonctions de l'AES directement sur un entier de 32 bits plutôt qu'en considérant 4 octets de 8 bits. C'est par exemple le cas des fonctions ShiftRows, RotWord, etc.

Introduction à la programmation vectorielle

Les processeurs x86 possèdent des extensions vectorielles permettant de manipuler directement des données sur 128, 256 voire 512 bits. Dans le cadre de l'AES, a priori on n'a besoin que des registres 128 bits. En pratique, l'AES étant un algorithme extrêmement répandu, Intel propose directement une extension matérielle réalisant les tours de l'AES. La liste de toutes les instructions spécifiques et extension est disponible sur le site d'Intel (https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html). Cherchez les extensions AES que vous pouvez utiliser et vérifier que votre processeur les possède avec `cpuid` ou `lscpu`. N'hésitez pas à `grep` la sortie. Implémentez le chiffrement, puis le déchiffrement avec ces extensions et finissez par l'expansion de la clé. Comparez les performances après chaque étape et observez le code généré.

Valentin Egloff