Analyse par corrélation de consommation de courant sur l'AES

Oberthur Technologies Crypto Group

Master SFPN, Université Pierre et Marie Curie

Le but de ce TP est d'utiliser la technique d'analyse par corrélation de consommation de courant ou CPA (Correlation Power Analysis) sur un algorithme AES implémenté sur une carte à puce.

Contexte. Suite à la capture d'un espion, les autorités ont retrouvé une carte à puce qui sert vraisemblablement à chiffrer les communications entre l'espion et son employeur. Après enquête, on a réussi à mettre la main sur le code embarqué sur la carte. Cependant, la clé secrète utilisée étant chargée sur la carte indépendamment du code, cela ne suffit pas pour la retrouver. Avec toutes ces informations, on a demandé aux étudiants du Master SFPN de retrouver la clé secrète embarquée dans la carte.

Éléments à disposition. Pour préparer l'attaque, nous disposons d'un environnement complet avec:

- le code assembleur 8051 embarqué sur la carte `AES.a51`;
- un assembleur 8051 `as8051`;
- un "pretty-printer" de 8051 pour visualiser les adresses de chaque instruction dans le code `pp8051`;
- un simulateur de 8051 pour exécuter du code pas à pas `vm8051`.

Ces outils permettront de repérer dans l'algorithme quels sont les points d'intérêts, c'est à dire, les moments de l'exécution de l'algorithme qui vont manipuler des données sensible. Une documentation succinte est disponible dans `ext/doc/VM8051_userguide.pdf`.

La consommation de courant de la carte varie en fonction de la donnée qui est manipulée. C'est l'analyse de cette consommation qui va permettre de monter une attaque CPA. Pour cela, nous disposons d'un exécutable qui simule la consommation de courant de la carte `target_AES`. La "trace" de l'exécution peut être stockée dans un fichier de trace avec l'option `-o`. Par exemple

```
./target_AES 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
```

va calculer le chiffré du message `0x000102030405060708090A0B0C0D0E0F`, tandis que

```
./target_AES -o trace.dat 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
```

va aussi calculer le chiffré, mais va également stocker la trace de consommation de l'exécution dans le fichier `trace.dat` au format décrit dans l'Annexe C.

Une fois les trace acquise, il faudra monter l'attaque. Pour cela nous avons à disposition

- une bibliothèque pour analyser les traces `libtraceio.a`. Elle contient entre autres des fonctions pour lire un fichier de trace (voir Annexe C).
- une bibliothèque pour produire un fichier de corrélations `libCPA.a`. Un fichier de corrélation va contenir le résultat d'une CPA (voir Annexe D).

Ces deux bibliothèques seront utilisées par le fichier `attackCPA.c` que nous devrons compléter. Une fois les corrélations obtenues, nous pourrons utiliser le programme `CPA_plotkeys` pour "visualiser" la meilleure hypothèse de clé, ou bien le programme `CPA_guess_key` pour l'afficher.

1 Préparation

Décompresser l'archive `EnvSCA_etu.tar.gz` dans son répertoire de travail. Aller dans le répertoire `EnvSCA_etu` puis taper

```
PATH="$PWD/ext/bin:$PATH"
```

On aura ainsi facilement accès à tous les outils nécessaires au TP.

Question1
Lors d'un AES, quelle fonction dépendant d'un octet de clé et d'un octet de message pouvons-nous cibler pour une attaque CPA ? Étudier le code `AES.a51`. À quel(s) endroit(s) de l'algorithme manipule-t-on une telle donnée ? À quelle adresse de code cela correspond-il (utiliser `pp8051`)?

Question2
A l'aide du simulateur `vm8051`, repérer les cycles correspondants aux instructions repérées à la question précédente pour chaque octet de clé.

Question3
Une étude du composant utilisé par l'espion laisse à penser que la consommation de courant est liée au poids de Hamming de la donnée manipulée (*i.e.* le nombre de bits à un d'un octet). Dans le fichier `attackCPA.c`, écrire le corps de la fonction de prédiction pour la donnée sensible que vous avez ciblée.
`double prediction_func (uint8_t key_hyp, uint8_t msg)`
(si besoin, on pourra utiliser la SBox de l'AES donnée dans le fichier `SBox_AES.h`).

Question4
Dans le fichier `attackCPA.c` Écrire le corps la fonction qui calcule le coefficient de corrélation linéaire entre deux vecteurs. On pourra s'aider de la formule donnée à l'Annexe A.
`double linear_correlation (double *leakages, double *predictions, unsigned int N)`.

Question5
Compléter le corps de la fonction qui effectue une CPA sur un ensemble de traces (voir l'Annexe B). Une fois terminé, on utilisera la commande `make` pour compiler et produire l'exécutable `attackCPA`.

Question6
Maintenant, nous pouvons monter l'attaque à proprement parler. Acquérir 1000 traces d'exécution à partir de `target_AES` pour 1000 entrées différentes sur 16 octets. Les traces d'exécution seront rangées dans le répertoire `data/exec_AES` et seront nommées `trace1.dat` à `trace1000.dat`. Pour générer aléatoirement les entrées, on pourra utiliser le programme `random_bytes` qui prend en argument le nombre d'octets à produire.

Question7
Utiliser votre programme `attackCPA` pour retrouver le premier octet de clé. Le programme fonctionne de cette manière:

```
> ./attackCPA <indice octet de clé> <moment à cibler> <fichier de sortie> <fichier de trace> ...
```

Le nombre de fichier de traces à utiliser n'est pas limité (on utilisera à bon escient le caractère spécial `*` du shell). On mettra les corrélations obtenues dans le fichier `data/AES_cpa0.dat`.

Question8
Trouver les 16 octets de clé de l'espion.

Question9
Modifier l'implémentation de l'AES dans le fichier `AES.a51` de manière à résister à la précédente attaque.

A Coefficient de corrélation linéaire

Le coefficient de corrélation linéaire de Pearson est l'outil central pour effectuer une CPA. Ce coefficient est défini pour deux vecteurs de N éléments $X = (X_1, \ldots, X_N)$ et $Y = (Y_1, \ldots, Y_N)$. On note $\overline{X}$ (resp. $\overline{Y}$) la moyenne des éléments du vecteur X (resp. du vecteur Y):

$$\overline{X} = \frac{1}{N} \sum_{i=1}^{N} X_i$$

Le coefficient de corrélation de Pearson $\rho(X, Y)$ vaut

$$\rho(X,Y) = \frac{\sum_{i=1}^{N} \left(X_i - \overline{X}\right)\left(Y_i - \overline{Y}\right)}{\sqrt{\sum_{i=1}^{N} \left(X_i - \overline{X}\right)^2} \sqrt{\sum_{i=1}^{N} \left(Y_i - \overline{Y}\right)^2}}.$$

B Analyse par corrélation de consommation de courant

La CPA consiste à corréler des fuites mesurées lors de l'exécution d'un algorithme avec la prédiction de ces fuites pour toutes les valeurs d'une portion de secret. L'intérêt réside dans le fait que l'algorithme et/ou l'appareil manipulent potentiellement des "petits" morceaux du secret (*i.e.* énumérables). Ce sont sur ces morceaux que nous pourrons faire une hypothèse.

Soit un algorithme de chiffrement qui utilise une clé $K = (K_1, \ldots, K_n)$. Soit un message $M = (M_1, \ldots, M_n)$. Considérons que l'algorithme calcule à un moment de son exécution la valeur $T_i = f(K_i, M_i)$ pour tout i, $1 \leq i \leq n$. Alors nous pouvons effectuer une CPA de la façon suivante.

Au préalable: lancer N exécutions de l'algorithme avec N messages différents $M^{(1)}, \ldots, M^{(N)}$, et conserver les traces d'exécution. La valeur de N est un paramètre important qui conditionne beaucoup la réussite d'une CPA.

- Repérer les moments où sont calculés $T_1 = f(K_1, M_1)$. La consommation de courant à ce moment pour chaque exécution est rangée dans le vecteur $\mathcal{L} = (L^{(1)}, \ldots, L^{(N)})$.
- Pour chaque valeur possible $\widehat{k}$ pour K_1 faire
  - Calculer la prédiction de la fuite à l'aide d'une fonction de prédiction de fuite φ pour chaque message, $Y^{(i)} = (\varphi(f(\widehat{k}, M_0^{(i)}))$ pour tout i, $1 \leq i \leq N$. On obtient le vecteur $\mathcal{Y} = (Y^{(1)}, \ldots, Y^{(N)})$.
  - Calculer le coefficient de corrélation linéaire entre le vecteur de prédiction et le vecteur de fuite. $\rho_{\widehat{k}} = \rho(\mathcal{L}, \mathcal{Y})$.
- Choisir pour K_1 la valeur $\widehat{k}$ pour laquelle $\rho_{\widehat{k}}$ est la plus élevée.

Il suffit ensuite de réitérer l'opération pour les autres morceaux de clé $K_2, \ldots, K_n$ (avec les mêmes traces).

C Fichier de trace

Un fichier de trace contient la consommation du courant à chaque cycle d'exécution d'un algorithme. Il contient en outre l'entrée utilisée pour l'exécution et la sortie obtenue. L'exemple suivant est un fichier de trace avec une entrée et une sortie de 8 octets sur une exécution de 10 cycles.

```
# input  : 8 : 03 A6 7C 88 71 B2 91 FE
# output : 8 : 72 8F 5C 43 98 9C 11 A2
# cycles : 10
1 0.0000
2 1.0358
3 0.4827
4 0.0000
5 3.3841
```

```
6 0.0000
7 2.5113
8 2.4425
9 0.0000
10 3.8630
```

Pour manipuler ce genre de fichiers, nous avons à notre disposition une structure de donnée:

```
typedef struct
{
  uint8_t *input;
  unsigned int input_len;
  uint8_t *output;
  unsigned int output_len;
  double *trace;
  unsigned int ncy;
} trace_container;
```

Le tableau `input` (resp. `output`) contient les octets de l'entrée (resp. la sortie) de taille `input_len` (resp. `output_len`) de l'algorithme. Le tableau `trace` contient les valeurs correspondant à la consommation de courant à chacun des `ncy` cycles d'exécution. On pourra utiliser les fonctions suivantes (dans `libtraceio.a`) pour lire/écrire de tels fichiers de trace:

- `int read_trace (FILE *trace_file, trace_container *container)`
  Alloue le contenu de `container` en fonction du fichier `trace_file`.
- `int write_trace (FILE *trace_file, trace_container *container)`
  Écrit le contenu de `container` dans le fichier `trace_file`.
- `void free_trace (FILE *trace_file, trace_container *container)`
  Libère le contenu de `container` (préalablement alloué par `read_trace`).

D Fichier de corrélations

Un fichier de corrélation contient pour chaque hypothèse effectuée la valeur du coefficient de corrélation pour 1 ou plusieurs cycles d'exécution d'un algorithme. Voici un exemple avec 4 hypothèses de clé et 10 corrélations calculées.

```
# hypothesis: key = 00
0 0.000000
1 0.000000
2 -0.019034
3 -0.031763
4 0.000000
5 0.044142

# hypothesis: key = 01
0 0.000000
1 0.000000
2 0.016792
3 -0.023267
4 0.000000
5 0.034152

# hypothesis: key = 02
0 0.000000
1 0.000000
2 -0.036819
```

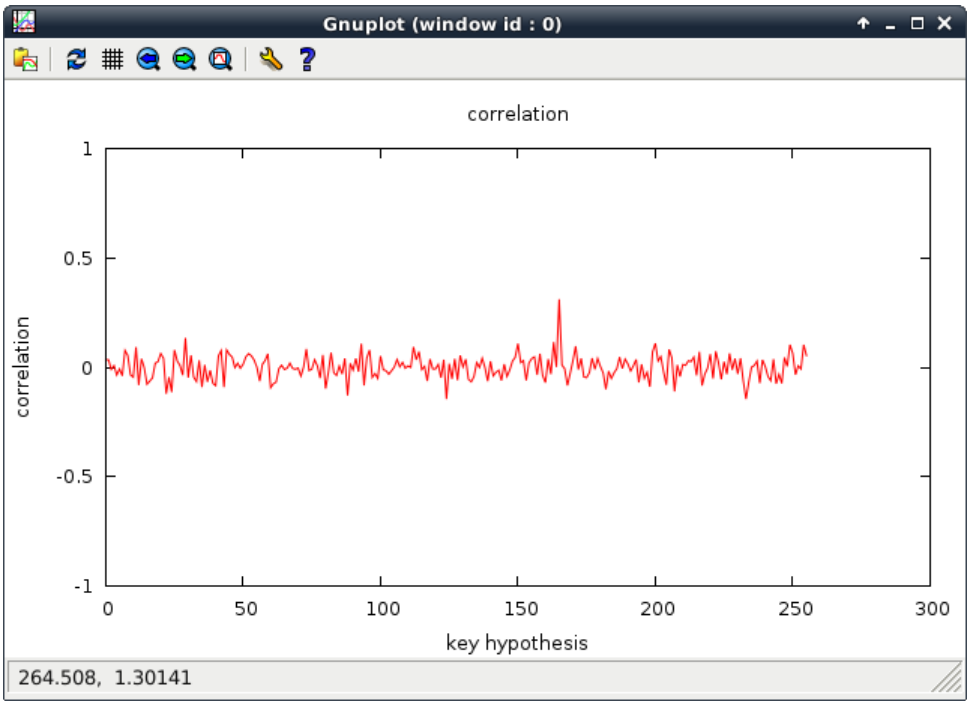
```
3 0.044402
4 0.000000
5 -0.012421

# hypothesis: key = 03
0 0.000000
1 0.000000
2 0.000448
3 -0.027299
4 0.000000
5 0.009460
```

Dans le cadre de ce TP, chaque hypothèse ne contiendra qu'une seule ligne (indexée par 0).

On pourra visualiser le contenu d'un fichier avec le script `CPA_poltkeys` (utilise `gnuplot`). Affiche la corrélation en fonction des hypothèse de clé pour le premier instant dans le fichier (ou un autre instant passé en second argument).

```
> CPA_plotkeys example0_cpa.dat
```



Enfin, on utilisera `CPA_guess_key` pour afficher l'hypothèse de clé qui possède la corrélation la plus élevée (qui correspond au pic observé à la figure précédente).

```
> CPA_guess_key example0_cpa.dat
A5
peak cycle : 0
```

Deux fichiers `example0_cpa.dat` et `example500_cpa.dat` contenant des corrélations pour 256 hypothèses de clé sur respectivement 1 instant et 500 instants) sont disponibles dans `ext/examples/`.

E Documentation pour les outils 8051

Le "pretty-printer" `pp8051` permet de formater un fichier assembleur 8051 en retirant les commentaires et en donnant à chaque instruction un label correspondant à son emplacement dans le code. Par exemple, un fichier assembleur `toto.a51` qui contient

```
        CSEG AT 0x0000
;; ceci est un commentaire
        mov A, R0
toto:
;; encore un commentaire
        orl A, 05Ah
        xrl A, #0x3B
        jmp toto

        END
```

L'appel à `pp8051` donnerait ceci:

```
> pp8051 toto.a51
x0000:  MOV   A, R0                             ;; l.3
x0001:  ORL   A, 0x5A                           ;; l.6
x0003:  XRL   A, #0x3B                          ;; l.7
x0005:  SJMP  0x0001                            ;; l.8
```

On y voit ainsi que l'instruction `xrl A, #0x3B` à la ligne 7 du fichier est à l'adresse `0x0003` du code. Son code machine est sur deux octets puisque l'instruction suivante est à l'adresse `0x0005`.

L'assembleur `as8051` prend en argument un fichier assembleur 8051 et produit sur la sortie standard le code en hexadécimal assemblé. On peut l'utiliser de la manière suivante

```
> as8051 file.a51 >file.hex
```

Le fichier de sortie `file.hex` contiendra le code (format HEX) assemblé correspondant au source `file.a51`. Un tel fichier peut être utilisé avec la machine virtuelle interactive `vm8051`:

```
> vm8051 file.hex
```

Voir la documentation pour connaître les différentes commandes disponibles dans `ext/doc`. La figure 1 montre la machine virtuelle interactive et décrit les lignes utiles pour le TP.

```
Regs                                        Ports
R0   : 0x00    00000000b                    P0   : 0xFF    11111111b
R1   : 0x00    00000000b                    P1   : 0xFF    11111111b
R2   : 0x00    00000000b                    P2   : 0xFF    11111111b
R3   : 0x00    00000000b                    P3   : 0xFF    11111111b
R4   : 0x00    00000000b
R5   : 0x00    00000000b
R6   : 0x00    00000000b
R7   : 0x00    00000000b

Sys
A    : 0x00    00000000b
B    : 0x00    00000000b
SP   : 0x07
DPTR : 0x0000
PC   : 0x0000  (LJMP  0x0030                    ; 020030)
PSW  : 0x00                RS = 0
states: 0

IE   : 0x00
IP   : 0x00
PCON : 0x00
TCON : 0x00

SCON : 0x00
SBUF : 0x00    00000000b

TMOD : 0x00
TL0  : 0x00    Timer0  (off):      0     13-bit timer
TH0  : 0x00
TL1  : 0x00    Timer1  (off):      0     13-bit timer
TH1  : 0x00


>
```

Figure 1: Interface de la machine virtuelle. On y voit l'état des différents registres de la machine. La ligne `PC` montre l'adresse courante du compteur ordinal ainsi que la prochaine instruction à s'exécuter. La ligne `states` donne le nombre de cycles écoulé depuis le début de l'exécution. La dernière ligne (prompt) reçoit les commandes de l'utilisateur.