

Sécurité et systèmes embarqués

Université Paris 8 – Vincennes à Saint-Denis
UFR MITSIC / M1 informatique

Séance 6 (TP) : Simple Power Analysis

N'oubliez pas :

- Les TPs doivent être rendus par courriel au plus tard le lendemain du jour où ils ont lieu avec “[sese]” suivi du numéro de la séance et de votre nom dans le sujet du mail, par exemple “[sese] TP6 Rauzy”.
- Quand un exercice demande des réponses qui ne sont pas du code, vous les mettez dans un fichier texte `reponses.txt` à rendre avec le code.
- Le TP doit être rendu dans une archive, par exemple un tar zippé obtenu avec la commande `tar czvf NOM.tgz NOM`, où `NOM` est le nom du répertoire dans lequel il y a votre code (idéalement, votre nom de famille et le numéro de la séance, par exemple “rauzy-tp6”).
- Si l’archive est lourde (> 1 Mo), merci d’utiliser <https://bigfiles.univ-paris8.fr/>.
- Les fichiers temporaires (si il y en a) doivent être supprimés avant de créer l’archive.
- Le code doit être proprement indenté et les variables, fonctions, constantes, etc. correctement nommées, en respectant des conventions cohérentes.
- Le code est de préférence en anglais, les commentaires (si besoin) en français ou anglais, en restant cohérent.
- **N’hésitez jamais à chercher de la documentation par vous-même sur le net!**

Dans ce TP :

- Écriture d’une exponentiation modulaire en assembleur.
- Analyse par canaux auxiliaires.

Exercice 0.

Récupérations des fichiers nécessaires.

1. Pensez à organiser correctement votre espace de travail, par exemple tout ce qui se passe dans ce TP pourrait être dans `~/sese/s6-tp/`.
2. Récupérez les fichiers nécessaires depuis la page web du cours, ou directement en ligne de commande avec `wget https://pablo.rauzy.name/teaching/sese/seance-6_tp.tgz`.
3. Une fois que vous avez extrait le dossier de l’archive (par exemple avec la commande `tar xzf seance-6_tp.tgz`), renommez le répertoire en votre nom (avec la commande `mv sese_seance-6_files votre-nom`). Si vous ne le faites pas tout de suite, pensez à le faire avant de rendre votre TP.

Exercice 1.

Découverte de l’environnement de travail.

1. Nous allons utiliser un CPU virtuel, simulé en Python. Celui-ci exécute des instructions données dans un assembleur très simple et nous permet de sonder sa consommation de courant simulée.

Ce CPU est simplifié par rapport à ce qui peut exister en vrai :

- il contient 32 registres (nommés `r0` à `r31`)
 - le registre `r31` est utilisé comme adresse de retour (cf les instructions `cal` et `ret`),
 - par convention le registre `r30` est utilisé comme pointeur de pile;
- il a accès à une RAM (1M de cases mémoire);
- il ne dispose d’aucun système de cache;
- l’accès à la mémoire ou aux registres est indifférenciée.

Chaque cycle d’exécution consiste en :

- récupérer l’instruction courante,
- décoder cette instruction,
- lire dans les registres et/ou dans la mémoire,
- faire le calcul correspondant à l’instruction,
- écrire dans les registres et/ou dans la mémoire.
- écrire dans un fichier l’activité électrique simulée de ce cycle.

Le jeu d’instructions de l’assembleur est assez réduit. Il y a seulement 27 instructions :

- `nop` : ne fait rien;
- `mov dst val` : copie la valeur de `val` dans `dst`;
- `not dst val` : écrit dans `dst` la valeur de la négation bit à bit de `val`;

- `and dst val1 val2` : écrit dans `dst` le et logique bit à bit de `val1` et `val2`;
- `orr dst val1 val2` : écrit dans `dst` le ou logique bit à bit de `val1` et `val2`;
- `xor dst val1 val2` : écrit dans `dst` le ou exclusif bit à bit de `val1` et `val2`;
- `lsl dst val1 val2` : écrit dans `dst` la valeur de `val1` décalée de `val2` bit vers la gauche;
- `lsr dst val1 val2` : écrit dans `dst` la valeur de `val1` décalée de `val2` bit vers la droite;
- `min dst val1 val2` : écrit dans `dst` le min de `val1` et `val2`;
- `max dst val1 val2` : écrit dans `dst` le max de `val1` et `val2`;
- `add dst val1 val2` : écrit dans `dst` la somme de `val1` et `val2`;
- `sub dst val1 val2` : écrit dans `dst` la différence de `val1` et `val2`;
- `mul dst val1 val2` : écrit dans `dst` le produit de `val1` et `val2`;
- `div dst val1 val2` : écrit dans `dst` le quotient entier de `val1` et `val2`;
- `mod dst val1 val2` : écrit dans `dst` le modulo de `val1` et `val2`;
- `ret` : saute à l'adresse contenu dans le registre `r31`;
- `cal addr` : met l'adresse de l'instruction suivante dans le registre `r31` puis saute à l'adresse `addr`;
- `cmp dst val1 val2` : écrit dans `dst` 1 si `val1 < val2`, -1 si `val1 > val2`, 0 sinon;
- `jmp addr` : saute à l'adresse `addr`;
- `beq addr val1 val2` : saute à l'adresse `addr` si `val1 = val2`;
- `bne addr val1 val2` : saute à l'adresse `addr` si `val1 ≠ val2`;
- `prn val` : affiche la valeur de `val` en base 10;
- `prx val` : affiche la valeur de `val` en base 16 sur 2 chiffres (un octet);
- `prX val` : affiche la valeur de `val` en base 16 sur sur 4 chiffres;
- `prc val` : affiche le caractère ASCII correspondant à la valeur de `val`;
- `prs addr val` : affiche la chaîne de caractère en RAM à l'adresse `addr` de longueur `val`;

Les valeurs (`val`, `val2`, `val2`) sont :

- soit une valeur immédiate, notée `#N` (`#13`, `#42`, `#51`, ...),
- soit un registre, noté `rN` (`r0`, `r1`, ..., `r31`),
- soit une case mémoire, notée `@N` (`@0`, `@1`, ...),
- soit une référence (adresse mémoire avec indirection), notée `!v` (`!r2`, `!@100`, ...),
- la dernière notation accepte également un décalage, donné après une virgule (`!r12,#-3`, `!r12,r2`).

Les destinations (`dst`) valides sont les valeurs modifiables, c'est-à-dire toutes sauf les valeurs immédiates.

Les adresses (`addr`) sont données soit sous forme de valeur, auquel cas elles correspondent à l'index de l'instruction dans le code, soit via un `label`. Les labels peuvent être défini n'importe où dans le code avec `label:` et auront pour valeur l'index de l'instruction qui les suit.

Vous pouvez tester un programme écrit dans `program.asm` en lançant le CPU virtuel dessus avec la commande `python3 bench.py program.asm conso.txt`. La sonde enregistrera l'activité électrique dans le fichier spécifié à la place de `conso.txt` (vous pouvez mettre `/dev/null` quand l'information ne vous intéresse pas).

L'exécution du programme commence à l'adresse du label `main`, qui est donc obligatoire.

→ Écrivez quelques programmes simples en assembleur pour vous familiariser avec l'environnement. Par exemple :

- Un programme qui affiche les valeurs de certains registres.
- Un programme qui change la valeur de certains registres.
- Un programme qui utilise les différentes instructions arithmétiques et logiques.
- Un "Hello, world!".
- Un programme qui fait un test conditionnel.
- Un programme qui fait une boucle.
- Un programme contient et appelle une petite fonction.

2. Jetez un œil à la l'activité électrique enregistrée par la sonde.
 - Arrivez-vous à y voir une corrélation avec vos programmes ?

Exercice 2.

Programmez un algorithme d'exponentiation modulaire.

1. Le but de cet exercice est de compléter une implémentation d'exponentiation modulaire de grand nombre, très utile en cryptographie. On a déjà vu en cours son utilisation dans RSA par exemple.

Pour rappel, le pseudocode de l'exponentiation modulaire est le suivant :

```

1 r := 1
2 b := b % m
3 tant que e != 0:
4     si e & 1 = 1 alors:
5         r := (r * b) % m
6     fin
7     b := (b * b) % m
8     e := e >> 1
9 fin

```

Une bibliothèque de grands nombres vous est fournie dans le fichier `bignum.asm`.

Un `bignum` est représenté en mémoire par un entier contenant sa taille et un tableau de ses chiffres en base 256. Par exemple le nombre `0x12ab34cd56ef78` serait représenté en mémoire de la façon suivante :

adresse dans la mémoire :	993	994	995	996	997	998	999	1000
valeur (en hexa) :	12	ab	34	cd	56	ef	78	7

Les fonctions de cette bibliothèque manipulent des pointeurs sur des `bignums`. L'adresse du `bignum` donné en exemple ci-dessus est 1000.

Par convention, la bibliothèque `bignum` utilise les registres `r25` à `r29` comme variables temporaires, et peut donc changer leur valeur sans les sauvegarder. Si vous avez besoin d'utiliser ces registres et de conserver leur valeur au travers d'un appel à une fonction de la bibliothèque `bignum`, vous devez vous charger de les sauvegarder en mémoire avant l'appel et de les rétablir après.

Le passage des arguments se fait par les registres `r20` à `r24` de la façon suivante :

- `bignum_print` : affiche la valeur du `bignum` sur lequel pointe `r20`.
- `bignum_init` : initialise (à zéro) un `bignum` de taille `r21` à l'adresse `r20`.
- `bignum_cleanup` : nettoie le `bignum` pointé par `r20` des 0 de poids forts (et réduit sa taille en conséquence).
- `bignum_zero` : met la taille du `bignum` pointé par `r20` à zéro si celui-ci vaut zéro, ne fait rien sinon.
- `bignum_clear` : met à zéro la valeur du `bignum` pointé par `r20` (sans réduire sa taille).
- `bignum_copy` : copie le `bignum` pointé par `r21` dans celui pointé par `r20`
- `bignum_cmp` : même chose que l'instruction `cmp` de l'assembleur mais avec `r20` comme résultat et la comparaison des `bignums` pointés par `r21` et `r24`.
- `bignum_sub` : soustrait au `bignum` pointé par `r20` celui pointé par `r24` (le résultat doit être positif).
- `bignum_add` : met à l'adresse pointée par `r20` le `bignum` résultat de l'addition de ceux pointés par `r21` et `r22` modulo celui pointé par `r24`.
- `bignum_rshift` : décale le `bignum` pointé par `r20` de `r21` bits vers la droite.
- `bignum_mul` : met à l'adresse pointée par `r20` le `bignum` résultat de la multiplication de ceux pointés par `r21` et `r22` modulo celui pointé par `r24`.

Avant de commencer à compléter l'implémentation de l'exponentiation modulaire, il est déjà important de vous familiariser avec la représentation des `bignums`.

→ Dans le fichier `main.asm`, initialisez la valeur de l'exposant (ligne 24) à `0x2ad50273`. Inspirez vous de ce qui est fait pour le message et le module.

2. On passe maintenant au fichier `modexp.asm`

L'exponentiation modulaire est ici programmée avec l'algorithme "square-and-multiply" (le même qu'on a vu en cours).

Ici, nous avons plusieurs particularités :

- Nous n'avons pas de fonction "carré" donc on multiplie le nombre à mettre au carré avec lui même.
 - La bibliothèque `bignum` ne gère pas les opérations en place (le résultat d'une opération doit être à un endroit différent en mémoire que ses opérandes). Pour pallier ce soucis, la bibliothèque `bignum` fournie `bignum_copy`, n'hésitez donc pas à utiliser une variable temporaire (une est déjà préparée dans le code fourni et le registre `r8` pointe dessus).
 - La bibliothèque `bignum` étant prévu pour la crypto, toutes les opérations qu'elles fournie (addition et multiplication) sont modulaires.
- Commencez par écrire la condition de la boucle après le label `_modexp_loop`.
3. Il faut sauter la multiplication si le bit de poids faible de l'exposant est à 0.
→ Faites un branchement conditionnel avant le label `_modexp_loop_mul`.
 4. Comme précisé plus haut, pour faire la multiplication correspondante à la 5ème ligne du pseudocode il faut utiliser une variable temporaire.
Ensuite, la multiplication peut être effectuée en utilisant la procédure `bignum_mul`.
→ Appelez les fonctions nécessaires pour faire la multiplication après le label `_modexp_loop_mul`.
 5. Il en va de même pour le carré.
→ Appelez les procédures nécessaires pour faire la multiplication après le label `_modexp_loop_sqr`.
 6. Il ne reste plus qu'à effectuer le décalage correspondant à la ligne 8 du pseudo code.
→ Faites le décalage après le label `_modexp_loop_shift`.
 7. Avec les valeurs présentent dans le fichier `main.asm` qui vous a été fourni (à savoir `0x53616c7574` pour la base, normalement `0x2ad50273` pour l'exposant, et `0x32ffc831c` pour le module), le résultat attendu est `0x2ac56b084`.
→ Testez votre code avec la commande `python3 bench.py main.asm /dev/null`.
 8. Si vous voulez essayer avec d'autres valeurs et vérifier vos résultats, il vous faudra implémenter l'algorithme square-and-multiply en Python :

```
1 def modexp (b, e, m):  
2     res = 1
```

```
3 b = b % m
4 while e != 0:
5     if e & 1 == 1:
6         res = (res * b) % m
7         b = (b * b) % m
8     e >>= 1
9 return res
```

Ensuite, les deux lignes de Python suivantes donneront le même résultat :

– `modexp(0x53616c7574, 0x2ad50273, 0x32ffc831c),`

– `(0x53616c7574 ** 0x2ad50273) % 0x32ffc831c.`

Cependant, implémenter l'algorithme square-and-multiply n'est pas inutile..

→ Comparez les deux calculs. Que se passe-t-il ? Pourquoi ?

Exercice 3.

Simple Power Analysis.

1. On rappelle que le secret est l'exposant, c'est donc entre cette valeur et l'activité électrique qu'on cherche des corrélations.

À l'aide de `gnuplot`, on peut visualiser graphiquement l'activité électrique enregistrée par la sonde. Le fichier `conso2png.gplot` contient un exemple de script avec lequel vous pouvez lancer `gnuplot` pour tracer un graphique dans une image PNG à partir d'un fichier de trace.

- (a) → Tracez quelques courbes de consommation pour des valeurs extrêmes de l'exposant (par exemple avec que des 0, que des 1, moitié-moitié, etc.).
 - (b) → Analysez ces traces (grossièrement, à l'œil). Que voyez-vous d'intéressant ?
2. Pour aider vos yeux non-entraînés à l'analyse de consommation, il y a une instruction supplémentaire dans le processeur qui permet d'instrumentaliser le code. En fait, il y a deux traces données par la sonde : l'une correspond à l'activité électrique à chaque instruction, et l'autre est par défaut tout le temps à zéro. L'instruction `dbg n` met la valeur de la seconde trace à `n` pour cette instruction.
 - (a) → Utilisez cette instruction dans le code de l'exponentiation modulaire pour vous aidez dans l'analyse.
 - (b) → Pour vous amuser, mettez une valeur d'exposant en choisissant indépendamment les octets au hasard (ou faites les mettre par quelqu'un-e d'autre et ne regardez pas le code), puis retrouvez la valeur directement depuis la courbe de consommation. Vérifiez ensuite que vous avez bien retrouvé la clef.