Pablo Rauzy <pr@up8.edu>
https://pablo.rauzy.name/teaching/is

Introduction à la sécurité

TP 4 : Simple Power Analysis

Dans ce TP :
— Attaque d'une signature RSA par analyse de consommation de courant.

Exercice 0.

Démarrage et découverte de l'environnement de travail SeseLab.

1. Récupérez les fichiers nécessaires à ce TP sur `https://pablo.rauzy.name/teaching/is/10-tp-simple-power-analysis.tgz`.
2. Installez la plateforme SeseLab depuis `https://pypi.org/project/seselab` avec la commande `pip3 install seselab`.
3. 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) ;
   — `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`, `val1`, `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 saufs 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 `seselab 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.

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

Exercice 1.

Simple Power Analysis.

1. Le programme `rsa.asm` qui vous est fourni fait un calcul de signature RSA : $s = m^d \bmod N$.
   → Exécutez le programme (sans enregistrer sa consommation de courant) et donnez les valeurs de m, N, et s.
2. Le programme `rsa.asm` fait appel à la fonction `modexp` définie dans la bibliothèque `modexp.asm` qui elle-même dépend de la bibliothèque `bignum.asm` qui est fournie avec SeseLab.
   → Que signifie "modexp" ? Donnez l'algorithme utilisé par cette fonction.
3. Notre but va être de retrouver l'exposant d de la clef secrète utilisée pour la signature.
   On suppose bien sûr qu'on a pas accès au contenu du fichier `rsa.asm` pour se placer dans un contexte "réaliste" d'attaque par analyse de consommation de courant.
   → Relancez l'exécution du programme `rsa.asm` cette fois-ci en enregistrant la consommation de courant dans un fichier `conso.txt`.
4. 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.
   → Analysez la trace de consommation de la signature RSA (grossièrement, à l'œil). Que voyez-vous d'intéressant ?
5. Pour aider nous aider dans notre attaque par analyse de consommation, il y a une instruction supplémentaire dans le processeur qui permet d'instrumentaliser le code, comme on le ferait avec une sonde et/ou un débuggeur lors d'une attaque en pratique.
   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 (fichier `modexp.asm`) pour vous aidez dans l'analyse.
   (b) → Pour vous amuser, changez la valeur de l'exposant utilisé dans `rsa.asm` (lignes 26–29) 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.