Timing attack et hyperthreading

Les processeurs modernes sont de plus en plus compliqués et difficiles à mettre en œuvre. Qu'en est-il de la sécurité des implémentations ? Peut-on exploiter les avancées technologiques pour obtenir de l'information sur les clefs de chiffrement ? Autant de questions auxquelles nous allons essayer de répondre.

1. Introduction

L'idée générale derrière ce type d'attaques déjà présenté dans [1], est qu'il est difficile d'implémenter un algorithme cryptographique de façon neutre. Par neutre, je veux dire qu'un attaquant observant notre implémentation (puissance, rayonnement...) n'obtient pas plus d'informations que s'il était devant une boîte noire. L'objectif de l'attaquant est de retrouver la clef secrète qui est utilisée par le système quand il chiffre des données confidentielles. Si par le biais d'un canal caché (side-channel attack), il est possible de retrouver la clef ou suffisamment de bits pour que la recherche exhaustive sur la partie de la clef manquante soit possible alors l'implémentation est compromise.
L'implémenteur et le cryptographe aimeraient avoir des critères de sélection pour évaluer la sécurité de leur implémentation. Pour cela, ils ont besoin d'évaluer la quantité d'information qui s'échappe lors de l'exécution. Il faut connaître le rapport signal/bruit, c'est-à-dire la quantité de données extractibles par l'attaquant. Pour illustrer les difficultés qu'entraînent une nouvelle technologie comme l'hyperthreading, je vais expliquer de nouvelles attaques contre le système RSA telles qu'elles ont été présentées dans [3].

2. Optimisation de l'exponentiation modulaire

L'implémentation de RSA a connu beaucoup de changements depuis sa création. L'opération centrale du système est le calcul de **C = M^e mod n** où **n** est le module public et e l'exposant secret. Cette opération est appelée " exponentiation modulaire ". Elle fait intervenir des multiplications modulaires et des élévations au carré (square and multiply) sur des entiers de grandes tailles (1024 bits). Pour effectuer cette opération, on peut prendre les bits de l'exposant les uns après les autres. C'est la méthode binaire que l'on a déjà vu dans [1]. Il existe d'autres variantes de square and multiply qui sont plus rapides. La méthode que l'on va cryptanalyser s'appelle " sliding window ". Elle consiste à évaluer l'exposant en **p** fenêtres **Fi** de taille variable **L(Fi)**. Sans entrer dans les détails (on peut trouver plus d'informations dans [4]), cette méthode fait appel à une table de précalcul. C'est précisément le fait d'avoir une table de précalcul qui va rendre possible la side channel attack. Tout ceci sera plus clair avec un exemple.
Prenons une taille de fenêtre **d = 3**, on va décomposer **e = 3665** soit **111001010001** en base 2. On obtient la décomposition suivante :
e = 111 00 101 0 001
La phase de précalcul consiste à stocker tous les **M^w** où **w** représente toutes les fenêtres non nulles possibles, c'est-à-dire
w = 3, 5, 7, . . . , 2^d – 1.

Methode binaire	Methode des fenêtres
if $e_{k-1} = 1$ **then** $C := M$ **else** $C := 1$ **for** $i = k - 2$ **downto** 0 $C := C.C \mod n$ **if** $e_i = 1$ **then** $C := C.M \mod n$ **return** C	$C := M^{F_{r-1}} \mod n$ **for** $i = p$-2 **downto** 0 $C := C^{2^{L(F_i)}} \mod n$ (1) **if** $F_i \neq 0$ **then** $C := C.M^{F_i} \mod n$ (2) **return** C

Figure 1 : Algorithmes d'exponentiation

bits	w	M^w
011	3	$M.M^2 = M^3$
101	5	$M^3.M^2 = M^5$
111	7	$M^5.M^2 = M^7$

Figure 2 : Table de précalcul

i	F_i	$L(F_i)$	(1)	(2)
3	00	2	$(M^7)^4 = M^{28}$	M^{28}
2	101	3	$(M^{28})^8 = M^{224}$	$M^{224}.M^5 = M^{229}$
1	0	1	$(M^{229})^2 = M^{458}$	M^{458}
0	001	3	$(M^{458})^8 = M^{3664}$	$M^{3664}.M = M^{3665}$

Figure 3 : Trace d'exécution

Enfin pour terminer sur la présentation des algorithmes d'exponentiation modulaire voici la comparaison en moyenne entre la méthode binaire et la méthode sliding window pour 512 bits d'exposant. Pour la méthode binaire on obtient 766 multiplications. Pour l'algorithme sliding windows avec une taille de fenêtre **d = 5**, on obtient seulement 607 multiplications. Ce gain est suffisamment important pour qu'une bibliothèque comme **Openssl** utilise cette méthode.

3. Accès mémoire

Maintenant nous allons traquer ce qui peut faire varier de façon significative le temps d'exécution d'un processus. Je n'ai bien sûr pas la place ici de faire un cours d'architecture des processeurs, néanmoins toute personne désireuse d'acquérir plus de connaissances dans ce domaine peut consulter la bible [2]. Je me concentrerai seulement sur les accès mémoire. La latence d'un accès mémoire dépend de sa position dans la hiérarchie mémoire. Si une donnée est dans un niveau de cache, alors le temps d'accès est plus rapide que si elle avait été en RAM. On parle de cache hit si la donnée est dans le cache et de cache miss dans le cas contraire.

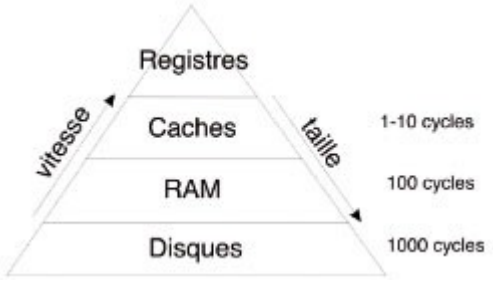

Figure 4 : Hiérarchie mémoire

La description complète d'une mémoire cache est trop complexe pour être abordée ici (encore une fois, pour plus de détails consultez [2]). On peut trouver en effet plusieurs niveaux de cache et des

caches dédiés aux instructions ou aux données. La taille du cache est relativement petite par rapport à la RAM. La taille est donc un paramètre important car certaines données ne pourront pas tenir dans le cache. La taille des éléments qui rentrent ou sortent du cache est important. Il ne faut s'imaginer que l'on charge seulement des entiers de 32 bits quand on en aura besoin. On va charger des blocs mémoire de taille fixée. Il est aussi important de savoir comment les blocs de données sont rangés et remplacés. Tous ces paramètres et bien d'autres encore influencent le temps d'accès aux données. Il est donc difficile quand on fait des accès aléatoires à une grande table de savoir comment va se comporter le programme.

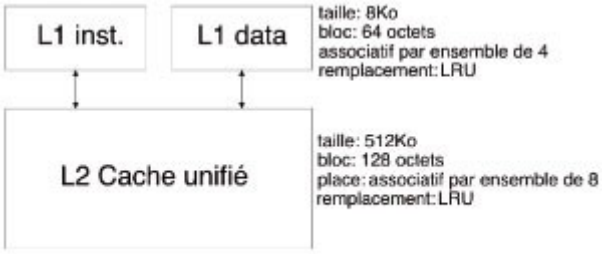

Figure 5 : Cache du Pentium 4

Dans la figure 5, on trouve les caractéristiques du cache de certains Pentium 4. Les caches du Pentium sont associatifs par ensemble. Cela veut dire que l'on peut placer chaque bloc dans un nombre m de positions (le cache est dit " m-associatif "). Il faut encore savoir dans quel bloc on va faire le remplacement. Pour cela on utilise un algorithme qui s'appelle Least Recently Use (LRU) qui effectue les remplacements dans les blocs les moins récemment utilisés.
Dans ce qui suit, je représenterai un cache avec seulement 2 sous-ensembles de 4 lignes et LRU. C'est largement suffisant pour comprendre le fonctionnement d'un cache. J'ai représenté dans la figure 6 des séries d'accès à 2 tables T et Q. L'accès à Q est un accès conditionnel qui ne se fait donc pas en permanence. Au démarrage le cache est vide, on le charge avec la table T : tous les accès mémoire sont des cache miss. Il faut aller les chercher en RAM. Quand on cherche à accéder à Q le bloc LRU (qui appartient donc à T) doit sortir du cache et la ligne de Q adéquate doit y être insérée. On cherche ensuite à réaccéder à T. Problème : certains blocs n'y sont plus. Pire, en les ramenant dans le cache on va faire sortir des blocs auxquels on doit accéder. On remarque donc que quand on fait un accès à Q, le nombre de cache miss est plus élevé. Pas seulement le nombre de cache miss, mais également le temps d'exécution et la consommation d'énergie. C'est ce que j'ai représenté dans la figure 6.

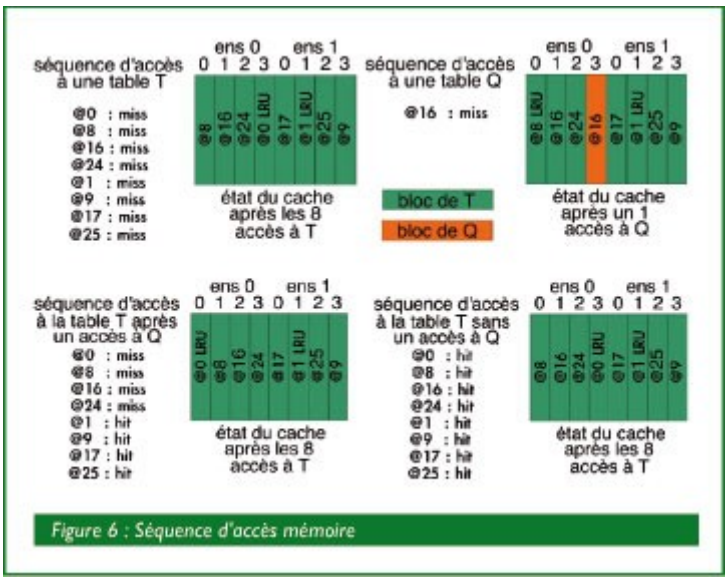

Figure 6 : Séquence d'accès mémoire

Si on revient à l'exemple du RSA avec sliding window, quand la fenêtre de l'exposant n'est pas nulle, on réalise une multiplication modulaire, une élévation à une puissance de 2 et un accès en table. Dans le cas contraire, on réalise seulement les élévations à des puissances de 2. Ainsi le nombre de cache miss est potentiellement plus important lors des itérations avec multiplication modulaire. Idéalement on voudrait que chaque accès à la table soit un cache miss afin d'être sûr d'observer un temps de latence plus long lors de l'itération. Le problème est que juste avant de commencer l'exponentiation, on calcule la table. Ceci signifie que la table est en grande partie dans le cache. Ceci n'arrange pas nos affaires, il faudrait pouvoir enlever la table du cache (flush table). Ensuite à chaque itération de l'algorithme avec une fenêtre non nulle, il faudrait encore nettoyer le cache pour être sûr que des bouts de la table ne sont pas dans le cache. Ainsi, en mesurant le nombre de cache miss dans l'exponentiation, on va pouvoir distinguer les itérations avec ou sans multiplications. On trouve directement le bit le moins significatif de chaque fenêtre.

Le Sliding Windows

Calcul de la table
Flush table
$C := M^{F_{p-1}} \mod n$
for $i = p\text{-}2$ **downto** 0
  $C := C^{2^{L(F_i)}} \mod n$ (1)
  if $F_i \neq 0$ **then** $C := C.M^{F_i} \mod n$ (2)
  mesure du ratio de miss
  Flush table
return C

Figure 7 : placement de la sonde

Cette attaque semble impossible en pratique, car il n'existe pas de moyen de mesurer le nombre de miss à chaque itération de l'algorithme. Il faudrait que le processeur soit capable d'exécuter 2 processus en même temps : un qui réalise l'exponentiation, l'autre qui serait un processus espion. Justement la solution arrive...

4. Hyperthreading

L'hyperthreading est une technologie développée par Intel pour améliorer les performances de ses processeurs. Sur un processeur monothread, la ressource de calcul est allouée par l'ordonnanceur de façon plus ou moins séquentielle et pour une durée déterminée t. Un unique processus, le processus courant A, fait alors une utilisation plus ou moins bonne des ressources du processeur. Si on rajoute un deuxième processeur, on peut exécuter 2 threads en simultané mais le potentiel des 2 processeurs reste toujours sous-exploité.

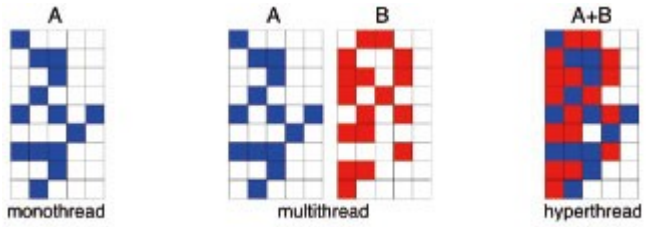

Figure 8 : Intérêt de l'hyperthreading (chaque case représente une ressource du processeur)

Avec la technologie hyperthreading, les ressources d'un unique processeur sont partagées par 2 threads qui s'exécutent simultanément. Avec un processeur hyperthreadé, on a virtuellement 2 processeurs dans 1. Dans la figure 8, j'ai représenté le cas idéal où les deux threads ne cherchent pas à accéder systématiquement aux mêmes ressources. Ainsi avec l'hyperthreading, on oscille entre 1 et 2 processeurs en 1 seul car les exigences des 2 processus sont souvent les mêmes. Le fait que les ressources du processeur soient partagées ne signifie pas que le deuxième thread peut accéder aux données du premier, du moins s'il n'y a pas de segment de mémoire partagée. Les espaces d'adressage des deux processus restent bien distincts, mais en revanche il n'y a qu'un seul cache. Ainsi, quand les données du processus A arrivent dans le cache, elles peuvent très bien faire sortir des données du processus B. Et voilà notre canal caché pour mesurer le nombre de cache miss !
Le processus espion est responsable de la table T qui recouvre tout le cache. Le processus effectue des opérations sur un sous-ensemble du cache et mesure les temps d'accès à chaque bloc. Ensuite, il passe au sous-ensemble suivant et ainsi de suite. En accédant au sous-ensemble, il éjecte aussi les blocs de l'exponantiation. On a implémenté la sonde qui mesure le temps d'accès et le nettoyeur de cache.
Les accès à la table Q représentent le processus d'exponentiation. Quand on fait une multiplication modulaire, le nombre d'accès est plus grand, ce qui implique que la mesure du processus espion est plus fortement perturbé. On détecte ainsi les cycles avec ou sans multiplication. Bien sûr, je vous passe tous les détails sordides d'implémentation (problèmes de TLB, **prefetch**...) et le fait qu'il y ait un bruit sur les mesures à cause des interruptions et des accès en mémoire pendant les multiplications modulaires. C'est exactement ce qui a été proposé dans [3].
On peut faire aussi beaucoup plus simple en créant simplement un processus espion qui récupère le nombre de cache miss dans les registres de monitoring du Pentium 4 et qui ensuite nettoie le cache. Ces registres ne sont pas accessibles avec de simples droits utilisateur, il faut les privilèges de root. Les instructions qui nous permettent d'accéder à l'information utile sont RDMSR (Read Model Specific Counter), WRMSR (Write Model Specific Counter) et RDPMC (Read Performance-Monitoring Counter).

5. Conclusion

Il est assez remarquable de constater que les mécanismes d'optimisation des processeurs peuvent remettre en cause la sécurité des systèmes de chiffrement. Sans même remettre en cause la sécurité mathématique d'un algorithme, on est capable d'exploiter les faiblesses de l'implémentation même dans des cas complexes comme le nôtre. Ce type de faiblesse est généralement trouvé lorsque l'on cherche à profiler son code pour améliorer les performances. On voit aussi que l'optimisation algorithmique d'un programme en utilisant des tables a toujours un coût qui est souvent négligé ou oublié, celui de la surface mémoire. Je laisse ici de nombreuses questions en suspens. Principalement, vous vous demandez comment éliminer ce type de faille. Est-ce la tâche de l'OS, du fabricant de processeur ou du programmeur ? La réponse dans un prochain numéro.