École Normale Supérieure
Département d'Informatique

Rapport de stage de L3

Année 2016-2017

Détection automatique des adresses exploitables d'un algorithme de cryptographie par timing du cache processeur

Théophile Wallez

Dirigé par Sylvain Guilley et Adrien Facon

Remerciements

Je remercie Sylvain Guilley et Adrien Facon pour m'avoir encadré pendant ce stage, David Naccache grâce à qui j'ai obtenu ce stage et Sébastien Carré pour ses réponses à mes nombreuses questions sur le cache.

Je tiens aussi à remercier Nicolas, Charles, Michael, Florent, Sofiane et Tui pour les nombreuses discussions sur divers domaines de la sécurité, ainsi que la bonne ambiance du bureau.

Table des matières

1 Introduction

J'ai effectué mon stage dans les laboratoires de Secure-IC, une entreprise qui développe des solutions de sécurité pour les systèmes embarqués, ce qui m'a permis de découvrir ce domaine. Le but du stage était d'utiliser du machine learning pour retrouver la clé d'un algorithme de cryptographie, en observant le cache du processeur.

Durant le premier mois, j'ai commencé par faire du traitement de données pour trouver quelles fonctions d'une librairie de cryptographie fuitent de l'information par le cache. Pendant le deuxième mois j'ai utilisé des techniques de machine learning pour essayer de trouver le premier bit de la clé.

2 Préliminaires sur le fonctionnement du cache et du système d'exploitation

2.1 Le cache et les timing-attacks

Pour des raisons de construction, l'accès à la mémoire vive (RAM) est lente (environ 300 cycles processeur). Les processeurs modernes disposent donc d'une mémoire « cache », qui est beaucoup plus rapide (de 2 à 30 cycles processeur), mais qui est aussi plus chère à produire et plus consommatrice d'énergie.

Ainsi, quand le processeur doit accéder à la RAM, il transfère les données dans le cache, et il pourra ultérieurement accéder aux données directement à partir du cache et donc plus rapidement. Les données sont cachées par paquets de 64 octets, que l'on appelle « ligne de cache » (cacheline).

Sur les processeurs Intel, on peut admettre que le cache est partagé entre les différents cœurs (pour être précis, le cache L3 est partagé par les différents cœurs et les adresses mises dans les caches L1 et L2 sont aussi dans le cache L3).

Si l'on regarde le processeur de manière abstraite, la présence du cache ne change rien à sa sémantique. On peut néanmoins observer les effets du cache en mesurant le temps que met le processeur à accéder à une adresse, en utilisant l'instruction `rdtsc` qui mesure le nombre de cycles effectués par le processeur. Cette instruction permet d'avoir une mesure de temps plus précise que la nanoseconde.

Il est facile de manipuler le cache :

— pour mettre une adresse dans le cache, il suffit d'y accéder,
— pour enlever une adresse du cache, on dispose de l'instruction `clflush` qui fait cela,
— pour savoir si une adresse est dans le cache, on peut mesurer le temps que l'on met pour y accéder.

À partir de là, on peut fabriquer un oracle qui permet de savoir si une adresse a été accédée entre un temps t_1 et t_2.

2.2 L'attaque Flush+Reload

Au temps t_1, on utilise `clflush` pour enlever l'adresse du cache.

Au temps t_2, on mesure le temps d'accès à cette adresse.

2.3 L'attaque Flush+Flush

Le temps d'exécution de `clflush` est plus long si l'adresse est dans le cache, comme montré dans la figure 1. On peut donc l'utiliser pour mesurer si une adresse est dans le cache au temps t_2, et en même temps l'enlever du cache au prochain temps t_1. Cela permet donc de pipeliner l'attaque.

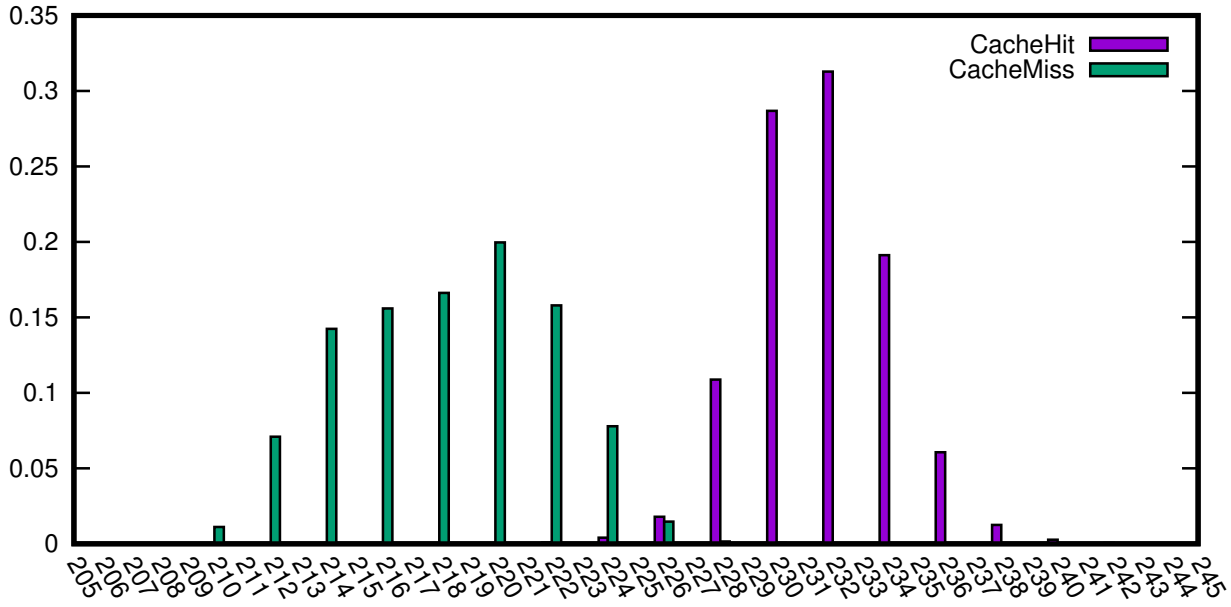

FIGURE 1 – Distribution du nombre de cycles nécessaires pour effectuer l'instruction `clflush` dans le cas où l'adresse est dans le cache (violet) et dans le cas où l'adresse n'est pas dans le cache (vert). On constate que le nombre de cycles est toujours pair, c'est parce que sur mon ordinateur, `rdtsc` renvoie toujours une valeur paire

Cette attaque a l'avantage d'être plus rapide, et moins détectable (via les compteurs de performance) [6].

2.4 Les librairies partagées

Certaines librairies partagées, comme OpenSSL, sont utilisées par beaucoup de programmes en même temps.

Plutôt que de charger le code de OpenSSL en mémoire plusieurs fois, le système d'exploitation charge le code une seule fois en mémoire, et partage cette mémoire entre tous les programmes utilisant OpenSSL.

Cela fait que la mise en cache d'OpenSSL est partagée entre les différents programmes.

2.5 La conséquence de ces faits

Considérons l'algorithme 1 d'exponentiation rapide à temps constant.

Le temps d'exécution de cet algorithme ne dépend pas des d_i. On peut néanmoins l'attaquer via le cache : quand le processeur exécute le code à l'intérieur du if ou du else, il doit d'abord aller

Algorithme 1 : Montgomery powering ladder [8]

Data : $d = d_n \dots d_{1[2]} \in \mathbb{N}$, $x \in G$
Result : $R_0 = x^d$
$R_0 \leftarrow 0$
$R_1 \leftarrow x$
/* Invariant 1 : R_1 = mult(x, R_0) */
/* Invariant 2 : $R_0 = x^{d_n \dots d_{i[2]}}$ at the end of loop i */
for $i \leftarrow n$ **to** 1 **do**
  if $d_i = 0$ **then**
    $R_1 \leftarrow$ mult(R_0, R_1)
    $R_0 \leftarrow$ square(R_0)
  else
    $R_0 \leftarrow$ mult(R_0, R_1)
    $R_1 \leftarrow$ square(R_1)
  end
end

récupérer le code en mémoire : avec l'attaque décrite plus haut sur le cache, on peut savoir dans quelle branche on passe à chaque itération, et donc découvrir les d_i.

OpenSSL a été exploité de cette manière avec l'attaque flush+reload [11].

2.6 Difficultés

Le comportement d'un processeur moderne est plus complexe que ce que l'on a décrit ci-dessus.

Les processeurs modernes changent leur fréquence dynamiquement afin d'économiser de l'énergie et tourner à une fréquence plus basse lorsqu'il y a peu de calculs à faire (frequency scaling).

Les processeurs modernes essayent de prédire à quelles adresses on va accéder afin de les mettre dans le cache avant que l'on y accède réellement (prefetching et branch-prediction).

Les processeurs modernes réordonnent les instructions qu'ils exécutent sans changer la sémantique. Ainsi, quand on fait un accès mémoire entre deux `rdtsc` pour savoir si cette adresse est dans le cache, il n'est pas impossible que le processeur fasse l'accès mémoire puis les deux `rdtsc`.

3 Exploitation du cache

Nous allons étudier ECDSA, un algorithme de signature utilisant des courbes elliptiques (voir Annexe B). On cherche à retrouver la clé privée utilisé pour la signature. Durant la signature, un entier aléatoire k, appelé « nonce » est généré cryptographiquement.

Si l'on fait deux signatures avec le même k, on peut obtenir la clé privée facilement, comme cela a été fait avec la PlayStation 3 [5].

De plus, apprendre quelques bits sur k pour une centaine de signatures suffit pour retrouver la clé [9].

Cet entier est utilisé comme exposant, et comme la courbe elliptique est un groupe abélien, on note les opérations additivement. Pour cette raison, on dit aussi « multiplication scalaire » à la

place de « exponentiation » et on ne parle pas de « mise au carré » ou de « multiplication » mais de « doublement » et de « addition ».

Les opérations d'addition et de doublement d'un point sur une courbe elliptique utilisent des formules différentes, qui sont dans les fonctions d'OpenSSL `EC_POINT_add` et `EC_POINT_dbl`.

Nous nous intéressons donc ici à retrouver quelques bits de k lorsque l'algorithme de multiplication scalaire est la méthode wNAF (voir Annexe C). Une telle attaque a déjà été réalisée [2].

Nous allons présenter comment on détermine automatiquement les adresses à observer pour effectuer cette tâche.

3.1 Analyse d'une librairie partagée

Pour chaque ligne de cache de OpenSSL, on lance l'algorithme qui nous intéresse sur un cœur (ici, la multiplication scalaire d'un point sur une courbe elliptique dont on cherche à déterminer le paramètre), et sur un autre cœur on analyse l'activité sur cette ligne de cache en itérant l'attaque Flush+Flush. Dans la suite, on appelera « trace » la suite des résultats de l'attaque Flush+Flush sur une adresse donnée durant l'exécution de l'algorithme.

Cela donne des graphiques comme dans les figures 2 et 3.

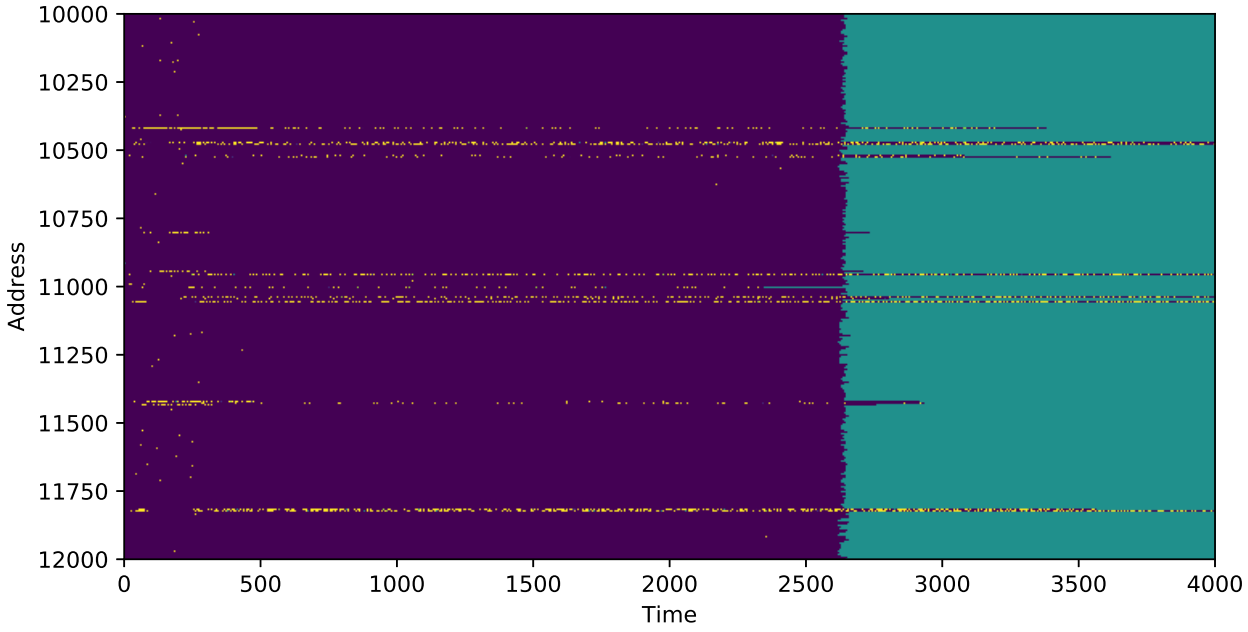

FIGURE 2 – Activité mémoire sur les adresses correspondant aux opérations sur les grands nombres. En jaune il y a de l'activité mémoire, en violet il n'y a pas d'activité mémoire, et le vert signifie que l'algorithme a terminé

On remarque que la longueur des lignes change selon la ligne de cache analysée. C'est parce que comme la ligne de cache en question est beaucoup utilisée, on passe notre temps à l'enlever du cache avec l'attaque Flush+Flush, donc le code que l'on analyse va faire beaucoup plus de cache-miss et va donc prendre plus de temps. Ce graphique permet donc déjà de voir quelles adresses sont intéressantes pour faire une attaque par dégradation de performances.

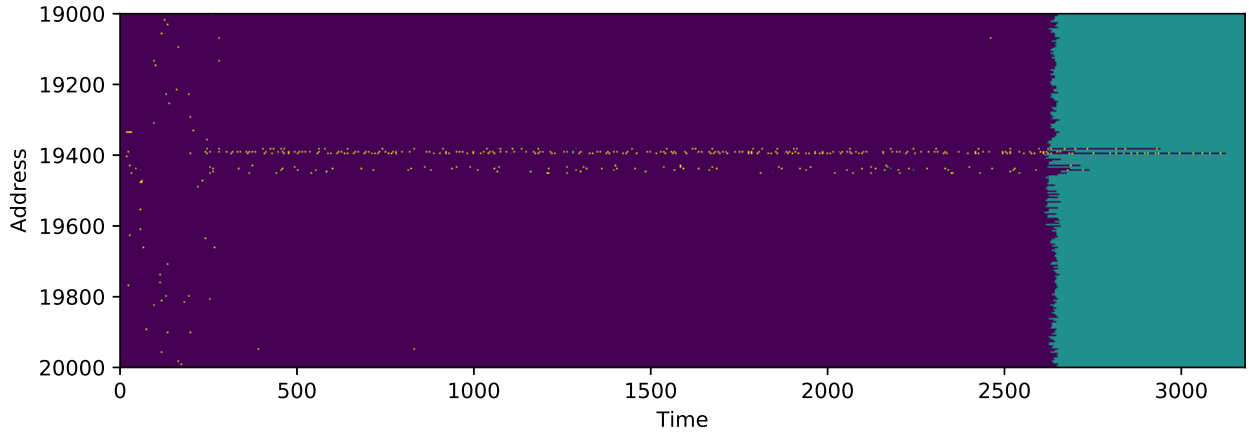

FIGURE 3 – Activité mémoire sur les adresses correspondant aux opérations sur les courbes elliptiques. En haut, `EC_POINT_dbl`, en bas, `EC_POINT_add`

3.2 Détection des fuites d'information

On veut savoir si les accès à une certaine ligne de cache dépendent de la clé.

3.2.1 Première approche

Une première manière d'aborder le problème est de se dire qu'une ligne de cache fuite de l'information si, quand on utilise deux clés différentes, l'activité sur cette ligne de cache change. On peut généraliser cette procédure à un ensemble de ligne de caches que l'on observe de manière synchronisée. On va ici faire cette procédure sur des paires d'adresses.

On utilise pour cela l'algorithme DTW (voir Annexe A), qui permet d'avoir une information sur la distance entre deux séries qui ne sont pas synchronisées. En effet, on ne peut pas comparer les séries point à point : d'une part, les séries que l'on mesure n'ont pas la même longueur, d'autre part ce qu'il se passe sur l'ordinateur pendant que l'on fait la mesure peut provoquer des petites translations ou dilatations dans la trace.

On mesure donc les traces pour deux clés différentes, et on regarde si ces deux traces sont différentes ou pas. Pour être plus précis, on prends les 200 lignes de cache les plus actives, et on fait cette mesure pour chaque paire d'adresse. On mesure des paires dans le but de retrouver un couple `EC_POINT_add` et `EC_POINT_dbl`.

Ce n'est finalement pas fructueux, car certaines adresses sont très actives et produisent des traces qui sont intrinsèquement difficiles à comparer, et cela donne des distances très hautes qui cachent les adresses qui fuitent de l'information exploitable. Sur ces adresses intrinsèquement difficiles à comparer, on constate qu'en appliquant la procédure avec deux fois la même clé, on obtient des grandes distances. Nous avons essayé de prendre ce phénomène en compte, sans succès.

3.2.2 Seconde approche

Une seconde manière d'aborder le problème est de se dire qu'une ligne de cache fuite de l'information, si à partir d'une trace, on peut retrouver la clé associée. Cette dernière tâche étant difficile, on va se limiter à reconnaître une clé parmi n clés différentes (ici, on a pris $n = 10$).

On a pour ce faire, utilisé l'algorithme des k plus proches voisins sur les traces générées par une adresse avec n clés différentes.

Il faut pour cela choisir une distance. On a commencé avec DTW, ce qui a permis de trouver automatiquement `EC_POINT_add`. Par contre, cela n'a permis de trouver `EC_POINT_dbl`.

Pour trouver une autre distance qui prends en compte les translations et dilatations, on a utilisé les basses fréquences de la transformée de Fourier. Cela a permis de trouver automatiquement `EC_POINT_dbl` et `EC_POINT_add`.

Pourquoi DTW n'a pas permis de trouver `EC_POINT_dbl` ?

C'est parce que DTW a un paramètre qui limite les dilatations autorisées, et `EC_POINT_dbl` est appelé très fréquemment, avec de temps en temps un appel à `EC_POINT_add` qui s'intercale entre deux appels à `EC_POINT_dbl`, mais qui provoque une dilatation assez faible que DTW corrige. Donc pour DTW, toutes les traces de `EC_POINT_dbl` se ressemblent.

Au contraire, la fonction `EC_POINT_add` est appelée assez peu souvent et le mécanisme limitant les dilatations ne permet pas de tous les synchroniser, ce qui fait que deux traces générées avec des clés différentes seront assez différentes pour DTW.

On voit donc que DTW corrige les dilatations d'amplitude inférieures à un seuil sans les quantifier, alors que la transformée de Fourier ne corrige pas les dilatation mais les quantifie. La transformée de Fourier permet donc de retrouver `EC_POINT_add`.

4 Machine learning pour retrouver la clé

On cherche à obtenir le bit de poids faible de la clé. Pour cela, on utilise des algorithmes d'apprentissage sur une trace où l'on observe de manière synchronisée `EC_POINT_add` et `EC_POINT_dbl`.

Les traces que l'on observe ne sont pas parfaites : entre deux traces censées être identiques, il peut y avoir des translations ou des dilatations. Cela fait que les algorithmes de machine learning classique vont avoir du mal à les traiter.

Collecter des données pendant un nuit nous a permis d'obtenir huit millions de traces.

4.1 Tentative de resynchronisation

Pour faciliter l'apprentissage, on aimerait bien que les traces soient synchronisées, afin que l'on puisse les comparer point à point et avoir de meilleures performances sur l'apprentissage.

On s'est dit qu'il était possible de trouver une adresse utilisée régulièrement qui peut servir à synchroniser.

C'est une adresse que l'on mesure de manière synchronisée avec l'adresse qui nous intéresse vraiment, pour obtenir le couple de traces $(T^{(1)}_{sync}, T^{(1)}_{info})$ et $(T^{(2)}_{sync}, T^{(2)}_{info})$.

Si l'on a généré les traces avec la même clé, on veut que

$$\forall s \in \operatorname*{argmin}_{s\in\mathbb{S}} \mathrm{d}_s(T^{(1)}_{sync}, T^{(2)}_{sync}), \mathrm{d}_s(T^{(1)}_{info}, T^{(2)}_{info}) \approx \mathrm{DTW}(T^{(1)}_{info}, T^{(2)}_{info})$$

(voir l'Annexe A pour les notations).

Pour vérifier à quel point cette approximation est correcte, on peut utiliser un écart relatif ou absolu.

On mesure donc cet écart pour toutes les adresses $info$, et on fait la moyenne : cela donne un score pour l'adresse *sync*.

Dans la moyenne, on pondère l'écart par $\frac{1}{1+\mathrm{DTW}(T^{(1)}_{info}, T^{(2)}_{info})}$ car si la DTW est grande, cela signifie que l'adresse est intrinsèquement difficile à mesurer, et donc l'écart est peu pertinent.

En pratique, les meilleures adresses en erreur absolue et en erreur relative ont des scores beaucoup trop hauts pour que cette méthode soit viable. Nous ne trouvons donc pas d'adresse de synchronisation.

C'est parce que les traces sont composées essentiellement de 0 et de 1, avec très peu de valeur entre les deux, et pour synchroniser un groupe de 1 avec un groupe de 1, il y a plein de façon de faire, donc on n'a pas unicité de la synchronisation qui minimise dans la DTW.

On a essayé de privilégier les déplacements en diagonale (faire $(+1, +1)$ dans la synchronisation) mais cela n'a rien changé.

Finalement, on choisit une longueur arbitraire, et on ajoute des 0 si la trace est trop petite, et on la tronque si elle est trop grande.

4.2 Différents résultats d'apprentissage

Nous avons utilisé Keras [4] pour entraîner des réseaux de neurones et scikit-learn [10] pour les autres algorithmes d'apprentissage.

On fait l'apprentissage sur des traces de `EC_POINT_add` et `EC_POINT_dbl` mesurées en parallèle, en sélectionnant l'adresse parmi celles identifiées avec la méthode des k plus proches voisins avec la transformée de Fourier.

On mesure le score d'un modèle en regardant le nombre de réponses justes sur un ensemble de traces qu'il n'a jamais vues.

4.2.1 Modèle linéaire

Il est généralement recommandé de commencer par des modèles simples, et les améliorer petit à petit, nous avons donc commencé par le plus simple des modèles : un modèle linéaire.

Ce modèle a permis de trouver le premier bit avec 60% de réponses justes.

4.2.2 Réseaux de neurones feed-forward

On a ensuite évalué l'utilisation un réseau de neurones avec une couche intermédiaire. Nous avons fait varier le nombre de neurones dans cette couche, mais cela n'a pas permis d'obtenir de score significativement meilleur que 50%.

On peut se demander pourquoi, puisque cette approche est strictement plus puissante que le modèle linéaire (dans le sens où si l'on dispose des poids d'un modèle linéaire, on peut créer des poids sur le réseau de neurones avec une couche intermédiaire qui lui donne le même comportement en sortie que le modèle linéaire). C'est parce que cette approche fait qu'il y a beaucoup trop de paramètres à entraîner et l'optimiseur n'arrive pas à converger vers une bonne solution en un temps raisonnable.

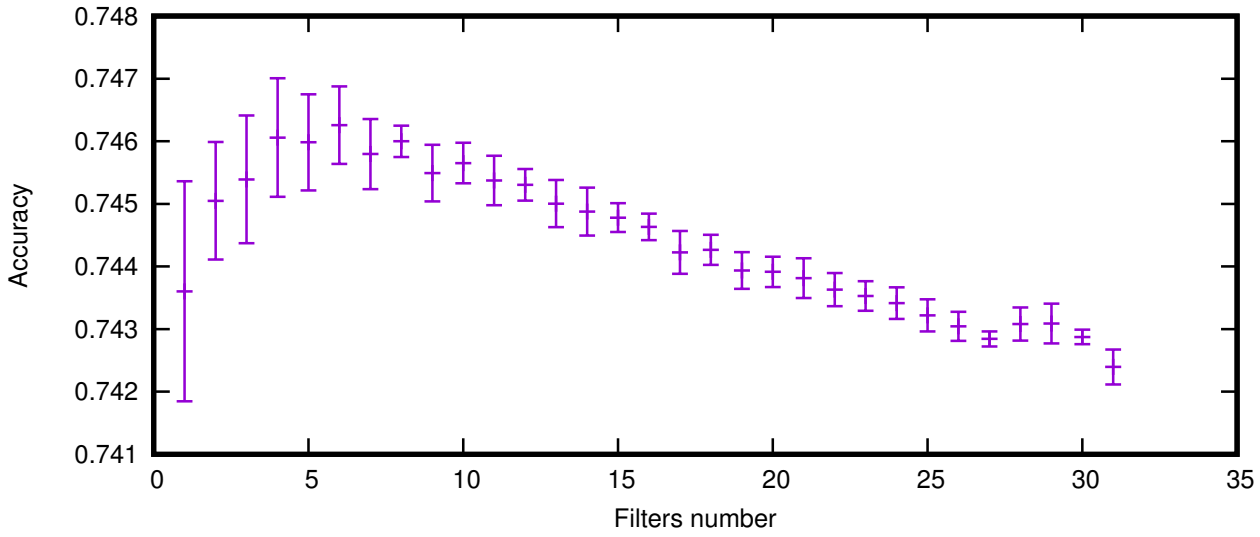

FIGURE 4 – Performance du réseau selon le nombre de filtres

4.2.3 Réseaux convolutionels

On a ensuite évalué l'utilisation des réseaux convolutionnels avec k filtres, puis de mettre un modèle linéaire. La performance avec différents k est décrite dans la figure 4.

On a ensuite rajouté une couche intermédaire ce qui a permis de monter à environ 75%, comme décrit dans la figure 5.

Nous avons essayé d'augmenter le nombre de convolutions, le nombre de filtres, le nombre de neurones dans la couche intermédaire, mais cela n'a pas amélioré le score.

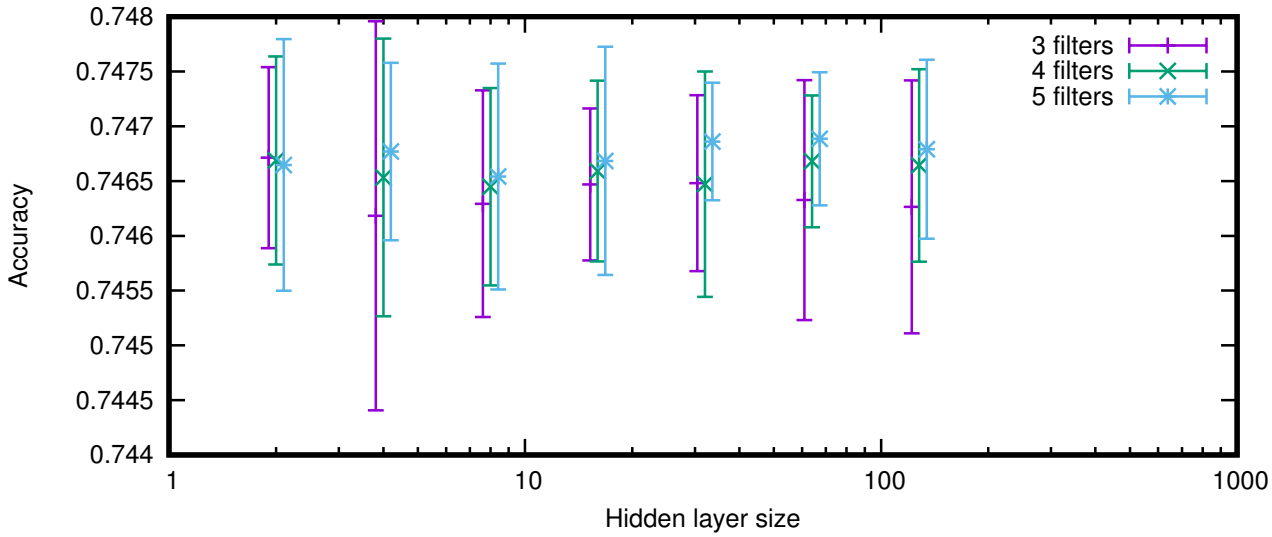

FIGURE 5 – Performance du réseau possédant une convolution et une couche cachée avec différents paramètres

4.3 Autres méthodes

Nous avons utilisé les random-forest, et en limitant la profondeur des arbres on arrive à un score de 73%.

Nous avons utilisé les réseaux réccurents LSTM [7], avec des couches de convolutions avant pour réduire la dimension, mais on arrive autour de 74%.

5 Protection contre ce type d'attaque

On a donc vu comment exploiter un algorithme de cryptographie implémenté naïvement. Pour ne pas pouvoir obtenir de l'information sur un secret, il suffit qu'il n'y ait pas de branchement conditionnel dépendant du secret ni d'accès à un tableau dont l'indice dépend du secret [1].

On peut donc corriger l'implémentation de Montgomery powering ladder, avec un échange conditionnel sécurisé dans l'algorithme 2.

Algorithme 2 : SecureConditionalSwap

Data : $(a, b) \in \mathbb{N}^2$, $c \in \{0, 1\}$
Result : (a, b) si $c = 0$ ou (b, a) si $c = 1$
`/* On exploite le fait que` -1 `est représenté avec tous ses bits à 1 */`
$t \leftarrow (a \oplus b) \wedge (-c)$
$a \leftarrow a \oplus t$
$b \leftarrow b \oplus t$

On peut ensuite adapter l'algorithme Montgomery powering ladder pour le sécuriser, dans l'algorithme 3.

Algorithme 3 : Secure Montgomery powering ladder

Data : $d = d_n \dots d_{1[2]} \in \mathbb{N}$, $x \in G$
Result : $R_0 = x^d$
$R_0 \leftarrow 0$
$R_1 \leftarrow x$
for $i \leftarrow n$ **to** 1 **do**
    `SecureConditionalSwap(`$(R_0, R_1), d_i$`)`
    $R_1 \leftarrow$ `mult(`R_0, R_1`)`
    $R_0 \leftarrow$ `square(`R_0`)`
    `SecureConditionalSwap(`$(R_0, R_1), d_i$`)`
end

Il faut néanmoins faire attention, car dans les phases d'optimisation, le compilateur peut détecter que `SecureConditionalSwap` fait un échange conditionnel et le traduire avec une condition. De plus, on est encore sensible aux attaques par analyse de courant ou émissions électromagnétiques. Il faut pour cela utiliser des méthodes de masquage pour décorréler le calcul des fuites [3].

6 Axes de recherche

Trouver un bit de l'exposant avec 75% de chance, ce n'est pas suffisant pour retrouver la clé. En machine learning, quand on se demande quelle serait la taille nécessaire du jeu de données, la

réponse est souvent : « plus », et on s'est demandés si l'ajout de données d'apprentissage permettrait d'avoir de meilleures performances.

Le graphique en figure 6 donne la performance selon le nombre de données sur lequel est entrainé le réseau de neurones.

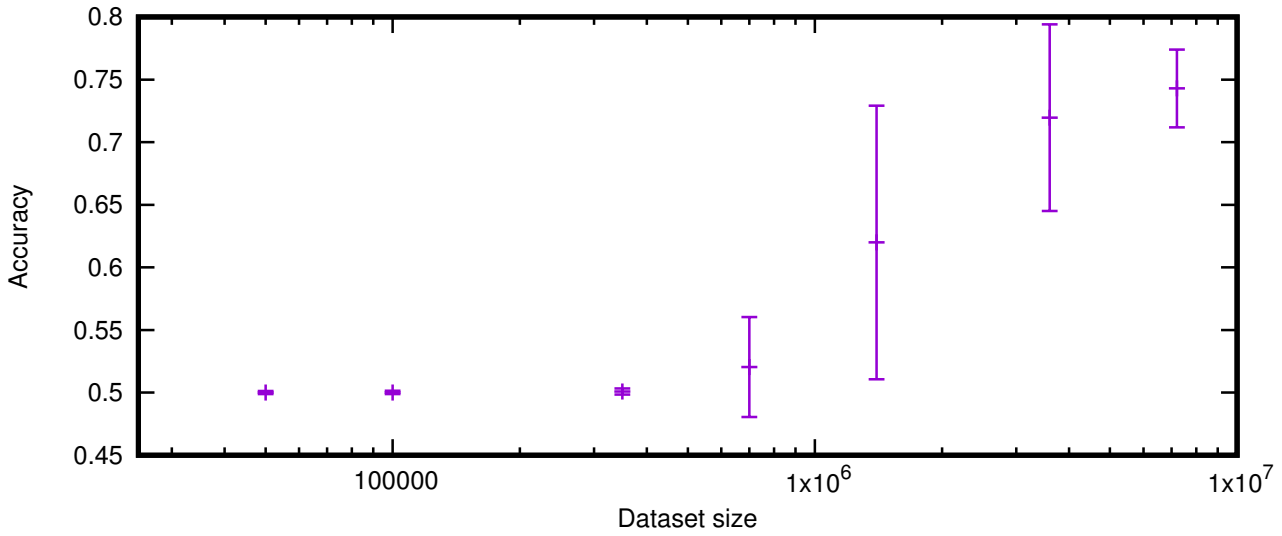

FIGURE 6 – Performance du réseau possédant une convolution et une couche cachée avec différents paramètres

On ne stagne pas à huit millions, donc on peut penser que l'on pourrait améliorer la performance du réseau en ayant plus de données.

On pourrait ensuite imaginer ne pas se limiter à un seul bit, mais obtenir le nombre de zéros à la fin de l'exposant (c'est à dire, la plus grande puissance de 2 divisant l'exposant), ce qui suffirait à avoir une attaque réaliste [11].

Une autre piste de recherche est de se dire que 75% de réussite montre que ces adresses fuitent beaucoup d'information exploitable, et on peut penser que les adresses qui ne sont pas bien choisies n'auront pas un score aussi haut. Ainsi, cela donnerait une nouvelle procédure permettant de réduire l'ensemble des adresses intéressantes à regarder.

7 Conclusion

J'ai choisi ce stage pour avoir une expérience dans le domaine de la sécurité, domaine qui m'intéresse depuis longtemps.

Il y a eu dans ce stage une forte dimension expérimentale à laquelle je n'étais pas habitué ; au début du stage la démarche scientifique se rapprochait beaucoup de la physique : faire des expériences sur une boîte noire (le processeur), et bâtir au fur et à mesure une théorie empirique de son fonctionnement.

Durant ce stage, j'ai plus fait le travail d'un data-scientist que celui d'un chercheur en sécurité : j'ai passé assez peu de temps à générer les traces par rapport au temps que j'ai passé à les analyser.

Néanmoins, j'ai découvert beaucoup d'aspects importants pour implémenter des algorithmes de cryptographie résistants aux attaques side-channel (masking, shuffling, hiding, cryptographic-constant-time, ...) et j'ai pu réaliser que la sécurité est un domaine beaucoup plus vaste que « juste » trouver failles permettant l'exécution de code arbitraire.

Références

[1] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1267–1279, New York, NY, USA, 2014. ACM.

[2] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. *"Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way*, pages 75–92. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[3] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. *Towards Sound Approaches to Counteract Power-Analysis Attacks*, pages 398–412. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[4] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[5] Fail0verflow. Console hacking 2010 : PS3 epic fail. Chaos Communication Congress, `https://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf`, 2010.

[6] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush : A fast and stealthy cache attack. In *DIMVA*, 2016.

[7] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. 9 :1735–80, 12 1997.

[8] Marc Joye and Sung-Ming Yen. *The Montgomery Powering Ladder*, pages 291–302. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[9] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Designs, Codes and Cryptography*, 30(2) :201–217, Sep 2003.

[10] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn : Machine learning in Python. *Journal of Machine Learning Research*, 12 :2825–2830, 2011.

[11] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the flush+reload cache side-channel attack. *IACR Cryptology ePrint Archive*, 2014 :140, 2014.

A DTW – Dynamic Time Warping

On considère un espace vectoriel normé $(V, \|\|)$. On dispose de deux séries temporelles $T^{(1)} \in V^n$ et $T^{(2)} \in V^m$. On aimerait savoir si elles sont similaires ou pas, sachant que l'on ne peut pas comparer point à point car n peut être différent de m, et qu'il peut y avoir des dilatations ou des translations à faire afin de les faire concorder.

Le cas d'application ici est que l'on veut comparer des signaux que l'on a échantillonés le plus vite possible, mais pas de manière régulière.

On appelle une synchronisation un élément $s \in (\llbracket 1, n \rrbracket \times \llbracket 1, m \rrbracket)^*$ où on note $\ell = |s|$ et $S^* = \bigcup_{k=1}^{+\infty} S^k$ et $\llbracket i, j \rrbracket = \{i, i+1, \ldots, j-1, j\}$, qui vérifie les propriétés suivantes :

$$\begin{cases} s_1^{(1)} = 1 \text{ et } s_1^{(2)} = 1 \\ s_\ell^{(1)} = n \text{ et } s_\ell^{(2)} = m \\ (s_{i+1}^{(1)} - s_i^{(1)}, s_{i+1}^{(2)} - s_i^{(2)}) \in \{(1,1),(1,0),(0,1)\} \end{cases}$$

On peut de plus imposer

$$|s_i^{(1)} - s_i^{(2)}| \leq w$$

pour $w \geq |n - m|$ (afin de pouvoir satisfaire la deuxième condition).

On note $\mathbb{S}$ l'ensemble des synchronisations.

Pour une syncronisation $s \in \mathbb{S}$, la distance associée est

$$\mathrm{d}_s(T^{(1)}, T^{(2)}) = \sum_{i=1}^{\ell(s)} \left\| T^{(1)}_{s_i^{(1)}} - T^{(2)}_{s_i^{(2)}} \right\|$$

On note alors

$$\mathrm{DTW}(T^{(1)}, T^{(2)}) = \min_{s \in \mathbb{S}} \mathrm{d}_s \left(T^{(1)}, T^{(2)} \right)$$

On peut le calculer en $O(nm)$ avec de la programmation dynamique, ou en $O(w \max(n, m))$ si l'on utilise le paramètre w.

B ECDSA

ECDSA est un algorithme de signature qui utilise les courbes elliptiques. On considère une courbe elliptique sur un corps fini $\mathbb{F}_p$ (usuellement, p est premier ou une puissance de deux). C'est un groupe abélien, que l'on note additivement. Les points de la courbe elliptiques sont des éléments de $\mathbb{F}_p^2$ On considère un point G de cette courbe elliptique, d'ordre n premier.

Génération des clés On choisit $s \in [\![1, n-1]\!]$ et on calcule $Q = sG$. s est la clé privée, Q est la clé publique.

Signature Pour signer un message m, on utilise une fonction de hachage cryptographique $H : \mathcal{M} \to \mathbb{N}$ et une fonction $\mathrm{int} : \mathbb{F}_p \to \mathbb{N}$

- Choisir k un entier aléatoire cryptographique dans $[\![1, n-1]\!]$
- Calculer $(i, j) = kG$, et $x = \mathrm{int}(i)$
- Calculer $y = k^{-1}(H(m) + sx) \mod n$
- On recommence la signature si $x = 0$ ou $y = 0$. La signature est (x, y)

Vérification de la signature Connaissant x et y on va calculer kG et vérifier que x est correct.

- Calculer $(i, j) = kG = y^{-1}(H(m) + sx)G = y^{-1}H(m)G + y^{-1}xQ$
- Vérifier que $x = \mathrm{int}(i)$

Attaques On constate que si l'on fait deux signatures différentes de m, m' avec le même k, les deux signatures sont (x, y) et (x, y'), alors

$$y - y' = k^{-1}(H(m) - H(m'))$$

donc

$$k = \frac{H(m) - H(m')}{y - y'}$$

puis

$$s = \frac{ky - H(m)}{x} \mod n$$

On a donc obtenu la clé privée. C'est ce qui a été fait pour retrouver la clé de signature des jeux de PlayStation 3 [5].

Avec seulement quelques bits sur k pour plein de signatures, on peut avec un peu plus de travail retrouver la clé privée [9].

C wNAF

Pour mettre un élément d'un groupe à la puissance d, l'algorithme d'exponentiation rapide utilise l'écriture en binaire

$$d = \sum_{i=0}^{n} d_i 2^i$$

où

$$\forall i, d_i \in \{0, 1\}$$

Sur une courbe elliptique, c'est très rapide d'inverser un point ($(x, y) \mapsto (x, -y)$ sur les courbes de Weierestrass), on peut donc imposer :

$$\begin{cases} d_i \in \{-1, 0, 1\} \\ \forall i, d_i \neq 0 \Rightarrow d_{i+1} = d_{i-1} = 0 \end{cases}$$

Il y a existence et unicité de cette écriture.

On peut généraliser cela à :

$$\begin{cases} d_i \in \{-(2^w - 1), -(2^w - 3), \ldots, -3, -1, 0, 1, 3, \ldots, 2^w - 3, 2^w - 1\} \\ \forall i, d_i \neq 0 \Rightarrow d_{i-w} = d_{i-(w-1)} = \cdots = d_{i-1} = d_{i+1} = \cdots = d_{i+(w-1)} = d_{i+w} = 0 \end{cases}$$

C'est l'écriture wNAF (w-ary Non Adjacent Form).

En pratique, $w = 3$ et cette écriture permet d'effectuer une exponentiation rapide en précalculant $-(2^w - 1)G, -(2^w - 3)G, \ldots, -3G, -G, G, 3G, \ldots, (2^w - 3)G, (2^w - 1)G$.