RAPPORT DE PROJET 3A: DEMI PARCOURS

Synthèse de S-box efficace

17 décembre 2018

Augustin BARIANT, X2016

TABLE DES MATIÈRES

1 RESUMÉ

Serpent est un algorithme de chiffrement par block présenté au concours AES en 1997. Il arrivera en finale et obtiendra la deuxième place, derrière Rijndael. Serpent utilise des S-boxes dans son chiffrement et son déchiffrement. Les S-boxes sont des permutations de 4 bits (ie des permutations de S_{16}), avec différentes propriétés de chiffrement que nous n'allons pas aborder dans ce rapport. Le projet s'articule autour de l'optimisation de calcul de S-boxes.

2 L'APPROCHE DE L'OPTIMISATION

2.1 Les S-boxes

Les S-boxes sont des permutations de S_{16}. Puisque chaque élément de $\{0, 1, ..., 15\}$ peut être écrit avec 4 bits, on peut aussi interpréter les S-boxes comme des bijections de $\{0, 1\}^4$ dans lui-même. Serpent utilise 8 S-boxes spécifiques pour le chiffrement, ainsi que leurs inverses utilisés pour le déchiffrement. Voici par exemple une des S-boxes : S_2

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$S_2(x)$	8	6	7	9	3	12	10	15	13	1	14	4	0	11	5	2

Maintenant, on peut écrire l'entrée et la sortie de S_2 en binaire, à l'aide d'une table de vérité.

x_3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
x_2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
x_1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
x_0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

s_3	1	0	0	1	0	1	1	1	1	0	1	0	0	1	0	0
s_2	0	1	1	0	0	1	0	1	1	0	1	1	0	0	1	0
s_1	0	1	1	0	1	0	1	1	0	0	1	0	0	1	0	1
s_0	0	0	1	1	1	0	0	1	1	1	0	0	0	1	1	0

Chaque colonne du tableau contient une représentation binaire de l'entier en entrée et de l'entier en sortie. x_0, x_1, x_2 et x_3 représentent l'entier x en entrée. De la même manière, s_0 correspond au bit le moins significatif de $S_2(x)$, s_1 à son deuxième bit le moins significatif etc...

2.2 Définition des opérations réalisables

On suppose que l'on possède 5 registres r_0, . . . , r_4, disponibles pour nos calculs, et quatre d'entre eux contiennent initialement les 4 bits d'entrée (r_i contient $x_i, 0 \leq i \leq 3$). r_4 est initialisé à 0. On a donc cet état initial :

r_4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
r_3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
r_2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
r_1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
r_0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Si l'on effectue des opérations sur les lignes, et que l'on retombe à la fin sur les lignes s_0 à s_3, alors effectuer ces opérations sur un entier entre 0 et 15 revient à appliquer S_2 à cet entier. Voici la liste des instructions x86 utilisables pour les S-boxes :

Instruction	Effet	Expression
and a, b	$a := a \cdot b$	a &=b
or a, b	$a := a + b$	a | = b
xor a, b	$a := a \oplus b$	a $\wedge$ = b
not a	$a := a \oplus 1$	a = $\sim$ a
mov a, b	$a := b$	a = b

Par exemple, on peut exécuter deux instructions $r_4 := r_0$; $r_0 := r_0 \cdot r_2$, ce qui nous donne ce nouvel état :

r_4	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
r_3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
r_2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
r_1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
r_0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1

Puis exécuter la série d'instruction suivante nous amène au résultat :
$r_0 := r_0 \oplus r_3; r_2 := r_2 \oplus r_1; r_2 := r_2 \oplus r_0; r_3 := r_3 + r_4; r_3 := r_3 \oplus r_1; r_4 := r_4 \oplus r_2; r_1 := r_3; r_3 = r_3 + r_4;$
$r_3 := r_3 \oplus r_0; r_0 := r_0 \cdot r_1; r_4 := r_4 \oplus r_0; r_1 := r_1 \oplus r_3; r_1 := r_1 \oplus r_4; r_4 := r_4 \oplus 1;$

$r_4 = s_3$	1	0	0	1	0	1	1	1	1	0	1	0	0	1	0	0
$r_3 = s_1$	0	1	1	0	1	0	1	1	0	0	1	0	0	1	0	1
$r_2 = s_0$	0	0	1	1	1	0	0	1	1	1	0	0	0	1	1	0
$r_1 = s_2$	0	1	1	0	0	1	0	1	1	0	1	1	0	0	1	0
r_0	0	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0

On a donc réussi à retrouver notre S-box S_2 avec une séquence de 16 instructions.

2.3 Notre algorithme

L'objectif de ce projet est d'implémenter un algorithme qui prend en paramètre une permutation P de S_{16} et qui détermine la plus petite séquence d'instruction qui, appliquée aux 5 registres initiaux, donne P. Le processeur pouvant avoir plusieurs cœurs, il faut aussi trouver des séquences d'instructions permettant le parallélisme qui donnent le même résultat. Pour l'instant, nous n'avons pas abordé ce sujet.

Supposons que nous voulions retrouver une permutation P donnée en paramètre. L'algorithme fonctionne par Brute-Force. Nous allons appliquer toutes les séquences d'instructions possibles aux registres initiaux, à la recherche de lignes égales à celles de la permutation P. On parcourt donc l'arbre des séquences d'instructions possibles en largeur tant que l'on ne trouve pas de solutions. En ce qui concerne le parallélisme, deux instructions peuvent être lancées en parallèle si aucune d'entre elles ne lit une ligne modifiée par l'autre instruction.

2.4 Problèmes rencontrés

Cette recherche exhaustive des solutions possède bien évidemment une complexité exponentielle. De plus, à chaque étape, il faut tester pas moins de 5 instructions différentes possibles, appliquées à 2 registres qu'il faut choisir parmi les 5 possibles (sauf pour le Not, où il suffit de choisir un seul registre). Cela fait donc $4*20+5 = 85$ instructions à tester. Sans introduire de conditions sur les instructions à appliquer, la complexité est bien trop élevée pour pouvoir retourner des solutions aux différentes S-boxes de Serpent.

Naturellement, il va donc falloir exclure certaines séquences d'instructions lors du parcours en profondeur. Voici les principales conditions imposées aux instructions pendant la recherche :

— La recherche s'arrête lorsque les registres ne peuvent plus générer une permutation, autrement dit si deux colonnes sont égales.

— A chaque étape, on stocke dans un HashSet l'ensemble des configurations déjà obtenues, et si l'on retombe sur une configuration du HashSet, on arrête la récursion.

— Aucune instruction autre que Mov n'a le droit d'écrire dans un registre le contenu d'un autre registre.

— Les registres non encore lus ne peuvent pas être modifiés par l'instruction Mov.

— Une instruction ne peut modifier un registre ni en une ligne de 0, ni en une ligne de 1.

— Les registres qui ont été affecté par l'instruction Not portent un flag. Ils ne peuvent pas être réinversés par l'instruction Not.

— Les instructions n'utilisent que 5 registres.

— La récursion requiert un nombre croissant de registres égaux aux lignes de la permutation en paramètre (représentée sous forme binaire). A chaque étape, on regarde combien de lignes de la table de vérité de S_2 sont présentes dans les registres, et on stocke cela dans une variable $NumberOfMatch$, allant de 0 à 4. Si ce nombre diminue après une instruction, on arrête la récursion.

Ce dernier point est le moins évident à gérer. Dans notre état courant des choses, l'algorithme fonctionne en théorie (testé sur des permutations nécessitant des séquences d'instructions de longueur abordable) mais est trop long pour trouver des séquences pour les S-boxes. Une autre solution consiste à exiger que $NumberOfMatch$ doive être au moins égal au $NumberOfMatch$ le plus haut jamais atteint par une séquence d'instruction lors du parcours en largeur. Dans ce cas, cela raccourcit la recherche, puisque un bon nombre de séquences sont filtrées brutalement dès que le $NumberOfMatch$ maximal augmente de 1. En vingt minutes, on trouve une solution pour S_2. Cette séquence d'instruction est de longueur 20, ce qui n'est pas satisfaisant car ce n'est pas la plus courte séquence d'instruction possible. En effet, on ne fait plus une recherche exhaustive, puisque l'on supprime

beaucoup de configurations lorsque $NumberOfMatch$ est incrémenté. On pourrait établir une nouvelle solution en autorisant un retard de 1-2 instructions pour atteindre chaque palier. Même dans ces cas, l'algorithme actuel est trop long.

2.5 Structure de l'algorithme

L'algorithme est écrit en java et divisé en 7 classes.

— *Instruction.java* énumère les différentes instructions possibles, sans prendre en compte les registres.

— *FullInstruction.java* énmuère les différentes instructions possibles, en prenant en compte les registres sur lesquels l'instruction est appliquée

— *InstructionCycle.java* renvoie une liste exhaustive de toutes les instructions possibles

— *WorkspaceKey.java* s'occupe de créer une clé de hachage associé à une séquence d'instruction, permettant de travailler avec le HashSet dont nous avons précédemment parlé.

— *Optimizer.java* est un objet associé à chaque séquence d'instruction. Il garde en mémoire la configuration courante des registres, ainsi que des flags "Read" et "Not". Il peut appliquer une instruction à la configuration, et procède à la vérification des conditions listées un peu plus haut.

— *OptimizerSolver.java* crée un objet Optimizer de base, et s'occupe de faire la récursion, en stockant les objets dans une liste.

— *Tests.java* permet de faire des tests automatiques sur OptimizerSolver.

2.6 Travaux à venir

L'algorithme n'est pas terminé, il reste encore certaines tâches à réaliser :
— Optimiser l'algorithme afin de réduire son temps de calcul.
— Prouver l'exhaustivité de l'algorithme, en particulier avec la dernière condition ($NumberOfMatch$ croissant).
— Gérer l'algorithme pour un parallélisme avec au moins deux unités d'exécution arithmétique.

3 BIBLIOGRAPHIE

Voici les documents que nous avons utilisé afin de réaliser ce travail :

— *Serpent : A Proposal for Advanced Encryption Standard* : https ://www.cl.cam.ac.uk/ rja14/Papers/serpent.pdf

— *Speeding up Serpent* : https ://www.ii.uib.no/ osvik/pub/aes3.pdf