Whitebox

David Wong Jacques Monin Hugo Bonnin

10 avril 2014

Table des matières

1 Introduction

1.1 Utilisation

1.2 Problèmes

l'attaquant à accès à la mémoire et donc il peut facilement récupérer la clef en analysant l'exécution du programme.
De base il peut aussi choisir ce qu'il envoie au programme et voir comment le programme l'encrypte ou le décrypte (chosen plaintext attack) (cas d'une blackbox)
Whitebox : l'attaquant a encore plus de possibilité puisqu'il contrôle l'environnement : accès à la mémoire, trace, breakpoints,...

1.3 Solution

Le but est de rendre l'extraction de clef impossible. Pour cela, nous allons prendre un algorithme connu : DES et y appliquer de la tabularisation ainsi que de la délinéarisation qui sont des concepts que nous expliquerons ultérieurement.

2 DES

2.1 Pourquoi DES ?

3 Principes et Concepts

3.1 Partial Evaluation

On cherche à cacher la clef. La première étape est de pré-calculer toutes les opérations qui sont déjà connu dans notre cas. Puisque dans une whitebox la clef est déjà connu, on pré-calcule toutes les sous-clefs et on cache l'étape du XOR avec ces sous-clefs dans des Lookup tables.

Les look up tables sont des tableaux qui listent les sorties possibles en fonction des entrées possibles.

3.2 Tabularizing

Cette seconde étape consiste à transformer toutes les opérations en Lookup tables.

3.3 Randomization and De-linearization

aussi appelé I/O-blocked Encoding :

Une fois que toutes les opérations ont étés codés en Lookup tables, il est encore facile de retrouver les matrices qui forment toutes ces transformations et donc d'en extraire la clef. Une façon de rendre ce travail trop fastidieux est d'encoder les entrées et sorties de ces Lookup tables avec des bijections (pour pouvoir les annuler d'une Lookup table à une autre).

4 Principes et concepts secondaires

4.1 Mixing Bijection

La création des Lookup tables dans la tabularization se fait à partir de matrices représentant les opérations du cipher. Souvent ces matrices contiennent beaucoup plus de 0 que de 1 ce qui rend les lookup tables trop simples et ce qui crée donc un nouveau problème. Pour éviter ce genre de problème on crée deux opérations au lieu d'une. La première composé d'une bijection. La deuxième composé de son inverse multiplié par l'opération qu'on veut complexifier.

5 Implementation

5.1 Partial Evaluation

On crée un programme prenant une clef en entrée et générant toutes les lookup tables dans un fichier qui sera utilisé pour compiler la whitebox plus tard.

5.2 Tabularizing

Dans ce même programme on transforme toutes les opérations en matrices. Chaque matrice est découpé en sous-matrices et la multiplications des sous-matrices au vecteur associé est transformée en Lookup tables. On utilise des blocs de matrice assez conséquent pour éviter les blocs nuls (il y a beaucoup de zéros dans ces matrices). Ensuite tous les XOR sont transformés en Lookup tables aussi qu'on appellera des XOR tables.

Après cette étape toutes les opérations ont étés codés en lookup tables

Par exemple, nous allons décomposer une matrice 16*16 en sous-matrices de 8x4.

Voici l'opération sans la décomposition

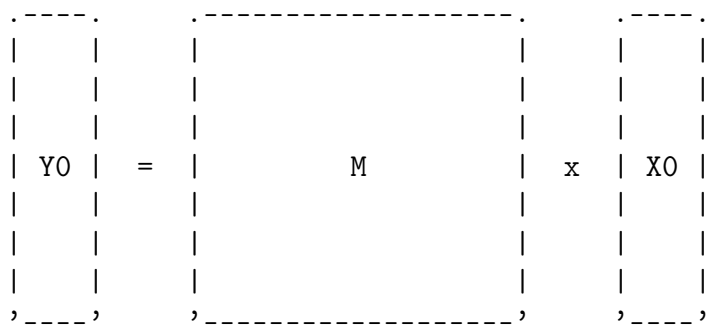

FIGURE 1 – Avant décomposition en sous-matrice

Voici l'opération après la décomposition en sous-matrice

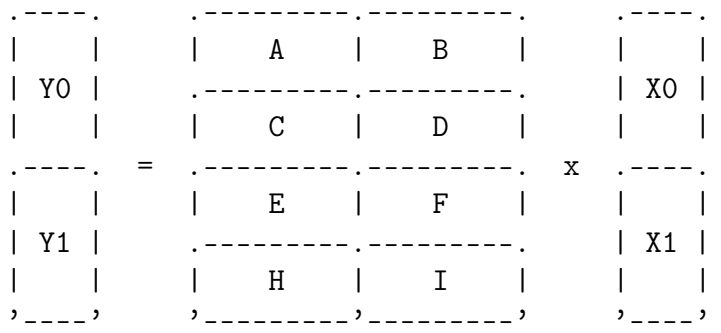

FIGURE 2 – Après décomposition en sous-matrice

Chaque sous-matrice va être la source d'une nouvelle look up table.

Cette look up table va être créée par le multiplication d'une sous-matrice avec toutes les possibilités d'input.

Elle aura donc $2^8 = 256$ entrées possibles et $2^4 = 16$ sorties possibles.

Ainsi pour une matrice (m,n), et pour une sous-matrice (m_1,n_1), il y aura $\frac{m*n}{m_1*n_1}$ look up tables créées qui prendront (2_1^n) valeurs et pourront engendrer (2_1^m) sorties.

5.3 Randomization

Pour compliquer la compréhension du programme on peut mélanger l'ordre dans lequel on envois les bits à notre programme. Puisqu'on a les matrices $M_1, M_2 et M_3$, il suffit de les multiplier par d'autres matrices de mélange avant et après.

Par exemple le premier round représenté par notre implémentation :

$M_1.M_2.data$

data étant un vecteur de 64bits

peut se remplacer par

$(M_1.R_1^-1).(R_1.M2).data$

* De-linearization :

Une fois que toutes les opérations sont transformés en lookup tables. On rajoute des encodages à ces look up tables.

Par exemple L_1 et L_2 deux look up tables qui se suivent.

$L_1 \circ L_2$

on crée une clef pour L_1 et on XOR cette clef à ses sorties. Ce qui fait que L_1 ressemble sous le manteau à $output = (L_1(input) \oplus k_1)$

et on décrypte cette encryption dans L_2 en faisant un XOR dans ses inputs (L_2 ressemble alors à output = $L_2(input \oplus k_1)$.

Si L_1 sort un octet, et L_2 prend un octet en entrée. Alors on a un encodage sur les 256 possibilités de la Look up table. Ce qui est assez facile à retrouver avec une recherche exhaustive. C'est pourquoi on crée un nombre de Look up tables différentes pour les XORs assez important pour rendre cette recherche exhaustive inefficace. servir

6 Notre implémentation

On utilise DES petite intro

6.1 Traduction des concepts

6.1.1 Bypass

puisqu'on a, au maximum un état de 96bits dans la mémoire, on va garder cet état de 96 bits tout le long et utiliser les bits qui ne servent à rien comme bypass.

6.2 Différentes étapes

6.2.1 Étape préliminaire

Avant que les 16 rounds ne s'exécutent, il est nécessaire que les bits d'input subissent une permutation initiale suivie d'une expansion de l'input de 64 bits en 96 bits qui permettra le bypass. Ces deux opération seront réalisées par une matrice M1 qui ne sera utilisée qu'une fois juste avant les 16 rounds et qui réalisera ceci :

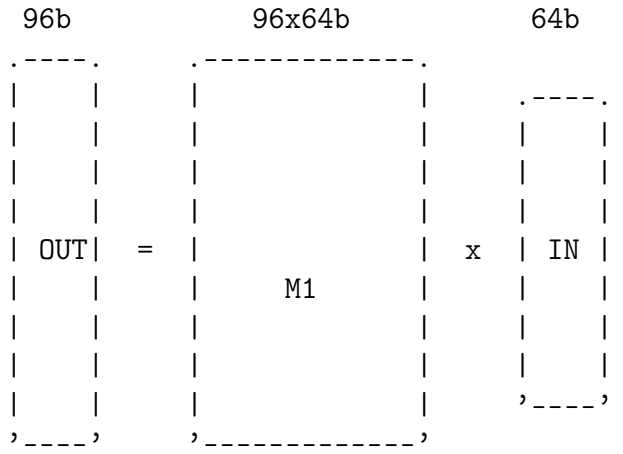

FIGURE 3 – M1 avant décomposition en sous-matrice

Nous décomposons M1 en sous-matrice de taille 4x8. Nous multiplions chacune de ces $\frac{96*64}{4*8} = 288$ sous-matrices par les $2^8 = 256$ différentes possibilités ayant $2^4 = 16$ résultats possibles.

6.2.2 Étape 1 à Étape 2

Nous devons convertir ce fonctionnement :

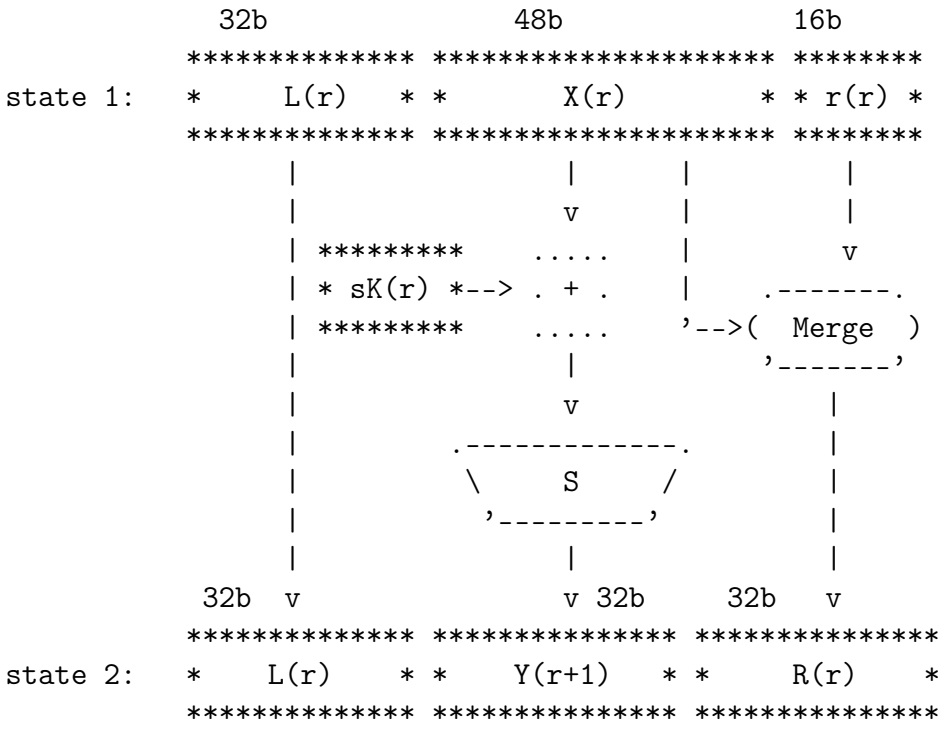

FIGURE 4 – Avant Tabularisation

En celui-ci :

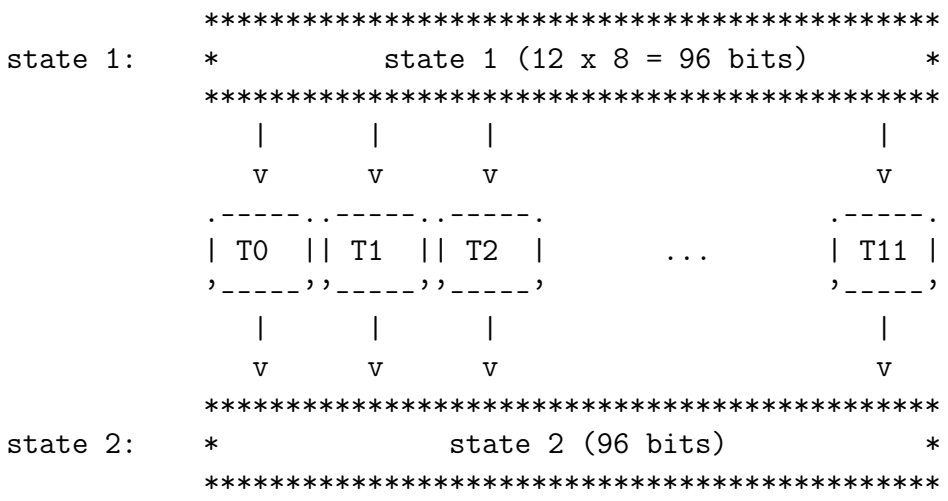

FIGURE 5 – Après Tabularisation

Pour ce faire, nous allons calculer 12 Look up Tables qui prendront 8 bits chacun ce qui recouvrira les 96 bits d'input.

Il y a :

8 look up tables non linéaires qui permettent le xor avec la clef et la substitution en les précalculant.

4 look up tables linaires qui nous serviront à bypasser les bits qui ne subissent pas de calculs.

6.2.3 Étape 2 à Étape 3

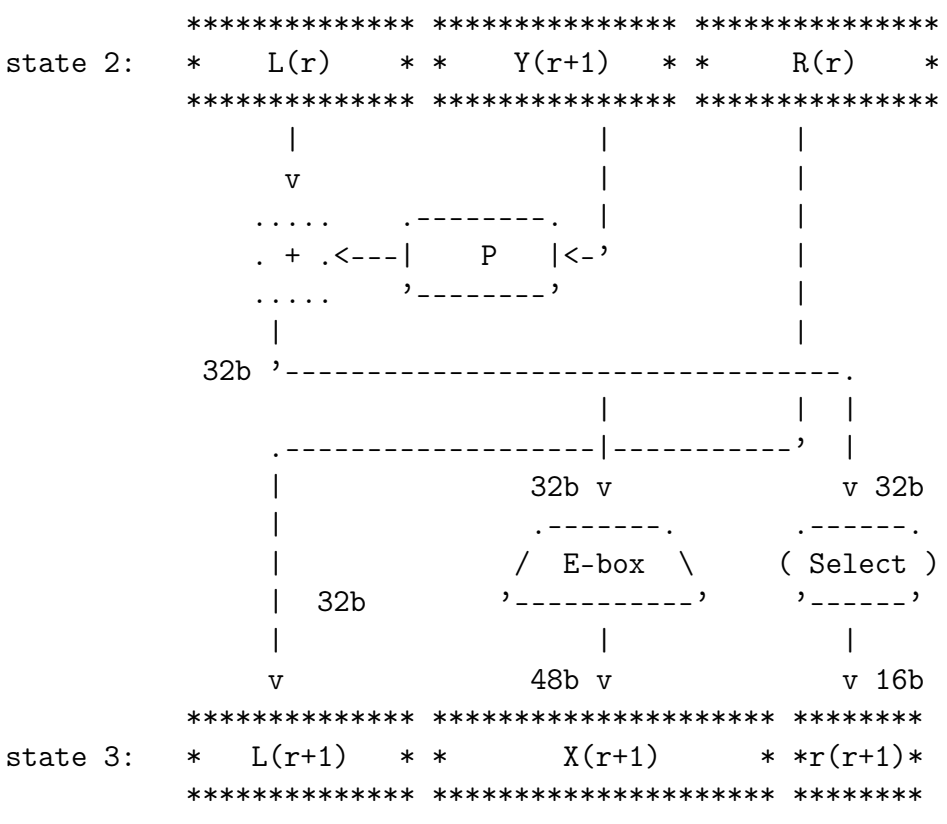

FIGURE 6 – Avant Tabularisation

7 Conclusion

8 Bibliographie