Implémentation et Analyse d'une White-box du DES

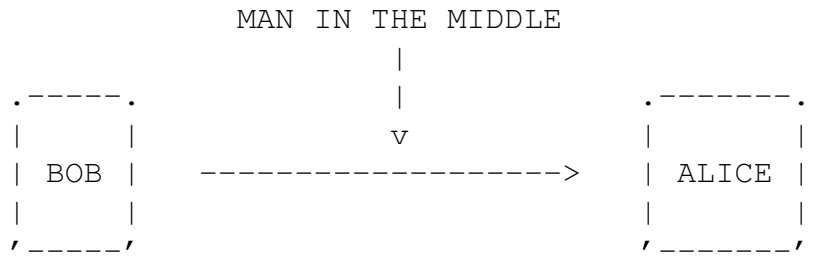
David Wong Jacques Monin Hugo Bonnin

16 avril 2014

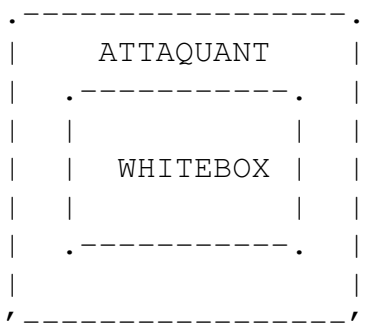
Table des matières

1 Introduction

La cryptographie est fondée depuis très longtemps sur ce principe simple : deux personnes veulent pouvoir communiquer entre elles sans pouvoir être écoutées (attaque type Man in the middle).
De nombreux block ciphers comme DES ou AES ont été inventés à travers l'histoire de la cryptographie et ont su rester solides contre la cryptanalyse, jusqu'à un certain point.



Avec les avancées technologiques des dernières années, de nouveaux besoins en cryptographie ont fait surface. Récemment, le besoin d'exécuter du code contenant des informations sensibles sur des machines potentiellement dangereuses a créé de nouveaux domaines de recherche en Cryptologie.



Man at the end : l'attaquant a un accès physique au système. Il peut l'inspecter, lui donner les entrées qu'il veut et analyser les sorties, désassembler le logiciel (IDA), voir les appels au système, voir les accès mémoire, émuler des parties du programme... et finalement ce qui nous intéresse : en extraire la clé.

1.1 Utilisation

Avant d'expliquer une potentielle solution à ce problème, voici quelques illustrations afin de mieux comprendre pourquoi on s'intéresse à cette nouvelle utilisation de la cryptologie :

1.1.1 Backdoors

Imaginons que le gouvernement français veuille fabriquer un backdoor dormant et infecter diverses machines dans les entreprises/associations/gouvernements de leur choix. Imaginons maintenant qu'il veuille envoyer des instructions à ce backdoor sans qu'elles puissent être interceptées et écoutées, parce que les instructions doivent rester secrètes et/ou parce que l'interception de telles instructions en clair pourrait élever des soupçons.

Une des solutions disponibles est d'encrypter ces instructions avec la clé publique/privée de ces backdoors avant de leur envoyer.

Personne ne peut lire ces instructions, personne ne peut envoyer d'instructions.

1.1.2 DRMs

Imaginons maintenant un système de distribution de film. On encrypte le film avec une clé et on l'envoie avec la clé en annexe. La clé en annexe est elle même encryptée avec la clé de l'utilisateur qui souhaite voir le film. Cette méthode est utilisée pour éviter d'encrypter le film à chaque utilisateur.
Le logiciel propriétaire, qui est donc notre fameuse Whitebox, décrypte la clé, décrypte le film ET ajoute un hash de la clé dans le film.
L'attaquant peut maintenant soit essayer d'extraire une des deux clés du programme, soit capturer la sortie vidéo pour copier le film. En copiant la sortie vidéo il copiera le hash intégré dans cet output et pourra être incriminé.

1.1.3 Autres utilisations

Il existe d'autres utilisations comme les mobile-agents ou les smartcards et sûrement bien d'autres encore qui n'ont pas encore été découvertes...

1.2 Solution

C'est là qu'intervient l'idée de la Whitebox. Le but est de rendre l'extraction d'une information importante impossible. Dans notre cas une clé.

Barak et cie ont prouvé qu'un obfuscateur parfait n'existait pas dans un article intitulé "On the (Im)possibility of Obfuscating Programs" [1]
Cela ne nous empêche pas de nous en approcher. Seulement, peu de théorie a encore vu le jour et les techniques que nous expliqueront par la suite, provenant de l'article de Chow et cie, sont basées sur nos connaissances actuelles des attaques et les observations faites sur la modification des algorithmes de base pour affaiblir ces attaques. Il faut noter que l'état de l'art, sans obfuscation pure, c'est-à-dire en dévoilant les techniques et l'algorithme utilisés, ne fournit pas de solution incrackable actuellement (on trouve la clé en l'ordre de quelques secondes). Ces solutions sont néanmoins utilisées commercialement lorsque les algorithmes et techniques utilisés sont méconnus, et rendent donc la tâche beaucoup plus difficile.
Pour finir sur notre explication, une Whitebox est généralement utilisée dans la decryption. Il faut donc, en plus de bloquer l'extraction d'une clé, empêcher l'opération inverse. C'est-à-dire l'encryption. Dans le cas du backdoor, si l'on arrivait à encrypter sans connaître la clé on pourrait alors construire nous même des instructions à destination des backdoors.

1. http ://www.iacr.org/archive/crypto2001/21390001.pdf

2 Data Encryption Standard

Avant de nous lancer dans l'implémentation de la version WhiteBox de DES (Data Encryption Standard), nous avons choisi d'implémenter notre propre version de DES afin de nous familiariser avec son fonctionnement (voir notre algorithme sur GitHub : `https://github.com/mimoo/DES`).
Nous nous intéresserons donc ici au fonctionnement de DES, ainsi qu'à notre approche de son implémentation.

2.1 Historique et intérêt

DES est un algorithme de chiffrement symétrique (ou chiffrement par bloc) qui utilise des clés de 56 bits et manipule des blocs de 64 bits. Il a été développé par IBM au début des années 70 à la demande du National Bureau of Standards (NBS) qui proposait de trouver un moyen de sécuriser des informations sensibles, non classifiées, du gouvernement. En 1976, après consultation de la National Security Agency (NSA), le NBS en sélectionne une version légèrement modifiée, qui sera publiée l'année suivante (le standard arrivant en 1979).
Bien qu'il soit recommandé de ne plus l'utiliser (la dernière version avant l'obsolescence datant de 1999), on peut encore parfois le trouver sous la forme de Triple DES.
L'intérêt principal d'utiliser DES dans un contexte de type WhiteBox est que cet algorithme a seulement besoin de transformations linéaires et de substitutions, ce qui facilite l'étude du principe. De plus l'algorithme de Triple DES étant encore parfois utilisé, il peut être intéressant de savoir en implémenter une version WhiteBox.

2.2 Génération des 16 clés

La clé initiale est représentée sur 64 bits (à laquelle on retire 1 bit de parité à chaque octet).
Un algorithme est utilisé afin de générer les 16 clés (correspondant aux 16 itérations de DES) issues de la clé initiale.
On commence par retirer les 8 bits de parité de la clé (1 bit à chaque octet). La clé obtenue fait alors 56 bits.

1. Permutation PC1 puis séparation en deux blocs (droite et gauche).
2. Rotation de chaque bloc vers la gauche (le bit en première position passe en dernière position).
3. Fusion des deux blocs

4. Permutation PC2 de 56 bits vers 48 bits qui correspond à la clé du même numéro que l'itération actuelle. On réutilise le bloc de 56 bits (d'avant la permutation PC2) pour la prochaine itération.

Après ces étapes le bloc est enfin chiffré. Pour déchiffrer ce dernier, on utilise le même algorithme, à ceci près que l'on change l'ordre d'utilisation des clés et que l'on inverse les deux blocs de gauche et de droite au début et à la fin de DES.

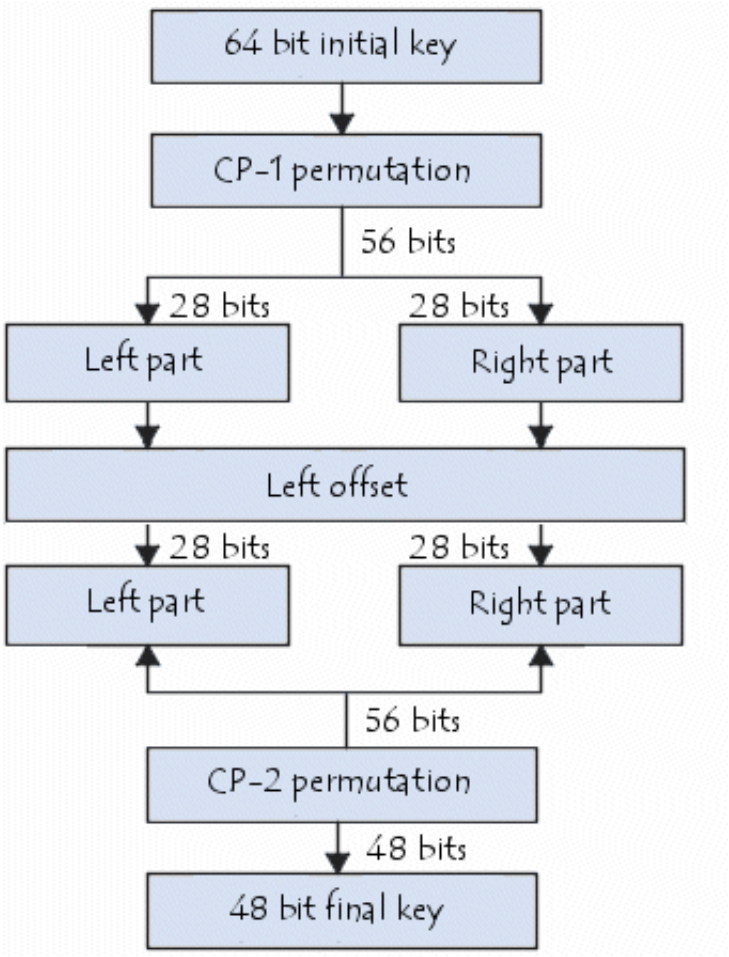


FIGURE 1 – Schéma d'une itération de génération de clé

2.3 Fonctionnement de DES

L'algorithme de DES peut se décomposer en 3 grandes étapes :

1. Une permutation initiale sur le bloc qui est ensuite divisé en deux blocs de 32 bits.
2. S'ensuit un cycle de 16 itérations effectuant les opérations suivantes :
   — Expansion du bloc de droite de 32 bits vers 48 bits (on copie 16 bits du bloc initial, le tout à l'aide d'un table d'expansion).
   — XOR entre le nouveau bloc de 48 bits et la clé correspondant au numéro d'itération actuelle.
   — Le bloc de 48 bits est divisé en 8 blocs de 6 bits. Chacun de ces sous-blocs passe à travers une table de substitution (qui prend 6 bits en input et donne 4 bits en output). Il y a en tout 8 tables de substitution, une pour chaque sous-bloc. La valeur en binaire des 4 bits du milieu détermine l'indice de colonne de la table et la valeur binaire des 2 bits extérieurs l'indice de ligne. Le bloc obtenu en fusionnant les 8 sorties de 4 bits a alors une taille de 32 bits.
   — Permutation sur ce bloc (les bits sont changés de place).
   — XOR entre ce bloc et le bloc de gauche.
   — Les deux blocs sont échangés (le bloc de gauche devient celui de droite et inversement).
3. Enfin, une fusion des deux blocs, suivie d'une permutation finale, inverse de la permutation initiale.

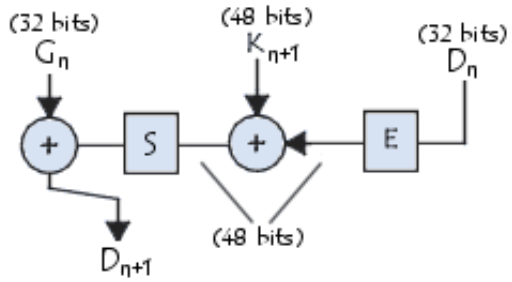


FIGURE 2 – Schéma d'une itération de DES

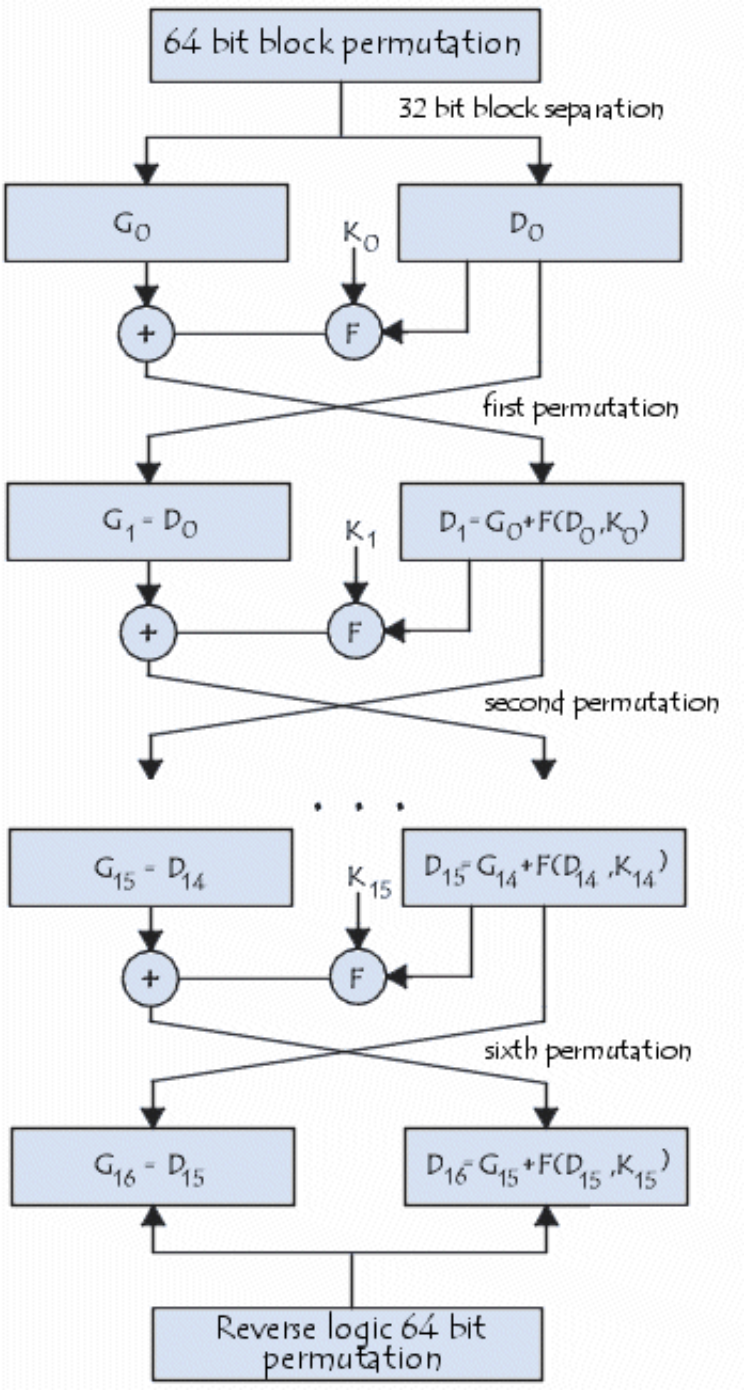


FIGURE 3 – Schéma général de DES

3 Principes et concepts fondamentaux

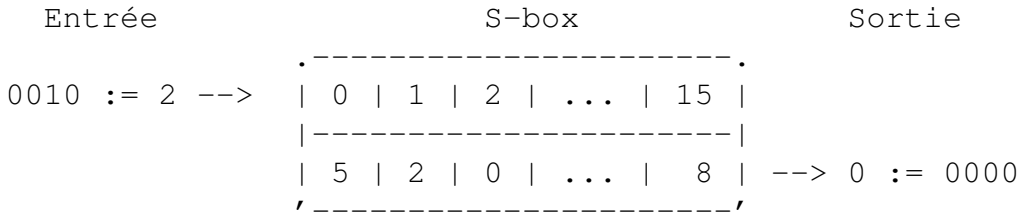
Les Whitebox, telles que définies actuellement, sont des Blackbox destinées à la décryption, à l'intérieur desquelles les clés sont cachées et le processus est assez complexe pour empêcher son inversion.
Plusieurs concepts ont été introduits par Chow et cie dans leur article [2]. Voici les trois principaux concepts :

3.1 Partial Evaluation

L'évaluation partielle consiste à pré-calculer ce que l'on connaît déjà puis à **mélanger l'opération du XOR** de la clé avec une autre opération, dans le cas de DES on mélange cette opération avec les substitutions après le XOR.
Dans DES, 16 sous-clés sont générées à partir de la clé principale puis sont XORées une à une durant les 16 itérations de l'algorithme. Ces 16 clés de 48bits et les différents résultats de leurs XORs, puis de leurs substitutions avec les 2^{48} différents vecteurs possibles à chaque itération sont pré-générés et stockés dans des S-box (Substitution boxes) qu'on appellera **Lookup tables** par la suite.

3.2 Tabularizing

La tabularisation consiste à transformer toutes les autres opérations en **Lookup tables**.



3.3 Input/Output Encoding

Une fois que toutes les opérations ont été transformées en Lookup tables, il est encore facile de retrouver les opérations initiales et donc d'en extraire la clé. Une façon de rendre ce travail trop fastidieux pour un attaquant est de délinéariser ces opérations en encodant les entrées et sorties de ces Lookup tables.

Soit L_1 et L_2 deux Lookup tables se suivant :
$output = L_2(L_1(input))$

2. http ://crypto.stanford.edu/DRM2002/whitebox.pdf

On peut encrypter la sortie de L_1 avec une clé k_1 et décrypter l'entrée de L_2 avec cette même clé. C'est un **cipher de Vernam** :
On remplace donc $L_1(input)$ par $L'_1(input) = L_1(input) \oplus k_1$
et $L_2(input)$ par $L'_2(input) = L_2(input \oplus k_1)$
Ce qui donne : $output = L'_2((L'_1(input)))$.

4 Principes et concepts secondaires

4.1 Randomization

Comme l'encryption des entrées/sorties, on peut aussi **mélanger** les bits sortant d'une opération, puis les remettre dans l'ordre avant la prochaine.

Par exemple, si L_1 et L_2 sont deux lookup tables se suivant :
$output = L_2 \circ L_1(input)$
On peut rajouter une bijection E entre les deux comme ceci :
$L_2 \circ E^-1 \circ E \circ L_1(input)$

4.2 Mixing Bijection

La création des Lookup tables dans la tabularisation 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 est une bijection. La deuxième est l'inverse de cette bijection multipliéé par l'opération qu'on veut complexifier.
Soit M_1 la matrice d'une des opérations que l'on transformera en lookup tables plus tard. On peut la multiplier par une autre matrice G choisie de façon pertinente pour augmenter le bruit dans M_1 (un nombre de 0 et de 1 qui parait aléatoire). De fait que l'opération devienne alors $G \cdot M_1$. Il faut bien sûr annuler G ensuite en créant une seconde opération G^{-1}.
L'opération devient alors : $G^{-1} \cdot (G \cdot M_1)$ où $G \cdot M_1$ est pré-évaluée comme une unique opération.

4.3 By-Pass Encoding

En général, on veut cacher ce qu'une opération fait. L'idée du bypass est d'élargir la taille de son entrée et la taille de sa sortie avec des bits inutils.

4.4 Combined Function Encoding

Lorsque deux opérations sont évaluées en même temps, on peut faire évaluer une entrée composée de leurs deux entrées respectives pour transformer le tout en une seule opération : $(P||Q)(input_P||input_Q)$.

4.5 Split-Path Encoding

Lorsque l'on donne un input de n bits, et que l'on reçoit un output de m bits tel que $m < n$ on a une perte d'information. Pour répondre à ce problème on lui concatène une fonction prenant le même input pour augmenter le nombre de bits de sortie.

4.6 External Encoding

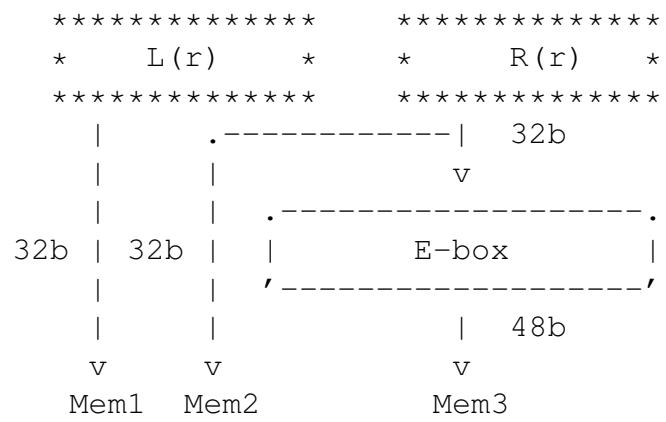
Pour éviter une extraction de l'implémentation de la whitebox, et pour éviter les attaques sur les premières et dernières itérations, on peut appliquer deux bijections à l'entrée et à la sortie du programme.

$Whitebox = E \circ DES(input) \circ G$

5 Implémentation

Notre implémentation d'une whitebox DES en C est disponible sur notre github : `https://github.com/mimoo/whiteboxDES`).
Toutes les Lookup tables seront créées par un programme **Ltablesgen.c** prenant une clé de 64bits en entrée et seront générées dans un fichier qui sera utilisé pour compiler la whitebox plus tard.

5.1 Bypass et Combined Function Encoding



Ci-dessus la première partie d'une itération de la fonction DES. On peut voir que :

1. 32bits de la partie gauche restent inchangés pour pouvoir faire un XOR avec la partie de droite encryptée dans la deuxième partie d'une itération.
2. 32 bits de la partie droite subissent des opérations (expansion, xor avec la clé, substitution, permutation).
3. 32 bits de la partie droite restent inchangés pour prendre la place de la partie gauche dans la prochaine itération.

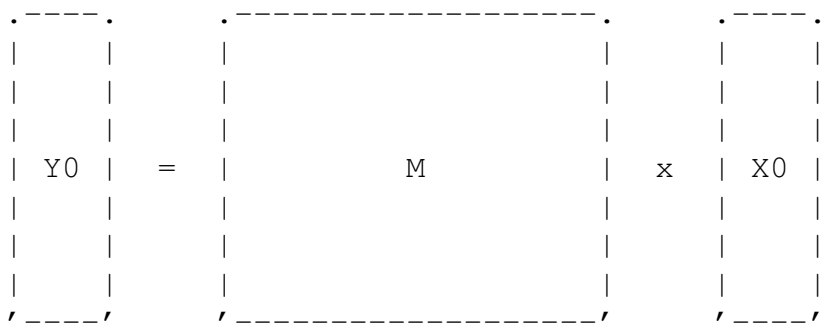
On voit qu'à un moment précis de l'algorithme de DES, la mémoire contient 96 bits. Une analyse de DES plus poussée nous montre facilement qu'il n'existe pas d'état où la mémoire contient plus d'information (en admettant que l'implémentation utilisée soit bien optimisée).
Au minimum la mémoire contient 64bits. On va donc garder cet état de 96 bits tout le long et utiliser les bits qui ne servent à rien comme **bypass**.

5.2 Partial Evaluation

Les 16 sous-clés sont d'abord pré-évaluées. Puis on pré-évalue l'opération du xor avec la clé, suivi de la substitution que l'on stocke dans des Lookup Tables (différentes à chaque itération puisqu'elles sont influencées par les différentes clés).

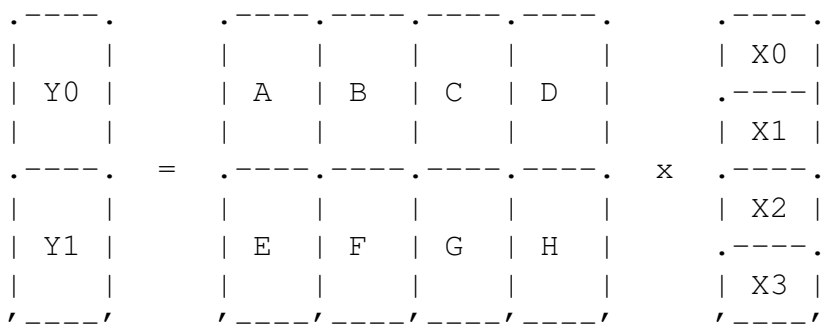
5.3 Tabularisation

La prochaine étape consiste à transformer toutes les opérations en Lookup tables. Pour ce faire on met d'abord sous forme de matrice chaque opération. Le découpage des opérations sous forme de matrice est expliqué plus loin.



Un **premier problème** survient :

Ces matrices contiennent généralement beaucoup trop de zéros (identités, permutations, ...).
On utilise d'abord la technique des **Mixing Bijection** pour séparer la matrice en deux matrices plus complexes (et donc en deux opérations). Mais cela ne suffit pas.
Pour éviter d'avoir trop de Lookup tables nulles et réduire le champ de recherche pour l'attaquant, chaque matrice est découpée en **blocs de sous-matrices**.
Les multiplications des sous-matrices aux vecteurs associés sont ensuite transformées en Lookup tables.



avec Y0 = A*X0 $\oplus$ B*X1 $\oplus$ C*X2 $\oplus$ D*X3
Un **second problème** survient :

On ne peut pas faire des Lookup tables trop grosses car leur taille augmente exponentiellement :

Pour 8 bits d'entrée, il y a 2^8 possibilités. Donc la taille de la Lookup table est de $2^8 * sizeof(output)$.
Pour 16 bits d'entrée, il y a 2^{16} possibilités. Donc la taille de la Lookup table est de $2^{16} * sizeof(output)$.
Avec notre exemple simplifié, en utilisant des Lookup tables prenant 8 bits en entrée, et donc en multipliant par deux le nombre de Lookup tables, on réduit la taille de notre fichier final par $(2^{16} - 2^8 + 2^8) * sizeof(output)$ pour chaque Lookup table. etc...

L'article de Link et cie [3] conseille, pour plus de sécurité, de choisir des blocs de 8x4 à la place de 8x8 que conseillent Chow et cie.

	MiB	Packed MiB
Chow et al. (4×4)	4.54	2.27
8×4	4.49	2.25
8×8	274.75	274.75
8×4, Joined Roots	16.40	14.20
8×4, JR, 3DES	48.83	42.42

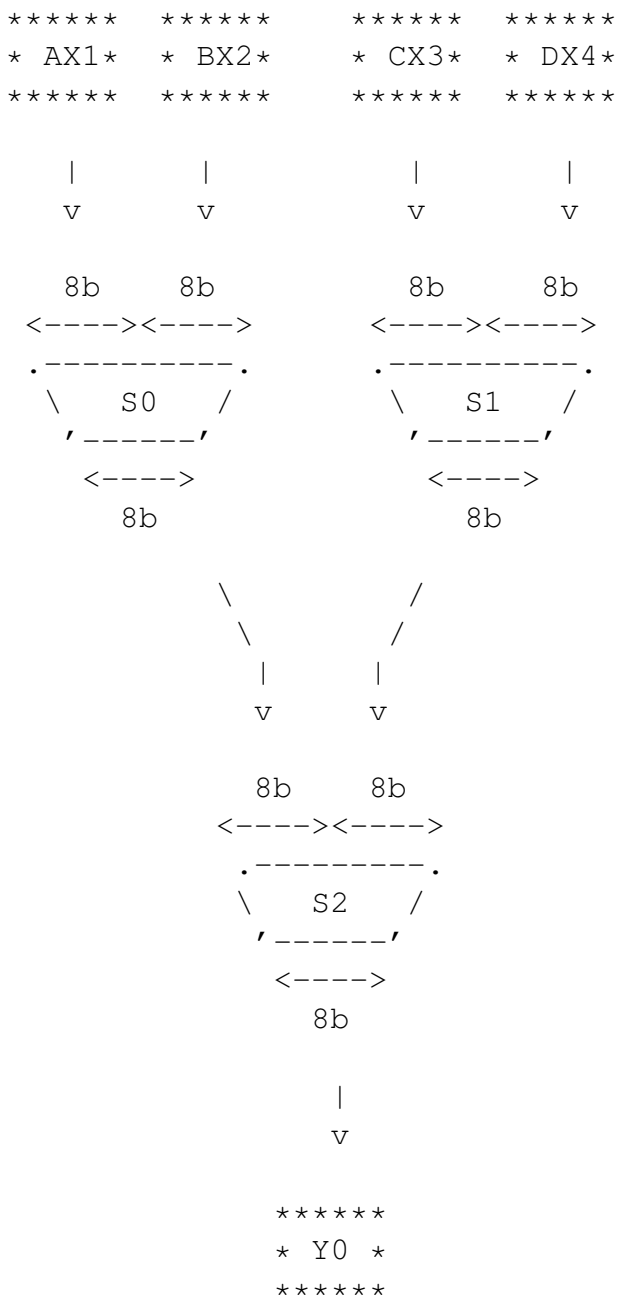
FIGURE 4 – Différence de taille selon la taille des blocs

3. http ://eprint.iacr.org/2004/025.pdf

Chaque sous-matrice va être la source d'une nouvelle lookup table qui sera créée par la multiplication de la sous-matrice avec toutes les possibilités d'input de 4 bits.

Ensuite chaque ligne de Lookup table donnera des outputs de 8 bits qu'il faudra XORer entre eux. Pour continuer la tabularisation, chaque XOR est pré-calculé dans ce qu'on appellera des XOR tables.

Ce qui donne des **réseaux** de lookup tables :



5.4 Split-Path Encoding

On a maintenant un réseau de XOR Tables qui prennent 16 bits en input et rendent 8 bits en output. Il y a donc une perte d'information, ce que l'on peut corriger en

modifiant ces Look up tables pour avoir un output de 16 bits avec 8 bits de bypass. $L'_1(input) = L_1(input)||R(input)$ avec R une fonction rendant 8 bits en output.

5.5 Input/Output Encoding

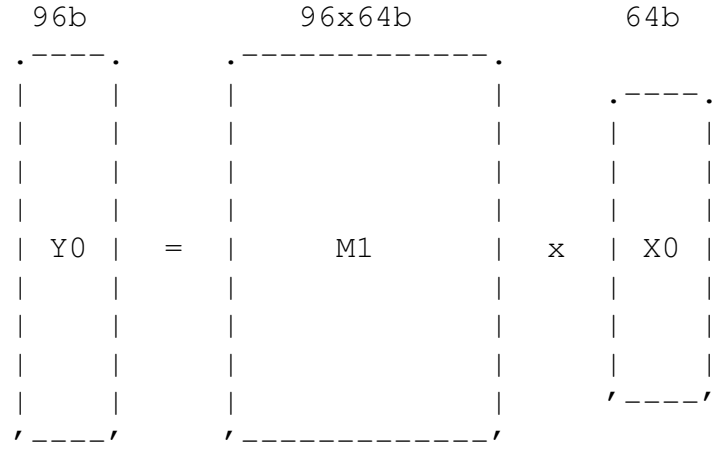
Ce réseau de XOR Tables est maintenant modifié en encryptant/décryptant les entrées-sorties entre opérations. On peut générer des clés de 16 bits en fonction de la clé principale.

5.6 Matrices

La suite explique comment les opérations ont été réduites en 3 matrices M_1, M_2 et M_3

5.6.1 Étape préliminaire

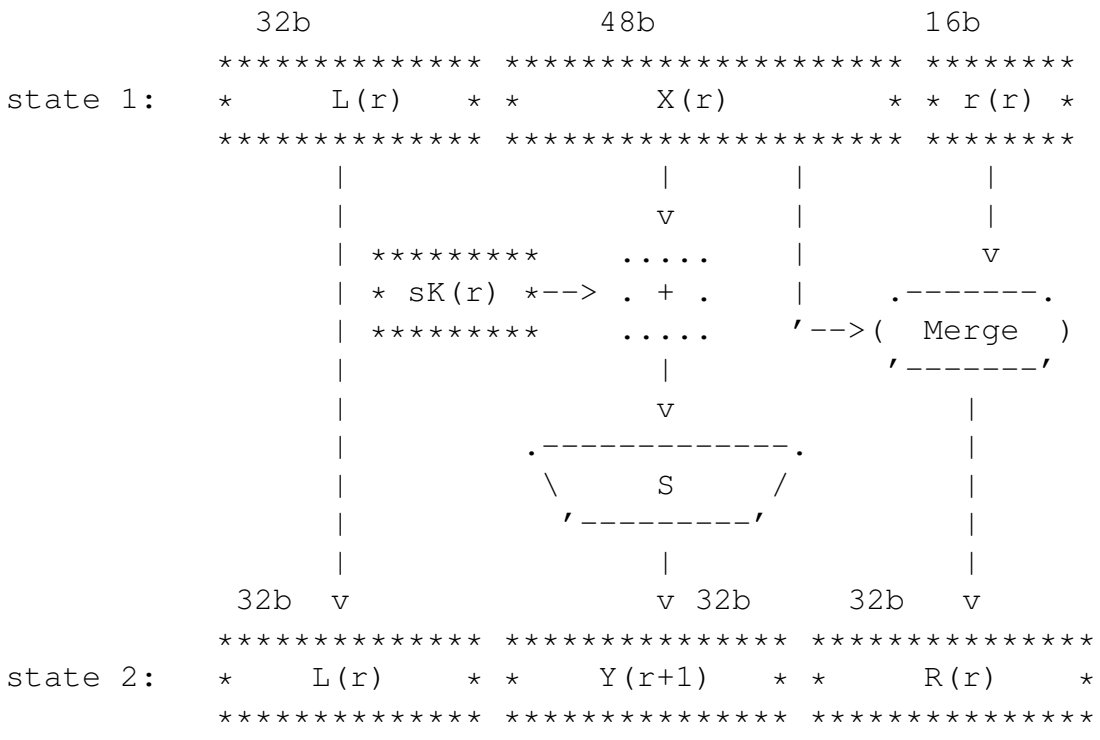
Avant que les 16 itérations ne s'exécutent, il est nécessaire que les bits d'input subissent une permutation initiale suivie d'une expansion de l'input de 64 bits en 96 bits qui permettra le bypass. Ces deux opérations seront réalisées par une matrice M1 qui ne sera utilisée qu'une fois juste avant les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



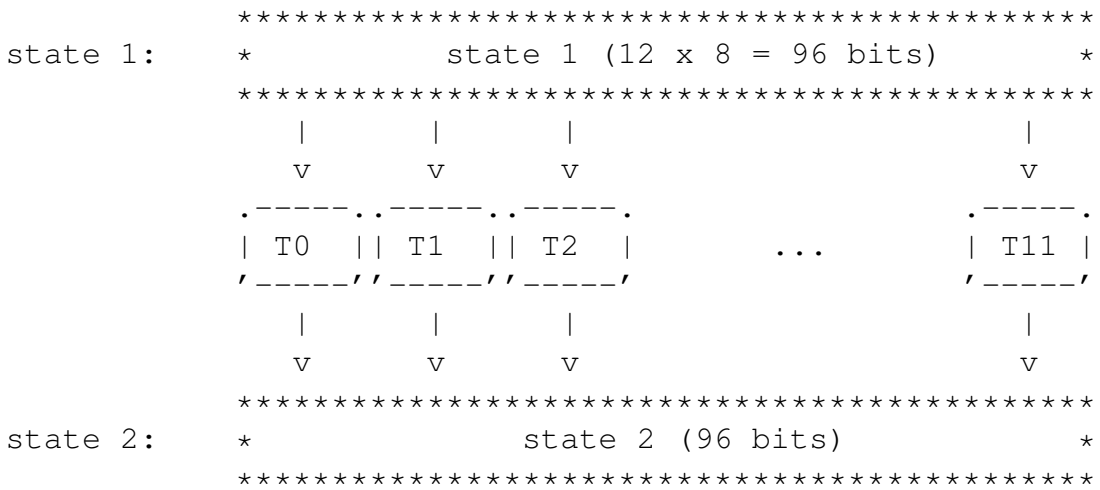
Nous décomposons M1 en sous-matrices de taille 4x8. Nous multiplions chacune de ces $\frac{96*64}{4*8}$ = 288 sous-matrices par les 2^8 = 256 différentes possibilités ayant 2^4 = 16 résultats possibles.

5.6.2 Étape 1 à étape 2

Nous devons convertir ce fonctionnement :



En celui-ci :

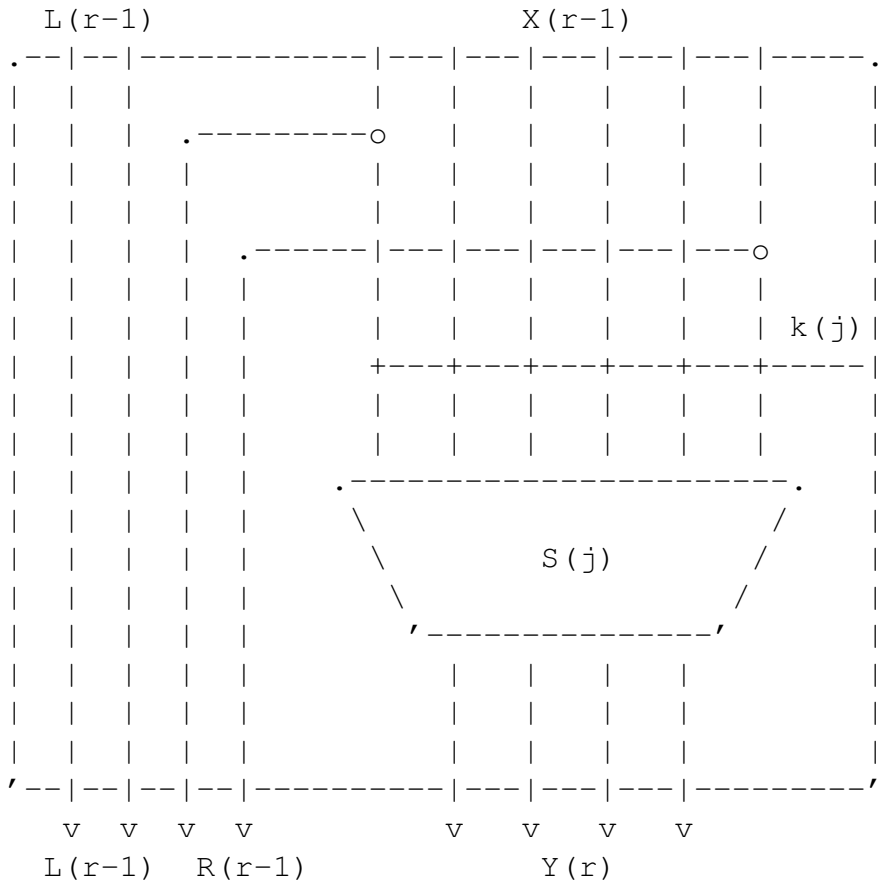


Pour ce faire, nous allons calculer 12 Lookup Tables qui prendront 8 bits chacun ce qui recouvrira les 96 bits d'input et qui seront composées de :

1. 8 lookup tables non linéaires qui permettent le XOR avec la clé et la substitution en les pré-calculant grâce à la formule :

$$T_i(x) = S_i(x \oplus k_i)$$

Une permutation est faite sur les données, chaque lookup table non linéaire va donc prendre en paramètre deux bits de L(r-1) et 6 bits de X(r-1) et renverra 2 bits de L(r-1), 2 bits de R(r-1) et 4 bits qui formeront Y(r). Elles sont de la forme :

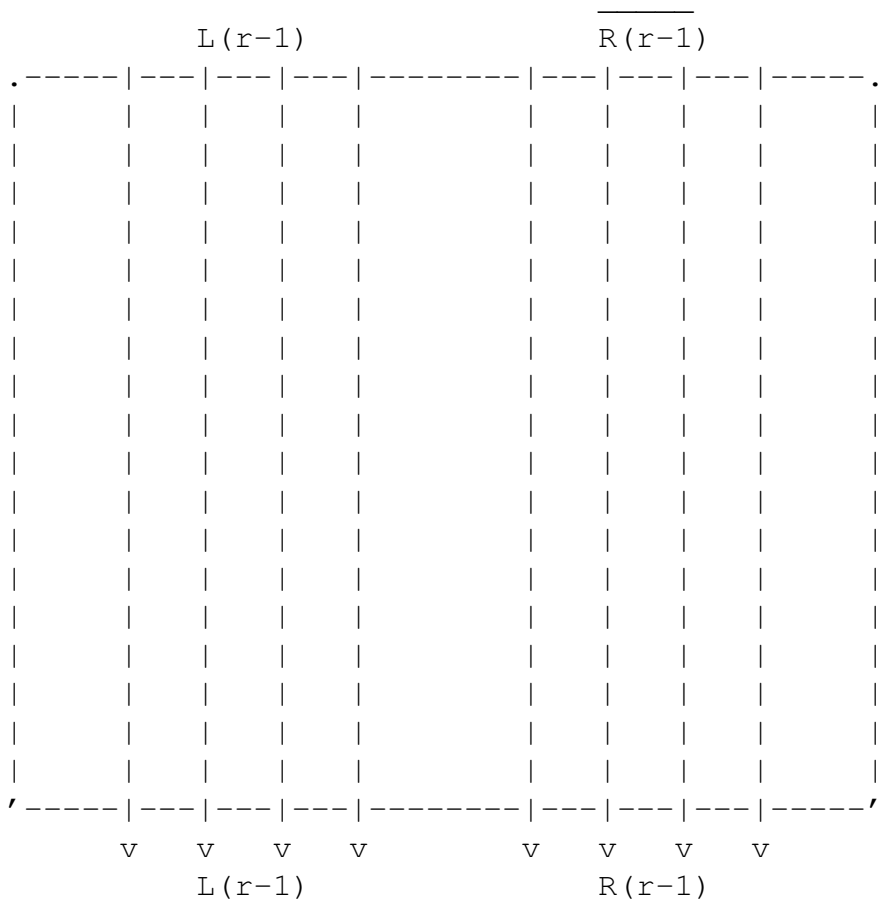


L'output de ces lookup tables est donc :

$$T_i^r = b_0 b_1 || b_2 b_7 || S_j(b_2 b_3 b_5 b_6 b_7 \oplus k_i^r)$$

2. 4 lookup tables linéaires qui nous serviront à bypasser les bits qui ne subissent pas de calculs.

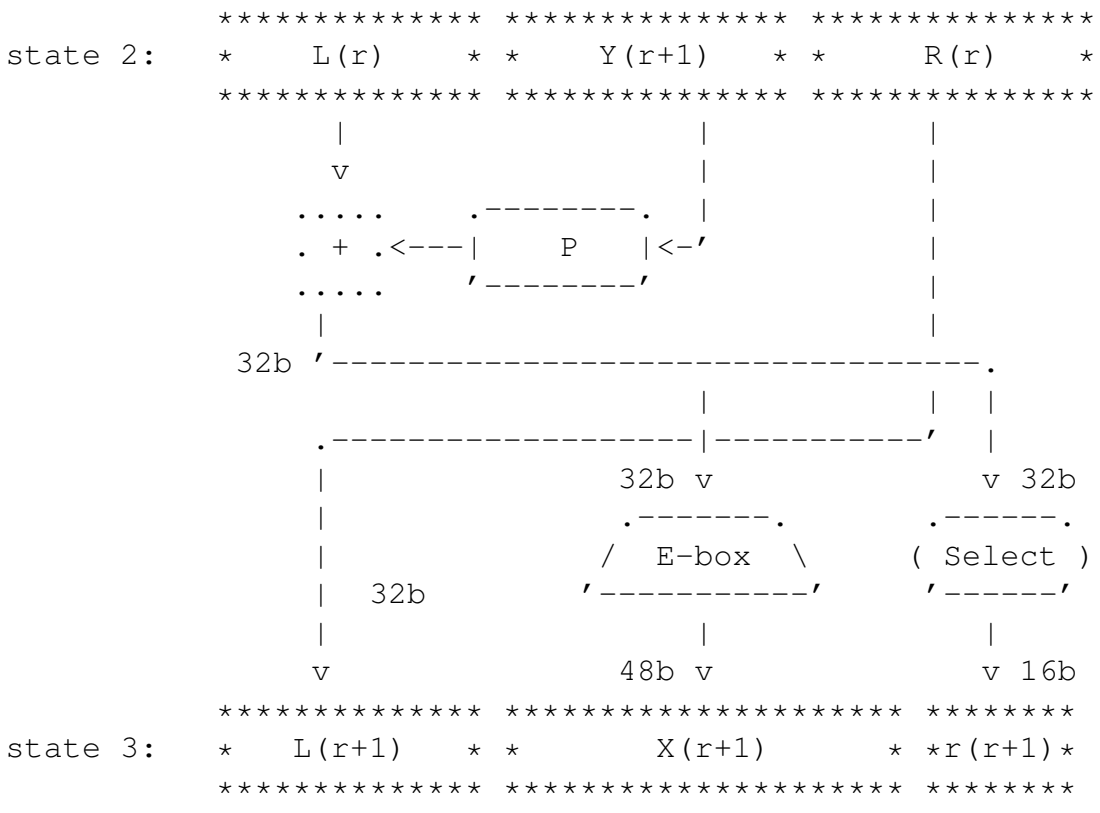
   Après permutation, les lookup tables linéaires prendront en paramètre 4 bits de la partie gauche de L et 4 bits de la partie des bits non étendus de R. Elles sont sous la forme :



   L'output de ces lookup tables est donc :

$$T_i^r = b_0b_1b_2b_3||b_4b_5b_6b_7$$

5.6.3 Étape 2 à étape 3



Comme pour l'étape précédente, nous voulons transformer les différentes opérations de cette étape en look up tables.
Pour ce faire, nous allons modéliser les différentes opérations en matrices puis la décomposer puisque la combinaison de la permutation et du XOR est une fonction affine.
Ces calculs vont être modélisés par M2 qui est une matrice (96,96) et qui réalise :

1. La permutation
2. Le xor avec la partie gauche
3. L'expansion pour la prochaine itération
4. Le placement de R(r) dans L(r+1)
5. L'extraction le vrai R(r).

5.6.4 Étape finale

L'étape finale va consister à ignorer les bypass (ie, ignorer r(r+1)), inverser la dernière expansion réalisée par M2, échanger L et R en cas de décryptage et faire la permutation finale.
Toutes ces étapes vont être réalisées grâce à une matrice M3 qui va être décomposée en sous-matrices et traduite en Lookup tables.

6 Conclusion

L'implémentation d'une whitebox demande une recherche et des efforts considérables. Il faut considérer les autres solutions disponibles (utilisation d'une API, d'une clé publique, ...) avant de commencer à entreprendre la construction d'une whitebox.

La taille que prend une whitebox est considérablement plus important que la taille d'un cipher nu. L'implémentation de Chow et al est par exemple de 4.5 MB. Une implémentation hardware est donc peu envisageable.

La sécurité des techniques publiées par Chow et cie provient essentiellement de l'encryption des entrées/sorties et de leur nombre. Mais cela ne suffit pas, aucune implémentation de whitebox dont les techniques et les algorithmes sont connus n'est sécurisée à ce jour.

Dans le cas où le lecteur serait intéressé par une solution professionnel du type whitebox, ces techniques seront utiles mais certainement non-suffisantes. D'autres sont surement utilisées mais non publiées afin de compliquer les attaques possibles.

7 Bibliographie

"A White-box DES Implementation for DRM Applications", Chow et cie.
http://crypto.stanford.edu/DRM2002/whitebox.pdf
"White-Box Cryptography" (PhD thesis), B. Wyseur.
https://www.cosic.esat.kuleuven.be/publications/thesis-152.pdf
"Practical cracking of white-box implementations", B. Wyseur.
http://www.phrack.org/issues.html?issue=68&id=8#article
"White-Box Cryptography" (Vidéo), Dmitry Khovratovich.
https://www.youtube.com/watch?v=om5AVTqB5bA
"On the (Im)possibility of Obfuscating Programs", Barak et cie.
http://www.iacr.org/archive/crypto2001/21390001.pdf
"Clarifying Obfuscation : Improving the Security of White-box Encoding", Hamilton E. Link and William D. Neumann.
http://eprint.iacr.org/2004/025.pdf