

Département d'Informatique

Groupe de Recherche En Complexité et Cryptographie

Implantation et optimisation des primitives cryptographiques

THÈSE

présentée et soutenue publiquement le 25 Octobre 2001

pour l'obtention du

Doctorat Paris 7

par

Thomas Pornin

Composition du jury

Rapporteurs : Antoine Joux (DCSSI)
Jean-Jacques Quisquater (UCL – Louvain-la-Neuve)

Examineurs : Guy Cousineau (LIAFA - Paris 7)
Henri Gilbert (France Telecom R&D)
Jacques Stern (ÉNS / Directeur de Thèse)
Serge Vaudenay (ÉPFL - Lausanne)
Jean Vuillemin (ÉNS)

Table des matières

1	Introduction à la cryptographie conventionnelle	1
1.1	Historique	1
1.2	Définitions	2
1.2.1	Fonctions de chiffrement symétrique	2
1.2.2	Complexité d'une attaque	4
1.2.3	Usage du chiffrement symétrique et sécurité	5
1.3	Description de DES	7
1.3.1	Historique de DES	7
1.3.2	Structure de DES	7
1.3.3	Sécurité de DES	10
1.4	L'attaque de Davies et Murphy	12
1.4.1	L'attaque originelle	12
1.4.2	L'attaque améliorée	17
1.4.3	Formalisation de l'attaque	18
1.5	Conclusion	24
2	Implantations classiques sur processeurs génériques	25
2.1	Capacités des processeurs polyvalents	25
2.1.1	Les processeurs 8 bits	26
2.1.2	L'âge d'or du CISC	28
2.1.3	L'avènement du RISC	31
2.1.4	La mémoire, poids mort de la vitesse	33
2.1.5	Résumé de la situation actuelle	34
2.2	Implantation de DES sur Pentium	35
2.2.1	État de l'art	35
2.2.2	Détail des capacités du Pentium	36
2.2.3	Représentation des permutations	37
2.2.4	La fonction de confusion	39
2.2.5	Performances	41
2.3	Algorithmes optimisés pour les processeurs génériques	42
2.3.1	RC4	42

2.3.2	DFC	44
2.4	Conclusion	49
3	<i>Bitslice</i>	51
3.1	Principes fondamentaux du <i>bitslice</i>	51
3.1.1	Du câble au registre	51
3.1.2	Conséquences opérationnelles	52
3.2	Méthodes d'implantation du <i>bitslice</i>	53
3.2.1	Orthogonalisation des données	53
3.2.2	Orthogonalisation des tables	57
3.3	Production automatique de code <i>bitslice</i>	61
3.3.1	Le langage de description d'algorithmes	61
3.3.2	Stratégie d'optimisation	63
3.3.3	Évolution future de bsc	65
3.4	Chiffrement symétrique d'un disque dur	66
3.4.1	Conditions opérationnelles	66
3.4.2	FBC	68
3.4.3	Implantation pratique de FBC	73
3.4.4	Sécurité de FBC	76
3.4.5	Vérification de l'intégrité d'un disque dur	77
3.5	Conclusion	79
4	FPGA	81
4.1	Présentation générale d'un FPGA	81
4.2	La carte Pamette	82
4.2.1	Vue d'ensemble	82
4.2.2	FGPA Xilinx	84
4.2.3	Programmation	86
4.3	DES-Cracker	87
4.3.1	Structure générale	88
4.3.2	DES sur carte Pamette	90
4.3.3	Mise en œuvre de la recherche	96
4.4	Prospectives	102
4.4.1	La loi de Moore	102
4.4.2	Gain pour les implantations matérielles	103
4.4.3	Le futur	104
5	Compromis matériel/logiciel	107
5.1	Présentation de A5/1	107
5.1.1	Historique de A5/1	108
5.1.2	Algorithme principal	108

5.1.3	Mise en œuvre de A5/1	110
5.1.4	Implantation de A5/1	111
5.2	Cryptanalyse logicielle de A5/1	112
5.2.1	Recherche exhaustive partielle	113
5.2.2	Compromis flux/temps/mémoire	115
5.3	Cryptanalyse matérielle de A5/1	116
5.3.1	Implantation de A5/1 sur FPGA	116
5.3.2	A5/1 et la recherche exhaustive	118
5.4	Compromis matériel/logiciel	120
5.5	Conclusion	123
6	Conclusion	125
	Bibliographie	127

Remerciements

Je tiens à exprimer mes plus vifs remerciements envers, tout d'abord, Jacques Stern, qui a su, par son encadrement efficace, diriger mes travaux et me forcer à la rigueur et la précision scientifiques nécessaires à l'accomplissement de cette thèse. Je lui adresse toute ma gratitude pour m'avoir permis de travailler dans d'aussi bonnes conditions, et la confiance en mes capacités qu'il a témoignée. Je remercie également les membres actuels et passés du Groupe de Recherche En Complexité et Cryptographie, qui m'ont conseillé, aidé et supporté pendant plusieurs années ; le GRECC possède une ambiance particulièrement adaptée à la recherche en cryptographie et au foisonnement des idées.

Je remercie particulièrement Serge Vaudenay, qui m'a fait découvrir la cryptographie et m'a initié aux algorithmes symétriques.

J'exprime toute ma gratitude à Jean Vuillemin, Mark Shand et Laurent Moll, qui m'ont permis de jouer avec leurs FPGA, m'ont patiemment expliqué comment m'en servir, et avec qui j'ai eu de longues discussions fructueuses.

Je remercie chaleureusement Éric Brunet, expert en $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$, et patient et méticuleux relecteur, tout comme Évelyne Prioux.

Je remercie mes rapporteurs, Antoine Joux et Jean-Jacques Quisquater, qui ont lu et commenté ma prose, ainsi que Guy Cousineau, Henri Gilbert, Serge Vaudenay et Jean Vuillemin, qui ont accepté de faire partie de mon jury.

Un grand merci à Joëlle et Valérie, qui déchargent quotidiennement les chercheurs des soucis administratifs, et au SPI, qui transforme les ordinateurs en outils utilisables.

Merci enfin à mes parents, sans qui ma vie ne serait pas vraiment ce qu'elle est actuellement, et à tous ceux que dans mon inconséquence j'ai malencontreusement oubliés.

Résumé

La cryptographie symétrique est l'étude des algorithmes de chiffrement de données, dont la clé de déchiffrement est identique à la clé de chiffrement. Ces algorithmes servent à rendre inintelligible une information, sauf pour les détenteurs d'une certaine convention secrète, afin de permettre son transfert sur des lignes de communication potentiellement espionnées, voire activement corrompues. Cette thèse s'intéresse au problème de l'implantation et de l'optimisation de ces algorithmes sur divers matériels informatiques, incluant les ordinateurs génériques et les FPGA (circuits reprogrammables). Les algorithmes classiques (DES, RC4, A5/1...) sont abordés, ainsi que des variantes de ces algorithmes afin de mieux s'adapter au matériel, sans sacrifier la sécurité; une nouvelle technique de programmation, dite « bitslice » ou « code orthogonal », est décrite, ainsi que les outils semi-automatiques de support de cette technique. Enfin, il est montré comment une utilisation adéquate de FPGA, en complément de stations de travail, permet d'accélérer grandement des travaux de cryptanalyse.

Abstract

Symmetric cryptography is the science of algorithms enciphering and deciphering data with the same secret key. Those algorithms are used to obfuscate data and make it unreadable to anyone but those who hold the key and can de-obfuscate the data and return it to its original layout; thus, they allow safe data transfers through unsafe and potentially hostile networks. This thesis is about the implementation and optimization of such algorithms on various devices, including generic purpose workstations and FPGAs (reprogrammable chips). The standard algorithms such as DES, RC4, A5/1 are studied, as well as other variations designed to match closely the intended supporting hardware, without sacrificing security. A new implementation technique, called "bitslice" or "orthogonal code", is described, along with an automatic tool to support such coding. Finally, this work explains how a fine-tuned mix between FPGA and generic workstations can greatly enhance cryptanalysis efficiency.

Chapitre 1

Introduction à la cryptographie conventionnelle

1.1 Historique

La cryptographie est, historiquement, l'art de cacher une information pour la rendre inintelligible à toute personne ne connaissant pas un certain secret. Les applications de la cryptographie remontent à la plus haute antiquité, et son usage est aussi ancien que celui de l'écriture ; on a retrouvé la trace de méthodes de chiffrement utilisées en Haute Égypte. Ces méthodes étaient pratiquées par l'intermédiaire d'appareils rudimentaires, voire par une simple opération de substitution mentale ; la sécurité du système était alors empiriquement définie par l'échec de ceux qui tentaient d'en percer les secrets.

Longtemps chasse gardée des militaires, la cryptographie commence à sortir de l'ombre vers la fin du XIX^e siècle. Mais la révolution principale en la matière a lieu à la fin des années trente, peu avant et pendant la seconde guerre mondiale. Les dispositifs cryptographiques sont alors de complexes assemblages d'engrenages et de relais électriques, dont le mécanisme est un secret jalousement surveillé. Afin de contrer les modèles allemands, les services secrets britanniques développent, en 1936, des méthodes d'analyse systématique, et surtout un appareil électronique permettant de mener à bien ces analyses ; cet appareil, premier vrai ordinateur jamais construit, est le Colossus[77], et il ouvre la porte à une nouvelle ère : désormais, la cryptographie est une science et devient une branche de l'informatique. L'histoire de ces temps héroïques du chiffrement, depuis l'usage de hiéroglyphes spéciaux jusqu'à la cryptanalyse de la machine allemande Enigma, est décrite en détail dans l'ouvrage majeur de David Kahn, *The codebreakers*[47].

L'après-guerre a vu le développement des algorithmes cryptographiques

modernes, à la sécurité quantifiée, analysée, partiellement prouvée, ainsi que le transfert définitif de la cryptologie dans la recherche publique. De nouveaux axes de recherche sont apparus, qui traitent de problèmes divers, tels que la réalisation d'une signature électronique, la vérification de l'intégrité d'un document informatique, l'authentification, *etc.*

Un changement des mentalités est également intervenu. Autrefois, quand Jules César assiégeait Avaricum¹ et devait envoyer ses ordres secrets à ses différentes légions, le plus important était la confidentialité des ordres ; la difficulté d'écriture et de lecture des ordres par l'émetteur et le destinataire légitime était secondaire. Le système utilisé était primitif (chaque lettre était décalée de trois rangs dans l'alphabet) mais efficace en ces temps-là. Désormais, la cryptographie est une aide à la réalisation de la sécurité informatique ; son rôle est d'empêcher les fraudes et piratages passifs ou actifs. En tant qu'outil de prévention, elle n'est pas le but du système informatique, mais plutôt un mal nécessaire. Les outils cryptographiques modernes se doivent d'être discrets, voire transparents. Leur efficacité informatique prime ; le défi posé aux cryptographes est de réaliser une sécurité satisfaisante et convaincante, en n'utilisant qu'une faible partie de la puissance du matériel informatique devant être protégé.

1.2 Définitions

1.2.1 Fonctions de chiffrement symétrique

Les systèmes de chiffrement modernes travaillent sur des données binaires ; on manipule toujours des suites de bits. Une *permutation indexée par une clé* est une fonction définie ainsi :

$$\begin{aligned} E : \mathcal{M} \times \mathcal{K} &\longrightarrow \mathcal{M} \\ (M, K) &\longmapsto E(M, K) \end{aligned} \tag{1.1}$$

où \mathcal{M} est un ensemble de mots binaires d'une certaine taille n qu'on appelle « messages », et \mathcal{K} est dénommé « espace des clés ». Cette fonction doit accepter une fonction inverse D définie comme suit :

$$\begin{aligned} D : \mathcal{M} \times \mathcal{K} &\longrightarrow \mathcal{M} \\ (M, K') &\longmapsto D(M, K') \end{aligned} \tag{1.2}$$

telle que pour toute clé K , il existe une clé K' telle que, pour tout élément M de \mathcal{M} , on ait :

$$D(E(M, K), K') = E(D(M, K'), K) = M \tag{1.3}$$

¹Bourges

Autrement dit, pour chaque clé K , E doit réaliser une permutation de l'ensemble \mathcal{M} des messages.

Un *système de chiffrement symétrique* est une permutation indexée par une clé telle que :

- l'application de cette permutation à un message est calculable aisément par un algorithme prenant en entrée le message et la clé K (dite de chiffrement) ;
- l'application de la permutation inverse est tout aussi calculable algorithmiquement, si la clé K' (de déchiffrement) est connue ;
- la clé K est identique à la clé K' ;
- sans la connaissance de la clé K , la fonction de chiffrement est computationnellement indistinguishable d'une permutation aléatoire.

Ce dernier point mérite quelques éclaircissements. On suppose le modèle suivant (utilisé, par exemple, dans [60]) :

- On dispose de quatre « boîtes noires », c'est-à-dire des appareils au fonctionnement opaque, prenant en entrée des messages M de l'espace \mathcal{M} des messages et donnant en sortie des messages du même espace.
- Les boîtes vont par paires et réalisent chacune des permutations de l'ensemble \mathcal{M} des messages ; chaque boîte d'une paire est la permutation inverse de l'autre boîte de la même paire.
- L'une des paires de boîtes implante un système de chiffrement symétrique utilisant une certaine clé K choisie aléatoirement et uniformément dans l'espace \mathcal{K} des clés ; l'autre paire calcule une permutation choisie aléatoirement et uniformément dans l'espace des permutations de \mathcal{M} .

Une personne, dénommée l'*attaquant*, doit tenter de reconnaître la paire de boîtes implantant la fonction de chiffrement de celle calculant la permutation choisie aléatoirement. L'attaquant connaît tous les détails de l'algorithme de chiffrement recherché, mais ignore tout de la clé. Il peut faire fonctionner les boîtes sur un grand nombre d'entrées, mais possède une puissance computationnelle bornée (aussi bien en temps qu'en mémoire). Le système de chiffrement est dit computationnellement indistinguishable d'une permutation aléatoire si l'attaquant ne peut pas deviner laquelle des paires de boîtes implante la fonction de chiffrement avec une probabilité substantiellement différente de $1/2$. Une procédure d'attaque permettant de briser cette indistinguishabilité est appelée un *distingueur*. Cette « procédure » est un algorithme utilisant éventuellement une source d'aléa externe.

1.2.2 Complexité d'une attaque

La complexité du travail de l'attaquant peut être mesurée suivant le travail calculatoire nécessaire, la nature et le nombre des accès aux boîtes noires, et la probabilité de succès. Les messages présentés à la boîte de chiffrement sont des *messages clairs* ; la sortie de cette boîte fournit des *messages chiffrés*. On notera que tout élément de l'ensemble \mathcal{M} des messages est à la fois un message clair et un message chiffré, suivant le contexte.

- La complexité en temps est mesurée en fonction du nombre d'instances de l'algorithme de chiffrement qui auraient pu être exécutées sur le même matériel dans le même temps. La complexité en mémoire est la taille en bits, octets ou toute autre unité, du stockage nécessaire à la réalisation de l'attaque.
- Si les requêtes de l'attaquant ne portent que sur des messages clairs ou chiffrés choisis aléatoirement et uniformément, l'attaque est dite à *clair connu*.
- Si les requêtes de l'attaquant sont des messages clairs choisis spécifiquement et envoyés à la boîte de chiffrement, l'attaque est dite à *clair choisi*.
- Si les requêtes de l'attaquant sont des messages chiffrés choisis spécifiquement et envoyés à la boîte de déchiffrement, l'attaque est dite à *chiffré choisi*.
- Si l'attaquant peut émettre toutes ses requêtes en un point de son calcul, recevoir ensuite toutes les réponses, puis terminer son attaque sans aucune autre requête, alors l'attaque est dite *non adaptative*.

La distinction entre les clairs choisis et les chiffrés choisis ne se fait pas dans le modèle explicité plus haut ; en revanche, cette distinction peut se faire dans le contexte d'une utilisation réelle d'un système de chiffrement.

L'attaque la plus générique contre un système de chiffrement symétrique est la *recherche exhaustive*. C'est une attaque à clair connu nécessitant très peu de requêtes à la boîte de chiffrement. Le principe est très simple : il s'agit d'essayer toutes les clés possibles, jusqu'à en trouver une qui « marche ». Pour contrer cette attaque, il faut que l'espace de clés soit suffisamment large pour que son parcours soit computationnellement infaisable. Des données plus précises sur ce qui est à la portée de la technologie actuelle seront données dans la section 4.4.

1.2.3 Usage du chiffrement symétrique et sécurité

Utilisation d'un algorithme de chiffrement

La principale utilisation du chiffrement symétrique est d'assurer la confidentialité d'un canal de communication. Deux entités, traditionnellement appelées « Alice » et « Bob » (et notées A et B), partagent un secret commun qu'elles utilisent sous la forme d'une clé de chiffrement. Alice chiffre les données qu'elle veut envoyer à Bob, qui peut les déchiffrer grâce à sa connaissance de la clé. Si un attaquant, Ève² (notée E), écoute la conversation, elle ne pourra en tirer aucune information sans la connaissance de la clé.

Un usage détourné est l'authentification : Alice veut prouver son identité à Bob ; pour cela, Bob lui envoie un message clair, et Alice doit retourner le message chiffré correspondant, suivant la clé secrète connue d'Alice et Bob seulement. Sans la connaissance de cette clé, une fausse Alice ne peut pas réaliser cette opération.

Si le chiffrement symétrique permet de réaliser la confidentialité et une certaine forme d'authentification, des besoins plus avancés tels que la signature électronique (qui permet d'assurer l'intégrité d'un message, ce qui contre les attaques actives, où Ève modifie les données transmises) demandent l'utilisation de primitives cryptographiques plus évoluées, où les clés de chiffrement et de déchiffrement sont non seulement différentes, mais aussi telles qu'il n'est pas computationnellement possible de calculer l'une en fonction de l'autre. Ces systèmes de chiffrement asymétriques utilisent des objets algébriques hautement structurés, et, s'ils sont fonctionnellement beaucoup plus riches que les fonctions de chiffrement symétriques, ils sont aussi beaucoup plus lourds en temps de calcul et moins souples d'utilisation dans des contextes à fortes contraintes (cartes à puce, canaux de communication lents, *etc*). Aussi, les protocoles modernes de sécurité mettent en œuvre des combinaisons d'algorithmes symétriques et asymétriques. Ces derniers sont souvent désignés sous le terme générique de « clé publique » et ne sont pas traités dans cette thèse.

Sécurité par transparence

On peut se demander pourquoi il est considéré que l'attaquant a accès à tous les détails de l'algorithme ; lui donner ce genre d'information ne peut en effet que l'aider. Il est cependant généralement considéré que conserver un système de chiffrement secret diminue la sécurité, pour les raisons suivantes :

- L'algorithme existe toujours sous une forme implantée, donc tangible

²De l'anglais *eavesdropper*, espion.

et potentiellement interceptable et analysable ; seule une clé peut rester à l'abri dans un cerveau humain. Il est donc sage de considérer que l'attaquant a, de toutes façons, accès aux détails de l'algorithme employé.

- L'analyse d'un algorithme de chiffrement est une opération très délicate ; publier les détails d'un algorithme, c'est s'assurer le concours de centaines de chercheurs en cryptographie de par le monde.
- La clé concentre le secret ; en cryptographie symétrique, la clé est quasiment toujours un mot binaire de taille fixée, et tous les mots de cette taille sont des clés valides. Ceci permet de chiffrer simplement le coût de la recherche exhaustive de la clé ; une telle recherche sur les algorithmes serait beaucoup plus difficile à estimer.

Le fait de conserver les détails de l'algorithme secrets est appelé habituellement « sécurité par obscurité », et est considéré comme une très mauvaise approche du problème de la sécurité informatique en général.

Sécurité empirique et preuves

La sécurité des systèmes de chiffrement symétrique est encore globalement empirique. On peut distinguer plusieurs classes d'attaques et concevoir des algorithmes dont la structure peut être prouvée résistante à de telles attaques ; on le verra notamment dans la suite de ce chapitre, pour le cas de l'algorithme DES.

Mais, de façon générale, le seul critère de sécurité connu est d'avoir résisté à une analyse publique par la communauté de recherche en cryptographie pendant plusieurs années. Afin de se donner plus de confiance, on considère qu'un algorithme n'est sûr que s'il résiste même à des attaques irréalistes, dites « académiques ». En effet, dans la pratique, il n'est pas raisonnable de supposer que l'attaquant dispose d'une mémoire plus grande que le système solaire, ou peut faire chiffrer par sa cible des camions entiers de bandes magnétiques contenant des données soigneusement choisies par lui. Mais la résistance à de telles conditions est considérée comme une preuve de compétence de la part du concepteur de l'algorithme, ce qui permet non seulement d'avoir un système de sécurité inattaqué, mais surtout d'être préalablement assuré de ce fait.

En résumé, la sécurité d'un protocole cryptographique est une affaire de prévision de risques et de confiance.

1.3 Description de DES

1.3.1 Historique de DES

DES signifie *Data Encryption Standard*. Le développement de ce standard a été lancé en 1972; il s'agissait de proposer un algorithme efficace, implantable facilement sur du matériel à bas coût, validable, versatile, et suffisamment sûr pour être proposé comme algorithme « universel » de chiffrement pour l'industrie et l'administration américaines (sauf les communications classées secrètes, dont l'utilisation et le transfert sont beaucoup plus encadrés).

Le NBS (*National Bureau of Standards*, devenu ensuite le NIST, *National Institute of Standards and Technology*) a lancé deux appels d'offre, en 1973 et 1974; IBM a envoyé comme réponse au second l'algorithme Lucifer[80]. Le NBS, IBM et la NSA (*National Security Agency*) ont ensuite collaboré pour modifier l'algorithme (afin, officiellement, d'augmenter sa sécurité) et régler quelques détails juridiques (IBM avait un brevet sur Lucifer). L'algorithme obtenu a été publié en 1975, et la norme elle-même[29] a été rendue publique en 1977.

Le DES est aussi connu sous le nom de DEA (*Data Encryption Algorithm*), selon la dénomination de l'ANSI (*American National Standards Institute*)[27].

1.3.2 Structure de DES

La structure générale de DES est un schéma de Feistel, du nom du concepteur de l'algorithme Lucifer[34]. Ce schéma utilise un certain nombre de tours (16, dans le cas de DES) et permet de réaliser une permutation de l'espace des messages clairs dans l'espace des messages chiffrés, telle que la permutation inverse soit facilement implantable, et même, dans une large mesure, partage une partie de l'implantation de la première permutation.

On travaille sur des blocs de données binaires de largeur paire $2n$ (dans le cas de DES, $2n = 64$); l'entrée du tour i est découpée en deux moitiés de taille n , la partie gauche L_i et la partie droite R_i . Une fonction f_i , dite « de confusion », prenant une entrée de taille n et une sortie de même taille, est utilisée à chaque tour. Cette fonction n'a pas besoin d'être inversible, et elle est en général indexée par une clé. La sortie du tour i , qui est aussi l'entrée du tour $i + 1$, est définie ainsi :

$$L_{i+1} = R_i \tag{1.4}$$

$$R_{i+1} = L_i \oplus f_i(R_i) \tag{1.5}$$

Autrement dit, la moitié gauche est combinée par un OU EXCLUSIF avec le résultat de la fonction f_i appliquée à la moitié droite ; puis les deux moitiés sont échangées.

Il est facile de voir que ce tour est inversible ; l'inverse se calcule ainsi :

$$R_i = L_{i+1} \tag{1.6}$$

$$L_i = R_{i+1} \oplus f_i(L_{i+1}) \tag{1.7}$$

Dans le cas de DES, le texte clair subit d'abord une permutation fixe de ses bits (c'est la permutation IP), et le texte chiffré est égal au permuté par la permutation inverse IP^{-1} de la sortie du dernier tour après échange des deux moitiés de cette sortie ; cet échange supplémentaire (qui n'est pas spécifique à DES) permet de rendre le déchiffrement très similaire au chiffrement (seul l'ordre d'usage des fonctions de confusion f_i change). La figure 1.1 illustre un schéma de Feistel à quatre tours.

Les 16 fonctions de confusions de DES sont toutes identiques, à ceci près qu'elles utilisent chacune une clé de 48 bits, et que ces clés sont différentes pour les 16 tours. Ces 16 clés sont extraites de la clé maître (qui a une taille de 56 bits) suivant la méthode suivante : la clé maître est placée dans deux registres à décalage de 28 bits ; pour chaque tour, les deux registres sont décalés d'un ou deux bits (suivant le tour concerné), puis les 48 bits sont copiés depuis 48 des 56 bits formés par ces deux registres ; cette extraction est la même pour tous les tours. Ces registres sont adaptés à une implantation matérielle ; les implantations logicielles ont tendance à simplement définir par des tables la provenance exacte, dans la clé maître, de chaque bit de chaque sous-clé.

La fonction de confusion de DES est constituée des phases suivantes :

- l'entrée, sur 32 bits, est étendue à 48 bits (c'est l'expansion E) par duplication de certains bits ;
- cette valeur de 48 bits est combinée par OU EXCLUSIF avec la sous-clé K_i du tour ;
- les 48 bits résultants sont regroupés en 8 mots de 6 bits ;
- chacun des mots de 6 bits sert d'entrée à une table, nommée *boîte S*, dont la sortie est un mot de 4 bits ; il y a 8 boîtes S différentes ;
- les 8 sorties de 4 bits des boîtes S sont concaténées pour former un mot de 32 bits ;
- ces 32 bits sont permutés suivant la permutation fixe P .

Ce processus est illustré par la figure 1.2 ; seule la sous-clé K_i change dans les différents tours de DES.

Les boîtes S sont des tables spécifiées dans la norme de DES[29]. Leur sortie est équilibrée ; pour toute valeur donnée de leurs deux bits d'entrée

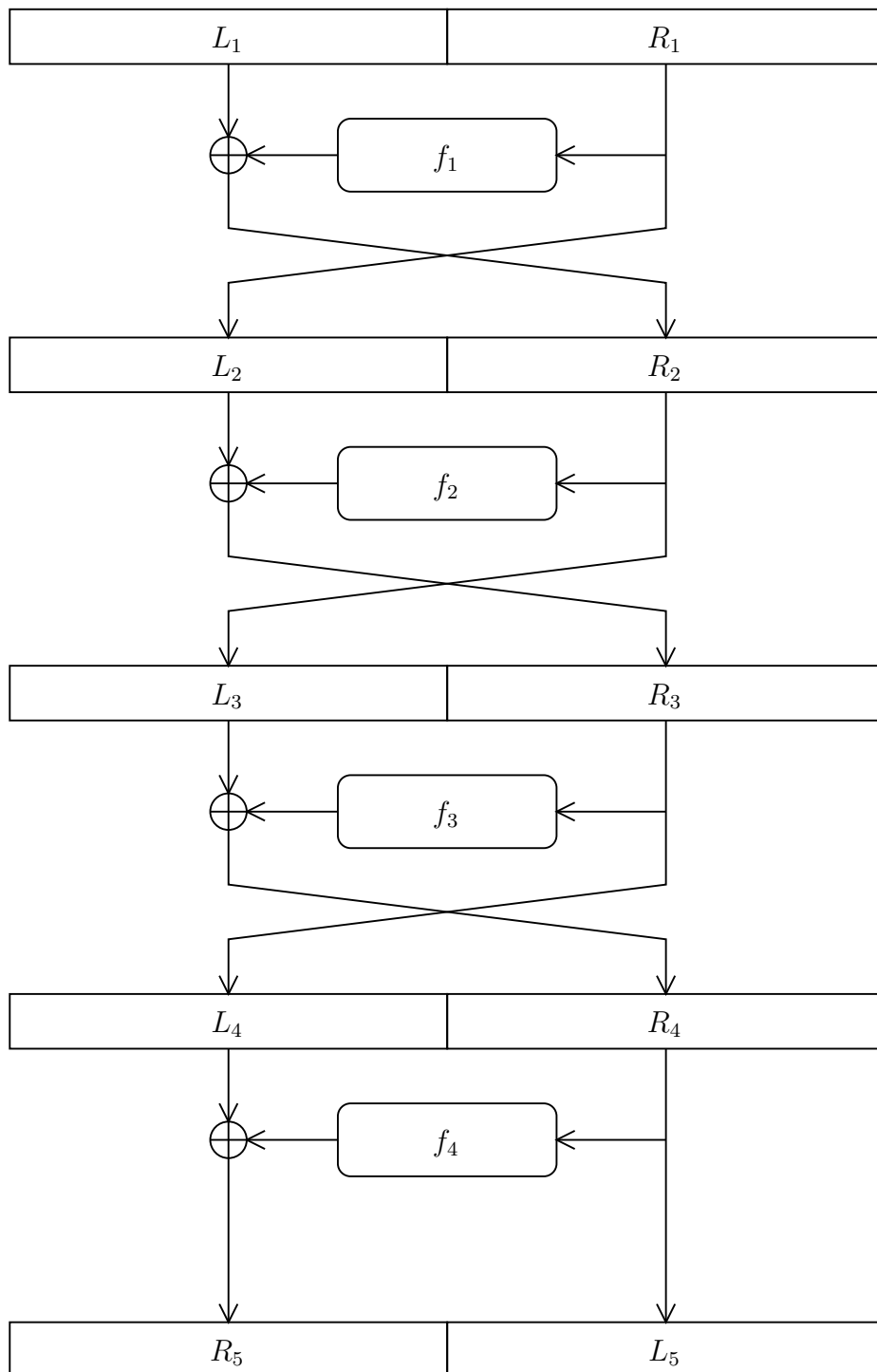


FIG. 1.1 : Un schéma de Feistel à quatre tours

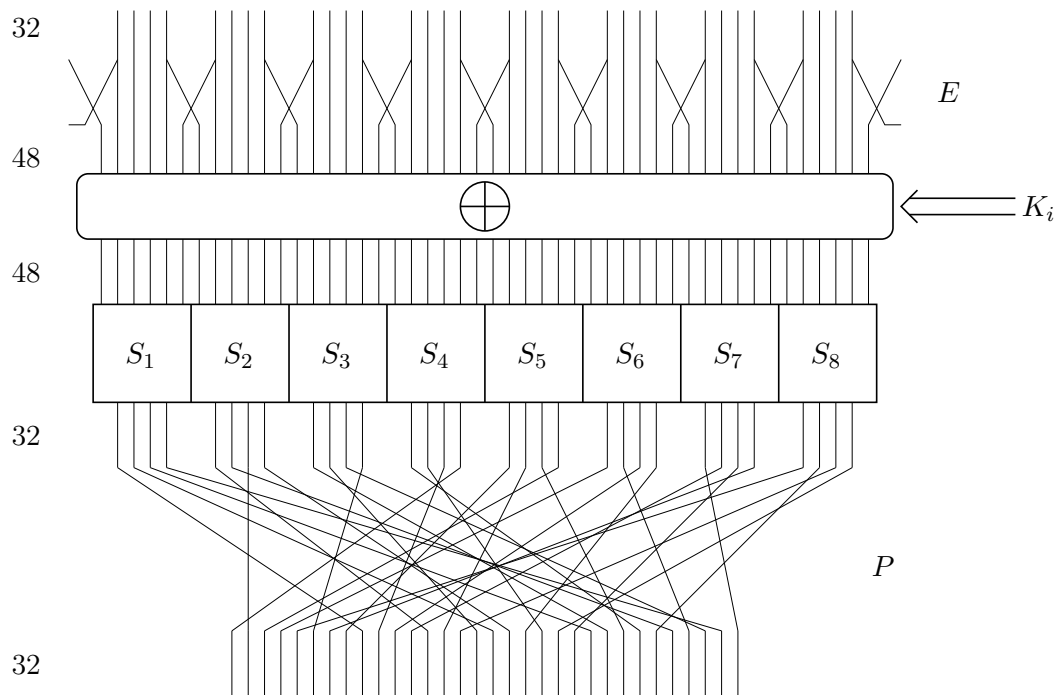


FIG. 1.2 : La fonction de confusion de DES

extrêmes, chaque boîte S se comporte comme une permutation de l'ensemble des mots binaires de 4 bits formant ses 4 entrées centrales.

Comme on le verra dans les chapitres suivants, DES est particulièrement adapté aux implantations matérielles, mais est peu efficace en logiciel. Les permutations initiale et finale (IP et IP^{-1}), par exemple, n'ont pas de justification cryptographique (étant fixes, elles sont aisément inversibles par tout attaquant) mais représentent exactement l'échange d'un mot de 64 bits entre un bus de données de largeur 8 bits, et huit registres à décalages de 8 bits ; autrement dit, elles permettent une implantation matérielle plus efficace, avec moins de latence et de silicium utilisé, dans ce genre de situation.

1.3.3 Sécurité de DES

La sécurité de DES est un sujet qui a été longuement étudié depuis la publication de sa norme ; le développement de DES ayant été confidentiel, il a longtemps été suggéré qu'il était possible qu'une faiblesse (une « *back-door* ») ait été introduite volontairement par la NSA, afin de permettre le déchiffrement rapide par cette institution de messages chiffrés en DES. Lors d'une présentation à Crypto'2000[20], Don Coppersmith, un des membres

de l'équipe de développement de DES, a assuré que la NSA n'a pas altéré le développement de DES, mais a plutôt constaté que l'équipe d'IBM avait redécouvert quelques méthodes d'attaques de cryptosystèmes dont la NSA avait jusqu'alors le secret. DES a été prévu pour contrer ces attaques, et non pour y céder.

Les seules attaques connues à ce jour sur DES sont :

- la recherche exhaustive de la clé ;
- la cryptanalyse différentielle ;
- la cryptanalyse linéaire ;
- l'attaque dite de Davies et Murphy.

La recherche exhaustive de la clé est rendue possible par la faible longueur de cette dernière : en effet, si la clé fait *stricto sensu* 64 bits, seuls 56 bits sont utilisés effectivement, les 8 autres étant ignorés et pouvant servir de contrôle de parité. Une propriété de complémentation de DES (si on inverse chaque bit du texte clair et de la clé, le chiffré obtenu est également l'inverse bit à bit du chiffré précédent) permet de diviser par deux, sous certaines conditions, le coût de cette attaque. Cette recherche exhaustive a été effectuée plusieurs fois, aussi bien en logiciel[31] que sur du matériel spécialisé (cf. section 4.3).

La *cryptanalyse différentielle* a été découverte en 1990 par Eli Biham et Adi Shamir[9, 13, 12] puis appliquée à de nombreux autres cryptosystèmes (par exemple Feal[10], Lucifer[4], des versions réduites de RC5[50], *etc.*). Des variantes de la cryptanalyse différentielle (différentielles impossibles[8], différentielles tronquées[49], attaques « boomerangs »[86]) ont aussi été proposées. Toutes ces méthodes reposent, schématiquement, sur l'idée d'analyser des paires de couples clair/chiffré, en « suivant » les différences binaires entre deux textes clairs, ou entre deux textes chiffrés, le long de l'algorithme. Dans le cas de DES, une attaque utilisant 2^{47} couples clair/chiffré choisis par l'attaquant permet de retrouver la clé. Le temps d'analyse est certes plus court qu'une recherche exhaustive de la clé, mais les conditions opérationnelles (l'attaquant doit faire chiffrer par sa victime 2^{47} mots de 64 bits qu'il a choisis, c'est-à-dire un million de giga-octets) sont telles que cette attaque est peu réaliste. Don Coppersmith a déclaré[19, 20] que l'équipe de développement de DES avait déjà connaissance de la cryptanalyse différentielle en 1975, et avait particulièrement renforcé les boîtes S contre cette attaque ; en tous cas, il a été constaté[12, 11] que de subtils changements des boîtes S rendaient quasiment toujours le DES plus fragile face à la cryptanalyse différentielle. On notera que le fait que les cryptanalyses différentielles soient souvent impossibles à tester ouvre la porte à des erreurs ; ainsi, une cryptanalyse de l'algorithme Skipjack[51] a été publiée en 1999, puis prouvée erronée par Louis Granboulan en 2001[41].

En 1993, Mitsuru Matsui a publié [62, 63] une nouvelle attaque nommée

cryptanalyse linéaire qui, dans le cas de DES, permet une attaque ne nécessitant la connaissance « que » de 2^{43} couples clair/chiffré, qui n'ont pas besoin d'être choisis spécifiquement par l'attaquant. Cela représente quelques 65 000 giga-octets de données, et autant pour leur version chiffrée, ce qui rend cette attaque guère plus réaliste que la précédente. Son principe est d'établir une équation linéaire entre certains bits du texte clair, du texte chiffré et de la clé, qui soit vraie avec une probabilité différente de 0,5 ; l'analyse de nombreux couples clair/chiffré permet de découvrir le OU EXCLUSIF des bits de clé concernés. La variante présentée dans [63] combine ces équations linéaires avec la fonction de confusion de DES et permet de retrouver 26 bits secrets d'un seul coup (les 30 autres étant trouvés par une recherche exhaustive en quelques heures sur une simple station de travail).

La cryptanalyse linéaire a été essayée sur d'autres algorithmes ; des liens entre les cryptanalyses linéaire et différentielle ont été exhibés[17], ce qui explique que DES résiste relativement bien à la cryptanalyse linéaire, puisqu'il a été consolidé contre la cryptanalyse différentielle. La résistance à une classe englobant ces deux attaques a été formalisée par Serge Vaudenay sous le nom de *théorie de la décorrélation*[83], puis plus particulièrement dans le cas des schémas de Feistel[84].

La structure même des schémas de Feistel a été largement étudiée dans un cadre plus général que DES et Lucifer ; il a été montré dans [60] qu'avec des fonctions de confusion « parfaites » suivant certains critères, ce schéma permettait de construire un cryptosystème symétrique prouvé sûr. Des variations sur le schéma de Feistel ont aussi été étudiées et utilisées ; par exemple, les fonctions de hachage MD5[65] et SHA-1[78] utilisent des schémas de Feistel étendus, où les données en cours sont découpées en respectivement quatre et cinq blocs.

L'attaque de Davies et Murphy sera étudiée en détail dans la section suivante.

1.4 L'attaque de Davies et Murphy

1.4.1 L'attaque originelle

L'attaque de Davies et Murphy a d'abord été présentée dans [26] puis améliorée par Eli Biham et Alex Biryukov dans [7]. L'attaque originelle utilise une propriété commune aux schémas de Feistel ; en utilisant les notations de la section 1.3.2, pour tout tour i (sauf le dernier), nous avons les deux

équations suivantes :

$$f_i(R_i) = L_i \oplus R_{i+1} \quad (1.8)$$

$$R_{i+1} = L_{i+2} \quad (1.9)$$

ce qui nous donne ceci :

$$f_i(R_i) = L_i \oplus L_{i+2} \quad (1.10)$$

Ceci est vrai pour tous les tours sauf le dernier. On peut effectuer le OU EXCLUSIF de ces équations pour les tours pairs, et pour les tours impairs ; après simplification, on obtient les deux équations suivantes :

$$R \oplus L' = \bigoplus_{i=1}^{r/2} f_{2i}(R_{2i}) \quad (1.11)$$

$$L \oplus R' = \bigoplus_{i=1}^{r/2} f_{2i-1}(R_{2i-1}) \quad (1.12)$$

pour le cas d'un schéma de Feistel à r tours, r pair, où (comme dans le cas de DES et de la figure 1.1) les deux moitiés de la sortie du dernier tour sont échangées. L et R sont les moitiés gauche et droite de l'entrée (c'est-à-dire que $L = L_1$ et $R = R_1$) et L' et R' sont les moitiés gauche et droite de la sortie ($L' = R_{r+1}$ et $R' = L_{r+1}$).

Autrement dit, chaque couple clair/chiffré nous donne accès au OU EXCLUSIF des sorties des fonctions de confusion des étages pairs, ainsi qu'à celui des sorties des fonctions de confusion des étages impairs. Si un motif non uniformément distribué apparaît dans la sortie des fonctions de confusion, cette propriété pourrait permettre de faire apparaître ce biais après analyse d'un grand nombre de couples clair/chiffré ; si ce biais est dépendant de la clé, une information sur cette dernière pourrait ainsi être obtenue. C'est le principe même de l'attaque de Davies et Murphy.

Considérons le cas de la fonction de confusion du tour i de DES ; sa sortie est le permuté par P (qui est une permutation fixée dans la norme et identique pour tous les tours) des sorties des boîtes S . Si on considère deux boîtes S voisines : leurs deux sorties combinées forment un mot de 8 bits, dépendant de 12 bits d'entrée ; or, si 12 bits de K_i interviennent dans cette entrée, seuls 10 bits de R_i y ont une influence. La figure 1.3 illustre ce fait.

Deux bits d'entrée sont dupliqués, et chacun est combiné avec deux bits différents de la sous-clé K_i , c'est-à-dire deux bits de la clé maître de DES (puisque chaque bit de chaque sous-clé est en fait un bit de la clé maître). Suivant le OU EXCLUSIF des deux bits de clé combinés avec les deux instances d'un bit dupliqué, les deux entrées correspondantes de la paire de boîtes S

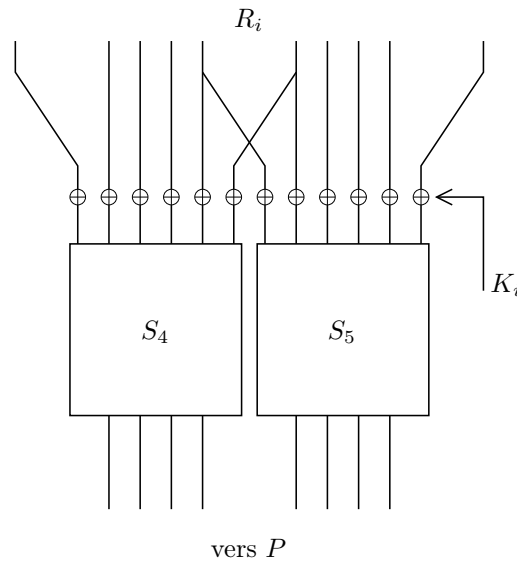


FIG. 1.3 : Deux boîtes S voisines de DES

seront donc toujours identiques, ou toujours différentes. Comme il y a deux bits dupliqués, ce sont deux OU EXCLUSIF de deux bits de clés qui déterminent quel ensemble de $2^{10} = 1024$ entrées possibles pour la paire de boîtes S sera effectivement parcouru sur un grand nombre de couples clair/chiffré.

Ainsi, potentiellement, la sortie de deux boîtes S voisines présentera, en sortie de la fonction de confusion de chaque tour, un motif parmi quatre, ce motif étant dépendant de la clé. Du fait de la définition même des boîtes S et de l'égalité de ces boîtes et de la permutation P au cours de l'algorithme, il n'y a en fait que deux distributions possibles du OU EXCLUSIF de la sortie de deux boîtes S voisines sur plusieurs tours de DES ; le choix entre ces deux distributions est directement dépendant du OU EXCLUSIF de plusieurs bits de la clé. Des détails sur ce calcul peuvent être trouvés dans [26].

L'attaque est alors menée ainsi : l'attaquant collecte de nombreuses paires clair/chiffré et calcule, pour chacune, le OU EXCLUSIF de la sortie de deux boîtes S voisines sur les tours pairs, et aussi sur les tours impairs. Il établit, dans chaque cas, la distribution choisie grâce à la méthode statistique dite du « maximum de vraisemblance » ; synthétiquement, cette méthode revient à choisir la distribution qui a le plus de chances de donner le résultat mesuré. Comme on peut travailler indépendamment sur les tours pairs et sur les tours impairs, on obtient ainsi deux bits « indirects » de la clé (un bit indirect est une combinaison linéaire de plusieurs bits ; c'est une information aussi valable que la connaissance d'un vrai bit pour ce qui est de la recherche exhaustive du reste de la clé).

Mathématiquement, comme on travaille sur un motif de 8 bits, pouvant prendre 256 valeurs, on représente les deux distributions possibles comme deux vecteurs u et v de \mathbf{R}^{256} ; u_i , la i -ème coordonnée de u ($0 \leq i \leq 255$) est la probabilité d'obtenir la sortie i . Trivialement, ces coordonnées sont toutes des nombres réels positifs dont la somme vaut 1 pour chaque vecteur. On définit aussi les vecteurs u' et v' qui sont les déviations de u et v par rapport aux distributions uniformes, c'est-à-dire que :

$$\forall i \in \{0..255\}, u_i = \frac{1}{256} + u'_i \quad (1.13)$$

$$v_i = \frac{1}{256} + v'_i \quad (1.14)$$

Nous avons donc :

$$\sum_{i=0}^{255} u'_i = 0 \quad (1.15)$$

$$\sum_{i=0}^{255} v'_i = 0 \quad (1.16)$$

Par ailleurs, du fait de la définition des boîtes S, on a la propriété suivante (qui n'est pas générique aux schémas de Feistel) :

$$u' + v' = 0 \quad (1.17)$$

Supposons que nous possédons M paires clair/chiffré indépendantes ; parmi ces M paires, chaque valeur i du motif de 8 bits apparaît m_i fois. Si la distribution théorique est u , la probabilité d'un tel événement est proche de :

$$p_1 = \prod_{i=0}^{255} u_i^{m_i} \quad (1.18)$$

On définit de même p_2 dans le cas de v . Comparer p_1 et p_2 est équivalent à comparer leurs logarithmes :

$$\log p_1 = \sum_{i=0}^{255} m_i \log u_i \quad (1.19)$$

On a donc :

$$\log p_1 = \sum_{i=0}^{255} m_i \log\left(\frac{1}{256} + u'_i\right) \quad (1.20)$$

d'où :

$$\log p_1 = \sum_{i=0}^{255} m_i \log(1 + 256u'_i) - M \log 256 \quad (1.21)$$

car la somme des m_i est M . Comme chaque u'_i est petit devant $1/256$ (car DES ne présente pas de biais statistique flagrant, et le biais recherché n'apparaît qu'après analyse d'un grand nombre de couples clair/chiffré), on peut remplacer les logarithmes par leur développement limité autour de 1, et obtenir les deux équations suivantes :

$$\log p_1 + M \log 256 \approx 256 \sum_{i=0}^{255} m_i u'_i \quad (1.22)$$

$$\log p_2 + M \log 256 \approx 256 \sum_{i=0}^{255} m_i v'_i \quad (1.23)$$

Nous ramenons donc l'attaque à la comparaison des deux valeurs suivantes :

$$s_1 = \sum_{i=0}^{255} m_i u'_i \quad (1.24)$$

$$s_2 = \sum_{i=0}^{255} m_i v'_i \quad (1.25)$$

Ces deux valeurs, s_1 et s_2 , sont le produit scalaire de m avec u' et celui de m avec v' . Nous pouvons majorer ces produits scalaires en utilisant la norme euclidienne sur \mathbf{R}^{256} . Si on note $N(x)$ la norme euclidienne de x , nous avons :

$$s_1 = m \cdot u' \leq N(m)N(u') \quad (1.26)$$

$$s_2 = m \cdot v' \leq N(m)N(v') \quad (1.27)$$

m est le résultat de l'analyse de paires clair/chiffré ; ce vecteur suit donc une distribution précise, mais peut varier autour de cette distribution. Chaque m_i compte le nombre d'occurrences de la valeur i pour le motif de huit bits ; cette valeur est proche de $M/256$ car la distribution de la valeur du motif est proche de la distribution uniforme (si cette distribution était très éloignée de la distribution uniforme, DES serait un fort mauvais cryptosystème). Donc la variance de m_i est proche de $(M/256)(255/256)$, que nous approchons par $M/256$.

Donc, la différence entre m et sa valeur théorique (valeur qui est M fois le vecteur de distribution) est un vecteur dont les coordonnées sont des valeurs de l'ordre de $(\sqrt{M})/16$; donc la norme de ce vecteur de déviation est en

moyenne \sqrt{M} . La norme de m , quant à elle, est proche de $M/16$. Pour conclure quoi que ce soit des paires clair/chiffré, la déviation recherchée (la différence entre s_1 et s_2) ne doit pas être plus petite que la déviation naturelle de m par rapport à sa distribution. Ainsi, M doit être suffisamment grand pour que :

$$N(m)(N(u') + N(v')) \geq \sqrt{M} \quad (1.28)$$

ce qui se réécrit ainsi :

$$M \geq \frac{256}{(N(u') + N(v'))^2} \quad (1.29)$$

Dans le cas de DES, cela nous conduit à une attaque utilisant au moins 2^{52} paires clair/chiffré, qui révèle potentiellement deux bits de clé. Ce résultat est obtenu avec les deux boîtes S 7 et 8 (les autres paires de boîtes S sont nettement moins favorables à cette attaque). Avec 2^{55} paires clair/chiffré ou plus, la probabilité de succès de cette attaque (c'est-à-dire, deviner correctement deux bits indirects de la clé) dépasse 50%. Il serait tentant d'utiliser les mêmes couples clair/chiffré pour attaquer d'autres paires de boîtes S voisines, mais les autres paires nécessitent une quantité bien plus importante de textes clairs connus.

Comme la recherche exhaustive de la clé nécessite seulement 2^{56} essais au pire (et 2^{55} en moyenne), cette attaque ne remet pas en cause la sécurité de DES.

1.4.2 L'attaque améliorée

En 1994, Eli Biham et Alex Biryukov ont présenté au congrès Eurocrypt[7] une amélioration de l'attaque de Davies et Murphy. L'idée est relativement simple : plutôt que d'attaquer 16 tours de DES, mieux vaut en attaquer seulement 14. Il s'avère que chaque couple clair/chiffré fournit l'entrée des fonctions de confusion du premier et du dernier tour ; l'entrée de deux boîtes S se calcule à partir de cette entrée en « devinant » 12 bits de clé.

L'attaque améliorée se déroule comme suit : $2^{12} = 4096$ attaques sont menées de front, chacune travaillant sur une combinaison possible des 12 bits de clé intervenant dans l'entrée des boîtes S 7 et 8 du premier tour. Chacune de ces attaques est une hypothèse sur la clé utilisée effectivement ; l'attaque sera plus probante sur l'instance de DES qui correspond à la clé effectivement utilisée. Ainsi, pour chaque instance, la déviation du motif de 8 bits est mesurée par rapport à une distribution uniforme ; l'instance qui fournit la meilleure déviation est considérée comme étant celle qui a travaillé sur les 12 vrais bits de clé. Cette attaque révèle le bit de parité (comme dans l'attaque de Davies et Murphy) et également 12 autres bits. Les mêmes couples

clair/chiffré peuvent servir pour l'attaque duale, où cette fois-ci ce sont les 12 bits de clé servant dans l'entrée du dernier tour qui sont devinés. Au total, avec 2^{52} paires clair/chiffré, cette attaque peut retrouver 26 bits de clé avec une probabilité de succès d'environ 53%.

On peut affiner cet algorithme en classant les candidats suivant l'importance de la déviation mesurée ; ceci augmente le coût de la recherche exhaustive finale nécessaire pour retrouver le reste de la clé, puisque plusieurs candidats devront être essayés. Si on considère que les couples clair/chiffré et les chiffrements devant être essayés par l'attaquant représentent la même complexité, alors le meilleur compromis est atteint avec 2^{50} paires clair/chiffré et un coût calculatoire de 2^{50} chiffrements DES, pour une probabilité de succès de l'attaque de 51%.

Cette attaque améliorée n'est pas non plus une menace sur la sécurité de DES, à cause du grand nombre de couples clair/chiffré nécessaires. Cependant, on peut conserver l'idée que le premier et le dernier tour de chiffrement peuvent être « éliminés », c'est-à-dire que la sécurité se mesurera sur deux tours de moins (donc 14 tours dans le cas de DES). La meilleure attaque connue à ce jour sur DES, à savoir la cryptanalyse linéaire de Matsui[62, 63], travaille également sur 14 tours, en « devinant » exhaustivement certaines parties de la clé sur le premier et le dernier tour.

1.4.3 Formalisation de l'attaque

L'attaque de Davies et Murphy peut se formaliser dans un cadre plus général ; on peut alors établir un critère de sécurité qui donne une résistance prouvée à cette attaque. Ce travail a été présenté au congrès Asiacrypt'98[70].

On considère un schéma de Feistel à r tours (on supposera r pair, par souci de simplification) ; la fonction de confusion du tour i est notée f_i , et n bits de la sortie des f_i forment un motif de 2^n valeurs possibles qui peuvent suivre q distributions différentes (quoique proches de la distribution uniforme) qui dépendent du matériel secret (la clé) utilisé au tour concerné. Chaque distribution sera représentée par un vecteur dans \mathbf{R}^{2^n} , comme précédemment.

Pour chaque couple clair/chiffré, nous pouvons calculer le OU EXCLUSIF de $r/2$ de ces motifs de n bits. Cette valeur a une certaine distribution qui dépend de plusieurs bits de clé ; nous pouvons calculer les différentes distributions possibles, et pour chacune l'information correspondante sur la clé. L'attaque consiste en l'utilisation de nombreux couples clair/chiffré afin de déterminer la distribution de ce OU EXCLUSIF et d'obtenir ainsi l'information correspondante sur la clé.

À chaque étage, il ne peut y avoir que q distributions possibles. L'opération OU EXCLUSIF étant commutative, seul compte au total le nombre de

chaque type de distributions. Il y a $r/2$ étages, donc le nombre de combinaisons possibles est :

$$\binom{r/2 + q - 1}{q - 1} \quad (1.30)$$

Dans la pratique, certaines de ces combinaisons peuvent être identiques ; par exemple, dans le cas de DES, il n'y a au final que deux distributions différentes.

Nous allons maintenant utiliser une nouvelle représentation des distributions : un vecteur distribution de \mathbf{R}^{2^n} peut être représenté par une fonction de \mathbf{Z}_2^n dans \mathbf{R} qui associe à un vecteur de n bits la coordonnée correspondante du vecteur distribution. Sur une telle fonction, on peut appliquer la transformée de Walsh-Hadamard-Fourier.

Transformée de Walsh-Hadamard-Fourier

La transformée de Walsh-Hadamard-Fourier[76] est en fait une représentation d'une fonction dans la base de Fourier, que l'on définit ainsi : c'est l'ensemble des fonctions v_y , pour y dans \mathbf{Z}_2^n , définies ainsi :

$$\begin{aligned} v_y : \mathbf{Z}_2^n &\longrightarrow \mathbf{R} \\ x &\longmapsto (-1)^{y \cdot x} \end{aligned} \quad (1.31)$$

où $y \cdot x$ est le produit scalaire de y et de x (c'est-à-dire le nombre de bits à 1 dans $y \wedge x$, où \wedge désigne le ET logique bit à bit).

Les coefficients de Fourier d'une fonction a de \mathbf{Z}_2^n dans \mathbf{R} sont calculés comme suit :

$$\hat{a}(y) = \sum_x a(x)v_y(x) \quad (1.32)$$

pour tout y dans \mathbf{Z}_2^n . On peut retrouver a à partir de \hat{a} en appliquant la transformation de Fourier inverse :

$$a(x) = 2^{-n} \sum_y \hat{a}(y)v_y(x) = 2^{-n} \hat{a}(x) \quad (1.33)$$

pour tout vecteur x de \mathbf{Z}_2^n .

Dans le formalisme de Fourier, le OU EXCLUSIF de la sortie de deux étages du cryptosystème devient une convolution des deux distributions ; en effet, si a représente la distribution d'un motif de n bits de la sortie de la fonction de confusion du premier étage, et b est la distribution du même motif de la sortie du second étage, alors la fonction c représentant la distribution du OU EXCLUSIF des deux motifs est telle que, pour tout x vecteur de n bits :

$$c(x) = \sum_{y \oplus z = x} a(y)b(z) \quad (1.34)$$

Mais, comme l'addition dans \mathbf{Z}_2^n n'est autre que le OU EXCLUSIF bit à bit, et comme pour tout vecteur binaire x on a $x \oplus x = 0$, on peut réécrire cette équation ainsi :

$$c(x) = \sum_y a(x-y)b(y) \quad (1.35)$$

Une convolution de deux fonctions se calcule aisément sur les représentations en base de Fourier : il suffit en effet de multiplier deux à deux les coefficients. Dans les notations précédentes, cela s'exprime de la façon suivante :

$$\hat{c}(x) = \hat{a}(x)\hat{b}(x) \quad (1.36)$$

Nous allons prouver un résultat similaire pour les déviations des distributions par rapport à la distribution uniforme. Pour une distribution représentée par la fonction a , nous notons a' la fonction telle que, pour tout x , on ait $a(x) = 2^{-n} + a'(x)$. Si c est le produit de convolution de a et b , alors c' est le produit de convolution de a' et b' . En effet, si on note d la fonction constante égale à 2^{-n} , ses coefficients de Fourier $\hat{d}(x)$ valent 1 si $x = 0$, et 0 sinon. Nous avons alors les équations suivantes :

$$\hat{a} = \hat{a}' + \hat{d} \quad (1.37)$$

$$\hat{b} = \hat{b}' + \hat{d} \quad (1.38)$$

$$\hat{c} = \hat{c}' + \hat{d} \quad (1.39)$$

$$\hat{c} = \hat{a}\hat{b} \quad (1.40)$$

De ces équations on déduit :

$$\hat{c}' + \hat{d} = \hat{a}'\hat{b}' + \hat{d}^2 + \hat{d}(\hat{a}' + \hat{b}') \quad (1.41)$$

Comme $\hat{d}(x)$ ne peut valoir que 0 ou 1, nous avons clairement $\hat{d}^2 = \hat{d}$; par ailleurs, nous avons $\hat{a}'(0) = \hat{b}'(0) = 0$, car pour toute fonction u , $\hat{u}(0)$ est égal à la somme des $u(x)$ pour tout x dans \mathbf{Z}_2^n , somme qui vaut 0 dans le cas de a' et b' . En intégrant ces résultats dans l'équation 1.41, on obtient :

$$\hat{c}' = \hat{a}'\hat{b}' \quad (1.42)$$

Résistance à l'attaque de Davies et Murphy

Afin de trouver une borne minimale à la complexité de l'attaque de Davies et Murphy, nous allons expliciter une borne maximale de la déviation du motif de la fonction de confusion f par rapport à la distribution uniforme. En effet, la distribution de sortie que nous pouvons mesurer à l'aide de couples

clair/chiffré n'est qu'une mesure de cette distribution, qui est d'autant plus fine que nous avons beaucoup d'exemples. Pour que l'attaque fonctionne, la déviation à mesurer ne doit pas être masquée par les fluctuations naturelles d'une mesure autour de son espérance.

Dans la suite, nous considérons M couples clair/chiffré, et le vecteur m dans \mathbf{Z}_2^n où m_i est le nombre de couples clair/chiffré tels que le OU EXCLUSIF des motifs de n bits des fonctions de confusion des $r/2$ étages pairs ait la valeur binaire i . Le vecteur m/M est une approximation de la distribution théorique pour une fonction de chiffrement « parfaite », c'est-à-dire indistinguible d'une permutation aléatoire de l'espace des messages clairs dans l'espace des messages chiffrés (nous utilisons ici la représentation d'une distribution par un vecteur dans \mathbf{R}^{2^n}).

En norme euclidienne, la déviation « naturelle » de m par rapport à son espérance est de l'ordre de \sqrt{M} (cf. section 1.4.1). On utilise à nouveau la méthode du maximum de vraisemblance ; si Y est un majorant de la déviation dans \mathbf{R}^{2^n} des distributions possibles du motif par rapport à la distribution uniforme, alors $2Y$ sera un majorant de la norme de la différence de deux distributions (par inégalité triangulaire), et on pourra dire qu'une attaque n'a de chances raisonnables de succès que si :

$$2Y \frac{M}{2^{n/2}} \geq \sqrt{M} \quad (1.43)$$

ce qui peut se réécrire ainsi :

$$M \geq \frac{2^n}{4Y^2} \quad (1.44)$$

On notera que ce résultat utilise les mêmes approximations que celles utilisées dans la description de l'attaque originelle de Davies et Murphy (section 1.4.1) ; en particulier, n doit être suffisamment grand pour que 2^{-n} soit négligeable devant 1.

Nous pouvons obtenir ce majorant Y en fonction des coefficients de Fourier de la fonction de distribution du motif considéré. En effet, la norme euclidienne dans \mathbf{R}^{2^n} n'est autre que la norme L^2 dans l'espace des fonctions de \mathbf{Z}_2^n dans \mathbf{R} , et le produit scalaire de deux fonctions a et b est :

$$a \cdot b = \sum_x a(x)b(x) \quad (1.45)$$

La base de Fourier (v_y) est une base orthogonale selon ce produit scalaire, et tous les v_y ont $2^{n/2}$ comme norme L^2 ; les coefficients de Fourier sont les

produits scalaires de la fonction avec les vecteurs de cette base. Donc, si a' est une fonction de \mathbf{Z}_2^n dans \mathbf{R} , sa norme L^2 est :

$$N(a') = 2^{-n/2} N(\hat{a}') \quad (1.46)$$

On peut donc écrire :

$$N(a') \leq 2^{n/2} \max_x |\hat{a}'(x)| \quad (1.47)$$

ce qui nous donne notre majorant Y :

$$Y = 2^{n/2} \max_x |\hat{a}'(x)| \quad (1.48)$$

où a' est la déviation par rapport à la distribution uniforme du motif de n bits dans le OU EXCLUSIF des sorties de $r/2$ étages du chiffrement.

Les coefficients de Fourier de la distribution du OU EXCLUSIF de $r/2$ motifs s'obtiennent en multipliant terme à terme les coefficients de Fourier des distributions des $r/2$ motifs, donc, si μ est le plus grand (en valeur absolue) des coefficients de Fourier de la distribution du motif en sortie d'un étage, le plus grand des coefficients de Fourier de la distribution du motif sur $r/2$ étages sera $\mu^{r/2}$.

On obtient donc le critère de sécurité suivant :

- Calculer les coefficients de Fourier de la fonction représentant la déviation par rapport à la distribution uniforme de la distribution du motif choisi dans la sortie de la fonction de confusion de chaque étage. Dans le cas de DES, cela est effectué par énumération des entrées possibles de deux boîtes S voisines.
- Prendre le plus grand de ces coefficients en valeur absolue ; on le note μ .
- Le nombre de couples clair/chiffré nécessaires à l'attaque de Davies et Murphy sur le motif choisi est au moins égal à :

$$\frac{1}{4\mu^r} \quad (1.49)$$

La sécurité globale du schéma est donc ramenée à l'énumération des motifs pouvant présenter un biais par rapport à la distribution uniforme. Dans le cas de DES, en utilisant le même motif que Davies et Murphy (la sortie de deux boîtes S voisines), le critère donne une sécurité de l'ordre de $1,5 \times 2^{52}$, ce qui rejoint les conclusions de Davies et Murphy (c'est-à-dire que le critère donne une valeur pertinente).

Approximations et sécurité réelle

Au cours de l'analyse décrite précédemment, un certain nombre d'approximations ont été faites. La plus importante d'entre elles est la majoration de la différence entre deux distributions en mesurant la différence entre chaque distribution et la distribution uniforme, et en appliquant l'inégalité triangulaire. Dans le cas de DES, il n'y a que deux distributions possibles, symétriques par rapport à la distribution uniforme, aussi l'inégalité triangulaire devient, dans ce cas précis, une égalité. Pour ce qui est d'autres schémas, cette méthode peut amener à calculer un critère de sécurité plus faible que la sécurité réelle (ce qui revient à dire qu'il s'agit d'un critère de sécurité et non d'une attaque).

Les autres calculs sont également sujets à des approximations :

- Nous avons considéré que les 2^n coordonnées du vecteur m sont des variables indépendantes entre elles, ce qui n'est pas vraie, puisque leur somme est toujours M . Dans le cas de DES, on travaille sur des motifs de 8 bits, donc il y a 256 coordonnées ; aussi nous pouvons négliger cet effet.
- Les variables m_i ne suivent pas, en fait, une loi normale, mais une loi binômiale qui s'approche d'une loi normale quand M devient grand, par application du théorème central limite. Dans la pratique, cette distinction n'a pas lieu d'être dès que M est supérieur à quelques milliers.
- La déviation du motif mesuré par rapport à la distribution uniforme est supposée petite, ce qui est de toutes façons une caractéristique très souhaitable d'un système de chiffrement symétrique, qui est censé être indistinguable computationnellement d'une permutation aléatoire.

Dans la section 1.4.2, nous avons vu comment l'amélioration de l'attaque par Eli Biham et Alex Biryukov[7] permet une attaque plus rapide en traitant séparément le premier et le dernier tour. L'applicabilité d'une telle méthode dépend des détails de la fonction de confusion utilisée dans ces deux tours et est difficile à formaliser ; une étude de sécurité ne devrait donc appliquer le critère de résistance à l'attaque de Davies et Murphy que sur les tours intérieurs. Autrement dit, il faut rajouter deux tours « de sécurité ».

La recherche et l'énumération des motifs potentiellement « dangereux » reste un problème ouvert. Davies et Murphy ont également étudié dans leur article[26] les motifs offerts par trois boîtes S voisines, mais ces motifs ne donnent pas de meilleure attaque. En fait, si le triplet de boîtes S contient les boîtes 7 et 8, alors l'attaque résultante a la même efficacité que l'attaque sur les deux boîtes 7 et 8 seules ; sinon, elle nécessite plus de couples clair/chiffré qu'il n'est possible d'en calculer (DES travaille sur des mots de 64 bits et ne peut chiffrer que 2^{64} textes clairs différents).

1.5 Conclusion

Dans ce chapitre, nous avons étudié la sécurité du standard de chiffrement DES. Cet algorithme a été défini il y a plus de 25 ans et reste très employé de nos jours ; nous allons voir, dans les chapitres suivants, comment DES et d'autres cryptosystèmes s'adaptent aux évolutions de la technique, que ce soit pour leur implantation ou leur cryptanalyse, en logiciel sur des processeurs génériques, ou sur du matériel spécialisé.

Chapitre 2

Implantations classiques sur processeurs génériques

2.1 Capacités des processeurs polyvalents

Depuis l'invention en 1971 par Intel du premier micro-processeur, le 4004, l'informatique n'a cessé de se concentrer sur le modèle de l'ordinateur polyvalent : doté d'un processeur central sans capacité particulière, et de périphériques génériques tels qu'un écran ou un stockage de masse (bande magnétique, disque dur), il tranche avec les *mainframes*, énormes systèmes centralisés qui ont fait la gloire de la décennie précédente. Des systèmes tels que la série des fameux PDP de Digital géraient des dizaines de terminaux et hébergeaient autant de sessions interactives. Les processeurs étaient souvent épaulés par un support matériel évolué de cet environnement particulier, afin de conserver la réactivité de toutes ces interfaces.

A contrario, l'ordinateur « personnel », destiné à servir à un seul utilisateur à la fois, est construit autour d'un processeur unique, dans une architecture dépouillée ; tout est fait en logiciel. Le comble de ce centralisme sera atteint avec le ZX81 de Sinclair, dont le processeur central est requis pour la génération du signal vidéo envoyé à l'écran (ce qui l'amputait de la majorité de sa puissance de calcul). Ce modèle s'est tellement imposé sur le marché, depuis l'explosion de l'informatique « familiale » (au début des années 1980), que désormais les remplaçants des *mainframes*, devenus serveurs de calcul, disque, Web ou autres protocoles réseaux, sont des ordinateurs personnels flanqués d'un peu plus de mémoire et de disques durs plus gros et plus fiables.

Quant aux matériels embarqués, notamment les cartes à puces, ils sont essentiellement des répliques des micro-ordinateurs personnels qui se fabri-

quaient dix à quinze ans auparavant, dans un emballage rendu compact par le progrès technologique. Nous allons détailler les capacités et performances de ces matériels et nous verrons comment leur structure interne conditionne l'efficacité de certains algorithmes cryptographiques.

2.1.1 Les processeurs 8 bits

Si le 4004 traitait les informations par blocs de 4 bits, ses successeurs se sont rapidement adaptés à l'octet, qui est devenu l'unité standard de groupement des données ; en effet, l'octet peut prendre 256 valeurs différentes, ce qui permet de coder un jeu de caractères suffisamment étendu pour englober différentes variations typographiques, notamment les caractères accentués des différentes langues d'Europe occidentale. Les mots de 7 bits, suffisants pour le codage ASCII (qui ne comporte pas d'accent, mais l'anglais peut s'en passer), ont longtemps été utilisés pour la télématique, et laissent encore maintenant des traces durables (par exemple, quand on envoie un courrier électronique en français, si ce courrier doit traverser divers serveurs d'outre-atlantique, il peut y perdre ses accents) ; mais l'octet devient la norme. De façon assez amusante, le monde occidental s'éveille à la présence d'autres écritures, et le nouveau standard d'encodage des caractères, l'Unicode, travaille avec des mots de 16 bits ; comme l'octet est parti pour durer, le consortium Unicode et l'ISO ont mis au point un codage de taille variable compatible avec le codage ASCII, l'UTF-8. Nous pouvons tenir pour acquis que tous les processeurs génériques conçus dans les prochaines années sauront adresser des octets efficacement.

Les premiers micro-processeurs, et toute la lignée de ce qu'on appelle couramment les processeurs « 8 bits », accèdent à la mémoire octet par octet et travaillent sur des valeurs de même taille. Le cœur du chemin de données est un registre appelé accumulateur. Ce registre est une mémoire de la taille d'un octet ; les instructions élémentaires sont classables dans les catégories suivantes :

- accès mémoire,
- instructions arithmétiques et logiques,
- contrôle de flux.

Les opérations de contrôle de flux sont les branchements conditionnels et les opcodes d'appel de fonction (avec sauvegarde de l'adresse de retour). Les instructions arithmétiques sont les opérations mathématiques simples telles que l'addition, la soustraction, la négation (qui se fait couramment par complément à deux). La multiplication, opération plus complexe, n'est apparue que sur des variantes tardives des processeurs 8 bits, telles que le 8080 d'Intel, ou le 6809 de Motorola. Les instructions logiques sont les

opérations binaires sur les valeurs : OU logique bit à bit, décalages, *etc.*

La gestion des accès mémoire est la partie la plus complexe du processeur ; elle utilise un ou plusieurs registres d'adresse, sur lesquels les opérations arithmétiques et logiques ne sont pas possibles directement, mais qui sont manipulables par les modes d'adressage. Les modes d'adressage sont les moyens plus ou moins complexes d'exprimer l'adresse d'un octet comme variation sur le contenu des registres d'adresse et de certaines zones mémoires désignées par ces mêmes registres.

Prenons comme exemple un processeur 8 bits « avancé », le 6809. Ce processeur, dû à Motorola, a équipé la famille des MO5 et TO7, fer de lance du plan « Informatique pour tous » du milieu des années 1980. L'ancêtre direct du 6809, le 6805, équipe un grand nombre de cartes à puce bas coût. Le 6809 dispose des registres suivants :

- deux accumulateurs 8 bits, A et B. Ensemble, ils forment le registre D, de 16 bits.
- deux registres d'adresse génériques, X et Y, de 16 bits. Le 6809 peut adresser 65536 octets différents, soit 64 Ko.
- deux registres d'adresse spéciaux, U et S, de 16 bits, qui sont utilisés dans les opcodes de gestion de pile que le 6809 possède nativement.
- un registre de page, DP, de 8 bits ; c'est en fait un demi-registre d'adresse, qui emporte les 8 bits de poids fort de l'adresse ; cela permet de gérer les opcodes ne spécifiant que les 8 bits de poids faible, et, dans une certaine mesure, de fabriquer du code déplaçable en mémoire.
- un registre d'état, contenant notamment le bit de *carry*, c'est-à-dire la retenue de la dernière opération arithmétique effectuée.

Le 6809 sait effectuer des additions, soustractions et opérations logiques sur des mots de 8 et 16 bits (il est plus rapide quand il travaille sur des octets), ainsi qu'une multiplication des contenus des registres A et B ; mais sa force de calcul réside dans ses modes d'adressages. Le 6809 peut, par exemple, en une seule instruction gérée efficacement par le processeur, référencer une donnée de 16 bits dont l'adresse du premier octet est elle-même dans deux octets consécutifs de la mémoire, dont l'adresse est dans le registre X. Dans la même opération, le 6809 peut postincrémenter le registre X d'une ou deux unités ; cette opération favorise le traitement de blocs de données dans une boucle. Ceci retranscrit assez bien un constat effectué très tôt par les concepteurs de processeurs : dans un ordinateur, la majeure partie du temps est passée à déplacer de l'information, la trier, la ranger ; le calcul effectif tient un rôle relativement mineur. Les algorithmes cryptographiques sont un peu à part, car ils calculent beaucoup avec peu de données.

Les processeurs 8 bits sont typiquement cadencés jusque vers 4 MHz ; les instructions élémentaires s'exécutent en 4 cycles horloge, voire plus. Les

modèles les plus répandus sont les suivants :

- le 6805 et le 6809, déjà évoqués ;
- le 6502 de Motorola, réputé pour ne contenir que 4000 portes logiques ; il équipait l’Apple II, et son frère jumeau (le 6510) fit la gloire du Commodore 64, modèle d’ordinateur le plus vendu à ce jour ;
- le 8080 d’Intel ; le processeur 8086, qui équipa les premiers PC, a été conçu pour exécuter nativement les logiciels conçus pour le 8080, et même encore maintenant, ces logiciels peuvent (du moins théoriquement) être lancés sur les plus puissants des Pentium. La firme Zilog a produit un clone du 8080, le Z80, possédant quelques instructions supplémentaires, plus rapide et moins cher ; le Z80 était le pilier central des ZX Sinclair et des Amstrad CPC.

D’autres processeurs (tels le 80C51 de Phillips, et le ST506 de ST Microelectronics) sont similaires et en usage massif dans les applications embarquées (cartes à puces, magnétoscopes, électronique automobile, distributeurs de café, *etc*).

2.1.2 L’âge d’or du CISC

CISC signifie *Complex Instruction Set Code*. C’est l’aboutissement du concept mis en place dans les processeurs 8 bits : le jeu d’instructions connues nativement par le processeur est très complexe, avec des modes d’adressage alambiqués mettant en jeu plusieurs registres et plusieurs accès successifs à la mémoire. La palme en la matière revient au VAX, dont on dit que certaines instructions « font le café. »

Les deux processeurs CISC majeurs seront le 68000 de Motorola, et le 8086 d’Intel. Ces deux processeurs sont conçus respectivement en 1978 et 1980 et ont depuis rencontré un grand succès. Le 68000 sera le processeur central des Atari ST et Amiga, et surtout du premier MacIntosh (sorti en 1984). Le 68000 possède 8 registres de calcul et 8 registres d’adresse, et des modes d’adressages largement plus complexes que ceux déjà évolués de son ancêtre, le 6809. Plus que jamais, le travail du processeur est de faire du routage de données, et le calculateur d’adresses est plus gros et optimisé que l’unité arithmétique et logique.

Le 68000 connaîtra plusieurs descendants, du 68010 au 68060. Si le 68000 était cadencé à 8 MHz et utilisait encore au moins 4 cycles par instruction, le 68060 est capable d’exécuter en moyenne 1,5 instruction par cycle, et de tourner à près de 100 MHz, ce qui le place en concurrence directe des Pentium entre 100 et 150 MHz. Néanmoins, Apple, fabricant des MacIntosh, abandonnera la lignée vers le milieu des années 1990 et se tournera vers les PowerPC, processeurs RISC. Les familles des Atari ST et Amiga étant entretemps de-

venues quasiment objets de collections, entretenus avec amour par quelques passionnés nostalgiques (principalement en Allemagne), les dérivés du 68000 ont perdu leurs débouchés dans le marché des ordinateurs de bureau ; aussi le 68060 n'a pas eu de successeur. Les stations de travail utilisant des 680x0 (stations Sun3, Hewlett-Packard, NeXT) ne suffisaient plus à justifier les coûts de production. En revanche, diverses variations sur le 68000 semblent avoir un avenir brillant en tant que cartes à puce et contrôleurs de bus, et c'est un dérivé du 68000 qui équipe les assistants numériques PalmPilot de 3Com.

Un point majeur à constater quant au 68000, c'est que ses registres ont une taille de 32 bits, ce qui permet au processeur de gérer nativement jusqu'à 4 giga-octets de mémoire, ce qui, 22 ans après la création du processeur, reste un quantité jugée importante pour une machine de bureau. Physiquement, le 68000 pouvait être connecté à 16 méga-octets de RAM, mais le 68030 possède toutes les connexions nécessaires pour générer des adresses complètes sur 32 bits.

Le 8086 est une toute autre histoire. Il a été conçu par Intel en 1980 pour contrer le 68000 de Motorola ; comme à l'époque Intel avait du mal à gérer des registres de 32 bits, et que par ailleurs Intel avait compris que conserver la compatibilité ascendante avec le code du 8080 fournissait l'assurance de ne pas perdre un marché déjà durement conquis, Intel a conçu le 8086 avec des registres 16 bits, et l'adressage mémoire se fait par une combinaison de deux registres, l'un étant décalé de 4 bits, puis ajouté à l'autre. Ce mode d'adressage, dit segmenté, permet d'accéder à 1 Mo de mémoire, ce qui semblait énorme, à l'époque (*« 640 KB ought to be enough for everyone »*, dit Bill Gates en 1981), et donna beaucoup de fil à retordre aux programmeurs.

Le PC d'IBM, conçu autour d'un 8086 (et dans le seul but de jeter un peu de confusion sur le marché afin de gêner Apple et son Apple II, dit-on), s'imposa, contre toute attente, grâce à son développement libre (c'est-à-dire qu'IBM n'avait pas protégé par brevet le PC, et de nombreux clones furent développés, notamment en extrême-orient). Le 8086 connut plusieurs successeurs : tout d'abord le 80186, qui ne servit que très peu comme processeur central, mais qu'on rencontre fréquemment sur les contrôleurs de disques durs ; puis le 80286, qui dispose d'une vraie protection mémoire (permettant un multi-tâche sûr et fiable), et le 80386, qui dispose de vrais registres de 32 bits, qui permettent de s'affranchir des segments et d'adresser 4 Go de mémoire. Les successeurs du 80386 n'apporteront pas de grandes modifications au jeu d'instructions, mais accéléreront beaucoup le traitement et rajouteront des modules matériels. Un 80386 peut être cadencé à 33 MHz, et effectue une addition en 2 cycles ; mais une lecture en mémoire lui prend 4 cycles. Le 80486 peut être amené à 120 MHz et effectue les opérations

arithmétiques et logiques en 1 cycle ; par ailleurs, certains 80486 emportent un coprocesseur arithmétique, qui peut effectuer des calculs sur des nombres à virgule flottante de 80 bits.

Le successeur du 80486 s'appelle le Pentium[23] ; les Pentium tardifs contiennent une unité de calcul dite MMX, dont on reparlera. En théorie optimisée pour le calcul « multimédia » (c'est-à-dire les transformées de Fourier discrètes), l'unité MMX fournit des registres de 64 bits utilisables notamment pour accélérer les copies de blocs de mémoire. Le Pentium a connu des dérivés : le Pentium Pro, puis le Pentium II et le Pentium III, ainsi que des variantes légèrement bridées ou optimisées pour certains travaux (Coppermine, Celeron, Xeon...). Quelques firmes concurrentes (AMD, Cyrix, NexGen) ont produit des clones de Pentium, aux capacités équivalentes et connaissant les mêmes instructions.

Courant 2001, on annonce des Pentium à près de 2 GHz. L'architecture interne est super-scalaire : cela signifie que le Pentium est en fait constitué de plusieurs processeurs qui se partagent le flux d'instructions autant que possible (deux instructions successives utilisant des registres différents peuvent être exécutées simultanément) ; les Pentium modernes réordonnent les instructions afin d'optimiser ces exécutions parallèles. En moyenne, tant qu'on travaille peu sur la mémoire (qui est nettement plus lente que le processeur, désormais), on peut compter sur une moyenne de deux instructions par cycle ; le multiplieur entier peut faire une multiplication de deux entiers de 32 bits (résultat sur 64 bits, dans deux registres) en 4 cycles. Le Pentium conserve une structure archaïque (pour cause de compatibilité ascendante) et dispose de 7 registres généraux, dont seulement 4 sont vraiment utilisables pour les opérations arithmétiques complexes (multiplications, divisions). Ceci est à comparer aux 16 registres du 68000, vingt ans plus tôt.

Les processeurs CISC sont un havre de paix pour les programmeurs en assembleur. Au milieu des années 1980, la mémoire était un luxe encore cher, et un ordinateur disposait typiquement de quelques dizaines de kilo-octets de RAM. La programmation en des langages évolués (tels que le C) ayant tendance à produire un code plus gros que son équivalent en assembleur, les contraintes des systèmes opérationnels imposait souvent une programmation directe en assembleur. Les opérations de plus en plus compliquées effectuables nativement par le processeur permettaient au programmeur de produire un code plus compact, en un temps de développement plus court. En revanche, cette complexité accrue rendait le travail des compilateurs d'autant plus difficile, ce qui rend l'usage de l'assembleur quasiment indispensable pour les routines les plus critiques.

Du point de vue du constructeur du processeur, le CISC est un casse-tête infernal ; l'immensité du dialecte à comprendre l'oblige à produire un empile-

ment de plusieurs technologies, notamment ce qu'on appelle du micro-code : un code machine travaillant sur des instructions plus simples, et interprétant les instructions complexes lors de l'exécution du programme. Une telle stratification fait monter les coûts de production car elle exige une grande quantité de silicium pour tourner efficacement. Seule la pression de l'immense marché du PC a justifié les coûts de développement des derniers Pentium, dont chaque modèle doit être vendu à plusieurs millions d'exemplaires pour atteindre son seuil de rentabilité.

2.1.3 L'avènement du RISC

Le RISC, pour *Reduced Instruction Set Code*, vient d'un constat effectué par IBM à la fin des années 1980. Ce constat est la règle des « 80/20 », c'est-à-dire que les compilateurs, dans 80% des cas, ne se servent que de 20% des instructions disponibles. En parallèle, la mémoire est devenue moins chère, et les méthodes de pagination et de mémoire virtuelle permettent de travailler sans trop de surcoût avec des programmes binaires de grande taille. D'où l'idée de concevoir des processeurs connaissant peu d'instructions, mais les exécutant rapidement. Le codage des instructions en mémoire doit être le plus plat possible, afin de rendre son interprétation possible directement par le matériel (et donc éliminer le micro-code). La faible quantité de silicium nécessaire réduit les émissions de chaleur et permet de cadencer les processeurs à des vitesses bien supérieures, ou de récupérer de la place pour intégrer de la mémoire cache.

Dans un processeur RISC typique, il y a beaucoup de registres (plusieurs dizaines), et toutes les opérations sont centrées sur ces registres. Le calculateur d'adresses, si important dans les processeurs CISC, disparaît complètement ! Il revient désormais à l'unité arithmétique et logique d'effectuer tous les calculs nécessaires ; l'accès à la mémoire se fait en déréférençant le contenu d'un registre, et c'est tout. Il n'y a plus de différence entre des registres de calcul et des registres d'adresse. Chaque instruction référence jusqu'à trois registres, deux opérandes et un registre accueillant le résultat.

IBM conçoit dans cette nouvelle optique les processeurs Power[61] qui équipent ses stations RS6000. Un dérivé du Power, le PowerPC, conçu en collaboration avec Motorola, devient le fer de lance de la série des MacIntosh d'Apple ; les derniers de la série se nomment G4, G5. Ils tournent à 500 MHz et peuvent effectuer 3 instructions par cycle.

L'idée du RISC s'est répandue comme une trainée de poudre ; divers autres constructeurs en ont produit, sur le même principe. Le consortium MIPS a conçu les processeurs centraux des stations Silicon Graphics, et on retrouve des MIPS dans beaucoup d'applications semi-embarquées, tels que

des routeurs. Ce sont typiquement ces matériels qui se retrouvent en position, à l'heure actuelle, de devoir chiffrer des données réseau à haut débit. Hewlett-Packard a également fait évoluer ses stations de travail vers un nouveau processeur, le PA-RISC[18], qui serait un modèle mineur si Intel ne l'avait choisi comme modèle pour son futur processeur RISC, l'Itanium (ex-Merced)[24], destiné à remplacer la lignée des Pentium. Sun a adopté les processeurs Sparc[66] pour ses serveurs et stations de travail ; les Sparc ont évolué vers les UltraSPARC, qui travaillent en 64 bits[67].

Mais le RISC « ultime » a été dessiné, conçu et produit par Digital, qui, en 1991, lance le processeur Alpha[21]. Ce processeur comporte 64 registres de 64 bits, dont 32 sont des registres « flottants », c'est-à-dire qui contiennent des données destinées aux calculs en virgule flottante. Sur les données entières, l'Alpha sait effectuer toutes les opérations logiques et arithmétiques courantes, sauf la division, qui doit être émulée logiciellement. Les premiers Alpha peuvent effectuer deux instructions par cycle, tant qu'il n'y a pas de conflit de ressources (accès en lecture par la deuxième instruction à un registre modifié par la première, double utilisation du multiplieur, *etc*) ; les modèles modernes (21264, dit ev6) peuvent lancer jusqu'à 4 instructions par cycle, et sont actuellement vendus à des fréquences de 1 GHz. Le jeu d'instructions est étudié pour être complètement parallélisable ; par exemple, il n'y a pas de registre d'état, pas de bit de retenue unique qui serait un goulot d'étranglement dans le code. Si on veut accéder à cette information après une opération arithmétique, on fait une comparaison des deux opérands, avec résultat dans un troisième registre : ces deux opérands sont forcément disponibles rapidement, puisque les opérations arithmétiques et logiques ne s'effectuent que sur les registres.

Le RISC est le modèle majeur des processeurs modernes et devrait continuer à prévaloir dans le futur proche, au moins dans le domaine des processeurs centraux des ordinateurs polyvalents. Le dernier bastion du CISC est le Pentium, qui est déjà très « RISC-ifié », avec ses registres génériques équivalents, et la sous-optimisation progressive des instructions complexes (le 8086 disposait d'une instruction spécifique pour effectuer une copie de mémoire à mémoire, en incrémentant les index afin d'être répétée dans une boucle disposant de son opcode spécifique ; sur le Pentium, il est plus efficace d'effectuer le transfert par plusieurs instructions simples, en passant par un registre).

On notera quand même que la simplification des instructions, aussi bien dans leur fonction que dans leur représentation en mémoire, entraîne une augmentation non négligeable de la taille du code programme. Un doublement de cette taille, par rapport à l'équivalent CISC, est typique.

2.1.4 La mémoire, poids mort de la vitesse

Au cours des années 1970 et 1980, la mémoire était essentiellement synchrone ; dans les ordinateurs, elle tournait à la même vitesse que le processeur et pouvait fournir les données aussi vite que le processeur les demandait. Le processeur pouvait ainsi puiser ses instructions en temps réel sans ralentissement.

Ceci n'est plus vrai de nos jours. La technologie de la mémoire n'a pas progressé aussi rapidement que celle des processeurs, et des contraintes fortes (telles que la vitesse de la lumière) ont commencé à imposer la situation actuelle : sur une station moderne, la mémoire est cinq fois plus lente que le processeur. La bande passante elle-même est conservée en utilisant des bus de données très larges (les dernières stations Alpha ont des bus de données de 256 bits de large, ce qui est technologiquement difficile à synchroniser), mais la latence induite lors de l'accès à une donnée par le processeur est importante.

Aussi tous les processeurs modernes utilisent des techniques de mémoire cache. La mémoire cache est un miroir transparent de la mémoire réelle, qui contient le code et les données les plus récemment (ou fréquemment) accédés. Le processeur n'accède en fait à la mémoire que par sa mémoire cache, qui est remplie par lignes : quand le processeur accède à une donnée en mémoire, qui ne se trouve pas dans le cache, un bloc d'octets consécutifs (contenant celui demandé) est transféré depuis la mémoire vers le cache (sur un Pentium, c'est ainsi 32 octets successifs qui arrivent au processeur). Ce transfert de bloc est peu coûteux par rapport à l'accès lui-même (à cause de la latence importante), et s'adapte bien à l'usage habituel de la mémoire par les programmes classiques (lorsque le processeur accède à une donnée, il accèdera souvent aux données placées juste après dans la mémoire, peu de temps plus tard).

Les architectures modernes finissent par utiliser plusieurs caches successifs, de tailles croissantes. Le cache de niveau 1, le plus petit, est aussi celui accédé le plus rapidement par le processeur ; une donnée réclamée par le processeur est disponible au cycle d'horloge suivant. Il est intégré dans la même puce que le processeur, et sa taille rappelle les temps héroïques des ordinateurs 8 bits : typiquement entre 16 et 64 Ko. La plupart des processeurs modernes distinguent la mémoire contenant du code de celle contenant des données, car cela permet de mieux optimiser les motifs d'accès à la mémoire, donc de mieux faire précharger, par le processeur, le contenu du cache. Le cache de niveau 2 est maintenant également sur le processeur, mais parfois en une puce séparée, accolée au processeur central. Ce cache a une taille de quelques centaines de kilo-octets ; si le processeur fait un accès en dehors de

son cache de niveau 1, la latence induite est de l'ordre de 10 cycles d'horloge. Certaines architectures, telles que l'Alpha, gèrent un cache de niveau 3, positionné en un emplacement privilégié de la carte mère.

Cette cascade de mémoire cache rend plus complexes les montages SMP (*Symmetric Multi-Processing*, c'est-à-dire plusieurs processeurs identiques se partageant la mémoire); mais son effet pour l'implantation de systèmes de chiffrement symétrique est relativement simple et peut se résumer à l'assertion suivante : hors du cache de premier niveau, point de salut. Le code et les données doivent tenir dans ce cache, sinon les latences induites sont catastrophiques pour les performances.

2.1.5 Résumé de la situation actuelle

Les systèmes cryptographiques doivent donc être implantés dans les architectures suivantes :

- systèmes embarqués, tels que les cartes à puce, dont les capacités sont très similaires à celles des ordinateurs 8 bits du début des années 1980 ;
- architectures CISC super-scalaires 32-bits, typiquement Pentium ;
- architectures RISC, surtout 64 bits.

Les systèmes embarqués n'ont en général que peu de problèmes de vitesse quant au chiffrement symétrique, la vitesse de transmission des données avec l'extérieur étant prédominante; ceci dit, ces processeurs sont globalement inefficaces pour les opérations arithmétiques complexes, telles que les multiplications de nombre de 32 bits (ou plus). La taille de donnée fondamentale de ces processeurs est l'octet.

Les architectures CISC et RISC seront utilisées dans les mêmes missions, principalement du chiffrement de données à haut débit (réseau fast-ethernet ou ATM, données vidéo, chiffrement de disque dur). L'usage d'un processeur polyvalent au lieu d'un ASIC¹ spécialisé est justifié par des raisons de coûts : les processeurs vendus en grande série ont un coût de vente très bas comparativement à leur coût de développement. Dans d'autres cas, la fonction de chiffrement doit être rajoutée de façon plus ou moins transparente à des stations de travail et des serveurs utilisés pour d'autres travaux; dans ce cas-là, non seulement la sécurité doit être assurée via le processeur polyvalent absolument pas optimisé pour le chiffrement, mais de plus cette sécurité se doit d'être discrète et de consommer peu de ressources.

Les derniers processeurs CISC, et les RISC, ont des capacités légèrement différentes mais néanmoins proches. On peut compter, très synthétiquement, sur les caractéristiques suivantes :

¹ *Application Specific Integrated Circuit*, c'est-à-dire circuit intégré non polyvalent.

- cadence : 500 MHz à 1,5 GHz
- deux à quatre instructions arithmétiques et logiques par cycle
- multiplication entière en 4 à 8 cycles
- cache mémoire de 32 Ko pour le code, autant pour les données

La différence entre le RISC et le CISC peut se traduire ainsi, du point de vue du cryptographe :

- les RISC ont beaucoup plus de registres ;
- les CISC savent faire quelques opérations complexes plus rapidement : par exemple, les rotations de bits (les RISC ne connaissent en général que les décalages, une rotation demande alors deux décalages et un OU logique) ;
- le développement sur RISC se fait en un langage évolué (le C, souvent) alors que sur CISC, l'optimisation maximale ne s'obtient qu'en programmant directement en assembleur.

Les processeurs Pentium avancés disposent par ailleurs d'une unité MMX qui, bien que son usage soit normalement destiné à l'application de filtres numériques, peut effectuer des opérations logiques bit à bit entre des mots de 64 bits. Le Pentium gagne ici un morceau de processeur RISC, avec 8 registres « généraux ».

2.2 Implantation de DES sur Pentium

Je décris ici une implantation de DES optimisée en assembleur pour Pentium, que j'ai réalisée en 1996. Elle nécessiterait quelques adaptations pour tourner efficacement sur les modèles plus récents, mais elle montre déjà quels sont les choix technologiques possibles, et pourquoi DES n'est pas conçu pour être efficace sur des processeurs polyvalents.

2.2.1 État de l'art

Les implantations optimisées de DES en logiciel se répartissent en deux catégories : celles conçues pour la recherche exhaustive de clés à partir d'un couple clair/chiffré, et celles pensées pour obtenir le meilleur débit de chiffrement. La première catégorie est à présent dominée par les implantations utilisant la technique du *bitslice*, ce qui sera traité dans le chapitre 3.

La plupart des implantations du chiffrement DES en logiciel ne sont pas complètement optimisées ; ce sont en général des programmes en langage C qui s'acquittent de la charge d'un chiffrement en un millier de cycles d'horloge sur un Pentium. Ces performances sont plus que suffisantes pour la plupart des usages de DES, à savoir le chiffrement d'une connexion réseau lente.

Cependant, il existe une implantation optimisée en assembleur pour la lignée des processeurs Intel Pentium, intégrée par Eric Young dans la bibliothèque `SSLLeay`[90], qui implante le protocole de communication SSL, qui utilise entre autres DES afin d'assurer une connexion réseau chiffrée.

En 1996, lorsque j'ai écrit mon implantation de DES, qui tourne en environ 450 cycles, la meilleure implantation d'Eric Young en demandait environ 500 ; cependant, en janvier 1997, Svend Olaf Mikkelsen a produit une nouvelle implantation pour `SSLLeay`, exploitant les recoins les plus obscurs des règles d'optimisation de code sur Pentium, qui s'exécute en environ 360 cycles sur un Pentium. Cette implantation est la plus rapide disponible à ce jour sur ce type de processeur ; la mienne ne fut détentrice du record que pendant quelques mois. Néanmoins, les règles d'optimisation changeant à chaque version du processeur, cette implantation optimale est loin d'être la plus rapide sur les Pentium II et suivants ; ceci illustre bien l'intérêt de programmer en un langage d'un peu plus haut niveau (tel que le C) et de laisser le compilateur gérer ces problèmes d'optimisation.

Mon implantation est donc utile surtout en tant qu'illustration des choix imposés par la structure du cryptosystème et les primitives de programmation du processeur cible.

2.2.2 Détail des capacités du Pentium

Le Pentium est issu de la longue lignée des processeurs Intel commençant au 8086 (qui lui-même peut être vu comme une amélioration du 8080). Le fait qu'il porte un nom alphanumérique, et pas seulement un numéro, vient de la prise de conscience par Intel qu'un nombre ne constitue pas, au niveau international, une marque déposable ; aussi Intel ne pouvait rien faire contre les clones de 80386 et 80486 vendus sous ces dénominations par AMD et Cyrix. Pentium est une marque déposée d'Intel.

Le Pentium dispose de sept registres généraux, ou du moins quasiment utilisables comme tels, de 32 bits :

- `eax`, `ebx`, `ecx` et `edx`, sur lesquels toutes les opérations arithmétiques et logiques sont possibles ;
- `esi` et `edi`, registres d'adresse capables de presque toutes les opérations arithmétiques ;
- `ebp`, registre d'adresse spécialisé dans le rôle de pointeur de cadre de pile, permettant d'accéder rapidement aux arguments et aux variables locales à une fonction écrite en C ; `ebp` est capable de quelques opérations arithmétiques et logiques.

Les registres `eax` à `edx` sont quasiment interchangeables, sauf pour les multiplications et divisions entières, et certains opcodes de gestion de boucles

et d'accès à des tableaux ; ces opcodes sont rarement utilisés car ils n'ont pas été autant optimisés que les instructions simples (car ils n'étaient que peu utilisés par les compilateurs, donc ils sont désormais lents, donc les compilateurs ne s'en servent pas, *etc*). De plus, l'octet de poids faible de chacun de ces quatre registres est manipulable comme un registre 8 bits à part entière, sous les noms respectifs de `a1`, `b1`, `c1` et `d1`.

Le Pentium est capable, sous certaines conditions, d'effectuer deux instructions par cycle ; il dispose en effet de deux unités de décodage et d'exécution d'instructions (les mauvaises langues disent que c'est normal, le Pentium n'étant jamais que deux 80486 côte à côte dans une même boîte). Ces deux unités de décodage sont appelées le *U-pipe* et le *V-pipe*. Certaines instructions (les plus simples, incluant les additions et les opérations logiques) peuvent être effectuées dans les deux unités, certaines, plus complexes, ne peuvent être effectuées que dans une seule des deux (les décalages, par exemple, ne peuvent être lancés que dans le *U-pipe*). Les instructions les plus complexes (les multiplications, notamment) ne peuvent pas être exécutées simultanément à une autre instruction. Les détails sont expliqués dans les manuels d'Intel[22], qui sont hélas partiellement faux (Intel a beaucoup de mal à admettre publiquement l'existence de bugs dans ses processeurs) ; une autre source d'information est la documentation écrite par Agner Fog[35].

Le Pentium est un processeur obsolète ; il n'est plus vendu en tant que processeur central de station de travail. Ses successeurs sont le Pentium Pro (qui a été un demi-échec commercial, trop cher à produire, et peu optimisé pour le vieux code 16 bits dont le système d'exploitation phare de l'époque, Windows 95, faisait encore grand usage), le Pentium MMX, le Pentium II et le Pentium III. Seul le dernier de la série est encore disponible sur le marché, ainsi qu'une version légèrement réduite du Pentium II, le Celeron. On trouve aussi des versions dites « Xeon », qui sont des Pentium II et III avec des caches de niveau 2 plus gros et un support étendu du multi-processeur. Les Pentium modernes peuvent exécuter les instructions par groupes de (au maximum) trois, réordonnent les instructions et renomment les registres pour combler les manques d'optimisation du code produit par les compilateurs.

2.2.3 Représentation des permutations

On a vu (1.3) que DES comporte principalement trois fonctions de manipulations de données bit à bit :

- la permutation initiale IP (et son inverse IP^{-1}),
- l'expansion E ,
- la permutation P .

La permutation initiale ne doit être conservée que si la conformance au standard est primordiale ; en effet, en tant que permutation fixe et connue, elle n'apporte pas de sécurité particulière. Comme cette permutation est coûteuse en temps de calcul, il est conseillé de concevoir un protocole tel que cette permutation ne soit pas utilisée. Mon implantation gère cette permutation initiale par des tables décrivant où vont les bits, en les groupant par 8. Le code correspondant, en C, est le suivant :

```
L = IP11[0][ 1 & 0xffUL ]
  | IP11[1][ (1 >> 8) & 0xffUL ]
  | IP11[2][ (1 >> 16) & 0xffUL ]
  | IP11[3][ (1 >> 24) & 0xffUL ]
  | IPr1[0][ r & 0xffUL ]
  | IPr1[1][ (r >> 8) & 0xffUL ]
  | IPr1[2][ (r >> 16) & 0xffUL ]
  | IPr1[3][ (r >> 24) & 0xffUL ];
```

Ce code calcule la partie gauche (de 32 bits) du résultat de la permutation initiale sur le mot de 64 bits dont la partie gauche est 1 et la partie droite est *r*. Un code similaire calcule la partie droite du résultat. Au total, quatre tables sont utilisées (IP11 et consœurs) ; chacune de ces tables est découpée en quatre sous-tables. Par exemple, IPr1[3][167] est le mot de 32 bits représentant la partie gauche du résultat de la permutation *IP* appliquée au mot quasiment nul, sauf dans son cinquième octet (celui de poids fort de la partie droite, contenant les bits 33 à 40 dans la numérotation du standard de DES), qui contient la valeur 167 (c'est-à-dire 10100111 en binaire).

La taille totale de ces 16 sous-tables de 256 valeurs de 32 bits est donc $16 \times 256 \times 4 = 16384$ octets, soit 16 Ko. Cela occupe deux fois la taille du cache de données de niveau 1 d'un Pentium de 1996 et tient à peine dans celui d'un Pentium II moderne. Même ainsi, le calcul de la permutation initiale *IP* demande 16 accès mémoire, soit un minimum de 16 cycles, en supposant que les calculs d'index soient effectués en parallèle, en travaillant sur des registres. Ce coût est prohibitif quand on considère que cette permutation n'apporte pas de sécurité et, finalement, ne calcule rien. On notera néanmoins que ce code est générique, c'est-à-dire qu'il peut implanter n'importe quelle permutation des 64 bits d'entrée, et n'utilise pas la forme particulière de cette permutation.

L'expansion *E* est plus simple à gérer, car elle est plus régulière. Dans la boucle principale, on verra que cette fonction se résume à un décalage et un masquage, qui retourne les 6 bits consécutifs de l'entrée de la fonction de confusion concernés par la boîte en cours. Ces 6 bits sont combinés par un OU EXCLUSIF, et le résultat sert d'index pour accéder aux tables représentant

les boîtes S.

Quant à la permutation P , elle est intégrée directement aux boîtes S : chaque boîte est considérée comme une fonction qui renvoie une valeur sur 32 bits, qui correspond aux quatre bits de sortie de la boîte, déjà en place (après passage par P). Les résultats des huit boîtes sont donc combinés par addition des valeurs de 32 bits (un OU logique aurait aussi fait l'affaire).

2.2.4 La fonction de confusion

Il s'agit ici du cœur de l'implantation ; il a été programmé directement en assembleur.

Afin de profiter de ces exécutions simultanées, j'ai groupé les accès aux boîtes S par deux, chacun travaillant sur un jeu de registres. Les sept registres généraux dont dispose le Pentium sont utilisés. On notera que cela inclut le pointeur de cadre de pile, qui doit donc être sauvegardé et restauré plus tard, et auquel on hésite habituellement à toucher ; le fait de devoir s'en servir pour d'autres calculs montre la pénurie de registres dans l'architecture du Pentium (pénurie due à l'obligation de maintien de la compatibilité binaire avec le code existant, le *legacy code*).

Voici le code correspondant à l'accès aux boîtes numéro 3 et 4 :

```

xor     %b1, %a1
xor     %d1, %c1
shrl   $8, %ebx
andb   $63, %a1
shrl   $8, %edx
andb   $63, %c1
movl   S+768(,%eax,4), %ebp
movb   4(%esi), %a1
addl   %ebp, %edi
movl   S+512(,%ecx,4), %ebp
addl   %ebp, %edi
movb   (%esi), %c1

```

Il s'agit de code en syntaxe AT&T, telle qu'utilisée habituellement par les outils de développement tournant sous Linux sur PC (par opposition à la syntaxe Intel, qu'on rencontre dans les outils MS-Dos et Windows).

À l'entrée dans ce code, les registres `ebx` et `edx` contiennent deux copies (décalées) de l'entrée de la fonction de confusion ; chacune sert à la moitié des boîtes. Cette double copie évite des soucis quant à la gestion du recouvrement des entrées des boîtes S (l'expansion E utilise certains bits deux fois ; dans le code assembleur, les deux utilisations sont réparties sur les deux registres).

Les registres `a1` et `c1` contiennent les deux morceaux de 6 bits de la sous-clé du tour considéré (ce calcul des sous-clés et leur partage en blocs de 6 bits fait partie de la diversification des clés, qu'on ne refait que quand on change la clé). Le registre `edi` contient le résultat partiel (ici, la sortie combinée après la permutation P des boîtes S 5 à 8), et `esi` pointe vers une table contenant la sous-clé du tour courant de DES, découpée en huit blocs de 6 bits.

Les deux premières lignes sont le OU EXCLUSIF de l'entrée de la fonction avec la sous-clé. Les quatre lignes suivantes décalent `ebx` et `edx` de 8 bits, afin de les préparer pour les deux boîtes suivantes ; en même temps, les contenus de `a1` et `c1` sont réduits de 8 à 6 bits par masquage, car les boîtes S ne prennent que 6 bits en entrée. On pourrait étendre les tables définissant les boîtes S à 8 bits en entrée (les deux bits de poids fort étant ignorés), mais cela impliquerait un quadruplement de la taille de ces tables, ce qui poserait des problèmes de saturation du cache ; de plus, les deux masquages sont essentiellement « gratuits » parce que les décalages de `ebx` et `edx` ne s'effectuent que dans le *U-pipe*, laissant donc des emplacements libres pour des instructions simples.

La ligne suivante effectue l'accès à la boîte S numéro 4 en utilisant un mode d'adressage légèrement complexe :

```
movl    S+768(,%eax,4), %ebp
```

L'adresse est celle de la table `S` augmentée de 768, ce qui donne une constante calculée lors de la fabrication du programme binaire, et inscrite directement dans l'instruction assembleur, à laquelle est ajoutée le contenu de `eax` multiplié par 4 ; cette multiplication (effectuée nativement par le décodeur d'adresse) correspond au fait que les entrées de la table `S` ont une taille de 4 octets (on a vu que la permutation P était inscrite directement dans les boîtes S, donnant des boîtes S dont la sortie a une taille de 32 bits). L'instruction `movl` déréférence cette adresse et va copier le mot de 32 bits correspondant en mémoire dans le registre `ebp`.

La ligne suivante est un rechargement de `a1` avec le morceau de sous-clé correspondant à la boîte 2, dans le bloc de code qui suit celui reproduit ici. Ce rechargement a lieu à ce moment parce qu'il va s'effectuer en même temps que l'accès mémoire de la ligne précédente ; l'instruction suivante, qui rajoute le contenu de `ebp` à la sortie de la fonction, doit attendre la fin de l'accès mémoire dans la table `S`, donc ne peut pas être lancée immédiatement. Cet ajout est fait ensuite (par l'instruction `addl`, qui effectue une addition ; une instruction `orl` ou `xorl`, codant respectivement un OU et un OU EXCLUSIF, aurait fait l'affaire). En parallèle à cette addition, l'accès à la boîte S numéro 3 est fait, par un calcul d'adresse similaire à celui de la boîte S numéro 4, mais avec `ecx` comme registre d'index.

Enfin, on ajoute le résultat de la boîte S numéro 3 (`addl %ebp, %edi`), puis on charge `c1` avec la sous-clé correspondant à la boîte S numéro 1, pour le bloc d'instructions suivant.

L'ensemble de ces douze instructions est prévu pour s'exécuter dans des conditions optimales, c'est-à-dire que le Pentium sera capable de les exécuter en six cycles d'horloge. Il y a quatre blocs successifs pour calculer la fonction de confusion d'un tour de DES, ce qui correspond à 24 cycles d'horloge.

2.2.5 Performances

Le calcul d'un tour de DES occupe 24 cycles pour le calcul brut ; à cela il faut rajouter un peu « d'administration », ce qui monte la facture à 28 cycles par tour si les 16 tours sont « dépliés », c'est-à-dire écrits successivement dans le code ; on peut faire un code plus compact en introduisant une boucle et un compteur, mais on perd 3 cycles pour la gestion de ce compteur (qu'il faut sauver et rechercher en mémoire, puisque tous les registres sont utilisés), et comme la version dépliée tient en entier dans le cache de niveau 1 du processeur, on ne gagne de ce côté que si le chiffrement DES est lui-même intégré dans une boucle de taille importante.

Au total, hors permutations initiale et finale, et sans compter le temps de diversification de la clé, mon implantation de DES tourne en 450 cycles d'horloge dans les conditions optimales, sur un Pentium (même compte sur un Pentium MMX). Cela correspond à un débit de l'ordre de 3500 Ko par seconde sur un Pentium MMX cadencé à 200 MHz. Ceci est équivalent au débit des disques durs d'entrée de gamme utilisés dans les PC du temps où le Pentium MMX était le haut de gamme. Autrement dit, un PC chiffrant son disque dur de façon transparente avec cette implantation de DES y passe l'intégralité de sa puissance de calcul. Pire encore, quand l'algorithme utilisé est le Triple-DES, le processeur central ne peut plus chiffrer qu'au tiers du débit du disque dur. Le temps de déchiffrement est similaire.

Cette implantation traditionnelle de DES est donc trop lente, et pour les raisons suivantes :

- Certains des calculs, en particulier les permutations, sont effectués via des tables ; or, même dans le cache de niveau 1, la mémoire est une entité lente ; la différence de vitesse entre la mémoire et le processeur va en augmentant, ce qui aggrave ce constat sur les processeurs récents.
- Le OU EXCLUSIF entre la sous-clé et l'entrée de la fonction de confusion s'effectue 6 bits par 6 bits, ce qui est un sous-emploi caractérisé de l'unité arithmétique et logique. Il est tentant de calculer ce OU EXCLUSIF d'un seul coup, mais cela fait ressurgir l'expansion E , dont le coût devient important.

- Une bonne partie du temps de calcul est dépensée en routage de données : décalage de registres et masquage, additions pour recombinaison des sorties des boîtes S.

La seule opération adaptée au processeur est l'accès à la table représentant les boîtes S, et encore cette dernière utilise-t-elle un index dont seuls 6 bits sont vraiment utiles ; alors que le décodeur d'adresses peut gérer des index de 32 bits.

En résumé, bien que mon implantation de DES ne soit pas complètement optimale, elle illustre bien les problèmes liés à l'implantation de cet algorithme, prévu pour être câblé dans des puces spécialisées, sur un processeur générique plus prévu pour le traitement des chaînes de caractères que le calcul cryptographique.

2.3 Algorithmes optimisés pour les processeurs génériques

Alors que DES était spécifiquement étudié pour s'adapter aux implantations matérielles, certains algorithmes développés ultérieurement l'ont été dans l'idée que l'architecture majeure serait un processeur générique, conçu pour réaliser efficacement des tâches qui n'ont que peu de choses à voir avec la cryptographie, mais plutôt avec le calcul numérique et le traitement de chaînes de caractères. Ce modèle économique semble être dominant actuellement : le micro-ordinateur le plus courant est le PC, et ses périphériques perdent leur « intelligence » au fur et à mesure des développements ; d'où l'apparition et la généralisation des « Winmodems » (modems réduits à l'état d'interface sur la ligne de téléphone, la modulation du signal étant prise en charge par le processeur central), des disques IDE (qui n'ont qu'une faible partie des capacités de récupération sur erreur des disques SCSI), *etc.*

2.3.1 RC4

RC4 est un générateur pseudo-aléatoire de flux binaire ; il a été développé en 1987 par Ron Rivest, de RSA Data Security, Inc. Il n'a jamais été vraiment spécifié, si ce n'est par analyse d'un code binaire le calculant ; le résultat d'une telle analyse a été publié anonymement en 1994 sur des listes de diffusion par courrier électronique. Bien que cela n'ait pas été confirmé officiellement, il semblerait que le code reconstitué soit bien conforme à l'original, et interopérable avec lui. Ce RC4 reconstitué est parfois appelé ARC4 (comme *Alleged RC4*) et il semblerait que l'acronyme RC signifie *Ron's Code*. On peut trouver de nos jours des descriptions d'ARC4 dans un certain nombre de pages

Web[89].

RC4 est particulièrement adapté aux processeurs capables de manipuler des octets, donc les processeurs 8 bits, mais aussi les processeurs centraux de stations plus modernes, car la manipulation des octets est indispensable au traitement des chaînes de caractères, donc à la mise en place d'applications lourdes telles qu'un serveur web.

Description de RC4

RC4 manipule un tableau de 256 octets, de contenus tous différents (ce tableau représente une permutation de 256 éléments). Ce tableau est constamment modifié au cours du fonctionnement de RC4. La clé est une chaîne de bits de longueur arbitraire entre 1 et 2048 bits.

L'initialisation du tableau de fait ainsi :

- Le tableau T est rempli par la permutation identité (la case numéro i est remplie par i , pour i allant de 0 à 255).
- La clé est étendue à 2048 bits en en concaténant autant de copies que nécessaire ; elle devient ainsi une suite de 256 octets, qu'on note K .
- Ces 256 octets sont injectés dans le tableau par l'algorithme suivant :

1. $c \leftarrow 0$
2. Pour i de 0 à 255
3. $c \leftarrow K[i] + T[i] + c \pmod{256}$
4. échanger $T[i]$ et $T[c]$
5. Fin pour i

RC4 utilise deux indices, compris entre 0 et 255, initialisés à 0, qu'on note x et y . À chaque itération de l'algorithme, un octet noté b est produit ainsi :

1. $x \leftarrow x + 1 \pmod{256}$
2. $y \leftarrow T[x] + y \pmod{256}$
3. échanger $T[x]$ et $T[y]$
4. $b \leftarrow T[x] + T[y] \pmod{256}$

Ainsi, toutes les opérations de chargement de la clé, et de fonctionnement principal, sont des accès à un tableau indicé par des octets, et des additions sur 8 bits ; ces opérations sont natives sur les processeurs 8 bits, ainsi que sur certains processeurs génériques tels que le Pentium.

Implantation et performances de RC4

RC4 s'implante de façon extrêmement directe, à partir de sa spécification. Voici la boucle principale d'un code en langage C qui réalise RC4 :

```

x = trunc8(x + 1);
y = trunc8(y + T[x]);
t = T[x];
T[x] = T[y];
T[y] = t;
b = trunc8(T[x] + T[y]);

```

Les variables `x`, `y`, `b` et le tableau `T` sont des variables représentant les éléments du même nom dans la spécification ; `t` est une variable temporaire ; `trunc8` est une macro qui réduit son argument modulo 256 (ce qui revient à tronquer à 8 bits). Sur une station de travail courante, on déclare `x`, `y`, `b`, `t` et les élément de `T` comme étant de type `unsigned char`, qui est usuellement (bien que ce ne soit pas garanti par le standard du langage C) une entité entière de 8 bits ; aussi le compilateur constate que la macro `trunc8` a une influence rigoureusement nulle, et donc ne génère aucun code pour cette macro.

Une fois compilé, voici le temps de calcul de 100 Mo de sortie de RC4 sur différentes architectures, en secondes :

Architecture	Temps de calcul
Pentium III	4,58
Alpha 21164	10,63
Alpha 21164a	7,29

Le Pentium III est cadencé à 550 MHz ; les deux Alpha ont une fréquence d'horloge de 500 MHz. Sur des traitements courants, au sens large, ces trois stations sont de performances équivalentes ; la différence entre le 21164 et le 21164a est que le 21164a dispose des extensions BWX, qui sont des instructions spécifiques pour manipuler des octets un par un, alors que le jeu d'instruction Alpha ne manipule nativement que les données de 32 et 64 bits. Ces instructions supplémentaires permettent de meilleures performances pour tout ce qui tient du traitement des chaînes de caractères, l'exemple emblématique de cette fonction étant le service de pages Web.

On peut constater que RC4 tourne particulièrement bien sur PC, et ses performances sont plutôt mauvaises sur l'Alpha, qui a été conçu principalement pour le calcul numérique. On constate également le gain de performance (+46%) induit par l'utilisation des instructions BWX.

2.3.2 DFC

DFC[38] est un algorithme que développé conjointement par plusieurs membres du GRECC (Groupe de Recherche en Complexité et Cryptographie)

et de France Telecom R&D afin d'être proposé comme candidat à l'AES[69], c'est-à-dire le nouveau standard de chiffrement américain, appelé à remplacer le DES. Le NIST (*National Institute of Standards and Technology*) est l'agence américaine chargée de développer et normaliser les standards technologiques dans tous les domaines industriels, notamment les télécommunications informatiques, et donc les applications cryptographiques. Afin de remplacer le DES, que sa courte clé et surtout la taille réduite de ses blocs rendaient trop peu sûrs en regard des avancées technologiques et des quantités de données à sécuriser, le NIST a voulu définir un nouveau standard, l'AES (*Advanced Encryption Standard*), algorithme de chiffrement par bloc opérant sur des blocs de 128 bits et utilisant des clés de 128, 192 et 256 bits.

Le NIST avait été très critiqué pour son manque de transparence dans la définition du DES ; des rumeurs de trous de sécurité volontairement introduits par la NSA ont contribué à freiner la généralisation de DES. Aussi, le NIST a voulu, pour l'AES, montrer sa « bonne volonté » en procédant à un appel à candidatures publiques et en détaillant son processus de sélection. Quinze équipes de cryptographie dans le monde ont répondu à cet appel, dont le GRECC ; le candidat de ce dernier était DFC. L'algorithme finalement retenu est Rijndael[25], proposé par une équipe belge.

DFC est assez représentatif du développement moderne des algorithmes cryptographiques, dans le but d'être optimisé pour l'architecture cible du NIST, qui était à l'époque un Pentium Pro, c'est-à-dire un représentant moderne de la famille des PC.

Description de DFC

Je décris ici la structure de DFC ; sa spécification complète est en dehors du cadre de ce mémoire.

DFC (pour Decorrelated Fast Cipher) est un algorithme de chiffrement à la conception relativement classique, car c'est un schéma de Feistel à huit tours. Sa fonction de confusion utilise une opération dite « de décorrélation », suivant la terminologie introduite par Serge Vaudenay[83, 84]. La décorrélation est une théorie permettant de chiffrer la résistance aux cryptanalyses linéaires et différentielles.

DFC opérant sur des blocs de 128 bits, suivant ainsi le cahier des charges du NIST, la fonction de confusion du schéma de Feistel utilise des entrées et des sorties de 64 bits. Cette fonction comporte les deux étapes suivantes :

- l'opération de décorrélation, via un calcul algébrique ;
- une permutation (relativement classique) de l'espace des messages de 64 bits.

La première étape utilise deux sous-clés, a et b , de 64 bits. Si x est l'entrée,

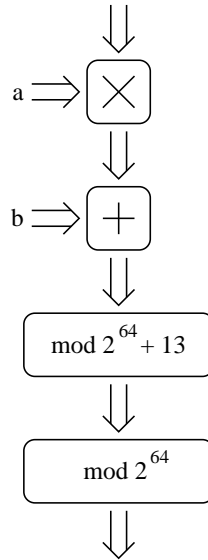


FIG. 2.1 : Étape de décorrélation de la fonction de confusion de DFC

alors la sortie de cette étape est :

$$f(x) = (ax + b) \pmod{(2^{64} + 13)} \pmod{2^{64}} \quad (2.1)$$

C'est-à-dire qu'une fonction affine est appliquée sur l'entrée ; les paramètres de cette fonction affine sont la clé, et l'opération s'effectue dans un corps fini ($2^{64} + 13$ est le plus petit nombre premier supérieur à 2^{64}). Le résultat est représenté numériquement sous sa forme canonique (entre 0 et $2^{64} + 12$, ce qui est un nombre de 65 bits au plus), puis réduit modulo 2^{64} (ce qui revient à « oublier » la valeur du bit de poids fort de la représentation sur 65 bits). Serge Vaudenay a montré comment cette opération apportait les caractéristiques voulues pour appliquer sa théorie de la décorrélation ; c'est cette théorie qui a nécessité l'emploi d'un module à la fois premier et supérieur à 2^{64} . La figure 2.1 montre cette étape.

La permutation appliquée sur la sortie de l'étape de décorrélation est plus simple, et ne fait appel qu'à des opérations natives et rapides des processeurs. L'entrée de 64 bits est découpée en deux mots de 32 bits ; 6 bits de la moitié de poids fort servent d'entrée à une table (nommée *RT*) qui comporte 64 entrées de 32 bits, et la sortie de cette table est combinée avec la moitié de poids faible par un OU EXCLUSIF bit à bit. La moitié de poids fort est elle-même combinée par un OU EXCLUSIF avec une constante (appelée *KC*) définie dans la spécification de DFC. Les deux moitiés sont ensuite recombinaées en échangeant leur places, et une constante de 64 bits (dénommée *KD*) est

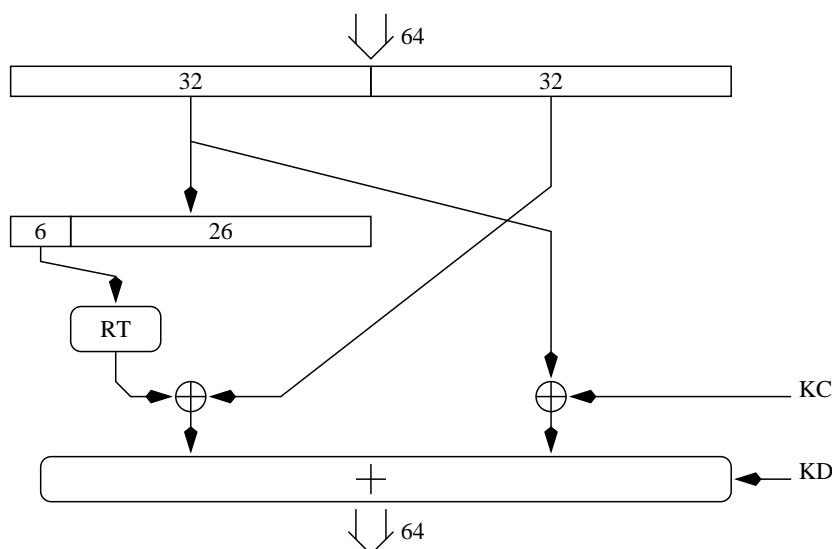


FIG. 2.2 : Permutation interne de la fonction de confusion de DFC

additionnée au mot de 64 bits recombinaé (cette addition est effectuée modulo 2^{64}). La figure 2.2 illustre cette permutation.

Les sous-clés sont calculées à partir de la clé maître en utilisant des DFC réduits : les deux sous-clés d'un tour sont le résultat du chiffrement par un DFC réduit à quatre tours des deux sous-clés du tour précédent ; ce DFC réduit utilise comme sous-clés des mots dérivés simplement (par découpage et décalage) de la clé maître. Le calcul de toutes les sous-clés nécessite le calcul de 32 tours de DFC, soit un coût quadruple d'un chiffrement.

Implantation et performances de DFC

Le cœur de DFC est constitué par le calcul de la fonction de décorrélation. Le reste de la fonction de confusion est simple, en termes d'opérations pour le processeur : un masquage avec décalage pour extraire 6 bits, un accès à une table, deux OU EXCLUSIF, une addition. Sur les processeurs 64 bits, il faudra compter quelques décalages de plus afin de découper et recombinaer les mots des 64 bits.

Fort heureusement, la réduction modulo $2^{64} + 13$ peut s'effectuer sans opération de division, opération coûteuse (on a vu notamment que le processeur Alpha ne dispose pas de cette opération, qui doit donc être réalisée logiciellement). Il est facile de voir que si a , x et b sont des quantités codées sur 64 bits, donc représentant des nombres compris entre 0 et $2^{64} - 1$, alors $ax + b$ est représentable sur 128 bits. On commence donc par calculer cette

valeur $P = ax + b$.

On découpe P en deux mots de 64 bits ainsi :

$$P = 2^{64}Q + R \quad (2.2)$$

puis on constate l'égalité suivante :

$$P = (2^{64} + 13)(Q - 13) + (13(2^{64} - 1 - Q) + 182 + R) \quad (2.3)$$

donc on obtient la relation suivante :

$$P = 13(2^{64} - 1 - Q) + 182 + R \pmod{2^{64} + 13} \quad (2.4)$$

$2^{64} - 1 - Q$ est en fait le « complément à un » de Q , c'est-à-dire l'inversion de tous ses bits; cette opération est native de la plupart des processeurs modernes (c'est l'opcode `not` sur PC, par exemple). On en revient donc à une multiplication par 13; cette multiplication peut s'effectuer grâce à des décalages et des additions, ou d'autres opcodes plus spécifiques : sur Alpha, les opcodes `s4add` et `s4sub`, qui réalisent les opérations respectives $4x + y$ et $4x - y$, et servent à accéder rapidement à des tableaux, permettent de calculer une multiplication par 13 en deux cycles.

On obtient donc une pré-réduction qui ramène le résultat sur 68 bits; le correctif nécessaire en fonction des 4 bits de poids fort peut être tabulé ou recalculé suivant une méthode similaire, suivant ce qui est le plus efficace sur l'architecture considérée.

Cette opération de décorrélation est spécialement prévue pour tourner efficacement sur processeur Alpha, et sur l'Itanium, successeur du Pentium. Ces deux processeurs manipulent nativement des données de 64 bits, et peuvent calculer rapidement une multiplication de deux telles valeurs, avec résultat sur 128 bits. Les processeurs UltraSparc (qui équipent les stations Sun modernes) ne sont pas aussi efficaces pour calculer DFC, car ils ne renvoient nativement que les 64 bits de poids faible du résultat d'une telle multiplication. Les PC ne savent multiplier que des valeurs sur 32 bits, mais le résultat est obtenu sur 64 bits; le jeu d'instruction du PC dispose par ailleurs d'opcodes spécialisés pour la propagation de retenues lors d'additions et de soustractions sur des nombres de plus de 32 bits.

Une fois optimisé, DFC peut effectuer un chiffrement en 304 cycles d'horloge sur un Alpha 21164a, 232 cycles sur un Alpha 21264, ou 240 cycles sur un Itanium (cette valeur est une estimation, effectuée par Terje Mathiesen sur la base des documentations fournies par Intel). Les implantations optimisées pour Alpha ont été effectuées par Robert Harley. Sur un processeur Pentium II, il faut 393 cycles pour calculer un chiffrement DFC. Ainsi, la

vitesse maximale de chiffrement avec DFC d'un processeur Alpha 21264 à 1 GHz, en supposant que les données à chiffrer sont fournies au processeur sans surcoût, est de l'ordre de 68 Mo/s, ce qui n'est pas suffisant pour suivre le rythme d'une carte réseau à 1 Gbit/s. Néanmoins, DFC est le candidat AES le plus rapide sur Alpha.

L'usage des multiplications rend très complexes les implantations matérielles de DFC.

2.4 Conclusion

Dans ce chapitre, nous avons vu que les possibilités des processeurs sont loin d'être identiques suivant leur modèle, et qu'un algorithme cryptographique ne sera pas conçu de la même façon suivant son emploi.

L'implantation efficace d'un algorithme de chiffrement reste une question de talent de programmation ; si, sur processeur RISC, les compilateurs C s'approchent fort bien de la production de code localement optimal, les méthodes de calcul mises en œuvre restent du domaine de l'intelligence humaine. La tendance actuelle est à la définition de l'algorithme en fonction des capacités du matériel qui sera utilisé pour l'implanter, plutôt que de concevoir l'algorithme de façon abstraite sur des critères purement cryptanalytiques ; et ces capacités sont dictées par des considérations commerciales mettant en jeu des intérêts dont la cryptographie n'est qu'une très faible portion.

L'AES est un effort vers la standardisation, afin de produire un algorithme sûr et efficace dans beaucoup de situations ; Rijndael a été choisi notamment parce qu'il pouvait être également rapide sur stations de travail et sur cartes à puces ; mais il reste limité par rapport à d'autres algorithmes plus spécialisés dans certaines utilisations. La standardisation absolue n'est donc pas forcément une méthode optimale de spécification d'éléments cryptographiques, du point de vue des performances.

Chapitre 3

Bitslice

3.1 Principes fondamentaux du *bitslice*

Nous avons vu dans le chapitre 2 comment étaient structurées habituellement les implantations de systèmes cryptographiques symétriques semblables au DES. Cette structure est raisonnablement efficace pour certaines opérations, par exemple la consultation d'une table ou la multiplication ; en revanche, des opérations simples mais importantes pour la diffusion de l'information et l'effet d'avalanche requis pour un bon cryptosystème, telles que les permutations fixes de bits, sont très coûteuses en temps de calcul.

Eli Biham a décrit en 1997[6] une méthode d'implantation permettant d'inverser ces difficultés, et donc particulièrement adaptée au cas de DES, qui fait grand usage de permutations de bits. Il lui a donné le nom de *bitslice* ; en réalité, cette technique était déjà connue, mais jamais vraiment documentée, dans d'autres domaines scientifiques (physique statistique, électronique...) sous le nom de *code orthogonal*.

3.1.1 Du câble au registre

L'implantation matérielle, sur une puce spécialisée (ASIC), d'un algorithme implique de traiter les bits manipulés un par un, et séparément les uns des autres. La fonction à réaliser est alors découpée en fonctions logiques élémentaires, jusqu'aux 16 opérations booléennes prenant deux bits en entrée et donnant un bit de sortie.

C'est cette représentation des données qui est utilisée dans le cas du code orthogonal : on sépare les différents bits, en utilisant une variable (conceptuellement, un registre du processeur) pour chaque bit de données traité. Les opérations logiques bit à bit sont implantées avec les primitives booléennes du processeur (les processeurs courants connaissent les OU, ET, NON et OU

EXCLUSIF ; certains possèdent quelques autres primitives). On reconstruit le calcul de la fonction telle qu'il serait effectué par un circuit matériel, en simulant logiciellement ce circuit.

Or, il s'avère que les algorithmes de chiffrement symétrique sont en général une suite d'instructions indépendantes des données utilisées (c'est une caractéristique souhaitable de toutes manières, afin de déjouer les attaques par chronométrage[55]). Donc, lorsque l'on calcule le chiffrement en utilisant les bits de rang 0 dans diverses variables, et les opérations logiques bit à bit du processeur, on calcule en fait la même opération de chiffrement, en utilisant les bits de rang 1, ainsi qu'avec les bits de rang 2, 3, *etc.* Ce n'est donc pas un chiffrement qui est ainsi calculé, mais autant de chiffrements en parallèle qu'il y a de bits dans les variables manipulées, c'est-à-dire de bits par registre du processeur. C'est ce parallélisme qui donne toute sa puissance à la méthode du *bitslice*.

3.1.2 Conséquences opérationnelles

Le *bitslice* a principalement les avantages suivants :

- Les registres, et l'unité arithmétique et logique du processeur, sont utilisés à leur plein potentiel. En effet, si on considère DES, la plus grande opération logique bit à bit est un OU EXCLUSIF sur deux mots de 48 bits ; sur un processeur 64-bits, un quart de chaque registre est alors ignoré si on utilise une implantation classique.
- Les permutations fixes de bits ne sont plus qu'un problème de routage de données, qui est résolu où il se doit, à savoir lors de la compilation et non lors de l'exécution : il s'agit simplement de prendre le « bon » registre.
- La conception générale de l'implantation s'adapte aux évolutions du matériel : si de futurs processeurs ont des registres d'encore plus grande taille, cela donnera plus d'exécutions parallèles de l'algorithme, et ces nouveaux registres seront toujours utilisés efficacement.
- Les méthodes d'optimisation du matériel peuvent être réutilisées pour le logiciel ; le coût d'une implantation *bitslice* est estimable *via* le nombre d'opérations booléennes nécessaires (car ces dernières sont, grosso-modo, évaluées séquentiellement), ce qui signifie que l'optimisation en vitesse d'exécution en logiciel est l'analogie de l'optimisation en nombre de portes, c'est-à-dire en surface de silicium, c'est-à-dire en coût de production, d'un ASIC.

En revanche, cette représentation des données a aussi un certain nombre d'inconvénients, donc voici un résumé :

- Les opérations arithmétiques complexes, telles que les additions et mul-

tiplications, deviennent très coûteuses : ce qui était une instruction élémentaire devient un assemblage complexe, similaire à ce que doivent gérer les constructeurs de processeurs ; les tables, telles que les boîtes-S de DES, deviennent complexes également.

- De grandes portions des processeurs, très optimisées et constituant une part important du coût du matériel, sont peu ou pas du tout utilisées. L'exemple-type est le multiplieur, qui n'a aucun usage intéressant dans une implantation *bitslice*.
- Le parallélisme intrinsèque des instances calculées du chiffrement restreint le domaine d'utilisation de la méthode : typiquement, le mode de chiffrement CBC est inemployable, car chaque bloc chiffré dépend du résultat du chiffrement précédent.
- Les données à traiter sont souvent disponibles sous la forme d'une suite d'octets dans la mémoire d'un ordinateur ; la répartition de ces données à un bit par registre est une opération coûteuse.
- Le code *bitslice* s'exprime de manière non-intuitive dans les langages de programmation courants, ce qui augmente les temps de développement.

On notera que les algorithmes de chiffrement asymétriques utilisent beaucoup de saut dépendants des données et ne sont donc pas raisonnablement optimisables par ce genre de technique.

La répartition des données entre les registres possède une solution plus rapide que l'algorithme naïf, qui sera détaillée dans la section 3.2.1. Diverses méthodes automatiques de traitement des tables afin de les calculer en *bitslice* seront présentées dans la section 3.2.2. Un nouveau langage de programmation, et son compilateur associé, permettant de programmer rapidement un algorithme de chiffrement sous une forme orthogonale, sera décrit dans la section 3.3. Enfin, un nouvel algorithme de chiffrement rapide, optimisé pour prendre avantage du *bitslice* et permettre le chiffrement automatique d'un disque dur, sera spécifié dans la section 3.4.

3.2 Méthodes d'implantation du *bitslice*

Dans toute la suite, on appellera « registre » une variable élémentaire dont la taille est un des registres du processeur ; les véritables registres doivent être considérés comme une mémoire cache sur ces variables.

3.2.1 Orthogonalisation des données

On suppose que l'on dispose d'un processeur capable de travailler sur des registres de n bits (typiquement, n vaut 32 ou 64). Les opérations élémen-

taires possibles sont les opérations logiques bit à bit, et les décalages. On souhaite donc calculer, en parallèle, n instances d'un algorithme cryptographique. On dispose de n entrées, chacune répartie sur un certain nombre de registres. On suppose tout d'abord que chaque entrée est de taille n , et que n est une puissance de 2 (c'est le cas sur tous les processeurs modernes).

On a donc n vecteurs de n bits, chacun étant placé dans un registre. On appelle (u_i) ($0 \leq i \leq n - 1$) ces valeurs. Chacune est constituée de n bits, numérotés de 0 à $n - 1$; elle définissent une matrice P dont les coefficients $p_{i,j}$ sont les bits des valeurs u_i : pour i et j compris entre 0 et $n - 1$, $p_{i,j}$ est le bit numéro j de u_i . On veut obtenir n vecteurs de n bits, nommés (v_i) , tels que les n bits de u_j soient les n bits numéro j des n vecteurs v_i ($0 \leq i \leq n - 1$), pour tout j de 0 à $n - 1$. Si on considère que les vecteurs v_i constituent la matrice Q ($q_{i,j}$ est le bit j de v_i), alors la relation qui lie les coefficients de P et de Q est :

$$\forall i, j \in \{0..n - 1\}, q_{i,j} = p_{j,i} \quad (3.1)$$

C'est-à-dire que Q est la transposée de P .

L'algorithme « naïf » de calcul d'une telle transposée consiste à extraire les bits d'entrée un par un, et à les combiner par des décalages et des OU logiques, afin de constituer la sortie. Le coût de cet algorithme est directement proportionnel au nombre de bits à traiter, qui est n^2 . Opérationnellement, le coût est de plusieurs cycles par bit de donnée, ce qui est supérieur au temps de chiffrement lui-même avec des systèmes de chiffrement par blocs classiques tels que l'AES. Mais les capacités de traitement du processeur peuvent être mieux exploitées; Donald Knuth décrit une bien meilleure méthode dans [52].

Cette méthode part de la constatation suivante : la transposition peut s'exprimer récursivement :

$${}^t \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} {}^t A & {}^t C \\ {}^t B & {}^t D \end{pmatrix} \quad (3.2)$$

Ainsi, la transposition d'une matrice de taille n revient à quatre transpositions de matrices de taille $n/2$, puis un échange de deux de ces quatre sous-matrices. Dans la représentation sur laquelle nous travaillons (chaque ligne de la matrice est un registre) cet échange s'effectue en un temps proportionnel à n à l'aide de masquages (c'est-à-dire de ET logiques de registres avec des constantes) et de décalages : la ligne 0 est combinée avec la ligne $n/2$, la ligne 1 est combinée avec la ligne $(n/2) + 1$, etc; chaque combinaison est constituée de quatre masquages (ET logique avec une constante), suivis de deux décalages puis de deux OU logiques. La figure 3.1 illustre la combinaison de deux lignes de la matrice.

On effectue la transposition récursivement; on doit donc d'abord effectuer la même opération sur des matrices de tailles $n/2$, et avant ça $n/4$, et encore

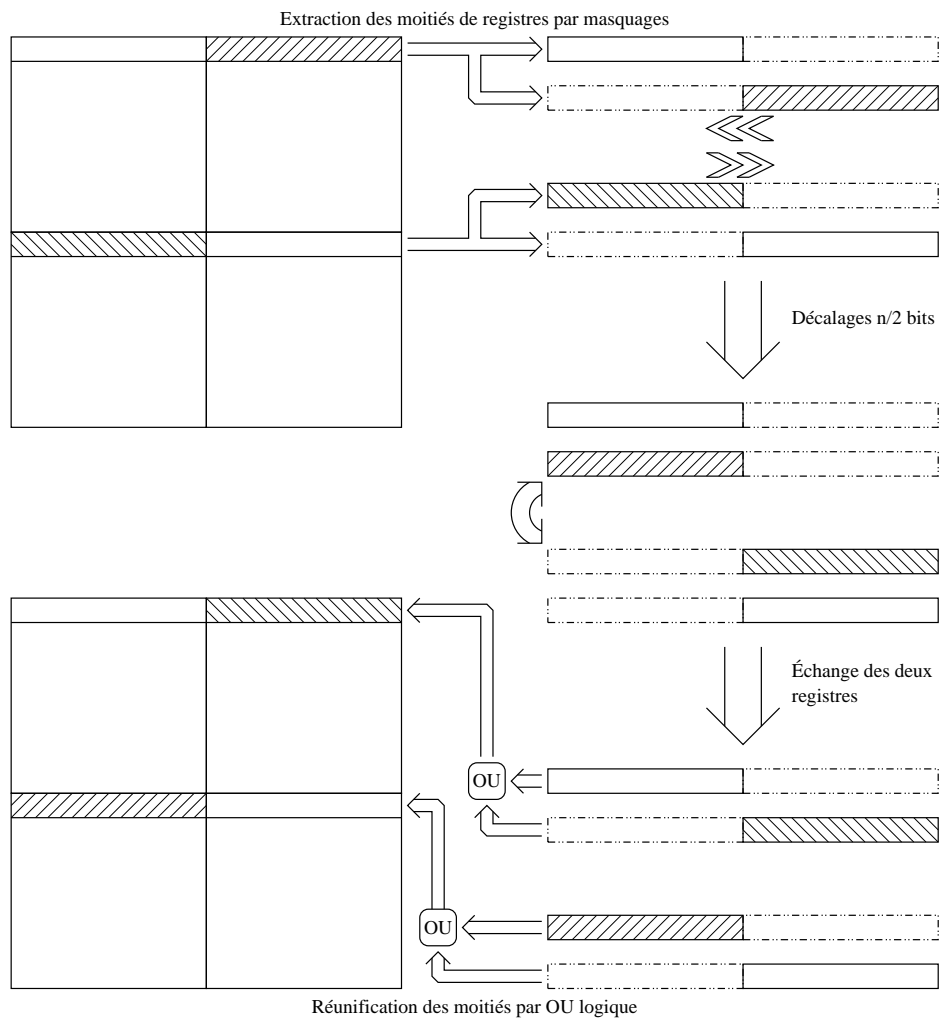


FIG. 3.1 : Échange final lors d'une transposition

avant $n/8$, *etc.* Le point important est que ces transformations s'effectuent avec des décalages constants sur une ligne donnée, et des opérations logiques bit à bit. Ainsi, quand on calculera tA et tB (dans les notations de l'équation 3.2), l'échange de champs de bits de taille $n/4$ dans la sous-matrice A , et celui dans la sous-matrice B , font intervenir les mêmes décalages. Comme ces deux sous-matrices sont stockées dans les mêmes registres, les décalages peuvent être effectués simultanément sur les deux matrices en une seule opération du processeur ; il en est de même des opérations logiques bits à bits. On pourra faire de même pour les opérations sur les matrices de taille $n/4$: on en traitera 4 simultanément. Le même principe se répète jusqu'aux matrices de taille 2 : chaque registre contient $n/2$ lignes de $n/2$ matrices.

Dans la pratique, on commence par les matrices de taille 2 ; on traite la première et la deuxième ligne de la matrice (stockée dans deux registres, qu'on nomme r_1 et r_2) comme ceci :

$$\begin{array}{rcll}
 r_3 & \leftarrow & r_1 & \wedge & 101010 \dots 10 \\
 r_4 & \leftarrow & r_1 & \wedge & 010101 \dots 01 \\
 r_5 & \leftarrow & r_2 & \wedge & 101010 \dots 10 \\
 r_6 & \leftarrow & r_2 & \wedge & 010101 \dots 01 \\
 r_1 & \leftarrow & r_3 & \vee & (r_5 \ggg 1) \\
 r_2 & \leftarrow & r_6 & \vee & (r_4 \lll 1)
 \end{array} \tag{3.3}$$

Après cette transformation, les $n/2$ sous-matrices 2×2 des deux premières lignes sont, individuellement, transposées. On effectue la même manipulation sur la troisième et la quatrième ligne, puis sur la cinquième et la sixième ligne, *etc.* Une fois toute la matrice traitée, les $n/2 \times n/2$ sous-matrices 2×2 de la matrice sont transposées.

Lors de la passe suivante, on utilise les constantes $11001100 \dots 00$ et $00110011 \dots 11$, des décalages de deux bits, et on combine la première ligne avec la troisième, la deuxième avec la quatrième, la cinquième avec la septième, la sixième avec la huitième, *etc.* À la fin de cette passe, les $n/4 \times n/4$ sous-matrices 4×4 de la matrice sont transposées. On continue ainsi pendant $\log n$ passes ; la dernière termine la transposition.

Le coût de chaque passe est $n/2$ échanges élémentaires, chacun étant constitué de quatre ET logiques avec des constantes, deux décalages et deux OU logiques. Il y a $\log n$ passes, ce qui donne un coût total de $4n \log n$ opérations logiques. Un code C implantant cet algorithme, compilé pour processeur Alpha avec le compilateur Compaq, effectue la transposition d'une matrice binaire 64×64 en 2294 cycles sur un Alpha 21164, et en 1471 cycles sur un Alpha 21264. Ceci donne un coût amorti d'environ 0,36 cycle d'horloge par bit de données traité sur ce processeur. Ce code C est fourni avec le logiciel `bsc` (cf. 3.3).

Dans le cas où on doit travailler sur des données de taille m différente de la largeur n des registres, on commence par traiter les sous-blocs de taille n parmi les données. Si m est plus petit que n , la méthode décrite ci-dessus reste globalement inchangée ; on se contente de rajouter des données « aléatoires » afin d'obtenir une matrice $n \times n$. Une partie des premières phases peut être ignorée, car elle ne travaillerait que sur des données explicitement dépourvues de signification. Si m est vraiment petit par rapport à n , l'algorithme naïf peut devenir plus efficace que la transposition améliorée ; l'emplacement de ce seuil dépend de l'architecture utilisée, mais est aux environs de $\log n$.

3.2.2 Orthogonalisation des tables

En code orthogonal, toute table doit être calculée comme un circuit. Une table indexée par un entrée de n bits et ayant une sortie sur k bits est la représentation générale d'une fonction booléenne de n entrées et de k sorties ; donc, sauf réduction particulière due à la structure des entrées de la table, cette transformation est la plus difficile à optimiser.

Il convient de définir ce qu'on entend par « optimal ». Le processeur qui exécute du code *bitslice* calcule les opérations booléennes séquentiellement ; on considère donc que l'efficacité du code (sa vitesse d'exécution) est inversement proportionnelle au nombre de ces opérations, c'est-à-dire la taille du circuit booléen calculé. C'est en fait une vision simplifiée du problème ; des détails seront développés dans la section 3.3.

Le problème de minimisation d'un circuit booléen implantant une table booléenne donnée est étudié depuis un certain temps, et connu pour être difficile. Shannon[79] a montré que la taille du circuit minimal implantant une fonction booléenne de n entrées et une sortie est en moyenne d'au moins $2^n/n$. Kabanets et Cai[46] ont étudié le problème, apparemment plus simple, de décider si le circuit minimal correspondant à une table de vérité donnée est de taille inférieure à une borne donnée, et ont montré que ce problème est « probablement » difficile ; en tous cas, aucun algorithme efficace pour le résoudre n'est actuellement connu.

Dans le cas des boîtes S de DES, les circuits les plus optimaux connus à ce jour ont été calculés par Matthew Kwan[57] ; la taille moyenne est de 56 portes (ET, OU, NON et OU EXCLUSIF) par boîte S, ou 51 si on rajoute quelques opérations logiques moins standard mais présentes sur les processeurs Alpha et Sparc. Ces résultats ont été intégrés dans le projet de recherche exhaustive de clé DES par logiciel qui a servi à remporter certains des défis de cryptanalyse posés par RSA Data Security[31].

Les BDD

Les *Binary Decision Diagrams* sont une méthode générique pour trouver des circuits implantant relativement efficacement des fonctions booléennes données. On se place dans le cas d'une fonction de n entrées et une seule sortie.

La brique élémentaire est le multiplexeur : c'est une fonction prenant trois entrées ; la troisième est l'entrée de « contrôle ». Si l'entrée de contrôle vaut 0, alors la sortie est égale à la première entrée, sinon elle est égale à la seconde. Le multiplexeur est une sorte d'aiguillage. Si on note c l'entrée de contrôle, a et b les deux autres entrées, alors la sortie du multiplexeur est :

$$\text{mux}(c, a, b) = (\neg c \wedge a) \vee (c \wedge b) \quad (3.4)$$

Le circuit minimal implantant un multiplexeur, dans le cas où le processeur connaît uniquement les quatre opérations booléennes « classiques » (ET, OU, NON et OU EXCLUSIF), comporte trois portes logiques :

$$\text{mux}(c, a, b) = a \oplus (c \wedge (a \oplus b)) \quad (3.5)$$

On va construire un arbre de multiplexeurs permettant d'aiguiller la bonne valeur de sortie pour chaque combinaison possible des entrées. Cet arbre sera un arbre binaire complet, dont la profondeur est égale au nombre d'entrées de la fonction. Les feuilles sont des constantes binaires, valant 0 ou 1. Le premier bit d'entrée de la fonction sert d'entrée de contrôle des multiplexeurs directement reliés aux feuilles ; le deuxième bit d'entrée de la fonction contrôle les multiplexeurs de rang immédiatement supérieur, *etc.* Il n'est pas difficile de voir que chaque combinaison des bits d'entrées sélectionne comme valeur de sortie une feuille ; si les bits d'entrée de la fonction forment la valeur binaire r (le premier bit étant le bit de poids faible), la feuille sélectionnée est la numéro r , en comptant les feuilles à partir de 0 de gauche à droite.

La figure 3.2 illustre cette construction sur la fonction f suivante :

x	$f(x)$	x	$f(x)$	x	$f(x)$	x	$f(x)$
0000	0	0100	0	1000	0	1100	1
0001	1	0101	1	1001	1	1101	1
0010	1	0110	1	1010	1	1110	1
0011	1	0111	0	1011	0	1111	0

Un BDD est optimisable en unifiant les branches qui calculent les mêmes expressions ; par ailleurs, quand un multiplexeur possède deux entrées identiques, une seule entrée a besoin d'être calculée, et la sortie du multiplexeur a

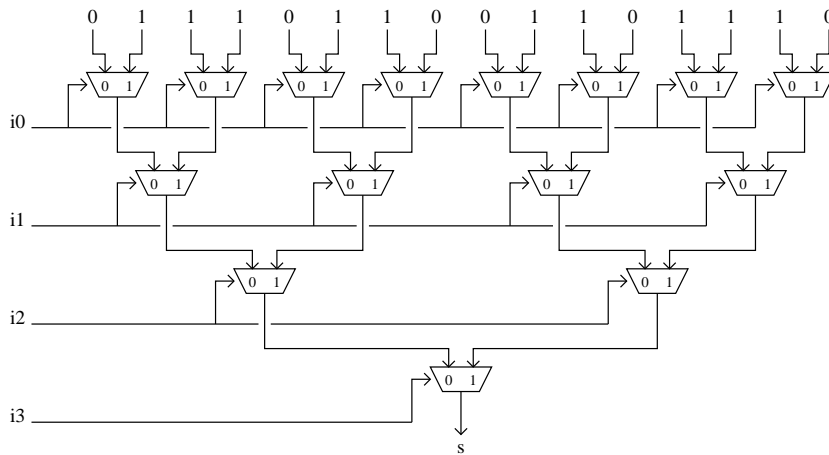


FIG. 3.2 : Un BDD

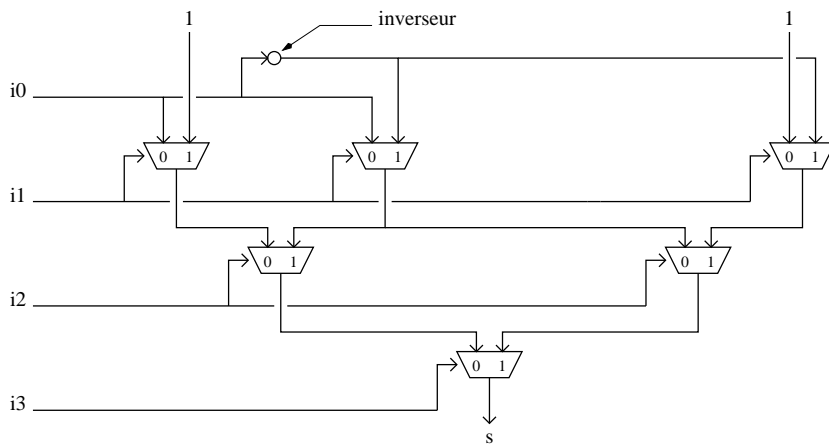
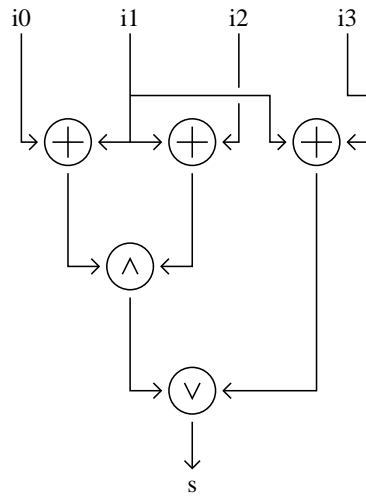


FIG. 3.3 : Un BDD optimisé

pour valeur cette entrée. Enfin, quand les deux entrées de données d'un multiplexeur sont les constantes 0 et 1, alors la sortie est égale soit à l'entrée de contrôle, soit à son inverse. La figure 3.3 montre le résultat de l'optimisation du BDD de la figure 3.2 implantant la fonction f .

Comme on le voit, dans le cas de la fonction f , le BDD non optimisé requiert l'évaluation de 15 multiplexeurs, soit 45 fonctions booléennes « classiques » ; mais le BDD optimisé ne comporte plus que 6 multiplexeurs et un inverseur, pour un total de 19 fonctions booléennes.

- Dans la pratique, on pratiquera quelques optimisations supplémentaires :
- Si un multiplexeur admet pour entrées deux fonctions qui sont toujours opposées, alors la sortie de ce multiplexeur est le OU EXCLUSIF de son

FIG. 3.4 : Circuit minimal de la fonction f

entrée « 0 » et de l'entrée de contrôle.

- On peut essayer ces optimisations sur les $n!$ arbres BDD pour les différentes permutations des n entrées de la fonction booléenne.

Enfin, Biham[6] a remarqué qu'il n'était pas utile de travailler avec des multiplexeurs ; on peut en effet obtenir le même « pouvoir de calcul » avec la fonction suivante :

$$\text{fmx}(c, a, b) = a \oplus (c \wedge b) \quad (3.6)$$

Ce « faux multiplexeur » effectue un « choix » entre a et $a \oplus b$ au lieu de a et b . En remplaçant tous les noeuds d'un BDD non optimisé par ces faux multiplexeurs, et en appliquant une transformation linéaire constante sur l'ensemble des feuilles, on obtient un circuit calculant bien la fonction. Les optimisations applicables sur ce nouvel arbre ne sont pas les mêmes que celles applicables sur un BDD, et un bon algorithme d'optimisation essaiera les deux méthodes, en conservant celle qui donne le meilleur résultat. La construction avec les faux multiplexeurs part avec l'avantage de ne nécessiter que deux fonctions booléennes élémentaires par noeud de l'arbre.

Ceci dit, aucune de ces optimisations ne trouve le circuit minimal de la fonction f , illustré par la figure 3.4, qui ne comporte que 5 fonctions booléennes élémentaires. Ces solutions ne sont donc pas pleinement satisfaisantes.

3.3 Production automatique de code *bitslice*

La représentation orthogonale des données permet une grande efficacité sur certains algorithmes cryptographiques, mais les langages de programmation traditionnels n'offrent pas d'outils syntaxiques adaptés à l'écriture d'un tel code ; les constructions d'un langage tel que le C sont prévues pour exprimer facilement des calculs sur des pointeurs, des boucles et des appels de fonction, toutes choses qui ne sont pas utiles pour la programmation *bitslice*.

Il a donc été développé, durant cette thèse, un outil, nommé `bsc`, qui se comporte comme un pré-compilateur : il prend en entrée un code dans un langage simple décrivant un algorithme, et produit un code C l'implantant en structure *bitslice*. Ce programme est disponible gratuitement [71, 72] et est censé pouvoir fonctionner sur n'importe quel système Unix ou apparenté. Je vais dans cette section décrire sa structure, son mode de fonctionnement, et ses améliorations futures.

3.3.1 Le langage de description d'algorithmes

Les algorithmes cryptographiques susceptibles d'être implantés en code orthogonal ont les caractéristiques suivantes :

- les primitives de calcul sont des opérations booléennes bit à bit, des permutations de bits et des tables de petite taille ;
- chaque primitive est susceptible d'être utilisée un grand nombre de fois ;
- l'exécution des instructions est indépendante des données traitées.

Cette dernière propriété est cruciale, car c'est elle qui permet le traitement en parallèle de plusieurs instances du système, ce qui est le principe même du *bitslice*. Une conséquence immédiate est que l'algorithme doit pouvoir s'exprimer simplement sans structure de boucle ou de saut conditionnels.

Les données manipulées sont toutes des valeurs binaires, regroupées en champs de bits d'une certaine largeur, qui représentent des données de cette même taille. La déclaration d'un tel champ s'effectue à l'aide de la syntaxe suivante :

```
bit x[32];
```

Ceci définit 32 valeurs binaires, regroupées sous le nom de `x`. Dans le cas de DES, les moitiés gauche et droite de la sortie de chaque tour sont déclarées ainsi, par exemple. Deux champs de bits spéciaux, `input` et `output`, doivent être définis ; ils représentent les entrées et les sorties de l'algorithme (dans le cas d'un chiffrement DES, les entrées sont le texte clair et la clé, et les sorties contiennent le texte chiffré).

On manipule ces variables dans des expressions similaires à celles du langage C, parenthésables, avec les opérateurs suivants :

= représente l'affectation : la valeur calculée à droite est écrite dans le champ de bits indiqué à gauche ; cette valeur est un champ de bits qui doit avoir, bien sûr, la même largeur que le champ destination.

&, | et ^ sont des opérateurs infixes calculant respectivement le ET, le OU et le OU EXCLUSIF de leurs opérandes.

&=, |= et ^= sont les contractions des opérateurs précédents avec l'affectation : l'expression $a \&= b$ a le même effet que $a = a \& b$.

[a # b] est la concaténation : cet opérateur fabrique un champ de bits formé de la juxtaposition de ses champs de bits opérandes ; le résultat est un champ de bits dont la largeur est la somme des largeurs des opérandes, et qui peut servir de destination d'un opérateur d'affectation. On peut indiquer plus de deux opérandes.

On peut également définir, et utiliser des tables et des fonctions de routage des données. La déclaration suivante :

```
ext E[48](32) {
    31, 0, 1, 2, 3, 4, 3, 4, 5, 6, 7, 8,
    7, 8, 9, 10, 11, 12, 11, 12, 13, 14, 15, 16,
    15, 16, 17, 18, 19, 20, 19, 20, 21, 22, 23, 24,
    23, 24, 25, 26, 27, 28, 27, 28, 29, 30, 31, 0
};
```

définit la fonction E, qui prend en entrée un champ de 32 bits, et rend un champ de 48 bits ; les bits de sortie de E sont, dans l'ordre, les bits 31, 0, 1, 2, 3, 4, 3, *etc.*, de l'entrée. On reconnaît là la définition de l'expansion E d'un tour de DES.

Pour déclarer une table, on utilise la syntaxe suivante :

```
tab S1[4](6) {
    14, 0, 4, 15, 13, 7, 1, 4, 2, 14, 15, 2, 11, 13, 8, 1,
    3, 10, 10, 6, 6, 12, 12, 11, 5, 9, 9, 5, 0, 3, 7, 8,
    4, 15, 1, 12, 14, 8, 8, 2, 13, 4, 6, 9, 2, 1, 11, 7,
    15, 5, 12, 11, 9, 3, 7, 14, 3, 10, 10, 0, 5, 6, 0, 13
};
```

Cette déclaration définit la table S1, prenant 6 bits en entrée et offrant une sortie sur 4 bits. Les valeurs indiquées sont les nombres représentés en binaire par la sortie de la table pour les 2^6 entrées possibles.

L'utilisation d'une fonction de routage ou d'une table s'indique de la même manière qu'un appel de fonction en C : S1(x) est la sortie de la table S1 pour l'entrée x.

À titre d'exemple, voici un tour de DES, codé dans ce langage. Un chiffrement DES complet est fourni avec `bsc` :

```

[t1 # t2 # t3 # t4 # t5 # t6 # t7 # t8] = E(r) ^ sk1(key);
t = l ^ P([S1(t1) # S2(t2) # S3(t3) # S4(t4) #
          S5(t5) # S6(t6) # S7(t7) # S8(t8)]);
l = r; r = t;

```

La première ligne calcule l'expansion E sur la partie droite de l'entrée du tour (stockée dans le champ de bits r), et effectue un OU EXCLUSIF du résultat avec la sous-clé du tour, qui est évaluée par la fonction de routage $sk1$ (dans DES, chaque sous-clé de 48 bits n'est qu'une extraction fixe de certains des bits de la clé). Le résultat est envoyé dans la concaténation des champs de bits $t1$ à $t8$: en quelque sorte, on sépare les 48 bits obtenus en 8 mots de 6 bits, qui seront envoyés dans les huit boîtes S . C'est exactement ce qui se passe dans les deux lignes suivantes : les 8 boîtes S sont invoquées, et leurs résultats concaténés, ce qui donne un mot de 32 bits, dont la forme permutée par la fonction P (qui implante la permutation P de DES) est combinée par OU EXCLUSIF avec la partie gauche de l'entrée du tour. Enfin, la dernière ligne échange les parties gauche et droite de la sortie du tour, complétant ainsi le schéma de Feistel constitutif de DES.

3.3.2 Stratégie d'optimisation

Analyse et optimisation des actions

Lors de l'analyse du code établi dans le langage décrit ci-dessus, `bsc` commence par séparer tous les bits manipulés comme autant de registres de 1 bit indépendants, puis établit la liste de toutes les opérations que subissent ces registres ; ces opérations agissent en lecture ou en écriture sur le contenu des registres. Dans la suite, j'appelle ces opérations des *actions*, dont voici la liste :

- `OR`, `AND` et `XOR` sont, respectivement, le OU, le ET et le OU EXCLUSIF de deux registres, avec résultat dans un troisième. Ces actions agissent en lecture sur les deux registres opérandes, et en écriture sur le registre destination.
- `NOT` calcule la négation logique d'un registre et écrit le résultat dans un autre.
- `COPY` copie le contenu d'un registre dans un autre.

En outre, des indications sur la portée des registres sont insérées dans la liste des actions : elles permettent à `bsc` de retrouver les limites des appels aux fonctions de routage et aux tables.

Ensuite, la liste des actions est « optimisée » afin d'éliminer les copies inutiles. Cette optimisation se déroule ainsi :

- Les registres implicites utilisés pour stocker les paramètres des fonctions

de routage et des tables sont dupliqués : de nouveaux registres sont créés pour chaque appel de fonction.

- Chaque fois qu’une opération de copie a lieu d’un registre vers un autre, ces deux registres sont marqués comme potentiellement équivalents.
- Les fausses équivalences sont éliminées : pour que deux registres u et v soient considérés comme équivalents, il faut que les deux conditions suivantes soient réunies :
 - chaque fois qu’une lecture de v suit une écriture de u , il doit y avoir une écriture de v entre ces deux actions ;
 - chaque fois qu’une lecture de u suit une écriture de v , il doit y avoir une écriture de u entre ces deux actions.
- Une fois les vraies équivalences isolées, elles sont propagées par transitivité.

La liste d’actions résultantes est écrite sous la forme d’un fichier C : chaque registre est déclaré sous la forme d’une variable locale de type entier (ce type est configurable ; les meilleures performances seront obtenues quand il correspond à la taille des registres du processeur), et les actions sont traduites en les opérateurs C correspondants. Le fichier C est alors prêt à être intégré dans le logiciel.

Le cas des tables

Les tables sont traduites en actions élémentaires en utilisant un BDD modifié avec de faux multiplexeurs (cf. section 3.2.2). Ces BDD ne sont pas complètement optimisés : seule la première ligne de multiplexeurs, celle dont les entrées sont des constantes valant 0 ou 1, est remplacée par les opérations élémentaires correspondantes (soit une constante, soit le premier bit d’entrée de la table, soit son inverse).

La raison principale de ce choix de sous-optimisation est la suivante : le critère d’optimalité décrit dans la section 3.2.2 est en fait trop simplifié. Dans un processeur moderne, plusieurs instructions peuvent être exécutées simultanément, sous réserve qu’elles ne soient pas en conflit d’utilisation de certaines ressources (sur Alpha, jusqu’à quatre opérations logiques bit à bit peuvent être effectuées en même temps si aucune d’elle n’écrit dans un registre lu par une des trois autres) ; par ailleurs, il y a moins de vrais registres que de variables référencées dans le code *bitslice*, ce qui implique que les registres du processeur sont utilisés comme un cache sur la mémoire, et que le compilateur C invoqué va devoir gérer un grand nombre de lectures et d’écritures en mémoire. Ces accès sont soumis à certaines contraintes qui compliquent fortement l’optimisation (par exemple, sur un Alpha 21164, quand un accès en écriture a lieu à un cycle d’horloge donné, un accès en lecture peut avoir

lieu au cycle suivant, mais pas pendant le cycle d'après). Ces contraintes impliquent que le circuit minimal n'est pas forcément le plus efficace sur une machine donnée, et que le code le plus optimal sur une architecture peut être de performances médiocres sur une autre. Dans son état actuel, `bsc` se veut être un outil générique, et donc laisse la fin du travail d'optimisation à l'outil existant qui maîtrise le mieux les paramètres de l'architecture cible, à savoir le compilateur C.

3.3.3 Évolution future de `bsc`

`bsc` est pour le moment une preuve de concept : il démontre que des outils relativement simples peuvent permettre de développer rapidement un code efficace utilisant la représentation orthogonale. Voici un inventaire des améliorations qui pourraient, à moyen terme, lui donner un vrai statut opérationnel.

Améliorations syntaxiques

Il est envisageable de rajouter le support de nouveaux opérateurs logiques, tels que NAND ou XORNOT. Certains processeurs supportent nativement ces opérateurs, et sinon ils sont émulables par des couples ou triplets d'opérateurs élémentaires. Ces nouveaux opérateurs peuvent alléger l'expression d'un algorithme cryptographique qui en ferait usage, et donc accélérer le développement. De même, des primitives de calcul d'additions ou soustractions binaires pourraient être rajoutées (ce qui suppose la gestion de la propagation des retenues).

Diverses structures syntaxiques pourraient aussi être introduites :

- des structures de blocs, qui permettent de limiter la portée des noms de variables et donc de réutiliser les noms de champs de bits ;
- de vraies fonctions avec des variables locales (avec une sémantique de remplacement de macros, ce qui n'est pas en soi une limitation, car le code orthogonal interdit les sauts dépendants des données, et donc, *a fortiori*, les appels récursifs de fonctions).

Enfin, l'intégration d'un préprocesseur du code source (semblable à celui du langage C) pourrait s'avérer utile.

Améliorations structurelles

Le temps de compilation du code C produit par `bsc` est actuellement très long, car les compilateurs ne sont pas prévus pour travailler sur ce genre de code, qui comporte beaucoup de variables locales et aucune structure de

boucle. Par ailleurs, sur l'architecture la plus répandue actuellement, le PC à processeur Intel (ou compatible), les meilleures performances en code orthogonal sont obtenues en utilisant les registres de 64 bits contenus dans l'unité MMX : cette unité est un coprocesseur normalement dédié au traitement du signal, mais elle permet d'effectuer des opérations logiques bit à bit entre ses registres, et est assez adaptée au *bitslice*. Or, aucun compilateur C ne permet de produire du code utilisant ce matériel ; il faut le programmer directement en assembleur.

Aussi, `bsc` aurait intérêt à incorporer son propre générateur de code assembleur. Une conséquence est que l'optimisation des accès mémoire, l'allocation des registres et la recherche des circuits optimaux devraient alors être pris en charge par `bsc`. Du fait de la complexité d'une telle entreprise, cette modification n'est envisageable qu'à long terme.

3.4 Chiffrement symétrique d'un disque dur

Les vingt dernières années ont vu l'émergence d'une nouvelle forme d'informatique : l'informatique mobile. Désormais, l'ordinateur peut être portable, et on peut emmener avec soi plusieurs giga-octets de données, éventuellement confidentielles. Or, un ordinateur portable n'est pas protégeable physiquement comme on pourrait le faire d'un système fixe placé dans une enceinte blindée. Aussi, toute donnée placée sur son disque dur doit être sécurisée cryptographiquement. Ceci pose des problèmes de performances, car les disques durs modernes proposent des débits de données élevés, et si la sécurité des données est importante, elle est plus un mal nécessaire qu'une fin en soi. Le chiffrement des données doit être le plus discret possible, et donc n'utiliser qu'une faible portion de la puissance de calcul offerte par l'ordinateur. Dans cette section sont décrites les raisons qui font que les algorithmes symétriques traditionnels tels que l'AES[69], et les chiffrements en flux similaires à RC4[89] (cf. section 2.3.1), n'offrent pas de solutions pleinement satisfaisantes ; il est également spécifié un nouvel algorithme, nommé FBC, qui permet d'obtenir des performances supérieures grâce à la technique du code orthogonal. Enfin, les différentes classes d'attaques possibles et la façon de les contrer sont discutées. Ce travail a été présenté à CHES'2001[73].

3.4.1 Conditions opérationnelles

Les disques durs sont des substrats magnétiques qui peuvent stocker des données binaires, mais qui possèdent par ailleurs une certaine rémanence ; de plus, l'alignement des têtes de lecture du disque n'est jamais complètement

garanti, ce qui fait qu'une écriture n'écrase pas totalement les données précédemment stockées à cet endroit. On peut utiliser ces informations lors d'une analyse physique du disque afin de reconstituer des données qui, d'un point de vue logique, ont été effacées[33, 68, 85]. Il existe des logiciels spécialisés dans l'effacement définitif de fichiers (cf. [28] pour une imposante liste de tels logiciels), mais leur fiabilité n'est pas assurée à 100%, pour, entre autres, les raisons suivantes :

- ces logiciels reposent sur les écritures successives de différents motifs, qui correspondent à une certaine technologie d'écriture des données[42] : des disques plus récents peuvent utiliser d'autres moyens ;
- les disques durs modernes possèdent des caches mémoire et peuvent, sur plusieurs écritures successives, n'effectuer physiquement que la dernière ;
- l'efficacité de l'effacement n'est pas formellement prouvée, mais seulement estimée en regard de la technologie d'analyse, qui progresse continuellement et de manière globalement non prédictible.

De toutes façons, le coût opérationnel d'un logiciel d'éradication physique de données est très important : chaque fichier effacé doit être réécrit une bonne dizaine de fois ; en plus des temps d'attente impliqués, cela augmente fortement l'usure du disque dur. Dans les systèmes d'exploitation modernes, l'effacement d'un fichier est une opération courante et rapide, une simple désallocation de l'espace occupé par le fichier ; une augmentation substantielle du coût de cette opération n'est pas raisonnable.

Il s'ensuit que toute donnée confidentielle arrivant sur le disque dur doit être chiffrée. Or, ces données sont traitées par des logiciels parfois complexes dont le fonctionnement n'est pas forcément pleinement spécifié, et qui peuvent créer des fichiers temporaires ou utiliser de la mémoire virtuelle contenant des données sensibles. La seule manière de protéger ces données dans de telles conditions est de chiffrer tout le disque dur. Il existe des logiciels permettant de chiffrer et déchiffrer automatiquement des fichiers sur le disque, indépendamment de l'application qui utilise ces fichiers ; voir par exemple [3] ou [59]. Malheureusement, le chiffrement utilisé, souvent de bonne qualité d'un point de vue sécurité, entraîne un débit de données nettement plus faible que ce que permet la mécanique du disque dur sous-jacent ; aussi, les utilisateurs restreignent le chiffrement à des zones très réduites de leur disque dur, et corollairement ne chiffrent pas les fichiers temporaires et la mémoire virtuelle utilisée. Le problème du chiffrement de mémoire virtuelle est globalement moins bien traité que le cas des fichiers, notamment parce qu'il suppose une intégration beaucoup plus poussée dans les niveaux les plus internes du système d'exploitation ; une implantation existe sur le système OpenBSD[75].

L'ordinateur portable moderne possède, schématiquement, les caractéristiques suivantes :

- cadence : 1 GHz ;
- capacité du disque dur : 10 à 40 Go ;
- débit du disque dur : 20 Mo/s.

On souhaiterait pouvoir suivre le rythme du disque dur en n'y consacrant qu'environ 15% de la puissance du processeur ; ceci implique un temps de chiffrement d'au plus 1 cycle d'horloge par bit traité. Or, on a vu dans le chapitre 2 que les meilleures implantations traditionnelles de systèmes de chiffrement symétrique ne descendent pas en dessous de 2 cycles par bit traité. Certains systèmes de chiffrement en flux sont suffisamment rapides, mais demandent un temps de mise en place qui s'accommode mal d'un accès non séquentiel ; or, ces accès font justement tout l'intérêt des disques durs par rapport aux bandes magnétiques, et les disques sont constamment utilisés ainsi. De plus les systèmes de chiffrement par flux sont souvent malléables, c'est-à-dire que le texte clair reste transformable de façon prédictible à travers le texte chiffré sans connaissance de la clé ; on verra dans la section 3.4.5 que c'est une caractéristique à éviter quand on cherche à se protéger contre les attaques actives.

Afin de réaliser les conditions voulues (chiffrement rapide, de tout le disque ou quasiment, non malléable et en n'utilisant qu'au plus 15% du processeur central), le mieux serait de disposer d'un matériel spécifique de chiffrement, implanté sur le contrôleur de disque, ou même directement sur le disque lui-même. Ces matériels sont très rares, et ne semblent pas disponibles pour l'utilisateur moyen. Comme on l'a vu dans la section 2.3, la tendance économique actuelle est de se reposer le plus possible sur un processeur générique et du logiciel, plutôt que d'utiliser des matériels « intelligents ». Les algorithmes de chiffrement disponibles ne remplissant pas les caractéristiques demandées, un nouvel algorithme est proposé dans la section suivante.

3.4.2 FBC

FBC est un acronyme signifiant *Fast Bitslice Cipher*. C'est un nouvel algorithme de chiffrement spécialisé dans la sécurisation d'un disque dur.

Structure de FBC

Le chiffrement d'un disque dur impose certes des contraintes drastiques sur l'efficacité devant être atteinte, mais procure également quelques avantages qui ne demandent qu'à être exploités :

- les accès en lecture et en écriture se font toujours par blocs de grande taille : l'unité élémentaire est le secteur, souvent de 512 octets, mais les systèmes d'exploitation modernes tendent de plus en plus à adopter une granularité de l'ordre de 4 ou 8 Ko afin de suivre au plus près les mécanismes de mémoire virtuelle intégrés au processeur, et qui manipulent des pages de cette taille ;
- le temps de diversification des clés est sans importance : en effet, cette opération n'est faite qu'une fois par lancement de l'ordinateur ;
- par essence, c'est un processeur central moderne qui effectue le chiffrement et le déchiffrement ; on peut donc optimiser l'algorithme pour ce genre d'architecture sans se soucier du matériel ou des cartes à puce.

Puisqu'on effectue toujours plusieurs chiffrements en parallèle, l'idée est de travailler en mode *bitslice* : si on utilise un système de chiffrement par blocs de 64 bits, chaque chiffrement d'un secteur de 512 octets correspond à 64 tels blocs. L'algorithme sera donc conçu à l'aide d'opérations booléennes élémentaires, et de permutations de bits (qui ne coûtent rien à l'exécution en code orthogonal). Puisque le temps de diversification des clés est sans importance, toutes les permutations et fonctions booléennes utilisées peuvent être rendues dépendantes de la clé, et calculées par un générateur pseudo-aléatoire cryptographiquement sûr afin de rendre imprédictibles les liens entre les divers éléments.

FBC est défini ainsi : il utilise un schéma de Feistel ; chaque fonction de confusion est un ensemble de fonctions booléennes agissant sur des bits d'entrée choisis aléatoirement en fonction de la clé. L'usage est habituellement de fabriquer une fonction de confusion ayant de bonnes caractéristiques cryptographiques, et de rajouter autant de tours que nécessaire afin d'atteindre la sécurité voulue. FBC prend une méthodologie légèrement différente : la fonction de confusion est prévue pour être rapide, mais apportant une sécurité relativement faible ; afin de compenser cela, de très nombreux tours sont mis bout à bout.

La fonction de chiffrement

Dans toute la suite, on manipule des mots binaires, dont on compte les bits de gauche à droite, à partir de 1. C'est la même numérotation que pour DES (cf. 1.3).

FBC agit sur des blocs de taille w bits (w est une valeur paire, par exemple 64) et comporte r tours. Pour des raisons pratiques d'implantation, w sera d'au plus 512, mais cette limitation n'est pas conceptuelle : elle est liée au fait que le générateur pseudo-aléatoire utilisé lors de la diversification des clés émet une suite d'octets ; une adaptation de ce générateur permettrait

d'utiliser une taille de bloc supérieure. L'entrée du tour i est découpée en deux mots binaires de taille $w/2$; la moitié gauche est appelée L_i , la moitié droite est R_i . Il existe, pour chaque tour, $w/2$ fonctions booléennes τ_i^j ($1 \leq j \leq w/2$) dépendantes de la clé, et deux permutations de $w/2$ éléments nommées ϕ_i et ψ_i , elles aussi calculées à partir de la clé, et qui vérifient les propriétés suivantes :

$$\forall j, \tau_i^j = \text{ET, OU, NON-ET ou NON-OU} \quad (3.7)$$

$$\forall j, \phi_i(j) \neq \psi_i(j) \quad (3.8)$$

Le résultat de la fonction de confusion est T_i , tel que pour tout j entre 1 et $w/2$, le bit j de T_i est égal à $\tau_i^j(R_i[\phi_i(j)], R_i[\psi_i(j)])$. Le résultat du tour est aussi l'entrée du tour suivant, définie ainsi :

$$L_{i+1} = R_i \quad (3.9)$$

$$R_{i+1} = L_i \oplus T_i \quad (3.10)$$

(\oplus désigne le OU EXCLUSIF bit à bit).

L'entrée du premier tour est le texte clair ; le texte chiffré est la sortie du dernier tour, dont les deux moitiés ont été échangées, c'est-à-dire la concaténation dans cet ordre de R_{r+1} et L_{r+1} . La figure 3.5 illustre un tour de FBC.

La diversification de la clé

Le reste de la spécification de FBC concerne la diversification de la clé ; il s'agit de déterminer, en fonction de cette dernière, les différents éléments constituant la fonction de confusion, à savoir les fonctions τ_i^j et les permutations ϕ_i et ψ_i . Ces éléments sont choisis aléatoirement et uniformément en fonction de la sortie d'un générateur pseudo-aléatoire dont la graine est la clé.

On utilise la fonction de hachage SHA-1[78] comme cœur du générateur ; cette fonction est définie comme prenant une entrée de taille quelconque, et rendant une valeur sur 160 bits. En fait, l'entrée est d'abord « allongée » (*padding*) jusqu'à un multiple de 512 bits, puis découpée en blocs de cette taille traités séquentiellement. Ici, on utilise SHA-1 sans ce padding, et sur des entrées de 512 bits exactement ; on note SHA' cette variante de SHA-1.

La clé K est une suite de bits de taille arbitraire, de longueur comprise entre 0 et 352 bits. Afin de déjouer les attaques par recherche exhaustive de la clé, on choisira une longueur de clé d'au moins 80 bits. La clé est allongée à 352 bits par ajout à droite d'autant de 0 que nécessaire ; on note K' cette nouvelle clé. On définit le générateur grâce à une variable-état S de 160 bits,

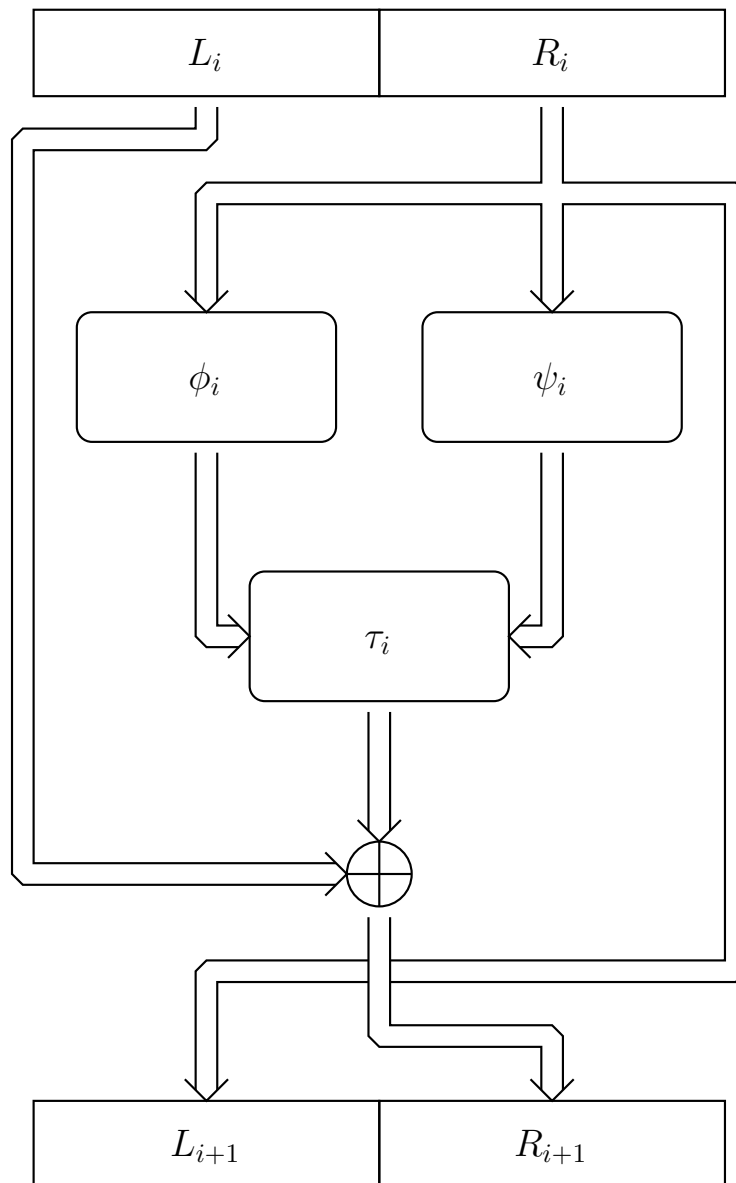


FIG. 3.5 : Un tour de FBC

qui sera considérée comme une chaîne de 20 octets, en commençant par les bits de gauche ; chaque octet étant représenté par 8 bits, le bit de poids fort est celui le plus à gauche. l'algorithme est le suivant :

1. $S \leftarrow 0$
2. $S \leftarrow \text{SHA}'(K' || S)$ ($||$ dénote la concaténation)
3. Le contenu de S est émis sous la forme de 20 octets successifs
4. Retour en 2

Le générateur émet ainsi autant d'octets que voulu. Cette construction est celle qui a été présentée à CHES'2001[73] mais d'autres générateurs pseudo-aléatoires, tels que ceux décrits dans l'annexe 3 du standard américain de signature électronique DSS[32], sont utilisables ; l'interopérabilité entre deux logiciels de chiffrement de disque dur n'importe que lors de l'usage de disques amovibles (tels que les disques Zip™ d'Iomega), et même dans ce cas, comme on le verra dans la section 3.4.3, le générateur pseudo-aléatoire utilisé n'a pas forcément besoin d'être le même dans les différentes implantations. La sécurité de FBC repose sur le fait que la suite de bits générée est imprédictible, et pas sur les caractéristiques précises du générateur.

On aura besoin de choisir des nombres aléatoires entre 0 et n , pour différents entiers n positifs et strictement inférieurs à $w/2$; pour ce faire, on définit m comme étant le plus grand entier multiple de $n + 1$ et inférieur ou égal à 256. On extrait alors du générateur un octet. Si la valeur b de cet octet est supérieure ou égale à m , on extrait un autre octet, et ainsi de suite jusqu'à obtenir une valeur b strictement inférieure à m . Comme m vaut au moins 129, le nombre moyen d'octets devant être produits par le générateur aléatoire est inférieur à 2. Le nombre aléatoire voulu est alors le reste de la division euclidienne de b par $n + 1$. Il est facile de voir que ce processus fournit effectivement des nombres entiers distribués uniformément entre 0 et n .

Le choix d'une permutation aléatoire de $w/2$ éléments s'effectue ainsi :

1. Remplir un tableau p de $w/2$ éléments avec les nombres entiers de 1 à $w/2$ en ordre croissant (c'est-à-dire, pour tout i de 1 à $w/2$, affecter la valeur i à $p[i]$).
2. Pour tout i de 2 à $w/2$:
 - choisir un entier aléatoire a_i compris entre 0 et $i - 1$, en utilisant la méthode décrite ci-dessus ;
 - si $a_i + 1 \neq i$, échanger les contenus de $p[a_i + 1]$ et $p[i]$.
3. La permutation choisie est représentée par le contenu de p (si $p[5] = 8$, alors la permutation envoie le contenu de sa cinquième entrée sur sa huitième sortie).

Cet algorithme choisit une permutation de $w/2$ entrées de façon équiprobable dans l'ensemble de telles permutations (cf. [54]).

On dispose désormais de tous les outils nécessaires pour déterminer, en fonction de la clé, les éléments constitutifs de chaque tour de FBC. On commence par initialiser le générateur pseudo-aléatoire, puis on tire les éléments de chaque tour, séquentiellement. Pour le tour i , on pratique ainsi :

1. On choisit aléatoirement la permutation ϕ_i .
2. On choisit aléatoirement la permutation ψ_i .
3. S'il existe au moins un j entre 1 et $w/2$ tel que $\phi_i(j) = \psi_i(j)$, retourner en 2.
4. Pour tout j entre 1 et $w/2$, on génère un octet aléatoire, dont le reste de la division euclidienne par 4 donne la fonction τ_i^j (ET, OU, NON-ET et NON-OU, pour, respectivement, 0, 1, 2 et 3).

La condition testée à l'étape 3 est réalisée, en moyenne, une fois sur $e \approx 2,7$ essais (elle est équivalente à trouver la permutation aléatoire $\psi_i \circ \phi_i^{-1}$ sans point fixe ; voir [53] pour plus de détails). Cette condition est prévue pour que chaque bit d'entrée de la fonction de confusion d'un tour serve d'entrée à deux fonctions τ_i^j différentes.

3.4.3 Implantation pratique de FBC

FBC est conçu pour être implanté en logiciel en utilisant les techniques de code orthogonal ; en effet, dans ce cas, les permutations ϕ_i et ψ_i ont un coût nul. Les fonctions τ_i^j correspondent à des fonctions natives du processeur, ou sont émulables pour un coût faible, de même que le OU EXCLUSIF de sortie de chaque tour.

Comme on veut chiffrer des blocs de taille unitaire importante (au moins 512 octets, soit 4096 bits), on va effectuer, de façon naturelle, un certain nombre de chiffrements en parallèle ; le bon usage du *bitslice* nous impose l'utilisation du mode ECB (comme *Electronic Code Book*), c'est-à-dire que les différents blocs de taille w sont chiffrés indépendamment les uns des autres ; ceci implique que deux blocs de taille w identiques donneront des chiffrés identiques, ce qui est détectable par simple observation du texte chiffré. Comme les données habituellement manipulées sont souvent partiellement redondantes, cette répétition de blocs peut survenir avec une fréquence très supérieure à ce qu'on obtiendrait pour des données purement aléatoires. La parade habituelle est d'utiliser un mode « compteur » : on combine chaque bloc avec un compteur, qui est le numéro logique du bloc, par une addition ou un OU EXCLUSIF. Cette opération est réversible, rapide, et permet de diversifier les textes clairs.

Pour ce qui est de l'orthogonalisation des données, qui est une opération relativement coûteuse, on peut l'éviter : en effet, si on manipule des blocs de w bits sur une machine disposant de registres de n bits, l'orthogonalisation est une permutation fixe et publique d'un bloc de wn bits. Si $w = 64$ et $n = 64$ (cas typique d'un chiffrement par blocs de 64 bits et d'un PC ou station de travail moderne), alors cette permutation agit sur des blocs de 4096 bits, c'est-à-dire exactement la taille d'un secteur sur le disque. Il suffit dès lors de définir le format de stockage sur le disque dur comme étant le mot de 4096 bits déjà permuté, ce qui, dans les faits, annule le coût de cette permutation. On voit déjà une telle idée dans la conception de l'algorithme *Serpent*[2], candidat à l'AES optimisé pour une sorte de *bitslice* interne. On notera que le mode « compteur » évoqué plus haut doit alors travailler sur des données permutées, ce qui reste simple dans le cas où la combinaison est un OU EXCLUSIF.

Une implantation effective de FBC utiliserait un outil similaire à `bsc` (cf. section 3.3) pour transformer la clé en un code exécutable effectuant le chiffrement ou le déchiffrement. Cette opération est relativement longue, à cause du temps de production du code binaire optimisé, et peut nécessiter l'accès à des outils (tels qu'un compilateur C) qui devraient, en théorie, être stockés sur le disque dur, donc eux-mêmes chiffrés. En conséquence, même si la diversification de la clé n'a lieu que lors du lancement de la machine, il est rentable de stocker le résultat de cette opération, c'est-à-dire le code binaire effectuant le chiffrement et le déchiffrement, sous une forme chiffrée sur le disque dur lui-même. Aussi, le plan général d'un logiciel implantant le chiffrement d'un disque dur à l'aide de FBC serait le suivant :

- Une zone du disque dur, de quelques dizaines de kilo-octets, est réservée pour stocker le résultat de la diversification des clés¹.
- Lors de l'installation du logiciel de chiffrement, la clé est choisie, et le code implantant FBC avec cette clé est produit.
- Le disque est entièrement chiffré (ou déclaré comme tel, s'il n'a jamais stocké de données).
- Le code binaire de FBC est chiffré avec un algorithme standard, tel que l'AES[69], et écrit dans la zone réservée. La clé utilisée est dérivée d'un mot de passe.
- Un code permettant à l'utilisateur de rentrer ce mot de passe et de déchiffrer le code binaire de FBC est écrit lui aussi dans la zone réservée. Ainsi, au lancement de l'ordinateur, l'utilisateur entre son mot de passe, ce qui permet le déchiffrement du reste du disque dur et la poursuite

¹Incidemment, une telle zone, habituellement inutilisée, existe sur l'immense majorité des PC, par suite du schéma usuel de partitionnement des disques durs.

de la procédure. Des variantes incluant une carte à puce peuvent être construites sur le même principe.

On notera que le fait que le code de (dé)chiffrement soit embarqué sur le disque dur lui-même a comme conséquence que la connaissance du générateur pseudo-aléatoire utilisé pour la diversification des clés n'est pas nécessaire pour utiliser le disque en lecture et en écriture. Voilà pourquoi les détails de ce générateur peuvent être laissés à la libre appréciation du concepteur du logiciel, pourvu que ce générateur soit cryptographiquement solide. L'interopérabilité est néanmoins réduite avec des architectures différentes : en effet, le code binaire est très spécifique à une famille de processeurs, et au système d'exploitation utilisé. On peut corriger ce défaut en chiffrant une version « abstraite » du système de chiffrement, analogue au code source manipulé par `bsc`. Ceci signifie qu'un générateur de code devra être exécuté pour chaque utilisation d'un disque chiffré sur une architecture différente ; ce traitement est spécifique aux disques amovibles, qui sont de performances mécaniques moindre qu'un disque inamovible, et peuvent donc se contenter d'un code peu optimisé (et donc générable rapidement avec un logiciel simple).

Du point de vue des performances, une instance de FBC travaillant avec des mots de 64 bits ($w = 64$) et 64 tours ($r = 64$) sur un Alpha 21164 peut chiffrer à une cadence de 2 cycles d'horloge par bit traité, dans un contexte mettant en jeu `bsc` et le compilateur C vendu par Compaq ; sur un Alpha 21264, ce temps tombe à 1,3 cycle par bit de données, ce qui est proche du temps souhaité (1 cycle par bit de données). La différence entre le 21164 et le 21264 s'explique par une limitation du premier de ces deux processeurs : lors du deuxième cycle après un accès mémoire en écriture, aucun accès en lecture ne peut avoir lieu. Or, le code orthogonal utilise plus de registres qu'il n'y en a sur le processeur, donc la mémoire doit être utilisée comme lieu de stockage temporaire, ce qui implique de nombreux accès, qui sont normalement effectués en parallèle des opérations logiques bit à bit (l'Alpha peut effectuer quatre opérations simultanément à chaque cycle sous certaines conditions). La contrainte sur les accès en lecture remet en cause ce parallélisme et diminue fortement l'efficacité du 21164. L'Alpha 21264 n'est pas sujet à cette contrainte, ce qui explique ses meilleures performances.

Le compilateur C invoqué par `bsc` n'est pas utilisé « normalement », c'est-à-dire que le code C généré n'a pas grand-chose à voir avec le code C typique que le compilateur traite habituellement. Un générateur de code spécialement optimisé pour le cas de FBC devrait permettre d'obtenir de meilleures performances et mener à une version opérationnelle de FBC.

3.4.4 Sécurité de FBC

La sécurité de FBC est essentiellement heuristique ; elle repose en partie sur l'idée que la plupart des attaques ont un coût exponentiel en le nombre de tours attaqués. Le grand nombre de tours de FBC devrait lui donner une résistance suffisante, même si la fonction utilisée à chaque tour n'est pas d'une grande solidité. Lors de CHES'2001, Adi Shamir a fait remarquer qu'un bit de données rentrant dans une fonction τ_i^j n'avait qu'une chance sur deux d'influer sur la sortie de cette fonction ; aussi, sur deux tours, une modification d'un bit de l'entrée avait une chance sur quatre d'induire une modification d'un bit seulement de la sortie. En utilisant cette propriété, on peut construire un distingueur de FBC d'une permutation aléatoire en utilisant 2^r couples clair/chiffré choisis : on chiffre des couples de textes clairs dont la différence est de seulement un bit, et on compte les cas où les deux chiffrés diffèrent également d'un bit. Cette probabilité est plus élevée que dans le cas où on dispose d'une permutation aléatoire, si r est inférieur à w . Si on dispose d'au moins autant de tours que de bits dans chaque bloc, alors ce distingueur ne fonctionne pas.

L'autre pan de la sécurité de FBC réside dans l'utilisation d'un générateur pseudo-aléatoire lors de la diversification des clés ; beaucoup d'attaques sont fondées sur des relations entre différents bits de clé intervenant à plusieurs endroits du système de chiffrement. Dans le cas de FBC, les multiples interventions de la clé dans l'algorithme sont reliées par le générateur, qu'on suppose cryptographiquement fort, c'est-à-dire qu'il est computationnellement impossible pour l'attaquant de deviner avec une probabilité substantiellement différente de $1/2$ un des bits produits par le générateur, même en connaissant tous les autres. D'une certaine manière, FBC est un schéma d'encodage de données, la véritable sécurité étant fournie par le générateur pseudo-aléatoire. Comme on l'a vu dans la section précédente, le générateur utilisé peut être changé à volonté, et, n'étant invoqué que lors de l'installation du disque dur, il peut être très lent, sans que cela affecte les performances de l'ordinateur après coup. On peut donc utiliser un générateur prouvé équivalent à un problème algorithmique difficile, tel que celui proposé par Blum, Blum et Shub[15].

On pourra enfin remarquer que le cadre d'utilisation de FBC permet de mieux cerner les attaques possibles. En effet, la quantité de texte clair ou chiffré connue de l'attaquant est, par nature, limitée à la taille du disque chiffré.

3.4.5 Vérification de l'intégrité d'un disque dur

Nous avons vu, dans les sections précédentes, dans quelles conditions on pouvait chiffrer le contenu d'un disque dur. Un tel chiffrement décourage les attaques passives, où l'attaquant tente d'apprendre quelque chose sur les données stockées sur le disque. Mais on peut imaginer un scénario où l'attaquant a un accès temporaire et discret au disque, et peut modifier son contenu. Dans ces conditions, une action destructive n'est pas contrainable, l'attaquant pouvant remplacer le contenu entier du disque par des zéros. On voudrait néanmoins être assuré qu'une telle action serait détectée immédiatement : si l'attaquant modifie, même de façon aléatoire, le contenu d'un fichier de données, cette modification doit être connue du propriétaire légitime des données au plus tard au moment où il veut s'en servir.

Aucun logiciel existant de chiffrement de disque dur ne fournit cette fonctionnalité. Il existe des logiciels permettant de calculer et stocker sur un autre périphérique des hachés des fichiers d'un disque dur ; le plus connu est Tripwire[82]. Mais cette méthode a plusieurs défauts dans le cas qui nous intéresse ici, à savoir celui d'un ordinateur portable :

- La vérification doit être effectuée lors du lancement de la machine, et peut prendre un temps important ; en fait, le contenu entier du disque doit être analysé, ce qui peut prendre plus d'une heure. Un tel temps n'est acceptable que pour un serveur, qui n'est relancé que sur une base mensuelle plutôt que quotidienne.
- Encore plus critique, les hachés des fichiers doivent être recalculés lors de l'arrêt de la machine ; cela ne concerne que les fichiers qui ont été modifiés pendant la session, mais la simple opération de rehashage de ces fichiers peut prendre un temps de l'ordre de plusieurs minutes ; l'utilisateur n'est que rarement prêt à consacrer ce temps, surtout si l'arrêt de l'ordinateur a été forcé par un épuisement des batteries ou un dysfonctionnement du système d'exploitation.
- Si un fichier est modifié, son haché complet doit être recalculé. Or, certains fichiers de grande taille sont souvent modifiés par ajout de données à la fin : ce sont les enregistrements de messages du système. Il est coûteux de relire, et hacher, un fichier de plusieurs méga-octets parce qu'une dizaine d'octets ont été rajoutés à la fin.
- Sur un ordinateur portable, les hachés seront placés à même le disque dur : l'utilisateur ne dispose pas, en général, d'un périphérique amovible et sécurisé, l'hypothèse de base étant que son périphérique portable, l'ordinateur lui-même, est potentiellement compromis. Il faut donc que ces hachés soient des MAC (*Message Authentication Code*), c'est-à-dire des hachés calculables et vérifiables seulement par le possesseur d'une

certaine clé secrète.

Un MAC couvrant l'ensemble du disque dur impose le calcul et la vérification de ce MAC à chaque lancement et à chaque arrêt de la machine, ce qui n'est pas réaliste. On voudra donc un MAC ayant les caractéristiques suivantes :

- Chaque fichier possède son propre MAC.
- Chaque MAC est rapide à calculer (le processeur est déjà bien occupé pour le chiffrement des données).
- L'ajout de données à la fin d'un fichier n'impose pas de relire le contenu préalable de ce fichier.

On propose ici une solution qui suppose qu'on chiffre le disque dur à l'aide d'un système de chiffrement par blocs en mode ECB, par exemple FBC. Soit E cet algorithme de chiffrement ; il agit sur des blocs de taille normalisée, qu'on peut numéroter séquentiellement. On va effectuer un double chiffrement de chaque bloc, de la façon suivante : le chiffré C du bloc P numéro N est $C = E(E(P) \oplus N)$. Le OU EXCLUSIF des blocs clairs formant un fichier est écrit sur le disque dur dans une zone spécifique à chaque fichier (cette structure existe sur la plupart des systèmes d'exploitation, incluant MacOS et Windows 2000, et s'appelle un *inode*) ; cette zone étant elle aussi chiffrée, ce OU EXCLUSIF est chiffré et est le MAC recherché.

L'attaquant ne peut pas modifier le fichier avec une probabilité non négligeable de rester indétecté, sous réserve que le système de chiffrement est parfait, c'est-à-dire indistinguable d'une permutation aléatoire ; en effet, le OU EXCLUSIF avec le numéro de bloc l'empêche de déplacer un bloc en dehors de son emplacement normal sans modifier de façon incontrôlée son contenu. Le grand avantage de ce MAC sur les systèmes à base de fonctions de hachage classiques est que lors de la modification d'un fichier, le nouveau MAC peut être calculé en fonction de l'ancien en relisant les données écrasées par la modification. Or, dans la pratique, l'immense majorité des modifications de fichiers est constituée soit de troncations à une taille nulle, soit d'ajouts à la fin du fichier. Dans les deux cas, la mise à jour du MAC ne nécessite aucune lecture supplémentaire (l'écriture dans l'*inode* est effectuée de toute manière, car cette structure contient les champs enregistrant la date de dernière modification du fichier). Par ailleurs, on voit que le OU EXCLUSIF avec le numéro de bloc réalise le mode « compteur » discuté dans la section 3.4.3. On obtient donc un chiffrement et une vérification d'intégrité couplés.

D'un point de vue opérationnel, le système d'exploitation doit mettre à jour le MAC à chaque modification, mais il doit aussi se garder d'utiliser un fichier tant que son MAC n'a pas été vérifié ; cette vérification peut être effectuée de façon asynchrone par un utilitaire lancé automatiquement à l'allumage de la machine, communiquant avec le système d'exploitation par

un protocole de verrouillage afin de savoir quels sont les fichiers à vérifier prioritairement.

Le grand désavantage de ce MAC est qu'il impose un double chiffrement, ce qui double le temps de calcul. L'usage d'un algorithme très performant tel que FBC en est d'autant plus important.

3.5 Conclusion

Nous avons vu, dans ce chapitre, comment la représentation orthogonale des données permet d'atteindre de meilleures performances sur des processeurs standard et polyvalents, surtout dans le cas du chiffrement symétrique. Des outils et méthodes permettant de travailler sur ce code *bitslice* ont été décrites. Ces recherches pourraient jouer un rôle important dans le futur, car les améliorations matérielles sont de plus en plus coûteuses en temps de développement, et la tendance générale de l'informatique durant ces vingt dernières années est de confier de plus en plus de responsabilités à un logiciel spécifique tournant sur un matériel générique, afin de diminuer les coûts de structure en augmentant les séries de production. On constate le même phénomène dans le matériel lui-même : de plus en plus de circuits sont remplacés par des circuits reprogrammables (FPGA) à la conception et à l'interface homogènes. Nous verrons dans le chapitre suivant comment les FPGA sont également de bons candidats à l'implantation de cryptosystèmes.

Chapitre 4

FPGA

4.1 Présentation générale d'un FPGA

FPGA signifie *Field Programmable Gate Array*, en français « tableau de portes programmables ». D'une manière très synthétique, il s'agit de processeurs reprogrammables, c'est-à-dire de circuits logiques dont le dessin, les fonctions logiques et le routage des données sont paramétrables logicielle-ment, sans limitation du nombre d'essais.

Ce genre de matériel est la pierre philosophale des planteurs matériels. En effet, lors de la conception d'un ASIC, le temps de prototypage est très long (de l'ordre de plusieurs semaines pour obtenir un prototype à partir d'un schéma) et chaque essai implique des coûts non négligeables en matériel (car cela revient à la fonte d'une puce, qui ne servira jamais qu'à effectuer des tests). Une puce reconfigurable à volonté, sans passer par une interaction avec une usine de fabrication de puces électroniques, permet de réduire considérablement le temps (et donc les coûts) de conception d'un nouveau processeur. De plus, si une puce donnée peut servir à la conception de plusieurs fonctions, alors cette puce sera utilisée à de nombreux exemplaires. Cette harmonisation des modèles permet d'augmenter les séries de production, donc de réaliser de substantielles économies d'échelle. Ces avantages financiers ont poussé à l'utilisation massive de FPGA, et on estime que dans le domaine des télécommunications, plus des trois quarts des puces utilisées sont des FPGA.

Ces considérations, fondamentalement, n'intéressent que peu le chercheur en cryptographie. En revanche, la possibilité de reconfigurer un matériel en cours d'usage lui ouvre des horizons fort larges ; on peut commencer à imaginer un algorithme de chiffrement fortement dépendant de la clé, dont la mise à la clé impliquerait une reconfiguration d'un FPGA. Par exemple, une

permutation de bits n'étant, en matériel, qu'un banal problème de routage de données (à résoudre lors de la compilation du schéma de la puce, et n'ayant aucun coût lors de son utilisation), on peut envisager d'user à foison de telles permutations dépendantes de la clé, la mise à la clé étant la compilation d'un nouveau schéma et son chargement sur le FPGA. Par ailleurs, le cryptographe peut, grâce aux FPGA, effectuer du développement d'algorithmes et d'attaques cryptanalytiques pour un coût restreint et dans une fenêtre temporelle compatible avec le travail d'un chercheur isolé.

Je vais décrire dans ce chapitre une carte FPGA que j'ai utilisée pour concevoir un système de recherche exhaustive de clés DES.

4.2 La carte Pamette

La carte Pamette est une carte PCI pour station de travail de type PC ou Alpha, comportant cinq puces FPGA provenant de chez Xilinx[88], le leader mondial en la matière. Elle est issue des travaux de l'ancienne équipe de DEC (*Digital Equipment Corporation*), basée à Paris ; cette équipe a ensuite été dissoute, puis DEC a été rachetée par Compaq ; le développement et la production des cartes Pamette ont été interrompus, et ces cartes ne sont plus en vente. En revanche, une firme écossaise, Nallatech[58], a mis sur le marché une série de cartes similaires dans le principe, et utilisant elles aussi des FPGA de Xilinx ; ce qui est dit dans ce chapitre s'applique sans trop de problèmes à ces nouvelles cartes, pourvu qu'on prenne en compte l'avancée technologique permanente.

4.2.1 Vue d'ensemble

Une carte Pamette se présente sous la forme d'une carte PCI standard, pourvue des éléments suivants :

- Cinq puces FPGA Xilinx ; il existe plusieurs générations de telles puces, et, de même, il existe plusieurs versions de la carte Pamette.
- De la mémoire statique, sous la forme de deux modules de 128 Ko chacun, connectés à deux des FPGA.
- Une interface PCI classique, prise en charge par un des cinq FPGA.

Il y a eu de nombreuses variations sur ce même thème, avec différents modèles de FPGA, en des nombres variés, et des interfaces directes supplémentaires, afin de connecter (par exemple) une entrée vidéo ou réseau directement sur les FPGA, sans passer par le bus PCI.

La figure 4.1 illustre ce schéma de principe. Dans le cas de la carte FPGA que j'ai utilisée pour implanter DES, il y a cinq FPGA XC4020 de chez

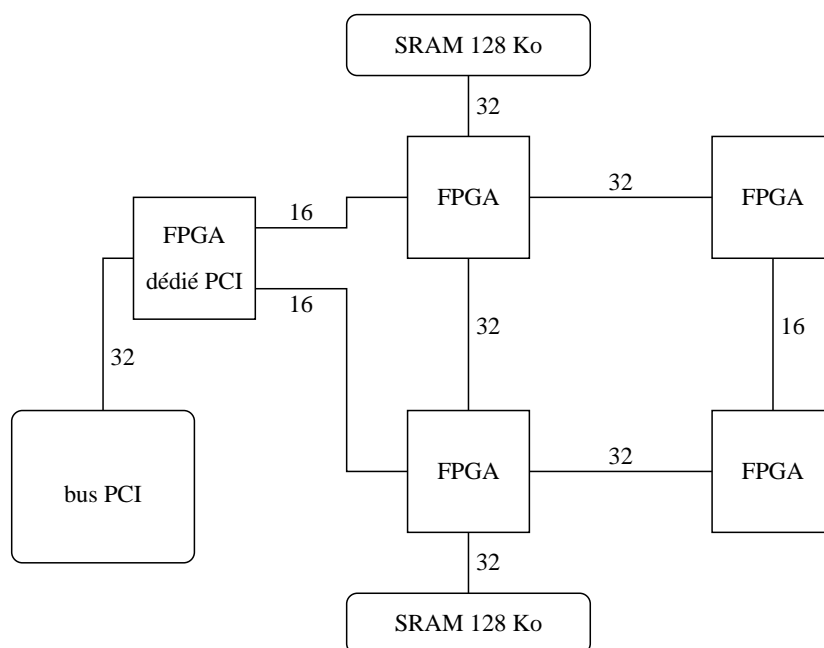


FIG. 4.1 : Schéma d'une carte Pamette

Xilinx; l'un d'eux est préchargé avec un schéma de gestion du bus PCI et n'est pas utilisable pour les besoins du programmeur. Sur les quatre autres, deux sont reliés par un bus de largeur 32 bits avec le FPGA contrôlant le bus PCI et avec les deux blocs de mémoire; ils sont par ailleurs reliés entre eux par un bus de largeur 32 bits. Les deux autres FPGA sont reliés chacun à un des deux précédents par un bus de largeur 32 bits, et sont connectés entre eux via un bus de largeur 16 bits de large. D'autres bus sont également présents sur la cartes : ceux qui servent au chargement des schémas dans les FPGA, et deux horloges, l'une cadencée au rythme du bus PCI (à 33 MHz), l'autre réglable (de 1 à 66 MHz).

La programmation s'effectue via les outils Xilinx, auxquels on fournit une *netlist* (liste des fonctions logiques à réaliser et des fils logiques reliant ces fonctions). Afin de faciliter la production de ces netlists, l'équipe conceptrice de la Pamette a programmé PamDC[81], une bibliothèque en C++, qui permet d'exprimer les fonctions logiques à l'aide des opérateurs habituels du C++, et de les générer au sein d'un programme, ce qui donne une grande souplesse de développement. La compilation en elle-même de la netlist par les outils Xilinx est assez longue (suivant le schéma compilé, elle peut prendre plusieurs heures, car le problème du placement de la logique et du routage des données est un problème complexe pour lequel on ne connaît pas de so-

lution computationnellement vraiment satisfaisante), mais le chargement du nouveau schéma est rapide (moins d'une seconde).

La carte ainsi programmée peut tourner à diverses fréquences, suivant la complexité du schéma, jusqu'à 66 MHz. L'interface PCI permet des accès rapides avec la mémoire centrale de la machine, à des débits forts honorables (de l'ordre de la bande passante du bus PCI lui-même, soit plus de 100 Mo par seconde), et on dispose aussi d'un bus plus lent mais nettement plus facile à mettre en œuvre (car une erreur de programmation sur le bus PCI peut entraîner un arrêt total de la machine hôte, ce qui rallonge le temps de développement).

4.2.2 FGPA Xilinx

Les FPGA de chez Xilinx sont organisés autour de la notion de CLB (*Configuration Logic Block*) : il s'agit d'une brique élémentaire, capable de quelques fonctions logiques reprogrammables. Le FPGA lui-même est un treillis rectangulaire de CLB, interconnectés par une matrice de bus dont les croisements sont tous des aiguillages paramétrables logiciellement. La figure 4.2 montre cet agencement ; la périphérie de la puce est constituée d'une série de portails de communication sur lesquels se branchent les bus extérieurs. Le FPGA XC4020 comporte $28 \times 28 = 784$ CLB, mais les modèles plus récents (de la série Virtex) comportent plus de 10000 CLB.

Le contenu de chaque CLB est le fruit d'une longue série d'expérimentations ; dans la série des XC40x0, Xilinx s'est stabilisé sur l'idée que la sous-unité utile et polyvalente est la fonction booléenne générique acceptant quatre entrées et proposant une sortie. Il existe 65536 telles fonctions, et on peut les implanter toutes avec une LUT (*Look-Up Table*) de 16 bits. Le CLB de la série Xilinx XC40x0 est illustré par la figure 4.3 ; on voit que ce CLB comporte les éléments suivants :

- deux fonctions booléennes génériques $4 \rightarrow 1$,
- une fonction booléenne générique $3 \rightarrow 1$,
- deux registres de capacité 1 bit,
- des multiplexeurs reliant ces divers éléments.

Au total, ce CLB dispose de 9 entrées principales, et de 4 sorties, deux d'entre elles étant reliées sur la sortie des registres, et les deux autres étant reliables sur la sortie des trois fonctions booléennes génériques. Chaque registre est un circuit fournissant sur sa sortie ce qu'il avait en entrée au cycle précédent ; un signal EN (*Enable*) permet de contrôler ces registres : si ce signal est à un, les registres enregistrent un nouveau contenu à la fin du cycle ; s'il est à zéro, les deux registres proposeront pendant le cycle suivant la même sortie que pendant le cycle courant. Le découpage en cycles est lui-même rythmé

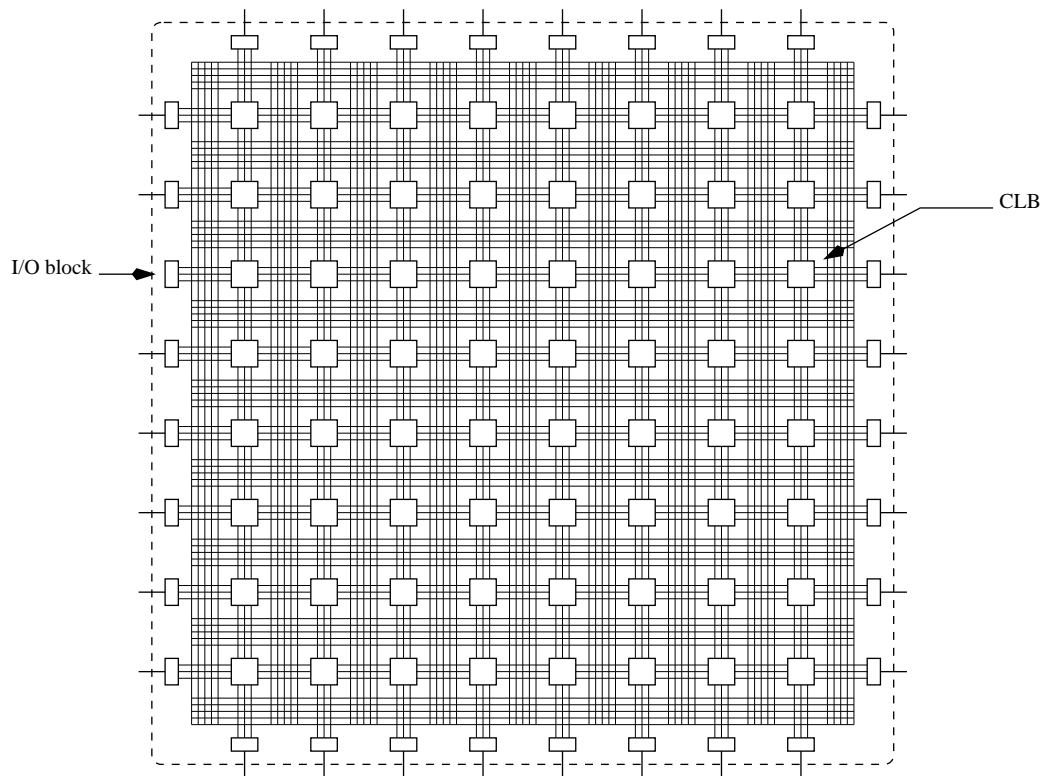


FIG. 4.2 : Schéma d'un FPGA Xilinx

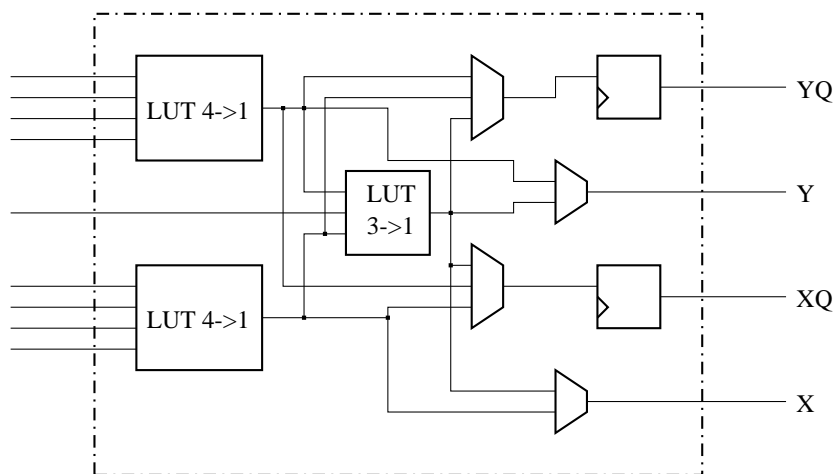


FIG. 4.3 : Schéma d'un Configurable Logic Block

par une des deux horloges système.

Les FPGA plus récents de chez Xilinx (Spartan, Virtex...) utilisent des CLB légèrement différents : la fonction $3 \rightarrow 1$ a disparu, et deux nouvelles fonctions $4 \rightarrow 1$ sont apparues. De plus, certains des CLB ont été remplacés par des blocs de mémoire directement inclus dans le corps du FPGA.

4.2.3 Programmation

Ce qui doit être envoyé à la carte Pamette est, fondamentalement, constitué de quatre schémas décrivant la programmation des quatre FPGA, au format *bitstream* standard (défini par Xilinx). Ces schémas sont générés par les outils logiciels fournis par Xilinx à partir d'une description du circuit sous la forme d'une *netlist*, qui est (en gros) une liste exhaustive, porte logique par porte logique et fil par fil, assortie d'éventuelles contraintes de placement (qui servent à guider le travail du compilateur Xilinx).

Dans le cas de la Pamette, la génération de ces *netlists* peut être effectuée grâce à PamDC : les opérateurs standard du C++ sont syntaxiquement surchargés pour exprimer les opérations élémentaires réalisables par un FPGA. Voici un fragment de code utilisant cette bibliothèque :

```
class Foo : public Node {
public:
    WireVector<Bool, 8> carry;

    Foo(void) : Node("F00")
    {
        internal(carry);
    }
    void logic(WireVector<Bool, 8> &in_a,
              WireVector<Bool, 8> &in_b,
              WireVector<Bool, 9> &out)
    {
        int i;

        for (i = 0; i < 8; i ++) {
            if (i == 0) {
                carry[i] = in_a[i] & in_b[i];
                out[i] = in_a[i] ^ in_b[i];
            } else {
                out[i] = in_a[i] ^ in_b[i] ^ carry[i - 1];
                carry[i] = (in_a[i] & in_b[i])

```

```

        ^ (in_a[i] & carry[i - 1])
        ^ (in_b[i] & carry[i - 1]);
    }
}
out[8] = carry[8];
}
};

```

Ce code définit une classe qui calcule un additionneur de deux valeurs de 8 bits, à résultat sur 9 bits. Cette classe définit les fils `carry` de la *netlist* ; ses deux entrées de 8 bits et sa sortie de 9 bits doivent être définies dans d'autres classes. La définition des fonctions logiques n'a pas à être optimale en termes de portes logiques, pour deux raisons :

- le substrat du circuit, c'est-à-dire le FPGA, possède une certaine granularité, à savoir les CLB ; ce qui est optimal au niveau du transistor ne l'est pas forcément pour un FPGA ;
- le compilateur Xilinx réécrit les fonctions logiques en établissant la liste exhaustive des valeurs de sortie en fonction des valeurs de l'entrée, donc toutes les expressions fonctionnellement équivalentes donnent le même résultat.

Le schéma est programmé sous la forme d'un fichier C++ complet ; une fois compilé, ce programme est lancé, et il produit les *netlists* requises par les compilateurs Xilinx. La compilation des *netlists* est une opération très longue, qui peut prendre plusieurs heures : la raison en est que le placement de la logique et le routage des données est un problème complexe, pour lequel on ne connaît pas d'algorithme polynômial ou même sous-exponentiel. On peut aider le compilateur en ajoutant des directives de placement dans le code, qui consistent à dire : « cette fonction logique est réalisée dans tel CLB ». Si les FPGA sont à peu près complètement utilisés, on est plus ou moins dans l'obligation de placer toutes les fonctions logiques, si on veut terminer la compilation en un temps humainement acceptable.

4.3 DES-Cracker

La construction d'une machine spécialisée, capable de cryptanalyser un chiffrement DES par recherche exhaustive de la clé, est une idée qui a été évoquée à de nombreuses reprises[44, 30, 37, 87], avec des estimations de son coût et des schémas de principe sur sa structure ; ces perspectives étaient une illustration du fait (supposé) que certaines organisations pas forcément

très fortunées (notamment la NSA¹) sont en mesure de décrypter des messages chiffrés avec DES de façon industrielle. Une telle machine a fini par être construite par l'EFF (*Electronic Frontier Foundation*)[36] pour un coût « faible » (de l'ordre de 250 000 dollars américains, coûts de développement compris). L'EFF est un groupe de pression américain qui veut promouvoir la liberté d'expression, le droit au chiffrement, le logiciel libre et autres sujets habituellement considérés comme « libertaires ». Leur champ de bataille actuel est le chiffrement des DVD et les actions légales correspondantes. L'EFF entendait démontrer par la réalisation d'un « DES-cracker » que les lois américaines restreignant l'export et l'import de produits de chiffrement empêchaient l'usage de produits suffisamment sûrs, et, partant, que le gouvernement américain avait délibérément restreint la liberté des citoyens et entreprises américains afin de mieux les espionner. Cette conclusion est bien sûr sujette à caution, mais la machine de l'EFF existe indéniablement.

Dans cette section, je vais détailler quelques points relatifs à la construction d'un système de recherche exhaustive de clés de chiffrement, et décrire une implantation d'un tel système que j'ai réalisé en 1998 sur une carte Parnette ; ce système est capable de retrouver une clé DES de 40 bits en quelques heures sur une unique carte. Une implantation similaire sur les mêmes puces Xilinx a été conçue par Jens-Peter Kaps et Christof Paar[48] indépendamment et à la même époque ; la mienne consomme environ 20% de CLB de moins pour le même nombre de clés essayées par seconde. Sur d'autres types de FPGA, d'autres implantations de DES ont été effectuées[39, 64].

4.3.1 Structure générale

La recherche exhaustive d'une clé de chiffrement est un problème à part. En effet, la plupart des traitements informatiques sont modélisables sous la forme d'un flux de données, flux qui est traité successivement par plusieurs modules disposés linéairement sur le chemin des données. C'est le cas d'un chiffrement conventionnel de données à destination d'un réseau ou d'un stockage de masse : les blocs à chiffrer arrivent un par un dans le bloc chiffreur, qui ne s'occupe ni de l'extraction de ces blocs dans le flux d'entrée, ni de la répartition de ces blocs sur le support d'arrivée. Pour augmenter le débit, dans le cas d'une réalisation matérielle du module, on a tendance à pratiquer des stratégies de *pipe-line*, c'est-à-dire que l'algorithme de chiffrement est découpé en sous-modules élémentaires, tous présents dans l'implantation matérielle ; chaque bloc de données passe un cycle dans chaque sous-module, mais un nouveau bloc peut rentrer à chaque cycle. Au prix d'une implantation

¹*National Security Agency*, une partie des services secrets américains.

matérielle plus imposante, on peut ainsi augmenter le débit. Les algorithmes de chiffrement tels que DES sont très adaptés à ces conceptions, car leur logique de fonctionnement est indépendante des données, donc ils peuvent travailler à flux constant.

Ces stratégies n'ont pas d'intérêt dans un système de recherche exhaustive. En effet, les données en entrées sont très réduites (un unique bloc chiffré et le bloc clair correspondant), et les données en sortie encore plus (la clé, une fois qu'elle est trouvée). De plus, la recherche est, par nature, intrinsèquement parallèle, donc on peut concevoir des modules implantant chacun un tour du système de chiffrement, chaque module exécutant indépendamment des autres un chiffrement complet grâce à une boucle; on économise ainsi le routage entre les différents modules, ce qui diminue la consommation d'énergie, permet de monter la fréquence de fonctionnement (car cette fréquence est désormais limitée par les temps de communications entre les portes logiques plus que par les temps de commutation des portes logiques elles-mêmes), et augmente la résistance aux pannes (les modules étant séparés, un dysfonctionnement d'un module n'entraîne pas de conséquence néfaste sur le fonctionnement des autres).

Enfin, on peut réduire les communications de sortie au maximum : idéalement, seul un bit de donnée doit être transmis, celui qu'un module émet pour signifier qu'il a trouvé une clé potentielle. Le système de contrôle de la machine, une banale station de travail, n'a qu'à chronométrer le temps mis par ce bit à apparaître, et, grâce à la connaissance de ce temps et du module ayant émis ce bit, peut réduire l'ensemble des clés ayant potentiellement conduit à l'activation de ce bit à quelques millions près, c'est-à-dire un ensemble facilement testable par un programme peu optimisé, en quelques secondes. Ceci est rendu possible par le fait que l'exécution d'un algorithme de chiffrement conventionnel s'effectue en un temps constant, indépendant des données (car ici, il n'y a pas de latence d'arrivée des données, puisqu'il n'y a pas de flux d'entrée). Enfin, les quartz d'horloge sont les composants d'ordinateur les plus précis (la fréquence est souvent garantie à 10^{-7} près), ce qui augmente encore la faisabilité de cette méthode.

Ceci nous amène au schéma du DES-Cracker sur carte Pamette. Les caractéristiques de ce schéma sont les suivantes :

- 16 modules de calcul de DES, répartis à raison de quatre par FPGA ;
- chaque module calcule un chiffrement DES en 16 cycles ;
- la fréquence de fonctionnement est de 25 MHz ;
- un compteur par FPGA met à jour les clés essayées.

Le système essaye donc 25 millions de clés DES à la seconde, et parcourt donc un espace de 2^{40} clés en un peu plus de 12 heures. La clé est trouvée en parcourant, en moyenne, la moitié de l'espace, ce qui correspond à 6 heures

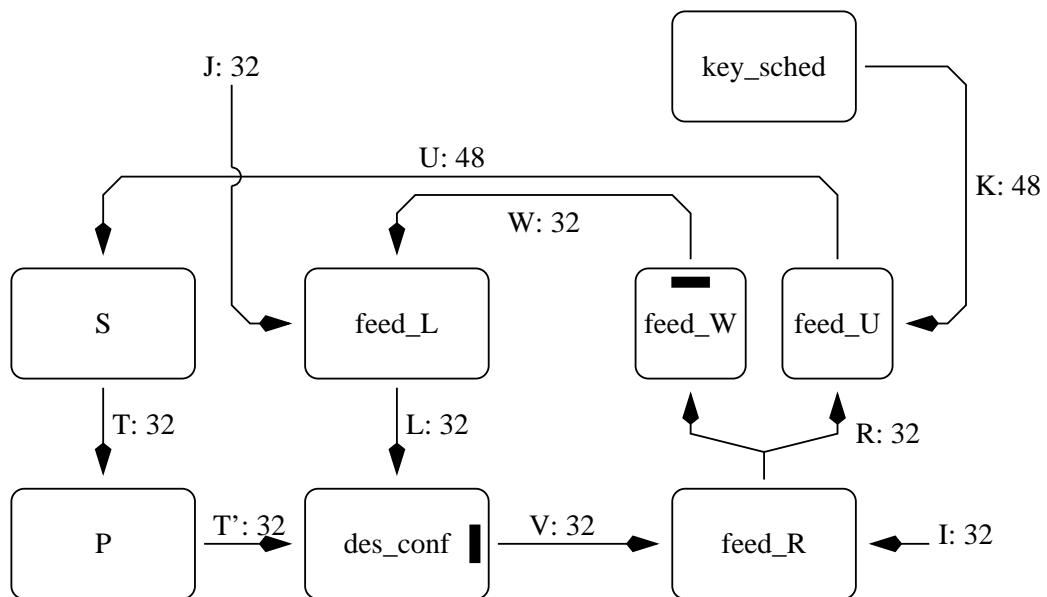


FIG. 4.4 : Schéma de DES sur Pamette

de calcul.

Ces chiffres correspondent à une carte de 1997 ; les FPGA modernes permettraient de placer quatre fois plus de logique et de cadencer le système deux fois plus vite, pour le même prix (de l'ordre de 4 000 EUR).

4.3.2 DES sur carte Pamette

- Un tour de chiffrement DES comporte les parties calculatoires suivantes :
- un OU EXCLUSIF avec la sous-clé du tour ;
 - 8 boîtes S ;
 - un OU EXCLUSIF avec la partie gauche de l'entrée.

Le reste n'est que routage.

La figure 4.4 reproduit la structure logique du code PamDC utilisé pour implanter DES ; voici la fonction de chaque module :

S : ce module calcule l'application des boîtes S sur une entrée U de 48 bits ; la sortie T a une taille de 32 bits.

P : il s'agit de la permutation P en sortie des boîtes S ; ce module ne contient que des informations de routage, c'est-à-dire d'équivalence entre différents éléments de la *netlist*. Son entrée T et sa sortie T' ont une taille de 32 bits.

feed_L : ce module fournit, un cycle sur 16, son entrée fixe J de 32 bits ;

pour les quinze autres cycles, il fournit W à la place. W est la moitié gauche du résultat du cycle précédent ; la sortie de ce module est L et fait 32 bits.

des_conf : ce module fournit V , mot de 32 bits qui est le OU EXCLUSIF des valeurs de L et T' du cycle précédent.

feed_R : ce module est analogue à **feed_L** : sa sortie est la constante I un cycle sur 16, et V le reste du temps.

feed_U : il s'agit de l'expansion E de DES et du OU EXCLUSIF entre l'entrée étendue et la sous-clé de ce tour.

feed_W : c'est un simple module de stockage : sa sortie W de 32 bits est la valeur de son entrée R au cycle précédent.

key_sched : ce module fournit successivement les 16 sous-clés de chiffrement DES.

Voici ce que représentent, dans un chiffrement DES, les différentes valeurs de la figure 4.4 :

- I, J : respectivement, les moitiés droite et gauche du texte clair ; ces valeurs sont fixes pendant toute la durée de la recherche exhaustive.
- L, R : respectivement, les moitiés gauche et droite de l'entrée du tour courant.
- T, T' : la sortie des boîtes S, respectivement avant et après la permutation P.
- K : la sous-clé K de 48 bits du tour courant.
- U : le résultat du OU EXCLUSIF entre la sous-clé K du tour courant et l'expansion de la moitié droite R à 48 bits.
- V : le résultat du OU EXCLUSIF de la sortie de la fonction de confusion et de la moitié gauche d'entrée du tour précédent.
- W : la moitié droite de l'entrée du tour précédent.

Ainsi, le fonctionnement reproduit, à chaque cycle, un tour de DES : la sous-clé K est présentée, de même que les moitiés gauche et droite de l'entrée du tour (L et R). L'entrée R est stockée pour le tour suivant (elle sera fournie sous le nom W), et est également étendue à 48 bits et combinée par OU EXCLUSIF à la sous-clé K . Le résultat est envoyé dans les boîtes S, puis dans la permutation P , puis est enfin combiné par OU EXCLUSIF avec L ; le résultat est stocké dans des registres pour être renvoyé au cycle suivant sous le nom V (ce sera la nouvelle entrée droite du tour). Les modules **feed_L** et **feed_R** ont pour rôle de fournir les sorties du tour précédent en tant qu'entrées, ou de fournir à la place le texte clair préenregistré, si on commence un nouveau chiffrement.

Nous allons voir comment, dans le détails, sont implantées ces différentes fonctions dans les CLB.

Boîtes S

Les boîtes S de DES sont des fonctions génériques prenant 6 bits en entrée et fournissant 4 bits en sortie. On peut trivialement décomposer chacune en quatre fonctions booléennes génériques prenant 6 bits en entrée et ayant une sortie de 1 bit, ce qui nous donne 32 telles fonctions.

La définition de ces fonctions a été effectuée selon des critères de sécurités très précis, et non pour des questions de simplicité d'implantation. Aussi, il n'existe pas d'expression booléenne de ces fonctions suffisamment simple pour permettre une implantation plus efficace que celle de fonctions génériques. En effet, la structure en CLB impose une granularité relativement élevée (l'unité élémentaire est le CLB), et le synchronisme de la structure de DES impose que les évaluations de tous les bits de sortie des boîtes S ne gagnent rien à s'effectuer plus rapidement que la plus lente d'entre elles.

Un CLB peut réaliser n'importe quelle fonction $5 \rightarrow 1$: si f est une telle fonction, on définit f_0 et f_1 ainsi :

$$f_0(x_1, x_2, x_3, x_4) = f(x_1, x_2, x_3, x_4, 0) \quad (4.1)$$

$$f_1(x_1, x_2, x_3, x_4) = f(x_1, x_2, x_3, x_4, 1) \quad (4.2)$$

La fonction f se calcule alors ainsi :

$$f(x_1, x_2, x_3, x_4, x_5) = \text{mux}(x_5, f_0(x_1, x_2, x_3, x_4), f_1(x_1, x_2, x_3, x_4)) \quad (4.3)$$

où $\text{mux}(c, a, b)$ est la fonction multiplexeur, qui renvoie a si $c = 0$, et b si $c = 1$. Autrement dit, on calcule parallèlement, en fonction des quatre premiers bits d'entrée, les résultats possibles de la fonction f suivant la valeur du cinquième bit, et un multiplexeur contrôlé par ce cinquième bit détermine laquelle des deux valeurs est conservée.

En revanche, un CLB seul ne peut calculer qu'un petit sous-ensemble des fonctions $6 \rightarrow 1$ possibles. En effet, il existe 2^{64} telles fonctions, mais le CLB n'est configurable que de 2^{40} façons possibles (les deux fonctions $4 \rightarrow 1$ ont 2^{16} configurations possibles chacune, et la fonction $3 \rightarrow 1$ en possède 2^8). De plus, la définition des boîtes S ne permet pas de partager suffisamment de portions du calcul pour donner des gains substantiels en termes de place occupée sur le FPGA ou de vitesse.

L'implantation d'un bit de sortie d'une boîte S est donc effectuée ainsi : deux CLB calculent les deux valeurs possibles du bit de sortie en fonction des cinq premiers bits d'entrées ; un des CLB renvoie la valeur correspondant au cas où le sixième bit vaudrait 0, et l'autre celle correspondant au cas où le sixième bit vaudrait 1. Ces deux valeurs sont envoyées dans une fonction $4 \rightarrow 1$ d'un CLB, en compagnie du sixième bit d'entrée de la boîte S et du

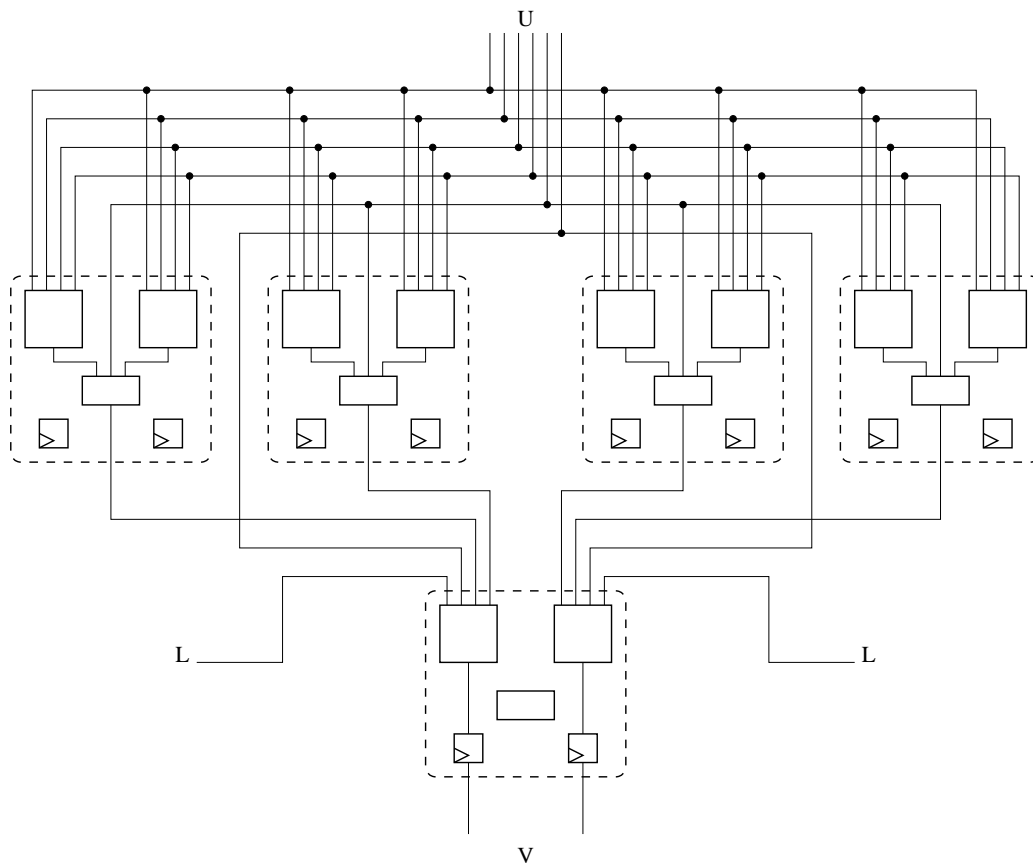


FIG. 4.5 : Calcul de deux bits de sortie d'une boîte S

bit de la partie gauche d'entrée du tour courant qui sera combiné, après la permutation P , par un OU EXCLUSIF, avec ce bit de sortie de cette boîte S. Autrement dit, deux CLB plus la moitié d'un troisième CLB permettent de calculer un bit de sortie du tour courant en fonction des six bits d'entrée de la boîte S concernée, et d'un bit d'entrée du tour courant. On fusionne ainsi les fonctions des modules S, P et `des_conf` explicités plus haut. La figure 4.5 illustre ce mécanisme.

La profondeur de calcul des boîtes S, de l'entrée de ces boîtes jusqu'aux registres stockant la sortie du tour, est donc de deux CLB. On remarquera que les registres du module `des_conf` prennent place naturellement dans les CLB qui servent à la fin du calcul des boîtes S. Cinq CLB calculent deux bits de sortie, il y a 32 bits de sortie, donc il faut 80 CLB pour implanter les modules S, P et `des_conf`.

Génération des sous-clés

Les sous-clés sont générées dans le module `key_sched`. Ce que la figure 4.4 ne montre pas, ce sont les éléments suivants, qui sont partagés entre les différentes instances de DES sur le même FPGA :

- un compteur sur 36 bits, qui avance d’une unité tous les 16 cycles ;
- un compteur sur 4 bits, qui avance d’une unité à chaque cycle, et qui génère les signaux *clk15*, *clk16* et *clk1*.

Les signaux d’horloge ont la signification suivante :

- *clk15* : il prend 1 pour valeur un cycle sur 16, et marque le dernier tour de chiffrement DES ; sa valeur est 0 le reste du temps.
- *clk16* : ce signal a pour valeur exactement celle qu’avait *clk15* lors du cycle précédent.
- *clk1* : la génération de clé DES repose sur deux registres à rotation circulaire de 28 bits, qui avancent soit d’un bit, soit de deux à chaque cycle ; *clk1* vaut 1 quand les deux registres doivent avancer d’un bit pour calculer la sous-clé du cycle suivant, 0 sinon.

L’implantation de ces compteurs sera détaillée dans la section 4.3.3, mais on peut déjà remarquer que si on décale d’un bit les deux registres à la fin du seizième tour (alors qu’il n’y a pas de dix-septième tour à préparer), on a ramené les registres à leur état initial. Le contenu des registres étant une permutation du contenu de la clé, il suffit de modifier ce contenu avec la différence dK bit à bit de deux clés successives pour passer à la clé suivante. Le compteur de 36 bits envoie justement non pas sa valeur interne mais la différence entre cette valeur et la précédente ; le module `key_sched` n’a donc qu’à effectuer un OU EXCLUSIF de cette différence avec le contenu de ses registres à décalage.

L’implantation d’un registre à décalage ne pose pas de problème particulier : chaque bit du registre est stocké dans un registre d’un CLB, et son entrée est branchée sur la sortie d’une des deux fonctions $4 \rightarrow 1$ du CLB. Cette fonction prend pour entrée le signal *clk1*, les sorties des registres stockant les deux bits précédents dans le registre, et la différence à appliquer (par un OU EXCLUSIF) à la valeur à stocker ; cette différence est fournie par le compteur sur 36 bits et vaut 0 au moins quinze tours sur seize. La figure 4.6 illustre cette implantation.

La sous-clé K étant prise en sortie des registres de stockage du registre à décalage, elle doit être mise à jour lors du cycle précédent. Aussi, la différence dK entre une clé et la suivante doit être fournie lorsque *clk15* est à 1.

Les deux registres totalisent une taille totale interne de clé de 56 bits ; il faut donc 28 CLB pour implanter le module `key_sched`.

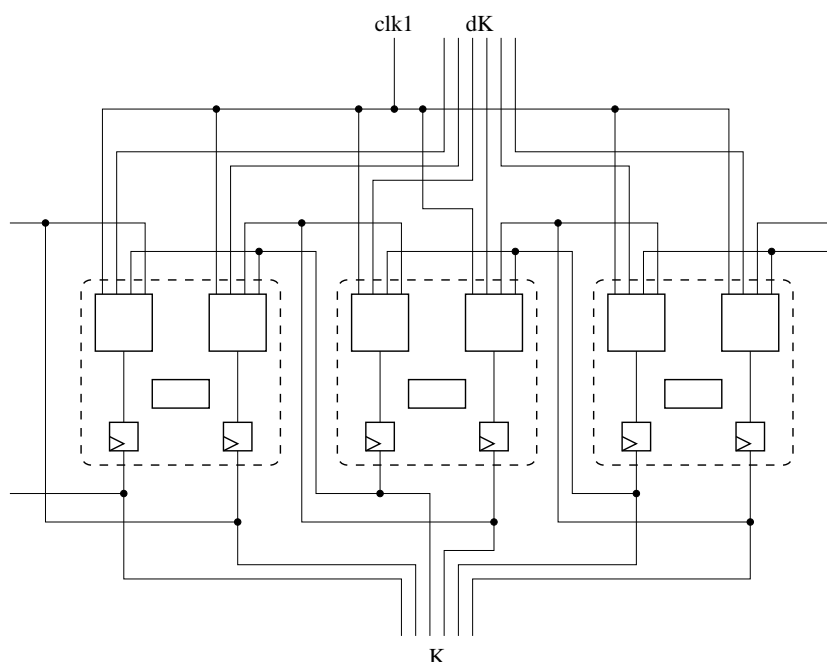


FIG. 4.6 : Génération des sous-clés

Expansion, action de la clé et rebouclage

Les modules `feed_U` et `feed_W` travaillent sur des entrées similaires mais la sortie U n'est pas identique à W (qui est stockée dans des registres), aussi ces deux modules ne peuvent pas être fusionnés entre eux ; en revanche, `feed_R`, qui fournit la partie droite de l'entrée du tour courant, peut être intégré à `feed_W` et `feed_U`.

Chaque bit de U est le fruit de la combinaison de trois bits : un bit de V , qui représente la partie droite de la sortie du tour précédent, un bit de K , la sous-clé du tour courant, et le signal `clk16`, qui indique si le bit d'entrée à utiliser doit être celui de V ou celui du texte clair servant à la recherche exhaustive. Une fonction $4 \rightarrow 1$ d'un CLB suffit amplement à cet usage ; le texte clair est inclus dans la définition de la fonction lors de la compilation du schéma. Les bits de W suivent un principe analogue, à part que seules deux entrées sont nécessaires : V et `clk16`. Là encore, une fonction $4 \rightarrow 1$ et un registre suffisent par bit de W .

U a une taille de 48 bits, et W fait 32 bits ; il faut donc 80 demi-CLB, ou encore 40 CLB pour implanter les modules `feed_R`, `feed_U` et `feed_W`. Quant à `feed_L`, qui fournit les bits de la moitié gauche de l'entrée du tour, il est similaire à `feed_W`, les registres en moins ; il occupe donc 16 CLB supplémentaires.

4.3.3 Mise en œuvre de la recherche

Nous avons vu comment implanter les différents modules permettant de calculer un chiffrement DES ; nous allons étudier ici comment utiliser ces modules dans une recherche exhaustive. Trois modules sont encore nécessaires :

- un compteur qui envoie au module `key_sched` la différence entre l'ancienne clé et la nouvelle, un cycle sur seize ;
- un compteur qui envoie les signaux `clk1`, `clk15` et `clk16` aux modules `feed_L`, `feed_R` et `key_sched` ;
- par instance de DES, un comparateur qui, à chaque fin de chiffrement, compare le texte chiffré obtenu avec celui attendu, et place un drapeau à 1 une fois qu'il a trouvé le bon texte chiffré ; ce drapeau est connecté à l'interface extérieure de la carte Pamette, et surveillé périodiquement par la station hôte.

L'horloge interne

Le générateur des signaux `clk` est la véritable horloge interne du schéma ; c'est l'*hortator* qui donne la cadence à l'ensemble des autres modules. C'est un simple compteur sur quatre bits, qui s'implante en deux CLB ; les quatre registres forment un mot binaire de quatre bits qui est incrémenté à chaque cycle. Si les sorties de ces quatre registres sont nommées a_0 , a_1 , a_2 et a_3 , alors les nouvelles valeurs qui sont présentées en entrée de ces quatre registres, pour stockage à la fin du cycle, sont calculées ainsi :

$$b_0 = \neg a_0 \tag{4.4}$$

$$b_1 = a_1 \oplus a_0 \tag{4.5}$$

$$b_2 = a_2 \oplus (a_1 \wedge a_0) \tag{4.6}$$

$$b_3 = a_3 \oplus (a_2 \wedge a_1 \wedge a_0) \tag{4.7}$$

Trois fonctions $4 \rightarrow 1$ supplémentaires sont requises pour générer les signaux `clk1`, `clk15` et `clk16`. Ces fonctions prennent en entrée les quatre valeurs a_0 , a_1 , a_2 et a_3 , et fournissent en sortie les trois signaux voulus. Les signaux `clk` sont stockés dans des registres, afin d'être disponibles dès le début du cycle ; cela nécessite que le compteur sur quatre bits travaille avec un cycle d'avance : c'est une simple question de valeur des registres à l'initialisation. La figure 4.7 explicite la conception de l'horloge interne.

Le générateur de clés

Ce générateur est un compteur sur 36 bits. En effet, on veut tester 2^{40} clés, et on peut placer quatre instances de DES par FPGA, donc seize en

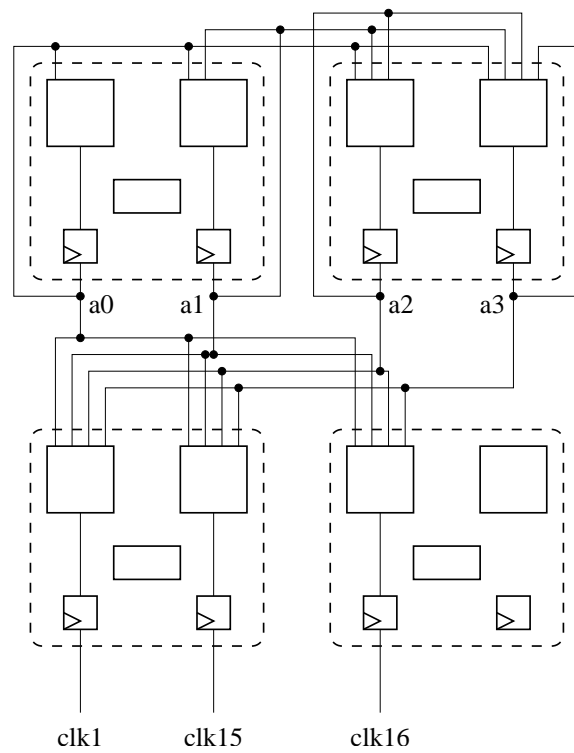


FIG. 4.7 : L'horloge interne

tout sur une carte Pamette. Chaque instance de DES n'a donc à essayer que 2^{36} clés.

La propagation d'une retenue est une opération qui peut s'avérer longue ; on dispose de circuits permettant de calculer, lors d'une addition générique, la retenue de rang n en profondeur $\log n$, mais ces circuits nécessitent beaucoup de portes logiques. Fort heureusement, il n'est pas nécessaire dans notre cas d'appliquer ces méthodes, car l'incrémentación n'a lieu qu'une fois tous les seize cycles. On se contente donc d'une propagation sans histoire.

On peut raisonnablement gérer trois bits du compteur à chaque cycle ; on groupe les bits du compteur en douze groupes de trois, chaque groupe étant géré par deux CLB. Le groupe possède une entrée : la retenue du groupe précédent au cycle précédent, et quatre sorties (stockées dans des registres) : les trois valeurs des trois bits gérés dans le groupe, et la retenue de sortie de ce groupe. Ainsi, si une valeur de 1 est présentée en entrée du premier groupe, suivie pendant les onze cycles suivants de la valeur 0, au bout du douzième cycle, la valeur numérique représentée en binaire par l'ensemble du registre a été incrémentée de une unité. C'est *clk15* qui tiendra le rôle de l'entrée du groupe de poids faible.

Les modules `key_sched` attendent du générateur de clé non pas la nouvelle clé mais la différence avec la précédente, un cycle sur seize (celui où *clk15* vaut 1, c'est-à-dire au moment du dernier tour, car c'est à ce cycle-là que les modules `key_sched` préparent la sous-clé qu'ils présenteront au cycle suivant). Aussi, le générateur doit-il stocker la valeur du compteur tous les seize cycles (la nouvelle valeur est disponible sur le coup de *clk15*), et calculer le OU EXCLUSIF de cette valeur avec l'ancienne. Il faut un CLB entier par bit envoyé : une des fonctions $4 \rightarrow 1$ prend en entrée la valeur du registre, l'éventuelle nouvelle valeur, et *clk15*, et renvoie dans le registre la nouvelle valeur si *clk15* vaut 1, l'ancienne valeur sinon (ceci implante une mémoire mise à jour tous les seize cycles) ; l'autre fonction prend en entrée l'ancienne valeur, l'éventuelle nouvelle valeur, et *clk15*, et renvoie vers les `key_sched` le OU EXCLUSIF de ces deux valeurs si *clk15* vaut 1, 0 sinon.

- Ainsi, à chaque fois que *clk15* vaut 1, il se passe les choses suivantes :
- la nouvelle valeur du compteur est extraite ;
 - la différence entre cette valeur et l'ancienne est envoyée aux modules `key_sched` ;
 - la nouvelle valeur est stockée ;
 - l'incrémentación du compteur est mise en route ; elle sera achevée douze cycles plus tard.

La figure 4.8 illustre un groupe de trois bits du compteur, et les trois CLB gérant la différence de deux clés successives. Le générateur de clés, au total, occupe 60 CLB (24 pour le compteur lui-même, 36 pour le calcul des

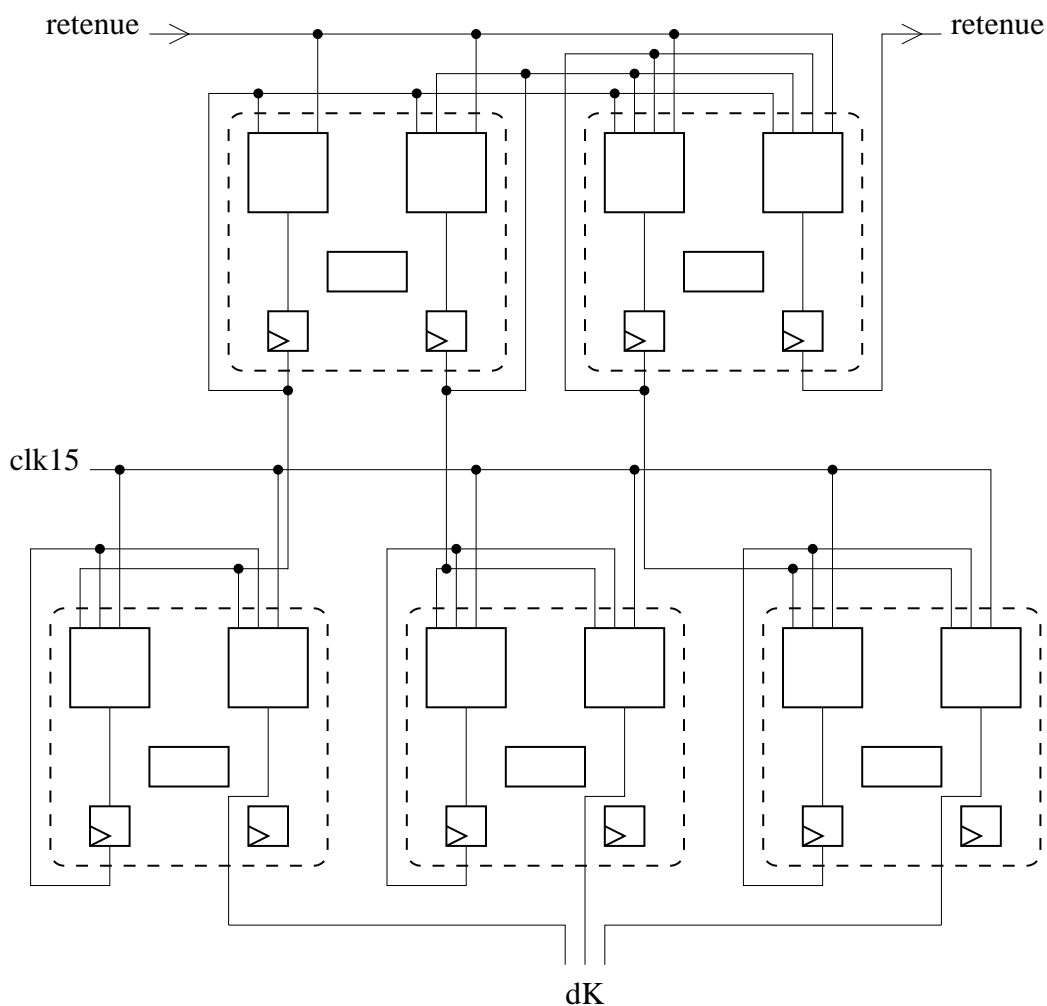


FIG. 4.8 : Le générateur de clés

différences).

Les comparateurs

Chaque module comparateur a pour mission de comparer la sortie de chaque chiffrement DES avec la sortie attendue (le texte chiffré connu). Un CLB peut vérifier huit bits d'un seul coup : chacune de ses deux fonctions $4 \rightarrow 1$ prend en entrée quatre des bits, et place sur sa sortie un 1 si et seulement si les quatre bits d'entrée sont corrects. Les deux sorties des fonctions $4 \rightarrow 1$ sont envoyées dans la fonction $3 \rightarrow 1$ qui calcule leur combinaison par un ET logique. En réalité, une des entrées de la fonction $3 \rightarrow 1$ n'est pas utilisée, et le CLB pourrait vérifier 9 bits d'un coup, mais cette possibilité ne fournit

pas de gain dans la pratique (huit CLB sont utilisés pour comparer 64 bits ; avec sept CLB, on ne peut traiter que 63 bits au mieux).

Ainsi, huit CLB fournissent à chaque cycle huit bits, chacun représentant l'adéquation de huit bits de sortie du tour courant avec huit bits du texte chiffré connu. Un neuvième CLB effectue le ET logique de ces huit bits ainsi que de *clk16* : la sortie de ce CLB vaut 1 si et seulement si le texte chiffré attendu a été trouvé exactement en sortie du seizième tour de DES.

Chaque comparateur est relié à un registre auquel est présenté, à chaque cycle, le OU logique de son contenu au cycle précédent et du bit de sortie du comparateur. Ainsi, ce registre, qui vaut initialement 0, passe à 1 (et y reste) dès que le bon texte chiffré a été obtenu (ce qui signifie, avec une forte probabilité, que la bonne clé a été essayée). Le contenu de ce registre est directement relié à un bus extérieur testable facilement depuis la machine hôte.

Au total, chaque comparateur utilise 9 CLB, et il faut compter un demi-CLB de plus pour le registre de stockage.

Programmation et exécution

Le mode de fonctionnement de l'ensemble est le suivant : le *bitstream* codant le schéma est envoyé aux cartes ; ce schéma contient le texte clair et le texte chiffré connus, intégrés à la définition des modules `feed_L`, `feed_R` et comparateur des différentes instances de DES. Il y a quatre FPGA ; dans chacun, quatre instances de DES avec leur comparateur et leur registre indiquant le succès de l'opération, et un compteur de clés sur 36 bits. Chacun des seize modules `key_sched` est conçu pour essayer une clé dont 20 bits sont fixes et 36 sont fournis par le générateur de clés. Les seize instances de DES essayent donc, quand le compteur passe par toutes ses valeurs possibles, $16 \times 2^{36} = 2^{40}$ clés DES. Le schéma peut être cadencé à 25 MHz ; chaque instance de DES nécessite seize cycles d'horloge pour essayer une clé, et il y a seize telles instances, donc la carte Pamette essaye 25 millions de clés à la seconde. À ce rythme-là, l'espace des 2^{40} clés à essayer est couvert en un peu plus de douze heures.

Pendant ces douze heures de calcul, la station hôte (de type Alpha ou PC, sous Windows NT, OSF ou Linux), consulte à intervalles réguliers le contenu des registres indicateurs de succès. Une telle consultation peut être effectuée plus de vingt fois par seconde sur le premier PC venu en utilisant moins de 1% de ses capacités de calcul. Le PC peut chronométrer avec cette précision l'apparition d'un 1 indiquant que la bonne clé a été trouvée (la probabilité que deux clés différentes fournissent le bon chiffré à partir du clair connu est négligeable, dans un espace de 2^{40} clés possible ; et, quand

bien même cela arriverait, il suffirait de relancer une recherche à partir de la clé suivante, en fixant la valeur d'initialisation des registres du générateur de clés et des modules `key_sched`). Grâce à la connaissance du temps écoulé entre le lancement des cartes et le moment où un registre de succès passe à 1, l'hôte peut déterminer la clé essayée à environ 1,2 millions près (c'est le nombre de clés essayées chaque vingtième de seconde). Le plus petit des PC vendus actuellement peut essayer toutes ces clés en moins d'une minute, et conclure la recherche.

Le bon fonctionnement du système suppose que les textes clairs et chiffrés, ainsi que les bits fixes des clés, sont intégrés à la définition des fonctions $4 \rightarrow 1$ des CLB correspondants. Cela peut se faire de façon simple en agissant sur le source C++ du code PamDC ; mais cela implique une recompilation complète du schéma, ce qui peut prendre environ 12 heures, soit un temps du même ordre que le parcours total de l'espace des clés. Afin de contourner cet inconvénient, on peut agir sur les fichiers représentant le schéma déjà compilé ; ces fichiers sont des fichiers texte et sont équivalents à un code assembleur : l'opération de traduction du contenu de ces fichiers en un *bits-tream* ne prend que quelques minutes. Par ailleurs, si on effectue plusieurs recherches successives, le PC hôte peut compiler un schéma pendant que la carte Pamette travaille sur le précédent.

Placement

Le placement des différentes fonctions dans les CLB du FPGA est primordial dans la construction du schéma : en effet, de ce placement dépend l'efficacité du routage des données. L'expérience montre que le temps pris par les données pour voyager entre deux CLB est du même ordre de grandeur que le temps de calcul des fonctions d'un des CLB, si le placement est raisonnablement optimisé. Le schéma présenté ici fait traverser trois CLB au plus par certains signaux à chaque cycle ; cette traversée et le routage correspondant s'effectuent en 40 ns, d'où les 25 MHz de fréquence maximale de fonctionnement.

Il est difficile d'améliorer ces performances : en effet, les permutations incluses dans DES nécessitent un routage non trivial. Ces permutations ont été calculées selon des critères de sécurité et ne sont pas décomposables simplement, aussi tout placement aura des chemins relativement longs (ceci contredit partiellement l'assertion selon laquelle les permutations de bits ont un coût nul dans une implantation matérielle ; ceci dit, elles sont bien moins coûteuses que dans une implantation logicielle). En revanche, la structure globale de DES telle qu'illustrée par la figure 4.4 doit être respectée : une préversion faisait se croiser les bus U et S ; l'embouteillage était tel qu'un

des signaux devait emprunter un chemin détourné qui contournait tout le bloc calculant DES, et amenait à un temps de calcul de 80 ns par cycle, c'est-à-dire la moitié des performances obtenues dans la version finale.

Les outils de compilation Xilinx s'avèrent incapables de trouver seuls un placement permettant des performances décentes, aussi toutes les fonctions ont été assignées, dans le code C++, à des CLB donnés, via les primitives de placement intégrées à PamDC. Dans la pratique, le générateur des signaux *clk* a été placé au centre, à côté du compteur de clés. Les quatre instance de DES utilisent des champs de forme rectangulaire dans la matrice de CLB, et occupent les quatre coins du FPGA.

Si on fait le compte des CLB utilisés, on trouve ceci :

module	taille	instances	total
S/P/des_conf	80	4	320
feed_R	40	4	160
feed_L	16	4	64
key_sched	28	4	112
comparateurs	8,5	4	34
compteur de clés	60	1	60
horloges	4	1	4
Total			754

Le FPGA XC4020 contient $28 \times 28 = 784$ CLB ; le schéma utilise donc 96% des CLB de chaque FPGA.

4.4 Prospectives

Dans la section précédente, j'ai décrit une implantation d'un système de recherche exhaustive de clés DES sur 40 bits ; nous allons étudier ici comment les performances de ce genre de système vont évoluer dans l'avenir.

4.4.1 La loi de Moore

Gordon Moore, co-fondateur d'Intel, a énoncé en 1965 la « loi » suivante : tous les 18 mois, les capacités des ordinateurs doublent (à prix de production constant). Cela signifie une montée en puissance exponentielle ; cette loi a été remarquablement vérifiée au cours du dernier quart de siècle. Voyons plus en détail comment se décompose cette loi.

Tout d'abord, le prix du matériel est directement lié à la surface des puces. Le processus de fabrication fait intervenir des disques de silicium comportant plusieurs puces, éventuellement plusieurs centaines ; les puces sont gravées et

assemblées couche par couche sur la surface de ce silicium. À chaque couche, le taux de rejet est important, car chaque poussière peut compromettre une puce entière ; au-delà d'une dizaine de couches, il ne subsiste plus assez de puces opérationnelles. Voilà pourquoi les processeurs restent des objets fondamentalement bidimensionnels ; de plus, si on considère que chaque poussière sur un disque va détruire la puce correspondante, on voit qu'on a intérêt à conserver des puces de taille réduite, afin de limiter au maximum les dégâts.

Il s'avère qu'au cours des 25 dernières années, tous les trois ans, avec une régularité impressionnante, l'industrie micro-électronique a pu produire des puces comportant quatre fois plus de transistors sur une surface donnée, donc pour le même prix, et cadencer ces puces à une fréquence double. L'opposition à une grande cadence de fonctionnement tient aux points suivants :

- une plus grande fréquence implique de plus brusques variations de courants, donc un échauffement supérieur ;
- la vitesse d'un influx électrique dans un conducteur est limitée à environ 200 000 km/s ;
- le temps de commutation d'une porte logique dépend du type et de la qualité des substrats utilisés dans les transistors.

Les méthodes utilisées par les fabricants ont été principalement les suivantes :

- amélioration des matériaux utilisés ;
- affinement de la gravure, pour rapprocher les transistors ;
- augmentation du nombre de couches ;
- utilisation du cuivre plutôt que l'aluminium (le cuivre résiste mieux à la chaleur) ;
- abaissement de la tension de fonctionnement (donc diminution de la consommation donc de l'échauffement).

Grâce à ces méthodes, la loi de Moore a tenu pendant un temps très long comparativement à la vitesse d'évolution du secteur des semi-conducteurs.

4.4.2 Gain pour les implantations matérielles

On peut remarquer qu'un doublement de la fréquence et un quadruplement du nombre de transistors pour un coût donné, devraient amener une multiplication par huit de la puissance de fonctionnement, là où la loi de Moore ne prévoit qu'un quadruplement. Mais les ordinateurs conventionnels ne peuvent pas suivre complètement ces progrès techniques, pour les raisons suivantes :

- Un ordinateur n'est pas qu'un processeur, c'est aussi une mémoire ; cette mémoire a un temps d'accès moyen qui n'augmente qu'avec le gain en fréquence ; les ordinateurs compensent par des stratégies de mémoire cache, qui ne sont pas parfaites.

- Le modèle de calcul des ordinateurs est de travailler avec des données d'une certaine taille, taille qui dépend du monde extérieur et qui n'évolue pas. Le fait de traiter des données de plus de 64 bits n'a qu'une utilité réduite.
- Les ordinateurs travaillent sur des flux de contrôle, à savoir des suites d'instructions qui ne sont, par nature, pas parallélisables ; diverses stratégies sont tentées (processeurs superscalaires, qui tentent d'exécuter plusieurs instructions à la fois), avec un succès mitigé.

Ces raisons expliquent pourquoi on n'observe qu'un quadruplement tous les trois ans de la puissance de calcul des ordinateurs. En revanche, dans le cas très particulier d'un système de recherche exhaustive de clés de chiffrement, on peut appliquer un parallélisme massif, et tout le travail s'effectue en local, dans des modules bien séparés, ainsi que l'illustre le DES-cracker sur FPGA que j'ai réalisé. On peut donc profiter au maximum des évolutions apportées par le progrès technique, ce qui nous donne bien huit fois plus de clés essayées par seconde pour un coût donné, tous les trois ans, soit un doublement de puissance chaque année.

En termes plus concis, cela signifie le gain suivant pour le cassage de cryptosystèmes symétriques : *un bit par an*.

4.4.3 Le futur

L'état de l'art actuel, attesté, est un DES-cracker capable de retrouver une clé DES de 56 bits en quelques jours, pour un coût de production de l'ordre de 100 000 dollars américains (l'EFF a dû dépenser plus, pour financer le développement, mais elle a publié l'intégralité du schéma au format VHDL, sur papier, dans une police de caractère prévue pour être scannée et reconstruite sur ordinateur ; c'était alors le moyen habituel de contourner les lois d'exportations américaines², qui ne s'appliquent pas aux livres, à cause de la liberté d'expression garantie constitutionnellement). Ce résultat est vieux de quelques années, et on peut considérer qu'un adversaire motivé dispose d'un budget de plusieurs millions de dollars. Au total, on peut chiffrer les possibilités de cassage par recherche exhaustive à 64 bits, en l'an 2001. Ce nombre dépend un peu de la facilité d'implantation de l'algorithme considéré dans une puce spécialisée : par exemple, RC6, algorithme de chiffrement utilisant des multiplications de nombres de 32 bits, est assez nettement plus coûteux à planter en matériel, que DES. Néanmoins, l'écart entre les deux est constant, de l'ordre d'au plus 5 bits de clé. Par ailleurs, de façon générale, on peut dire qu'une implantation sur FPGA tourne deux fois moins vite

²Ces lois ont été très assouplies en janvier 2000.

que son homologue sur ASIC, donc un bit de retard ; le développement sur FPGA étant beaucoup plus facile que sur ASIC (pas besoin d'accord avec un laboratoire équipé pour fabriquer quelques milliers de puces spécialisées), on peut estimer, en termes de sécurité informatique, que le cassage discret d'une clé de 64 bits est à la portée d'un grand nombre d'organisations, ou le sera dans moins de 10 ans.

Cette perspective est relativement inquiétante : cela signifie que les secrets protégés par une clé de 128 bits aujourd'hui, ne sont sûrs que jusque vers l'an 2070. Ceci n'est vrai que si la loi de Moore se maintient encore pendant 70 ans, ce qui n'a rien d'évident. En fait, il y a beaucoup de bonnes raisons pour que la loi de Moore cesse d'être valable à brève échéance ; principalement, elle s'est appuyée sur bon nombre d'innovations technologiques qui avaient été envisagées dès les années soixante-dix. Cette provision d'idées est à présent à peu près épuisée, et il n'y en a plus pour prendre la relève ; par exemple, on n'a pas d'idée pour graver plus finement que $0,05 \mu\text{m}$, finesse au-delà de laquelle les électrons sautent entre les fils par effet tunnel ; or, on produit industriellement à une finesse de $0,12 \mu\text{m}$. Cependant, il existe une très bonne raison pour laquelle je vais me garder de prédire la fin de la loi de Moore : cela a déjà été fait, sans succès, par beaucoup de monde au cours de la dernière décennie, et toujours pour tout un tas de raisons qui semblaient très bonnes alors. Gordon Moore lui-même a prédit, en 1997, que sa « loi » cesserait d'être valable en 2017.

Ceci étant, si la loi de Moore ne se maintient peut-être pas, il est en revanche très improbable qu'elle s'accélère. Aussi, en guise de conclusion de ce chapitre, je prédis que les secrets protégés par des clés de 128 bits ne seront pas cassés avant l'an 2070, sous réserve que l'algorithme utilisé ne comporte pas de faiblesse majeure découverte d'ici-là.

Chapitre 5

Compromis matériel/logiciel

5.1 Présentation de A5/1

Les systèmes de téléphonie mobile ont connu, au cours des cinq dernières années, un remarquable succès commercial, à tel point qu'il existe désormais plus de téléphones portables en usage en France que de téléphones fixes. Cela a des conséquences sur la confidentialité des conversations, et des documents divers autres que la voix qui commencent à être échangés par le même chemin. En effet, autant un câble téléphonique peut être protégé physiquement, autant une communication radio semble interceptable silencieusement pour un coût faible. Je ne me prononcerai pas sur la faisabilité électrotechnique d'une telle interception, car cela réclame des compétences que je ne possède pas, mais le cryptographe soucieux de sécurité doit, par discipline d'esprit, supposer que les verrous physiques sont cassés par l'attaquant, et que seuls des algorithmes bien conçus peuvent le mettre en échec.

Le standard dominant en téléphonie portable en Europe est, actuellement, le GSM. Le GSM comporte, dans le protocole, une procédure de chiffrement supposée assurer la confidentialité des échanges effectués entre le téléphone et la borne d'opérateur la plus proche. Cette procédure utilise un protocole en deux étapes :

1. détermination d'une clé de session via l'algorithme A3A8,
2. chiffrement des deux flux de données par l'algorithme A5.

A3A8 est un nom de code qui peut recouvrir à peu près n'importe quel algorithme, au choix de l'opérateur, suivant le *modus operandi* suivant : la borne émet un *challenge* à partir duquel le téléphone calcule la clé de session en faisant intervenir une clé secrète connue de sa carte SIM (une carte à puce intégrée au téléphone). La borne effectue le même calcul, car elle connaît la clé secrète elle aussi (en fait, elle contacte par réseau un serveur de l'opérateur,

qui connaît l'algorithme utilisé et la clé de l'utilisateur). Si la clé de session ne correspond pas des deux côtés, la borne coupe la communication. Cette procédure permet à la fois d'utiliser une clé de session non devinable d'après ce qui passe par la voie des airs, mais en plus elle identifie le porteur du téléphone, ce qui est à la base de la facturation. La spécification GSM propose un algorithme convenable pour A3A8, nommé COMP128 ; cet algorithme a été publié officiellement, et cryptanalysé ; heureusement, cette attaque, appliquée à la situation réelle, ne semble pas avoir de conséquences trop dramatiques.

A5 est un nom de code, qui recouvre en fait trois algorithmes différents, A5/0, A5/1 et A5/2. A5/0 est simplissime : c'est l'identité, qui ne chiffre rien. A5/0 est le seul algorithme de chiffrement autorisé dans les pays dits « sensibles » tels que la Syrie ou l'Iran. A5/1 est un système de chiffrement « fort », réservé à la Communauté Européenne et à l'Amérique du Nord ; A5/2 est dit « faible » et est fourni aux pays alliés (extrême-orient notamment). A5/2 a été publié en 1998 par Marc Briceno et Ian Goldberg, ainsi qu'une cryptanalyse très efficace (présentée à la *Rump Session* de Crypto'98), qui démontre que c'est à raison qu'A5/2 était dit « faible ».

5.1.1 Historique de A5/1

L'algorithme A5/1 a longtemps été conservé secret. Une première description est publiée par Ross Anderson dans le groupe de discussion Usenet sci.crypt[1]. Cette description, obtenue par analyse d'une implantation, était incorrecte sur plusieurs points, mais montrait déjà la structure interne du système.

Une seconde implantation a été publiée en 1999 par Marc Briceno, Ian Goldberg et David Wagner[16]. Elle diffère de celle de Ross Anderson par le détail des registres à décalage utilisés. Elle est distribuée sous la forme de code source en C, sous l'égide de la *Smartcard Developer Association*, qui annonce avoir « vérifié » l'exactitude de la description. Alex Biryukov, Adi Shamir et David Wagner en ont tiré une cryptanalyse[14], et annoncent, dans leur article, avoir reçu confirmation du groupement GSM qu'il s'agit bien de l'algorithme tel qu'utilisé dans la norme.

5.1.2 Algorithme principal

A5/1 est en fait un générateur pseudo-aléatoire d'un flux de bits ; ce flux de bits est combiné par un OU EXCLUSIF avec les données à chiffrer. C'est le principe du *One Time Pad*, avec un masque pseudo-aléatoire et donc non aléatoire ; le déchiffrement se pratique de façon identique au chiffrement,

par la génération du même flux, et un OU EXCLUSIF avec le texte chiffré. A5/1 est constitué de trois LFSR (*Linear Feedback Shift Registers*, c'est-à-dire registres à décalage à rétroaction linéaire), de tailles respectives 19, 22 et 23 bits, totalisant un état interne de 64 bits.

Le fonctionnement d'un LFSR est simple : chaque fois qu'il est sollicité, l'ensemble de ses bits est décalé d'une position. Le bit placé en bout de registre « tombe » : c'est la valeur de sortie du registre. À l'autre bout du registre, une place est laissée libre : elle est remplie avec une forme linéaire de la valeur du registre (avant décalage), c'est-à-dire, comme on travaille sur \mathbf{Z}_2 , un OU EXCLUSIF de certains bits précis du registre.

On peut associer à un LFSR de taille n un polynôme à coefficients dans \mathbf{Z}_2 , dit *polynôme de rebouclage*, de la forme :

$$P = 1 + \sum_{i=1}^n a_i X^i \quad (5.1)$$

où chaque a_i est non nul si et seulement si le bit i du registre à décalage est utilisé pour le rebouclage (en comptant les bits du LFSR avant décalage, à partir de 1 là où rentre le nouveau bit). On peut montrer qu'un LFSR parcourt, à partir d'un état interne non nul, l'ensemble des états internes non nuls possibles si et seulement si son polynôme P est de degré n (ce qui veut dire que le bit de sortie participe du rebouclage) et primitif, c'est-à-dire premier (non décomposable en produit de deux polynômes de degré strictement supérieur à 0) et tel que le polynôme X est générateur de l'ensemble des inversibles modulo P . Dans de telles conditions, le LFSR se comporte comme un générateur de bits pseudo-aléatoires dont le comportement statistique est satisfaisant pour la plupart des applications ; en revanche, un tel générateur, constitué uniquement d'un LFSR, n'est pas cryptographiquement intéressant car il est fortement prédictible ; et, même si le polynôme de rebouclage est inconnu, on peut le reconstruire à partir de $2n$ bits de sortie grâce à l'algorithme de Berlekamp-Massey[5]. Si on connaît le polynôme de rebouclage, on peut exprimer chaque bit de sortie comme un OU EXCLUSIF de certains des bits de l'état interne initial.

Dans le cas de A5/1, les trois LFSR ont pour polynômes de rebouclages les trois polynômes suivants :

$$P_1 = X^{19} + X^{18} + X^{17} + X^{14} + 1 \quad (5.2)$$

$$P_2 = X^{22} + X^{21} + 1 \quad (5.3)$$

$$P_3 = X^{23} + X^{22} + X^{21} + X^8 + 1 \quad (5.4)$$

Ces trois polynômes sont primitifs et le degré de chacun est égal à la taille du registre correspondant. De plus, ces polynômes sont relativement « creux »,

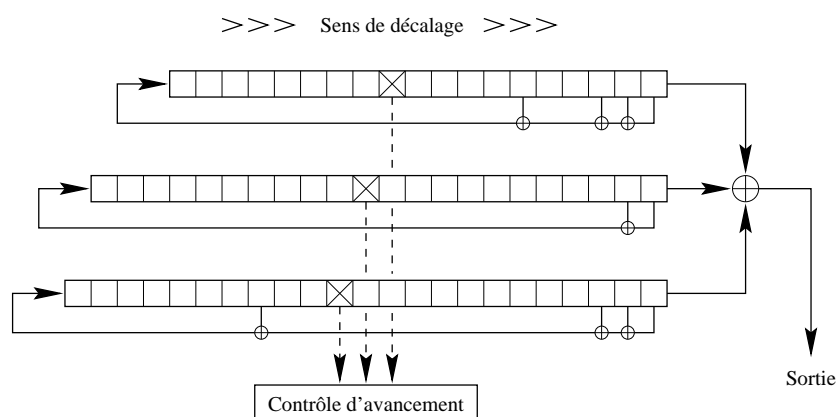


FIG. 5.1 : L'algorithme A5/1

c'est-à-dire que la fonction de rebouclage se calcule avec un nombre réduit de portes OU EXCLUSIF.

Toute la force de A5/1 réside dans son avancement sélectif de ces trois registres ; en effet, le mode d'opération de A5/1 s'effectue ainsi : à chaque cycle, un test est effectué pour chaque registre, test qui décide si le registre est avancé d'une unité ou pas. Pour ce test, un bit de chaque registre (respectivement, les bits 9, 11 et 11) est extrait, et la majorité de ces trois bits est calculée. Les registres dont le bit extrait est en accord avec cette majorité, sont décalés d'une unité. Ainsi, à chaque cycle, au moins deux des trois registres sont décalés. Une fois ces déplacements effectués, le bit terminal de chaque registre (respectivement, les bits 19, 22 et 23) est extrait, et le OU EXCLUSIF de ces trois bits est le bit de sortie d'A5/1 pour ce cycle.

C'est en ce décalage dépendant non-linéairement de l'état interne des registres que réside la sécurité d'A5/1 ; notamment, ce décalage fait influencer le devenir d'un registre en fonction des deux autres. La figure 5.1 décrit la structure interne d'A5/1.

5.1.3 Mise en œuvre de A5/1

Une communication GSM est découpées en *frames*, c'est-à-dire des blocs de taille fixe (quelques centaines de bits) ; chaque *frame* est chiffrée indépendamment. Ce système permet de partager le même canal radio entre plusieurs communications, et autorise une synchronisation plus efficace entre les intervenants. Chaque *frame* est numérotée, en commençant à 0 ; ce numéro est un nombre binaire de 22 bits.

Chaque *frame* comporte deux champs de données de 114 bits ; chacun

chiffre un des deux sens de la communication (car une communication téléphonique est bidirectionnelle). Pour chaque *frame*, A5/1 est réinitialisé ainsi :

- les trois registres sont initialisés à 0 ;
- la clé de session de 64 bits est rentrée, bit par bit, de la façon suivante : le bit est combiné par OU EXCLUSIF avec le bit 1 de chaque registre, puis les trois registres sont décalés d'une unité (le contrôle d'avancement est désactivé pendant cette procédure) ;
- le numéro de la *frame* est introduit de la même façon ;
- le contrôle d'avancement est alors activé ; A5/1 tourne pendant 100 cycles « à vide », c'est-à-dire que les 100 premiers bits de flux sont ignorés ;
- ensuite, A5/1 produit les 228 bits nécessaires au chiffrement.

Cette réinitialisation régulière permet de s'affranchir du fait qu'A5/1 n'est pas réversible : en effet, plusieurs états internes différents peuvent aboutir au même état interne après un cycle. Si A5/1 était utilisé pour générer une chaîne trop longue, il finirait par atteindre un cycle d'états internes nettement plus petit que 2^{64} (si A5/1 se comporte de façon statistiquement aléatoire, comme il est soupçonné, la longueur du cycle atteint, et le nombre moyen de bits à calculer pour atteindre ce cycle, sont de l'ordre de $\sqrt{2^{64}} = 2^{32}$). De plus, grâce au découpage en *frames*, les deux appareils de transmission n'ont besoin de se synchroniser que pendant 228 cycles, et un dysfonctionnement ponctuel du chargement de l'état interne n'affecte qu'une seule *frame*.

On remarquera que le mode d'entrée du numéro de *frame* implique que chaque bit de ce numéro influe sur les trois registres. La voix est compressée à 13 Kbits/s en GSM, donc il y a environ 114 *frames* par seconde (il y a deux voix à chiffrer, donc 26 Kbits/s). Les 2^{22} numéros de *frame* possibles permettent une communication continue d'un peu plus de 10 heures avant de devoir réutiliser un numéro de *frame* (cette réutilisation implique qu'une *frame* est chiffrée avec exactement le même flux qu'une autre, ce qui peut révéler beaucoup d'information sur ces deux flux).

5.1.4 Implantation de A5/1

A5/1 est conçu pour être implanté très efficacement en matériel ; quelques centaines de portes logiques suffisent. En revanche, les implantations logicielles sont difficiles : le calcul d'un décalage de LFSR nécessite plusieurs masquages et tests, et il faut compter plus de 10 cycles d'horloge pour chaque bit produit, un ratio plutôt mauvais, quand on remarque que les meilleurs cryptosystèmes de chiffrement par bloc peuvent descendre à 2 cycles/bit.

Quand on ne s'intéresse pas à la sortie de A5/1, mais seulement à l'évolution de son état interne, on peut utiliser une implantation logicielle décrite par Alex Biryukov, Adi Shamir et David Wagner[14], à base de tables. L'idée

est la suivante : on peut représenter les bits de sortie successifs de chaque registre dans une table (le plus long des registres a une taille de 23 bits, et suivra un cycle d'états internes de taille $2^{23} - 1$; les bits de sortie de ce registre sont donc représentables dans une table circulaire occupant 1 méga-octet). Comme les bits de sortie d'un LFSR donné sont exactement les bits de son état interne, on peut envisager le contenu du registre comme une fenêtre glissante sur cette table. L'état interne de A5/1 est alors un triplet d'indice dans les trois tables représentant les trois registres. Pour avancer de huit cycles l'état interne de A5/1, il suffit de récupérer dans les tables les contenus des 24 bits (8 bits par registre) qui peuvent potentiellement influencer sur l'avancée des trois LFSR pendant ces huit cycles, et d'aller lire dans une table précalculée de taille 2^{24} les trois avancées des indices représentant l'état des LFSR.

Cette implantation permet, sur une station de travail disposant de suffisamment de mémoire pour stocker les tables (quelques dizaines de méga-octets suffisent ; la table d'avancement est la plus grande des quatre), d'avancer l'état interne de A5/1 de huit cycles en faisant trois accès mémoire dans les trois tables représentant les registres, et un quatrième accès dans la table d'avancement ; quelques additions supplémentaires sont nécessaires. Cette méthode n'est pas intéressante pour un fonctionnement classique, car elle ne donne pas directement accès au flux de sortie, mais elle permet d'implanter efficacement le précalcul de la cryptanalyse de A5/1 décrite par Biryukov, Shamir et Wagner[14].

5.2 Cryptanalyse logicielle de A5/1

La force d'A5/1 est entièrement dans la séquence d'avancement de ses trois registres ; en effet, une fois cette séquence connue, les bits de sortie peuvent être exprimés comme de simples équations linéaires des bits de l'état interne ; une simple réduction de Gauss permet alors de résoudre le système, et de retrouver l'état interne. Dans toute cette section, on travaille sur \mathbf{Z}_2 , donc une équation linéaire est un OU EXCLUSIF de certaines inconnues binaires, et son résultat est 0 ou 1.

- Plusieurs cryptanalyses ont été proposées ; on en distingue deux classes :
- les recherches exhaustives sur une partie de l'état interne ;
 - les compromis taille de flux connu/temps/mémoire.

Dans tous les cas, on cherche à reconstituer l'état interne juste avant la génération des bits de flux effectivement utilisés pour chiffrer. On peut montrer qu'A5/1 est pseudo-réversible, c'est-à-dire que la complexité de la recherche inverse des états ayant pu mener à un état interne donné est linéaire

en fonction du nombre de cycles que l'on veut remonter[40]; en effet, pour un état donné, il n'y a en moyenne qu'un seul état prédecesseur possible.

5.2.1 Recherche exhaustive partielle

Dès la première publication (erronée) de A5/1, il a été suggéré de le cryptanalyser en faisant une recherche exhaustive partielle sur les registres, en devinant suffisamment de bits pour déduire les bits restant en fonction des bits de sortie. Principalement, les bits réglant l'avancement des registres pendant un certain nombre de cycles devait être produits *ex nihilo*. Ross Anderson[1] était le premier à suggérer ces méthodes, suivi de Golić[40]. Dans les deux cas, la complexité de la cryptanalyse est de l'ordre de 2^{45} opérations simples (Golić exprime une meilleure complexité, mais chacune de ses opérations « élémentaires » est en fait une inversion d'un système de plusieurs dizaines d'équations linéaires).

Toutes ces tentatives sont en fait des variations sur la cryptanalyse que je présente ici. L'idée principale est que la séquence d'avancement échappe à l'analyse; aussi, il faut effectuer une recherche exhaustive sur cette séquence pendant un certain nombre de cycles. Chaque essai permet de reconstruire l'état interne, et de le tester. L'état interne faisant 64 bits, on n'a besoin, en moyenne, que de 64 bits de flux de sortie connus. Cette attaque, co-écrite avec Jacques Stern, a été présentée à CHES'2000[74].

À chaque cycle d'horloge, quatre avancements sont possibles : soit les deux premiers registres avancent, soit les deux derniers, soit le premier et le dernier, soit les trois registres. Ces quatre possibilités sont exprimables sous la forme de deux équations linéaires, qui expriment l'égalité (ou l'inégalité) des bits de contrôle des deux premiers registres, et des bits de contrôle du premier et du troisième registre. Une égalité entre deux éléments de \mathbf{Z}_2 s'exprime par leur somme, valant 0; pour l'inégalité, leur somme doit valoir 1. Pendant ce même cycle d'horloge, on obtient un bit de sortie, et on connaît très précisément les bits d'état interne qui ont participé à la valeur de ce bit de sortie, puisqu'on a « deviné » la séquence d'avancement. Ceci nous donne une équation linéaire supplémentaire.

On ajoute ces équations linéaires dans un système qu'on réduit, par la méthode d'élimination de Gauss, au fur et à mesure, selon la méthode suivante :

1. On ajoute la première équation dans le système.
2. On appelle u_1 une inconnue dont le coefficient vaut 1 dans cette équation.

3. Quand on ajoute l'équation X au système qui comporte déjà n équations, on annule les coefficients des inconnues $u_i (1 \leq i \leq n)$ dans X comme suit : si X a un coefficient non nul pour l'inconnu u_1 , on remplace X par la somme de X et de la première équation ; on continue avec u_2 et la seconde équation, *etc.*
4. Si, après ces opérations, l'équation X n'est pas devenue triviale (elle ne comporte plus d'inconnue), alors on l'ajoute au système ; on choisit u_{n+1} comme étant une inconnue dont le coefficient est non nul dans X .

Cet algorithme garantit que pour l'équation numéro j , les coefficients des inconnues u_1, u_2, \dots, u_{j-1} sont nuls. Quand 64 équations sont assemblées, on obtient un système triangulaire dont la résolution est aisée. Quand on obtient une équation triviale, elle peut avoir deux formes : $0 = 0$, ce qui signifie que cette équation n'apporte aucune information par rapport aux équations déjà assemblées, et $0 = 1$, qui est une impossibilité, qui indique que la séquence d'avancement choisie n'est pas possible en regard des bits de sortie connus.

On peut remarquer que l'élimination de Gauss appliquée à une équation ne modifie pas les équations déjà en place dans le système ; aussi, on peut revenir en arrière sur les derniers choix. Ceci permet d'organiser la recherche exhaustive sous la forme d'un *backtrack*, c'est-à-dire un procédé récursif arborescent. On définit un système d'équations commun, et une fonction qui prend en entrée un état d'avancement des registres. Cette fonction tente d'ajouter, successivement, les quatre groupes de trois équations linéaires correspondant aux quatre avancements possibles des trois registres. Pour chaque ajout, si aucune équation de type $0 = 1$ n'a été trouvée, la fonction s'appelle récursivement sur le nouvel état d'avancement. Quand elle reprend le contrôle, elle enlève ces trois équations et ajoute à la place les trois équations correspondant à l'hypothèse suivante. Quand la fonction a épuisé ses quatre hypothèses, elle rend la main à sa fonction appelante.

Tant que le système est incomplet, l'expérience montre que les éliminations n'amènent qu'à très peu d'équations triviales ; on arrive donc en moyenne à explorer un arbre de taille $2^{64 \times 2/3}$, soit environ $2^{42,7}$, car on doit deviner deux équations (celles représentant l'avancement des registres) pour en ajouter trois au système (ces deux équations, et celle explicitant le bit de sortie). Ensuite, une fois que le système est complet, il contient suffisamment d'informations pour reconstruire l'intégralité de l'état interne. Cette reconstruction serait assez coûteuse, aussi, on se contente de continuer à pratiquer l'élimination gaussienne de nouvelles équations. Comme l'état interne est entièrement représenté, tout ajout de nouvelle équation donne forcément une équation triviale après élimination, et, sur les quatre hypothèses d'avancement des registres, une et une seule est valide, c'est-à-dire n'aboutit pas à

une équation $0 = 1$. L'équation calculant le bit de sortie a alors une chance sur deux de donner $0 = 1$ après élimination, donc, en moyenne, on poursuit ce mécanisme sur deux étapes.

Au total, on pratique $6 \times 2^{64 \times 2/3}$ éliminations gaussiennes d'équations linéaires par rapport à un système constitué d'une soixantaine de telles équations. Ceci nous donne une complexité moyenne de $2^{44,3}$ telles opérations (car, en moyenne, on explorera la moitié de l'arbre des séquences d'avancement possibles, avant de trouver le bon état interne). Cette complexité corrobore les résultats de Golić et Anderson. La différence, cependant, avec ces deux précédentes cryptanalyses, est que celle présentée ici a été implantée et testée en vraie grandeur ; les équations linéaires sont représentables par des mots de 65 bits, et la somme de deux équations est un OU EXCLUSIF bit à bit. Ces caractéristiques permettent de gérer efficacement l'élimination gaussienne quand on dispose d'une station de travail possédant de longs registres entiers. Sur une station Compaq XP1000, dotée d'un processeur Alpha 21264 à 500 MHz, le temps d'exploration de l'intégralité de l'arbre est de 400 jours, ce qui donne un temps moyen de cryptanalyse de 200 jours. La consommation mémoire est négligeable, et l'algorithme se parallélise relativement bien (il suffit de tronçonner l'arbre près de sa racine, et distribuer les branches).

Cette cryptanalyse est la meilleure connue à ce jour (en termes de complexité logicielle), quand on ne dispose que de peu de bits de clair connu (64 bits).

5.2.2 Compromis flux/temps/mémoire

Golić[40] a été le premier à présenter une idée de cryptanalyse de A5/1 en utilisant une variation du compromis temps/mémoire de Hellman[45]. L'idée est la suivante : précalculer l'inversion de A5/1 pour un grand nombre de sorties possibles (en tirant aléatoirement des états internes et en calculant la sortie correspondante). Ensuite, il suffit d'attendre que la sortie observée de A5/1 forme une des sorties possibles précalculées. On remplace le temps de calcul du compromis d'Hellman par la longueur du flux intercepté. Dans la version de Golić, le coût est prohibitif : il faut plusieurs milliers de giga-octets de stockage, et la quantité de flux devant être connu se chiffre en heures.

Puis Biryukov, Shamir et Wagner[14] ont présenté une variante de cette attaque, en appliquant de nombreuses optimisations ; l'idée est de ne s'intéresser qu'aux états de sortie commençant par une série fixe de quelques bits (16 bits dans leur article). La reconstruction des états internes donnant des sorties commençant par ces 16 bits est, de fait, très similaire à la cryptanalyse logicielle présentée précédemment. La longueur totale de flux à intercepter, c'est-à-dire la quantité de clair connu nécessaire à la réussite de la cryptana-

lyse, est de l'ordre de 25 000 bits, soit deux secondes de conversation.

5.3 Cryptanalyse matérielle de A5/1

A5/1 est un algorithme qui s'implante fort bien en matériel, car il utilise très peu de transistors (ce qui implique une faible consommation électrique, ce qui le rend très adapté aux applications mobiles, et donc aux téléphones portables). On peut donc construire des ASIC spécialisés essayant un grand nombre de clés en parallèle.

5.3.1 Implantation de A5/1 sur FPGA

Je décris ici succinctement une implantation de A5/1 sur la carte Pamette (cf. section 4.2) ; cette implantation permet d'estimer le coût d'une recherche exhaustive sur une partie de l'espace de clé.

Pour chaque registre à décalage :

- Chaque bit est contenu dans un registre d'un CLB.
- La fonction $4 \rightarrow 1$ correspondant à ce registre charge ce dernier avec la valeur de sortie du bit précédent.
- Cette fonction reçoit aussi deux autres entrées, qui servent à charger un nouvel état interne dans le registre (un signal emporte la valeur à charger, l'autre indique quand charger cette nouvelle valeur à la place du bit précédent du LFSR).
- Le contrôle d'avancée du LFSR s'effectue par la broche *enable* des registres (chaque registre recharge sa valeur avec son signal d'entrée à la fin du cycle d'horloge si et seulement si son signal *enable* est positionné à 1).

La figure 5.2 illustre ce mécanisme. Le signal *init* est le nouvel état interne à entrer dans les registres (cet état est typiquement généré par un compteur, dans le cadre d'une cryptanalyse par recherche exhaustive). Le signal *reload* indique quand le registre doit recharger son état interne avec *init*. L'avancée du registre est contrôlé par une fonction $4 \rightarrow 1$ qui prend en entrée les valeurs des bits centraux des trois LFSR d'A5/1, ainsi que le signal *reload* (car lors d'un rechargement de l'état interne, la broche *enable* de chaque bit de chaque LFSR doit être mise à 1).

Tous les bits de chaque registre sont remplacés, lors d'un décalage, par la valeur du bit précédent dans l'ordre d'avancée du LFSR ; tous, sauf le premier, qui est rechargé par une combinaison linéaire (un OU EXCLUSIF) de certains des bits du registre. Cette combinaison linéaire ne concerne que deux ou quatre bits (dans le cas de A5/1), donc peut être aisément implantée

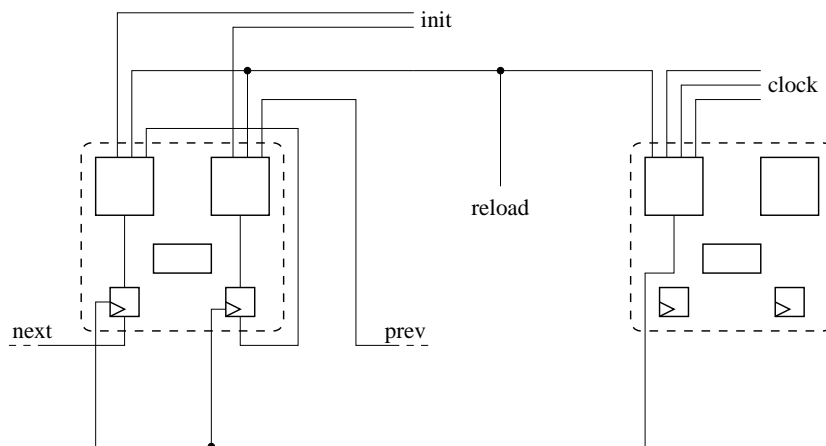


FIG. 5.2 : A5/1 sur carte Pamette

en utilisant une fonction booléenne $4 \rightarrow 1$. Le bit de sortie, enfin, est un simple OU EXCLUSIF des trois bits terminaux des registres, et est donc aussi calculable avec une fonction booléenne $4 \rightarrow 1$.

Faisons le compte des CLB utilisés :

module		taille
registre 1	corps	9,5
	rebouclage	0,5
	avancement	0,5
registre 2	corps	11
	rebouclage	0,5
	avancement	0,5
registre 3	corps	11,5
	rebouclage	0,5
	avancement	0,5
bit de sortie		0,5
total		35,5

Sur un Xilinx XC4020, disposant de 784 CLB, on peut faire tenir aisément 16 instances de cette implantation, et conserver de la place pour la génération des états internes à essayer, et la comparaison du flux de sortie avec le flux attendu. Par ailleurs, entre deux cycles, au plus deux CLB sont traversés, et le schéma est globalement très compact, avec un routage peu complexe et régulier. Ces caractéristiques permettent d'atteindre une cadence de fonctionnement de 50 MHz. On obtient, au total, 64 instances de A5/1 en parallèle sur une carte Pamette, pour un total de 3,2 Gbits/s de flux de

sortie calculé.

5.3.2 A5/1 et la recherche exhaustive

Dans la sous-section précédente, j'ai décrit une implantation de A5/1 sur carte Pamette; cette implantation ne serait pas utilisée telle quelle pour une cryptanalyse par recherche exhaustive de A5/1, mais probablement transposée sur une architecture spécialisée, à base de FPGA ou d'ASIC, à la manière de la machine de crackage de DES de l'EFF (cf. section 4.3).

Lors d'une cryptanalyse de A5/1, on veut retrouver la clé de session; mais A5/1 est semi-inversible (cf. section 5.2), c'est-à-dire qu'il est possible de retrouver, à partir d'un état interne donné, les quelques états internes pouvant y aboutir en un nombre donné de tours. L'état interne après chargement de la clé et du numéro de *frame*, mais avant les 100 cycles dont la sortie est ignorée, est une fonction linéaire connue et fixe de la clé. Aussi est-on ramené au problème de retrouver l'état interne d'A5/1 au moment où les premiers bits de flux utile ont été générés.

Si on essaye un état interne donné en regard de n bits de sortie connus, on obtiendra en moyenne 2^{64-n} états internes possibles. Il est inutile de tester chaque état interne face à 64 bits de sortie, puisque seulement un état sur 2^n donnera n bits corrects. Il est tentant de jouer sur le fait qu'un état donné n'a qu'une chance sur deux d'émettre un bit correct à chaque cycle; la moitié des essais échouera dès le premier bit, un quart au second bit, un huitième au troisième bit, *etc.* En moyenne, il suffit de faire tourner A5/1 pendant deux cycles pour chaque essai. Dans la pratique, une telle implantation n'est pas rentable sur du matériel tel que des ASIC ou des FPGA :

- il y a plusieurs dizaines d'instances en parallèle, qui doivent tourner à la même vitesse si on veut partager le compteur d'états internes possibles; un tel compteur a une taille de l'ordre de A5/1, voire supérieure; le surcoût dû à la désynchronisation des instances de A5/1 ne peut donc pas être négligé;
- la mesure par simple chronométrage du temps écoulé entre le lancement et l'annonce d'un état interne correct est plus aléatoire, car il repose sur un comportement moyen du registre qui n'est en rien garanti, surtout sur des espaces d'états internes relativement petits comparativement à l'espace total possible;
- un compteur pouvant suivre le rythme maximal (qui peut impliquer un nouvel essai à chaque cycle) est compliqué (et donc coûteux en place) à mettre au point.

Donc on est obligé de faire un compromis : pour chaque état interne, on essaye sur n bits; les 2^{64-n} états potentiels trouvés sont analysés extérieure-

ment par logiciel. On découpe l'espace des états internes en N blocs ; chaque bloc est chargé dans le matériel de recherche exhaustive, et, comme dans le cas de DES, la mesure est un chronométrage du temps mis pour trouver un état interne « correct », associé à la position du module qui a « trouvé ». Afin de ne pas perdre trop de temps en réinitialisation du système, il faut que chaque bloc soit suffisamment grand (donc N suffisamment petit) pour nécessiter au moins plusieurs secondes pour être parcouru ; mais si plusieurs états potentiels sont dans le même bloc, le premier de ces états masque les autres ; aussi devra-t-on avoir des blocs suffisamment petits (donc N suffisamment grand) pour que le « bon » état interne n'ait que des risques statistiquement faibles de se trouver dans le même bloc qu'un autre état potentiel. Au total, on peut envisager de comparer les flux sur $n = 32$ bits, et découper l'espace interne en 2^{35} blocs de 2^{29} états internes. Il serait difficile de faire substantiellement mieux (« mieux » signifiant « avec un n plus petit »).

Comme on utilise 64 instances d'A5/1 en parallèle, qui chacune nécessite 32 cycles pour effectuer son essai, une carte Pamette tournant à 50 MHz peut essayer 100 millions d'états internes possibles par seconde (ce qui place le temps de parcours d'un bloc de 2^{29} états interne à environ 5 secondes). On notera que la même carte pouvait essayer 25 millions de clés DES par seconde ; si on considère les cartes Pamette à FPGA Xilinx comme des modélisations fidèles de ce qui est possible sur une machine spécialisée à base d'ASIC, on peut conclure qu'il est quatre fois plus facile d'attaquer A5/1 qu'un DES à clé de 64 bits (DES lui-même a une clé de 56 bits, mais une adaptation de la diversification des clés ne change pas le coût de la recherche exhaustive par bit de clé, et permet d'imaginer un DES avec une taille de clé relativement arbitraire). A5/1 devient équivalent, de ce point de vue, à un DES avec clé de 62 bits. La machine de l'EFF coûte environ 100 000 dollars (hors coûts de développement) et peut attaquer 56 bits en trois jours ; une machine spécialisée dans la cryptanalyse brutale d'A5/1, devant résoudre un problème 64 fois plus difficile (passer de 56 à 62 bits, c'est multiplier le nombre d'essais par $2^6 = 64$), coûterait un peu plus de 6 millions de dollars et fournirait son résultat en trois jours au plus (donc, en moyenne, un jour et demi).

Par ailleurs, il s'avère que la clé de session ne fait pas toujours 64 bits : la clé est fournie par l'algorithme A3A8, qui est calculé par la carte SIM du téléphone GSM, et qui est spécifique (en théorie) à l'opérateur utilisé ; mais, dans la pratique, la plupart des opérateurs semblent utiliser le même algorithme, qui fournit une clé de seulement 54 bits (les dix bits restants étant considérés comme nuls). La raison de cet état de fait n'est pas connue publiquement. Si on veut utiliser cette information pour réduire la recherche exhaustive à 2^{54} essais (au lieu de 2^{64} , c'est-à-dire réduire la complexité d'un

facteur 1 000), il faut établir un compteur de clés et passer par les 100 cycles dont la sortie est ignorée; ceci monte le nombre de cycles par essai de 32 à 132. Le gain est donc, par rapport à une recherche exhaustive de l'état interne, de l'ordre de 250. Ceci amène le coût d'une machine de cryptanalyse de A5/1 à environ 25 000 dollars, pour un résultat en un jour et demi. C'est à la portée de beaucoup d'organisations.

5.4 Compromis matériel/logiciel

La situation est la suivante : d'un côté, on sait faire une cryptanalyse logique d'A5/1; cette méthode colle au plus près de l'obstacle théorique, qui est la prédiction de la séquence d'avancement des registres, mais elle nécessite une algorithmique un peu compliquée (pivot de Gauss, backtrack) qui se prête très mal aux implantations matérielles. De l'autre côté, on sait faire tourner remarquablement efficacement des implantations matérielles (ce n'est que logique, d'ailleurs, puisque A5/1 a été prévu pour être rapide, compact et économe en électricité, afin d'être implanté dans des matériels portables).

L'idée du compromis est de tenter de combiner ces deux méthodes : globalement, on va utiliser la cryptanalyse logique présentée précédemment, mais sa partie centrale sera remplacée par une recherche exhaustive sur des sous-espaces d'états internes dégagés par la première partie de la cryptanalyse logique; les états internes potentiels qui subsistent à cette recherche exhaustive peuvent ensuite être essayés logiquement un par un afin d'isoler le bon candidat. Ce travail a été publié dans [74].

Reprenons la cryptanalyse logique : on commence par effectuer n étapes, c'est-à-dire construire des systèmes de $3n$ équations linéaires, en explorant un arbre de taille 4^n et en utilisant n bits de flux de sortie. L'expérience montre que tant que n est inférieur à 21, les $3n$ équations sont quasiment systématiquement linéairement indépendantes, donc, par soucis de simplicité, on ignore le cas où une ou plusieurs de ces équations est éliminée par le pivot de Gauss.

Une fois qu'on dispose de ces $3n$ équations, on peut résoudre logiquement le système en complétant le pivot de Gauss : l'ensemble des solutions est un sous-espace affine de \mathbf{Z}_2^{64} , de dimension $64 - 3n$, ce qui se représente par une base de vecteurs binaires de cette taille, assortie d'un vecteur d'origine; c'est-à-dire qu'on peut calculer, selon la méthode classique de résolution d'équations linéaires, les vecteurs binaires de 64 bits v_i ($1 \leq i \leq 64 - 3n$) et

w tels que les solutions du système de $3n$ équations soient les vecteurs :

$$S = \left\{ w + \sum_{i=1}^{64-3n} \alpha_i v_i \mid \alpha_i \in \mathbf{Z}_2 \right\} \quad (5.5)$$

Cette base est alors envoyée à un matériel spécialisé, par exemple des cartes Pamettes montées dans la station d'accueil qui effectue l'analyse logicielle ; le matériel essaye alors l'ensemble des états internes possibles générés par cette base, puis indique lesquels correspondent aux bits de sortie de référence. Cette recherche exhaustive partielle s'effectue sur un nombre réduit de bits de sortie, comme dans le cas de la recherche exhaustive globale sur l'état interne ; le tri final peut se faire logiquement.

Pour résumer, on obtient ceci :

- 4^n systèmes de $3n$ équations linéaires à 64 inconnues ; chacun est résolu logiquement pour donner un espace affine de solutions caractérisé par $65 - 3n$ vecteurs formant une base et une origine. Chaque vecteur de cet espace donne les n premiers bits de sortie connus.
- $4^n 2^{64-3n}$ essais d'état interne, en matériel, en regard de p bits de sortie ; environ un essai sur 2^p donne un résultat positif.
- $4^n 2^{64-3n} 2^{-p}$ vérifications logicielles finales.

Le coût de la première partie de la cryptanalyse est globalement double du coût de la partie équivalente dans une cryptanalyse purement logicielle (car ici, on veut résoudre chaque système, ce qui implique la deuxième phase du pivot de Gauss, où on réduit chaque équation en fonction des suivantes et non plus des précédentes). Ceci amène à un coût de 800 jours-machine pour $2^{44,3}$ systèmes. La machine considérée est, comme précédemment, une station Compaq XP1000 avec un processeur Alpha 21264 à 500 MHz.

Le nombre de bits p à essayer dans le matériel spécialisé répond à la même problématique que dans le cas de la recherche exhaustive globale ; comme les essais sont découpés en blocs de taille 2^{64-3n} et que chaque essai a une probabilité de 2^{-p} d'être concluant, p doit être tel que la plupart des blocs n'aient aucun essai concluant ; ceci impose que p soit supérieur à $64 - 3n$. Dans la suite, on verra que n sera entre 16 et 20, et donc une valeur réaliste pour p est 24. On considère deux cartes Pamette utilisant des Xilinx XC4020, chacune pouvant calculer 3,2 milliards de cycles d'A5/1 par seconde ; on veut en calculer $4^n 2^{64-3n} p = 2^{64-n} p$ en tout.

On peut imaginer la cryptanalyse comme le passage de la barrière de complexité, à savoir l'analyse de la séquence d'avancement des registres à décalage. La cryptanalyse logicielle franchit cette colline au plus près, en voiture ; la cryptanalyse matérielle par recherche exhaustive survole le tout à haute altitude, en avion à réaction : l'avion est beaucoup plus rapide mais doit

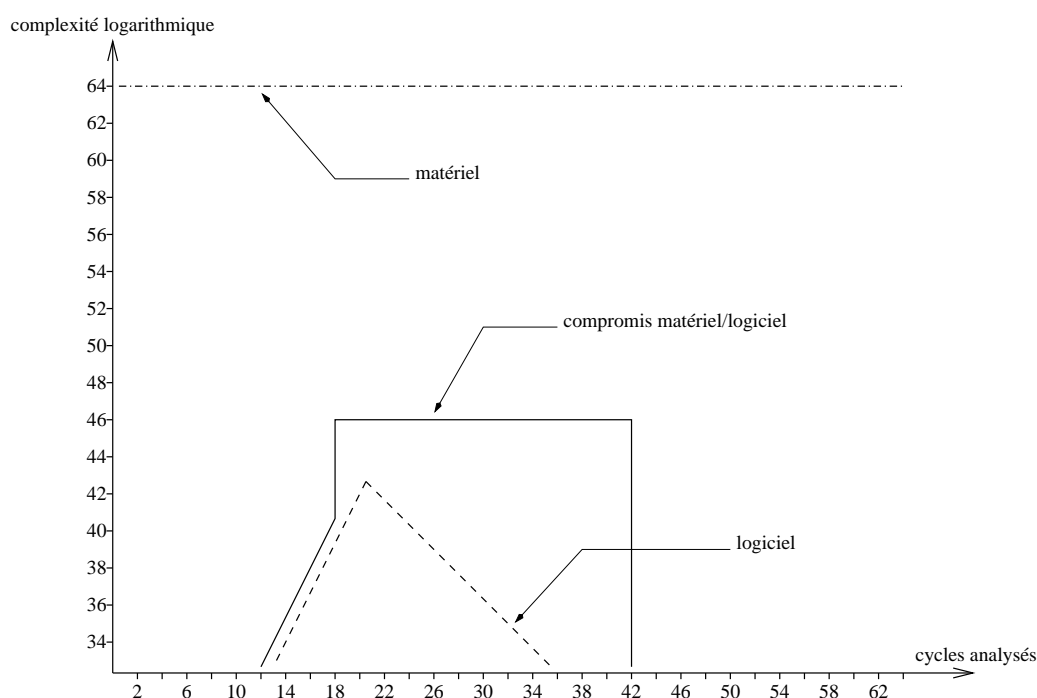


FIG. 5.3 : Le compromis matériel/logiciel

franchir beaucoup plus de distance. Le compromis matériel/logiciel revient à monter à mi-pente avec un camion, et à franchir le reste avec une catapulte. La figure 5.3 illustre ces complexités.

Voici donc les complexités et temps de calcul nécessaires, suivant la valeur de n , pour $p = 24$, sur une station XP1000 dotée de deux cartes Pamette à FPGA XC4020 :

n	partie logicielle		partie matérielle	
	complexité	temps	complexité	temps
16	2^{32}	0,16	2^{48}	12,2
17	2^{34}	0,62	2^{47}	6,11
18	2^{36}	2,48	2^{46}	3,05
19	2^{38}	9,92	2^{45}	1,53
20	2^{40}	39,7	2^{44}	0,76

Les complexités logicielles sont exprimées en nombre de systèmes de $3n$ équations à résoudre ; les complexités matérielles sont en nombre d'états internes à essayer contre $p = 24$ bits de sortie. Les temps sont donnés en jours. Toutes ces complexités et tous ces temps sont des pires cas : en moyenne, la cryptanalyse réussira en moitié moins de temps et d'efforts.

Le meilleur compromis est obtenu pour $n = 18$: le temps de calcul est limité par le matériel, et est en moyenne de un jour et demi. Cela forme 2^{36} systèmes, chacun d'entre eux comporte 54 équations linéaires, et chaque espace de clés est de taille $2^{10} = 1024$. Comme on compare sur $p = 24$ bits de sortie, on obtient $2^{36+10-24} = 2^{22}$ états internes potentiels, c'est-à-dire environ 4 millions. Ceux-là sont facilement vérifiables en quelques minutes par la station XP1000. D'ailleurs, la station n'est utilisée pendant l'opération qu'environ 80% du temps, et peut donc remplir quelques usages annexes. Les données transférées entre la station et les cartes Pamettes sont en quantité faible (puisqu'on ne transfère que les bases des espaces affines), et très largement dans les possibilités du bus PCI sur lequel les Pamettes sont connectées.

On obtient donc un temps de cryptanalyse de un jour et demi, avec un matériel coûtant environ 20 000 dollars en tout. Pour le même prix, on aurait pu acheter deux stations, qui auraient mené à bien une cryptanalyse logicielle en une centaine de jours. Le compromis logiciel/matériel améliore donc le rendement de la cryptanalyse d'un facteur 60 par rapport à une cryptanalyse purement logicielle, en utilisant du matériel qui reste polyvalent (car les Pamettes peuvent servir à d'autres usages). Le coût est rendu équivalent à celui d'une recherche exhaustive sur une clé de 54 bits, sans même profiter de cet avantage de 10 bits.

5.5 Conclusion

A5/1 est un algorithme particulièrement adapté au matériel, et d'une grande « élégance » ; toute sa force réside dans l'avancée irrégulière des registres à décalage utilisés. Sa taille est hélas trop petite, car la résistance d'A5/1 est celle de sa séquence d'avancée, qui est exponentielle en les deux tiers seulement de la taille de son état interne. Cet état ne faisant que 64 bits, A5/1 devient attaquable en une complexité de l'ordre de 2^{43} opérations, et diverses attaques par précalcul peuvent être mises en place (c'est le principe même de l'attaque de Biryukov, Shamir et Wagner). J'ai présenté une cryptanalyse logicielle qui illustre le degré de sécurité interne ; j'ai également décrit une implantation sur matériel dédié, et discuté du prix d'une machine spécialisée. Enfin, j'ai explicité l'idée du compromis entre le matériel et le logiciel : il s'agit, fondamentalement, d'appliquer une méthode logicielle en l'appuyant dans une partie de ses calculs par un matériel spécifique, éventuellement reconfigurable, beaucoup plus rapide pour les phases d'analyse systématique.

Je rappelle que je ne prétends rien quant à la facilité d'interception d'une communication GSM par voie radio ; la fréquence utilisée est modifiée plu-

sieurs fois au cours d'une même conversation, et le suivi de ces fréquences ne peut se faire que grâce aux informations qui sont chiffrées par A5/1. Aussi une interception et cryptanalyse radio ne devraient pouvoir se faire qu'en enregistrant une large gamme de fréquences, et je n'ai pas d'idée quant à la difficulté de la chose. En tous cas, le système de chiffrement des téléphones GSM est trop faible pour qu'on puisse avoir réellement confiance en lui, même s'il n'est tout de même pas aussi facilement espionnable qu'un téléphone filaire, qui, lui, n'est pas chiffré du tout.

Toujours est-il qu'on souhaite que les normes futures de téléphonie disposent d'un algorithme moins sous-classé ; le principe de construction d'A5/1 semble sain, et monter son état interne à 128 ou 192 bits permettrait d'atteindre des niveaux de sécurité très acceptables.

Chapitre 6

Conclusion

La cryptologie est désormais une science industrielle, où la qualité et les performances des implantations des algorithmes de chiffrement ont une importance cruciale. Dans ce travail, j'ai décrit les matériels qui sont amenés à réaliser ces fonctions, leurs capacités et leurs faiblesses. Au cours de ma thèse, j'ai étudié comment adapter les algorithmes cryptographiques à ces différents contextes :

- pour les processeurs centraux polyvalents, en utilisant notamment la multiplication, comme cela a été fait dans DFC (cf. section 2.3.2) ;
- pour les processeurs RISC, à l'aide de techniques de code orthogonal, automatisées grâce à **bsc** (cf. section 3.3) et appliquées au nouvel algorithme de chiffrement symétrique FBC (cf. section 3.4) ;
- pour les FPGA, dans le cadre d'une implantation de la cryptanalyse de DES par recherche exhaustive de la clé (cf. section 4.3)

J'ai par ailleurs travaillé sur la sécurité de DES (cf. section 1.4) et d'A5/1 (cf. chapitre 5) et montré comment, dans ce dernier cas, l'usage conjoint de matériel reconfigurable (FPGA) et de stations de travail génériques pouvait rendre certaines cryptanalyses extrêmement efficaces.

Au travers de ces recherches, j'observe les tendances générales suivantes :

- Économiquement, l'avenir est aux systèmes polyvalents et reconfigurables. Les processeurs génériques sont de plus en plus puissants, et les implantations sur FPGA peuvent rivaliser avec celles sur ASIC.
- Scientifiquement, la sécurité des systèmes de chiffrement symétrique est un domaine désormais bien connu ; on manque encore de preuves de sécurité, mais on sait fabriquer des algorithmes efficaces et qui résistent à l'analyse. Sur les quinze candidats présentés à l'AES, seuls deux ont été « cassés », et uniquement avec des attaques académiques, inapplicables dans un cas réel. La comparaison entre divers cryptosystèmes se fait plutôt sur les performances et l'adaptabilité à divers matériels.

- Industriellement, il reste encore à faire la liaison entre la recherche en cryptographie et les applications. Trop souvent, les cryptosystèmes utilisés réellement sont des versions amoindries ou modifiées des algorithmes issus de la recherche publique, et leur sécurité en est très affectée. Par exemple, l'état interne d'A5/1 est trop petit pour réaliser un chiffrement solide, bien que la structure d'A5/1 soit très correcte de ce point de vue.

La recherche publique en cryptographie symétrique est appelée à se rapprocher de l'industrie, à développer de nouvelles méthodes d'optimisation automatique, et à s'intéresser à des problèmes de plus en plus appliqués. On constate déjà un intérêt accru envers la résistance aux attaques utilisant des canaux cachés[43], c'est-à-dire des fuites d'information à travers le fonctionnement physique du matériel exécutant le cryptosystème ; l'attaque la plus aboutie est la DPA[56] (*Differential Power Analysis*) qui travaille sur la consommation électrique d'une carte à puce. Bien que ce soit un problème matériel, c'est algorithmiquement qu'on recherche la solution, car une adaptation logicielle est, dans ce cas-là aussi, bien moins coûteuse qu'une contre-mesure physique. Cette constatation sera, à mon sens, le fil directeur de la recherche cryptologique des prochaines années.

Bibliographie

- [1] R. Anderson, *A5 (Was: HACKING DIGITAL PHONES)*, (1994), article Usenet publié sur sci.crypt, alt.security et uk.telecom.
- [2] R. Anderson, E. Biham, et L. Knudsen, *Serpent: A New Block Cipher Proposal*, Proceedings of Fast Software Encryption 98, Springer-Verlag, 1998, pp. 222–238.
- [3] (anonyme), *Scramdisk*, 2000, <http://www.scramdisk.clara.net/>
- [4] I. Ben-Aroya et E. Biham, *Differential cryptanalysis of Lucifer*, Proceedings of Crypto'93, Springer-Verlag, 1993, pp. 187–199.
- [5] E. R. Berlekamp, *Algebraic Coding Theory*, McGraw-Hill, New York, 1968.
- [6] E. Biham, *A Fast New DES Implementation in Software*, Proceedings of Fast Software Encryption 97, Springer-Verlag, 1997, pp. 260–272.
- [7] E. Biham et A. Biryukov, *An Improvement of Davies' Attack on DES*, Proceedings of Eurocrypt'94, Springer-Verlag, 1994, pp. 461–467.
- [8] E. Biham, A. Biryukov, et A. Shamir, *Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials*, Proceedings of Eurocrypt'99, Springer-Verlag, 1999, pp. 12–23.
- [9] E. Biham et A. Shamir, *Differential cryptanalysis of DES-like cryptosystems*, Journal of Cryptology, vol. 4 (1991), no. 1, pp. 3–72.
- [10] E. Biham et A. Shamir, *Differential cryptanalysis of Feal and N-Hash*, Proceedings of Eurocrypt'91, Springer-Verlag, 1991, pp. 1–16.
- [11] E. Biham et A. Shamir, *Differential cryptanalysis of Snefru, Khafre, REDOC-II, LOKI and Lucifer*, Proceedings of Crypto'91, Springer-Verlag, 1992, pp. 156–171.
- [12] E. Biham et A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard*, Springer-Verlag, 1993.
- [13] E. Biham et A. Shamir, *Differential cryptanalysis of the full 16-round DES*, Proceedings of Crypto'92, Springer-Verlag, 1993.

- [14] A. Biryukov, A. Shamir, et D. Wagner, *Real Time Cryptanalysis of A5/1 on a PC*, Proceedings of Fast Software Encryption 2000, à paraître.
- [15] L. Blum, M. Blum, et M. Shub, *Comparison of two pseudo-random number generators*, Proceedings of Crypto'82, Springer-Verlag, 1983, pp. 61–79.
- [16] M. Briceno, I. Goldberg, et D. Wagner, *A pedagogical implementation of A5/1*, 1999, <http://www.scard.org/gsm/body.html>
- [17] F. Chabaud et S. Vaudenay, *Links between differential and linear cryptanalysis*, Proceedings of Eurocrypt'94, Springer-Verlag, 1994, pp. 363–374.
- [18] Hewlett-Packard Company, *PA-RISC References*, 2000, <http://www.cpus.hp.com/techreports/parisc.shtml>
- [19] D. Coppersmith, *The Data Encryption Standard (DES) and its Strength Against Attacks*, Tech. Report RC18613, IBM, 1992.
- [20] D. Coppersmith, *The development of DES*, août 2000, lecture invitée à Crypto'2000.
- [21] Digital Equipment Corporation, *Alpha Documentation Library*, 1999, <http://gatekeeper.dec.com/pub/Digital/info/semiconductor/literature/dsc-library.html>
- [22] Intel Corporation, *Intel Architecture Optimization Manual*, 1997, <http://developer.intel.com/design/pentium/manuals/242816.htm>
- [23] Intel Corporation, *Intel Architecture Processors*, 1997, <http://developer.intel.com/design/processor/>
- [24] Intel Corporation, *Itanium Processor Family*, 1997, <http://developer.intel.com/design/ia-64/>
- [25] J. Daemen et V. Rijmen, *AES Proposal: Rijndael*, Proceedings from the First Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST), 1998.
- [26] D. Davies et S. Murphy, *Pairs and Triplets of DES S-Boxes*, Journal of Cryptology, vol. 10 (1997), no. 3, pp. 195–206.
- [27] *Data Encryption Algorithm*, Tech. Report X3.92, American National Standards Institute (ANSI), 1981.
- [28] S. Dean, *Disk and File Shredders: A Comparison*, 2001, http://www.fortunecity.com/skyscraper/true/882/Comparison_Shredders.htm

- [29] *Data Encryption Standard*, Tech. Report FIPS 46, National Institute of Standards and Technology (NIST), 1977.
- [30] W. Diffie et M. Hellman, *Exhaustive Cryptanalysis of the NBS Data Encryption Standard*, Computer, vol. 10 (1977), no. 6, pp. 74–84.
- [31] distributed.net, *Le projet DES*, 1998, <http://www.distributed.net/des/>
- [32] *Digital Signature Standard*, Tech. Report FIPS 186-2, National Institute of Standards and Technology (NIST), 2000.
- [33] M. Slusarczyk *et al*, *Emergency Destruction of Information Storing Media*, Tech. report, Institute for Defense Analysis, 1987.
- [34] H. Feistel, *Cryptography and Computer Privacy*, vol. 228, mai 1973, pp. 15–23.
- [35] A. Fog, *How to optimize for the Pentium family of microprocessors*, 1999, <http://www.agner.org/assem/>
- [36] Electronic Frontier Foundation, *EFF Homepage*, 2001, <http://www.eff.org/>
- [37] G. Garon et R. Outerbridge, *DES Watch: An examination of the Sufficiency of the Data Encryption Standard for Financial Institution Information Security in the 1990's*, Cryptologia, vol. 15 (1991), no. 3, pp. 177–193.
- [38] H. Gilbert, M. Girault, T. Pornin, G. Poupard, J. Stern, et S. Vaudenay, *Decorrelated Fast Cipher: an AES Candidate*, Proceedings from the First Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST), 1998.
- [39] I. Goldberg et D. Wagner, *Architectural Considerations for Cryptanalytic Hardware*, Tech. report, University of Berkeley, CS252 Report, 1996.
- [40] J. D. Golić, *Cryptanalysis of Alleged A5 Stream Cipher*, Proceedings of Eurocrypt 97, Springer-Verlag, 1997, pp. 239–255.
- [41] L. Granboulan, *Flaws in differential cryptanalysis of Skipjack*, Proceedings of Fast Software Encryption 2001, Springer-Verlag, 2001.
- [42] P. Gutmann, *Secure Deletion of Data from Magnetic and Solid-State Memory*, Proceedings of USENIX Security Symposium, 1996.
- [43] C. Hall, J. Kesley, B. Schneier, et D. Wagner, *Side Channel Cryptanalysis of Product Ciphers*, Proceedings of ESORICS'98, Springer-Verlag, 1998, pp. 97–110.

- [44] M. Hellman, *DES will be totally insecure within 10 years*, IEEE Spectrum, vol. 16 (1979), pp. 32–39.
- [45] M. E. Hellman, *A cryptanalytic time-memory trade-off*, IEEE Transactions on Information Theory, vol. 26, 1980, pp. 401–406.
- [46] V. Kabanets et J.-Y. Cai, *Circuit Minimization Problem*, Symposium on Theory of Computing (STOC), ACM, 2000, pp. 73–79.
- [47] D. Kahn, *The codebreakers*, MacMillan, 1967.
- [48] J.-P. Kaps et C. Paar, *Fast DES Implementation for FPGAs and its Application to a Universal Key-Search Machine*, Selected Areas in Cryptography'98, Springer-Verlag, 1998.
- [49] L. R. Knudsen, *Truncated and higher order differentials*, Proceedings of Fast Software Encryption 95, Springer-Verlag, 1995, pp. 196–211.
- [50] L. R. Knudsen et W. Meier, *Improved Differential Attacks on RC5*, Proceedings of Crypto'96, Springer-Verlag, 1996, pp. 216–228.
- [51] L. R. Knudsen, M. J. B. Robshaw, et D. Wagner, *Truncated Differentials and Skipjack*, Proceedings of Crypto'99, Springer-Verlag, 1999, pp. 165–180.
- [52] D. Knuth, *The Art Of Computer Programming*, vol. 3, Addison-Wesley, 1969, pp. 7 et 573.
- [53] D. Knuth, *The Art Of Computer Programming*, vol. 1, Addison-Wesley, 1969, pp. 177–179.
- [54] D. Knuth, *The Art Of Computer Programming*, vol. 2, Addison-Wesley, 1997 (2nd ed.), p. 139.
- [55] P. Kocher, *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, Proceedings of Crypto'96, Springer-Verlag, 1996, pp. 104–113.
- [56] P. Kocher, J. Jaffe, et B. Jun, *Differential Power Analysis*, Proceedings of Crypto'99, Springer-Verlag, 1999, pp. 388–397.
- [57] M. Kwan, *Bitslice DES*, 2000, <http://www.darkside.com.au/bitslice/>
- [58] Nallatech Ltd., *Nallatech – The Algorithms To Hardware Company*, 2000, <http://www.nallatech.com/>
- [59] Software Professionals Ltd, *Encryption for the Masses*, 2001, <http://www.e4m.net/>
- [60] M. Luby et C. Rackoff, *How to construct pseudo-random permutations from pseudo-random functions*, Proceedings of Crypto'85, Springer-Verlag, 1985.

- [61] International Business Machines, *IBM Microelectronics*, 2000, <http://www.chips.ibm.com/>
- [62] M. Matsui, *Linear cryptanalysis method for DES cipher*, Proceedings of Eurocrypt'93, Springer-Verlag, 1994, pp. 386–397.
- [63] M. Matsui, *The first experimental cryptanalysis of the Data Encryption Standard*, Proceedings of Crypto'94, Springer-Verlag, 1994, pp. 1–11.
- [64] M. McLoone et J. V. McCanny, *A High Performance FPGA Implementation of DES*, Signal Processing Systems Design and Implementation, IEEE, 2000.
- [65] *The MD5 Message Digest Algorithm*, Tech. Report RFC 1321, MIT Laboratory for Computer Science and RSA Data Security Inc., 1992.
- [66] Sun Microsystems, *SPARC: A Microprocessor with a Past and a Future*, 2000, <http://www.sun.com/microelectronics/sparc/sparc.html>
- [67] Sun Microsystems, *UltraSPARC Microprocessor Family*, 2000, <http://www.sun.com/microelectronics/UltraSPARC/>
- [68] *A Guide to Understanding Data Remanence in Automated Information Systems*, Tech. report, National Computer Security Centre, 1991.
- [69] National Institute of Standards et Technology (NIST), *AES Development Effort*, 2000, <http://csrc.nist.gov/encryption/aes/>
- [70] T. Pornin, *Optimal Resistance Against the Davies and Murphy Attack*, Proceedings of Asiacrypt'98, Springer-Verlag, 1998, pp. 148–159.
- [71] T. Pornin, *Automatic Software Optimization of Block Ciphers using Bitslicing Techniques*, non publié, 1999.
- [72] T. Pornin, *bsc, le compilateur bitslice*, 1999, <ftp://ftp.ens.fr/pub/di/users/pornin/bsc-current.tar.gz>
- [73] T. Pornin, *Transparent Harddisk Encryption*, Proceedings of CHES'2001, Springer-Verlag, 2001.
- [74] T. Pornin et J. Stern, *Software-Hardware Trade-Offs: Application to A5/1 Cryptanalysis*, Proceedings of CHES'2000, Springer-Verlag, 2000, pp. 318–327.
- [75] N. Provos, *Encrypting Virtual Memory*, Proceedings of USENIX Security Symposium, 2000.
- [76] W. Rudin, *Fourier Analysis on Groups*, Interscience Publishers Inc., 1962.
- [77] A. E. Sale, *Colossus and the German Lorenz Cipher*, Proceedings of Eurocrypt'2000, Springer-Verlag, 2000.

- [78] *Secure Hash Standard*, Tech. Report FIPS 180-1, National Institute of Standards and Technology (NIST), 1995.
- [79] C. E. Shannon, *The synthesis of two-terminal switching circuits*, vol. 28, 1949, pp. 59–98.
- [80] J. L. Smith, *The Design of Lucifer, A Cryptographic Device for Data Communications*, Tech. Report RC3326, IBM, 1971.
- [81] H. Touati et M. Shand, *PamDC: a C++ Library for the Simulation and Generation of Xilinx FPGA Designs*, 1999,
<http://research.compaq.com/SRC/pamette/PamDC.pdf>
- [82] Inc. Tripwire, *Tripwire*, 2001, <http://www.tripwire.com/>
- [83] S. Vaudenay, *Provable Security for Block Ciphers by Decorrelation*, Proceedings of STACS'98, Springer-Verlag, 1998, pp. 249–275.
- [84] S. Vaudenay, *Feistel Ciphers with L2-Decorrelation*, Proceedings of SAC'98, Springer-Verlag, 1999, pp. 1–14.
- [85] V. Veeravalli, *Detection of Digital Information from Erased Magnetic Disks*, Master's thesis, Carnegie-Mellon University, 1987.
- [86] D. Wagner, *The Boomerang Attack*, Proceedings of Fast Software Encryption 99, Springer-Verlag, 1999, pp. 156–170.
- [87] M. J. Wiener, *Efficient DES Key Search*, Proceedings of Crypto'93, Springer-Verlag, 1994.
- [88] Xilinx, *Xilinx: Programmable Logic Devices*, 2000,
<http://www.xilinx.com/>
- [89] A. Yang, *The Algorithm (Stream Cipher) RC4 (ARC-4)*, 2000,
<http://www.achtung.com/crypto/rc4.html>
- [90] E. Young, *Bibliothèque SSLeay*, 1998,
<ftp://pub/crypt/cryptography/libs/SSL/SSLeay-0.9.0b.tar.gz>

Résumé

La cryptographie symétrique est l'étude des algorithmes de chiffrement de données, dont la clé de déchiffrement est identique à la clé de chiffrement. Ces algorithmes servent à rendre inintelligible une information, sauf pour les détenteurs d'une certaine convention secrète, afin de permettre son transfert sur des lignes de communication potentiellement espionnées, voire activement corrompues. Cette thèse s'intéresse au problème de l'implantation et de l'optimisation de ces algorithmes sur divers matériels informatiques, incluant les ordinateurs génériques et les FPGA (circuits reprogrammables). Les algorithmes classiques (DES, RC4, A5/1...) sont abordés, ainsi que des variantes de ces algorithmes afin de mieux s'adapter au matériel, sans sacrifier la sécurité; une nouvelle technique de programmation, dite « bitslice » ou « code orthogonal », est décrite, ainsi que les outils semi-automatiques de support de cette technique. Enfin, il est montré comment une utilisation adéquate de FPGA, en complément de stations de travail, permet d'accélérer grandement des travaux de cryptanalyse.

Abstract

Symmetric cryptography is the science of algorithms enciphering and deciphering data with the same secret key. Those algorithms are used to obfuscate data and make it unreadable to anyone but those who hold the key and can de-obfuscate the data and return it to its original layout; thus, they allow safe data transfers through unsafe and potentially hostile networks. This thesis is about the implementation and optimization of such algorithms on various devices, including generic purpose workstations and FPGAs (reprogrammable chips). The standard algorithms such as DES, RC4, A5/1 are studied, as well as other variations designed to match closely the intended supporting hardware, without sacrificing security. A new implementation technique, called "bitslice" or "orthogonal code", is described, along with an automatic tool to support such coding. Finally, this work explains how a fine-tuned mix between FPGA and generic workstations can greatly enhance cryptanalysis efficiency.