Thèse de doctorat de l'université Pierre et Marie Curie

Spécialité Informatique

EDITE de Paris

présentée par

María Naya-Plasencia

pour obtenir le grade de

Docteur de l'université Pierre et Marie Curie

Chiffrements à flot et fonctions de hachage : conception et cryptanalyse

soutenance prévue le 16 novembre 2009

devant le jury composé de

Anne Canteaut	INRIA Paris-Rocquencourt	Directrice de thèse
Henri Gilbert	Orange Labs	Rapporteur
Willi Meier	FHNW (Suisse)	Rapporteur
Pascale Charpin	INRIA Paris-Rocquencourt	Examinatrice
Pierre-Alain Fouque	ENS	Examinateur
Thomas Johansson	Lund University (Suède)	Examinateur
Antoine Joux	Université de Versailles - DGA	Examinateur
Michèle Soria	Université Pierre et Marie Curie	Examinatrice

María Naya-Plasencia

Chiffrements à flot et fonctions de hachage : conception et cryptanalyse

INRIA-équipe SECRET
Domaine de Voluceau
78153 Le Chesnay

Remerciements

Tout d'abord je veux remercier Anne Canteaut infiniment. Premièrement pour avoir été meilleure que la meilleure directrice de thèse que je pouvais imaginer. Ensuite, pour être non seulement une directrice de thèse exceptionelle, mais aussi une personne exceptionelle, et pour avoir offert son support permanent et m'avoir permis de connaître le plaisir de travailler avec elle. Je lui suis très reconnaissante de m'avoir montré son savoir-faire. Sans elle, cette thèse n'aurait jamais été lisible, et encore moins en français. Je veux aussi remercier et m'excuser auprés de Damien et Alice pour tout le temps d'Anne que je leur ai volé, surtout pendant qu'elle relisait (et re-relisait) ma thèse.

Je souhaite remercier Henry Gilbert et Willi Meier de m'avoir fait l'immense honneur d'être mes rapporteurs de thèse. Merci à Henri Gilbert pour sa lecture et ses commentaires détaillés qui ont aidé à améliorer le manuscrit, malgré son calendrier très chargé. Je tiens aussi à remercier Willi Meier pour son accueil pendant mon séjour au FHNW, ses travaux qui m'ont tant intéressée et inspirée et pour ses précieuses discussions et ses efforts pour mon futur post-doctorat (que j'ai hâte de commencer) ; et aussi pour avoir pu se déplacer à Paris pour ma soutenance, malgré son agenda très chargé.

Je tiens aussi à remercier Pascale Charpin, Pierre-Alain Fouque, Thomas Johansson, Antoine Joux et Michèle Soria d'avoir accepté d'appartenir à mon jury de thèse. Merci à Pascale Charpin pour son intêret, pour son éternelle gentillesse. Merci aussi à Pièrre-Alain Fouque pour ses remarques sur le manuscrit. Merci à Thomas Johansson pour ses travaux, qui ont inspiré la première partie de ma thèse ainsi que d'avoir pu se déplacer jusqu'à Paris pour une soutenance en français. Merci à Antoine Joux pour ses cours inoubliables du Master de Versailles. Merci à Michèle Soria de m'avoir fait l'honneur d'assister a ma soutenance.

Cette thèse n'aurait évidemment pas été la même sans l'équipe-projet SECRET de l'INRIA (ancien projet CODES). Merci à tous pour l'ambiance, pour rendre tous les jours si agréables, pour les cafés, pour faire en sorte que le trajet jusqu'à Rocquencourt vale toujours la peine. Merci aux permanents d'être si accessibles et proches, par ordre des bureaux : Pascale Charpin, Anne Canteaut, Nicolas Sendrier, Jean-Pierre Tillich et Daniel Augot (et sa famille pour les inoubliables barbecues). Merci au bureau 1, qui exprime le jumelage existant entre Clermont et Oviedo, et pour avoir été le meilleur bureau du monde : Scarab, Yann, Raghav, Andrea, Céline, Ayoub, Gregory. Merci à tous les autres que j'ai eu la chance de croiser : les deux Stéphanes, Alexander, Sumanta, Cédric, Deepak, Michaël, Françoise, Pierre, Marine, Marion, Ludovic, Bhaskar, Fabien, Denise, Christina, Mamdouh,

Claude, Raphael, Maxime, Vincent, Benoit, Anne, Sunandan, Christophe, Fred et Mathieu. Je vous dois un gâteau. Un énorme merci à Christelle, d'être si efficace, de résoudre toujours les soucis adminitratifs et pour sa bonne humeur. Merci aussi à Matthis et à Laurine de leurs visites au projet et de leurs dessins qui ont rendu mon bureau beaucoup plus joli. Je tiens à remercier particulièrement Andrea, Yann, Scarab, Alexander, Fred, Stéphane, Christelle, les deux Ma(t)thieux et Geneviève pour tant de moments passés ensemble hors du projet. Je veux remercier Willi Meier, Jean-Philippe Aumasson, Thomas Peyrin, Eric Brier, Gaëtan Leurent, Andrea Roeck, Yann Laigle-Chapuy, Kristian Matusiewicz, Ivica Nikolic, Yu Sasaki et Martin Schlaffer, avec lesquels travailler a été un veritable plaisir.

Il est très important pour moi de remercier mes co-auteurs de Shabal : Emmanuel, Anne, Benoit, Christophe, les trois Thomas, Aline, Jean-François, Pascal, Jean-René, Celine et Marion, pour la tonne de réunions, de repas, d'emails et de bons moments qu'on a partagé.

Un grand merci à tous les gens que j'ai rencontrés aux journées C2 pour les parties de loup garou jouées : Pierre-Louis, Léonard, Patrick, Nadia, Iwen, Guilhem, Aurelie, Damien,... et tant d'autres.

Je veux remercier mon professeur de l'INT Abdallah M'hamed de m'avoir introduit à la cryptographie. Merci aux professeurs du Master d'Algèbre Appliquée de Versailles : Vincent Cossard, Louis Goubin, Mireille Martin-Deschamps, Jacques Patarin. Merci à Nicolas Courtois de m'avoir suggéré de contacter Anne Canteaut pour mon stage de Master II. Je veux aussi remercier Amparo Sabater Fúster du CESIC de Madrid de son accueil pendant mon séjour à Madrid et de m'avoir motivé pour commencer cette thèse.

Merci à "los Pumas", qui ont fait en sorte que les soirées des lundis ou mardis (principalement) aient moins l'air d'un début de semaine. Pour toutes les perruques gagnées, et pour tout ce que nous avons appris en musique (ou pas) : Yann, Sylvain, Anne, Marion, Marcio, Gaëtan, Christophe, Axel, Thomas, Celine, Vincent, avec une mention spéciale à Joana, sans qui cette soutenance de thèse n'aurait jamais été possible, entre autres parce que le dossier de soutenance ne serait jamais arrivé à temps. Je tiens à remercier spécialement aussi Bea, Lore, Andrea, Fred et Scarab qui sont partis à "l'étranger" avant la fin de ma thèse et m'ont beaucoup manqué. Un grand merci à Bea pour tous les années inoubliables que nous avons passées ensemble à Madrid et à Paris, et Lore d'avoir décidé de venir à Paris à la place de l'Oklahoma. Merci aussi à Antoine, d'être toujours si intéressant. Merci beaucoup à la famille de Fab d'être si accueillante et pour tous les moments très agréables passés ensemble. Je tiens à remercier Marie-Paule, pour tous les repas chez elle, d'être venue à ma soutenance, et pour les toasts.

Je tiens à remercier toute ma famille (gracias mil). Merci à Andresín d'avoir été la joix de l'année. Merci à mes grand parents Anita, Eugenio y Cruz de leur amour inconditionel, de leur support, de croire en moi, d'être les meilleurs. Je veux remercier "mi abuela Cruz" pour tout ce qu'elle nous a apporté, elle me manque énormement. Je veux remercier mes parents Ana y Josete et mon frère Guille ; dont je suis si fière. Merci de m'avoir fait sentir chez moi à travers le téléphone, de m'avoir poussé toujours quand il le fallait, de savoir comment me remonter le moral. D'avoir TOUJOURS été là pour moi. D'être un modèle pour moi. De m'avoir appris à chercher les choses vraiment importantes dans la vie. Merci à Guille pour nos conversations interminables au télephone qui me font toujours

raccrocher avec un énorme sourire. Merci à tous mes oncles, tantes, cousins et cousines, particulièrement Miguel qui a pu venir à Paris pour ma soutenance, malgré la date déjà prise pour lui.

Finalement, parce qu'habituellement ça se fait comme ça, un grand merci à Fab pour m'avoir supportée quand j'étais insupportable, pour avoir été toujours là pour moi, pour ses salades et ses essais de "tortillas" qui m'ont alimentée pendant la rédaction de cette thèse. Merci de m'avoir fait rigoler quand j'en pouvais plus, et avoir rechargé mes forces tant de fois comme j'en ai eu besoin. Merci de l'intérêt montré pour mes sujets de recherche, de ses remarques, de nos discussions, de m'avoir permis de partager tout ceci avec lui,... et tant d'autres choses. Merci de tout mon coeur, Tortillo.

Overview

Symmetric cryptography is one of the two big branches of cryptography. It essentially includes three types of algorithms : stream ciphers, block ciphers and hash functions. During my thesis I have worked mostly on stream ciphers and hash functions. Stream ciphers are encryption algorithms that cipher the message progressively, as it is available. They are competitive under hardware or software constraints. Hash fonctions have lots of applications, most of themrelated to authenticity. They take a message of a variable length and produce a value of a fixed length.

Recently, both have been the subjet of two international competitions : eSTREAM, launched in 2004 by the European network of excellence ECRYPT for stream ciphers, and SHA-3, started in 2008 by the American Institute of standards and technology (NIST) for hash functions. These two competitions have made a public call for submissions of algorithms ; after a public phase dedicated to the analysis of the received candidates, they select one (or more than one, in the case of eSTREAM) whose use would be recommended. My thesis has taken place in the context of these two competitions, as it has started during the evaluation process of eSTREAM (that ended on 2008) and it finishes one year after the start of SHA-3. For this reason, I have worked mostly on the cryptanalysis and design of those two families of algorithms. This thesis is then divided in two parts.

The first part focuses on stream ciphers. We present attacks against the different versions of a stream cipher algorithm that was submitted to the eSTREAM projet : Achterbahn. Most notably, we describe the attacks that we have proposed on the last versions of the algorithm and that have been presented at the international workshops *FSE 2007, SASC 2007 et WEWoRC 2007* [NP07a, NP07b, NP08]. Also in this first part of the thesis, we present a more theoretical work that has been launched by the previous attacks : they use a parity-check relation to detect a bias on the keystream, and this bias is most of the times hard to compute. We have found several ways of computing these biases efficiently and of building the best possible parity-check relations in certain cases. This will help us to better build the attacks while it gives us some criteria for the design of the type of stream ciphers that we have analysed. Parts of these results have been presented at the *IEEE International Symposium on Information Theory - ISIT 2009* [CNP09a].

The second part of the thesis is naturally on hash functions. First, we present the algorithm that we have designed and submitted to the SHA-3 competition as our proposal

to the next hash standard. It has passed the first two rounds, and it is still on the competition [BCCM+09a]. After presenting the algorithm, we thoroughly describe the analysis that we have performed on its internal permutation, which are the best known analysis for the moment. This study appears in Section 5.5. Next, we present the attacks of several hash functions that have been submitted to the SHA-3 competition. These attacks, using mostly different techniques, are against the hash functions MCSSHA-3 (and its different versions), Ponic, Maraca, CubeHash, Essence and Lane. All of these attacks have been sent as communications to the NIST diffusion list, and some of them have already been presented at international workshops and conferences : *WEWoRC 2009* [ANP09], *the International conference on finite fields and their applications - Fq9* [CNP09c], *the Australasian Conference on Information Security and Privacy - ACISP 2009* [ABM+09] and *Asiacrypt 2009* [MNPN+09].

Part I

This is the part on stream ciphers. In the first half, we propose several attacks against Achterbahn and its different versions that are summarized in Table 2.1. After those attacks, a new version of Achterbahn with a keystream limit of 2^{44} bits has been proposed. There are no known attacks against this last one. From a general point of view, these attacks have contributed new tools for correlation attacks against stream ciphers :

1. as it is explained in the second half of this part, when parity-check equations are built with $t+1$ variables (when the combination function is t-resilient), we can show that it is always better to use linear approximations,
2. we have proposed a method for reducing the needed keystream length using some bits that we did not use before because of decimation and
3. we have introduced an algorithm that allows to speed up the search in some cases where the registers can be treated separately.

In the second part, we propose a new vision of parity-check equations, which considers these relations independently from an approximation of the involved function, in opposition of what has always been done. The biggest and most immediate impact of our work is the reduction of the complexity of computation of the associated parity-check bias in all cases ; in some cases computing the bias is possible even by hand while the previously known algorithm was infeasible. This results are mainly based on the following theorems.

Theorem 1. *Let* $\boldsymbol{x_1}, \ldots, \boldsymbol{x_v}$ *be* v *sequences with minimal periods* $T_1, \ldots, T_v$*, and* f *a Boolean function of* v *variables. Let*

$$\mathcal{T} = \{\sum_{i=1}^{m} c_i M_i, \;\; c_i \in \{0,1\}\}$$

where $M_i = q_i \mathrm{lcm}(T_{\ell_i+1}, \ldots, T_{\ell_{i+1}})$ *with* $q_i > 0$*,* $\ell_1 = 0$ *et* $\ell_{m+1} = n$*. We consider that* $\mathcal{T}$ *does not contain multiples of* T_j*, for any* $n < j \leq v$*.*

Then, we have that

$$\mathcal{E}(PC_{f,\mathcal{T}}) = \frac{1}{2^{n2^{m-1}}} \sum_{a \in \mathbf{F}_2^{n2^{m-1}}} \prod_{c=0}^{2^m-1} \mathcal{E}(f_{|\chi(c,a)}),$$

where $f_{|b}$ is the restriction of f when the variables of indices $j_1, \ldots, j_n$ are fixed to the bits of $b \in \mathbf{F}_2^n$, and where $\chi(c,a)$ is an n-bit vector defined on Page 47.

Theorem 2. *Let $\boldsymbol{x_1}, \ldots, \boldsymbol{x_v}$ be v sequences with minimal periods $T_1, \ldots, T_v$, and f a Boolean function of v variables. Let*

$$\mathcal{T} = \{\sum_{i=1}^{m} c_i M_i, \;\; c_i \in \{0,1\}\}$$

where $M_i = q_i \mathrm{lcm}(T_{\ell_i+1}, \ldots, T_{\ell_{i+1}})$ with $q_i > 0$, $\ell_1 = 0$ et $\ell_{m+1} = n$. We consider that $\mathcal{T}$ does not contain multiples of T_j, for any $n < j \leq v$.

Then, we have that

$$\mathcal{E}(PC_{f,\mathcal{T}}) = \sum_{a \in \mathbf{F}_2^{n2^{m-1}}} \prod_{c=0}^{2^m-1} \mathcal{E}(f \oplus \varphi_{(\chi(c,a),0)}),$$

where $\varphi(u,0)$ denotes the linear function $(x_1, \ldots, x_v) \mapsto u \cdot (x_1, \ldots, x_n)$

Another conclusion that we have obtained is that the bias of a parity-check is entirely determined by the linear approximations of the involved combining function. In other words, the Walsh transform of the combining function f provides all the information needed for computing the bias of a parity-check build from it.

We have proposed a new lower bound of the bias that gives us information even when the parity check has been built from an unbiased approximation ; this is the first known result in this case.

Theorem 3. *Let $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_v$ be v sequences with minimal periods $T_1, \ldots, T_v$, and f a Boolean function of v variables. Let*

$$\mathcal{T} = \{\sum_{i=1}^{n} c_i M_i, c_i \in \{0,1\}\}$$

where $M_i = q_i \mathrm{lcm}(T_{\ell_i+1}, \ldots, T_{\ell_{i+1}})$ with $q_i > 0$, $\ell_1 = 0$ and $\ell_{n+1} = n$. We consider that $\mathcal{T}$ does not contain multiples of T_j, for any $n < j \leq v$. Then, for any Boolean function g of n variables of the form

$$g(x_1, \ldots, x_n) = \sum_{i=1}^{m} g_i(x_{\ell_i+1}, \ldots, x_{\ell_{i+1}}) \tag{1}$$

where each g_i is a Boolean function of $(\ell_{i+1} - \ell_i)$ variables, we have that

$$\mathcal{E}(PC_{f,\mathcal{T}}) \geq \left[\mathcal{E}(f \oplus g)\right]^{2^m}.$$

Corollary 1. *With the notations of the previous theorem, we have that*

$$\mathcal{E}(PC_{f,\mathcal{T}}) \geq \max_{\alpha \in \mathbf{F}_2^n} [\mathcal{E}(f \oplus \varphi_{\alpha,0})]^{2^m}.$$

It is important to note that this corollary gives us, as we said, a lower bound on the bias even if the functions f and $x \mapsto x_1 \oplus \ldots \oplus x_n$ are not correlated.

When the combining function is t-resilient, we have given a simple formula for computing the bias of any parity-check built with $t+1$ variables, and this bias depends exclusively on the bias of the associated linear approximation.

Theorem 4. *Let f be a t-resilient function. The bias of any parity check built from a linear approximation φ of f with $(t+1)$ variables and a bias of $\mathcal{E}(f \oplus \varphi) = \varepsilon$ is ε^{2^m} where 2^m is the number of terms appearing in the parity check.*

In the case of plateaued functions, we have obtained several results. When considering parity-check relations built from $t+2$ variables, we have obtained an upper bound on the bias, and we show that this bound can be acheived under certain conditions.

We have shown how to build the best parity check equations from $t+2$ variables (for plateaued functions), section 3.3. In the case of $t+k$ with $k > 2$ we have presented some conjectures, but it remains as futur work to proof them.

Part II

The second part is on hash functions. As we said before, we first present our hash function Shabal and then we describe the attacks against other hash functions that we have performed.

Shabal is an innovative and performant hash function with a new mode of operation submitted to the SHA-3 competition. It has been recently selected for the second round of the competition because all of these advantages. Its description can be found in Section 5.1 or in the specifications document [BCCM+09a].

A very thorough study of the permutation used in Shabal and a security analysis of its influence on the security of Shabal and of reduced versions can be found in Section 5.5. We show that the existence of distinguishers on the permutation does not threaten the security of Shabal because, due to the mode of operation, they are not applicable. We have found properties of the permutation, that have been exhausted, applicables with the mode of operation and their effect is to reduce the security margin for preimageattacks, which now corresponds to 9 words (288 bits) which is large enough to be confident in its security.

The second half of this part focuses on cryptanalysis of other hash functions proposed to the SHA-3 competition. A summary of these attacks can be found in Table 6.1. All of these attack have been performed during the first round of the competition, before knowing the algorithms selected for the second round. Then, most of them had probably an impact on this decision. Among the analysed hash functions, two have not been chosen for the first round, three have been chosen for the first round but not for the second and one has passed to the second round.

1. MCSSHA : We have presented second preimage attacks against all proposed versions of MCSSHA, a collision attack against MCSSHA-3 and a preimage attack in a particular case against MCSSHA-4 [ANP09]. We have proposed a design criteria for resisting these second preimage attacks, that is a bigger internal state. It is probably because of this that MCSSHA has not been selected for the second round of the competition.
2. CubeHash : We have performed the first known analysis of this hash function [ABM+09]. The submitted version, CubeHash8-1, has generic preimage attacks a little below the limit of the NIST requirements, as described in its own documentation. For some other versions, with particular values of a parameter b, we have described a technique that speeds up the attack, using the symmetry of the state. We have also performed some other analysis of it.
3. Maraca : We have presented a collision attack on Maraca that gives also a collision on the internal state [CNP09c] with a smaller complexity than the generic attacks. The main interest of this cryptanalysis is the use of differential properties of the involved permutation that introduce some weaknesses and that, at our knowledge, have never been exploited before. Also, these properties are, in some sense, in contradiction with the security criterion of the classical differential cryptanalysis.
4. Lane : We have performed a semi-free start collision on both (256 and 512-bit) complete versions of Lane [MNPN+09]. Our attacks are an improved application of the rebound attack and this allows us to use a differential path that introduces differences only in a part of the state. It exploits a weak diffusion of the permutation. Even if they do not threaten directly the security of the hash function, the main impact of these attacks is to invalidate the Merkle-Damgård proof.
5. ESSENCE : We have presented the first cryptanalysis against both versions of the complete hash function ESSENCE. Those are collision attacks that use a low-weight differential path and the degrees of freedom in the message to reduce complexity. The attacks presented in my thesis have been improved by us [NPRA+09], reducing more the complexity. For proving the correctness of the attack, an example of a collision for 29 out of the 32 rounds is given.

Introduction générale

La cryptographie symétrique est une des deux grandes branches de la cryptographie. Elle inclut essentiellement trois familles d'algorithmes : les chiffrements par blocs, les chiffrements à flot, et les fonctions de hachage. Pendant le parcours de cette thèse je me suis intéressée principalement aux chiffrements à flot et aux fonctions de hachage. Nous verrons les fonctionnalités et l'importance de ces primitives dans les chapitres correspondants. Mais un de leurs points communs est d'avoir fait récemment l'objet de deux compétitions internationales : eSTREAM initiée en 2004 par le réseau d'excellence européen ECRYPT pour les chiffrements à flot, et SHA-3 lancée en 2008 par l'organisme de standardisation américain NIST pour les fonctions de hachage. Ces deux compétitions ont sollicité les propositions d'algorithmes par un appel à soumissions public, puis ont étudié et analysé la sécurité des candidats proposés afin d'en sélectioner un (ou plusieurs dans le cas de eSTREAM) dont on puisse recommander l'utilisation. Ma thèse s'est donc déroulée dans le contexte de ces deux concours, puisqu'elle a commencé pendant la phase d'évaluation de eSTREAM (qui s'est terminée en 2008) et qu'elle se finit presque un an après la début du concours SHA-3. Cette actualité m'a donc conduite à travailler sur la cryptanalyse et la conception de ces deux types d'algorithmes. Cette thèse est donc naturellement divisée en deux parties.

La première traite des chiffrements à flot, où nous présentons des attaques sur les versions successives de l'algorithme de chiffrement à flot Achterbahn, candidat à eSTREAM, en particulier des attaques que nous avons développées sur les dernières versions du système et que nous avons notamment présentées à la conférence *FSE 2007* [NP07a] et aux colloques internationaux SASC 2007 et WEWoRC 2007 [NP07b, NP08]. Ensuite nous verrons un travail plus théorique qui a été motivé par les attaques précédentes, et qui va nous permettre de calculer les biais des relations utilisées de manière plus précise. Cela va donc nous aider à mieux concevoir les attaques, mais aussi nous donner des critères pour la conception des systèmes du même type que celui que nous avons analysé. Ces travaux ont eux été présentés à la conférence *IEEE International Symposium on Information Theory - ISIT 2009* [CNP09a].

La deuxième partie traite des fonctions de hachage. D'abord on introduit un algorithme que nous avons conçu et soumis à la compétition internationale SHA-3 dans le but de devenir le prochain standard de hachage. Sa candidature a été jusqu'à présent retenue pour passer les deux premiers tours de la compétition [BCCM+09a]. Ensuite, je présente les

attaques de plusieurs fonctions de hachage, soumises au concours SHA-3, que j'ai réalisées pendant ma thèse. Ces attaques, qui utilisent pour la plupart des techniques différentes, portent sur les fonctions MCSSHA-3 (et ses variantes), Ponic, Maraca, CubeHash, Essence et Lane. Elles ont toutes fait l'objet d'articles envoyés sur la liste de diffusion du NIST, et certaines ont déjà été présentées dans des colloques internationaux : *WEWoRC 2009* [ANP09], *the International conference on finite fields and their applications - Fq9* [CNP09c], *the Australasian Conference on Information Security and Privacy - ACISP 2009* [ABM+09] et *Asiacrypt 2009* [MNPN+09].

Première partie

Chiffrement à flot

Chapitre 1

Introduction au chiffrement à flot

1.1 Chiffrement à flot

Le chiffrement à flot, appelé également chiffrement par flux ou chiffrement à la volée, est une de deux grandes familles de chiffrements à clé secrète. Dans un chiffrement à flot synchrone additif, le texte chiffré est obtenu en combinant par ou exclusif bit-à-bit le message clair avec une suite binaire secrète, de même longueur que le message. Tous les chiffrements à flot que nous étudierons ici sont de ce type, même s'il existe d'autres catégories de systèmes, comme les chiffrements à flot auto-synchronisants. Cette suite binaire, appelée suite chiffrante, peut être

- soit une suite aléatoire entièrement secrète partagée par les deux utilisateurs : cette situation correspond à la technique du masque jetable ;
- soit une suite pseudo-aléatoire, c'est-à-dire produite à partir d'une clé secrète par un générateur pseudo-aléatoire.

Les principaux avantages du chiffrement à flot sont que chaque nouveau bit transmis peut être chiffré ou déchiffré indépendamment des autres, en particulier sans qu'il soit nécessaire d'attendre les bits suivants, et aussi que le processus de déchiffrement ne propage pas les erreurs de transmission. Un autre avantage est que ces algorithmes ne nécessitent pas de padding, c'est-à-dire d'ajout de bits au message clair pour que sa taille soit un multiple de la taille de bloc. Ceci est très utile dans les applications où l'on transmet des paquets courts et où la bande passante est limitée. Ces systèmes sont donc très utilisés dans les transmissions bruitées et aussi dans des contextes où il est primordial de pouvoir chiffrer et déchiffrer très rapidement ou au moyen de ressources très limitées.

1.2 Générateurs pseudo-aléatoires

L'algorithme de chiffrement à flot le plus simple est le célèbre chiffre du masque jetable, qui offre une sécurité parfaite mais qui nécessite l'échange au préalable d'une clé secrète aussi longue que le message à chiffrer. Il ne peut donc pas être utilisé en pratique sauf dans des cas extrêmement particuliers. Les algorithmes de chiffrement à flot employés en

pratique sont donc des versions affaiblies du chiffre du masque jetable dans lesquelles la suite aléatoire secrète est remplacée par une suite produite par un générateur pseudo-aléatoire.

Un générateur pseudo-aléatoire est un procédé qui, à partir d'une initialisation de taille fixée appelée *graine* ou *germe*, engendre de manière déterministe une suite de très grande longueur que l'on ne peut pas distinguer d'une suite aléatoire quand on ne connaît pas la graine en un coût inférieur à celui de la recherche exhaustive de la graine.

Dans un algorithme à flot additif, le chiffrement consiste à additionner (par ou exclusif bit à bit) le texte clair à la sortie d'un générateur pseudo-aléatoire initialisé par la clé secrète. Le destinataire du chiffré effectue la même opération pour retrouver le message clair. On voit ici que la suite pseudo-aléatoire doit être reproductible par toute personne connaissant la clé secrète.

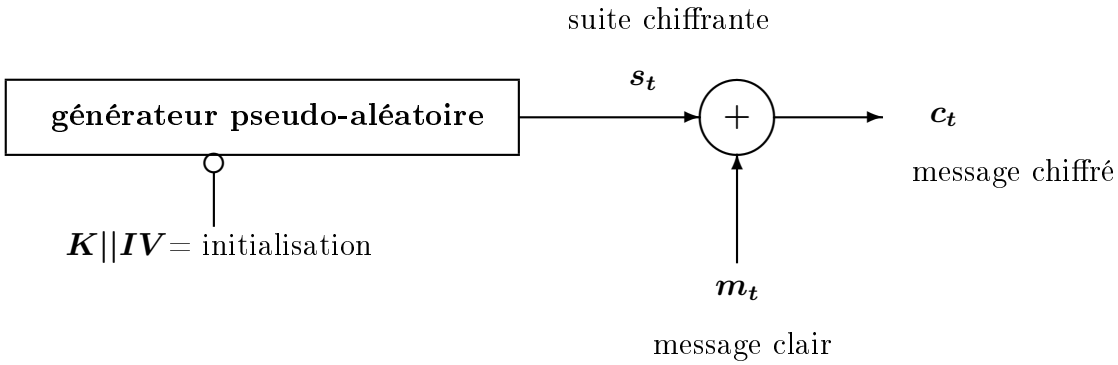

FIG. 1.1 – Chiffrement à flot synchrone additif

Dans la plupart des applications on utilise des générateurs pseudo-aléatoires dont l'initialisation est calculée à partir de deux données : la clé secrète et une quantité publique usuellement appelée valeur initiale. Généralement on a besoin de réinitialiser le générateur sans changer la clé. Par exemple, pour les communications téléphoniques, on échange des trames d'environ 1000 bits. Si on chiffrait toute une conversation à partir du même état initial, il serait très compliqué de gérer les pertes de paquets. En effet, les protocoles de communication transmettent généralement les messages sous forme de paquets de taille fixée ; la valeur initiale correspond donc souvent au numéro de paquet, la clé secrète restant inchangée au cours d'une même communication.

1.3 Les principales attaques

Pour évaluer la sécurité d'un algorithme de chiffrement à flot, on se place usuellement dans le contexte d'une attaque à clair connu, ce qui signifie que l'attaquant dispose d'une certaine quantité de suite chiffrante. On dit qu'un chiffrement à flot est résistant à une certaine attaque si celle-ci a une complexité plus grande que $2^{|K|}$ en temps ou en mémoire,

où $|K|$ est la taille de la clé. La complexité en données peut être limitée par le concepteur de l'algorithme à des valeurs plus petites que $2^{|K|}$.

Les attaques, en fonction de leur impact, peuvent être classées suivant ce qu'elles cherchent à faire : certaines retrouvent la clé, d'autres retrouvent l'état interne, d'autres peuvent prédire le bit suivant de suite chiffrante, et d'autres peuvent distinguer la suite chiffrante d'une suite aléatoire. Ici elles sont ordonnées de la plus à la moins puissante, les attaques par distingueurs étant donc les moins puissantes.

Ensuite, en fonction de la technique appliquée, nous pouvons aussi faire un autre classement d'attaques, les deux principales familles étant les attaques par corrélation et les attaques algébriques. Les attaques par corrélation ont été introduites par Siegenthaler [Sie85] pour la cryptanalyse des générateurs par combinaison des LFSRs, et elles ont ensuite été améliorées par Meier et Staffelbach [MS88, MS89] (sous le nom d'attaques par corrélation rapides), qui ont montré que ces attaques se ramenaient à un problème de correction d'erreurs. Plus tard, diférents algorithmes ont été proposés pour résoudre ce problème de décodage [JJ99b, JJ99a, CT00, CJS00, JJ00, CJM02]. Les attaques par corrélation sont du type diviser pour mieux régner, et elles sont donc applicables à tout système avec un état interne formé par plusieurs parties mises à jour indépendamment les unes des autres. L'attaque repose alors sur l'existence d'éventuelles corrélations entre la sortie de la fonction de sortie et un sous-ensemble de ses entrées, formée par la partie que nous avons séparée de l'état interne.

Les attaques algébriques ont été proposées en 2003 par Courtois et Meier [CM03a, Cou03, FA03]. L'idée de base est d'exprimer le chiffrement comme un système d'équations algébriques multivariées et ensuite de résoudre ce système pour retrouver les bits de clé.

On va maintenant s'intéresser à un type de générateur pseudo-aléatoire particulier : celui à base de registres à décalage à rétroaction (FSR), car ce sont des générateurs très utilisés dans les chiffrements à flot.

1.4 FSRs

Un registre à décalage à rétroaction, usuellement désigné par FSR pour « Feedback shift register », est un dispositif permettant d'engendrer une suite infinie qui satisfait une relation de récurrence. Plus précisément, un FSR de longueur L sur $\mathbf{F}_2^\ell$ est composé d'un registre à L cellules contenant chacune ℓ bits. Les $L \times \ell$ bits contenus dans le registre forment l'état interne du FSR.

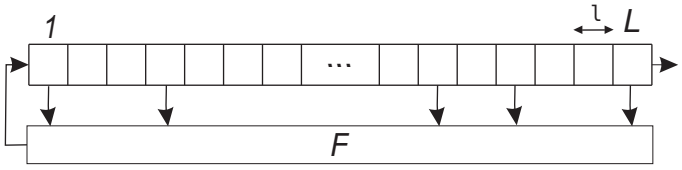

FIG. 1.2 – Registre à décalage avec rétroaction F.

Ces L cellules sont initialisées par L mots de ℓ bits $s_0, \ldots, s_{L-1}$. Ce registre à décalage est contrôlé par une horloge externe. Au cours de chaque unité de temps, chaque mot de ℓ bits est décalé d'une cellule vers la droite. Le contenu de la cellule la plus à droite, s_t, sort du registre à l'instant t, alors que la cellule la plus à gauche reçoit le mot de rétroaction, s_{t+L}. La valeur de ce dernier est obtenue par une combinaison des valeurs des autres cellules. Si la fonction de rétroaction, F, est linéaire, on parlera de LFSR, pour « Linear feedback shift register ».

1.5 Le projet eSTREAM

La plupart des chiffrements à flot connus avant avril 2005 avaient été attaqués, comme par exemple RC4, A5/1 et E0. Le projet eSTREAM [ECRa] est un projet lancé par le réseau européen ECRYPT sur la conception de nouveaux chiffrements à flot dédiés. Ceux-ci sont les seules constructions qui permettent d'atteindre un débit de chiffrement supérieur à celui d'un algorithme par blocs (de l'ordre de quelques cycles du processeur par octet en logiciel), ou qui puissent être implémentés par un circuit électronique de petite taille et à faible consommation. Trente-quatre nouveaux algorithmes ont été proposés en avril 2005 et ont été évalués pendant 3 ans.
Il y a deux types d'algorithmes proposés :

- orientés « software » : il s'agit de systèmes de chiffrement à flot pour les applications logicielles qui ont besoin d'une grande vitesse de chiffrement. La taille de clé pour ces algorithmes devait être 128 bits, et celle de l'IV 64 ou 128 bits.
- orientés « hardware » : il s'agit de systèmes de chiffrement à flot pour les applications matérielles qui ont des ressources très limitées en mémoire, en portes logiques ou en consommation de puissance. La taille de clé pour ces algorithmes devait être 80 bits, et celle de l'IV 32 ou 64 bits.

La fin de la première phase d'évaluation a été en février 2006. Quelques algorithmes ont été éliminés. La plupart des algorithmes à vocation matérielle, c'est-à-dire pour lesquels on souhaite une mise en œuvre par un circuit électronique de petite taille et de faible consommation, ont été cryptanalysés pendant la phase 1 (et, pour la plupart, modifiés ensuite) ce qui montre bien la nécessité qu'on avait de poursuivre des recherches pour aboutir à la conception d'un système qui soit à la fois sûr et efficace dans ce contexte.
Pendant la deuxième phase qui a eu lieu d'août 2006 à avril 2007, les modifications des algorithmes après les cryptanalyses n'étaient plus permises. Cette deuxième phase a abouti en avril 2007 à la sélection de 16 finalistes. Leurs spécifications ont d'ailleurs fait l'objet d'un livre [RB08]. En avril 2008, le premier portfolio avec les chiffrements recommandés a été rendu public. Il y avait 4 algorithmes dans la catégorie « software » et 4 dans la catégorie « hardware ». En septembre 2008, après la cryptanalyse d'un des finalistes « hardware », le dernier rapport en date sur les travaux du projet eSTREAM a été publié, avec un finaliste en moins dans cette catégorie[1].

[1] http ://www.ecrypt.eu.org/stream/

Chapitre 2

Cryptanalyse de Achterbahn

Dans ce chapitre je vais présenter la cryptanalyse d'un des candidats de eSTREAM, Achterbahn. D'abord, je vais présenter le modèle général du générateur par combinaison de FSRs sur lequel est construit Achterbahn. Puis je détaillerai trois attaques sur ce type de générateur proposées par Johansson, Meier et Muller pour cryptanalyser la première version d'Achterbahn. Ensuite je vais décrire plusieurs améliorations d'une de ces trois attaques, les dernières étant celles que nous avons introduites pour cryptanalyser les versions récentes de Achterbahn.

2.1 Le générateur utilisé

2.1.1 Modèle général et notations

Le modèle général utilisé dans Achterbahn est un générateur pseudo-aléatoire à base de FSRs binaires, c'est-à-dire, de FSRs où chaque cellule ne contient qu'un bit. Un générateur par combinaison de FSRs binaires est composé de n FSRs qui fonctionnent en parallèle, et dont les sorties sont combinées au moyen d'une fonction booléenne à n variables. C'est la sortie de cette fonction f à l'instant t qui fournit le bit correspondant de suite chiffrante. Dans toute la suite nous allons noter R_i le registre i, L_i sa longueur. La suite produite par R_i est notée $\boldsymbol{x_i} = (x_i(t))_{t\geq 0}$. Il s'agit d'une suite périodique de période $T_i \leq 2^{L_i}$. La suite chiffrante est notée

$$\boldsymbol{S} = (S(t))_{t\geq 0}$$

Le générateur par combinaison de LFSRs est un modèle très ancien et étudié, mais qui est vulnérable à de nombreuses attaques, comme les attaques par corrélation [Sie85, MS88, MS89, JJ99a, CT00, JJ00, CJS00, CJM02, Jou09], algébriques [CM03b, CM03a, Cou03], par distingueur [Gol96, JJ99b, CHJ02, MH04, EJ05]...

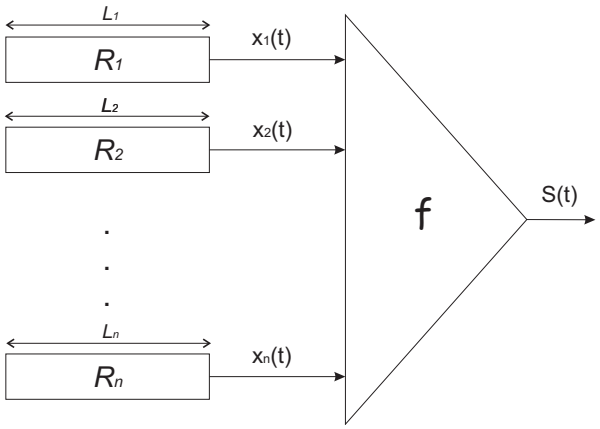

FIG. 2.1 – Générateur pseudo-aléatoire par combinaison de FSRs

2.1.2 Achterbahn version 1

Un des algorithmes proposés dans le cadre de eSTREAM [ECRa] et qui est passé à la phase 2 est Achterbahn, créé par B. Gammel, R. Göttfert et O. Kniffler [GGK05a]. Achterbahn est un système de chiffrement à flot construit suivant le modèle des générateurs pseudo-aléatoires par combinaison de plusieurs registres à décalage, mis à jour indépendamment les uns des autres. Il est orienté « hardware ».

Les registres à décalage utilisés sont des NLFSRs, c'est-à-dire des registres à décalage à rétroaction non-linéaire. Au total, il y a huit registres, de longueurs comprises entre 22 et 31 bits. Ces registres sont primitifs, au sens où la période de la suite engendrée par le registre R_i de longueur L_i est égale à : $T_i = 2^{L_i} - 1$, car l'état tout-à-zéro est un point fixe.

Les sorties de ces huit registres sont combinées par une fonction booléenne, f, à huit variables, qui a une non-linéarité 64 et qui est 4-résiliente :

$$f(x_1, \ldots, x_8) = x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5x_7 \oplus x_6x_7 \oplus x_6x_8 \oplus x_5x_6x_7 \oplus x_6x_7x_8$$

La sortie de cette fonction f est la suite chiffrante.

L'initialisation des registres est faite grâce à une clé K de 80 bits et une valeur publique, IV, de la même longueur. Dans un premier temps, on introduit les bits de $K||IV$ en parallèle dans les registres. Puis, on fait des mises à jour des registres en rajoutant les bits qu'il nous reste encore de $K||IV$. Ensuite, on met le dernier bit de chaque registre à 1, pour assurer que l'état entièrement nul ne puisse pas se produire. Finalement on fait encore quelques mises à jour des registres comme « tours de chauffe ».

Jusqu'ici, nous avons décrit la version réduite d'Achterbahn. La version complète est différente dans le sens où les entrées de la fonction booléenne ne sont pas directement les bits de sortie des registres, mais une combinaison linéaire de ces bits. On a donc des filtres linéaires à la sortie de chaque registre pour déterminer chaque entrée de la fonction

booléenne. Ces filtres sont initialisés de la même façon que les registres. La version réduite n'est qu'un cas particulier du cas complet.

Les principales faiblesses de cet algorithme de chiffrement viennent du fait que la fonction booléenne n'est pas très dense, c'est-à-dire que sa forme algébrique normale comporte peu de monômes, et que les registres sont autonomes et trop courts. Il a été cryptanalysé par T. Johansson, W. Meier et F. Muller [JMM06] avec une complexité de 2^{55} pour la version réduite et de 2^{61} pour la complète. Nous allons maintenant décrire ces attaques sur le modèle général des générateurs par combinaison de FSRs.

2.2 Attaques de base sur les combinaisons de FSRs

2.2.1 Relations de parité

Dans toute la suite on utilisera les notations employées à la figure 2.1.

La plupart des attaques que nous allons décrire exploitent l'existence de *relations de parité* pour la suite chiffrante (parity-checks en anglais). Il s'agit de relations linéaires qui lient les bits de $\boldsymbol{S}$ à différents instants.

Définition 2.1. Soient $\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}$ n suites et soit f une fonction booléenne à n variables. Soient $M_1, \ldots, M_s$ s entiers positifs. L'ensemble $\mathcal{T}$ des *combinaisons à coefficients* $\{0,1\}$ *de* $M_1, \ldots, M_s$ est noté

$$\mathcal{T} = \langle M_1, \ldots, M_s \rangle = \left\{ \sum_{i=1}^{s} c_i M_i, c_i \in \{0,1\} \right\}$$

La relation de parité construite sur f par rapport à $\mathcal{T}$ est la suite $\boldsymbol{PC_{f,\mathcal{T}}}$ définie par

$$PC_{f,\mathcal{T}}(t) = \bigoplus_{\tau \in \mathcal{T}} f(x_1(t+\tau), \ldots, x_n(t+\tau)), \forall t \geq 0.$$

Le principe de ces attaques va donc être de trouver des relations de parité biaisées pour le générateur. Rappelons que le produit des périodes de 2 suites est une période de leur somme et de leur produit.

Pour obtenir des relations de parité biaisées on utilisera le résultat suivant.

Proposition 2.2. *Soient* $\boldsymbol{s_1}, \ldots, \boldsymbol{s_n}$ *n suites de période* $T_1, \ldots, T_n$. *Soit* $\boldsymbol{S}$ *la suite définie par :*

$$\forall t \geq 0, \quad S(t) = \bigoplus_{i=1}^{n} s_i(t).$$

Alors, pour tout $t \geq 0$,

$$\bigoplus_{\tau \in \langle T_1, \ldots, T_n \rangle} S(t+\tau) = 0$$

où $\langle T_1, \ldots, T_n \rangle$ *est l'ensemble des* 2^n *combinaisons à coefficients dans* $\{0,1\}$ *des entiers* $T_1, \ldots, T_n$.

Preuve : On fait la démonstration par récurrence. Si $n = 1$,

$$\bigoplus_{\tau \in \langle T_1 \rangle} S(t+\tau) = \bigoplus_{\tau \in \{0, T_1\}} s_1(t+\tau) = s_1(t) \oplus s_1(t+T_1) = 0,$$

car T_1 est une période de $\boldsymbol{s_1}$.

On suppose que la proposition est vraie pour $(n-1)$ suites. On va voir que dans ce cas-là elle est aussi vraie pour n.

On sait que $S'(t) = \bigoplus_{i=1}^{n-1} s_i(t)$ vérifie

$$\bigoplus_{\tau' \in \langle T_1, \ldots, T_{n-1} \rangle} S'(t+\tau') = 0, \forall t \geq 0.$$

Soit $S(t) = \bigoplus_{i=1}^{n} s_i(t) = S'(t) \oplus s_n(t)$, alors

$$\begin{aligned}
\bigoplus_{\tau \in \langle T_1, \ldots, T_n \rangle} S(t+\tau) &= \bigoplus_{\tau' \in \langle T_1, \ldots, T_{n-1} \rangle} \bigoplus_{i \in \{0, T_n\}} S(t+\tau'+i) \\
&= \bigoplus_{\tau' \in \langle T_1, \ldots, T_{n-1} \rangle} [S'(t+\tau') \oplus s_n(t+\tau') \oplus S'(t+\tau'+T_n) \\
&\quad \oplus \; s_n(t+\tau'+T_n)] \\
&= \bigoplus_{\tau' \in \langle T_1, \ldots, T_{n-1} \rangle} S'(t+\tau') \oplus \bigoplus_{\tau' \in \langle T_1, \ldots, T_{n-1} \rangle} S'(t+\tau'+T_n) \\
&\quad \oplus \bigoplus_{\tau' \in \langle T_1, \ldots, T_{n-1} \rangle} s_n(t+\tau') \oplus \bigoplus_{\tau' \in \langle T_1, \ldots, T_{n-1} \rangle} s_n(t+\tau'+T_n),
\end{aligned}$$

où les deux premiers termes sont nuls d'après de l'hypothèse de récurrence ; il nous reste donc :

$$\bigoplus_{\tau \in \langle T_1, \ldots, T_n \rangle} S(t+\tau) = \bigoplus_{\tau' \in \langle T_1, \ldots, T_{n-1} \rangle} s_n(t+\tau') \oplus \bigoplus_{\tau' \in \langle T_1, \ldots, T_{n-1} \rangle} s_n(t+\tau'+T_n),$$

où, pour tout terme de la première somme, il y en a un dans la deuxième qui en est séparé de T_n, donc le total va être égal à zéro, car T_n est une période de $\boldsymbol{s_n}$. Finalement on trouve :

$$\bigoplus_{\tau \in \langle T_1, \ldots, T_n \rangle} S(t+\tau) = 0.$$

On démontre de cette manière que la propriété se vérifie pour tout n.

□

Exemple 2.3. Considérons la suite $\boldsymbol{S}$ définie par

$$S(t) = x_1(t) \oplus x_2(t)x_3(t).$$

Alors, une relation de parité pour $\boldsymbol{S}$ est :

$$\forall t \geq 0,\ S(t) \oplus S(t+T_1) \oplus S(t+T_2T_3) \oplus S(t+T_1+T_2T_3) = 0.$$

Il suffit en effet d'appliquer le résultat précédent avec $s_1(t) = x_1(t)$ et $s_2(t) = x_2(t)x_3(t)$. En effet, la suite $\boldsymbol{s_2}$ est périodique de période T_2T_3 (qui n'est pas toujours la période minimale, mais ceci n'est pas important dans notre cas).

2.2.2 Attaque exploitant une restriction linéaire de la fonction de combinaison

Principe de l'attaque. Soit $I = \{i_1, \ldots, i_v\} \subset \{1, \ldots, n\}$ un ensemble de v entrées de f et $a_1, \ldots, a_v \in \mathbf{F}_2^v$ tels que la fonction obtenue en fixant les entrées d'indices $i_1, \ldots, i_v$ de f à $a_1, \ldots, a_v$ est une fonction linéaire ℓ avec α termes, d'indices notés $k_1, \ldots, k_\alpha$. Pour plus de clarté, on supposera dans toute la suite, sans perdre de généralité, que $\{i_1, \ldots, i_v\} = \{1, \ldots, v\}$, c'est-à-dire que ce sont les v premières variables de f qui sont fixées. Notre hypothèse est donc : $\forall (x_{v+1}, \ldots, x_n) \in \mathbf{F}_2^{n-v}$,

$$\begin{aligned} f(a_1, \ldots, a_v, x_{v+1}, \ldots, x_n) &= \ell(x_{v+1}, \ldots, x_n) \\ &= x_{k_1} \oplus \ldots \oplus x_{k_\alpha}. \end{aligned}$$

On construit une relation de parité avec 2^α termes qui correspond à la somme des $[x_{k_1}(t+\tau) \oplus \ldots \oplus x_{k_\alpha}(t+\tau)]$ où τ décrit toutes les combinaisons des périodes des registres intervenant dans ℓ. Cette relation est égale à 0 à chaque instant $t \geq 0$. On en déduit que, si la sortie des v premiers registres vaut $(a_1, \ldots, a_v)$, on a

$$\bigoplus_{\tau \in \langle T_{k_1}, \ldots, T_{k_\alpha} \rangle} S(t+\tau) = 0.$$

On définit un état favorable comme un état du générateur où les entrées d'indices $i_1, \ldots, i_v$ de f sont égales à $a_1, \ldots, a_v$ à un instant t_0 et aux $(2^\alpha - 1)$ instants $t_0 + \tau, \tau \in \langle T_{k_1}, \ldots, T_{k_\alpha} \rangle$ intervenant dans la relation de parité. À l'aide d'un tableau, on va enregistrer le nombre de mises à jour du générateur qu'il nous faut pour passer d'un état favorable au suivant.

Un état favorable doit donc vérifier 2^α équations pour chacun des v registres, ce qui implique qu'il existe

$$2^{\sum_{i=1}^{v} L_i - v2^\alpha} \text{ états favorables.}$$

De même, l'écart moyen entre deux états favorables du générateur est de

$$2^{v2^\alpha}.$$

L'attaque consiste alors, pour chaque instant t_0 pour lequel la relation de parité est satisfaite, à supposer que cet instant correspond à un des $2^{\sum_{i=1}^{v} L_i - v2^\alpha}$ états favorables et à vérifier si la relation est encore satisfaite pour l'état favorable suivant. Ce processus est itéré

un nombre de fois Δ suffisamment grand pour que la probabilité qu'un état soit détecté comme un état favorable alors qu'il ne l'est pas soit très faible. Par exemple, Johansson, Meier et Muller suggèrent de l'itérer $\Delta = 256$ fois [JMM06]. Cet algorithme fournit une attaque par distingueur, mais il permet aussi de retrouver l'état initial complet des v premiers registres à l'instant t_0 grâce à la technique décrite par l'algorithme 1

Algorithme 1 Attaque utilisant une restriction linéaire de la fonction de combinaison

/*Précalcul*/
Construire un tableau contenant tous les états favorables $(r_1, \ldots, r_v)$ et le temps pour aller à l'état favorable suivant.
/*Attaque*/
pour $t_0 = 0$ à 2^{v2^α} **faire**
  si $(\bigoplus_{\tau \in \langle T_{k_1}, \ldots, T_{k_\alpha} \rangle} S(t_0 + \tau) = 0)$ **alors**
    pour chaque état favorable $(r_1, \ldots, r_v)_0$ **faire**
      $i = 0$
      $t_1 = 0$
      tantque $(\bigoplus_{\tau \in \langle T_{k_1}, \ldots, T_{k_\alpha} \rangle} S(t_0 + t_1 + \tau) = 0)$ et $i \leq \Delta$ **faire**
        $t_1 \leftarrow$ temps pour aller à l'état favorable suivant
        état favorable $\leftarrow$ état favorable suivant
        $i \leftarrow (i + 1)$
        si $i = \Delta + 1$ **alors**
          état des v premiers registres à l'instant $t_0 \leftarrow (r_1, \ldots, r_v)_0$
        finsi
      fin tantque
    fin pour
  finsi
fin pour

Complexité.

– On a besoin d'un nombre de bits supérieur ou égal à la somme des périodes des suites qui interviennent dans la restriction linéaire ℓ :

$$\sum_{j=1}^{\alpha} T_{k_j} \simeq \sum_{j=1}^{\alpha} 2^{L_{k_j}} \text{ bits.}$$

On a en plus besoin de $\Delta 2^{v2^\alpha}$ bits, car 2^{v2^α} est l'écart moyen entre deux états favorables et Δ est le nombre d'itérations qu'on effectue pour s'assurer qu'on est bien dans un état favorable déterminé. On peut prendre $\Delta = 2^8$ comme dans [JMM06]. Donc, finalement on a besoin de

$$2^{v2^\alpha + 8} + \sum_{j=1}^{\alpha} 2^{L_{k_j}} \text{ bits de suite chiffrante.}$$

– La complexité en temps de calcul est de

$$2^{1+\sum_{j=1}^{v} L_j} \text{opérations},$$

car, pour chaque t_0 et chaque état favorable de départ on doit tester si la relation de parité est satisfaite en moyenne deux fois. Or, on a $2^{\sum_{j=1}^{v} L_j - v2^{\alpha}}$ états favorables et $2^{v2^{\alpha}}$ bits entre deux états favorables.

Regrouper les registres. Si on regroupe les registres quand on définit les suites $\boldsymbol{s_i}$ pour construire la relation de parité comme à la proposition 2.2, on ne change que la complexité en données. Il faut donc un compromis entre la période maximale intervenant dans la relation de parité et son nombre de termes, car les deux interviennent dans le nombre de bits de la suite chiffrante dont on a besoin.

Retrouver la clé. Une fois qu'on connaît les états des registres fixés, avec une attaque dans le milieu on peut retrouver la clé de 80 bits avec une complexité de 2^{41} opérations. On commence par inverser les mises à jour de ces registres jusqu'à arriver à $t = 0$. Une fois qu'on arrive là, on peut aussi inverser l'introduction des bits de la valeur publique IV, car ceux-ci sont connus. Quand on arrive au point d'inverser l'insertion de la clé, on peut, d'une part, faire une recherche exhaustive sur les 40 premiers bits en enregistrant en mémoire les états qu'on obtiendrait pour ces registres là si on avait introduit ces bits. D'autre part, on peut faire une recherche exhaustive sur les 40 autres bits en faisant le chemin inverse de l'initialisation, à partir des états qu'on avait obtenus avant l'introduction de l'IV. Pour chaque état on cherche en mémoire une coïncidence entre les deux tables. À la fin on va donc trouver $2^{80-\sum_{j=1}^{v} L_j}$ clés possibles.

Exemple d'attaque sur Achterbahn version 1. Johansson, Meier et Muller ont utilisé cette attaque sur Achterbahn version 1, en fixant à 0 les registres 5 et 6, de longueur 27 et 28. Quand ces variables sont fixées, la restriction de f est la fonction linéaire :

$$\ell = y_1 \oplus y_2 \oplus y_3 \oplus y_4.$$

La complexité en temps de l'attaque est de 2^{56} et en données de 2^{40}. Dans la version complète, en prenant en compte les filtres linéaires, la complexité en temps devient 2^{73}.

2.2.3 Attaque exploitant l'existence de certaines structures linéaires pour la fonction de combinaison

Définition 2.4. Soit f une fonction booléenne à n variables. Pour tout $a \in \mathbf{F}_2^n$, la *dérivée de f par rapport à a* est la fonction booléenne à n variables $D_a f$, définie par

$$D_a f(x) = f(x + a) \oplus f(x), \forall x \in \mathbf{F}_2^n.$$

De plus, a est une *structure linéaire pour f* si la fonction $D_a f$ est constante.

Dans cette attaque on utilise le fait que f est linéaire par rapport aux variables $x_{i_1}, \ldots, x_{i_v}$, *i.e.*,

$$\forall j, 1 \leq j \leq v, D_{e_{i_j}} f(x) = 1, \forall x \in \mathbf{F}_2^n$$

où e_{i_j} est le mot qui vaut 0 partout sauf en i_j. Ceci veut dire que, quand on change le bit i_j de x on ajoute toujours 1 à $f(x)$. C'est équivalent au fait que x_{i_j} intervient linéairement.

Principe de l'attaque On écrit la fonction de combinaison f sous la forme

$$f(x_1, \ldots, x_n) = \ell(x_{i_1}, \ldots, x_{i_v}) \oplus g(x_{j_1}, \ldots, y_{j_p}),$$

où $\ell = \bigoplus_{j=1}^{v} x_{i_j}$ est la partie linéaire qui ne contient que les v variables qui interviennent linéairement et g est la partie non-linéaire, dans laquelle interviennent p variables différentes de $x_{i_1}, \ldots, x_{i_v}$.

On construit une relation de parité avec 2^v termes en sommant les bits de la suite chiffrante aux instants $t+\tau, \tau \in \langle T_{i_1}, \ldots, T_{i_v} \rangle$. Comme les $i_1, \ldots, i_v$ sont les indices des variables intervenant dans la partie linéaire, seuls les registres d'indices $j_1, \ldots, j_p$ interviennent pour déterminer

$$\bigoplus_{\tau \in \langle T_{i_1}, \ldots, T_{i_v} \rangle} S(t+\tau).$$

Donc si on fait une recherche exhaustive sur ces registres, et qu'on calcule la relation de parité sur la suite $\sigma(t) = g(x_{j_1}(t), \ldots, x_{j_p}(t))$, on doit retrouver les mêmes valeurs que pour la relation de parité appliquée à la suite chiffrante, car les termes linéaires n'interviennent plus.

Mais si on prend en compte le paradoxe des anniversaires, on n'a pas besoin de faire une recherche exhaustive car on peut effectuer une attaque par compromis temps-mémoire. Il va y avoir une collision d'états des registres qui interviennent dans g si on prend $2^{\sum_{i=1}^{p} L_{j_i}/2}$ états aléatoires des registres qui apparaissent dans g et le même nombre de bits de la suite chiffrante. On garde en mémoire les résultats des relations de parité de chaque état aléatoire choisi et les Δ suivants (Δ est le nombre de fois qu'il faut évaluer les relations de parité pour s'assurer que la probabilité de fausse alarme est très petite). Ensuite, au fur et à mesure qu'on calcule les relations de parité sur la chaîne de bits de la suite chiffrante, on va trouver une collision entre Δ évaluations consécutives et les valeurs enregistrées en mémoire. L'attaque permet alors de retrouver l'état initial des registres d'indices j_1 à j_v.

Complexité

– La complexité en données va être

$$2^{\sum_{i=1}^{p} \frac{L_{j_i}}{2}},$$

correspondant au nombre de bits de la suite chiffrante dont on a besoin pour trouver une collision.

– La complexité calculatoire sera de

$$\Delta 2^{\sum_{i=1}^{p} \frac{L_{j_i}}{2}} \text{ opérations.}$$

On peut prendre $\Delta = 2^7$.

Exemple sur Achterbahn version 1. Johansson, Meier et Muller utilisent également cette attaque pour Achterbahn version 1. Les registres intervenant dans la partie non-linéaire sont ceux d'indices 5, 6, 7 et 8. La complexité en temps est donc de $2^{57.5}$ et en données de $2^{57.5}$ pour la version réduite. Pour la version complète, l'attaque est plus coûteuse que la recherche exhaustive.

2.2.4 Attaque exploitant une approximation linéaire de la fonction de combinaison

Principe de l'attaque. Il s'agit d'une attaque par distingueur. Elle exploite l'existence d'approximations linéaires de f ayant un petit nombre de termes. Une telle approximation de f va en effet fournir une relation de parité biaisée pour la suite chiffrante. Les différentes valeurs prises par cette relation vont donc permettre de distinguer la suite chiffrante d'une suite aléatoire. Le biais est ici défini de la façon suivante.

Définition 2.5. Soit f une fonction booléenne à n variables. Le biais de f, noté $\mathcal{E}(f)$, est défini par :

$$\mathcal{E}(f) = 2\Pr[f(x) = 0] - 1 = \frac{1}{2^n}\left[\sharp\{x \in \mathbf{F}_2^n, f(x) = 0\} - \sharp\{x \in \mathbf{F}_2^n, f(x) = 1\}\right].$$

La fonction est équilibrée si $\mathcal{E}(f) = 0$. Ceci peut aussi être écrit :

$$\Pr[f(x) = 0] = \frac{1}{2}(1 + \mathcal{E}(f)).$$

Si f et g sont 2 fonctions booléennes, le biais entre f et g est le biais $\mathcal{E}(f \oplus g)$. Ce biais est aussi appelé biais de l'approximation de f par g.

Pour estimer le nombre d'évaluations de la relation de parité nécessaires pour obtenir un distingueur en fonction du biais de la relation de parité, on utilise le résultat suivant, dû à Baignères, Junod et Vaudenay [BJV04]. Étant donnée une source qui génère une suite de N variables aléatoires à valeurs dans $\mathbf{F}_2$, dans notre cas, indépendantes et identiquement distribuées suivant une distribution D, un distingueur à N échantillons est un algorithme qui prend en entrée une suite de N réalisations de ces variables aléatoires, et produit la valeur 0 si D correspond à D_0 ou la valeur 1 si D correspond à D_1, où D_0 et D_1 sont deux distributions connues. Dans notre cas, D_0 sera la distribution uniforme. L'attaque repose sur le résultat suivant, utilisé également pour la cryptanalyse linéaire.

Lemme 2.6. *[BJV04] Considérons une variable aléatoire X à valeurs dans $\mathbf{F}_2$ de distribution D proche de la distribution uniforme, c'est-à-dire telle que pour chaque élément x de son support on a*

$$\Pr_D[X = x] = \frac{1}{2}(1 + \varepsilon) avec\ |\varepsilon| \lll 1.$$

Alors pour un nombre d'échantillons égal à

$$N = \frac{d}{\varepsilon^2}$$

où d est un réel quelconque, la probabilité d'erreur du distingueur qui distingue X d'une variable aléatoire uniformément distribuée est de l'ordre de $\Phi(-\sqrt{d}/2)$ où Φ est la fonction de distribution de la loi normale :

$$\Phi(x) = \frac{1}{2\pi}\int_{-\infty}^{x} \exp(-\frac{t^2}{2})dt.$$

On va prendre, pour le résultat précédent, $d = 1$, ce qui correspond à une probabilité d'erreur de l'ordre de 0.3. Nous allons également utiliser dans la suite le lemme suivant, connu en cryptographie sous le nom de « piling-up lemma » qui calcule le biais de la somme de 2 variables aléatoires indépendantes.

Lemme 2.7. *Soient X_1 and X_2 deux variables aléatoires indépendantes à valeurs dans $\mathbf{F}_2$, la première avec un biais ε_1 et la deuxième ε_2. Alors on a que*

$$\Pr[X_1 \oplus X_2 = 0] = \frac{1}{2}(1 + \varepsilon_1\varepsilon_2),$$

donc le biais de $X_1 \oplus X_2$ sera $\varepsilon_1\varepsilon_2$.

L'attaque se déroule de la façon suivante :

– On cherche

$$\ell(y_1, \ldots, y_n) = \bigoplus_{j=1}^{m} y_{i_j}$$

une approximation linéaire de f avec m termes. Soit $\varepsilon = \mathcal{E}(f \oplus \ell)$ son biais, c'est-à-dire

$$\Pr[f(y_1, \ldots, y_n) = \ell(y_{i_1}, \ldots, y_{i_m})] = \frac{1}{2}(1 + \varepsilon).$$

– On construit une relation de parité à partir de ℓ en remarquant que, pour

$$\ell(t) = \bigoplus_{j=1}^{m} x_{i_j}(t),$$

et $\mathcal{T} = \langle T_{i_1}, \ldots, T_{i_n}\rangle$, on a

$$PC_{\ell,\mathcal{T}}(t) = \bigoplus_{\tau \in \langle T_{i_1}, \ldots, T_{i_m}\rangle} \ell(t + \tau) = 0.$$

Comme $\Pr[S(t) = \ell(t)] = \frac{1}{2}(1+\varepsilon)$, on a, d'après le piling-up lemma,

$$\Pr\left[\bigoplus_{\tau\in\langle T_{i_1},\ldots,T_{i_m}\rangle} S(t+\tau) = 0\right] \geq \frac{1}{2}(1+\varepsilon^{2^m})$$

– L'attaque consiste à distinguer $\boldsymbol{S}$ d'une suite aléatoire en calculant le nombre de fois où

$$\bigoplus_{\tau\in\langle T_{i_1},\ldots,T_{i_m}\rangle} S(t+\tau) = 0$$

pour $\varepsilon^{-2^{m+1}}$ valeurs de t, comme on a vu dans le lemme 2.6. Le test statistique consiste à comparer le nombre de fois où la relation s'annule à un seuil détérminé par la probabilité de fausse alarme et la probabilité de non-détection [Sie85, BG09].

Algorithme 2 Attaque par distingueur exploitant une approximation linéaire de la fonction de combinaison

```
/*Attaque*/
n = 0
pour t = 0 à ε^{-2^{m+1}} faire
   n = n + (⊕_τ S(t+τ))
fin pour
si n < seuil alors
   S est la suite chiffrante.
finsi
```

Complexité.

– En données :

$$\varepsilon^{-2^{m+1}} + \sum_{j=1}^{m} T_{i_j} \text{bits de suite chiffrante .}$$

– En temps :

$$\varepsilon^{-2^{m+1}} \text{opérations.}$$

Amélioration en combinant l'attaque avec une recherche exhaustive. On peut améliorer dans certains cas l'attaque précédente en effectuant une recherche exhaustive sur l'état initial de certains des m registres qui interviennent dans l'approximation linéaire ℓ. Supposons que l'on effectue cette recherche sur les m' premiers registres de l'approximation, d'indices $i_1, \ldots i_{m'}$. Alors, pour toute valeur fixée des m' premiers registres, la quantité $\bigoplus_{j=1}^{m'} x_{i_j}(t)$ est aussi fixée et on a :

$$\Pr\left[S(t) = \bigoplus_{j=1}^{m'} x_{i_j}(t) \oplus \bigoplus_{j=m'+1}^{m} x_{i_j}(t)\right] = \frac{1}{2}(1+\varepsilon).$$

Soit $\mathcal{T} = \langle T_{i_{m'+1}}, \ldots, T_{i_m}\rangle$. La relation de parité

$$PC_{\ell',\mathcal{T}}(t) = \bigoplus_{\tau\in\mathcal{T}} \left(S(t+\tau) \oplus \bigoplus_{j=1}^{m'} x_{i_j}(t+\tau)\right) = 0$$

possède $2^{m-m'}$ termes. Notons η le biais de la relation $\boldsymbol{P}C_{f,\mathcal{T}}$, c'est-à-dire

$$\Pr\left[\bigoplus_{\tau\in\langle T_{i_{m'+1}},\ldots,T_{i_m}\rangle} \left(S(t+\tau) \oplus \bigoplus_{j=1}^{m'} x_{i_j}(t+\tau)\right) = 0\right] = \frac{1}{2}(1+\eta).$$

On a donc

$$\eta \geq \varepsilon^{2^{m-m'}}.$$

Alors l'attaque consiste à faire une recherche exhaustive sur 2^k états initiaux où $k = \sum_{j=1}^{m'} L_{i_j}$. Pour chacun de ces états possibles, la relation de parité est calculée sur N échantillons pour détecter le biais. Comme remarqué dans [HJ07], la formule habituelle [HJ06, JMM06, MP06] du nombre d'échantillons dont on a besoin pour distinguer la suite chiffrante,

$$N \sim \frac{1}{\eta^2},$$

donne une valeur un peu sous-estimée.

En fait, ce problème peut être considéré comme un problème de décodage où le mot reçu correspond à la suite formée par les N évaluations des relations de parité. En effet, ce mot reçu peut être vu comme le résultat d'une transmission d'un mot de code par un canal binaire symétrique avec probabilité d'erreur $p = \frac{1}{2}(1-\eta)$. Alors, le nombre d'échantillons N dont on a besoin pour décoder est

$$N = \frac{k}{C(p)},$$

où $C(p)$ est la capacité du canal, c'est-à-dire, dans le cas du canal binaire symétrique avec probabilité d'erreur p,

$$C(p) = 1 + p\log_2(p) + (1-p)\log_2(1-p).$$

De plus quand $p = \frac{1}{2}(1-\eta)$ avec $|\eta| \ll 1$, on trouve que $C(p) \sim \frac{\eta^2}{2\ln(2)}$, et donc

$$N \sim \frac{2k\ln 2}{\eta^2}, \tag{2.1}$$

où 2^k est le nombre des états internes possibles des registres sur lesquels on fait la recherche exhaustive.

La complexité en données de l'attaque est donc réduite à :

$$2\ln 2 \sum_{j=1}^{m'} L_{i_j} \varepsilon^{-2^{m-m'+1}} + \sum_{i=m'+1}^{m} T_{i_j}.$$

La complexité en temps est maintenant égale à :

$$2\ln 2 \sum_{j=1}^{m'} L_{i_j} \varepsilon^{-2^{m-m'+1}} 2^{\sum_{j=1}^{m'} L_{i_j}}.$$

On voit ici que la complexité en données diminue quand le nombre de registres sur lesquels on fait une recherche exhaustive augmente.

La complexité en temps, elle, dépend de la longueur des registres sur lesquels on fait une recherche exhaustive et aussi du biais. Il faut trouver un compromis entre les deux termes.

Exemple sur Achterbahn version 1. Johansson, Meier et Muller utilisent cette attaque pour Achterbahn version 1. La fonction

$$\ell' = y_1 \oplus y_2 \oplus y_3 \oplus y_4 \oplus y_6$$

est l'approximation linéaire de f utilisée et son biais est $\mathcal{E}(f + \ell') = \varepsilon = \frac{1}{2}$. La relation de parité va avoir $2^5 = 32$ termes.

La complexité en temps est de 2^{64} et en données de 2^{64} pour la version réduite et aussi pour la complète.

Si l'on fait une recherche exhaustive sur l'état du premier registre (longueur 22), la complexité en temps devient 2^{54} et en données 2^{32} pour la version réduite. Dans la version complète, en prenant en compte les filtres linéaires, la complexité en temps devient alors 2^{61}.

2.3 Premières améliorations de l'attaque exploitant une approximation de la fonction de combinaison

2.3.1 Achterbahn version 2

Une fois que la première version d'Achterbahn a été cryptanalysée par T. Johansson, W. Meier et F. Muller [JMM06] de la façon qu'on a vue précédemment, les auteurs d'Achterbahn ont proposé différentes fonctions booléennes de combinaison dans [GGK05b]. Avec ces fonctions, Achterbahn ne résistait pas aux versions des attaques précédentes et ils ont donc proposé une nouvelle version de leur chiffrement à flot. Achterbahn version 2 [GGK06b] est formé de 10 registres au lieu de huit, de longueurs entre 19 et 32 bits.

La nouvelle fonction booléenne proposée est plus dense et 5-résiliente :

$$\begin{aligned} f(y_1,\ldots,y_{10}) &= y_1 \oplus y_2 \oplus y_3 \oplus y_9 \oplus g(y_4,y_5,y_6,y_7,y_{10}) \oplus (y_8 \oplus y_9) \\ &\quad (g(y_4,y_5,y_6,y_7,y_{10}) \oplus h(y_1,y_2,y_3,y_4,y_5,y_6,y_7,y_{10})) \end{aligned}$$

avec

$$\begin{aligned} g(y_4,y_5,y_6,y_7,y_{10}) &= (y_4(y_5 \oplus y_{10} \oplus y_5y_{10}) \oplus y_5(y_6 \oplus y_7 \oplus y_6y_7) \\ &\quad \oplus y_6(y_4 \oplus y_{10} \oplus y_4y_{10}) \oplus y_7(y_4 \oplus y_6 \oplus y_4y_6) \\ &\quad \oplus y_{10}(y_5 \oplus y_7 \oplus y_5y_7) \end{aligned}$$

et

$$\begin{aligned} h(y_1,y_2,y_3,y_4,y_5,y_6,y_7,y_{10}) &= y_2 \oplus y_5 \oplus y_7 \oplus y_{10} \oplus y_3 \oplus y_4 \oplus y_3y_6 \oplus y_4y_6 \\ &\quad \oplus (y_1 \oplus y_2)(y_3 \oplus y_3y_6 \oplus y_4y_6 \oplus y_5y_6). \end{aligned}$$

De plus, à cause de l'existence d'une attaque utilisant des approximations quadratiques qui nécessite une suite chiffrante de 2^{64} bits, les auteurs ont limité la longueur maximale de suite chiffrante engendrée à partir d'un même couple (K, IV) à 2^{63} bits.

Bien que la fonction booléenne soit clairement plus complexe que celle de la version précédente, la faiblesse due à la petite longueur des registres autonomes existe toujours.

Cette version a, elle aussi, été cryptanalysée : M. Hell et T. Johansson [HJ06] ont publié la cryptanalyse de la version 2 d'Achterbahn en juin 2006. Elle a une complexité, pour la version réduite comme pour la complète, de 2^{59}.

Nous allons maintenant décrire cette attaque sur le modèle général des générateurs par combinaison de FSRs.

2.3.2 Amélioration en décimant par la période d'un registre

La cryptanalyse de Achterbahn version 2 par Hell et Johansson [HJ06] est une attaque par distingueur qui utilise une relation de parité biaisée entre les bits de la suite chiffrante et qui permet aussi de récupérer l'état initial de quelques registres et ensuite de retrouver la clé. Il s'agit d'une attaque par corrélation qui utilise une approximation quadratique q de la fonction de combinaison f :

$$q(y_1,\ldots,y_n) = \bigoplus_{j=1}^{s} y_{i_j} \oplus \bigoplus_{i=1}^{m} (y_{j_i}y_{k_i})$$

avec m termes quadratiques et qui vérifie :

$$\Pr[f(y_1,\ldots,y_n) = q(y_1,\ldots,y_n)] = \frac{1}{2}(1+\varepsilon).$$

Nous construisons des relations de parité comme celles introduites précédemment, qui font disparaître les termes quadratiques en sommant :

$$q(t) = \bigoplus_{j=1}^{s} x_{i_j}(t) \oplus \bigoplus_{i=1}^{m} x_{j_i}(t)x_{k_i}(t)$$

à 2^m instants différents $(t+\tau)$, où τ varie dans l'ensemble

$$\mathcal{T} = \langle T_{j_1}T_{k_1}, \ldots, T_{j_m}T_{k_m} \rangle.$$

Avec tout cela on peut construire la relation de parité définie par :

$$PC_{q,\mathcal{T}}(t) = \bigoplus_{\tau \in \mathcal{T}} q(t+\tau) = \bigoplus_{\tau \in \mathcal{T}} \left(x_{i_1}(t+\tau) \oplus \ldots \oplus x_{i_s}(t+\tau) \right).$$

Maintenant, on décime la suite $\boldsymbol{PC_{q,\mathcal{T}}}$ par les périodes de r suites parmi $\boldsymbol{x_{i_1}}, \ldots, \boldsymbol{x_{i_s}}$. On peut supposer ici, sans perdre de généralité, que les périodes des r premières suites ont été choisies. Maintenant on peut définir une nouvelle relation de parité, $\boldsymbol{PC'_{q,\mathcal{T}}}$, par :

$$PC'_{q,\mathcal{T}}(t) = PC'_{q,\mathcal{T}}(tT_{i_1} \ldots T_{i_r}), \quad \forall t \geq 0.$$

De cette façon, l'influence de ces r registres dans la relation de parité $\boldsymbol{PC'_{q,\mathcal{T}}}$ correspond à l'addition d'une constante pour tout $t \geq 0$, donc 0 ou 1 pour toutes les relations de parité.

L'attaque consiste à réaliser la recherche exhaustive sur les états initiaux des $(s-r)$ registres restants, c'est-à-dire ceux avec des indices $i_{r+1}, \ldots, i_s$. Pour chaque valeur possible de ces états initiaux, on calcule la suite :

$$PC(t) = \bigoplus_{\tau \in \mathcal{T}} \left[S(tT_{i_1} \ldots T_{i_r} + \tau) \oplus \bigoplus_{j=r+1}^{s} x_{i_j}(tT_{i_1} \ldots T_{i_r} + \tau) \right] \tag{2.2}$$

avec $\mathcal{T} = \langle T_{j_1}T_{k_1}, \ldots, T_{j_m}T_{k_m} \rangle$. On a

$$\Pr[PC(t) = 0] \geq \frac{1}{2}(1 + \varepsilon^{2^m}).$$

Hell et Johansson ont observé après avoir publié ces attaques dans [HJ06] que le biais total pouvait être beaucoup plus grand que cette borne. Nous montrerons dans la suite qu'on peut prouver que l'égalité se vérifie dans des cas particuliers, comme remarqué dans [GG07]. Un cas intéressant d'égalité est quand f est t-résiliente, et qu'on construit les relations de parité à partir des termes d'une approximation linéaire de $(t+1)$ variables. Cette expression nous donnera aussi les biais des relations de parité utilisées dans [HJ07, HJ06] construites à partir des approximations quadratiques, car ces relations peuvent aussi être construites à partir des approximations linéaires. Ce résultat va être utilisé dans nos attaques, car nous allons travailler avec des approximations linéaires de $(t+1)$ variables. Le chapitre suivant sera entièrement consacré au calcul du biais des relations de parité par des algorithmes efficaces et aux cas où la borne fournie par le piling-up lemma est atteinte, comme celui que nous venons de mentionner.

Complexité.

Si on utilise la borne sur le biais qu'on vient de calculer, on peut distinguer la suite chiffrante $\boldsymbol{S}$ d'une suite aléatoire et aussi retrouver les états initiaux des $(s-r)$ registres d'indice $i_{r+1}, \ldots, i_s$.

- On a 2^m termes dans chaque relation de parité. Cela veut dire qu'on a besoin de calculer

$$2 \ln 2 \varepsilon^{-2^{m+1}} \sum_{j=r+1}^{s} (L_{i_j} - 1)$$

valeurs de $PC(t)$ pour effectuer l'attaque par distingueur. De plus, $PC(t)$ est défini par (2.2), ce qui signifie que l'attaque a besoin de

$$2 \ln 2 \varepsilon^{-2^{m+1}} 2^{\sum_{j=1}^{r} L_{i_j}} \sum_{j=r+1}^{s} (L_{i_j} - 1) + \sum_{i=1}^{m} 2^{L_{j_i} + L_{k_i}} \text{ bits de la suite chiffrante,}$$

où les L_{i_j} sont les longueurs des registres associés aux périodes par lesquelles on a décimé, et le dernier terme correspond à la distance maximale entre les bits qui apparaissent dans chaque relation de parité.

- La complexité en temps est

$$2 \ln 2 \varepsilon^{-2^{m+1}} 2^{m + \sum_{j=r+1}^{s} (L_{i_j} - 1)} \sum_{j=r+1}^{s} (L_{i_j} - 1)$$

où $i_{r+1}, \ldots, i_s$ sont les indices des registres sur lesquels ont fait une recherche exhaustive, et donc ceux dont on va trouver l'état initial.

Exemple sur Achterbahn version 2.

Hell et Johansson [HJ06] ont utilisé cette attaque contre Achterbahn version 2 avec l'approximation quadratique suivante :

$$Q(x_1, \ldots, x_{10}) = x_1 \oplus x_2 \oplus x_3 x_8 \oplus x_4 x_6.$$

Ensuite, ils déciment par la période du deuxième registre, qui a une longueur de 22. Ensuite, ils font une recherche exhaustive sur le premier registre, de longueur 19. La complexité en temps est de 2^{67} et la complexité en données 2^{64} . La complexité donnée dans [HJ06], $2^{59.02}$, est obtenue en utilisant l'estimation $N = \varepsilon^{-2}$ à la place de celle donnée par la formule (2.1). Utilisant la petite longueur des registres, ils ont proposé un algorithme qui permet de réduire la complexité en temps en-dessous de la complexité en données, donc la complexité finale de l'attaque sera 2^{64}.

2.3.3 Amélioration de l'attaque avec l'utilisation d'approximations linéaires

Je vais maintenant montrer comment j'ai amélioré l'attaque précédente contre Achterbahn version 2 et réduit la complexité à 2^{53}.

Pour cette attaque j'ai utilisé l'idée d'associer les variables pour réduire le nombre de termes qu'on aura dans la relation de parité, comme vu précédemment dans [JMM06]. Le seul effet négatif que cela pourrait avoir sur la complexité finale de l'attaque est d'augmenter le nombre de bits nécessaires de la suite chiffrante ; mais en faisant attention, on peut garder la même complexité en données et réduire la complexité en temps.

L'approximation choisie. Pour commencer, j'ai recherché toutes les approximations quadratiques de f avec un ou deux termes quadratiques, à la suite de l'attaque de Hell et Johansson, qui exploitait une approximation quadratique. Ensuite, en prenant en compte un compromis entre le nombre de termes, le nombre de variables, le biais, etc,... j'ai démontré qu'aucune approximation n'était meilleure pour l'attaque que les approximations linéaires. Il convient de remarquer que, comme la fonction de combinaison f est 5-résiliente, toute approximation de f a au moins 6 variables. De plus, le plus grand biais qui correspond à une approximation de f par une fonction à 6 variables est donné par une fonction de degré un, comme prouvé dans [CT00, Th.3]. Après avoir analysé toutes les approximations linéaires de f, on a trouvé que la meilleure est :

$$g(x_1, \ldots, x_{10}) = x_8 \oplus x_6 \oplus x_4 \oplus x_3 \oplus x_2 \oplus x_1.$$

Elle vérifie

$$\Pr\left[f(x_1, \ldots, x_{10}) = g(x_1, \ldots, x_{10})\right] = \frac{1}{2}(1 + 2^{-3}).$$

Relations de parité. Pour $\mathcal{T} = \langle T_1T_8, T_2T_6 \rangle$, on construit une relation de parité de la façon suivante :

$$PC_{g,\mathcal{T}}(t) = g(t) \oplus g(t + T_1T_8) \oplus g(t + T_2T_6) \oplus g(t + T_1T_8 + T_2T_6),$$

avec

$$g(t) = x_8(t) \oplus x_6(t) \oplus x_4(t) \oplus x_3(t) \oplus x_2(t) \oplus x_1(t).$$

Les termes des suites $\boldsymbol{x_8}$, $\boldsymbol{x_6}$, $\boldsymbol{x_2}$, $\boldsymbol{x_1}$ vont disparaître et, donc, $\boldsymbol{PC_{g,\mathcal{T}}}$ est une suite qui ne dépend que des suites $\boldsymbol{x_3}$ et $\boldsymbol{x_4}$.

D'après le piling-up lemma, le biais de

$$PC(t) = PC_{g,\mathcal{T}}(t) = S(t) \oplus S(t + T_1T_8) \oplus S(t + T_2T_6) \oplus S(t + T_1T_8 + T_2T_6)$$

est supérieur ou égal à $2^{-3\times4} = 2^{-12}$. Ceci veut dire qu'on a besoin de

$$2^{3\times4\times2} \times 2 \times (L_4 - 1) \times \ln(2) = 2^{29}$$

évaluations de la relation de parité pour détecter le biais. Si on décime la suite $\boldsymbol{PC}$ par la période du registre R_3, on aura besoin de

$$2^{29}T_3 + T_1T_8 + T_2T_6 = 2^{29+23} + 2^{29+19} + 2^{27+22} = 2^{52} \text{ bits de suite chiffrante,}$$

et la complexité en temps sera de $2^{29} \times 2^{L_4-1} = 2^{53}$ car on ne fait une recherche exhaustive que sur l'état initial du registre R_4. Cette complexité est 2^{53} alors que la complexité en temps de l'attaque précédente était de 2^{64}.

2.4 Attaques exploitant des approximations linéaires de la fonction de combinaison sur Achterbahn-80/128

2.4.1 Description d'Achterbahn-80/128

Après la cryptanalyse des deux premières versions [JMM06, HJ06], Achterbahn a été à nouveau modifié, donnant naissance à Achterbahn-128/80 [GGK06a] publié en juin 2006. Il s'agit en fait de deux variantes d'un même système, Achterbahn-128 ayant une clé de 128 bits alors qu'Achterbahn-80 utilise une clé de 80 bits. Dans la première version d'Achterbahn-128/80, la longueur maximale de suite chiffrante produite avec les mêmes valeurs de K et IV était limitée à 2^{63}.

Spécifications principales de Achterbahn-128.

Il s'agit d'un générateur de suite chiffrante composé de 13 NLFSRs primitifs, dont la période est $2^L - 1$ pour un registre de longueur L. La longueur du registre i est $L_i = 21 + i$ pour $i = 0, 1, \ldots, 12$. La suite qu'on utilise comme entrée de la fonction booléenne n'est pas directement la suite qui sort du registre mais une version décalée de celle-ci. Le décalage dépend du numéro du registre, mais il est connu et reste toujours le même au cours du temps.

La sortie du générateur est celle de la fonction booléenne F à 13 variables où les entrées correspondent aux suites de sortie des registres décalées correctement. La fonction booléenne F est la suivante :

$$x_6 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_8 \oplus x_{10} \oplus x_{12} \oplus x_{13} \oplus x_1x_6 \oplus x_3x_{11} \oplus x_3x_{12} \oplus x_9x_5 \oplus x_5x_{13} \oplus x_7x_6 \oplus x_7x_9 \oplus x_7x_{11} \oplus x_7x_{12} \oplus x_7x_{13} \oplus x_8x_9 \oplus x_8x_{13} \oplus x_9x_{10} \oplus x_9x_{11} \oplus x_{10}x_{11} \oplus x_{10}x_{12} \oplus x_{10}x_{13} \oplus x_{11}x_{13} \oplus x_1x_6x_9 \oplus x_1x_6x_{11} \oplus x_1x_6x_{12} \oplus x_1x_6x_{13} \oplus x_2x_3x_9 \oplus x_2x_3x_{13} \oplus x_2x_5x_{11} \oplus x_2x_5x_{12} \oplus x_2x_9x_{10} \oplus x_2x_{10}x_{11} \oplus x_2x_{10}x_{12} \oplus x_2x_{10}x_{13} \oplus x_3x_4x_9 \oplus x_3x_4x_{13} \oplus x_3x_5x_9 \oplus x_3x_5x_{11} \oplus x_3x_5x_{12} \oplus x_3x_5x_{13} \oplus x_3x_8x_9 \oplus x_3x_8x_{13} \oplus x_3x_9x_{11} \oplus x_3x_9x_{12} \oplus x_3x_{10}x_{11} \oplus x_3x_{10}x_{12} \oplus x_3x_{11}x_{13} \oplus x_3x_{12}x_{13} \oplus x_4x_5x_9 \oplus x_4x_5x_{13} \oplus x_4x_9x_{10} \oplus x_4x_{10}x_{13} \oplus x_5x_8x_9 \oplus x_5x_8x_{13} \oplus x_5x_9x_{10} \oplus x_5x_{10}x_{13} \oplus x_6x_7x_9 \oplus x_6x_7x_{11} \oplus x_6x_7x_{12} \oplus x_6x_7x_{13} \oplus x_7x_9x_{11} \oplus x_7x_9x_{12} \oplus x_7x_{11}x_{13} \oplus x_7x_{12}x_{13} \oplus x_8x_9x_{10} \oplus x_8x_{10}x_{13} \oplus x_9x_{10}x_{11} \oplus x_9x_{10}x_{12} \oplus x_{10}x_{11}x_{13} \oplus x_{10}x_{12}x_{13} \oplus x_1x_6x_9x_{11} \oplus x_1x_6x_9x_{12} \oplus x_1x_6x_{11}x_{13} \oplus x_1x_6x_{12}x_{13} \oplus x_2x_3x_4x_9 \oplus x_2x_3x_4x_{13} \oplus x_2x_3x_8x_9 \oplus x_2x_3x_8x_{13} \oplus x_2x_4x_6x_9 \oplus x_2x_4x_6x_{13} \oplus x_2x_4x_9x_{10} \oplus x_2x_4x_{10}x_{13} \oplus x_2x_5x_9x_{11} \oplus x_2x_5x_9x_{12} \oplus x_2x_5x_{11}x_{13} \oplus x_2x_5x_{12}x_{13} \oplus x_2x_6x_8x_9 \oplus x_2x_6x_8x_{13} \oplus x_2x_8x_9x_{10} \oplus x_2x_8x_{10}x_{13} \oplus x_2x_9x_{10}x_{11} \oplus x_2x_9x_{10}x_{12} \oplus x_2x_{10}x_{11}x_{13} \oplus$

$x_2x_{10}x_{12}x_{13} \oplus x_3x_4x_5x_9 \oplus x_3x_4x_5x_{13} \oplus x_3x_4x_6x_9 \oplus x_3x_4x_6x_{13} \oplus x_3x_5x_8x_9 \oplus x_3x_5x_8x_{13} \oplus x_3x_5x_9x_{11} \oplus x_3x_5x_9x_{12} \oplus x_3x_5x_{11}x_{13} \oplus x_3x_5x_{12}x_{13} \oplus x_3x_6x_8x_9 \oplus x_3x_6x_8x_{13} \oplus x_3x_9x_{10}x_{11} \oplus x_3x_9x_{10}x_{12} \oplus x_3x_{10}x_{11}x_{13} \oplus x_3x_{10}x_{12}x_{13} \oplus x_4x_5x_9x_{10} \oplus x_4x_5x_{10}x_{13} \oplus x_5x_8x_9x_{10} \oplus x_5x_8x_{10}x_{13} \oplus x_6x_7x_9x_{11} \oplus x_6x_7x_9x_{12} \oplus x_6x_7x_{11}x_{13} \oplus x_6x_7x_{12}x_{13}$.

Ses propriétés les plus importantes [1] sont :
- équilibre
- degré algébrique = 4
- ordre de résilience = 8
- non-linéarité = 3584
- immunité algébrique = 4.

Spécifications principales de Achterbahn-80.

Dans ce cas, le générateur est formé de 11 registres, qui correspondent aux registres 1 à 11 du cas précédent, c'est-à-dire tous sauf le premier et le dernier. La fonction booléenne de sortie, G, est une restriction de F :

$$G(x_1, \ldots, x_{11}) = F(0, x_1, \ldots, x_{11}, 0) \tag{2.3}$$

Donc le générateur Achterbahn-128 contient Achterbahn-80 comme sous-structure. Les propriétés les plus importantes de G sont :
- équilibre
- degré algébrique = 4
- ordre de résilience = 6
- non-linéarité = 896
- immunité algébrique = 4.

Initialisation des registres.

L'initialisation des registres est faite grâce à une clé K et une valeur publique IV. Nous notons $K||IV$ la concaténation des bits de K avec les bits de IV. Dans un premier temps, on introduit les bits de $K||IV$ en parallèle dans les registres. Puis, on fait des mises à jour des registres en ajoutant les bits restants encore de $K||IV$. Chaque registre produit un bit de sortie. Ces bits sont combinés par la fonction booléenne de sortie, qui produit un nouveau bit. On fait alors des nouvelles mises à jour des registres en ajoutant ce dernier. Cette opération se répète 32 fois. Ensuite on met le dernier bit de chaque registre à 1, pour assurer que l'état où toutes les cellules des registres ont la valeur 0 ne puisse pas se produire. Finalement on fait encore quelques mises à jour à blanc des registres.

Cet algorithme a été modifié par rapport aux versions précédentes. Le but de cette modification est qu'il ne soit plus possible de récupérer la clé K directement à partir des états initiaux des registres.

[1] Les propriétés des fonctions booléennes ainsi que leur importance dans les systèmes de chiffrement à flot sont décrites dans [Can06].

2.4.2 Attaque par distingueur sur Achterbahn-80

Cette version, ainsi que Achterbahn-128, a aussi été cryptanalysée indépendamment de mes travaux par Hell and Johansson [HJ07], mais avec des complexités plus grandes que celles de mes attaques. Cette attaque-ci est très similaire à l'amélioration de l'attaque contre Achterbahn version 2 qui a été décrite dans la section précédente.

Mon attaque utilise l'approximation linéaire suivante de la fonction de combinaison G :

$$\ell(x_1, \ldots, x_{11}) = x_1 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_{10}.$$

Comme G est 6-résiliente, ℓ est la meilleure approximation par une fonction à 7 variables. Pour $\ell(t) = x_1(t) \oplus x_3(t) \oplus x_4(t) \oplus x_5(t) \oplus x_6(t) \oplus x_7(t) \oplus x_{10}(t)$, la suite chiffrante $\boldsymbol{S}$ vérifie

$$\Pr[S(t) = \ell(t)] = \frac{1}{2}(1 - 2^{-3}).$$

Relations de parité. On construit les relations de parité de la façon suivante :

$$PC(t) = \ell(t) \oplus \ell(t + T_4T_7) \oplus \ell(t + T_6T_5) \oplus \ell(t + T_4T_7 + T_6T_5).$$

Les termes qui contiennent les suites $\boldsymbol{x_4}$, $\boldsymbol{x_5}$, $\boldsymbol{x_6}$, $\boldsymbol{x_7}$ disparaissent dans $\boldsymbol{PC}$, donc $\boldsymbol{PC}$ dépend exclusivement des suites $\boldsymbol{x_1}$, $\boldsymbol{x_3}$ et $\boldsymbol{x_{10}}$.

Sommer quatre fois l'approximation a pour effet de multiplier le biais au moins quatre fois, donc le biais de

$$PC(t) = S(t) \oplus S(t + T_7T_4) \oplus S(t + T_6T_5) \oplus S(t + T_7T_4 + T_6T_5)$$

est supérieur ou égal à $2^{-4\times3}$.

On décime maintenant $\boldsymbol{PC}$ par la période de R_{10}, qui apparaît dans la relation de parité, et on redéfinit ainsi une nouvelle relation de parité

$$PC'(t) = PC(t(2^{31} - 1)).$$

Ensuite, il faut effectuer une recherche exhaustive sur les états initiaux des registres R_1 et R_3. Donc on a besoin de

$$2^{3\times4\times2} \times 2 \times (46 - 2) \times \ln(2) = 2^{30}$$

évaluations de la relation de parité $PC'(t)$ pour détecter le biais. La complexité en temps est de $2^{30} \times 2^{L_1+L_3-2} = 2^{74}$.

Le nombre de bits de la suite chiffrante dont on a besoin est

$$2^{30} \times T_{10} + T_4T_7 + T_6T_5 = 2^{61}.$$

2.4.3 Amélioration de l'attaque avec un algorithme qui accélère la recherche exhaustive

Maintenant je vais présenter une attaque par distingueur sur la version d'Achterbahn avec 128 bits de clé, Achterbahn-128. Cette attaque permet également de retrouver l'état initial de deux registres.

On considère l'approximation suivante de la fonction de combinaison F :

$$\ell(x_0, \ldots, x_{12}) = x_0 \oplus x_3 \oplus x_7 \oplus x_4 \oplus x_{10} \oplus x_8 \oplus x_9 \oplus x_1 \oplus x_2.$$

Alors, pour

$$\ell(t) = x_0(t) \oplus x_3(t) \oplus x_7(t) \oplus x_4(t) \oplus x_{10}(t) \oplus x_8(t) \oplus x_9(t) \oplus x_1(t) \oplus x_2(t),$$

on a

$$\Pr[S(t) = \ell(t)] = \frac{1}{2}(1 + 2^{-3}).$$

Relations de parité. La période de la suite correspondant à la somme des registres 0, 3 et 7 est le ppcm de T_0, T_3 et T_7, *i.e.* $2^{59.3}$ car T_0 T_3 et T_7 ont des diviseurs communs. On va noter cette valeur $T_{0,3,7}$. Soit $\mathcal{T} = \langle T_{0,3,7}, T_4T_{10}, T_8T_9 \rangle$.

Si on construit des relations de parité de la façon suivante :

$$PC_{\ell,\mathcal{T}}(t) = \bigoplus_{\tau \in \mathcal{T}} \ell(t + \tau),$$

les termes contenant les suites $\boldsymbol{x_0}$, $\boldsymbol{x_3}$, $\boldsymbol{x_7}$, $\boldsymbol{x_4}$, $\boldsymbol{x_{10}}$, $\boldsymbol{x_8}$, $\boldsymbol{x_9}$ disparaissent de $\boldsymbol{PC_{\ell,\mathcal{T}}}$, donc $\boldsymbol{PC_{\ell,\mathcal{T}}}$ ne dépend que des suites $\boldsymbol{x_1}$ et $\boldsymbol{x_2}$:

$$\begin{aligned} PC_{\ell,\mathcal{T}}(t) &= \bigoplus_{\tau \in \langle T_{0,3,7}, T_{4,10}, T_{8,9} \rangle} \ell(t + \tau) \\ &= \bigoplus_{\tau \in \langle T_{0,3,7}, T_{4,10}, T_{8,9} \rangle} x_1(t + \tau) \oplus x_2(t + \tau) \\ &= \sigma_1(t) \oplus \sigma_2(t), \end{aligned}$$

où $\boldsymbol{\sigma_1}$ et $\boldsymbol{\sigma_2}$ sont les relations de parité calculées sur les suites générées par R_1 et R_2.

Sommer huit fois l'approximation a pour effet de multiplier le biais au moins huit fois, donc le biais de l'approximation de

$$\sigma(t) = PC_{F,\mathcal{T}}(t) = \bigoplus_{\tau \in \mathcal{T}} S(t + \tau)$$

par $\sigma_1(t) \oplus \sigma_2(t)$, où $\boldsymbol{S}$ est la suite chiffrante, est supérieur ou égal à $2^{-8\times3}$. Donc :

$$\Pr[\sigma(t) \oplus \sigma_1(t) \oplus \sigma_2(t) = 0] \geq \frac{1}{2}(1 + \varepsilon^8).$$

Nous verrons dans la suite que cette borne sur le biais est atteinte. Ceci veut dire qu'on a besoin de

$$N = 2^{3\times 8\times 2} \times 2 \times (45-2) \times \ln(2) = 2^{54}$$

valeurs de $\sigma(t)\oplus\sigma_1(t)\oplus\sigma_2(t)$ pour détecter le biais, quand on fait une recherche exhaustive sur les registres R_1 et R_2.

Je décris maintenant un nouvel algorithme pour calculer la somme $\sigma(t) \oplus \sigma_1(t) \oplus \sigma_2(t)$ pour les N valeurs de t. Cet algorithme a une complexité plus petite que l'algorithme trivial qui calcule les 2^{54} relations de parité pour tous les états initiaux des registres R_1 et R_2. Ici on utilise $(2^{54}-2^8)$ valeurs de t car ce nombre est un multiple de T_2 : $(2^{54}-2^8) = T_2\times(2^{31}+2^8)$. On peut alors écrire :

$$\begin{aligned}
\sum_{t'=0}^{2^{54}-2^8-1} \sigma(t') \oplus \ell(t') &= \sum_{k=0}^{T_2-1} \sum_{t=0}^{2^{31}+2^8-1} \sigma(T_2t+k) \oplus \ell(T_2t+k) \\
&= \sum_{k=0}^{T_2-1} \sum_{t=0}^{2^{31}+2^8-1} \sigma(T_2t+k) \oplus \sigma_1(T_2t+k) \oplus \sigma_2(T_2t+k) \\
&= \sum_{k=0}^{T_2-1} \left[(\sigma_2(k)\oplus 1) \left(\sum_{t=0}^{2^{31}+2^8-1} \sigma(T_2t+k) \oplus \sigma_1(T_2t+k) \right) + \right. \\
&\quad \left. \sigma_2(k) \left((2^{31}+2^8) - \sum_{t=0}^{2^{31}+2^8-1} \sigma(T_2t+k) \oplus \sigma_1(T_2t+k) \right) \right],
\end{aligned}$$

parce que quand t varie, $\sigma_2(T_2t+k)$ est constante pour une valeur fixée de k.

Maintenant, on peut obtenir $\sigma(t)$ à partir de la suite chiffrante et on peut réaliser une recherche exhaustive sur l'état initial du R_1. Plus précisément :

- On choisit un état initial pour le registre R_2, par exemple l'état tout à 1. On calcule et on stocke un vecteur binaire V_2 de longueur T_2 :

  $$V_2[k] = \sigma_2(k),$$

  où la suite $\boldsymbol{x_2}$ est générée à partir de l'état initial choisi. La complexité de cette étape est $T_2 \times 2^3$ opérations.
- Pour chaque état initial possible du registre R_1 :
  - on calcule et on stocke un vecteur V_1 formé par T_2 entiers de 32 bits :

    $$V_1[k] = \sum_{t=0}^{2^{31}+2^8-1} \sigma(T_2t+k) \oplus \sigma_1(T_2t+k).$$

    La complexité de cette étape est $2^{54} \times (2^4+2^5) = 2^{59.58}$ pour chaque état initial possible pour le registre R_1, où 2^4 correspond au nombre d'opérations requises pour calculer chaque $(\sigma(t)+\sigma_1(t))$ et $(2^{31}+2^8)\times 2^5 = (2^{31}+2^8)\times 32$ est le coût de sommer $2^{31}+2^8$ entiers de 32 bits.

- Pour chaque i possible de 0 à $T_2 - 1$:
  - on définit V_2' de longueur T_2 :

$$V_2'[k] = V_2[k + i \mod T_2].$$

    En fait, $(V_2'[k])_{k<T_2}$ correspond à $(\sigma_2(k))_{k<T_2}$ quand l'état initial du registre R_2 correspond à l'état interne après avoir fait tourner le registre R_2 i fois en partant de l'état initial tout à un.
  - Avec les deux vecteurs qu'on a obtenus, on calcule :

$$\sum_{k=0}^{T_2-1} \left[(V_2'[k] \oplus 1)\, V_1[k] + V_2'[k] \left(2^{31} + 2^8 - V_1[k]\right) \right]. \tag{2.4}$$

quand on fait cela avec le bons états initiaux des registres R_1 et R_2, on trouvera le biais attendu. La différence principale avec la recherche exhaustive classique utilisée dans [HJ06, HJ07, JMM06] est que le vecteur $V_1[k]$ est calculé indépendamment du choix de l'état initial de R_2. En comparaison, l'algorithme classique a une complexité en temps de 2^{102} pour les mêmes paramètres.

Algorithme 3 Attaque sur Achterbahn-128 : algorithme pour trouver les états initiaux des registres R_1 et R_2

Initialiser R_2 à l'état tout à 1
pour $k = 0$ à $T_2 - 1$ **faire**
  $V_2[k] = \sigma_2(k)$
fin pour
pour chaque état initial possible pour R_1 **faire**
  pour $k = 0$ à $T_2 - 1$ **faire**
    $V_1[k] = \sum_{t=0}^{2^{31}+2^8-1} \sigma(T_2 t + k) \oplus \sigma_1(T_2 t + k)$
  fin pour
  pour chaque état initial i possible pour R_2 **faire**
    pour $k = 0$ à $T_2 - 1$ **faire**
      $V_2'[k] = V_2[k + i \mod T_2]$
    fin pour
    $\sum_{k=0}^{T_2-1} \left[(V_2'[k] \oplus 1)\, V_1[k] + V_2'[k] \left(2^{31} + 2^8 - V_1[k]\right) \right]$
    si le biais est détecté **alors**
      retourner les états initiaux de R_1 et R_2
    finsi
  fin pour
fin pour

La complexité en temps de l'attaque est :

$$2^{L_1-1} \times \left[2^{54} \times \left(2^4 + 2^5\right) + T_2 \times 2 \times T_2 \times 2^5 \right] + T_2 \times 2^3 = 2^{80.58},$$

où $2 \times T_2 \times 2^5$ est le temps nécessaire pour calculer la somme décrite par (2.4). En fait, on peut accélérer le processus en réécrivant la somme (2.4) de la façon suivante

$$\sum_{k=0}^{T_2-1} (-1)^{V_2[k+i]} \left(V_1[k] - \frac{2^{31}+2^8}{2} \right) + T_2 \frac{2^{31}+2^8}{2}.$$

Le but maintenant est trouver le i qui maximise la somme, ce qui est la même chose que de calculer la corrélation mutuelle maximale de deux suites de longueur T_2. On peut faire cela d'une façon efficace avec une transformée de Fourier rapide, comme expliqué dans [Bla85, pages 306-312] ou dans [CJM02, Ber07, BGM06, Jou09]. La complexité finale sera en $O(T_2 \log T_2)$. Mais ceci ne change pas notre complexité totale, car le terme le plus grand est le premier.

La complexité en temps est donc, finalement :

$$2^{L_1-1} \times \left[2^{54} \times \left(2^4 + 2^5 \right) + O(T_2 \log T_2) \right] + T_2 \times 2^3 = 2^{80.58}.$$

La longueur de la suite chiffrante dont on a besoin est $T_{0,3,7} + T_4 T_{10} + T_8 T9 + 2^{54} < 2^{61}$ bits.

On peut appliquer l'algorithme à l'attaque sur Achterbahn-80 décrite dans la section précédente et sa complexité en temps est alors réduite à :

$$2^{L_1-1} \times \left[2^{30} \times \left(2^3 + 2^{2.59} \right) + O(T_3 \log T_3) \right] + T_3 \times 2^2 = 2^{54.8}.$$

Nous pouvons aussi l'appliquer sur Achterbahn-v2, ce qui conduit à une complexité en temps de 2^{32}.

2.4.4 Retrouver la clé

Comme l'expliquent Hell et Johansson dans [HJ07], si on retrouve les états initiaux de tous les registres, on est capable de retrouver la clé aussi car toutes les étapes d'initialisation qui n'utilisent pas la clé deviennent inversibles. On peut voir facilement qu'une fois qu'on a trouvé les états initiaux de deux registres, la complexité de la recherche des autres est plus petite. Pour les autres registres qui apparaissent dans l'approximation utilisée, c'est évident : on applique la même méthode mais simplifiée par le fait que maintenant on connaît deux variables. Pour les autres registres on peut utiliser la même méthode mais avec d'autres approximations linéaires en tirant parti des variables déjà connues. Une fois qu'on a trouvé l'état initial de tous les registres, on peut inverser tous les étapes d'initialisation jusqu'à la fin de la deuxième étape, qui correspond à l'introduction des bits de la clé. À ce moment il y a deux méthodes proposées dans [HJ07]. La première consiste à faire tourner le registre R_i en sens inverse $(|k| - L_i)$ fois pour chaque i. On fait ceci pour toutes les valeurs possibles des $|k| - L_m$ derniers bits de la clé, où L_m vaut 21 pour Achterbahn-128 et 22 pour Achterbahn-80. Quand tous les registres ont les mêmes L_m premiers bits, on a trouvé les bons $|k| - L_m$ bits de la clé. La deuxième méthode proposée est une attaque dans le milieu avec compromis temps-mémoire comme celle décrite dans [JMM06]. On trouve les complexités suivantes :

- pour Achterbahn-80 : 2^{58} en temps ou 2^{48} en mémoire et 2^{40} en temps.
- pour Achterbahn-128 : 2^{107} en temps ou 2^{47} en mémoire et 2^{88} en temps.

Mais j'ai pu montrer qu'il était possible de faire mieux. Expliquons la technique pour Achterbahn-128. L'idée est qu'on n'a pas besoin de défaire tous les tours des registres dans l'attaque dans le milieu. Si on sépare la clé en deux parties composées des 40 premiers bits et des 88 derniers, on peut effectuer une recherche exhaustive pour la première partie et stocker dans une table les états des registres obtenus après avoir appliqué le processus d'initialisation pour chaque ensemble de 40 bits. Ensuite, si on fait une recherche exhaustive sur les 88 bits restants, et on fait tourner en sens inverse les registres depuis les états connus, on trouvera une coïncidence dans la table. Mais on n'a pas besoin d'effectuer cette recherche sur tous les 88 bits restants de clé. À la place, on l'effectue sur les 73 derniers bits (ce qui veut dire que 15 tours ne s'inversent pas). Une fois qu'on a fait cela, on veut que, pour tout i, les premiers $(L_i - 15)$ bits de l'état du registre i coïncident avec les derniers $(L_i - 15)$ bits des états des registres enregistrés dans la table. Autrement dit, pour le registre R_0, on trouvera une coïncidence sur 6 bits, et pour le registre R_{12}, sur 18 bits. On ne doit pas s'inquiéter des coïncidences venant de mauvaises valeurs des 73 bits, car le nombre de fausses alarmes possibles de ce type est :

$$2^{88-15} \times \frac{2^{40}}{2^{329-13\times 15}} = 2^{-21}.$$

En effet, $(329 - 13 \times 15)$ est le nombre de bits qu'on considère pour une coïncidence, 2^{40} est la taille de la table, et 2^{73} est le nombre de possibilités pour la recherche exhaustive. Comme on peut le vérifier, avec une telle coïncidence, on a trouvé 113 bits de la clé. Les 15 bits restants peuvent être trouvés avec une très petite complexité en faisant tourner les registres jusqu'à ce qu'on trouve l'état désiré. La complexité finale pour l'étape de recherche de la clé pour Achterbahn-128 une fois qu'on a les états initiaux pour tous les registres est de 2^{73} en temps et $2^{40} \times (329 - 13 \times 15) \simeq 2^{48}$ en mémoire. Si on fait la même chose avec Achterbahn-80, on obtient une complexité de 2^{32} en temps et 2^{37} en mémoire, ou bien, avec un autre compromis temps mémoire, une complexité en temps de 2^{39} et 2^{30}.

Conclusion.

J'ai proposé [NP07a] une attaque contre Achterbahn-80 qui nécessite 2^{55} opérations. Donc on peut considérer comme complexité totale la complexité en données, qui est 2^{61}, car elle est plus grande. Mon attaque contre Achterbahn-128 a, elle, une complexité en temps de $2^{80.58}$ et moins de 2^{61} bits de la suite chiffrante doivent être générés. Après cela, on peut retrouver la clé de Achterbahn-80 avec une complexité de 2^{40} en temps et 2^{41} en mémoire (la complexité en temps est plus petite que pour la partie distingueur de l'attaque). Pour Achterbahn-128 on peut retrouver la clé avec une complexité de 2^{73} en temps et 2^{48} en mémoire.

2.4.5 Amélioration de l'attaque avec contrainte sur la longueur maximale de suite chiffrante

Longueur maximale de suite chiffrante.

Dans la première version d'Achterbahn-128/80, la longueur maximale de suite chiffrante produite à partir des mêmes valeurs de K et IV était limitée à 2^{63}. Suite à mes attaques et à celles de Hell et Johansson sur cette version [HJ07, NP07a], les auteurs ont proposé une nouvelle limitation de la longueur de suite chiffrante [GGK07]. Cette limite est devenue 2^{52} pour Achterbahn-80 et 2^{56} pour Achterbahn-128. Je présente maintenant une attaque contre ces deux générateurs, basée sur les attaques précédentes et qui prend en compte ces nouvelles modifications. L'attaque contre la variante de 80 bits, Achterbahn-80, a une complexité en temps de $2^{64.85}$ et nécessite moins de 2^{52} bits de suite chiffrante. L'attaque contre Achterbahn-128 nécessite 2^{104} opérations et moins de 2^{56} bits de suite chiffrante.

Attaque par distingueur contre Achterbahn-80.

Je vais maintenant décrire une nouvelle attaque contre Achterbahn-80 [NP08] que j'ai d'abord présentée à la « rump session » de SASC 2007. Cette attaque a une complexité de $2^{64.85}$ et elle utilise une approximation linéaire de la fonction de combinaison. Il s'agit d'une attaque par distingueur, qui consiste à distinguer la suite chiffrante d'une suite aléatoire. Toutefois nous retrouvons ainsi les états initiaux de certains registres et donc nous pouvons ensuite retrouver la clé secrète entière, comme nous l'expliquons plus loin. Cette attaque ressemble beaucoup à l'attaque précédente, mais maintenant on utilise une idée qui nous permet de réduire la longueur de suite chiffrante nécessaire. L'attaque utilise toujours l'approximation linéaire suivante de la fonction de combinaison G :

$$\ell(x_1, \ldots, x_{11}) = x_1 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_{10}.$$

La suite chiffrante $\boldsymbol{S}$ vérifie

$$\Pr[S(t) = \ell(t)] = \frac{1}{2}(1 - 2^{-3}).$$

Si on construit la relation de parité suivante pour $\mathcal{T} = \langle T_3T_7, T_4T_5 \rangle$,

$$PC_{\ell,\mathcal{T}}(t) = \ell(t) \oplus \ell(t + T_3T_7) \oplus \ell(t + T_4T_5) \oplus \ell(t + T_3T_7 + T_4T_5), \tag{2.5}$$

les termes contenant les suites $\boldsymbol{x_3}$, $\boldsymbol{x_4}$, $\boldsymbol{x_5}$, $\boldsymbol{x_7}$ disparaîtront de $\boldsymbol{PC_{\ell,\mathcal{T}}}$, et donc $\boldsymbol{PC_{\ell,\mathcal{T}}}$ dépend uniquement des suites $\boldsymbol{x_1}$, $\boldsymbol{x_6}$ et $\boldsymbol{x_{10}}$. La période T_4T_5 est 2^{51} et la période T_3T_7 est plus petite que 2^{49} car T_3 et T_7 ont des facteurs communs. Alors pour construire ces relations de parité on a besoin de N bits où N est inférieur à la limite imposée. Sommer quatre fois l'approximation comme en (2.5) a pour effet d'élever le biais à une puissance supérieure ou égale à 4, donc le biais de

$$PC(t) = PC_{G,\mathcal{T}}(t) = S(t) \oplus S(t + T_7T_3) \oplus S(t + T_4T_5) \oplus S(t + T_7T_3 + T_4T_5)$$

est au moins $2^{-4\times 3}$. On va décimer $\boldsymbol{PC}$ par la période du registre R_1 intervenant dans la relation de parité, ce qui signifie qu'on construit une nouvelle relation de parité :

$$PC'(t) = \sigma(t(2^{22}-1)).$$

Maintenant, si on faisait comme dans l'attaque précédente et qu'on effectuait une recherche exhaustive sur l'état initial des registres R_6 et R_{10}, on aurait besoin de

$$2^{3\times 4\times 2} \times 2 \times (58-2) \times \ln(2) = 2^{30.29} \text{ évaluations}$$

de la relation de parité $PC'(t)$ pour détecter le biais. Comme on va décimer par la période du premier registre, on aurait besoin de $2^{30.29} \times 2^{22} = 2^{52.29}$ bits de suite chiffrante pour réaliser l'attaque. Dans ce cas, on dépasserait la limite 2^{52} ce qui n'est pas possible.

Nous décrivons maintenant la principale différence avec l'attaque précédente [NP07a]. Au lieu de prendre seulement le premier bit de la suite chiffrante et de décimer par la période du premier registre R_1 $2^{30.29}$ fois, on va prendre les quatre premiers bits de la suite chiffrante et, pour chacun d'entre eux, on va obtenir une suite de $\frac{2^{30.29}}{4} = 2^{28.29}$ bits en décimant par la période du premier registre $2^{28.29}$ fois. Donc, on considère les premiers $2^{50.29}$ bits de la suite chiffrante et on calcule les $4 \times 2^{28.29} = 2^{30.29}$ relations de parité :

$$\begin{aligned} PC(tT_1+i) &= S(tT_1+i) \oplus S(tT_1+i+T_7T_3) \oplus S(tT_1+i+T_4T_5) \\ &\quad \oplus S(tT_1+i+T_7T_3+T_4T_5) \end{aligned}$$

pour $i \in \{0,\ldots,3\}$ et $0 \leq t < 2^{28.29}$.

De cette façon, le nombre de bits de suite chiffrante dont on a besoin va être réduit à $2^{28.29} \times 2^{22} = 2^{50.29}$ et respecte la longueur maximale de suite chiffrante permise. Maintenant, on effectue une recherche exhaustive sur l'état initial des registres R_6 et R_{10}, en adaptant à notre nouvelle attaque l'algorithme introduit à la section précédente. On doit calculer, pour chacune des suites définies précédemment, donc pour chaque $i \in \{0,1,2,3\}$, la somme suivante :

$$S_i = \sum_{t'=0}^{2^{28.29}-1} PC(t'T_1+i) \oplus PC_{\ell,\mathcal{T}}(t'T_1+i).$$

Soit

$$T' = 2^{28.29} - 2 \times T_6 = 2^{28.29} - 2^{28} + 2 = 2^{25.83}.$$

On note $PC(t)$, $\sigma_6(t)$ et $\sigma_{10}(t)$ les relations de parité calculées à l'instant t avec la suite chiffrante, la suite générée par le registre R_6 et celle générée par le registre R_{10}

respectivement. Alors on peut décomposer la somme S_i de la façon suivante :

$$\begin{aligned}
S_i &= \sum_{k=0}^{T'}\sum_{t=0}^{2} PC((T_6t+k)T_1+i) \oplus PC_{\ell,\mathcal{T}}((T_6t+k)T_1+i) \\
&+ \sum_{k=T'+1}^{T_6-1}\sum_{t=0}^{1} PC((T_6t+k)T_1+i) \oplus PC_{\ell,\mathcal{T}}((T_6t+k)T_1+i) \\
&= \sum_{k=0}^{T'}\sum_{t=0}^{2} PC((T_6t+k)T_1+i) \oplus \sigma_{10}((T_6t+k)T_1+i) \oplus \sigma_6((T_6t+k)T_1+i) \\
&+ \sum_{k=T'+1}^{T_6-1}\sum_{t=0}^{1} PC((T_6t+k)T_1+i) \oplus \sigma_{10}((T_6t+k)T_1+i) \oplus \sigma_6((T_6t+k)T_1+i) \\
&= \sum_{k=0}^{T'} \left[(\sigma_6(kT_1+i)\oplus 1) \left(\sum_{t=0}^{2} PC((T_6t+k)T_1+i) \oplus \sigma_{10}((T_6t+k)T_1+i) \right) \right. \\
&+ \left. \sigma_6(kT_1+i) \left(3 - \sum_{t=0}^{2} PC((T_6t+k)T_1+i) \oplus \sigma_{10}((T_6t+k)T_1+i) \right) \right] \\
&+ \sum_{k=T'}^{T_6-1} \left[(\sigma_6(kT_1+i)\oplus 1) \left(\sum_{t=0}^{1} PC((T_6t+k)T_1+i) \oplus \sigma_{10}((T_6t+k)T_1+i) \right) \right. \\
&+ \left. \sigma_6(kT_1+i) \left(2 - \sum_{t=0}^{1} PC((T_6t+k)T_1+i) \oplus \sigma_{10}((T_6t+k)T_1+i) \right) \right].
\end{aligned}$$

La somme peut être écrite de la manière précédente parce que $\sigma_6((T_6t+k)T_1+i)$ est constant pour des valeurs fixées de k et i. On peut alors procéder de la façon suivante :

- On choisit un état initial pour le registre R_6, par exemple l'état avec toutes les cellules à 1. On calcule et on stocke un vecteur binaire V_6 de longueur T_6, $V_6[k] = \sigma_6(k)$ où la suite avec laquelle on calcule $\sigma_6(k)$ est engendrée à partir de l'état initial choisi. La complexité de cette étape est de $T_6 \times 2^2$ opérations.
- Pour chaque état possible du registre R_{10} (donc 2^{31-1} possibilités) :
  - on calcule et stocke quatre vecteurs $V_{10,i}$, où $i \in \{0,1,2,3\}$, chacun composé de T_6 entiers de 2 bits.

$$V_{10,i}[k] = \sum_{t=0}^{m} PC((T_6t+k)T_1+i) \oplus \sigma_{10}((T_6t+k)T_1+i),$$

où $m = 2$ si $k \leq T'$ et $m = 1$ si $k > T'$. La complexité de cette étape est :

$$2^2\left(3 \times 2^{25.83} + 2(2^{27} - 1 - 2^{25.83})\right)(2^3+2) \;=\; 2^2 2^{28.29} 2^{3.3} = 2^{33.69}$$

pour chaque état initial possible du registre R_{10}, où 2^2 est le nombre de vecteurs qu'on calcule, 2^3 correspond au nombre d'opérations dont on a besoin pour calculer chaque $(PC(t) + \sigma_{10}(t))$ et $2^{28.29} \times 2$ est le coût de sommer $2^{28.29}$ entiers de 2 bits.

- Pour chaque p de 0 à $T_6 - 1$:
  - on définit $V_{6,i}$ de longueur T_6, pour tout $i \in \{0,1,2,3\}$:

$$V_{6,i}[k] = V_6[k + p + i \mod T_6].$$

    En fait, $(V_{6,i}[k])_{k<T_6}$ correspond à $(\sigma_6(k))_{k<T_6}$ quand l'état initial de registre R_6 correspond à l'état interne qu'on obtient après avoir fait tourner le registre R_6 $(i+p)$ fois à partir de l'état avec toutes les cellules à un.
  - Avec les huit vecteurs que l'on vient d'obtenir $(V_{10,0}, \ldots, V_{10,3})$ et $(V_{6,0}, \ldots, V_{6,3})$, on calcule pour chaque $i \in \{0,1,2,3\}$:

$$\begin{aligned} W_i \;=\; & \sum_{k=0}^{T'} \left[(V_{6,i}[k] \oplus 1)\, V_{10,i}[k] + V_{6,i}[k]\,(3 - V_{10,i}[k]) \right] + \\ & \sum_{k=T'+1}^{T_6-1} \left[(V_{6,i}[k] \oplus 1)\, V_{10,i}[k] + V_{6,i}[k]\,(2 - V_{10,i}[k]) \right]. \end{aligned}$$

    Lorsque l'on fait cela avec les bons états initiaux des registres R_6 et R_{10}, on retrouvera un biais important pour les quatre W_i.

  La complexité de cette étape sera $2^2 \times T_6 \times 8 = 2^{32}$ pour chaque p, donc $2^{32} \times 2^{27} = 2^{59}$. Mais on peut l'accélérer de la manière suivante :
  on définit un vecteur,

$$V'_{10,j}[k] = V_{10,j}[k] + ct$$

  où $ct = 0$ si $k \leq T'$ et $ct = 0.5$ si $k > T'$.
  Pour chaque i on va calculer :

$$\sum_{k'=0}^{T_6-1} (-1)^{V_{6,i}[k+p]} \left(V'_{10,i}[k] - \frac{3}{2} \right) + (T' \times 1.5 + (T_6 - T') \times 1).$$

  Le but est maintenant de trouver l'entier p qui maximise cette somme, cela revient à calculer le maximum de la corrélation mutuelle de deux suites de longueur T_6. On peut faire cela d'une façon efficace avec une transformée de Fourier rapide. La complexité finale pour calculer cette somme sera $T_6 \log_2(T_6)$.
  Donc, la complexité totale de cette étape est $4T_6 \log_2(T_6) \approx 2^{34}$.

Maintenant je vais expliquer la raison du choix de la valeur 4 pour le nombre i_{max} de suites $(PC(tT_1 + i))_{t \geq 0}$. Sur la figure 2.2 nous pouvons voir les probabilités de fausse alarme et de non-détection pour des différentes valeurs de i_{max}. Dans tous les cas, on considère $n = \frac{2^{30.29}}{i_{max}}$. On n'a pas essayé de valeurs plus grandes que $2^{30.29}$ qui auraient pû nous mener à des valeurs plus grands ede i_{max} pour deux raisons très simples :

- ceci augmenterait la complexité en temps et, de plus,
- à cause de la relation de parité, la complexité en données ne va pas descendre en dessus de 2^{51}.

Nous avons choisit $i_{max} = 4$ parce que cette valeur nous donnait à peu près la même probabilité de fausse alarme et de non-détection, et parce que avec $i_{max} = 4$ on descend déjà sous la limite demandée. On a une bonne marge avec cette valeur, on aurait pu prendre des valeurs plus grandes de i_{max}, (voire $i_{max} = 8$) qui nous donnent aussi des valeurs acceptables pour ces deux probabilités, mais, comme nous venons de le dire, à cause de la relation de parité, la complexité en données n'aurait pas été améliorée.

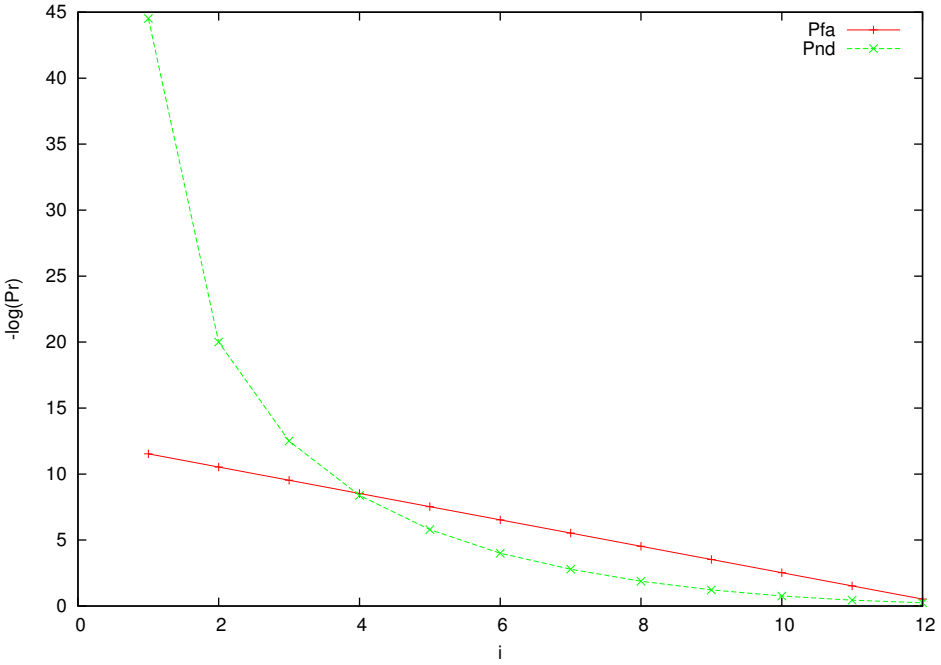

FIG. 2.2 – Relation entre les $P_{\text{fa},i_{max}}$ et $P_{\text{nd},i_{max}}$ pour des i_{max} différents.

Ensuite on va calculer en détail les probabilités de fausse alarme et de non-détection pour notre attaque. D'abord on va considérer comme suit le seuil θ pour détecter le biais ε :

$$\theta = 0.55 \times \varepsilon = 2^{-12.86}.$$

Soit n la longueur des suites qu'on utilise et i_{max} le nombre de suites. La probabilité de fausse alarme pour i_{max} suites est la probabilité que, pendant qu'on essaie les mauvais états initiaux des registres R_6 et R_{10} (ce qui devrait engendrer une suite aléatoire) on trouve un biais plus grand que $2^{-12.86}$ ou plus petit que $-2^{-12.86}$ pour tous les i_{max} $W_0, \ldots, W_{i_{max}-1}$. En utilisant la borne de Chernoff sur la queue de la distribution binomiale on obtient :

$$P_{\text{fa},i_{max}}(\theta) = (P_{\text{fa},1}(\theta))^{i_{max}} \leq (2e^{-2\theta^2 n})^{i_{max}},$$

où $P_{\text{fa},1}$ est la probabilité de fausse alarme pour une suite. Dans notre cas $n = 2^{28.29}$ et $i_{max} = 4$, donc $P_{\text{fa},4} = 2^{-64.5}$. Le nombre d'états initiaux qui vont passer le test sans être le bon est $(2^{56} - 1) \times 2^{-64.5} = 2^{-8.5}$. La probabilité de non-détection est la probabilité que,

alors qu'on essaie les bons états initiaux des registres R_6 et R_{10}, on trouve un biais entre $-2^{-12.86}$ et $2^{-12.86}$. Pour une seule suite, la probabilité de non-détection vérifie :

$$P_{\mathrm{nd},1}(\theta) \leq 2e^{-2(\varepsilon-\theta)^2 n} = 2^{-10.38},$$

Donc la probabilité de non détection pour i_{max} suites, c'est-à-dire la probabilité de pas dépasser le seuil pour une des i_{max} $W_0, \ldots, W_{i_{max}-1}$ sera de :

$$P_{\mathrm{nd},i_{max}}(\theta) = 1 - (1 - P_{\mathrm{nd},1}(\theta))^{i_{max}}.$$

On l'utilise seulement pour les bons états initiaux. Cette probabilité augmente avec le nombre i_{max} de suites utilisées. Dans notre cas, $i_{max} = 4$, la probabilité de ne pas dépasser le seuil pour l'une des quatre suites est : $P_{\mathrm{nd},4}(\theta) = 2^{-8.38}$.
La complexité en temps est finalement

$$2^{L_{10}-1} \times \left[2^{33.69} + 4T_6 \log_2 T_6\right] + T_6 \times 2^2 = 2^{64.85}.$$

La longueur de suite chiffrante nécessaire est

$$2^{28.29} \times T_1 + T_3T_7 + T_4T_5 = 2^{50.29} + 2^{48.1} + 2^{51} < 2^{52}.$$

Attaque par distingueur contre Achterbahn-128.

Maintenant, nous présentons une attaque par distingueur de la version 128 bits de Achterbahn avec laquelle on retrouve aussi les états initiaux de deux registres [NP08].

On prend comme précédemment l'approximation linéaire suivante de la fonction de combinaison F :

$$\ell(x_0, \ldots, x_{12}) = x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_7 \oplus x_8 \oplus x_9 \oplus x_{10}.$$

Son biais est donné par $\Pr[S(t) = \ell(t)] = \frac{1}{2}(1 + 2^{-3})$.

Soit $\mathcal{T} = \langle T_3T_8, T_1T_{10}, T_2T_9 \rangle$. Si on construit une relation de parité comme suit :

$$PC_{\ell,\mathcal{T}}(t) = \sum_{\tau \in \langle T_{3,8}, T_{1,10}, T_{2,9} \rangle} \ell(t+\tau),$$

les termes contenant les suites $\boldsymbol{x_1}$, $\boldsymbol{x_2}$, $\boldsymbol{x_3}$, $\boldsymbol{x_8}$, $\boldsymbol{x_9}$, $\boldsymbol{x_{10}}$ disparaîtront de $\boldsymbol{PC_{\ell,\mathcal{T}}}$, alors $\boldsymbol{PC_{\ell,\mathcal{T}}}$ est une suite qui dépend uniquement des suites $\boldsymbol{x_0}$, $\boldsymbol{x_4}$ et $\boldsymbol{x_7}$. On note alors :

$$PC_{\ell,\mathcal{T}}(t) = \bigoplus_{\tau \in \langle T_{3,8}, T_{1,10}, T_{2,9} \rangle} x_0(t+\tau) \oplus x_4(t+\tau) \oplus x_7(t+\tau) = \sigma_0(t) \oplus \sigma_4(t) \oplus \sigma_7(t), \quad (2.6)$$

où $\sigma_0(t)$, $\sigma_4(t)$ et $\sigma_7(t)$ sont les relations de parité calculées avec les suites engendrées par les NLFSRs R_0, R_4 et R_7. Sommer huit fois l'approximation comme en (2.6) a pour effet d'élever le biais à la puissance 8, donc le biais de

$$PC(t) = PC_{F,\mathcal{T}}(t) = \bigoplus_{\tau \in \langle T_{3,8}, T_{1,10}, T_{2,9} \rangle} S(t+\tau),$$

où $\boldsymbol{S}$ est la suite chiffrante, est $2^{-8\times 3}$. Alors :

$$\Pr[PC(t) \oplus \sigma_0(t) \oplus \sigma_4(t) \oplus \sigma_7(t) = 0] = \frac{1}{2}(1 + \varepsilon^8).$$

Ceci implique qu'on a besoin de $2^{3\times 8\times 2} \times 2 \times (74-3) \times \ln(2) = 2^{54.63}$ évaluations de $PC(t) \oplus \sigma_0(t) \oplus \sigma_4(t) \oplus \sigma_7(t)$ pour détecter le biais, quand on effectue une recherche exhaustive sur les registres R_0, R_4 et R_7.

On va utiliser l'algorithme proposé précédemment pour l'attaque de Achterbahn-128 pour calculer la somme $PC(t) \oplus \sigma_0(t) \oplus \sigma_4(t) \oplus \sigma_7(t)$ pour toutes les valeurs de t. Cet algorithme a une complexité plus petite qu'une recherche exhaustive simultanée sur les états initiaux des registres R_0, R_4 et R_7 en même temps. On va l'utiliser en considérant ensemble le registre R_0 et le registre R_4.
La complexité va finalement être

$$2^{L_0-1} \times 2^{L_4-1} \times \left[2^{54.63} \times \left(2^4 + 2^{4.7}\right) + O(T_7 \log T_7)\right] + T_7 \times 2^3 = 2^{104}.$$

La longueur de suite chiffrante nécessaire est

$$2^{54.63} + T_{1,10} + T_{2,9} + T_{3,8} = 2^{54.63} + 2^{53} + 2^{53} + 2^{53} < 2^{56} \text{ bits.}$$

Retrouver la clé.

Comme on l'a expliqué dans les attaques précédentes, avec une variante d'une attaque dans le milieu on peut récupérer la clé une fois qu'on a retrouvé l'état initial de quelques registres, et avec une complexité plus petite que celle de l'attaque par distingueur qu'on vient de décrire. Donc la complexité totale de l'attaque qui retrouve la clé sera la même que celle du distingueur.

Conclusion.

On a proposé une attaque contre Achterbahn-80 en $2^{64.85}$ qui nécessite moins de 2^{52} bits de suite chiffrante. On a aussi proposé une attaque contre Achterbahn-128 en 2^{104} qui nécessite moins de 2^{56} bits de suite chiffrante. Ensuite on peut récupérer la clé de Achterbahn-80 avec une complexité de 2^{40} en temps et 2^{41} en mémoire (la complexité en temps est plus petite que celle de la partie distingueur). Pour Achterbahn-128 on peut retrouver la clé avec une complexité de 2^{73} en temps et 2^{48} en mémoire.

Les complexités des meilleures attaques qui retrouvent la clé, publiées contre toutes les versions d'Achterbahn sont résumées dans la table suivante, où nous pouvons voir mes attaques en bleu :

Suite à mes attaques, une nouvelle longueur maximale de suite chiffrante de 2^{44} a été proposée. Sur celle-ci, il n'y a pas d'attaques connues.

Avec un point de vue plus général, ces attaques ont apporté de nouveaux outils à utiliser dans la cryptanalyse par corrélation des chiffrements à flot. On verra dans la section suivante que dans le cas des relations de parité construites avec $t+1$ variables, quand la

Achterbahn	temps	données	mémoire	attaque
v1	2^{61}	2^{32}	2^{30}	[JMM06]
v2	$2^{64}/2^{51}$	2^{59}	2^{30}	[HJ06]
80	$2^{72.90}$	$2^{58.24}$	2^{30}	[HJ07]
128	$2^{96.52}$	$2^{63.81}$	2^{48}	[HJ07]
v2	$2^{53}/2^{32}$	2^{53}	2^{37}	Section 2.3.3
80	2^{55}	2^{61}	2^{30}	Section 2.4.2/4
128	$2^{80.58}$	2^{61}	2^{48}	Section 2.4.3/4
80-limit.	$2^{64.85}$	$<2^{52}$	2^{30}	Section 2.4.5
128-limit.	2^{104}	$<2^{56}$	2^{48}	Section 2.4.5

TAB. 2.1 – Complexités des attaques pour retrouver la clé, publiées sur les différentes versions d'Achterbahn

fonction de combinaison est t-résiliente, il est toujours mieux d'utiliser des approximations linéaires. On a également proposé une méthode pour réduire la longueur de suite chiffrante dont on a besoin, en utilisant des bits « gaspillés » par la décimation. On a aussi introduit un algorithme qui permet d'accélérer la recherche exhaustive dans certains cas, où les registres peuvent être traités séparément. Enfin, pendant la rédaction de cette thèse j'ai remarqué que l'algorithme proposé indépendemment par C. Berbain dans [Ber07, pages 26-27] dans un autre contexte (des registres à décalage linéaire) était comparable à celui que nous avons présenté pour diminuer la complexité de l'attaque.

Chapitre 3

Relations de parité

Reprenons les attaques de la section précédente. Ce sont des attaques par corrélation sur un algorithme de chiffrement à flot basé sur des FSRs combinés par une fonction booléenne, qui prend en entrée les sorties des registres et sort un seul bit à chaque instant. Ceci est un modèle très commun pour construire des chiffrements à flot. Le résultat de cette construction peut être vu comme une suite produite par une fonction dont les entrées dépendent du temps. Les variables d'entrée (les sorties des FSRs donc) vont avoir une période déterminée par la fonction de rétroaction du registre. Les attaques par corrélation, comme celles du chapitre précédent, utilisent une approximation de la fonction de combinaison par une fonction de moins de variables. Si les entrées des fonctions sont uniformément distribuées, comme on peut l'attendre d'un FSR bien construit, on va trouver entre les deux suites, la suite chiffrante et son approximation, le même biais que celui entre les sorties des deux fonctions. La sortie de la fonction de combinaison est connue, puisqu'elle correspond à la suite chiffrante. La question est : comment calculer la suite résultant de la combinaison des FSRs du générateur par l'approximation ? On pourrait, par exemple, faire une recherche exhaustive sur les états initiaux de tous les registres reliés aux variables qui interviennent dans l'approximation, jusqu'à trouver les états qui nous donnent le biais attendu entre la sortie de l'approximation et la suite chiffrante. Cette méthode, proposée à l'origine par Siegenthaler [Sie85], est en pratique beaucoup trop coûteuse pour être utilisée toute seule sur des systèmes bien construits. Ceci est la motivation pour construire des relations de parité : faire disparaître m termes dans l'approximation, pour pouvoir calculer les sorties de la nouvelle approximation d'une façon moins coûteuse car avec les m termes on fait disparaître n variables. Donc, typiquement, dans les attaques par corrélation, les relations de parité sont calculées à partir d'une approximation g de la fonction de combinaison f, et le biais de la relation de parité, $\mathcal{E}(PC_{f,\mathcal{T}})$, est estimé à partir du biais de l'approximation, $\mathcal{E}(f \oplus g)$. Pour faire disparaître les m termes, on peut sommer l'approximation aux 2^m instants définis par leurs périodes, c'est-à-dire, dans l'ensemble$\langle T_1, ..., T_m \rangle$, ensemble des 2^m combinaisons à coefficients $\{0, 1\}$ des entiers $T_1, \ldots, T_m$, où T_i est une période du registre R_i. On obtient pour la somme de 2^m termes faisant intervenir l'approximation g de la fonction f, un biais égal au biais de l'approximation $\mathcal{E}(f \oplus g)$ élevé à la puissance le nombre des termes de la somme 2^m, mais uniquement dans le cas où les termes de la

somme à chaque instant sont tous indépendants. Sinon, cette valeur n'est qu'une borne inférieure :

$$\mathcal{E}(PC_{f,\mathcal{T}}) \geq \mathcal{E}(f \oplus g)^{2^m}$$

Dans ce chapitre on va présenter une nouvelle façon de construire et définir des relations de parité ainsi que des méthodes pour calculer leur biais exact, beaucoup plus rapides que les méthodes connues précédemment. Ce travail, effectué avec Anne Canteaut, a été en partie publié dans [CNP09a].

3.1 Nouvelle vision des relations de parité

Traditionnellement, les relations de parité utilisées dans les attaques par corrélation contre des chiffrements à flot à base de FSRs combinés sont construites à partir d'une approximation déterminée de la fonction de combinaison f. Ceci est logique, vu l'objectif de réduction du nombre de FSRs impliqués, comme nous venons de l'expliquer.

Soit g une approximation de f (on considère que les variables de g sont un sous-ensemble de l'ensemble des variables de f). Soit $\mathcal{T} = \langle M_1, \ldots, M_m \rangle$ et $J \subset \{j_1, \ldots, j_n\}$ un sous-ensemble des variables de f dont la période divise un des $M_i, 1 \leq i \leq m$. Parfois, dans la suite, notamment dans l'énoncé des résultats, on supposera que $J = \{1, \ldots, n\}$ et que M_i est un multiple des périodes des variables d'indices $\{\ell_i + 1, \ldots, \ell_{i+1}\}$ avec $\ell_1 = 0$ et $\ell_{m+1} = n$. On décomposera alors les vecteurs de $\mathbf{F}_2^v$ en entrée de f en (x, y) avec $x \in \mathbf{F}_2^n$ et $y \in \mathbf{2}^{\mathbf{v-n}}$. Dans l'attaque, on décompose l'approximation g sous la forme

$$g(x_1, \ldots, x_v) \quad = \quad g'(x_{j_1}, \ldots, x_{j_n}) \oplus \tilde{g}(x_1, \ldots, x_v)$$

où g' ne contient que des variables de J. Ainsi, quand on va construire la relation de parité $PC_{g,\mathcal{T}}$, tous les termes de g' vont disparaître. Notons $h = f \oplus g$, et ε son biais, c'est-à-dire

$$\varepsilon = \mathcal{E}(h) = \mathcal{E}(f \oplus g).$$

Pour

$$f'(x_1, \ldots, x_v) = \left(f(x_1, \ldots, x_v) \oplus \tilde{g}(x_1, \ldots, x_v)\right),$$

on a

$$h(x_1, \ldots, x_v) = f'(x_1, \ldots, x_v) \oplus g'(x_{j_1}, \ldots, x_{j_n}).$$

Alors, pour l'ensemble $\mathcal{T}$, la même relation de parité est obtenue quand on la calcule avec h ou avec f' :

$$PC_{f',\mathcal{T}}(t) = \bigoplus_{\tau \in \mathcal{T}} h(t + \tau) = \bigoplus_{\tau \in \mathcal{T}} f'(t + \tau).$$

et cette relation aura un biais, comme on l'a déjà vu, borné par

$$\mathcal{E}(PC_{h,\mathcal{T}}) \geq \varepsilon^{2^m}.$$

Dans les attaques, on exploite donc le biais de la relation $PC_{f',\mathcal{T}}(t) = PC_{f,\mathcal{T}}(t) \oplus PC_{\tilde{g},\mathcal{T}}(t)$. La valeur du premier terme est donné par la suite chiffrante tandis que le deuxième, qui contient souvent peu de variables, est soit déterminé par recherche exhaustive sur les états des registres impliqués, soit éliminé par décimation... Ici, nous supposerons que l'attaquant a directement accès aux valeurs de la suite $PC_{f',\mathcal{T}}$

L'ensemble $\mathcal{T}$ des instants sur lesquels on somme la suite peut être défini de plusieurs façons : par les périodes des m termes qui disparaissent dans la relation de parité ou bien par un nombre plus petit de périodes mais incluant toutes les périodes des termes considérés, le cas extrême étant $\mathcal{T} = \langle T_{j_1} \cdots T_{j_n} \rangle$, avec $m = 1$. Ainsi, à une même approximation, il peut naturellement correspondre plusieurs relations de parité.

Exemple 3.1. On va illustrer ceci avec un exemple, où on utilise la fonction de combinaison de Achterbahn-80, G. Si on considère l'approximation suivante :

$$g(x_1, x_2, x_3, x_4, x_7, x_9, x_{10}) = x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_7 \oplus x_9 \oplus x_{10}$$

de biais $\varepsilon = 2^{-3}$, et qu'on construit la décomposition

$$\begin{aligned} h_1(x_1, \ldots, x_{11}) &= f'_1(x_1, \ldots, x_{11}) \oplus g'_1(x_1, \ldots, x_{11}) \\ \text{avec } f'_1(x_1, \ldots, x_{11}) &= G(x_1, \ldots, x_{11}) \oplus x_1 \oplus x_2 \oplus x_7 \\ \text{et } g'_1(x_{j_1}, \ldots, x_{j_s}) &= x_3 \oplus x_4 \oplus x_9 \oplus x_{10}. \end{aligned}$$

On peut alors construire
- avec $\mathcal{T}_1 = \langle T_3T_{10}, T_4T_9 \rangle$

$$PC_1(t) = PC_{h_1,\mathcal{T}_1}(t) = f'_1(t) \oplus f'_1(t + T_3T_{10}) \oplus f'_1(t + T_4T_9) \oplus f'_1(t + T_3T_{10} + T_4T_9),$$

- avec $\mathcal{T}_2 = \langle T_3, T_4T_9T_{10} \rangle$

$$PC_2(t) = PC_{h_1,\mathcal{T}_2}(t) = f'_1(t) \oplus f'_1(t + T_3) \oplus f'_1(t + T_4T_9T_{10}) \oplus f'_1(t + T_3 + T_4T_9T_{10}),$$

toutes ces relations ayant un biais $\varepsilon \geq 2^{-12}$. Les deux relations de parité construites sont différentes, puisqu'on somme f'_1 à des instants différents.

Maintenant, pour plus de simplicité et également de nombreuses autres raisons, on va définir un scénario où on peut parler des relations de parité construites à partir d'une fonction, sans avoir besoin d'avoir d'abord défini une approximation. Autrement dit, on va introduire une nouvelle vision des relations de parité où on va considérer d'emblée la séparation qu'on vient de voir et où on ne va travailler qu'avec f'. Une fois seulement qu'on aura construit la relation de parité, on pourra lui associer des approximations de f qui auraient pu être utilisées pour la construire.

À part la simplicité qu'on va apprécier plus tard, un des intérêts principaux de cette vision est qu'elle montre clairement qu'à une seule relation de parité on peut associer plusieurs approximations. De cette façon, on peut donc rechercher la meilleure approximation pour calculer une borne sur le biais, par exemple.

Typiquement, dans les attaques du chapitre précédent, on définissait une borne sur le biais de la relation de parité à partir de l'approximation qu'on avait utilisée, mais cette approximation n'était pas toujours celle, parmi toutes les approximations qu'on pouvait associer à cette relation de parité, qui avait le meilleur biais.

Exemple 3.2. Reprenons la fonction utilisée dans Achterbahn-80, G, et considérons ses approximations g_1 et g_2 :

- $g_1(x_1, x_2, x_3, x_4, x_7, x_9, x_{10}) = x_1 \oplus x_2 \oplus x_7 \oplus x_3x_{10} \oplus x_4x_9$
  ce qui, en regroupant $f'(x_1, \ldots, x_{11}) = G(x_1, \ldots, x_{11}) \oplus x_1 \oplus x_2 \oplus x_7$, définit

$$h_1(x_1, \ldots, x_{11}) = f'(x_1, \ldots, x_{11}) \oplus x_3x_{10} \oplus x_4x_9, \text{ avec } \varepsilon_1 = \mathcal{E}(h_1) = 2^{-5};$$

- $g_2(x_1, x_2, x_3, x_4, x_7, x_9, x_{10}) = x_1 \oplus x_2 \oplus x_7 \oplus x_3 \oplus x_{10} \oplus x_4 \oplus x_9$
  ce qui, en regroupant $f'(x_1, \ldots, x_{11}) = G(x_1, \ldots, x_{11}) \oplus x_1 \oplus x_2 \oplus x_7$, définit

$$h_2(x_1, \ldots, x_{11}) = f'(x_1, \ldots, x_{11}) \oplus x_3 \oplus x_{10} \oplus x_4 \oplus x_9, \text{ avec } \varepsilon_2 = \mathcal{E}(h_2) = 2^{-3}.$$

Si on construit la relation de parité à quatre termes définie par $\mathcal{T} = \langle T_3T_{10}, T_4T_9 \rangle$

$$\begin{aligned} PC_{f',\mathcal{T}}(t) &= h_1(t) \oplus h_1(t + T_3T_{10}) \oplus h_1(t + T_4T_9) \oplus h_1(t + T_3T_{10} + T_4T_9) \\ &= h_2(t) \oplus h_2(t + T_3T_{10}) \oplus h_2(t + T_4T_9) \oplus h_2(t + T_3T_{10} + T_4T_9) \\ &= f'(t) \oplus f'(t + T_3T_{10}) \oplus f'(t + T_4T_9) \oplus f'(t + T_3T_{10} + T_4T_9), \end{aligned}$$

on obtient la même relation, puisque les f' correspondant sont les mêmes. Par contre, si on calcule les bornes sur le biais comme cela a été fait jusqu'à présent, on obtient $\mathcal{E}(PC_{f',\mathcal{T}}) = 2^{-20}$ si on a calculé la relation de parité à partir de l'approximation g_1 et $\mathcal{E}(PC_{f',\mathcal{T}}) = 2^{-12}$ si on l'a fait avec g_2, ce dernier étant son biais exact, comme nous avons pu le démontrer et comme nous le verrons plus loin.

Chaque relation de parité qu'on peut construire sur f peut donc être associée à une ou plusieurs de ses approximations.

Définition 3.3. Soit $\mathcal{T} = \langle M_1, \ldots, M_m \rangle$ l'ensemble d'instants utilisés pour construire une relation de parité $PC_{f,\mathcal{T}}$. Alors, l'ensemble des approximations associées à $\mathcal{T}$ est l'ensemble des fonctions booléennes g telles que, $\forall t \geq 0$, $PC_{g,\mathcal{T}}(t) = 0$. Parmi toutes ces approximations associées, g_τ est l'approximation à m termes construite avec les n variables qui interviennent dans la relation de parité, et telle que la période minimale du $i$ème terme de g_τ est M_i.

Dans l'exemple 3.1, g_τ de PC_1 est l'approximation quadratique suivante : $x_3x_{10} \oplus x_4x_9$. Une remarque importante est que toute relation de parité peut être construite à partir d'une approximation affine. On verra plus tard que, dans le cas des fonctions résilientes, ceci est très important puisque c'est le biais de l'approximation affine associée qui détermine en grande partie le biais de la relation de parité. La question de la détermination du biais exact de ces relations de parité sera étudiée dans la section suivante.

Le fait que plusieurs approximations de f peuvent conduire à la même relation de parité nous permet de donner les bornes inférieures suivantes sur le biais de $PC_{f,\mathcal{T}}$.

Théorème 3.4. *Soient $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_v$ v suites de périodes minimales $T_1, \ldots T_v$ et f une fonction booléenne à v variables. Soit*

$$\mathcal{T} = \{\sum_{i=1}^{n} c_i M_i, c_i \in \{0,1\}\}$$

où $M_i = q_i ppcm(T_{\ell_i+1}, \ldots, T_{\ell_{i+1}})$ avec $q_i > 0$, $\ell_1 = 0$ et $\ell_{n+1} = n$. On considère que $\mathcal{T}$ ne contient aucun multiple de T_j, pour tout $n < j \leq v$. Alors, pour toute fonction booléenne g de n variables de la forme

$$g(x_1, \ldots, x_n) = \sum_{i=1}^{n} g_i(x_{\ell_i+1}, \ldots, x_{\ell_{i+1}}) \tag{3.1}$$

où chaque g_i est une fonction booléenne de $(\ell_{i+1} - \ell_i)$ variables, nous avons

$$\mathcal{E}(PC_{f,\mathcal{T}}) \geq [\mathcal{E}(f \oplus g)]^{2^m}.$$

Le point important à remarquer ici est que $\mathcal{E}(f \oplus g)$ nous donne une borne inférieure sur le biais de la relation de parité pour n'importe quel choix de l'approximation g de la forme (3.1). L'approximation linéaire de f par la somme des n premières variables est souvent prise en compte, mais toute approximation qui utilise ces variables peut être choisie, comme nous l'énonçons dans le corollaire suivant. Pour la suite, pour tout $\alpha \in \mathbf{F}_2^v$, φ_α représente la fonction linéaire à v variables : $x \mapsto \alpha \cdot x$, où $x \cdot y$ est le produit scalaire habituel.

Corollaire 3.5. *Avec les notations du théorème 3.4, nous avons que*

$$\mathcal{E}(PC_{f,\mathcal{T}}) \geq \max_{\alpha \in \mathbf{F}_2^n} [\mathcal{E}(f \oplus \varphi_{\alpha,0})]^{2^m}.$$

Il est important de remarquer que ce corollaire nous donne une borne inférieure sur le biais de la relation de parité même si les fonctions f et $x \mapsto x_1 \oplus \ldots \oplus x_n$ ne sont pas corrélées. Ceci est le premier résultat connu dans ces conditions ; l'impossibilité de déduire une estimation du biais dans des cas identiques a été mentionnée dans [GG07].

Par exemple, prenons la fonction booléenne $f(x_1, x_2, x_3) = x_1x_2 \oplus x_2x_3 \oplus x_1x_3$ et l'approximation $g = x_1 \oplus x_2$.

On a $\mathcal{E}(f \oplus g) = 0$. Pourtant, on peut construire la relation de parité suivante

$$PC(t) = f(t) \oplus f(t + T_1) \oplus f(t + T_2) \oplus f(t + T_1T_2)$$

qui a un biais non nul :

$$\Pr[PC(t) = 0] = \frac{1}{2}(1 + 2^{-3}) \neq \frac{1}{2}.$$

Dans le même temps, d'autres approximations g avec un plus grand degré peuvent nous donner une meilleure borne. Mais, comme toute fonction booléenne est complètement déterminée par sa transformée de Walsh, *i.e.* par les biais de toutes ses approximations linéaires, nous avons trouvé que nous pouvons calculer $\mathcal{E}(PC_{f,\mathcal{T}})$ à partir des biais des approximations linéaires de f uniquement.

3.2 Comment calculer le biais des relations de parité efficacement

Si on construit une relation de parité en sommant 2^m fois une fonction f à v variables, où n parmi ces v variables sont impliquées dans la relation de parité, et qu'on veut calculer son biais exact, la méthode naïve de calcul du biais par recherche exhaustive aurait une complexité de 2^{v2^m}. Si on utilise le fait que chaque variable dans la relation de parité est utilisée deux fois, on peut faire une recherche exhaustive moins naïve avec complexité $2^{n2^{m-1}} \times 2^{2^m(v-n)}$. Cette complexité devient vite très grande avec le nombre de termes de la relation de parité, et le biais est généralement incalculable avec cette méthode. Cette dernière était cependant la meilleure connue avant nos résultats pour calculer le biais des relations de parité. Ceci était peut-être dû à la complexité du sujet et à la difficulté pour analyser et comprendre les relations de parité. Nous avons cherché et trouvé des façons de réduire la complexité du calcul du biais et également des résultats qui permettent de déterminer le biais exact de certaines relations de parité très utilisées de façon immédiate, et qui donnent aussi la façon d'associer les termes d'une relation de parité dans certains cas pour obtenir le meilleur biais possible.

3.2.1 Calculer le biais des relations de parité en fixant des variables

Définition 3.6. Soit f une fonction booléenne à v variables. On considère un sous-ensemble J des variables de f de taille n, $J = \{j_1, \ldots, j_n\} \subset \{1, \ldots, v\}$. Soit b un vecteur composé de n bits, $n \leq v$. Dans toute la suite, la restriction de f quand on fixe les variables dont les indices sont dans J aux valeurs correspondant à b est notée $f_{|b}$.

Pour simplifier, on considère aussi que les périodes minimales des différentes suites $\boldsymbol{x}_i$ sont premières entre elles, puisque ceci n'affecte pas nos résultats (en fait, cela pourrait uniquement les améliorer). Nous allons commencer par présenter le cas plus simple où l'ensemble $\mathcal{T}$ fait intervenir n variables et contient 2^n éléments, c'est-à-dire, le cas où g_τ est linéaire.

Relation de parité avec g_τ linéaire à n termes.

Dans ce cas, l'ensemble des instants $\mathcal{T}$ pour lesquels on va sommer la fonction est $\mathcal{T} = \langle T_{j_1}, \ldots, T_{j_n} \rangle$ et le nombre de termes dans la relation de parité est 2^n. La relation de

parité est en effet définie par

$$PC(t) = PC_{f,\mathcal{T}}(t) = \bigoplus_{\tau \in <T_{j_1},\ldots,T_{j_n}>} f(x_1(t+\tau),\ldots,x_v(t+\tau)).$$

Son biais vérifie $\mathcal{E}(PC_{f,\mathcal{T}}) \geq \varepsilon^{2^n}$, où ε peut être, comme nous l'avons vu avant, le biais de toute approximation g de f dont tous les termes disparaissent dans $PC_{f,\mathcal{T}}$. On atteindrait la borne de ε^{2^n} si toutes les variables étaient indépendantes, mais ce n'est pas le cas. Les variables qui sont dépendantes sont les n variables dont la période intervient dans la relation de parité. Puisque tous les périodes sont premières entre elles, les autres variables sont indépendantes, parce qu'elles sont répétées à des instants de temps séparés par des écarts différents de leur période. Ceci veut dire qu'on a $2^n \times n$ variables dépendantes dans la relation de parité et $2^n \times (v-n)$ variables indépendantes. Les variables dépendantes sont répétées une fois chacune, donc le nombre de variables différentes parmi les variables dépendantes est $2^{n-1} \times n$. Si on fixe ces variables dépendantes à une valeur quelconque a, la fonction restriction obtenue aura un biais facile à calculer, car on n'a qu'à multiplier les biais $\mathcal{E}(f_{|b})$ pour les valeurs de b appropriées. On peut calculer la valeur exacte de ce biais en multipliant car, maintenant, toutes les variables qui ne sont pas fixées sont indépendantes. Pour calculer le biais total de la relation de parité, on doit sommer les produits des biais $\mathcal{E}(f_{|b})$ pour chaque valeur possible des $2^{n-1} \times n$ variables dépendantes, et diviser ensuite par $2^{n(2^{n-1})}$. Ainsi, pour calculer le biais de la relation de parité on va faire :

$$\Pr[PC(t) = 0] = \frac{1}{2^{n(2^{n-1})+1}} \sum_{a \in \mathbf{F}_2^{n(2^{n-1})}} (1+\eta_a).$$

Décomposons ce vecteur a de $n2^{n-1}$ bits sous la forme de n vecteurs $a_1,\ldots,a_n$, chacun de 2^{n-1} bits. A chaque a, on va associer 2^n vecteurs $\chi(c,a), 0 \leq c < 2^n$, chacun de n bits. Ces vecteurs correspondent aux valeurs prises par le vecteur $(x_{j_1},\ldots,x_{j_n})$ à l'instant $t + \sum_{i=1}^n c_i T_{j_i}$ avec $c = \sum_{i=1}^n c_i 2^{i-1}$. Le bit k du vecteur $\chi(c,a), 1 \leq k \leq n$, est défini de la manière suivante :

- si $c_k = 1$, c'est-à-dire, si T_{j_k} intervient dans le décalage $\tau = \sum_{i=1}^n c_i T_{j_i}$ considéré, alors $x_{j_k}(t + \sum_{i=1}^n c_i T_{j_i})$ doit correspondre à $x_{j_k}(t + \sum_{i=1, i\neq k}^n c_i T_{j_i})$. Donc, on a

$$\chi_k(c,a) = \chi_k(c - 2^k, a).$$

- si $c_k = 0$, c'est-à-dire si T_{j_k} n'intervient pas dans le décalage considéré, alors $x_{j_k}(t + \sum_{i=1}^n c_i T_{j_i})$ prend une valeur indépendante des valeurs précédentes. Cette valeur correspond au premier bit non encore utilisé dans le vecteur a_k, c'est-à-dire

$$\chi_k(c,a) = a_{k,2^k q + r + 1}$$

où $c = 2^{k+1}q + r, r < 2^k$. En effet, on remarque d'abord que l'ensemble $\{c', 0 \leq c' < 2^{k+1}q\}$ est composé de $2^k q$ paires de la forme $(c', c' + 2^k)$ avec $c'_k = 0$. Donc, pour $0 \leq c' < 2^{k+1}q$, on a utilisé $2^k q$ bits de a_k. Ensuite, tous les c' dans l'ensemble

$\{2^{k+1}q, \ldots, 2^{k+1}q + r - 1\}$ vérifient $c'_k = 0$ car $r < 2^k$. Donc, les vecteurs $\chi(c, a)$ correspondants ont utilisé r bits de a_k. Ainsi, au total, on a utilisé $2^k q + r$ bits de a_k, ce qui donne la formule attendue pour l'indice du premier bit de a_k non utilisé.

Exemple 3.7. Nous allons voir un exemple pour $n = 3$. Le vecteur a est vu comme un vecteur de $(\mathbf{F}_2^4)^3$, (a_1, a_2, a_3) et on note $a_{i,j}$ le bit j de $a_i \in \mathbf{F}_2^4, 1 \leq i \leq 3, 0 \leq j \leq 3$. Le tableau suivant donne alors la valeur des 8 vecteurs $\chi(c, a)$ pour $0 \leq c < 8$.

c	$\chi_3(c,a)$	$\chi_2(c,a)$	$\chi_1(c,a)$
0	$a_{3,0}$	$a_{2,0}$	$a_{1,0}$
1	$a_{3,1}$	$a_{2,1}$	$a_{1,0}$
2	$a_{3,2}$	$a_{2,0}$	$a_{1,1}$
3	$a_{3,3}$	$a_{2,1}$	$a_{1,1}$
4	$a_{3,0}$	$a_{2,2}$	$a_{1,2}$
5	$a_{3,1}$	$a_{2,3}$	$a_{1,2}$
6	$a_{3,2}$	$a_{2,2}$	$a_{1,3}$
7	$a_{3,3}$	$a_{2,3}$	$a_{1,3}$

Maintenant, on peut écrire η_a comme :

$$\eta_a = \prod_{c=0}^{2^n-1} \mathcal{E}(f_{|\chi(c,a)})$$

Nous pouvons donc énoncer le théorème suivant.

Théorème 3.8. *Soient $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_v$ v suites de périodes minimales $T_1, \ldots T_v$ et f une fonction booléenne à v variables. Soit*

$$\mathcal{T} = \{\sum_{i=1}^{n} c_i M_i, c_i \in \{0, 1\}\}$$

où $M_i = q_i ppcm(T_{\ell_i+1}, \ldots, T_{\ell_{i+1}})$ avec $q_i > 0$, $\ell_1 = 0$ et $\ell_{n+1} = n$. Comme indiqué précédemment, on considère que $\mathcal{T}$ ne contient aucun multiple de T_j, pour tout $n < j \leq v$. Alors nous avons

$$\mathcal{E}(PC_{f,\mathcal{T}}) = \frac{1}{2^{n(2^n-1)}} \sum_{a \in \mathbf{F}_2^{n \cdot 2^{n-1}}} \prod_{c=0}^{2^n-1} \mathcal{E}(f_{|\chi(c,a)})$$

où $f_{|b}$ désigne la restriction de f quand les variables d'indices $j_1, \ldots, j_n$ sont fixées aux bits de $b \in \mathbf{F}_2^n$.

Ce résultat fournit un algorithme pour calculer le biais de la relation de parité beaucoup plus efficace que la méthode naïve, comme nous allons le voir plus loin dans le cas général.

Relation de parité avec g_τ non linéaire.

Nous généralisons maintenant le cas précédent au cas où le nombre d'éléments dans $\mathcal{T}$ est 2^m, avec $2^m < 2^n$, où n est le nombre de variables intervenant dans la construction de la relation de parité. Ceci veut dire qu'on a associé plusieurs variables pour construire la relation de parité, par exemple, on a considéré la suite $(x_{j_1} + x_{j_2})$ comme une seule suite de période $T_{j_1}T_{j_2}$, comme dans notre attaque contre Achterbahn. Nous avons $\mathcal{T} = \{M_1, \ldots M_m\}$ où M_i est un multiple non nul du ppcm de $T_{\ell_i+1}, \ldots, T_{\ell_{i+1}}$ avec $\ell_1 = 0$ et $\ell_{m+1} = n$. On peut définir la relation de parité comme :

$$PC(t) = PC_{f,\mathcal{T}}(t) = \bigoplus_{\tau \in \mathcal{T}} f(x_1(t+\tau), \ldots, x_v(t+\tau)).$$

Maintenant, comme on a fait dans la section précédente, on fixe les variables dépendantes à toutes les valeurs possibles et on calcule le biais de cette relation de parité de la manière suivante :

$$\Pr[PC(t) = 0] = \frac{1}{2^{n \cdot 2^{m-1}+1}} \sum_{a \in \mathbf{F}_2^{n \cdot 2^{m-1}}} (1 + \frac{1}{\eta_a}).$$

Ici a est un vecteur de taille $n2^{m-1}$. Nous allons lui associer n vecteurs $(a_1, \ldots, a_n)$ de 2^{m-1} bits chacun. Nous allons utiliser la correspondance entre les valeurs de $\tau = \sum_{i=1}^m c_i M_i$ dans $\mathcal{T}$ et les entiers c, $0 \leq c \leq 2^m - 1$ définis par $c = \sum_{i=1}^m c_i 2^{i-1}$. Alors, la valeur du mot de n bits $(x_1(t+\tau), \ldots, x_n(t+\tau))$ sera égale à $\chi(c, a) = (\chi_1(c, a), \ldots, \chi_n(c, a))$ où, pour tout k tel que $\ell_i < k \leq \ell_{i+1}$, on a

$$\chi_k(c, a) = \begin{cases} \chi_k(c - 2^i, a) & \text{si } c_i \neq 0 \\ a_{k, 2^i q + r + 1} & \text{si } c = 2^{i+1}q + r, r < 2^i. \end{cases}$$

En effet, si $c_i \neq 0$, nous obtiendrons que c et $c' = c - 2^i$ sont associés à une paire (τ, τ') avec $\tau - \tau' = M_i$. Comme M_i est multiple d'une période de $\boldsymbol{x_k}$, nous pouvons déduire que $\chi_k(c, a) = \chi_k(c', a)$.

Si $c_i = 0$, la valeur correspondant de $x_k(t+\tau)$ est indépendante des valeurs précédentes et doit être définie par un bit de a_k qui n'a pas encore été utilisé pour des valeurs plus petites de c. Le nombre de bits de a_k qui ont été utilisés par les vecteurs précédents $\chi_k(c', a)$ pour $c' < 2^{i+1}q$ est $2^i q$ car l'ensemble $\{0, \ldots, 2^{i+1}q - 1\}$ est formé par $2^i q$ paires de la forme $(c', c' + 2^i)$ avec $c'_i = 0$. Par ailleurs, tous les c' dans $\{2^{i+1}q, \ldots, 2^{i+1}q + r - 1\}$ vérifient $c'_i = 0$ parce que $r < 2^i$. C'est pour cela qu'il y a exactement $(2^i q + r)$ bits de a_k qui ont été utilisés pour $\chi_k(c', a)$, $c' < 2^{i+1}q + r$.

Exemple 3.9. Nous considérons un ensemble $\mathcal{T}$ composé de 2^3 éléments qui incluent les périodes de 4 suites :

$$\mathcal{T} = \langle T_1 T_2, T_3, T_4 \rangle.$$

Alors, les mots de 4 bits $\chi(c,a)$, $0 \leq c < 8$, sont définis par le mot de 16 bits a de la façon suivante, où les éléments en gras sont ceux déjà utilisés pour des valeurs plus petites de c :

$$\begin{aligned}
\chi(0,a) &= (a_{00}a_{10}a_{20}a_{30}) & \chi(4,a) &= (a_{02}a_{12}a_{22}\boldsymbol{a_{30}}) \\
\chi(1,a) &= (\boldsymbol{a_{00}}\boldsymbol{a_{10}}a_{21}a_{31}) & \chi(5,a) &= (\boldsymbol{a_{02}}\boldsymbol{a_{12}}a_{23}\boldsymbol{a_{31}}) \\
\chi(2,a) &= (a_{01}a_{11}\boldsymbol{a_{20}}a_{32}) & \chi(6,a) &= (a_{03}a_{13}\boldsymbol{a_{22}}\boldsymbol{a_{32}}) \\
\chi(3,a) &= (\boldsymbol{a_{01}}\boldsymbol{a_{11}}\boldsymbol{a_{21}}a_{33}) & \chi(7,a) &= (\boldsymbol{a_{03}}\boldsymbol{a_{13}}\boldsymbol{a_{23}}\boldsymbol{a_{33}})
\end{aligned}$$

La définition de $\chi(c,a)$ nous permet d'exprimer le biais de $PC_{f,\mathcal{T}}$ en fonction des biais des restrictions de f quand les variables d'indice $j_1, \ldots, j_n$ sont fixées.

On peut écrire η_a comme :

$$\eta_a = \prod_{c=0}^{2^m-1} \mathcal{E}(f_{|\chi(c,a)}),$$

et nous pouvons donc énoncer le théorème suivant :

Théorème 3.10. *Soient $\boldsymbol{x_1}, \ldots, \boldsymbol{x_v}$ v suites de périodes minimales $T_1, \ldots, T_v$, et f une fonction booléenne à v variables. Soit*

$$\mathcal{T} = \{\sum_{i=1}^{m} c_i M_i, \;\; c_i \in \{0,1\}\}$$

où $M_i = q_i \mathrm{ppcm}(T_{\ell_i+1}, \ldots, T_{\ell_{i+1}})$ avec $q_i > 0$, $\ell_1 = 0$ and $\ell_{m+1} = n$. Nous supposons que $\mathcal{T}$ ne contient pas de multiples de T_j, pour tout $n < j \leq v$.

Alors, nous avons

$$\mathcal{E}(PC_{f,\mathcal{T}}) = \frac{1}{2^{n2^{m-1}}} \sum_{a \in \mathbf{F}_2^{n2^{m-1}}} \prod_{c=0}^{2^m-1} \mathcal{E}(f_{|\chi(c,a)}),$$

où $f_{|b}$ désigne la restriction de f quand les variables d'indices $j_1, \ldots, j_n$ sont fixées aux bits de $b \in \mathbf{F}_2^n$.

Preuve :

$$\begin{aligned}
\Pr[PC_{f,\mathcal{T}}(t) = 0] &= \frac{1}{2^{n2^{m-1}}} \sum_{a \in \mathbf{F}_2^{n2^{m-1}}} \Pr[PC_{f,\mathcal{T}}(t) = 0 | (x_1(t+\tau), \ldots, x_n(t+\tau), \tau) = a] \\
&= \frac{1}{2^{n2^{m-1}}} \sum_{a \in \mathbf{F}_2^{n2^{m-1}}} \Pr[PC_{f,\mathcal{T}}(t) = 0 | (x_1(t+\tau), \ldots, x_n(t+\tau)) = \chi(c,a), \\
&\qquad \forall \tau = \sum_{i=1}^{m} c_i M_i].
\end{aligned}$$

Quand les valeurs des n premières variables dans tous les termes de $PC_{f,\mathcal{T}}$ sont fixées, le piling-up lemma peut être appliqué car les $(v-n)2^m$ variables qui restent sont statistiquement indépendantes. La raison est que τ n'est pas un multiple des périodes T_j, pour $n < j \leq v$. Alors, on peut déduire que le terme correspondant à a dans la somme précédente est égal à :

$$\begin{aligned}\Pr[PC_{f,\mathcal{T}}(t)=0|a] &= \frac{1}{2}\left[1+\prod_{\tau\in\mathcal{T}}\mathcal{E}(f(x(t+\tau),y(t+\tau))|x(t+\tau)=\chi(c,a))\right] \\ &= \frac{1}{2}\left[1+\prod_{c=0}^{2^m-1}\mathcal{E}(f_{|\chi(c,\alpha)})\right],\end{aligned}$$

avec $\tau = \sum_{i=1}^{m} c_i M_i$. Alors on déduit que

$$\Pr[PC_{f,\mathcal{T}}(t)=0] = \frac{1}{2}\left[1+\frac{1}{2^{n2^{m-1}}}\sum_{a\in\mathbf{F}_2^{n2^{m-1}}}\prod_{c=0}^{2^m-1}\mathcal{E}(f_{|\chi(c,\alpha)})\right]$$

où, de façon équivalente, que

$$\mathcal{E}(PC_{f,\mathcal{T}}) = \frac{1}{2^{n2^{m-1}}}\sum_{a\in\mathbf{F}_2^{n2^{m-1}}}\prod_{c=0}^{2^m-1}\mathcal{E}(f_{|\chi(c,\alpha)}).$$

□

Avec ce résultat nous obtenons un algorithme pour calculer la valeur exacte de $\mathcal{E}(PC_{f,\mathcal{T}})$. L'étape de précalcul doit calculer et stocker dans un tableau les 2^n valeurs de

$$\mathcal{E}(f_{|b}) = \frac{1}{2^n}\sum_{y\in\mathbf{F}_2^{v-n}}(-1)^{f(b,y)}, \quad \forall b\in\mathbf{F}_2^n.$$

Cette étape a besoin de 2^v évaluations de f. Ensuite, pour calculer le biais de la relation de parité nous devons calculer, pour tout $a \in \mathbf{F}_2^{n2^{m-1}}$, le produit des 2^m valeurs précalculées dont les indices sont donnés par $\chi(c,a)$, pour $0 \leq c < 2^m$. Ceci coûte $2^{n2^{m-1}} \times 2^m$ opérations sur des entiers. Ceci conduit à une complexité totale de $2^{n2^{m-1}+m} + 2^v$, ce qui est beaucoup plus petit que la complexité du calcul trivial, qui est de 2^{v2^m} évaluations de f. Si on prend un exemple avec $v = 13$, $n = 7$ et $m = 3$, comme dans l'attaque sur Achterbahn-128, la complexité de calcul du biais est de 2^{23} à la place de 2^{76}. Ceci a été implémenté et nous a permis de calculer les biais exacts des relations de parité ce qu'on n'aurait pas pu faire avec l'algorithme naïf par recherche exhaustive. Par exemple, pour la fonction de combinaison de Achterbahn-128, à 13 variables, on a calculé le biais de toutes les relations de parité possibles qu'on peut construire avec $n = 6$ et $m = 3$. On a calculé chaque biais avec une complexité de calcul de 2^{27}. Avec l'algorithme naïf, la complexité du calcul de chaque biais aurait été de 2^{80}.

Algorithme 4 Calcul du biais exact de $PC_{f,\mathcal{T}}$

ENTRÉES : f, une fonction booléenne à v variables et

$$\mathcal{T} = \{\sum_{i=1}^{m} c_i M_i, \;\; c_i \in \{0,1\}\}$$

où $M_i = q_i \text{ppcm}(T_{\ell_i+1}, \ldots, T_{\ell_{i+1}})$ avec $q_i > 0$, $\ell_1 = 0$ and $\ell_{m+1} = n$.
/*Précalcul*/
pour tout $b \in \mathbf{F}_2^n$ **faire**
  $E_b \leftarrow \mathcal{E}(f_{|b}) = \frac{1}{2^n} \sum_{y \in \mathbf{F}_2^{v-n}} (-1)^{f(b,y)}$
fin pour
/*Calcul du biais*/
$\mathcal{E} \leftarrow 0$
pour tout $a \in \mathbf{F}_2^{n2^{m-1}}$ **faire**
  pour c de 0 à $2^m - 1$ **faire**
    $\mathcal{A} \leftarrow 1$
    $b \leftarrow \chi(c, a)$
    $\mathcal{A} \leftarrow \mathcal{A} \times E_b$
  fin pour
  $\mathcal{E} \leftarrow \mathcal{E} + \mathcal{A}$
fin pour
SORTIES : $\mathcal{E}(PC_{f,\mathcal{T}}) = \frac{1}{2^{n2^{m-1}}} \mathcal{E}$.

3.2.2 Calculer le biais des relations de parité avec la transformée de Fourier

Une autre expression similaire pour le biais de $\mathcal{E}(PC_{f,\mathcal{T}})$ peut être obtenue à partir des coefficients de Walsh de f, *i.e.* à partir de tous les biais $\mathcal{E}(f + \varphi_{b,0}), b \in \mathbf{F}_2^n$, c'est-à-dire des biais de toutes les approximations linéaires faisant intervenir les variables de J.

Théorème 3.11. *Soient $\boldsymbol{x_1}, \ldots, \boldsymbol{x_v}$ v suites avec des périodes minimales $T_1, \ldots, T_v$ et f une fonction booléenne à n variables. Soit*

$$\mathcal{T} = \Big\{ \sum_{i=1}^{m} c_i M_i, \;\; c_i \in \{0,1\} \Big\}$$

où $M_i = q_i \text{ppcm}(T_{\ell_i+1}, \ldots, T_{\ell_{i+1}})$ avec $q_i > 0$, $\ell_1 = 0$ and $\ell_{m+1} = n$. Nous supposons que $\mathcal{T}$ ne contient aucun multiple de T_j, pour $n < j \leq v$. Alors, nous avons

$$\mathcal{E}(PC_{f,\mathcal{T}}) = \sum_{a \in \mathbf{F}_2^{n2^{m-1}}} \prod_{c=0}^{2^m-1} \mathcal{E}(f \oplus \varphi_{(\chi(c,a),0)}).$$

Preuve : Par définition, le biais $\mathcal{E}(PC_{f,\mathcal{T}})$ correspond au biais de la fonction booléenne à $v2^m$ variables, qui à $x_0, \ldots, x_{2^m-1}$ avec $x_i \in \mathbf{F}_2^v$ associe

$$\bigoplus_{c=0}^{2^m-1} f(x_c),$$

avec la condition $x_{i,c} = x_{i,c'}$ quand $|c' - c| = 2^i$ pour un $1 \leq i \leq n$. Ce qui impose que $x_{i,c} = x_{i,c'}$, ce qui est équivalent à multiplier le biais par

$$\frac{1}{2} \sum_{b \in \mathbf{F}_2} (-1)^{b \cdot (x_{i,c} \oplus x_{i,c'})}$$

parce que

$$\sum_{b \in \mathbf{F}_2} (-1)^{b \cdot x} = \begin{cases} 0 & \text{si } x = 1 \\ 2 & \text{si } x = 0. \end{cases}$$

Soit

$$C = \prod_{i=1}^{n} \prod_{(c,c'), c'-c=2^i} \sum_{\lambda_{i,c} \in \mathbf{F}_2} (-1)^{\lambda_{i,c} \cdot (x_{i,c} \oplus x_{i,c'})}. \tag{3.2}$$

Alors on déduit que

$$\begin{aligned} \mathcal{E}(PC_{f,\mathcal{T}}) &= \frac{1}{2^{v2^m}} \sum_{(x_0,\ldots,x_{2^m-1}) \in \mathbf{F}_2^{v2^m}} \prod_{c=0}^{2^m-1} (-1)^{f(x_c)} C \\ &= \frac{1}{2^{v2^m}} \prod_{c=0}^{2^m-1} \left(\sum_{x_c \in \mathbf{F}_2^v} (-1)^{f(x_c)} \right) C \\ &= \frac{1}{2^{v2^m}} \sum_{\lambda \in \mathbf{F}_2^{n2^{m-1}}} \prod_{c=0}^{2^m-1} \left(\sum_{x_c \in \mathbf{F}_2^v} (-1)^{f(x_c) \oplus x_c \cdot \chi(c,\lambda)} \right). \end{aligned}$$

La dernière égalité vient du fait qu'il y a $n2^{m-1}$ couples (c, c') avec $c' - c = 2^i$ pour un certain $1 \leq i < n$. Alors $n2^{m-1}$ éléments $\lambda_{i,c}$ de $\mathbf{F}_2$ interviennent dans la formule (3.2). De plus, on peut voir par définition que $\lambda_{i,c}$ peut être défini par $\lambda_{i,c} = \chi(c, \lambda)_i$ où λ est un vecteur à $n2^{m-1}$ bits. Si on utilise que

$$\sum_{x_c \in \mathbf{F}_2^v} (-1)^{f(x_c) \oplus x_c \cdot \chi(c,\lambda)} = 2^v \mathcal{E}(f \oplus \varphi_{\chi(c,\lambda)})$$

on déduit que

$$\mathcal{E}(PC_{f,\mathcal{T}}) = \sum_{\lambda \in \mathbf{F}_2^{n2^{m-1}}} \prod_{c=0}^{2^m-1} \mathcal{E}(f \oplus \varphi_{(\chi(c,\lambda),0)}).$$

□

Cette expression nous donne un algorithme pour calculer le biais très similaire au précédent, mais ici nous devons précalculer et stocker les coefficients de Walsh de f qui correspondent à tous les éléments de la forme $(b, 0)$ avec $b \in \mathbf{F}_2^n$.

La complexité en temps pour calculer le biais avec cette méthode est aussi $2^{n2^{m-1}} \times 2^m + 2^v$.

3.2.3 Combiner les deux méthodes : fixer des variables et la transformée de Fourier

En utilisant l'équivalence des résultats précédents, on obtient l'égalité suivante :

Proposition 3.12. *Avec les notations du théorème 3.11, on a*

$$2^{n\cdot 2^{m-1}} \cdot \sum_{a\in\mathbf{F}_2^{n\cdot 2^{m-1}}} \prod_{j=0}^{2^m-1} \mathcal{E}(f \oplus \varphi_{(\chi(c,a),0)}) = \sum_{a\in\mathbf{F}_2^{n\cdot 2^{m-1}}} \prod_{j=0}^{2^m-1} \mathcal{E}(f_{|}\chi(c,a))$$

On peut maintenant combiner les deux techniques pour calculer le biais d'une relation de parité. Si on construit une relation de parité associée à g_τ linéaire avec n variables, on peut séparer n en n_1 et n_2, où n_1 représente les variables qu'on fixe et n_2 les variables avec lesquelles on construit les approximations linéaires. Dans ce cas, en utilisant les vecteurs $\chi(c, a)$ et en considérant les n_1 premiers vecteurs associés aux n_1 variables qu'on fixe, et les n_2 derniers vecteurs associés aux variables qu'on utilise pour les approximations linéaires, on a que :

Proposition 3.13.

$$\mathcal{E}(PC_{f,\mathcal{T}}) = \frac{1}{2^{n_1\cdot 2^{n-1}}} \sum_{a\in\mathbf{F}_2^{n\cdot 2^{n-1}}} \prod_{j=0}^{2^n-1} \mathcal{E}\left((f \oplus \varphi_{(0,\chi_2(c,a),0)})|\chi_1(c,a)\right),$$

où $\chi(c, a) = (\chi_1(c, a), \chi_2(c, a))$ et $\chi_1(c, a)$ incluent les n_1 premières variables alors que $\chi_2(c, a)$ inclut les n_2 variables suivantes.

On peut faire naturellement de même quand la relation de parité n'est pas associée à une fonction g_τ linéaire.

3.2.4 Quand f est résiliente

Définition 3.14. Une fonction booléenne est dite t-résiliente quand elle est équilibrée et que la distribution de ses sorties ne change pas si on fixe t ou moins de ses variables d'entrée.

Dans les générateurs par combinaison, on utilise toujours des fonctions de combinaison qui ont un ordre de résilience élevé afin de résister aux attaques par corrélation [Sie84,

Sie85]. Mais, l'ordre de résilience ne peut pas être aussi élevé que l'on veut car, pour une fonction à n variables, on a : $t + \deg f \leq n - 1$. On va tout d'abord étudier ce qu'il se passe avec des fonctions t-résilientes quand on construit des relations de parité impliquant $(t + 1)$ variables.

Relation de parité impliquant $t + 1$ variables.

Ceci est la situation des attaques contre Achterbahn. On va prendre l'exemple des attaques contre Achterbahn-80 effectuées par Hell et Johansson [HJ07] et des nôtres [NP07a]. Dans la première attaque, Hell et Johansson utilisaient l'approximation suivante de la fonction de combinaison G :

$$g_1 = x_1 \oplus x_2 \oplus x_7 \oplus x_3x_{10} \oplus x_4x_9, \quad \mathcal{E}(G \oplus g_1) = 2^{-5}$$

pour construire la relation de parité définie par

$$\mathcal{T} = \langle T_3T_{10}, T_4T_9 \rangle.$$

Donc, d'après la borne du piling-up lemma, on pouvait dire que $\mathcal{E}(PC_{G\oplus g_1,\mathcal{T}}) \geq 2^{-20}$. Mais Hell et Johansson ont calculé son biais exact (avec une recherche exhaustive et une complexité 2^{36}, alors qu'avec notre algorithme on peut le calculer en complexité 2^{11}), et ont trouvé qu'il était $\mathcal{E}(PC_{G\oplus g_1,\mathcal{T}}) = 2^{-12}$.

Dans notre attaque, on a utilisé le théorème suivant.

Théorème 3.15. *[CT00] Soit f une fonction à v variables et t son ordre de résilience. Alors, pour tout J de taille $t + 1$ la meilleure approximation de f par une fonction à variables dans J est la fonction affine*

$$\bigoplus_{j\in J} x_j \oplus \delta, \ \delta \in \{0, 1\} \ .$$

À cause de ce résultat on a choisi une approximation linéaire,

$$g_2 = x_1 \oplus x_2 \oplus x_7 \oplus (x_3 \oplus x_{10}) \oplus (x_4 \oplus x_9), \quad \mathcal{E}(G \oplus g_2) = 2^{-3},$$

pour construire la même relation de parité. On a pu faire ceci car, comme on l'a expliqué dans la section précédente, toute relation de parité peut être construite à partir d'une approximation linéaire. La borne qu'on a obtenue correspond alors au biais exact de la relation, $\mathcal{E}(PC_{G\oplus g_1,\mathcal{T}}) = \mathcal{E}(PC_{G\oplus g_2,\mathcal{T}}) = 2^{-12}$. On a pu montrer le théorème suivant, aussi mentionné plus tard par Göttfert et Gammel [GG07] :

Théorème 3.16. *Soit f une fonction t-résiliente. Le biais de toute relation de parité construite à partir d'une approximation linéaire φ de f à $(t + 1)$-variables avec un biais $\mathcal{E}(f \oplus \varphi) = \varepsilon$ est ε^{2^m} où 2^m est le nombre de termes présents dans la relation de parité.*

Preuve : Le théorème est facile à prouver si on utilise l'expression du théorème 3.11 :

$$\mathcal{E}(PC_{f,\mathcal{T}}) = \sum_{a \in \mathbf{F}_2^{n 2^{m-1}}} \prod_{c=0}^{2^m-1} \mathcal{E}(f \oplus \varphi_{(\chi(c,a),0)}).$$

Le produit

$$\prod_{c=0}^{2^m-1} \mathcal{E}(f \oplus \varphi_{(\chi(c,a),0)})$$

ne s'annule pas si et seulement si $\chi(c,a)$ est le mot tout à un pour tout $0 \leq c < 2^m$. En effet, quand f est t-résiliente, le seul vecteur $b \in \mathbf{F}_2^{t+1}$ tel que $\mathcal{E}(f + \varphi_{(b,0)}) \neq 0$ est $b = 1_{t+1}$. Donc le mot a défini par $a_{i,j} = 1$ pour tout $1 \leq i \leq n$ et $0 \leq j < 2^{m-1}$ est le seul qui satisfait la condition. La somme définissant $\mathcal{E}(PC_{f,\tau})$ ne contient donc qu'un seul terme :

$$\mathcal{E}(PC_{f,\tau}) = \left[\mathcal{E}(f \oplus \varphi)\right]^{2^n}.$$

□

On va voir quelques exemples, les mêmes que ceux utilisés dans la section précédente. On rappelle que g_2 est une approximation linéaire de G, la fonction de combinaison 6-résiliente d'Achterbahn-80 :

$$g_2(x_1, x_2, x_3, x_4, x_7, x_9, x_{10}) = x_1 \oplus x_2 \oplus x_7 \oplus x_3 \oplus x_{10} \oplus x_4 \oplus x_9,$$

avec $\mathcal{E}(G \oplus g_2) = \varepsilon = 2^{-3}$. Dans ce cas on peut calculer les biais des relations de parité suivantes, dérivées chacune de g, avec le théorème précédent, puisque g_2 a $t+1$ variables, où $t = 6$ est l'ordre de résilience de G :

- pour $f_1'(x_1, \ldots, x_{11}) = G(x_1, \ldots, x_{11}) \oplus x_1 \oplus x_2 \oplus x_7$

$$PC_1(t) = f_1'(t) \oplus f_1'(t + T_3T_{10}) \oplus f_1'(t + T_4T_9) \oplus f_1'(t + T_3T_{10} + T_4T_9), \quad \varepsilon = 2^{-12}.$$

- pour $f_1'(x_1, \ldots, x_{11}) = G(x_1, \ldots, x_{11}) \oplus x_1 \oplus x_2 \oplus x_7$

$$PC_2(t) = f_1'(t) \oplus f_1'(t + T_3) \oplus f_1'(t + T_4T_{10}T_9) \oplus f_1'(t + T_3 + T_4T_{10}T_9), \quad \varepsilon = 2^{-12}.$$

- pour $f_1'(x_1, \ldots, x_{11}) = G(x_1, \ldots, x_{11})$

$$PC_3(t) = f_3'(t) \oplus f_3'(t + T_1T_2T_7) \oplus f_3'(t + T_3T_4T_9T_{10}) \oplus f_3'(t + T_1T_2T_7 + T_3T_4T_9T_{10}), \quad \varepsilon = 2^{-12}.$$

- pour $f_1'(x_1, \ldots, x_{11}) = G(x_1, \ldots, x_{11}) \oplus x_2 \oplus x_4 \oplus x_9 \oplus x_{10}$

$$PC_4(t) = f_4'(t) \oplus f_4'(t + T_1T_3T_7), \; \varepsilon = 2^{-6}.$$

Relation de parité impliquant $t+k$ variables.

Maintenant on peut se demander si on peut dire quelque chose sur le biais quand on construit les relations de parité avec plus de $(t+1)$ variables où t est l'ordre de résilience de la fonction, c'est-à-dire, avec $(t+k)$ et $k>1$. On ne peut pas appliquer le théorème de la section précédente. Mais, nous pouvons appliquer le corollaire 3.5.

On va faire un petit résumé de la situation pour une fonction f t-résiliente. On considère une relation de parité $PC = PC_{f,\mathcal{T}}$ construite à partir de n variables et qui contient 2^m termes.

- Si $n < t+1$:

$$\mathcal{E}(PC) = 0.$$

- Si $n = t+1$:

$$\mathcal{E}(PC) = \varepsilon^{2^m}$$

où ε est le biais de l'approximation linéaire qu'on peut construire avec les variables impliquées dans l'ensemble $\mathcal{T}$ utilisé dans la relation de parité.

- Si $n = t+1+k$:

$$\mathcal{E}(PC) \geq \varepsilon^{2^m}$$

où ε est le biais de la meilleure approximation linéaire qu'on peut construire en utilisant des variables parmi les n considérées dans $\mathcal{T}$.

3.3 Quand f est une fonction plateau t-résiliente

Nous étudions dans une section séparée les fonctions plateaux car, grâce à leur forme particulière, elles nous permettent de calculer le biais des relations de parité d'une façon très performante, quand $n = t+2$. Ensuite on va montrer comment construire les relations de parité qui ont le meilleur biais, résultat qui sera généralisable partiellement à tout type de fonction. On va donner une borne supérieure de ce biais pour $n = t+k$, $\forall k$, et finalement on verra quelques exemples illustratifs. La notion de fonction plateau a été définie à l'origine [ZZ99] en termes de coefficients de Walsh mais, dans notre contexte, nous utiliserons la formulation équivalente suivante.

Définition 3.17. [ZZ99] Une fonction booléenne f est une fonction plateau si le biais de toutes ses approximations linéaires appartient à $\{0, \pm\varepsilon\}$

On peut alors démontrer [CCCF00] que le biais ε des approximations linéaires biaisées d'une fonction plateau est de la forme 2^{i-n} où i est un entier. Ce sont des fonctions très utilisées en cryptographie. Par exemple, les fonctions booléennes d'Achterbahn sont plateaux. Elles sont plus simples à étudier quand on construit des relations de parité, car les biais de toutes leurs approximations linéaires biaisées ont la même valeur absolue.

3.3.1 Comment calculer le biais efficacement (quand $n = t + 2$)

On va faire l'étude avec des fonctions plateaux $t-$résilientes. On va utiliser l'expression de la section précédente sur le biais des relations de parité :

$$\mathcal{E}(PC_{f,\mathcal{T}}) = \sum_{a \in \mathbf{F}_2^{n \cdot 2^{m-1}}} \prod_{c=0}^{2^m-1} \mathcal{E}(f \oplus \varphi_{(\chi(c,a),0)}) \tag{3.3}$$

où tous les $\mathcal{E}(f + \varphi_{(u,0)}), u \in \mathbf{F}_2^n$ ont la même valeur absolue quand ils ne sont pas nuls. Pour un certain a la valeur du produit va être 0, si au moins un des $\mathcal{E}(f + \varphi_{(u,0)})$ qui y intervient est zéro, et il va être $\pm\varepsilon^{2^m}$ si aucun des biais qui intervient n'est nul, où ε est le biais de toutes les approximations linéaires biaisées de f.

Montrons maintenant pourquoi, dans le cas où on construit des relations des parité avec $t + 2$ variables, si ce produit est non nul, il vaut $+\varepsilon^{2^m}$ et non $-\varepsilon^{2^m}$. Comme on vient de le dire, quand le produit n'est pas nul, aucun $\mathcal{E}(f \oplus \varphi_{(u,0)})$ n'est nul, ce qui signifie que le poids de Hamming du vecteur $\chi(c, a)$ vérifie pour tout $0 \leq c \leq 2^m - 1 : wt(\chi(c, a)) \geq t+1$. Sinon, puisque f est $t-$résiliente, le biais de l'approximation correspondante serait nul. Considérons l'ensemble de vecteurs de $(t + 2)$ bits $\{\chi(c, a), 0 \leq c \leq 2^m\}$, et un élément de poids $t + 1$ dans cet ensemble. On note i la position de $\{1, \ldots t + 2\}$ qui n'est pas dans le support de cet élément. Alors, pour $c = \sum_{j=1}^{m} c_j 2^j$, si $c_i = 0, \chi(c + 2^i, a) = \chi(c, a)$, car le bit i de $\chi(c, a)$ vaut 1 et les autres sont nécessairement égaux à 1. De même, si $c_i = 1$, $\chi(c - 2^i, a) = \chi(c, a)$.

Donc si $wt(\chi(c, a)) = t+1$, l'approximation associée apparaît deux fois dans le produit des biais, et donc les signes vont être égaux 2 à 2. Comme chaque élément $\chi(c, a)$ de poids $(t + 1)$ apparaît un nombre pair de fois dans le produit et que le produit a 2^m termes, l'élément $\chi(c, a)$ de poids $(t + 2)$, apparaît donc aussi un nombre pair de fois. Le signe du produit est donc toujours positif.

Donc, dans le cas des fonctions plateaux t-résilientes avec $n = t + 2$, calculer $\mathcal{E}(PC_{f,\mathcal{T}})$ revient à calculer N_s, qui est le nombre de a dans $\mathbf{F}_2^{n2^{m-1}}$ pour lesquels aucun $\mathcal{E}(f \oplus \varphi_{(\chi(c,a),0)})$ n'est nul. On propose ici une méthode efficace pour calculer N_s. On peut faire ceci en analysant toutes les approximations linéaires biaisées qu'on peut construire en utilisant les n variables qui apparaissent dans $\mathcal{T}$, réparties en m groupes définies par les valeurs de M_i où $\mathcal{T} = \langle M_1, \ldots, M_m \rangle$: pour chaque approximation linéaire associée, on groupe ses variables de la même façon que les m groupes définies par $\mathcal{T}$. Comme les valeurs prises par les variables des m groupes ne peuvent pas être les mêmes pour deux approximations différentes, ces valeurs diffèrent pour au moins un groupe. La question maintenant est de calculer N_s, c'est-à-dire, combien de combinaisons des différents groupes on peut faire suivant la structure de la relation de parité, de façon que tous les termes dans la relation de parité soient associés à une des approximations linéaires biaisées.

Exemple 3.18. Pour une fonction 1-résiliente à 5 variables, on construit la relation de parité associée à

$$\mathcal{T} = \langle T_1 T_2, T_3 \rangle.$$

Cet ensemble $\mathcal{T}$ implique $n = t + 2 = 3$ variables, dont les indices sont dans $J = \{1, 2, 3\}$. La relation de parité a 2^m termes avec $m = 2$. La forme de $\mathcal{T}$ décompose J en $m = 2$ groupes de variables : $G_1 = \{3\}$ et $G_2 = \{1, 2\}$. Supposons que la fonction étudiée a 3 approximations linéaires biaisées impliquant les variables de J, x_1, $x_2 \oplus x_3$ et $x_1 \oplus x_2 \oplus x_3$. Alors on détermine la valeur de chaque approximation sur chacun des 2 groupes de variables G_1 et G_2

	G_2	G_1
x_1	(1,0)	0
x_2+x_3	(0,1)	1
$x_1+x_2+x_3$	(1,1)	1

TAB. 3.1 – Exemple des groupes G_i formés avec des approximations

La valeur de N_s que nous allons calculer dépend alors de ces décompositions.

Nous allons voir que le nombre N_s peut être facilement calculé avec une formule dans le cas où toutes les approximations linéaires biaisées sont égales au vecteur tout à 1 sur tous les groupes, sauf un pour lequel elles sont toutes différentes. Dans ce cas, la formule pour calculer N_s est :

$$N_s = N^{2^{m-1}},$$

où N est le nombre d'approximations linéaires biaisées associées aux n variables. Si ceci n'est pas le cas, on peut aussi calculer N_s facilement avec un algorithme.

3.3.2 Comment construire les relations de parité pour avoir un meilleur biais

Quand $n = t + 2$**.**

Nous allons donc démontrer le résultat sur le valeur N_s dans la configuration optimale que nous venons de décrire et nous l'illustrerons par le cas $m = 3$, qui est représenté dans la table 3.2. Les variables $j_1, \ldots, j_n$ impliquées dans $\mathcal{T}$ sont donc réparties en m groupes déterminés par les périodes apparaissant dans M_i, $1 \leq i \leq m$ où $\mathcal{T} = \langle M_1, \ldots, M_m \rangle$. Le nombre de vecteurs a de $\mathbf{F}_1^{n2^{m-1}}$ pour lesquels on obtient le produit ε^{2^m} dans (3.3) est déterminé par le nombre de possibilités pour les coordonnées de $\chi(c, a)$, $0 \leq c < 2^m$ telles que ces 2^m vecteurs correspondent à des approximations linéaires biaisées de f. Dans l'exemple avec $m = 3$, on compte le nombre de possibilités pour $\alpha_1, \beta_1, \gamma_1, \delta_1, \alpha_2, \beta_2, \gamma_2, \delta_2, \alpha_3, \beta_3, \gamma_3, \delta_3$

telles que les 8 vecteurs décrits par les lignes de la table 3.2 correspondent à des approximations linéaires biaisées de f. Comme $n = t+2$, les approximations linéaires biaisées de f sont

c	G_3	G_2	G_1
0	α_3	α_2	α_1
1	β_3	β_2	α_1
2	γ_3	α_2	β_1
3	δ_3	β_2	β_1
4	α_3	γ_2	γ_1
5	β_3	δ_2	γ_1
6	γ_3	γ_2	δ_1
7	δ_3	δ_2	δ_1

TAB. 3.2 – Exemple de configuration des groupes avec $m = 3$.

des fonctions qui impliquent soit les n variables de $J = \{j_1, \ldots, j_n\}$, soit toutes les variables de J sauf une. Notons I l'ensemble des indices $i \in J$ tels que $\mathcal{E}\left[f \oplus \bigoplus_{j \in J \setminus \{i\}}\right] = \pm\varepsilon \neq 0$. Supposons que I soit inclus dans un des groupes, par exemple le groupe m. Pour que les approximations linéaires définies par les 2^m vecteurs de la table soient biaisées, il faut que les vecteurs des colonnes correspondant aux groupes de 1 à $(m-1)$ soient toujours égaux au vecteur tout à 1. Par contre, les 2^{m-1} vecteurs du groupe m peuvent être pris dans l'ensemble des combinaisons linéaires possibles des variables du groupe m qui, additionnées à toutes les variables des autres groupes, donnent une approximation linéaire biaisée. Le nombre de possibilités pour chaque vecteur de la première colonne est donc exactement le nombre N d'approximations linéaires biaisées impliquant les variables de J. Le nombre de vecteurs a qui correspondent à un produit non nul dans la formule (3.3) est donc

$$N_s = N^{2^{m-1}},$$

ce qui signifie que dans ce cas, le biais de la relation de parité est $\mathcal{E}(PC_{f,\mathcal{T}}) = N^{2^{m-1}}\varepsilon^{2^m}$.

Si maintenant l'ensemble des variables I qui définit les approximations biaisées est réparti dans plusieurs groupes, par exemple les groupes 2 et 3, on peut voir que la seule valeur correcte possible à la colonne 1 est toujours le vecteur tout à un. Par contre, toute valeur différente du vecteur tout à un dans la colonne du groupe 3 impose que l'élément correspondant du groupe 2 soit le vecteur tout à un. La valeur de α_2 détermine donc celle de α_1 et de γ_1. On peut observer comment deux circuits fermés sont créés, le bleu-vert et

le noir-rouge. Dans ce cas, on obtient :

$$N_s = N^{2^{3-2}} = N^2.$$

Si les variables de l'ensemble I sont maintenant réparties dans les 3 groupes, le nombre de fois que le produit est différent de zéro sera :

$$N_s = N.$$

Quand m augmente, il y a d'autres possibilités de configurations des groupes où aucune de ces formules n'est applicable. Mais on peut alors calculer N_s partie par partie. Dans tous les cas, N_s peut être déterminé par un algorithme simple et il apparaît clairement que la valeur obtenue dans le cas où toutes les variables de I sont dans un seul groupe est la plus grande possible.

Nous venons donc de démontrer le théorème suivant.

Théorème 3.19. *Soit f une fonction booléenne plateau à v variables et t-résiliente. Soit N le nombre d'approximations linéaires biaisées que nous pouvons construire avec des variables parmi un ensemble $J = \{j_1, \ldots, j_n\}$ de taille $n = t+2$. Soit I l'ensemble des indices $i \in J$ tels que $\mathcal{E}\left[f \oplus \bigoplus_{j\in J\setminus\{i\}}\right] = \pm\varepsilon \neq 0$. Considérons la relation de parité $\boldsymbol{PC_{f,\mathcal{T}}}$ définie par $\mathcal{T} = \langle M_1, \ldots, M_m\rangle$ impliquant les n variables définies par l'ensemble J. Si I est inclus dans un seul groupe parmi les m définis par $\mathcal{T}$, alors le biais de la relation de parité est*

$$\mathcal{E}(PC_{f,\mathcal{T}}) = 2^{m-1}\varepsilon^{2^m}.$$

Par ailleurs, nous conjecturons que, pour tout m, ce biais est le plus grand que l'on puisse obtenir pour une relation de parité à 2^m termes. Nous avons pour l'instant démontré cette conjecture uniquement pour $m \leq 3$.

Conjecture 3.20. *Soit f une fonction plateau t-résiliente. Le biais de toute relation de parité avec 2^m termes qui utilise $t+2$ variables est, au plus $N^{2^{m-1}}\varepsilon^{2^m}$ où N est le nombre d'approximations biaisées de f qu'on peut construire avec ces $(t+2)$ variables et ε le biais associé. De plus, cette borne supérieure est atteinte si et seulement si la relation de parité est de la forme suivante*

$$PC_{f,\mathcal{T}}(t) = \sum_{\tau\in\mathcal{T}} f(x_1(t+\tau), \ldots, x_v(t+\tau)), \text{ avec } \mathcal{T} = \langle M_1, \ldots, M_m\rangle$$

où les périodes T_{j_i}, pour tous les indices j_i telles que $\bigoplus_{1\leq k\leq n, k\neq i} x_{j_k}$ est une approximation linéaire biaisée de f, divisent un unique M_ℓ.

Exemple 3.21. Considérons $f = x_1x_2 \oplus x_2x_3 \oplus x_1x_3$, qui est une fonction plateau à 3 variables 0-résiliente. Les approximations linéaires biaisées sont x_1, x_2, x_3 (avec biais $\frac{1}{2}$) et $x_1 + x_2 + x_3$ (avec biais $-\frac{1}{2}$). Nous allons calculer le biais $\mathcal{E}(PC_{f,\mathcal{T}})$ pour $\mathcal{T} = \langle T_1, T_2\rangle$. Il y a donc les variables x_1 et x_2 qui interviennent dans la relation de parité. Avec ces variables,

nous pouvons construire deux approximations biaisées, donc $N = 2$. Comme nous l'avons vu précédemment, dans ce cas, nous avons :

$$N_s = 2^{2^{2-2}} = 2.$$

Le biais sera donc

$$\mathcal{E}(PC_{f,\mathcal{T}}) = 2\varepsilon^{2^2} = 2^{-3}.$$

Quand $n = t + k$, avec k quelconque. Nous conjecturons également que le fait que $N^{2^{m-1}}$ soit le valeur maximale de N_s est valide pour toute valeur de n et pas seulement pour $n = t+2$. Mais en plus de cette conjecture on ne sait pas prouver que dans la formule (3.3), le produit des biais correspondant à chaque $a \in \mathbf{F}_2^{n2^{m-1}}$ est toujours positif. Donc on va supposer qu'il peut y avoir des cas où certains produits ont un biais qui est différent des autres. Ceci signifie que la valeur de $\mathcal{E}(PC)$ sera plus petite. On peut donc définir une borne supérieure correspondant à la valeur de $\mathcal{E}(PC)$ dans le cas des relations de parité où $n = t + k$:

$$\mathcal{E}(PC) \leq \varepsilon^{2^m} N_s$$

où N est le nombre d'approximations linéaires biaisées qu'on peut construire avec les n variables et $N_s \leq N^{2^{m-1}}$ d'après notre conjecture. On peut donc généraliser une partie de la conjecture précédente à $n = t + k$:

Conjecture 3.22. *Soit f une fonction plateau t-résiliente. Le biais de toute relation de parité avec 2^m termes qui utilise $(t+k)$ variables est, au plus $N^{2^{m-1}}\varepsilon^{2^m}$ où N est le nombre d'approximations biaisées de f qu'on peut construire avec ces $(t+k)$ variables et ε le biais associé.*

3.3.3 Exemples

Comment construire la meilleure relation de parité avec 8 variables pour une fonction plateau $6-$résiliente et calculer son biais. On va considérer comme fonction plateau la fonction booléenne G à 11 variables de Achterbahn-80, qui est $6-$résiliente. Le biais de toutes les approximations linéaires biaisées est $\pm 2^{-3}$. On va construire des relations de parité avec $n = 8$ variables. On considère tous les sous-ensembles possibles de 8 variables, et pour chacun d'eux, on regarde combien d'approximations linéaires peuvent être construites avec ces variables, c'est-à-dire combien de sous-ensembles de 7 ou 8 variables de l'ensemble de 8 variables considérées correspondent à une approximation linéaire biaisée.

On va utiliser l'exemple des variables d'indices $J = \{1, 4, 5, 6, 7, 9, 10, 11\}$. Pour simplifier on va représenter chaque sous-ensemble de $\{1, \ldots, n\}$ de ces variables par un vecteur de $\mathbf{F}_2^n$, ici en hexadécimal, où le bit le plus à droite représente x_1 et le plus à gauche x_{11}. L'ensemble J correspond ici donc à `0x779`. Dans notre cas, avec ces variables on peut construire les approximations suivantes :

$$x_1 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_9 \oplus x_{10} \oplus x_{11}, \text{ pour } \texttt{0x779}.$$

$$x_1 \oplus x_4 \oplus x_6 \oplus x_7 \oplus x_9 \oplus x_{10} \oplus x_{11}, \text{ pour } \texttt{0x769}.$$
$$x_1 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_9 \oplus x_{10} \oplus x_{11}, \text{ pour } \texttt{0x771}.$$
$$x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_9 \oplus x_{10} \oplus x_{11}, \text{ pour } \texttt{0x778}.$$

On peut maintenant répartir en 3 groupes ces variables de deux manières différentes, ce qui va donc nous définir deux relations de parité à 8 termes différents :

- Considérons d'abord le cas où les groupes sont $G_1 = \{1, 4, 5\}$, $G_2 = \{6, 7, 9\}$ et $G_3 = \{10, 11\}$.
  Ils définissent une relation de parité construite avec
  $$\mathcal{T} = \langle T_1T_4T_5, T_6T_7T_9, T_{10}T_{11}\rangle.$$
  On voit que la variable manquante dans les 3 approximations à 7 variables est toujours dans le groupe G_1. Ceci signifie le nombre N_s de fois où le produit
  $$\prod_{c=0}^{2^m-1} \mathcal{E}(f \oplus \varphi_{(\chi(c,a),0)})$$
  n'est pas nul, est égal à $N_s = 4^{2^{m-1}}$. Le biais final qu'on trouve pour cette relation de parité est :
  $$(2^{-3})^{2^m} \times 4^{2^{m-1}} = 2^{-16},$$
  qui coïncide avec le biais calculé par notre algorithme décrit à la section 3.3.1. Avant nos travaux, le calcul du biais de cette relation de parité nécessitait 2^{56} opérations, notre algorithme le calcule avec une complexité 2^{35}, et dans ce cas, on a été capable de le calculer à la main.
  Ceci est le meilleur biais qu'on peut obtenir pour une relation de parité à 8 variables construite pour la fonction de combinaison de Achterbahn-80.
- Considérons maintenant le cas où les groupes sont $G_1 = \{1, 9, 10\}$, $G_2 = \{5, 6, 7\}$ et $G_3 = \{4, 11\}$:
  Ils définissent une relation de parité construite avec
  $$\mathcal{T} = \langle T_1T_9T_{10}, T_5T_6T_7, T_4T_{11}\rangle.$$
  On obtient ici que l'approximation `0x769` correspond à une variable manquante dans G_2, `0x771` dans G_3 et `0x778` dans G_1. Ceci va signifier que le nombre de fois que $\prod_{c=0}^{2^m-1} \mathcal{E}(f \oplus \varphi_{(\chi(c,a),0)})$ n'est pas zéro va être plus petit que dans le cas précédent.
  Le biais final qu'on trouve ici est :
  $$(2^{-3})^{2^m} \times 108 = 2^{-17.25},$$
  où on va maintenant voir comment on a obtenu $N_s = 108$ partie par partie. L'idée est de regarder tous les cas en se ramenant aux cas plus simples à partir de la table 3.2.

Par exemple, on commence par regarder combien il y a de cas où tous les mots $\alpha_1, \ldots, \delta_1$ correspondant au groupe 1 sont différents du mot tout à 1. Le résultat est le premier terme de la somme (3.4) ci-dessous, 1, car dans ce cas, tous les mots pour les autres des groupes doivent être déterminés si on ne veut pas que le résultat du produit soit nul. Ensuite, on étudie combien il y a de cas où trois des mots $\alpha_1, \ldots, \delta_1$ sont différents du mot tout à 1, et un seul est égal à 1. On a 4 possibilités pour choisir lequel des mots vaut 1, mais une seule solution pour chacune, donc le deuxième terme est 4. On continue comme cela. Par exemple, le terme le plus compliqué, qui est la cinquième parenthèse, correspond au cas où tous les $\alpha_1, \beta_1, \gamma_1$ et δ_1 valent 1. Dans ce cas, il faut regarder récursivement ce qu'il se passe quand : les quatre mots $\alpha_2, \beta_2, \gamma_2$ et γ_2 sont différents de 1, trois sont différents de 1,... De cette façon, on peut calculer N_s et on obtient

$$N_s = (1) + (4) + (18) + (27) + (1 + 4 + 12 + 16 + 16) = 108. \tag{3.4}$$

Ce biais coïncide avec le biais exact qu'on a calculé.

3.3.4 Conclusion

Nous avons proposé ici une nouvelle vision des relations de parité qui détache, d'une certaine façon, les relations de parité des approximations, au contraire de ce que avait été fait jusqu'ici. L'impact le plus pratique et immédiat de nos travaux est la grande réduction de la complexité du calcul du biais des relations de parité dans tous les cas, le calcul à la main étant, dans des cas particuliers même possible dans des configurations où on avait avant une complexité non-réalisable.

Une des conclusions importantes que nous avons retenue est que nous pouvons calculer le biais des relations de parité à partir seulement des approximations linéaires de la fonction considérée. Effectivement, avec la transformée de Walsh de la fonction de combination f nous avons toute l'information nécessaire pour calculer le biais d'une relation de parité construite avec elle.

Nous avons proposé une nouvelle borne inférieure sur le biais qui nous donne de l'information même sur des relations de parité construites avec une approximation non biaisée, ceci étant le premier résultat connu dans cette situation. Dans le cas des fonctions t-résilientes, nous avons donné une formule simple pour calculer le biais de toute relation de parité construite avec $t+1$ variables, et ce biais dépend uniquement du biais de l'approximation linéaire associée. Dans le cas des fonctions plateau, nous avons obtenu plusieurs résultats. Dans le cas des relations de parité construites à partir de $t+2$ variables nous obtenons une borne supérieure sur le biais et nous montrons que cette borne peut être atteinte dans certains cas (il reste à prouver notre conjecture sur la valeur maximale de N_s). Un objectif futur est d'étendre ces résultats au cas $t+k$ avec $k > 2$. D'autres travaux à venir auront pour but de regarder s'il est possible de prouver des résultats similaires dans le cas des fonctions non-plateau. Nous avons montré comment construire les relations de parité avec $(t+2)$ variables à partir d'une fonction plateau avec le meilleur biais possible.

D'un autre côté, nous avons commencé à regarder, et cela reste donc un travail intéressant à faire, comment appliquer ces résultats sur d'autres types des relations de parité que celles traitées ici. Par exemple, nous pouvons donner une formule exacte du calcul du biais de la relation de parité utilisée dans les attaques avec des LFSRs à la place de NLFSRs, c'est-à-dire sur les relations de parité construites avec des multiples de petit poids des polynômes de rétroaction [MS88, CT00, MH04, EJ05, Did07, LC09]. Dans ce cas, nous pouvons, comme nous venons de montrer, au lieu d'appliquer le piling-up lemma, éliminer les dépendences entre variables avec des restrictions, et faire la substitution correspondant à la relation linéaire connue sur un bit. Comme ce sont des relations de parité avec beaucoup moins de termes que celles présentées ici (le nombre de termes n'est pas exponentiel en le nombre de variables à éliminer), ces résultats sont moins importants que ceux que nous venons de présenter, mais intéressants, et ils pourraient nous aider à mieux comprendre les similitudes entre les différentes relations de parité. Il reste à voir si on pourrait aussi l'appliquer à d'autres types de relations de parité que ceux mentionnés ici.

Conclusion

Nous nous sommes demandé quels enseignements on pouvait tirer des travaux de cette partie sur la conception et la cryptanalyse de chiffrements à flot. Les attaques sur Achterbahn peuvent être traduites en attaques génériques sur le modèle général de la combinaison des FSRs, donc elles vont nous permettre de dimensionner le système pour éviter que ce type d'attaque ne soit réalisable. D'un autre côté, ces attaques sont basées sur des compromis temps-données-mémoire, dans le sens où nous pouvons faire des attaques avec une très grande complexité en données et très petite en temps, ou l'inverse, et ce qu'on cherche est l'attaque qui nous donne les meilleures complexités en général. Par exemple, si on considère une fonction de combinaison f à n variables, nous pouvons construire une relation de parité avec le plus grand biais possible, $\varepsilon = 1$, et le plus petit nombre de termes dans la relation de parité :

$$f(t) + f(t + T_1 T_2 \dots T_n).$$

Mais cette relation de parité nécessite que $T_1 T_2 \dots T_n$ bits de la suite chiffrante soient produits pour pouvoir être calculée. Dans le cas d'Achterbahn comme de n'importe quel autre algorithme raisonnable, cette quantité est trop grande.

Par ailleurs, si nous regardons le nombre de variables de l'approximation qui ne sont pas incluses dans la relation de parité (c'est-à-dire les variables de l'approximation qu'on ne fait pas disparaître par la relation de parité mais par d'autres moyens comme une recherche exhaustive et la décimation), nous remarquons alors que, plus ce nombre est petit, plus le biais sera plus petit (donc plus il y aura de termes dans la relation de parité). Comme, d'une part, plus le biais est petit, plus la complexité en temps est grande, et que d'autre part, plus le nombre de variables non incluses est grand, plus la complexité en temps ou en données est grande, on doit trouver un compromis entre ces deux paramètres pour obtenir la meilleure complexité.

Si on se place dans le cas des fonctions de combinaison t-résilientes ce qui est le cas usuel en cryptographie,une question très naturelle est : pourrait-on monter de meilleures attaques de ce type contre Achterbahn avec plus de variables que $t+1$ dans l'approximation utilisée ? Nous allons voir comment nos résultats sur les relations de parité nous aident à répondre en partie à cette question. On va reprendre l'exemple d'Achterbahn où la fonction de combinaison est une fonction plateau. Pour simplifier, on suppose que la longueur de suite chiffrante est limitée de sorte que les termes de la relation de parité ne peuvent impliquer que 2 variables. Dans la conjecture 3.22, nous disons que le biais d'une relation

de parité construite avec $t+k$ variables va avoir un biais inférieur ou égal à $N^{2^{m-1}}\varepsilon^{2^m}$. Dans le cas de Achterbahn-80, $\varepsilon = 2^{-3}$. Comme nous l'avons dit, nous cherchons à améliorer la complexité en utilisant une relation de parité construite avec plus de variables. Nous allons donc considérer que le nombre de variables de l'approximation qui n'interviennent pas dans la relation de parité est 3, comme dans le meilleur cas avec $t+1$ variables, car, de cette manière, la complexité ne sera pas affectée de ce côté-là. Dans notre cas, nous avons $t+k = 2m+3$, où $2m$ sera le nombre de variables qui disparaissent dans la relation de parité, car nous avons supposé que chaque terme en contient deux. Alors $m = \lceil \frac{t+k-3}{2} \rceil$. Nous allons regarder si on peut construire une meilleure attaque à partir d'une approximation avec $t+2 = 8$ variables. Dans ce cas, $m = 3$. Si nous utilisons l'inégalité de la conjecture 3.20, qui est démontrée dans le cas $m = 3$, nous obtenons que cette attaque sera plus efficace si la relation de parité à un biais supérieur à celle pour $t+1$ variables, c'est-à-dire si :

$$2^{-3\times 4} < N^{2^{m-1}} 2^{-3\times 2^m},$$

et donc

$$N > 2^3$$

est une condition nécessaire (mais pas suffisante) pour trouver une attaque avec $t+2$ variables de meilleure complexité qu'avec $t+1$ variables. Nous savons que, à partir d'un sous-ensemble de $t+2 = 8$ variables, nous pouvons trouver, au plus, $t+3$ approximations linéaires biaisées. Mais, dans le cas d'Achterbahn-80, il n'existe aucun sous-ensemble de 8 variables avec lequel on peut construire 9 approximations linéaires biaisées. On déduit donc que nous ne pouvons pas construire une meilleure attaque de ce type avec $t+2$ variables.

Du point de vue de la conception des chiffrements à flot, nous pouvons aussi déduire, d'après notre technique pour construire les relations de parité avec le meilleur biais, une stratégie pour affecter, par exemple, la longueur des registres en fonction de la fonction de combinaison, afin de rendre la construction de bonnes relation de parité la plus compliquée possible. En tant qu'attaquant nous pouvons mieux choisir le compromis que nous allons adopter, puisque nous avons plus d'information « a priori » sur les relations de parité qu'avant.

Comme nous l'avons dit dans la section précédente, il y a encore beaucoup de choses à faire sur les relations de parité, et les exemples donnés ici montrent l'importance de ces travaux. Ils peuvent par exemple, aider à prouver qu'une attaque du même type que les nôtres (à condition qu'elle n'utilise pas de nouvelles techniques, mais seulement les techniques connues) est la meilleure attaque possible sur un certain algorithme.

Deuxième partie

Fonctions de hachage

Chapitre 4

La compétition SHA-3

4.1 Fonctions de hachage cryptographiques

L'étude des fonctions de hachage cryptographiques est une des trois grandes branches de la cryptographie symétrique. Les fonctions de hachage sont des fonctions qui, étant donné un message quelconque d'une longueur arbitraire, renvoient une valeur de longueur fixe, ℓ_h. Elles sont largement utilisées dans de nombreuses d'applications en sécurité informatique, comme dans les codes d'authentification de messages (MACs), signatures numériques et d'autres formes d'authentification. Elles doivent être faciles à calculer, et vérifier des propriétés particulières, parmi lesquelles, les trois les plus importantes sont :

- Trouver deux messages, $\mathcal{M}$ et $\mathcal{M}'$, qui donnent le même haché, $\mathcal{H}(\mathcal{M}) = \mathcal{H}(\mathcal{M}')$, doit être difficile. On dit alors que $\mathcal{H}$ est résistante aux collisions.
- À partir d'un message $\mathcal{M}$ et de son haché $\mathcal{H}(\mathcal{M})$, trouver un autre message, $\mathcal{M}'$, tel que $\mathcal{H}(\mathcal{M}) = \mathcal{H}(\mathcal{M}')$ doit être difficile. On dit que $\mathcal{H}$ est résistante à la recherche d'un deuxième antécédent.
- À partir d'un haché $\mathcal{H}$, il doit être difficile de trouver un message $\mathcal{M}$ tel que $\mathcal{H}(\mathcal{M}) = \mathcal{H}$. On dit que $\mathcal{H}$ est résistante à la recherche d'un antécédent.

Ces définitions sont assez informelles et il n'existe pas de consensus sur leur définition [RS04, Rog06]. Nous allons définir de deux façons ce qu'on considère comme « difficile », ce qui va nous conduire à deux types de résistances aux attaques : une au sens strict et une au sens faible. Afin de clarifier ces notions nous allons nous concentrer sur les fonctions de hachage itératives. Elles sont définies d'une part par une fonction de compression qui prend en entrée une valeur de chaînage et un bloc du message qu'on veut hacher, et qui sort une valeur qui sera la valeur de chaînage suivante, et d'autre part par un mode opératoire qui décrit comment itérer cette fonction de compression jusqu'à ce que tous les blocs qui forment le message soient introduits pour générer le haché.

4.1.1 Résistance au sens strict

On rappelle que la longueur du haché est ℓ_h. Dans ce cas, et d'après le paradoxe des anniversaires, un attaquant qui demande à hacher $2^{\ell_h/2}$ messages différents et aléatoires

va trouver une collision entre les hachés de deux de ces messages,$\mathcal{H}(\mathcal{M}_i) = \mathcal{H}(\mathcal{M}_j)$. Ceci est l'attaque générique qu'on va toujours pouvoir effectuer. Aussi, pour qu'une fonction de hachage soit résistante aux attaques par collision, on va demander qu'il n'existe pas d'attaque sur elle qui nécessite moins de calculs que l'attaque générique. Mais la formalisation de cette notion est difficile car aucun défi n'est donné à l'attaquant. Pour pouvoir parler de la complexité d'une telle attaque, il est nécessaire de « randomiser » le problème et de travailler avec une famille de fonctions de hachage. Ceci est possible en considèrant une valeur initiale de la variables de chaînage aléatoire, comme dans le document de soumission de Shabal [BCCM+08, page 114] ou en introduisant un sel comme dans la construction HAIFA [BD07]. On dira alors que la fonction de hachage résiste aux collisions s'il existe pas d'algorithme qui, en moins de $2^{\ell_h/2}$ appels à la fonction de hachage trouve une collision pour une valeur aléatoire de l'IV (ou du sel).

Si la variable de chaînage est aussi de taille ℓ_h, cette définition peut être énoncée de la façon suivante, qui est plus restrictive, et qu'on va donc garder comme définition générale : une fonction de hachage est dite résistante aux collisions si pour une fonction $\mathcal{H}$ choisie aléatoirement dans la famille, trouver deux messages $\mathcal{M}$ et $\mathcal{M}'$ tels que $\mathcal{H}(\mathcal{M}) = \mathcal{H}(\mathcal{M}')$ nécessite au moins $2^{\ell_h/2}$ appels à la fonction de compression. L'attaque générique dans le cas de la recherche d'un (deuxième) antécédent pour un haché aléatoire (resp. un message aléatoire) coûte 2^{ℓ_h} appels à la fonction. Donc, pour qu'une fonction soit résistante à la recherche d'un (deuxième) antécédent on ne doit pas pouvoir trouver un (deuxième) antécédent avec moins de 2^{ℓ_h} appels à la fonction de compression. Ce type d'attaques n'existent pas sur le standard SHA-2, dont nous allons parler plus tard.

4.1.2 Résistance au sens faible

Dans le cas de la résistance au sens faible, on va considérer comme le coût total de l'attaque le produit de la complexité en temps et de la complexité en mémoire. Soit C_{t_c} la complexité en temps de la meilleure attaque par collision et soit C_{m_c} sa complexité en mémoire. Pour que la fonction soit résistante aux collisions il faut que toute attaque vérifie $C_{t_c}C_{m_c} \geq 2^{\ell_h/2}$. Dans le cas de la résistance à la recherche d'un (deuxième) antécédent, où C_{t_p} et C_{m_p} sont respectivement les complexités en temps et mémoire de la meilleure recherche pour la fonction d'un (deuxième) antécédent, il faut que $C_{t_p}C_{m_p} \geq 2^{\ell_h}$. Il est clair que la complexité de ces attaques vérifiant des contraintes plus fortes, ces attaques seront plus puissantes et porteront plus atteinte à la fonction que celles définissant la notion de résistance au sens strict. Une polémique existe sur ce sujet. Certains pensent qu'il faut qu'une fonction de hachage soit résistante aux attaques au sens strict, et d'autres qu'il suffit qu'elle leur résiste au sens faible. Mon avis à ce sujet est que, si on peut disposer de fonctions résistantes au sens strict elles seront plus sûres. Aussi les attaques qui ne font que violer cette résistance, et non la résistance au sens faible, nous font remarquer des faiblesses de la fonction qui ne devraient pas exister et que l'auteur n'avait pas prévues dans la plupart des cas. Très souvent ces faiblesses mènent à d'autres attaques plus puissantes. Je pense donc que les attaques qui contredisent la résistance au sens strict doivent être prises en compte d'une certaine manière. Parmi les attaques qui contredisent la résistance

au sens faible, on peut aussi introduire deux sous-catégories : les attaques théoriques et les attaques pratiques, la différence principale entre les deux cas étant que la complexité dans le deuxième cas est réalisable. Nous verrons plus tard les classifications plus ou moins officielles que nous allons suivre. Il peut y avoir aussi d'autres précisions pour définir la complexité, par exemple s'il faut prendre en compte la complexité des précalculs, dans le cas où on suppose que l'attaque ne commence que quand l'attaquant connaît l'IV, mais on ne va pas entrer dans le détail ici.

Une dernière remarque sur la longueur ℓ_h est qu'elle doit être choisie de telle façon que les attaques génériques soient hors de portée. Des valeurs typiques de la taille de la sortie, en bits, sont 224, 256, 384 et 512.

4.1.3 Résistances à d'autres attaques

Les fonctions de hachage sont aussi supposées être résistantes aux multicollisions, presque-collisions, length extension attacks, distingueurs, et également, selon les classes auxquelles elles appartiennent, aux collisions libres ou semi-libres...

4.2 Les grandes constructions de fonctions de hachage

Nous allons voir ici très rapidement quelques notions très élémentaires sur des constructions de fonctions de hachage.

4.2.1 Construction de Merkle-Damgård

La construction de Merkle-Damgård [Dam89, Mer89] utilise une fonction de compression d'une façon itérative. Le message est divisé en blocs d'une longueur fixée. À chaque instant i le bloc de message M_i est introduit dans la fonction de compression, ainsi que la sortie de la fonction de compression précédente. Si c'est le premier bloc, une valeur initiale sera utilisée. La sortie de la fonction de compression à cet instant i sera alors utilisée comme entrée dans l'insertion de bloc suivante M_{i+1}, ou bien, si on vient d'insérer le dernier bloc, elle sera utilisée pour calculer le haché. La sortie du bloc précédent peut intervenir dans la fonction de compression suivante de plusieurs manières, comme celles spécifiées par les schémas Davies-Meyer, Matyas-Meyer-Oseas... mais nous n'allons pas entrer ici dans le détail de ces schémas. Pour la suite, quand nous en aurons besoin nous expliquerons les constructions utilisées en détail. Usuellement, le fait que le message soit paddé en utilisant sa longueur est appelé « Merkle-Damgård strengthening ». Il est important de remarquer ici que, si la fonction de compression est résistante aux collisions, la fonction de hachage le sera aussi. La construction de Merkle-Damgård est amplement utilisée, notamment dans les familles MD et SHA.

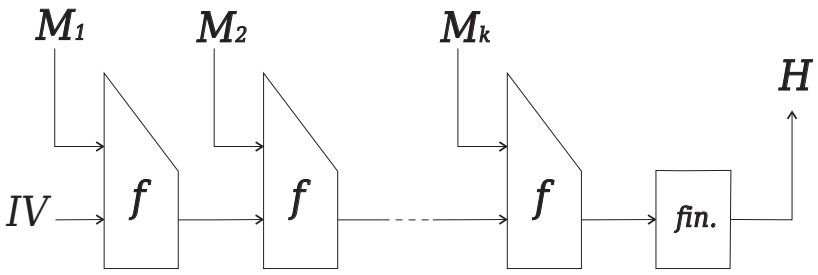

FIG. 4.1 – Vision générale de la construction de Merkle-Damgård

4.2.2 La famille SHA

SHA est l'acronyme de « Secure Hash Algorithm ». Il s'agit d'une famille de fonctions de hachage conçues par la NSA et publiées par le NIST. Comme on l'a déjà dit, elles utilisent une construction de Merkle-Damgård. SHA-0 est l'ancêtre de SHA-1. Des attaques ont été trouvées sur SHA-0 peu après la modification du standard SHA, principalement par Joux et al., et par Biham et Chen [CJ98, BC04, BCJ+05]. SHA-1 est la plus utilisée. Des attaques théoriques ont été trouvées sur SHA-1, notamment celles de Wang et al. [WYY05, WY05]. SHA-1 est le standard actuel ainsi que SHA-2. Il n'y a pas d'attaques connues sur SHA-2, mais à cause de sa ressemblance avec SHA-1, sa sécurité est remise en question.

4.2.3 La construction éponge

La construction éponge a été introduite récemment. Elle génère un haché à partir d'un message de taille variable. Le message est aussi divisé en blocs. Pour traiter les blocs de message on utilise principalement une transformation simple f, et deux étapes sont utilisées : l'étape d'absorption et l'étape de pressage. Pendant la première, le bloc de message est XORé à une partie de l'état interne, ces insertions sont entrelacées avec des applications de la transformation f. Pendant l'étape de pressage, des blocs de l'état interne peuvent être extraits, à des endroits séparés par des applications de la transformation f. Cette construction a été prouvée indifférentiable si la transformation f est aléatoire par Bertoni, Daemen and Peeters [BDPA08].

4.3 Le concours SHA-3 lancé par le NIST

Les fonctions de hachage actuellement standardisées ont subi plusieurs attaques et affaiblissements pendant ces dernières années : spécialement les attaques sur la famille MD4 de fonctions de hachage [WYY05, WY05]. Donc SHA-1 souffre d'attaques théoriques, même si personne n'a encore réussi à trouver une collision, et SHA-2 à cause sa ressemblance avec SHA-1, a perdu en grande partie la confiance des cryptologues. Elles nécessitent donc un remplaçant. C'est pour cette raison que le NIST, l'Institut National pour les Standards et la Technologie américain, a décidé de lancer une compétition [Nat07, Nat] pour trouver un successeur à ces standards : SHA-3. La date-limite pour soumettre était octobre 2008.

Une soixantaine de propositions ont été envoyées, et parmi celles-ci, 56 ont été jugées bien spécifiées et conformes aux contraintes imposées. Elles ont donc été retenues pour continuer la compétition dans la phase 1. En juillet 2009, 14 propositions ont été conservées, les autres étant éliminées. Ensuite, seulement 5 finalistes seront sélectionnés, pour n'en garder, à la fin, qu'un seul, qui sera le nouveau standard de hachage, SHA-3.

Parmi la soixantaine de candidats soumis il y a des constructions très différentes. Il y a des modes opératoires différents (Merkle-Damgård, constructions éponge, modes intermédiaires, wide pipe, narrow pipe...). Tous les candidats, les commentaires qu'ils ont suscités et les attaques, sont répertoriés sur la page « SHA-3 zoo [ECRb] », qui dépend de ECRYPT. Les différentes catégories d'attaques que nous avons évoquées y sont désignées par des couleurs : les attaques mettant à mal la résistance au sens strict dont on parlait dans la section précédente, sont signalées en jaune, et les attaques contre la résistance au sens faible sont en orange et rouge, où le rouge est réservé aux attaques pratiques. Le NIST a annoncé à la première conférence SHA-3 qu'il considérait les fonctions qui ne respectent pas la résistance au sens strict comme blessées, et celles qui ne respectent pas la résistance au sens faible comme cassées. Nous ne parlerons pas des candidats en détail ici, car on décrira dans les sections suivantes chacun des candidats sur lesquels on a travaillé.

Chapitre 5

Conception d'une fonction de hachage : Shabal

Dans le cadre de la compétition du NIST, j'ai travaillé depuis mars 2008 avec Emmanuel Bresson, Anne Canteaut, Benoît Chevallier-Mames, Christophe Clavier, Thomas Fuhr, Aline Gouget, Thomas Icart, Jean-François Misarsky, Pascal Paillier, Thomas Pornin, Jean-René Reinhard, Céline Thuillet et Marion Videau à la conception d'une fonction de hachage que nous avons soumis à la compétition SHA-3. Notre fonction s'appelle Shabal, et ce en l'honneur au rugbyman français Sébastian Chabal, très rapide, efficace et performant. On verra plus tard la description détaillée de cette fonction. Elle est une des plus rapides de la compétition et a une sécurité prouvée, puisque la sécurité de son mode opératoire était prouvée lors de sa soumission. Après la publication de certains distingueurs qui ne compromettent pas sa sécurité, nous avons intégré l'existence de ces distingueurs à la preuve. Mon apport principal à Shabal, en plus d'avoir très activement participé à la conception du mode et de la permutation utilisée dans Shabal a été l'analyse de la sécurité des différentes versions que nous avons conçues, en cherchant des cryptanalyses. Suite à des attaques que j'ai trouvées avant la soumission pour des paramètres modifiés de Shabal, nous avons rajouté une mise à jour finale que j'ai dû choisir optimale, comme on le verra dans une des sections suivantes. Après la soumission, je suis naturellement les commentaires qui paraissent sur Shabal ; j'ai aussi amélioré les distingueurs existants sur la permutation interne, qu'on a pu intégrer dans la nouvelle preuve de sécurité, et qui ne menacent en aucun cas la sécurité de Shabal. Après une description détaillée de Shabal, j'expliquerai les principaux aspects où je suis intervenue dans la justification de la conception. Puis, je décrirai les attaques qui nous ont décidés à rajouter une mise à jour finale, les derniers distingueurs que j'ai trouvés sur la permutation et pourquoi ils n'affectent pas la sécurité de Shabal.

5.1 Description de Shabal

Pendant la rédaction de cette thèse, les candidats de la compétition du NIST qui sont passés au deuxième tour ont été publiés, Shabal étant l'un d'entre eux. Je tiens à dire ici, puisque je ne vais pas m'intéresser aux implémentations, que Shabal a de très bonnes performances sur tous les plateformes. Concrètement, en software, elle a une vitesse d'approximativement 8 cycles par octet. Shabal est parmi les 3 candidats les plus rapides du deuxième tour de la compétition, comme au premier tour. Dans cette section je vais décrire comment fonctionne notre algorithme, Shabal ainsi que les raisons pour lesquelles on a effectué ces choix. Je présenterai la description générale sans me préocuper des descriptions destinées aux implémentations qui correspondent à des versions plus performantes. Tout d'abord je vais présenter quelques aspects généraux que je vais utiliser tout au long de la description.

Notations.

Dans cette section, nous allons introduire des notations de base. Soient x et y des mots de n bits, avec $n = 32$ pour Shabal . Alors $x \oplus y$ est le ou exclusif (XOR) bit à bit de x et y. Par $x \wedge y$ on représente le et logique bit à bit de x et y. On représente le complément de x par $\overline{x}$. Finalement, $x \lll j$ désigne la rotation de x de j bits à gauche et $x \ll j$ désigne le décalage de x de j bits à gauche. La différence entre rotation et décalage est que dans la rotation, les bits qui disparaissent par la gauche reviennent à droite, alors que dans le décalage, ils ne reviennent pas et les nouveaux bits qui arrivent à droite sont des zéros. L'addition modulo 2^{32} sera représentée par $\boxplus$, c'est-à-dire, si X et Y sont des vecteurs de mots de 32 bits, $X \boxplus Y$ sera le vecteur de mots qui contient les mots de X et de Y sommés entre eux sans retenue d'un mot au suivant. On utilise une notation similaire pour la soustraction.

5.1.1 Description du mode opératoire

Le mode opératoire de Shabal utilise une permutation paramétrée $\mathcal{P}$ et il est prouvé être indifférentiable d'un oracle aléatoire.

Soit ℓ_h la longueur de la sortie de Shabal-ℓ_h. Nous supposons que cette longueur appartient à l'ensemble $\{192, 224, 256, 384, 512\}$, pour simplifier, mais il est possible aussi de générer toutes les longueurs de sortie souhaitées plus petites que 512 avec une troncature.

Shabal utilise un état interne qui est divisé en trois parties : $(A, B, C) \in \{0,1\}^{\ell_a} \times \{0,1\}^{\ell_m} \times \{0,1\}^{\ell_m}$. Pour les initialiser, on leur donne des valeurs initiales (A_0, B_0, C_0). Un buffer auxiliaire $W \in \{0,1\}^{64}$ est utilisé comme compteur pour le nombre de blocs du message. Cette partie ayant un rôle particulier, elle ne sera pas comptée comme faisant partie de l'état interne. Shabal hache des blocs de message de ℓ_m bits d'une façon itérative. Comme nous l'avons dit précédemment, Shabal utilise une permutation paramétrée $\mathcal{P}$ avec $\mathcal{P} : \{0,1\}^{\ell_m} \times \{0,1\}^{\ell_a} \times \{0,1\}^{\ell_m} \times \{0,1\}^{\ell_m} \to \{0,1\}^{\ell_a} \times \{0,1\}^{\ell_m}$. Par définition, pour tous

les paramètres $(M, C) \in \{0,1\}^{\ell_m} \times \{0,1\}^{\ell_m}$, la fonction

$$\mathcal{P}_{M,C} : (A, B) \mapsto \mathcal{P}_{M,C}(A, B) = \mathcal{P}(M, A, B, C)$$

est une permutation.

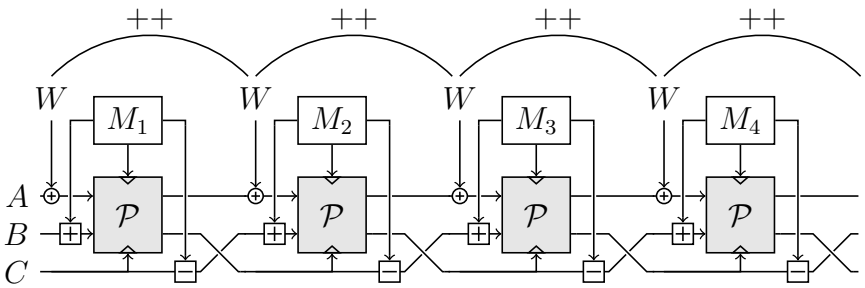

FIG. 5.1 – Le mode opératoire : tours d'insertion du message

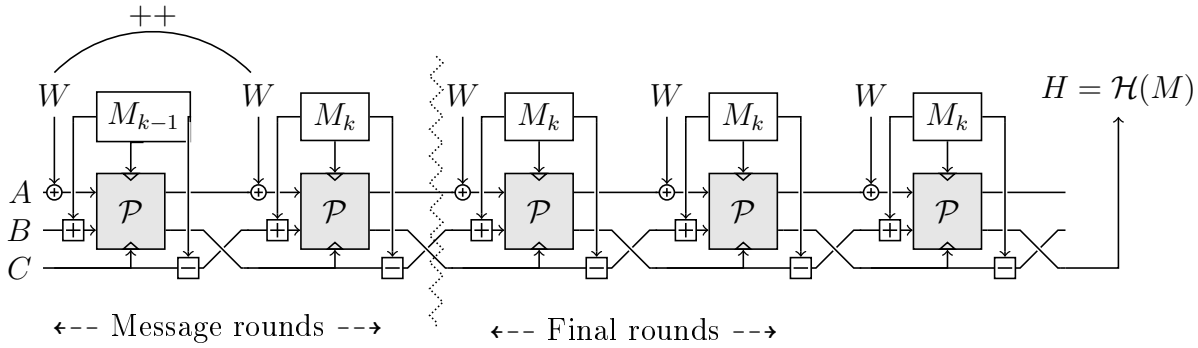

FIG. 5.2 – Tours finaux : première représentation

Nous pouvons voir une représentation graphique à la Figure 5.1. Une fois que le dernier bloc de message a été introduit, les tours finaux du mode opératoire se déroulent comme suit : le dernier bloc est stocké et réintroduit dans la fonction, trois fois, comme s'il s'agissait de 3 nouveaux blocs, à l'exception du fait que le compteur n'est plus incrémenté. Après la troisième insertion, le fils B qui sort directement de $\mathcal{P}$ sera le haché, ou, si nécessaire, on tronquera sa valeur pour produire le haché. Comme le bloc de message est le même dans ces trois derniers tours, les plus et les moins qui apparaissent peuvent s'éliminer, les tours finaux se simplifiant alors comme le montre la Figure 5.3

La fonction de tour, $\mathcal{R}$, est une fonction qui prend en entrée un état interne S, un bloc de message M_w, et un compteur w. Elle applique la permutation $\mathcal{P}$ paramétrée par C et M_w à l'entrée formée par $A \oplus w$ et $B \boxplus M_w$. Elle donne en sortie un nouvel état interne $S' = (A', B', C') = (A', C \boxminus M_w, C')$, où A' et C' sont les deux sorties de la

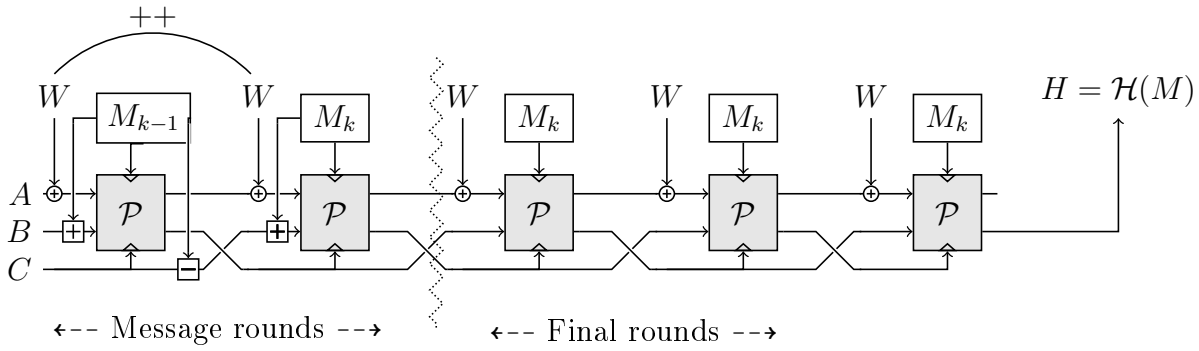

FIG. 5.3 – Tours finaux : deuxième représentation

permutation $\mathcal{P}$. Elle incrémente aussi le compteur de un, $w \leftarrow w + 1$. Nous pouvons dire que $(S', w + 1) = \mathcal{R}(M_w, S, w)$.

Nous allons donner ici un point de vue plus synthétique sur Shabal.

Sur la preuve de sécurité.

Nous n'allons pas détailler ici la preuve du mode opératoire utilisé, nous invitons le lecteur à la lire dans la documentation de la soumission de Shabal. On ne va pas non plus détailler la preuve du mode hors du modèle du chiffrement idéal, mais dans le cas où la permutation utilisée, $\mathcal{P}$, n'est pas considérée idéale et où il existe certains distingueurs sur elle [BCCM+09b]. On va seulement énoncer les conséquences de ces preuves : Shabal est

- indifférentiable d'un oracle aléatoire,
- résistant aux attaques par collision et
- résistant aux attaques par recherche d'antécédent et de deuxième antécédent,

si on considère la permutation $\mathcal{P}$ comme un chiffrement idéal et aussi, si on considère $\mathcal{P}$ comme une permutation paramétrée pour laquelle il existe des distingueurs « à clés liées », comme on le verra plus tard.

5.1.2 L'initialisation et la permutation interne

Dans la section précédente, nous avons décrit le mode opératoire sur lequel Shabal est basé. Nous allons maintenant décrire la permutation interne. Shabal n'utilise que des blocs de message de $\ell_m = 512$ bits. Comme demandé par le NIST, Shabal possède 2 paramètres ajustables, qui sont $p \geq 2$ et $r \geq 2$. Nous allons voir à quoi ils servent. L'état interne (A, B, C), correspond à un buffer de taille $(1024 + 32r)$ bits, vu comme un ensemble de vecteurs de mots de 32 bits. En fait, B et C sont des vecteurs de 16 mots et A est un vecteur de r mots. Ceci signifie que $\ell_a = 32r$. Le compteur w, qui n'est pas considéré comme une partie de l'état interne, est vu comme un vecteur de deux mots. Shabal est alors défini comme suit.

Algorithme 5 Réprésentation schématique de Shabal.

Initialisation : $(A, B, C) \leftarrow (A_0, B_0, C_0)$

Tours d'insertion du message : $M = M_1, \ldots, M_k$

Pour w de 1 à k **faire**

1. $B \leftarrow B \boxplus M_w$
2. $A \leftarrow A \oplus w$
3. $(A, B) \leftarrow \mathcal{P}_{M_w,C}(A, B)$
4. $C \leftarrow C \boxminus M_w$
5. $(B, C) \leftarrow (C, B)$

fin du faire

Tours finaux :

Pour i de 0 à 2 **faire**

1. $B \leftarrow B \boxplus M_k$
2. $A \leftarrow A \oplus k$
3. $(A, B) \leftarrow \mathcal{P}_{M_k,C}(A, B)$
4. $C \leftarrow C \boxminus M_k$
5. $(B, C) \leftarrow (C, B)$

fin du faire

Sortie : $H = \mathrm{msb}_{\ell_h}(C)$

Initialisation.

La valeur initiale de w est -1. De cette façon, une fois qu'on a traité le préfixe qui possède deux blocs, l'indice du premier bloc du message sera $w = 1$. Ce préfixe sera noté (M_{-1}, M_0) où $M_{-1}[0] = \ell_h$, $M_{-1}[15] = \ell_h + 15$, $M_0[0] = \ell_h + 16$ et $M_0[15] = \ell_h + 31$. Comme option, nous pouvons précalculer les valeurs de l'état interne après l'insertion de ces deux blocs de préfixe, et considérer les valeurs obtenues comme le nouvel IV. Cette dernière option est équivalente à la précédente.

La permutation paramétrée $\mathcal{P}$.

Nous allons voir maintenant la description de la permutation paramétrée $\mathcal{P}$ de Shabal. Pour ceci, nous allons utiliser une construction basée sur des NLFSRs (Figure 1.2).

Les fonctions non linéaires utilisées sont les suivantes :

$$\mathcal{U} : x \mapsto 3 \times x \bmod 2^{32} \text{ et } \mathcal{V} : x \mapsto 5 \times x \bmod 2^{32}.$$

Algorithme 6 La permutation paramètrée $\mathcal{P}$.

Entrées : M, A, B, C
Sorties : A, B

Pou i de 0 à 15, faire :
– $B[i] \leftarrow B[i] \lll 17$
i suivant

Pour j de 0 à $p-1$, faire :
– Pour i de 0 à 15, faire :
  – Calculer

$$\begin{aligned} A[i+16j \bmod r] \leftarrow\ & \mathcal{U}\big(A[i+16j \bmod r] \oplus \mathcal{V}(A[i-1+16j \bmod r] \lll 15) \\ & \oplus C[8-i \bmod 16]\big) \\ & \oplus B[i+o_1 \bmod 16] \\ & \oplus (B[i+o_2 \bmod 16] \wedge \overline{B[i+o_3 \bmod 16]}) \\ & \oplus M[i] \end{aligned}$$

    où $(o_1, o_2, o_3) = (13, 9, 6)$
  – $B[i] \leftarrow (B[i] \lll 1) \oplus \overline{A[i+16j \bmod r]}$
– i suivant
j suivant

Pour j de 0 à 35, faire :
– $A[j \bmod r] \leftarrow A[j \bmod r] + C[j+3 \bmod 16]$
j suivant

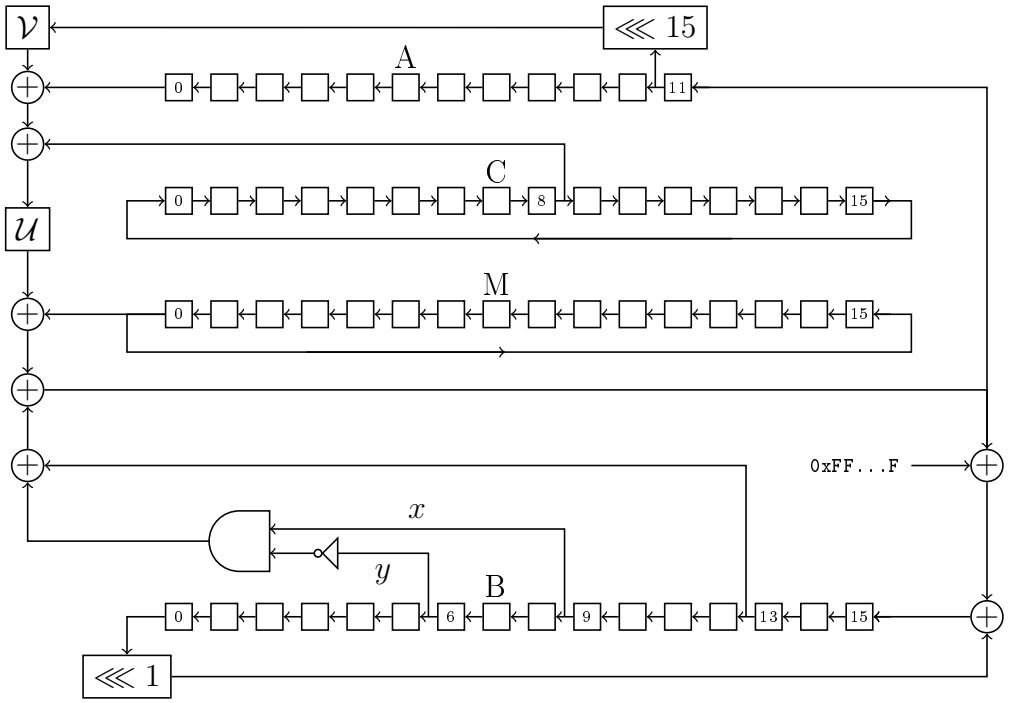

FIG. 5.4 – Structure principale de la permutation paramètrée $\mathcal{P}$ utilisée dans Shabal.

Les valeurs des offsets $(o_1, o_2, o_3) = (13, 9, 6)$ ont été choisies pour être optimales, comme on le verra plus tard. Les paramètres (p, r) pourraient prendre plusieurs valeurs pour atteindre la sécurité souhaitée, mais, notre choix pour la définition de Shabal a été de les fixer à deux valeurs spécifiques : $p = 3$ et $r = 12$.

La dernière mise à jour de A, qui est la dernière action de $\mathcal{P}$ et n'est pas représentée à la Figure 5.4, est la suivante : $A[j \bmod r] \leftarrow A[j \bmod r] + C[3 + j \bmod 16]$. Elle n'est pas généralisable à une valeur quelconque du paramètre r, mais elle a été choisie par recherche exhaustive comme la meilleure mise à jour, comme nous allons le voir plus tard. Si r devait être changé, une nouvelle recherche devrait être menée pour redéfinir cette mise à jour.

Les paramètres ajustables

Shabal comme nous l'avons déjà dit, a deux paramètres ajustables. On va ici les définir précisément et évaluer l'influence qu'ils ont sur la sécurité :

Paramètre p : c'est le nombre de tours complets du registre B effectués pour chaque appel de $\mathcal{P}$. Plus p est grand, plus la sécurité est grande et plus la performance en vitesse se dégrade. Augmenter p a pour effet de renforcer la permutation paramétrée. En tous cas, il faut toujours que $p \geq 2$. Pour Shabal on a choisi $p = 3$.

Paramètre r : c'est la rémanence de A. La plus petite valeur de r est 2 pour permettre l'insertion du compteur W de 64 bits à chaque tour sur $A[0]$ et $A[1]$. Analyser l'effet

de r sur la sécurité est un peu plus complexe que pour p. D'un côté, r représente une marge de sécurité car de lui dépend la taille totale de l'état interne et donc, plus r est grand, plus l'état interne qu'il faut contrôler dans une attaque est grand. Ça se traduit en une plus grande capacité, notion définie dans [BDPA07]. En même temps, un r trop grand peut aboutir à une très faible diffusion, par exemple des états où on ne réutilise pas les mots de A et, dans ce cas, la sécurité serait gravement affaiblie. Une borne supérieure de r est $16p$.

Comme nous avons déjà dit, dans Shabal nous avons fixé $(p, r) = (3, 12)$.

Nous allons maintenant essayer d'expliquer comment et pourquoi on a fait ces choix dans la construction de Shabal. Je vais surtout parler et expliquer en détail les parties auxquelles j'ai le plus participé.

5.2 Le mode opératoire et ses versions successives

Avant d'arriver au mode opératoire de Shabal, de nombreuses idées ont été proposées, étudiées et éliminées. Nous allons voir quelques considérations sur le mode qu'on a dû prendre en compte afin de fixer les tailles minimales des paramètres qui y interviennent. Ceci a été une longue recherche de plusieurs mois, jusqu'à ce que nous trouvions un mode qui nous convenait. Je tiens à préciser que nous étions un groupe de concepteurs très nombreux, et arriver à ce que tout le monde soit d'accord n'a pas été simple.

5.2.1 Ancien mode numéro 1 et attaques par recherche d'un deuxième antécédent

La première version du mode opératoire de Shabal est celle représentée à la figure 5.5. L'idée d'utiliser une insertion de message multiple n'est pas nouvelle : elle est employée par exemple dans RadioGatun [BDAP06]. Un effet direct de la double insertion est d'augmenter la diffusion des différences en entrée, ce qui complique la recherche de chemins différentiels. Toutefois, il est naturel de s'interroger sur la complexité des attaques par recherche d'un deuxième antécédent contre ce type de construction. En effet, toute fonction de hachage de type Merkle-Damgård dont l'état interne est de taille s est vulnérable à une attaque générique par recherche d'un deuxième antécédent de complexité $2^{\frac{s}{2}}$.

Plus précisément, on peut trouver un deuxième antécédent de tout message de 2^k blocs avec une complexité 2^{s-k}. En effet, pour tout message $\mathcal{M} = M_1 \dots M_{2^k}$, on considère la liste $\mathcal{S} = \{S_1, \dots, S_{2^k}\}$ des états internes de la fonction après chaque itération. On choisit ensuite 2^{s-k} messages de longueur ℓ, $\mathcal{M}^* = M_1^* \dots M_\ell^*$ et on calcule l'état interne de la fonction après insertion de chacun d'eux. Alors, il existe (avec probabilité supérieure à $1/2$) un des 2^{s-k} messages $\mathcal{M}^*$ qui aboutit à un des 2^k états internes, S_i. Le message $\mathcal{M}' = \mathcal{M}^* M_{i+1} \dots M_{2^k}$ conduit donc au même haché que $\mathcal{M}$ [MvOV97, Fact 9.37].

De plus, Kelsey et Schneier ont montré récemment qu'il était possible de mener une attaque de complexité similaire même dans le cas où le message est encodé par la procédure

MD-strenghtening, notamment quand il est facile de trouver des points fixes de la fonction de compression [KS05]. C'est le cas par exemple de la construction Davies-Meyer.

Il est donc légitime de se demander si, dans le cas particulier de la construction Merkle-Damgård où l'on spécifie la manière dont le message est inséré, comme dans le cas des éponges ou du mode de la figure 5.5 par exemple, il est possible de diminuer la complexité de cette attaque générique par recherche d'un deuxième antécédent.

Mode opératoire.

Nous décrivons tout d'abord l'attaque sur le mode opératoire général suivant, qui inclut à la fois les constructions de type éponge et l'ancienne version numéro 1 du mode opératoire de Shabal décrite à la figure 5.5.

L'état interne est constitué de deux parties, A et B, de tailles respectives ℓ_m et $\gamma = \ell_s - \ell_m$. La quantité γ correspond à la « capacité » dans le cas d'une construction éponge, c'est-à-dire au paramètre défini dans [BDPA07] pour mesurer la sécurité : un mode opératoire a une capacité γ s'il est indifférentiable d'un oracle aléatoire avec moins de $2^{\frac{\gamma}{2}}$ évaluations de la fonction de compression. Ceci implique que générer une collision interne doit coûter au moins $2^{\frac{\gamma}{2}}$ évaluations de la fonction de compression, car trouver une collision interne servirait pour distinguer la fonction de hachage d'un oracle aléatoire.

À chaque itération, le bloc de message est inséré sur la partie a par une loi de groupe

$$A \leftarrow A \odot M_i$$

et il est inséré sur la partie B au moyen d'une fonction $\mathcal{I}_B$ exo-associative par rapport à $\odot$, c'est-à-dire telle que

$$\mathcal{I}_B(\mathcal{I}_B(B, M_1), M_2) = \mathcal{I}_B(B, M_1 \odot M_2)$$

pour tout B et tout couple de messages (M_1, M_2).

Les éponges et le mode décrit à la figure 5.5, par exemple, suivent ce modèle.

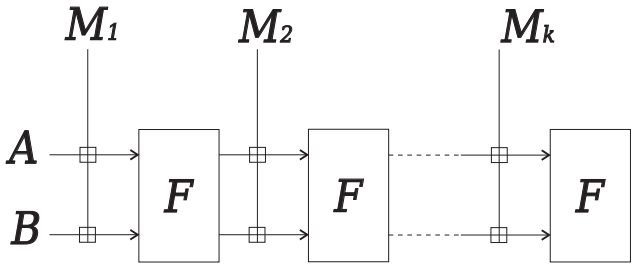

FIG. 5.5 – Ancienne version numéro 1 du mode opératoire de Shabal

On s'intéresse maintenant successivement aux cas suivants :
- F est inversible et dépend éventuellement d'un compteur de blocs ;
- F n'est pas inversible et ne dépend pas d'un compteur de blocs ;
- F n'est pas inversible et dépend d'un compteur de blocs.

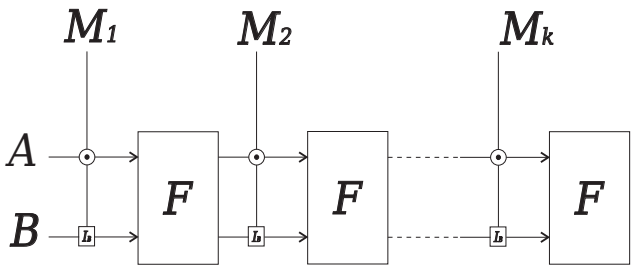

FIG. 5.6 – Mode général incluant les éponges et l'ancienne version numéro 1 du mode de Shabal

Cas où la fonction F est inversible.

On peut trouver un second antécédent de tout message $\mathcal{M}$ en complexité $2^{\frac{\gamma}{2}}$ 5.7.

Considérons une liste $\mathcal{L}_1$ de $2^{\frac{\gamma}{2}}$ messages $\mathcal{M}_1 = M_1 \dots M_\ell$, de taille ℓ avec $\ell \geq \frac{\gamma}{2\ell_m}$. Pour chaque $\mathcal{M}_1$, on note (A_ℓ, B_ℓ) l'état interne de la fonction après insertion de $\mathcal{M}_1$. Puis, on forme la liste $\mathcal{B}_1$ des valeurs de B obtenues après insertion d'un bloc supplémentaire $M_{\ell+1} = A_\ell^{-1} \odot A_0$ où x^{-1} est l'inverse de x pour la loi $\odot$ et A_0 est la partie A de l'état initial (on peut remplacer A_0 par n'importe quelle autre constante).

Considérons maintenant S l'état interne final obtenu en calculant $h(\mathcal{M})$. On considère une liste $\mathcal{L}_2$ de $2^{\frac{\gamma}{2}}$ messages de taille ℓ, $\mathcal{M}_2 = M'_1 M'_2, \dots, M'_\ell$. On calcule l'état interne (A'_0, B'_0) antécédent de S pour chacun de ces messages puis $(\alpha, \beta) = F^{-1}(A'_0, B'_0)$. On choisit ensuite un bloc $M'_0 = A_0^{-1} \odot \alpha$ et on fait la liste $\mathcal{B}_2$ des valeurs $\mathcal{I}_B^{-1}(\beta, M'_0)$. Il existe alors, avec une probabilité au moins 1/2, une collision entre $\mathcal{B}_1$ et $\mathcal{B}_2$, c'est-à-dire, un couple de messages $M_1 \dots M_{\ell+1}$ et $M'_0 M'_1 \dots M'_\ell$ qui conduisent au même état interne. Le message $\mathcal{M}' = M_1 \dots (M_{\ell+1} \odot M'_0) M'_1 \dots M'_\ell$ a donc le même haché que le message initial $\mathcal{M}$. On utilise ici le fait que l'insertion du message sur la partie B est exo-associative par rapport à la loi $\odot$, ce qui permet d'identifier la composition de l'insertion de $M_{\ell+1}$ et de celle de M'_0 à l'insertion d'un unique bloc, $M_{\ell+1} \odot M'_0$.

Il est important de noter que cette attaque fonctionne également

- si chaque itération de F est paramétrée par un compteur de blocs ;
- si on utilise un padding de type MD-strenghtening (il suffit de prendre ℓ de la taille souhaitée).

Cas où la fonction F n'est pas inversible.

On peut trouver un second antécédent de tout message $\mathcal{M}$ de taille 2^k en complexité $2^{\gamma-k}$ quand il n'y a ni padding, ni compteur.

On a un message $\mathcal{M}$ donné ayant au moins 2^k blocs, $M_1, \dots, M_{2^k}$. En notant A_i la valeur de la partie A de l'état après la $i$ème itération de F, on peut décomposer l'insertion de chaque M_i, $i \geq 2$, sur la partie A en deux insertions, celle de $M'_i = (A_{i-1})^{-1} \odot A_0$ et celle de $M''_i = (M'_i)^{-1} \odot M_i$. Pour que l'insertion de M'_i suivie de celle de M''_i corresponde

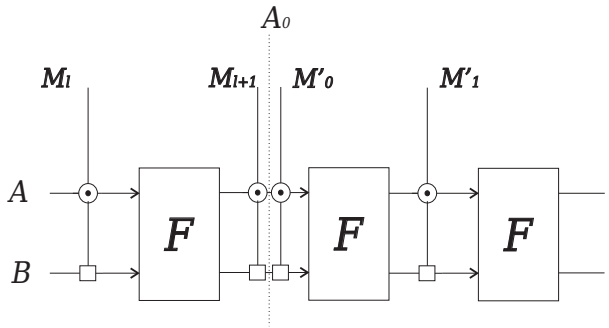

FIG. 5.7 – Procédure de l'attaque

à l'insertion de M_i sur la partie B de l'état interne, il faut donc que, pour tout B,

$$\mathcal{I}_B(\mathcal{I}_B(B, M_i')), M_i'') = \mathcal{I}_B(B, M_i' \odot M_i'') = \mathcal{I}_B(B, M_i),$$

ce qui est le cas puisqu'on a supposé que la fonction $\mathcal{I}_B$ est exo-associative par rapport à $\odot$. De cette manière, nous avons fait en sorte que A soit toujours égal à sa valeur initiale A_0 entre ces deux insertions (c'est-à-dire, à l'endroit défini par les lignes pointillées sur la figure 5.8). On constitue alors la liste $\mathcal{B}_1$ des 2^k valeurs de B obtenues aux endroits désignés par les lignes pointillées.

On considère maintenant $2^{\gamma-k}$ messages $\mathcal{N}$ de taille ℓ avec $\ell \geq \frac{\gamma}{2\ell_m}$. Pour chaque $\mathcal{N}$, on note (A_ℓ, B_ℓ) l'état interne de la fonction après insertion de $\mathcal{N}$. Puis, on forme la liste $\mathcal{B}_2$ des valeurs de B obtenues après insertion d'un bloc supplémentaire $N_{\ell+1} = A_{\ell-1}^{-1} \odot A_0$. On trouve donc de cette manière un couple de valeurs de $\mathcal{B}_1$ et $\mathcal{B}_2$ identiques. Le message

$$\mathcal{M}' = \mathcal{N}, (N_{\ell+1} \odot M_i''), M_{i+1}, \ldots, M_{2^k}$$

va donc avoir le même haché que $\mathcal{M}$. On a donc une attaque qui trouve un deuxième antécédent du message de longueur 2^k avec une complexité de $2^{\gamma-k}$.

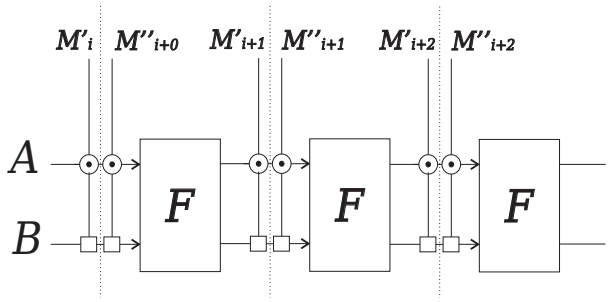

FIG. 5.8 – Recherche d'un deuxième antécédent quand F n'est pas inversible

Cas où F n'est pas inversible et paramétrée par un compteur.

On suppose maintenant que la fonction F est paramétrée par un compteur de blocs W représenté par un mot de taille ℓ_w. Alors, pour tout message $\mathcal{M}$ de $2^{\frac{\gamma+\ell_w}{2}}$ blocs avec $\gamma \geq \ell_w$,

on peut trouver un message $\mathcal{M}'$ ayant le même haché avec une complexité $2^{\frac{\gamma+\ell_w}{2}}$. Il suffit en effet de trouver simultanément une collision sur B et sur le compteur.

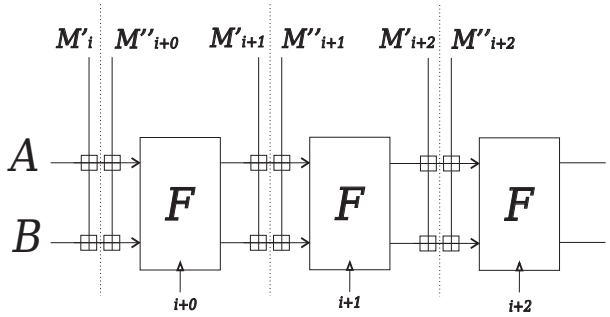

FIG. 5.9 – Représentation de l'attaque avec compteur

Une attaque consiste à construire comme précédemment $2^{\frac{\gamma+\ell_w}{2}}$ points où $A = A_0$. On forme alors les 2^{ℓ_w} listes de valeurs de B, chaque liste correspondant à une valeur donnée du compteur. L'objectif est alors de trouver une collision dans au moins une des listes. Chaque liste est donc de taille $2^{\frac{\gamma-\ell_w}{2}}$. La probabilité qu'il n'y ait pas de collision dans une liste de taille $2^{\frac{\gamma-\ell_w}{2}}$ est de l'ordre de

$$P_{\text{fail}} \simeq \exp(-\frac{2^{\gamma-\ell_w}}{2^{\gamma+1}}) = \exp(-2^{-(\ell_w+1)}).$$

La probabilité qu'il y ait une collision dans aucune des 2^{ℓ_w} listes est donc

$$(P_{\text{fail}})^{2^{\ell_w}} \simeq \exp(-2^{-(\ell_w+1)} \cdot 2^{\ell_w}).$$

Cas où F n'est pas inversible avec MD-strenghtening.

Par la même technique que pour l'insertion d'un compteur, on obtient une attaque en seconde préimage en $2^{\frac{\gamma+\ell_w}{2}}$ où ℓ_w est la taille du compteur qui stocke le nombre de blocs. Toutefois, on peut a priori utiliser la méthode de Kelsey et Schneier permettant de trouver des messages expansibles pour diminuer la complexité de l'attaque.

En résumé :

Dans le cas du mode opératoire de la figure 5.5 même avec une fonction F non inversible, par exemple avec la construction Davies-Meyer, on obtient une attaque qui trouve un deuxième antécédent avec une complexité $2^{\gamma+\ell_w-k}$. Pour un compteur de $\ell_w = 64$ bits il faut donc une capacité γ qui soit au moins égale à deux fois la taille de la sortie, i.e. au moins 1024 bits.

5.2.2 Evolutions vers le mode actuel

Les attaques par recherche d'un deuxième antécédent que je viens de décrire nous obligent donc à avoir un état interne relativement gros, ce qui rendait la conception de la permutation $\mathcal{P}$ difficile et son implémentation coûteuse. Nous avons donc fait évoluer le mode en remplaçant la permutation sur tout l'état (comme dans les éponges) par une permutation sur une partie de l'état, paramétrée par le reste de l'état. L'intérêt d'utiliser une permutation $\mathcal{P}$ qui prend en paramètre une des parties de l'état interne (i.e. un des fils), à la place d'une grosse permutation qui agit sur tout l'état interne comme à la figure 5.10 est, premièrement, de meilleures performances, car la permutation peut être plus rapide à calculer. Ensuite, elle permet aussi une meilleure diffusion sur ses sorties. Ceci nous a donc conduits à la version numéro 3 du mode opératoire, décrite à la figure 5.11.

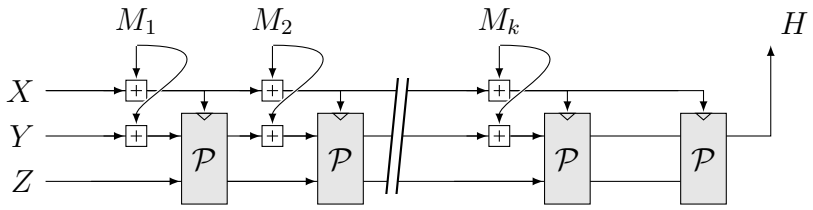

FIG. 5.10 – Ancienne version 2 du mode opératoire

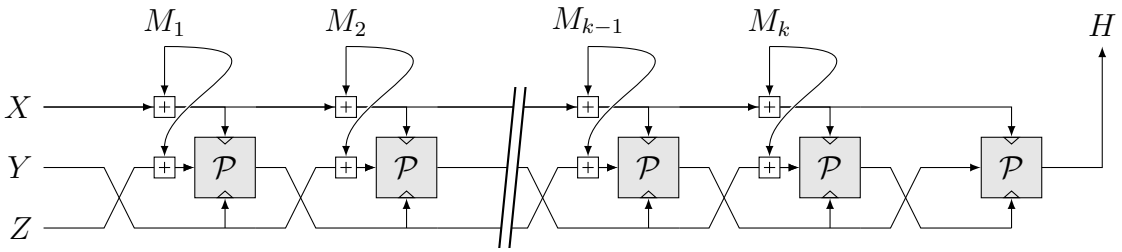

FIG. 5.11 – Ancienne version 3 du mode opératoire

Ces versions du mode opératoire exploitaient également l'idée d'accumulateur (X sur la figure 5.11) qui collecte les bits de tous les blocs de message et empêche notamment les collisions internes sur un seul tour. La preuve d'indifférentiabilité de cet ancien mode numéro 3 avait été rédigée ; ce mode était donc à sécurité prouvée. Dans le modèle du chiffrement idéal, les preuves de résistance aux collisions, aux recherches d'antécédent et de deuxième antécédent avaient aussi été établies.

Finalement, un nouveau mode (celui de la figure 5.2) équivalent a été trouvé, dans lequel l'accumulateur de blocs de message, la partie X, pouvait disparaître ce qui permettait d'obtenir de meilleures performances. Toutefois, même si ces deux modes (ceux des

figures 5.2 et 5.11) sont deux points de vue différents d'un même mode dans le modèle du chiffrement idéal (c'est-à-dire, quand $\mathcal{P}$ est un chiffrement idéal), je crois aujourd'hui que, quand on choisit concrètement la permutation, la sécurité du mode de la figure 5.2 perd un peu par rapport à celle du mode de la figure 5.11, ce qui est bien normal, car l'état interne perd 512 bits. Même si dans l'ancien mode un attaquant était toujours capable de fixer la valeur du fil X à une valeur donnée m en insérant le message $M = m - X$, ceci utilisait tous ses degrés de liberté. Dans le mode opératoire de Shabal, on peut, par exemple, fixer C à une valeur voulue à la sortie de $\mathcal{R}$. Ceci nous laisse moins de bits de l'état interne non contrôlés que dans l'ancien mode numéro 3, mais largement assez pour respecter la sécurité attendue. En tous cas, la sécurité du mode finalement utilisé étant prouvée même avec des permutations imparfaites, elle est largement assez grande pour donner à Shabal toute la sécurité nécessaire.

Le compteur de 64 bits, ainsi que les tours finaux ont été introduits plus tard pour offrir une meilleure assurance. Le compteur améliore la résistance contre la plupart des attaques ; les tours finaux servent notamment à s'assurer que le degré algébrique de chaque bit de la sortie est suffisament élevé.

Dans le cas de Shabal la capacité est $(\ell_a+\ell_m)/2$ bits, et pour $\ell_a = 32\times r = 32\times 12 = 384$ on obtient une capacité de 448 bits. Ceci veut dire que les collisions internes sont beaucoup plus difficiles à obtenir que les collisions standard sur le haché de sortie même dans le cas où $\ell_h = 512$ car $\frac{\ell_h}{2} = 256$ reste beaucoup plus petit que la capacité.

Cette capacité élevée dans Shabal implique que de grandes parties de l'état interne ne sont pas contrôlables par l'adversaire.

L'effet du compteur w. Nous avons proposé l'utilisation d'un compteur w dans la construction générique ainsi que dans Shabal. Ce compteur n'est pas indispensable pour la preuve d'indifferentiabilité, mais il renforce la résistance contre les points fixes (déjà très grande grâce au croisement des fils B et C), ainsi que la résistance à la recherche d'un deuxième antécédent en rajoutant 2 mots à la marge de sécurité de 12 mots.

Sortie de la fonction de hachage. La sortie de la fonction de hachage Shabal est déterminée par la construction générale que nous avons décrite précédemment. Nous avons choisi comme sortie les $\ell_h/32$ derniers mots de B obtenus directement après avoir appliqué la dernière permutation paramétrée, car ce sont les mots de plus grand degré.

5.3 La permutation paramétrée $\mathcal{P}$

Dans cette section on va donner les critères de conception qui ont conduit au choix de la permutation $\mathcal{P}$ telle qu'elle est dans Shabal. Nous allons nous concentrer sur les points les plus importants. Comme nous l'avons expliqué précédemment, P est fondée sur des NLFSR : essentiellement, elle peut être vue comme un NLFSR (celui de B) avec une mémoire active. Cette mémoire est constituée d'un autre NLFSR (A) et de deux registres statiques (C et M). Le registre A modifie B comme B modifie A.

Pourquoi une permutation ?

On ne voulait pas que les tours internes puissent faire perdre de l'entropie à l'état interne. Pour cela, $\mathcal{P}$ devait être une permutation pour toutes les valeurs des paramètres M et C, ce qui est assuré par la structure basée sur des NLFSR pour B comme pour A.

La fonction $\mathcal{P}$ peut être décomposée en une partie initiale avec une rotation $\lll 17$ sur l'entrée B :

$$B_i \mapsto B_i \lll 17$$

et $16p$ itérations de

$$\begin{cases} (A,B) & \mapsto & \pi_{M,C}(A,B) \\ (M_0,\ldots,M_{15}) & \mapsto & (M_1,\ldots,M_{15},M_0) \\ (C_0,\ldots,C_{15}) & \mapsto & (C_{15},C_0,,\ldots,C_{14}) \end{cases}$$

π étant le pas élémentaire suivant :

$$\begin{array}{rccl} \pi_{M,C}: & \{0,1\}^{32r}\times\{0,1\}^{512} & \to & \{0,1\}^{32r}\times\{0,1\}^{512} \\ & (A_0,\ldots,A_{r-1},B_0,\ldots,B_{15}) & \mapsto & (A_1,\ldots,A_r,B_1,\ldots,B_{16}) \end{array}$$

$$\begin{aligned} A_r &= \mathcal{U}(A_0 \oplus \mathcal{V}(A_{r-1} \lll 15) \oplus C_8) \oplus B_{o_1} \oplus (B_{o_2} \wedge \overline{B_{o_3}}) \oplus M_0 \\ B_{16} &= (B_0 \lll 1) \oplus \overline{A_r} \end{aligned}$$

où (o_1,o_2,o_3) sont trois valeurs d'offset fixées. La fin de $\mathcal{P}$ est composée d'une transformation finale de l'état interne

$$(A,B,C) \mapsto (A+\sigma(C),B,C)$$

où $\sigma(C)$ est un vecteur de r mots construit à partir des 16 mots de C.

Donc, $\mathcal{P}$ sera une permutation si et seulement si le pas élémentaire π est une permutation.

Avec la description donnée précédemment, pour des valeurs données de M et C, $\pi_{M,C}$ peut être inversée en utilisant :

$$\begin{aligned} B_0 &= (B_{16} \oplus \overline{A_r}) \ggg 1 \\ A_0 &= \mathcal{V}(A_{r-1} \lll 15) \oplus C_8 \oplus \mathcal{U}^{-1}(A_r \oplus B_{o_1} \oplus (B_{o_2} \wedge \overline{B_{o_3}}) \oplus M_0). \end{aligned}$$

Comme tout état de $\mathcal{P}$ peut être inversé avec l'équation précédente, $\mathcal{P}$ est une permutation.

Registre A.

L'utilité de A est d'améliorer l'effet de diffusion. A est une mémoire des différences. Pour annuler les différences stockées dans A, on devrait introduire des différences dans d'autres mots du message, ce qui rend très difficile (à notre avis impossible) l'annulation des différences dans la sortie B. La fonction de rétroaction du registre A est :

$$A_{t+r} = \mathcal{U}(A_t \oplus \mathcal{V}(A_{t+r-1} \lll 15) \oplus C_{8-t}) \oplus \mathcal{G}(B_{o_1+t}, B_{o_2+t}, B_{o_3+t}) \oplus M_t, \quad \forall t \geq 0$$

pour une fonction $\mathcal{G}$ dont la définition sera discutée plus tard.

Deux mots du registre A sont utilisés pour calculer le mot suivant : A_t, qui assure que $\mathcal{P}$ est bien une permutation et qui a aussi un effet de mémoire avec retard des différences, puisque si une différence rentre dans A, elle réapparaît r tours plus tard ; et A_{t+r-1} qui est une sorte de mémoire immédiate, puisque toute différence qui rentre dans A est immédiatement réutilisée au tour suivant. Puisque A_{t+r-1} est la mémoire immédiate, nous voulions qu'elle intervienne sur le mot suivant d'une façon non linéaire après avoir subi une rotation. La rotation a pour effet de changer de position les bits le moins significatifs en entrée des fonctions non linéaires $\mathcal{U}$ et $\mathcal{V}$, sinon ils seraient linéaires. L'influence du registre A sur la rétroaction de B devait aussi être non linéaire, ce qui rend plus difficile le contrôle des différences.
Les mots de C sont introduits dans A directement par un XOR, car normalement on ne peut pas contrôler ce qu'il y a dans C puisque c'est une sortie de la permutation du tour précédent. Les mots de C ne sont pas introduits dans le même ordre que les mots de A. La raison de ceci est que, dans le tour t, le registre C correspond à la sortie B du tour précédent. Alors, quand le tour t commence, le registre A dépend linéairement des r derniers mots de C. Ainsi, si un attaquant arrive à trouver un chemin différentiel tel que les derniers r mots de C et les mots correspondants de A ont les mêmes différences à la fin du tour $(t-1)$, ces différences pourraient s'annuler pendant le tour t, si les mots de C et A étaient pris dans le même ordre, ce qui n'est donc pas le cas ici. C'est la raison pour laquelle à chaque pas on utilise $C_{8-t \bmod 16}$ à la place de $C_{t \bmod 16}$.

M est directement introduit par un XOR, de telle façon que toute différence dans M affectera A. La rotation $\lll 17$ qu'on applique à B avant le premier pas nous assure que des différences identiques dans B et M ne s'annuleront après le premier tour de $\mathcal{P}$ que si elles ont un poids 32, ce qui introduirait de nombreuses différences pour le tour suivant, où elles ne seraient que sur M.

Les fonctions $\mathcal{U}$ et $\mathcal{V}$ vues comme boîtes S. La fonction de rétroaction du registre A utilise deux fonctions très simples,

$$\mathcal{U}(x) = 3 \times x \bmod 2^{32} \text{ et } \mathcal{V}(x) = 5 \times x \bmod 2^{32},$$

qui augmentent le degré et la non-linéarité. Leur présence complique la recherche des chemins différentiels de probabilité élevée par une technique du type « modifier et corriger ». Utiliser deux fonctions non-linéaires permet de garantir que l'insertion de deux blocs de message différents fera apparaître au moins une différence en entrée de $\mathcal{U}$ ou $\mathcal{V}$ au bout de deux tours. Ceci est démontré dans le document de Shabal et n'est pas assuré si une seule fonction non-linéaire est utilisée.

$\mathcal{U}$ et $\mathcal{V}$ ont été choisies pour être aussi simple que possible et permettre des implémentations efficaces. Aussi, les constantes 3 et 5 utilisées sont inversibles modulo 2^{32}, ce qui implique que $\mathcal{U}$ et $\mathcal{V}$ sont des permutations, de façon à ne pas perdre de l'entropie. Un

autre avantage de ces deux fonctions est qu'elles ne peuvent pas transformer une différence symétrique en une autre différence symétrique, comme prouvé à la proposition 1 de la section 11.3.2 du document de Shabal.

Registre B.

La fonction de rétroaction non-linéaire utilisée dans le registre B est définie par :

$$B_{t+16} = (B_t \lll 1) \oplus \overline{A_{t+r}}, \quad \forall t \geq 0.$$

Le registre A intervient dans la rétroaction du registre B par le XOR de A_{t+r}. Il n'y a pas besoin d'utiliser une opération plus compliquée car A_{t+r} est une fonction non-linéaire de A et B. Utiliser le dernier mot calculé de A signifie utiliser celui qui a le plus grand degré.

L'insertion de B_t par un XOR est nécessaire pour assurer que $\mathcal{P}$ est une permutation. La rotation $B_t \lll 1$ est utilisée pour éviter que les différences dans B après un tour complet de $\mathcal{P}$, c'est-à-dire après 16 pas élémentaires, ne correspondent aux différences initiales des chemins différentiels qui ne génèrent pas de différence dans A.

Le fait de complémenter A_{t+r} bit à bit fait que l'état tout à zéro n'est pas un point fixe trivial.

À cause de l'implémentation, la fonction $\mathcal{G}$ utilisée dans la partie de la rétroaction de B qui prend en entrée des mots de B devait avoir un petit nombre d'entrées. Le plus petit nombre possible était trois. Les offsets (o_1, o_2, o_3) ont été choisis avec une recherche empirique sur tous les triplets possibles, afin de maximiser la résistance aux attaques différentielles qui utilisent des chemins de petit poids.

Pour construire $\mathcal{G}$ nous voulions utiliser des opérations simples, disponibles dans un processeur 32 bits. C'est à cause de cela que $\mathcal{G}$ est une fonction bit à bit. Nous voulions aussi qu'elle soit équilibrée pour que B_t et B_{t+16} ne soient pas corrélés, et qu'elle soit non-linéaire. Ces conditions impliquent que $\mathcal{G}$ doit être de degré 2, car le degré d'une fonction équilibrée avec n variables est au plus $(n-1)$. Ainsi, 3 était le nombre minimal de variables à prendre pour satisfaire les conditions précédentes.

Toutes les fonctions qui vérifient ces propriétés sont équivalentes, à une permutation affine près des entrées. Dans chaque cas, elles ont 4 approximations linéaires biaisées, avec un biais de $\pm 3/4$.

Nous avons choisi la fonction utilisée $\mathcal{G}$ en prenant en compte le fait que, si une même variable, par exemple x_1, intervient dans toutes les approximations linéaires biaisées, alors la recherche de chemins différentiels sera plus difficile. Toute fonction équilibrée à trois variables ayant cette propriété aura x_1 comme terme linéaire :

$$g(x_1, x_2, x_3) = x_1 + q(x_2, x_3).$$

Nous avons ensuite choisi comme fonction quadratique q une fonction qui ne soit pas symétrique pour éviter que, si $o_2 > o_3$, $B_{t+o_2} = 0$ implique $q(B_{t+o_2}, B_{t+o_3}) = 0$ et $q(B_{t+2o_2-o_3}, B_{t+o_2}) = 0$. Finalement :

$$\mathcal{G}(B_{o_1+t}, B_{o_2+t}, B_{o_3+t}) = B_{o_1+t} \oplus (B_{o_2+t} \wedge \overline{B_{o_3+t}}).$$

La transformation finale.

La transformation finale ou mise à jour finale de A

$$(A, B, C) \mapsto (A + \sigma(C), B, C)$$

qui est appliquée à l'état interne après les $16p$ pas de π sert à renforcer la permutation inverse $\mathcal{P}^{-1}$. Effectivement, comme nous allons le voir plus tard, sans cette dernière transformation, alors que la permutation $\mathcal{P}$ est très forte dans le sens où tous les mots en sortie de $\mathcal{P}$ dépendent de tous les mots en entrée et de tous les mots des paramètres M et C, ce ne serait pas le cas de la permutation inverse $\mathcal{P}^{-1}$, si elle était réduite à $16p$ pas de $\pi^{-1}_{M,C}$. Ceci pourrait être dangereux si contrôler plus de 12 mots de B en sortie de $\mathcal{P}^{-1}$ était possible en choisissant le message : dans ce cas notre marge de sécurité, qui est de 12 mots, pourrait disparaître car on pourrait fixer le fil B en avant avec le $\boxminus$ du mode, et en arrière avec la propriété qu'on vient d'énoncer. Ceci pourrait arriver pour $p = 2$. Pour éviter la possibilité d'attaques dans cette veine, on a rajouté cette dernière transformation à la construction basée sur des NLFSRs de $\mathcal{P}$. Pour éliminer cette faiblesse, la transformation finale fait que la partie A de la sortie soit dépendante de C. Quand on calcule la fonction vers l'arrière dans une attaque par recherche d'un (deuxième) antécédent, l'entrée C de $\mathcal{P}^{-1}$ dans le tour t dépend de M_t car le bloc de message a été soustrait avant d'appliquer $\mathcal{P}^{-1}$. De cette façon, l'entrée A de $\pi^{-1}_{M,C}$ dépend de M et C.

Pour trouver le vecteur $\sigma(C)$ qui définit cette transformation nous avons cherché des vecteurs qui donnaient une dépendance maximale entre les mots de B en sortie de $\mathcal{P}^{-1}$ et les mots de M pour $p = 1$, $p = 2$ et $p = 3$. Pour des raisons d'implémentation, notre recherche a été restreinte à des vecteurs $\sigma(C)$ qui pouvaient être calculés avec une simple boucle de la forme : pour i de 0 à $s - 1$,

$$\sigma(C)[i \bmod r] \leftarrow \sigma(C)[i \bmod r] + C[(-1)^e i + \text{offset}].$$

Nous avons réalisé une recherche exhaustive pour la taille de la boucle s, la direction e et l'offset pour le choix que nous avons fait de r, *i.e.* $r = 12$. Une autre condition était que chaque mot de la partie de A de la sortie de $\mathcal{P}$ devait dépendre d'un ensemble différent des mots de C. Le vecteur $\sigma(C)$ que nous avons choisi peut être calculé avec une boucle de taille 36, avec $e = 0$ et offset $= 3$, *i.e.* pour i de 0 à 11,

$$\sigma(C)[i] = C[i + 3] + C[i + 3 + r] + C[i + 3 + 2r].$$

Il est important de remarquer que chaque $A[i]$ dépend de trois mots de C qui sont différents pour les différents i. Si une valeur différente de r est utilisée, les valeurs de s, e et l'offset doivent être recalculées.

5.4 Proposition pour inclure un sel

Dans la documentation de Shabal nous n'avons pas spécifié comment utiliser un sel car la resistance aux collisions était définie de manière formelle par une famille de fonctions

obtenue en considérant l'IV et le compteur comme aléatoires. Mais, nous pouvons également définir un sel comme cela est fait dans d'autres propositions. Il y a plusieurs façons de le faire. Je vais décrire ici ma proposition, qui n'est pas officielle du point de vue de Shabal, mais simplement mon avis personel. Donc on considère un sel de 64 bits, formé par S_0, les 32 bits moins significatifs, et S_1, les 32 bits les plus significatifs. Je propose d'insérer le sel à deux endroits : au début et à la fin :

1. Pendant l'initialisation, le nouveau bloc de message M_{-1} sera le suivant. Tous ses mots sont définis comme dans la version sans sel, sauf les deux mots

$$M_{-1}[0] = \ell_h \boxplus S_0 \text{ et } M_{-1}[1] = \ell_h \boxplus 1 \boxplus S_1,$$

   c'est-à-dire, on utilise le même M_{-1} que sans le sel et en plus on insère les deux mots du sel en $M_{-1}[0]$ et $M_{-1}[1]$.

2. Pendant la finalisation, avant de commencer les tours finaux, on XORe le sel au compteur qui va rester constant pendant ces tours, et qui est de la même taille.

Ceci ne devrait pas changer les performances, le nombre d'appels à la fonction de tour est le même avec ou sans sel.

5.5 Analyse de la sécurité de Shabal et des versions réduites

Je vais introduire ici un schéma qui sert à mieux comprendre les attaques suivantes. Sans lui, les attaques peuvent paraître très compliquées, mais avec le dessin tout devient très visuel, et les explications sont plus simples à suivre. Il s'agit d'un « patron » représenté à la figure 5.12 sur lequel on va pouvoir écrire. Par la suite, et dans des cas particuliers, je dessinerai les opérations correspondantes pour pouvoir suivre le texte. Pour comprendre comment il fonctionne, il faut garder à l'esprit que le temps est représenté suivant l'axe vertical orienté vers le bas (*i.e.*, l'origine est en haut). Une ligne représente l'état des registres B et A pour un instant déterminé qui est noté à gauche de chaque registre, et où on considère l'instant 0 comme t_0. Le registre de gauche est le registre B et celui de droite est le registre A. La première ligne représente l'état initial de deux registres $(B_0, \ldots, B_{15}, A_0, \ldots, A_{11})$. On va utiliser la notation suivante : pour chaque nouveau mot calculé, on incrémentera l'indice du mot précédent. De cette façon, après les 3 tours, les mots de la partie B à la sortie de la permutation sont notés $(B_{48}, \ldots, B_{63})$ et les mots de la partie A $(A_{48}, \ldots, A_{59})$. Comme ce sont des registres à décalage, pour passer d'une ligne à la suivante on décale tous les mots de un vers la gauche, le mot le plus à gauche disparaît, et le nouveau mot généré est celui qui rentre à droite. Ceci signifie que toutes les diagonales partant d'en haut à droite et finissant en bas à gauche sont occupées par le même mot. Au milieu on précise les mots de M et C qui interviennent à chaque ligne pour calculer les deux mots suivants.

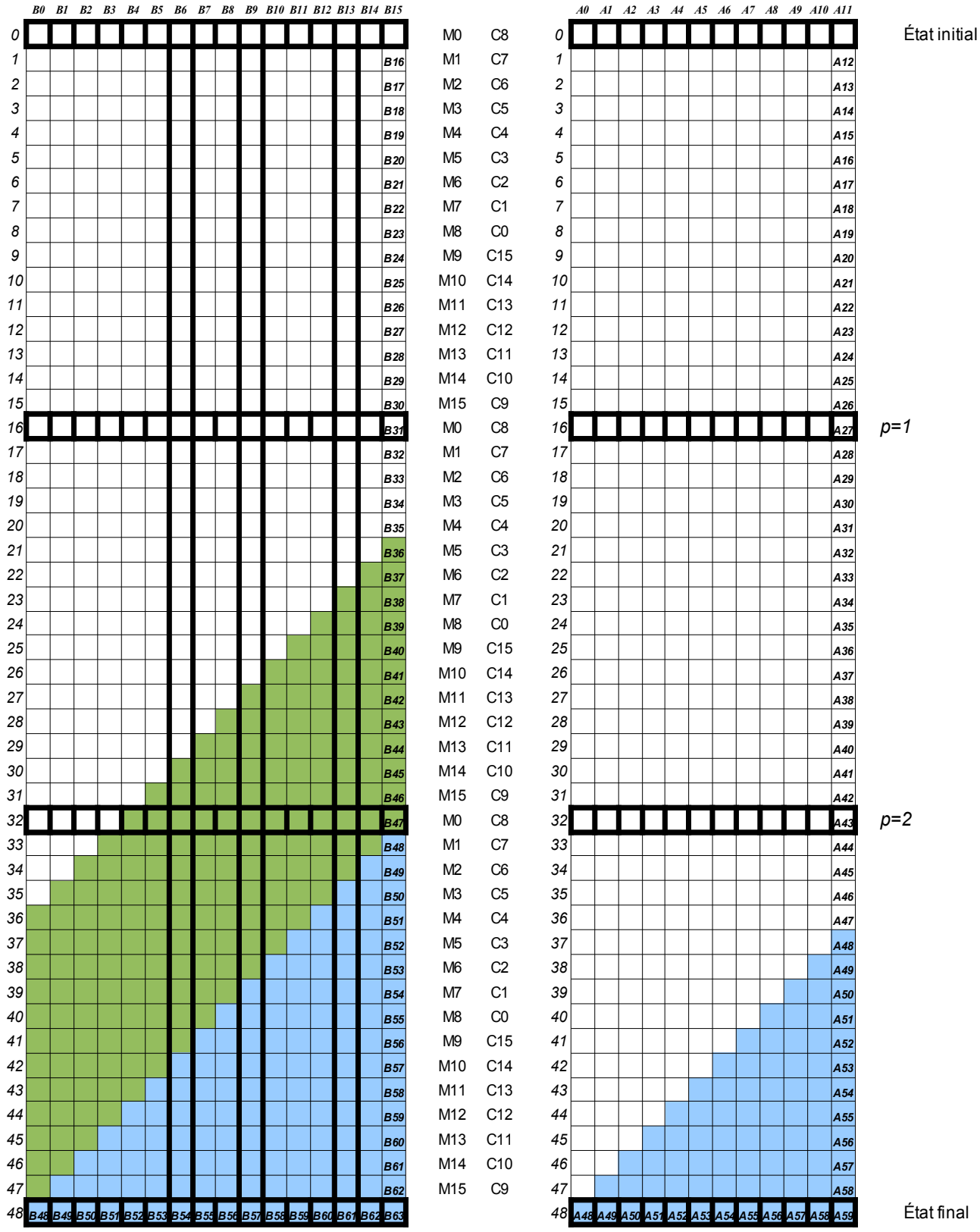

FIG. 5.12 – Représentation graphique générale de la permutation $\mathcal{P}$

Comme on va essayer d'inverser $\mathcal{P}$, dans la plupart des cas, on considère la sortie de $\mathcal{P}$ connue. Ceci est représenté par les mots colorés en bleu. Les mots coloriés en vert sont des mots également connus, car si on utilise

$$(B_i \lll 1) = B_{i+16} \oplus \overline{A}_{i+12}. \tag{5.1}$$

pour $36 \leq i \leq 47$ on peut calculer 12 mots de B : $B_{36}, \ldots, B_{47}$. Les colonnes en gras dans le registre B représentent les mots qui interviennent dans le calcul du mot suivant de B par la fonction $\mathcal{G}$. Le mot le plus à gauche de B intervient aussi pour calculer le mot suivant, ainsi que les deux mots des extrémités du registre A. Si dans une ligne i toutes ces cases sont coloriées ainsi que les M_i et C_i correspondants, on peut colorier le nouveau mot de B de la ligne $i+1$. Ceci correspond à l'équation qui définit $\mathcal{P}$. De cette manière et avec l'équation précédente que nous avons utilisée pour trouver les mots verts, on va essayer de « remonter » dans le dessin.

L'idée principale maintenant est que, chaque fois qu'on considère un nouveau mot comme connu (de M ou de C), on peut le colorier, ainsi que tous les mots qu'on peut calculer avec lui.

Nous allons définir ici les 3 types d'analyse que nous allons effectuer, pour pouvoir comprendre mieux dans chaque cas ce que nous voulons faire :

- *Type a :* ce sont des cryptanalyses par recherche d'un (deuxième) antécédent de la version affaiblie de Shabal sans la dernière transformation, Weakinson-NoFinalUpdateA. Dans ce cas la permutation interne n'inclut pas la mise à jour finale de A. On cherche à fixer à une valeur de notre choix le plus grand nombre possible de mots de B à la sortie de $\mathcal{R}^{-1}$, à l'aide du message, sur lequel on a une liberté totale. Le mode est donc pris en compte. Chaque mot B_i en sortie dépend de M_i suivant le mode opératoire. Nous pouvons représenter ceci dans le patron avec l'état de M sur l'état initial de B, ces deux états étant reliés par l'équation

  $$B_i = (\beta_i \boxplus M_i) \lll 17,$$

  où β est la valeur de B à la sortie de P^{-1}. Les mots de C utilisés dans la permutation (c'est-à-dire, ceux coloriés dans le patron) seront connus si et seulement si les mots de M de même indice sont aussi connus. Ceci est le cas des dessins 5.13 et 5.14 ainsi que des tableaux 5.1 et 5.3.
- *Type b :* c'est sont des analyses de la permutation $\mathcal{P}^{-1}$ isolée, indépendamment du mode. Dans ce cas, le mode n'est pas pris en compte, et donc si on connaît C en entrée de $\mathcal{P}^{-1}$ on peut inverser la mise à jour finale de A. Ces analyses ressemblent donc plus à celles du *type a* qu'à celles sur le vrai Shabal. Elles sont utilisées pour faire des distingueurs sur $\mathcal{P}$ et $\mathcal{P}^{-1}$, comme on le verra plus tard, et ne sont pas applicables à Shabal, entre autre à cause des tours finaux du mode opératoire. Elles sont représentées par le patron où tous les mots de C sont connus. Ce type d'analyse est le cas des dessins 5.15 et les tableaux 5.2 et 5.4.
- *Type c :* elles ressemblent aux analyses du *type a*, mais sur le vrai Shabal et non sur une version affaiblie. Dans ce cas, nous cherchons aussi à fixer le plus grand

nombre possible des mots de B en sortie de $\mathcal{P}^{-1}$ à une valeur choisie, à l'aide des mots de message. Dans ce cas donc, le mode est pris en compte ainsi que le final update de A. Il s'agit des scénarios les plus réalistes pour attaquer Shabal, et aussi des plus difficiles à réaliser. Sur le patron nous représentons ce type d'attaques avec le vecteur du final update sous l'état final de A. Comme C dépend des mots de M, nous pouvons remplir ce vecteur en fonction des mots de M connus, puisque M est la quantité qu'on choisit librement. Des mots coloriés du vecteur signifiront que nous connaissons aussi les mots correspondants de A. La dépendance entre C et M est la même que dans les attaques du *type a*. À ces attaques correspond le dessin 5.16.

Je vais tout d'abord introduire deux attaques du *type a*, avec $p = 1$ et $p = 2$. Ces attaques montées à la main ont été présentées dans le document Shabal. Elles exploitent les faiblesses de $\mathcal{P}^{-1}$ de Weakinson-NoFinalUpdateA qu'on vient de voir. Ensuite, on verra les résultats obtenus en automatisant ces attaques pour $p = 3$ et des analyses du *type b*. Finalement, on verra des attaques du *type c* sur le vrai Shabal, pour vérifier qu'elles ne diminuent pas la sécurité de Shabal.

5.5.1 Attaques sur la version sans la mise à jour finale

Dans cette section nous allons voir deux attaques du *type a* qui ont été inclus dans la documentation de Shabal.

Avec $p = 1$.

Nous allons d'abord décrire l'attaque par recherche d'un (deuxième) antécédent sur la version affaiblie de Shabal avec $p = 1$ et sans la transformation finale. Cette attaque correspond à la figure 5.13. Soit $\mathcal{M}$ le message donné de k blocs. Comme Shabal a un compteur, nous allons chercher un autre message $\mathcal{M}'$ de la même longueur que $\mathcal{M}$. $\mathcal{M}'$ peut être séparé en trois parties : les premiers k_1 blocs, 2 blocs intermédiaires et les $(k - k_1 - 2)$ derniers blocs. Nous choisissons aléatoirement N_1 messages $(M_1, \ldots, M_{k_1})$ formés par k_1 blocs de message. Nous pouvons calculer N_1 états internes S_{k_1}, obtenus pour chacun de ces messages à partir de l'état initial. De la même manière, nous pouvons choisir N_2 vecteurs $(M_{k_1+3}, \ldots, M_k)$ de $k - k_1 - 2$ blocs de message. Depuis l'état interne final qu'on a obtenu quand $\mathcal{M}$ a été haché, nous pouvons calculer en arrière les états internes S_{k_1+2}, juste avant l'insertion de M_{k_1+3} pour ces N_2 messages. Maintenant nous pouvons utiliser les deux blocs intermédiaires M_{k_1+1} et M_{k_1+2} pour trouver une collision sur les 16 mots de la partie B de S_{k_1+1}, ce qui veut dire que trouver une collision sur le reste de l'état interne suffira pour trouver un message avec la même valeur de haché que $\mathcal{M}$.

Soit β une valeur ciblée de $\{0,1\}^{512}$ pour la partie B de S_{k_1+1}. Pour chacune des N_1 valeurs de $S_{k_1} = (A_{k_1}, B_{k_1}, C_{k_1})$, on choisit $M_{k_1+1} = C_{k_1} - \beta$, ce qui veut dire que la partie B de S_{k_1+1}, qui correspond à $C_{k_1} - M_{k_1+1}$, est égale à β.

La différence avec Shabal est que nous sommes aussi capables d'aller vers l'arrière. Soient $A_0, \ldots, A_{11}$ et $B_0, \ldots, B_{15}$ les valeurs des registres A et B de S_{k_1+1}, et soient $A_{16}, \ldots, A_{27}$ et $B_{16}, \ldots, B_{31}$ les valeurs de A et B après avoir appliqué $\mathcal{P}$. Ces sorties sont connues car

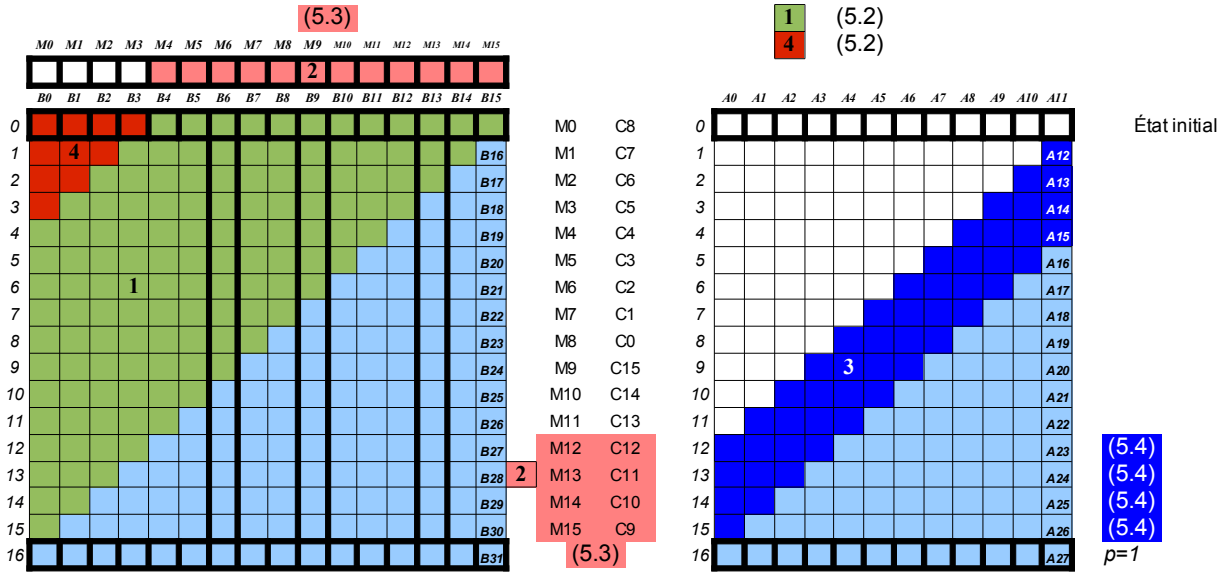

FIG. 5.13 – Représentation graphique de l'attaque avec $p = 1$

elles sont incluses dans l'état interne S_{k_1+2}. Par définition comme on l'a déjà vu, nous avons que, pour tout $0 \leq i \leq 15$,

$$(B_i \lll 1) = B_{i+16} \oplus \overline{A_{i+12}}. \tag{5.2}$$

Ceci signifie que B_i est complètement déterminé par S_{k_1+2}, pour i de 4 à 15. Nous pouvons alors choisir les mots $M_{k_1+2,i}$ d'indice i du bloc de message M_{k_1+2} pour $4 \leq i \leq 15$ afin que :

$$B_i = (\beta_i + M_{k_1+2,i}) \lll 17. \tag{5.3}$$

Pour trouver les valeurs de $M_{k_1+2,i}$, $0 \leq i \leq 3$, qui mènent aux valeurs espérées de β_i, $0 \leq i \leq 3$, nous calculons maintenant les valeurs de A_{12}, A_{13}, A_{14} et A_{15} avec l'équation suivante pour i de 12 à 15 :

$$A_{i+12} = \overline{B_{i+6}} B_{i+9} \oplus B_{i+13} \oplus M_{k_1+2,i} \oplus \mathcal{U}(A_i \oplus \mathcal{V}(A_{i+11} \lll 15) \oplus (C_{8-i} + M_{k_1+2,8-i})) \tag{5.4}$$

ou, de façon équivalente,

$$A_i = \mathcal{V}(A_{i+11} \lll 15) \oplus (C_{8-i} + M_{k_1+2,8-i}) \oplus \mathcal{U}^{-1}(A_{i+12} \oplus \overline{B_{i+6}} B_{i+9} \oplus B_{i+13} \oplus M_{k_1+2,i}).$$

Il est important de se rendre compte que, à l'aide de l'équation (5.2), nous pouvons aussi écrire (5.4) en fonction de B_i et B_{i+16} à la place de A_{i+12}. Le calcul de $A_{12}, \ldots, A_{15}$ fait intervenir $A_{23}, \ldots, A_{27}$, les B_i pour $i \geq 18$ et $C_9, \ldots, C_{12}$ qui sont connus car ils peuvent être déduits de S_{k_1+2} et $M_{k_1+2,9}, \ldots, M_{k_1+2,12}$. Alors, nous pouvons calculer $B_0, \ldots B_3$ à partir de (5.2), et nous choisissons $M_{k_1+2,0}, \ldots, M_{k_1+2,3}$ tels que

$$B_i = (\beta_i + M_{k_1+2,i}) \lll 17, \ \forall 0 \leq i \leq 3.$$

Finalement, nous avons trouvé N_1 préfixes $M_1, \ldots, M_{k_1}, M_{k_1+1}$ et N_2 suffixes M_{k_1+2}, $M_{k_1+3}, \ldots, M_k$ qui mènent à deux ensembles d'états internes S_{k_1+1} dont les parties B sont égales à une valeur donnée β. Pour $N_1 = N_2 = 2^{32\times\frac{16+12}{2}} = 2^{32\times 14}$, nous allons donc trouver une collision au sein de ces deux ensembles d'états internes. Alors, un message $\mathcal{M}'$ avec le même haché que $\mathcal{M}$ peut être trouvé avec $2^{32\times 14}$ appels à la fonction de tour, ce qui est mieux que l'attaque générique par recherche d'un deuxième antécédent pour une longueur de haché de $\ell_h = 512$.

Un antécédent peut être construit par la même méthode. Nous n'avons qu'à choisir aléatoirement un état interne final dont la partie B est (partiellement) déterminée par le haché donné. Alors, avec l'attaque précédente, nous pouvons trouver un message qui aboutit à cet état final.

Avec $p = 2$.

Pour Weakinson-NoFinalUpdateA avec $p = 2$, la même méthode peut être appliquée. Cette attaque est maintenant représentée à la figure (5.14). Pour $p = 2$, nous sommes capables de fixer 12 mots de la partie B de S_{k_1+1}. Si nous considérons l'inversion des N_2 valeurs de S_{k_1+2}, les variables connues de S_{k_1+2} sont maintenant $A_{32}, \ldots, A_{43}$ et $B_{32}, \ldots, B_{47}$. Comme dans le cas précédent, tous les B_i, pour i de 20 à 31, sont complètement déterminés par S_{k_1+2} et B_{i+16} avec (5.2) (étape 1). Ceci n'impose pas de conditions sur M_{k_1+2}. Maintenant nous attribuons à $M_{k_1+2,12}$, $M_{k_1+2,13}$, $M_{k_1+2,14}$ et $M_{k_1+2,15}$ des valeurs arbitraires fixées, par exemple 0 (étape 2). Alors, nous voulons que l'entrée du premier pas élémentaire de $\mathcal{P}$ pour le tour $(k_1 + 2)$ vérifie

$$B_i = (\beta_i \boxplus M_{k_1+2,i}) \lll 17, \ \forall 12 \leq i \leq 15.$$

Plusieurs valeurs intermédiaires de A_i, B_i et M_i peuvent être déduites maintenant avec les relations suivantes :

$$\begin{aligned} A_{i+12} &= \overline{B_{i+6}} B_{i+9} \oplus B_{i+13} \oplus M_{k_1+2,i} \oplus \mathcal{U}(A_i \oplus \mathcal{V}(A_{i+11} \lll 15) \\ &\quad \oplus (C_{8-i} + M_{k_1+2,8-i})) \qquad (5.5) \\ (B_i \lll 1) &= B_{i+16} \oplus \overline{A_{i+12}}. \qquad (5.6) \end{aligned}$$

En fait, nous suivons le cheminement suivant : de la connaissance de B_{13}, B_{14}, B_{15}, avec (5.6) pour $13 \leq i \leq 15$, nous obtenons A_{25}, A_{26}, A_{27} (étape 3). Avec (5.5) pour $25 \leq i \leq 27$, nous obtenons $M_{k_1+2,i}$ pour $9 \leq i \leq 11$ (étape 4). Alors, nous obtenons les valeurs voulues pour les mots 9 à 11 de la partie B de S_{k_1+1} (étape 5) si et seulement si

$$B_i = (\beta_i \boxplus M_{k_1+2,i}) \lll 17, \ \forall 9 \leq i \leq 11.$$

Maintenant, avec les valeurs de $B_9, B_{10}, B_{11}, B_{12}$ nous pouvons déterminer $A_{21}, A_{22}, A_{23}, A_{24}$ avec (5.6) pour $9 \leq i \leq 12$ (étape 6).

D'un autre côté, la connaissance de $M_{k_1+2,i}$ pour $9 \leq i \leq 15$ nous donne les valeurs de $A_{28}, A_{29}, A_{30}, A_{31}$ avec (5.5) pour $28 \leq i \leq 31$ (étape 7). Alors, $A_{28}, A_{29}, A_{30}, A_{31}$ donnent les valeurs de $B_{16}, B_{17}, B_{18}, B_{19}$ par (5.6) avec $16 \leq i \leq 19$ (étape 8).

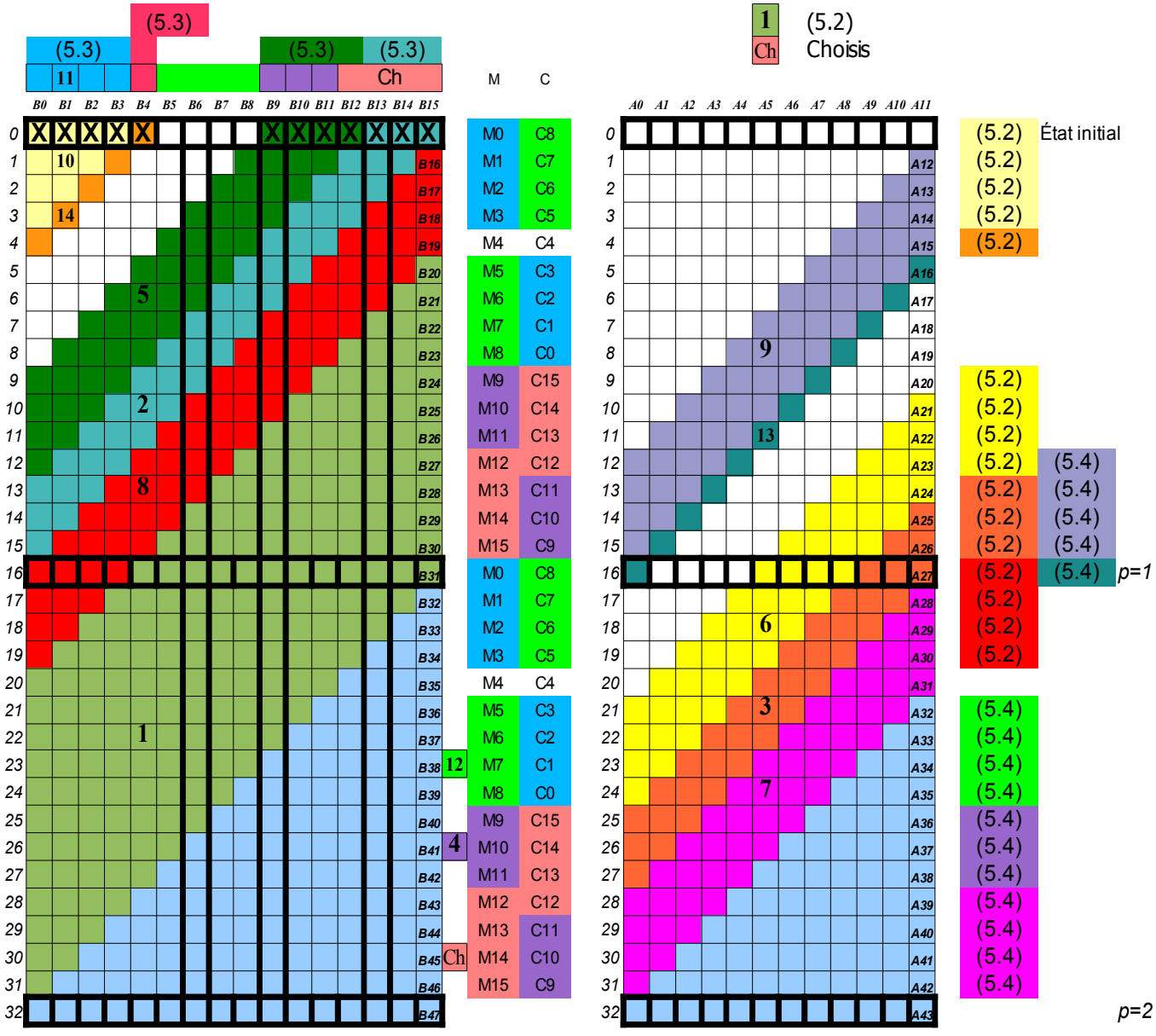

FIG. 5.14 – Représentation graphique de l'attaque avec $p = 2$

Maintenant, $A_{23}, A_{24}, A_{25}, A_{26}, A_{27}$ donnent les valeurs de $A_{12}, A_{13}, A_{14}, A_{15}$ avec (5.5) pour $12 \leq i \leq 15$ (étape 9), car nous connaissons B_j pour $j \geq 18$ et $M_{k_1+2,j}$ pour $9 \leq j \leq 15$.

Avec $A_{12}, A_{13}, A_{14}, A_{15}$ et $B_{16}, B_{17}, B_{18}, B_{19}$, nous obtenons B_0, B_1, B_2, B_3 avec (5.6) pour $0 \leq i \leq 3$ (étape 10). Alors, pour

$$M_{k_1+2,i} = (B_i \ggg 17) \boxminus \beta_i, \quad \forall 0 \leq i \leq 3,$$

nous obtenons les valeurs voulues pour les 4 premiers mots de la partie B de S_{k_1+1} (étape 11).

Les blocs de message $M_{k_1+2,i}$ pour $5 \leq i \leq 8$ peuvent être déduits à partir de (5.5) pour $21 \leq i \leq 24$ (étape 12) car $A_{21}, A_{22}, A_{23}, A_{24}$ et $M_{k_1+2,0}, M_{k_1+2,1}, M_{k_1+2,2}, M_{k_1+2,3}$ sont connus. La connaissance de A_{27}, A_{28}, $M_{k_1+2,0}$ et $M_{k_1+2,8}$ détermine A_{16} avec (5.5) pour $i = 16$ (étape 13). À partir de A_{16} et B_{20}, nous obtenons B_4 avec (5.6) pour $i = 4$ (étape 14). Finalement nous choisissons $M_{k_1+2,4}$ pour que

$$B_4 = (\beta_4 \boxplus M_{k_1+2,4}) \lll 17.$$

Ainsi, nous avons trouvé un bloc de message M_{k_1+2} tel que 12 mots de la partie B de S_{k_1+1} sont égaux aux mots correspondants de β (*i.e.* tous les mots sauf ceux de 5 à 8). Alors, une attaque par recherche d'un deuxième antécédent peut être montée en trouvant une collision sur le reste de l'état interne, de taille $16+4+r$ mots. Ceci a un coût de $2^{32\times 16}$ appels à la fonction de tour, ce qui est la même complexité que l'attaque générique par recherche d'un deuxième antécédent pour une longueur de haché de $\ell_h = 512$ bits.

Si à la place de Weakinson-NoFinalUpdateA nous regardons Shabal pour $p = 2$, c'est-à-dire, en incluant la transformation finale de A, le mieux qu'on peut faire est de fixer 7 mots, ce qui donne des attaques moins bonnes que l'attaque générique pour $p = 2$. Nous n'allons pas entrer dans les détails de cette partie-là, puisque nous le ferons pour $p = 3$, ce qui est plus intéressant, et qui utilise les mêmes idées et techniques.

5.5.2 Automatiser la recherche des dépendances

Comme nous l'avons dit précédemment, après avoir trouvé ces attaques à la main, nous avons automatisé la recherche des relations entre les mots de la partie B de la sortie de $\mathcal{P}^{-1}$ et les mots du message. Nous allons regarder maintenant ces tables de dépendance pour $p = 2$ et $p = 3$. Il s'agit des tables 5.1 et 5.3 pour la version affaiblie Weakinson-NoFinalUpdateA, ce qui correspond à des analyses de *type a*. Sur la table 5.1 C signifie que le mot correspondant de B dépend de M_i (c'est à dire par l'intermédiaire de C) à cause de la soustraction définie par le mode opératoire après l'appel à $\mathcal{P}$, et M signifie que le mot correspondant de B dépend de M_i à cause de l'addition imposée par le mode avant l'appel à $\mathcal{P}$. La table 5.3 a été construite de la même façon, mais pour simplifier, nous allons seulement représenter la dépendance par un 1 et l'indépendance par un 0. Nous allons aussi voir les tables qui définissent les dépendances pour la permutation isolée $\mathcal{P}^{-1}$. Il s'agit des tables 5.2 et 5.4 (analyses de *type b*) :

	$M[0]$	$M[1]$	$M[2]$	$M[3]$	$M[4]$	$M[5]$	$M[6]$	$M[7]$	$M[8]$	$M[9]$	$M[10]$	$M[11]$	$M[12]$	$M[13]$	$M[14]$	$M[15]$
$B[15]$	0	0	0	0	0	0	0	0	0	0	0	1	0	C	0	M
$B[14]$	0	0	0	0	0	0	0	0	0	0	1	0	0	0	C,M	0
$B[13]$	0	0	0	0	0	0	0	0	0	1	0	0	0	M	0	C
$B[12]$	C	0	0	0	0	0	0	0	1	0	0	0	M	0	0	0
$B[11]$	0	C	0	0	0	0	0	1	0	0	0	M	0	0	0	0
$B[10]$	0	0	C	0	0	0	1	0	0	0	M	0	0	0	0	0
$B[9]$	0	0	0	C	0	1	0	0	0	M	0	0	0	0	0	0
$B[8]$	0	0	0	0	C,1	0	0	0	M	C	0	0	0	0	0	1
$B[7]$	0	0	0	1	0	C	0	M	0	C	C	0	0	0	1	1
$B[6]$	0	0	1	0	0	0	C,M	0	0	0	C	C	0	1	1	0
$B[5]$	0	1	0	0	0	M	0	C	0	0	0	C	C,1	1	0	0
$B[4]$	1	0	0	0	M	0	0	0	C	0	0	1	C,1	C	0	0
$B[3]$	0	0	0	M	0	0	0	0	0	C	1	1	0	C	C	1
$B[2]$	0	0	M	0	0	0	0	0	0	1	C,1	0	0	0	C,1	C
$B[1]$	C	M	0	0	0	0	0	0	1	C,1	0	C	0	1	0	C,1
$B[0]$	C,M	C	0	0	0	0	0	1	1	0	C	0	C,1	0	1	0

TAB. 5.1 – Relations de dépendance entre les mots de B en sortie de $\mathcal{R}^{-1}$ et les mots de M avec $p = 2$ dans Weakinson-NoFinalUpdateA

	$M[0]$	$M[1]$	$M[2]$	$M[3]$	$M[4]$	$M[5]$	$M[6]$	$M[7]$	$M[8]$	$M[9]$	$M[10]$	$M[11]$	$M[12]$	$M[13]$	$M[14]$	$M[15]$
$B[15]$	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
$B[14]$	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
$B[13]$	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
$B[12]$	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
$B[11]$	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
$B[10]$	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
$B[9]$	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
$B[8]$	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1
$B[7]$	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1
$B[6]$	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0
$B[5]$	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0
$B[4]$	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
$B[3]$	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1
$B[2]$	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0
$B[1]$	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	1
$B[0]$	0	0	0	0	0	0	0	1	1	0	0	0	1	0	1	0

TAB. 5.2 – Relations de dépendance entre les mots de B en sortie de $\mathcal{P}^{-1}$ et les mots de M avec $p = 2$

	$M[0]$	$M[1]$	$M[2]$	$M[3]$	$M[4]$	$M[5]$	$M[6]$	$M[7]$	$M[8]$	$M[9]$	$M[10]$	$M[11]$	$M[12]$	$M[13]$	$M[14]$	$M[15]$
$B[15]$	0	1	1	0	0	0	1	1	0	1	0	1	0	1	0	1
$B[14]$	0	0	1	1	0	1	1	0	0	1	1	0	1	0	1	1
$B[13]$	0	0	0	1	1	1	0	0	0	1	1	1	0	1	1	1
$B[12]$	0	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1
$B[11]$	0	0	1	1	0	1	1	1	0	1	1	1	1	1	1	1
$B[10]$	0	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1
$B[9]$	1	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1
$B[8]$	1	0	0	0	1	0	1	0	1	1	1	1	1	1	1	1
$B[7]$	1	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1
$B[6]$	1	1	1	0	1	0	1	1	1	1	1	1	0	1	1	1
$B[5]$	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
$B[4]$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$B[3]$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$B[2]$	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
$B[1]$	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
$B[0]$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

TAB. 5.3 – Relations de dépendance entre les mots de B en sortie de $\mathcal{R}^{-1}$ et les mots de M avec $p = 3$ dans Weakinson-NoFinalUpdateA

Du tableau 5.4 nous allons pouvoir déduire en réordonnant la matrice, que dans le cadre des attaques de *type b* pour $p = 3$, le nombre maximal de mots de B que nous allons

	$M[0]$	$M[1]$	$M[2]$	$M[3]$	$M[4]$	$M[5]$	$M[6]$	$M[7]$	$M[8]$	$M[9]$	$M[10]$	$M[11]$	$M[12]$	$M[13]$	$M[14]$	$M[15]$
$B[15]$	0	0	0	0	0	0	1	1	0	0	0	1	0	1	0	0
$B[14]$	0	0	0	0	0	1	1	0	0	0	1	0	1	0	0	1
$B[13]$	0	0	0	0	1	1	0	0	0	1	0	1	0	0	1	1
$B[12]$	0	0	0	1	1	0	0	0	1	0	1	0	0	1	1	1
$B[11]$	0	0	1	1	0	0	0	1	0	1	0	0	1	1	1	1
$B[10]$	0	1	1	0	0	0	1	0	1	0	0	1	1	1	1	1
$B[9]$	1	1	0	0	0	1	0	1	0	0	1	1	1	1	1	0
$B[8]$	1	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1
$B[7]$	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	1
$B[6]$	0	0	1	0	1	0	0	1	1	1	1	1	0	1	1	1
$B[5]$	0	1	0	1	0	0	1	1	1	1	1	0	1	1	1	1
$B[4]$	1	0	1	0	0	1	1	1	1	1	0	1	1	1	1	0
$B[3]$	0	1	0	0	1	1	1	1	1	0	1	1	1	1	0	1
$B[2]$	1	0	0	1	1	1	1	1	0	1	1	1	1	0	1	1
$B[1]$	0	0	1	1	1	1	1	0	1	1	1	1	0	1	1	1
$B[0]$	0	1	1	1	1	1	0	1	1	1	1	0	1	1	1	1

TAB. 5.4 – Relations de dépendance entre les mots de B en sortie de $\mathcal{P}^{-1}$ et les mots de M avec $p = 3$

pouvoir fixer à une valeur choisie est 8, le message étant complètement déterminé après ceci. Pour avoir un tableau du *type b* comme 5.4 sur la permutation $\mathcal{P}^{-1}$ complètement rempli de 1, on doit faire $p > 5$. Dans le cas de Shabal, grâce à la transformation finale, avec $p = 3$ tous les mots de B en sortie de $\mathcal{P}^{-1}$ dépendent de tous les mots de M, donc la matrice associée est complètement remplie de 1. Dans ce cas, nous avons essayé au lieu de fixer des mots de M, de fixer des relations entre eux, qui nous permettent de défaire la dernière transformation sans avoir besoin de fixer tous les mots du message. Dans ce cas, on a trouvé qu'il y avait trois mots de B pour lesquels nous n'avons pas besoin de fixer tout le message, mais seulement certains de ses mots et des relations entre les autres. Ces mots sont $B[15]$, $B[14]$ et $B[11]$. Ceci signifie que, au mieux, on va être capable de fixer 3 mots de B en sortie de $\mathcal{P}^{-1}$, ce qui donne une attaque beaucoup plus coûteuse que l'attaque générique. Nous allons voir ceci un peu plus en détail dans la section suivante.

5.5.3 Distingueurs sur $\mathcal{P}$

Le concept de distingueurs à clé choisie.

Les seules analyses qui ont été publiées par des cryptographes externes au groupe de concepteurs depuis l'apparition de Shabal, sont des distingueurs sur la permutation paramétrée $\mathcal{P}$ qui ne s'appliquent pas à la fonction de compression. En effet, Aumasson a publié une analyse du type cube distinguisher [Aum09] avec une complexité élevée, ensuite Knudsen, Matusiewicz et Thomsen ont publié des observations sur Shabal [KMT09], la plus intéressante étant l'existence de points fixes dans $\mathcal{P}$, qui comme nous l'avons déjà dit, ne s'appliquent pas à la fonction de compression, et finalement, Aumasson, Mashatab et Meier [AMM09] ont publié d'autres observations sur Shabal basées sur la symétrie entre les utilisations de C et de M dans $\mathcal{P}$ ainsi que sur la propriété de $\mathcal{U}$ qui conserve les différences sur le bit de poids fort. Nous allons essayer de définir et généraliser ce concept de distingueurs et d'adapter nos attaques, qui sont plus puissantes, pour construire les meilleurs distingueurs connus sur la permutation de Shabal, distingueurs qui ne menacent

pas sa sécurité malgré tout.

Le type de distingueurs que nous allons utiliser sont ceux qui permettent de distinguer $\mathcal{P}$ d'un chiffrement idéal dans le jeu suivant :

1. Le challenger choisit aléatoirement l'entrée (A, B) et les paramètres (M, C).
2. L'attaquant effectue des requêtes $\mathcal{P}_{M,C}(A, B)$ où une partie de (M, C) est connue et l'autre est choisie librement et d'une façon adaptative.
3. À partir des réponses à ses requêtes, l'attaquant peut distinguer $\mathcal{P}$ d'un chiffrement idéal.

Nouveaux distingueurs pour $\mathcal{P}$ et $\mathcal{P}^{-1}$.

Avec les attaques pour $\mathcal{P}^{-1}$ qui ont été présentées dans la section précédente, nous pouvons construire des distingueurs très efficaces sur $\mathcal{P}$.

À l'aide du tableau 5.4 de dépendances pour $p = 3$, et qui est utile pour une analyse du *type b* nous pouvons construire de nombreux distingueurs. Nous allons voir un exemple qui montre comment calculer la valeur de $B[15]$ en sortie de $\mathcal{P}^{-1}$ à partir de 4 mots de M seulement (figure 5.15).

Exemple : la relation pour $B[15]$. Ici, nous montrons que $B[15]$ peut être calculé à partir des entrées de $\mathcal{P}^{-1}$, A' et B', C et de seulement 4 mots de M. Les étapes sont celles de la figure 5.15 :

- Avec C et A', nous pouvons défaire la transformation finale sur A et calculer les 12 mots $A[48], ..., A[59]$.
- Avec (5.2) pour i de 36 à 47, nous calculons $B[36], \ldots, B[47]$ (étape 1).
- Maintenant nous utilisons l'équation (5.4) (étape 2).
  Avec (5.4) pour $i = 38$ on obtient que$A[38]$ dépend de $A[49], A[50], B[44], B[47], B[51], C[2], M[6]$.
  Avec (5.4) pour $i = 39$ on obtient que $A[39]$ dépend de $A[50], A[51], B[45], B[48], B[52], C[1], M[7]$.
  Avec (5.4) pour $i = 43$ on obtient que $A[43]$ dépend de $A[54], A[55], B[49], B[52], B[56], C[13], M[11]$.
  Avec (5.4) pour $i = 45$ on obtient que $A[45]$ dépend de $A[56], A[57], B[51], B[54], B[58], C[11], M[13]$.
- Maintenant (étape 3) avec (5.2) pour $i = 33$, on obtient $B[33]$ à partir de $B[49]$ et $A[45]$.
  Avec (5.2) pour $i = 31$, on obtient $B[31]$ à partir de $B[47]$ et $A[43]$.
  Avec (5.2) pour $i = 27$, on obtient $B[27]$ à partir de $B[43]$ et $A[39]$.
- Nous appliquons (5.4) pour $i = 27$ (étape 4), pour calculer $A[27]$ qui dépend de $A[38]$, $A[39]$, $B[33]$, $B[36]$, $B[40]$, $C[13]$, $M[11]$.
- Avec (5.2) pour $i = 15$, nous obtenons $B[15]$ à partir de $B[31]$ et $A[27]$ (étape 5).

Donc $B[15]$ dépend seulement de $M[6], M[7], M[11], M[13]$.

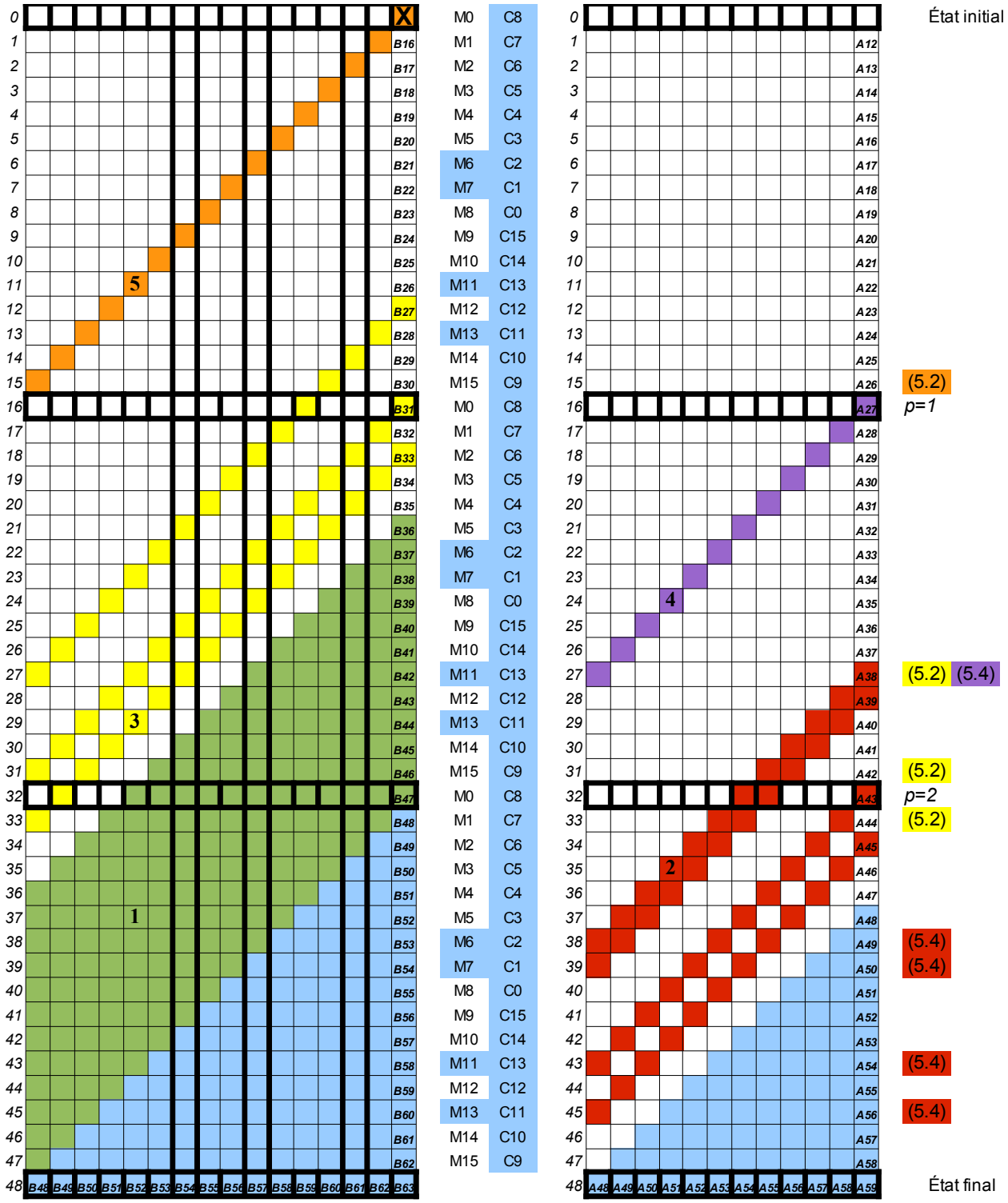

FIG. 5.15 – Représentation graphique du distingueur de $\mathcal{P}$ avec $p = 3$ sur $B[15]$.

Un distingueur à clé choisie pour $\mathcal{P}^{-1}$. Avec l'algorithme que nous venons de montrer, nous pouvons construire un distingueur sur $\mathcal{P}^{-1}$ d'une façon triviale et qui n'a besoin que d'une requête, car l'adversaire peut calculer certains mots de $\mathcal{P}^{-1}$ sans appeler $\mathcal{P}$. Il fonctionne comme suit :

1. Le challenger choisit (A', B') et (M, C).
2. L'adversaire connaît C et 4 mots de $M[6], M[7], M[11], M[13]$ seulement. Il fait une requête $\mathcal{P}^{-1}_{M,C}(A', B')$.
3. À partir de (A', B') et les quatre mots connus de M, il calcule $B[15]$ et vérifie si le résultat correspond à la valeur de $B[15]$ obtenue en réponse.

Un distingueur à clé choisie pour $\mathcal{P}$. Nous pouvons aussi construire un distingueur sur $\mathcal{P}$ avec deux requêtes dans le modèle suivant :

1. Le challenger choisit aléatoirement (A, B) et deux ensembles de paramètres (M, C), (M', C') avec $M[i] = M'[i]$ pour $i \in \{6, 7, 11, 13\}$.
2. L'adversaire connaît C, C' et $M[6], M[7], M[11], M[13]$; l'autre partie de M et M' est inconnue. Il fait les requêtes $\mathcal{P}_{M,C}(A, B)$ et $\mathcal{P}_{M',C'}(A, B)$.
3. À partir des réponses, il calcule $B[15]$ et vérifie s'il obtient la même valeur.

En utilisant les relations d'indépendance précédentes, nous pouvons, à partir de n'importe quelle valeur de (A', B', C), choisir un M tel que certains mots de B correspondent à une valeur choisie. Ensuite, nous pouvons montrer que le plus grand nombre des mots de B que nous pouvons fixer à une valeur choisie est 8. Mais étendre cette propriété pour la fonction de compression complète est beaucoup plus difficile, à cause de la transformation finale de A. De plus, ces distingueurs ne représentent aucun danger pour la fonction de compression Shabal. Dans le contexte des attaques par recherche d'un deuxième antécédent, quand on calcule en arrière, la valeur de C utilisée dans la permutation est fixée. Quand nous calculons vers l'avant, la valeur de C sera $B' \boxplus M$. Si nous voulons utiliser la propriété précédente pour fixer, par exemple, 8 mots de B, nous devons connaître C avant de pouvoir déterminer M. Donc, parce que C dépend de M, nous ne pouvons pas appliquer la propriété observée.

Distingueurs sur la fonction de compression $\mathcal{R}$.

Nous voulons préciser ici que ce que nous voulons distinguer est la restriction de la fonction de compression $\mathcal{R}$ aux fils qui correspondent à des entrées ou à des sorties de la permutation $\mathcal{P}$. Sinon, il serait très facile de distinguer, par exemple, le fil B', qui est $C \boxminus M$. Si nous considérons non seulement $\mathcal{P}$, mais la fonction complète $\mathcal{R} : (A, B, C, M) \mapsto (A', B', C')$ définie par

$$\begin{aligned} (A', C') &= \mathcal{P}_{M,C}(A, B \boxplus M) \\ B' &= C \boxminus M, \end{aligned}$$

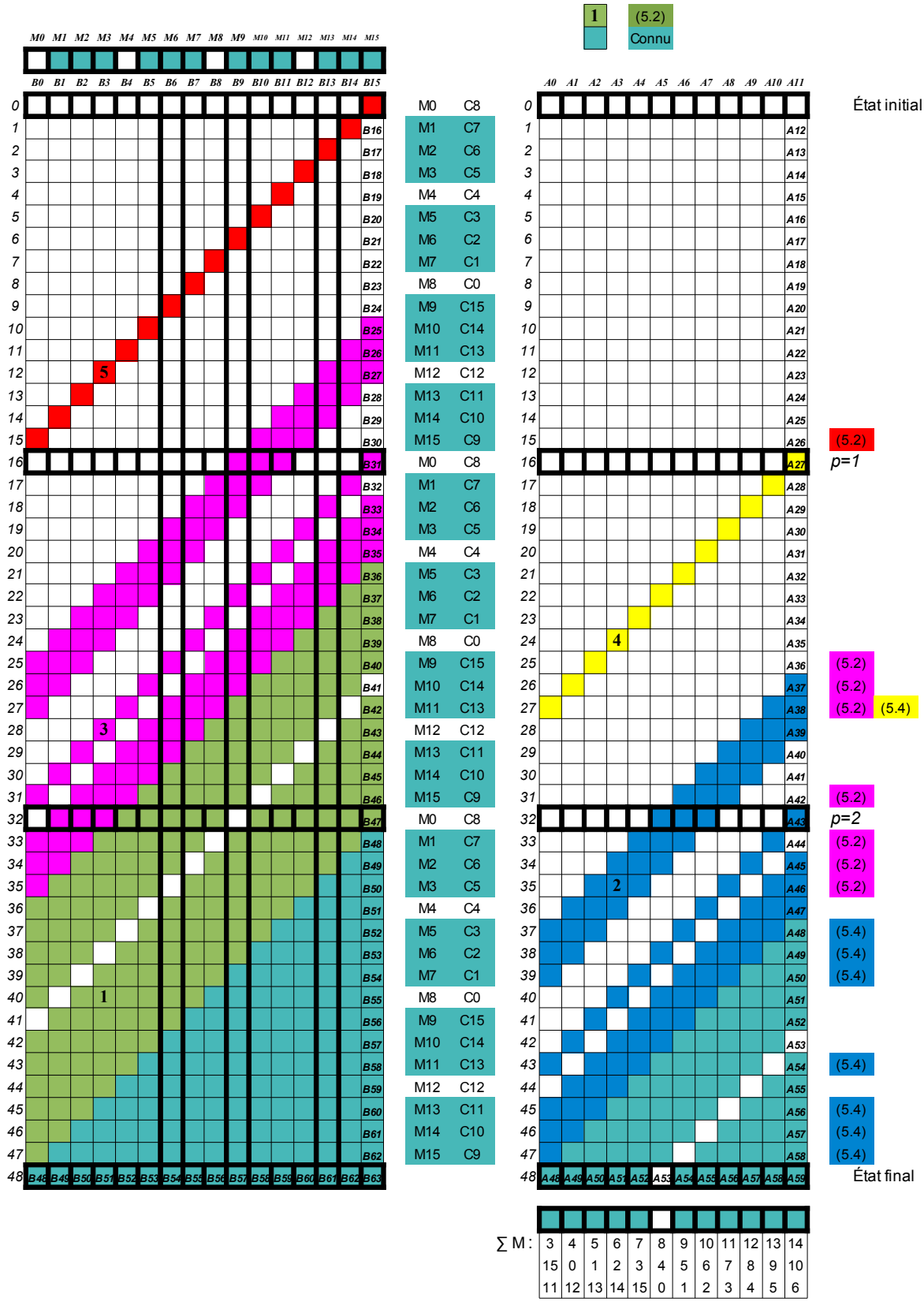

FIG. 5.16 – Représentation graphique du distingueur sur B_{15} et $\mathcal{R}$

quand nous regardons les fils (A, B) et (A', C'), qui correspondent aux entrées et sorties de $\mathcal{P}$ dans $\mathcal{R}$, aucun distingueur n'avait été présenté jusqu'ici. Dans ce cas, quand nous calculons en arrière, il est impossible de déterminer un mot de B à partir de la connaissance de (A', B', C') et de quelques mots de M seulement. La raison est que $\mathcal{P}$ est paramétrée par $C = B' \boxplus M$ qui dépend de M, et que C est utilisé pour la transformation finale sur A (*i.e.*, la première opération de $\mathcal{P}^{-1}$). Mais, même si le calcul de chaque $B[i]$ implique tous les mots de M, il pourrait nécessiter moins de mots d'information. Par exemple, $B[15]$, comme cela est montré à la figure 5.16, est complètement déterminé par $M[1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 14, 15]$ et par $M[4] \boxplus M[8] \boxplus M[12]$ et $M[0] \boxplus M[4] \boxplus M[12]$. Nous avons effectué une recherche exhaustive, comme vu dans la section précédente, qui prenait en compte la transformation finale, pour déterminer le nombre de bits d'information de M que nous avons besoin de connaître pour déterminer chaque mot de B. Avec la technique décrite précédemment, nous trouvons que seulement trois mots, $B[15]$, $B[14]$ ou $B[11]$, ne dépendent pas de tous les mots d'information de M. Il est important de remarquer que les dépendances de ces trois mots sont différentes, ce qui signifie que calculer le triplet $(B[11], B[14], B[15])$ implique la connaissance de tous les mots d'information de M.

Ceci est une analyse par distingueur de *type c* qui peut affecter la résistance de la fonction de hachage aux attaques sur la recherche d'un (deuxième) antécédent, car l'attaquant peut, en calculant en arrière, choisir plusieurs blocs de message qui mènent vers une valeur choisie et fixée pour 3 mots de B. Ainsi, un deuxième antécédent peut être trouvé pour Shabal avec complexité $2^{\frac{\ell_a + 2\ell_m - 3\times 32}{2}}$, à la place de $2^{\frac{\ell_a + 2\ell_m}{2}}$.

De toutes façons, cette complexité est beaucoup plus grande que celle de l'attaque générique pour le plus grand ℓ_h, qui est 512 bits. Comme aucun distingueur meilleur que celui-ci n'a été trouvé par recherche exhaustive, il s'agit de la meilleure attaque que nous pouvons construire sur Shabal utilisant ce type de distingueurs.

5.6 Conclusion

Shabal est donc une des fonctions les plus novatrices et performantes de la compétition du NIST, et elle est aussi à sécurité prouvée. Elle a été sélectionnée au deuxième tour de la compétition pour toutes ces raisons.

À la fin de la rédaction de cette thèse (fin août 2009), les concepteurs des fonctions qui sont au deuxième tour ont 15 jours pour proposer une petite modification sur l'algorithme pour la suite de la compétition.

Comme nous avons déjà expliqué, l'existence de distingueurs sur la permutation interne $\mathcal{P}$ n'affecte pas la sécurité de Shabal, puisque à cause du mode opératoire, les distingueurs ne sont pas applicables. Les « distingueurs » sur $\mathcal{R}$ n'affectent pas la sécurité non plus, et réduisent la marge de sécurité de seulement 3 mots, celle-ci devenant égal à 9 mots, c'est-à-dire $9 \times 32 = 288$ bits.

Nous avons déjà parlé de la difficulté de prendre des décisions quand le groupe d'auteurs est très nombreux. L'équipe Shabal n'a donc pas encore décidé s'il fallait faire une modification, et si oui laquelle. Nous y travaillons depuis longtemps, et nous étudions plusieurs

propositions. Mon avis personnel est qu'aucune modification n'est nécessaire, puisque des distingueurs et attaques présentées dans ce chapitre ont peu d'influence sur la sécurité de Shabal. Je pense que si aucune attaque n'est trouvée d'ici-là, Shabal méritera d'être considéré comme finaliste.

Chapitre 6

Cryptanalyse de fonctions de hachage

Dans ce chapitre je vais présenter les cryptanalyses de certaines propositions à la compétition SHA-3 du NIST que j'ai réalisées au cours de ma thèse. Toutes ces cryptanalyses ont été développées pendant la première phase, avant de connaître les candidats qui passent au deuxième tour, ce qui a été annoncé vers la fin de la rédaction de cette thèse. Les attaques sont assez différentes les unes des autres : certaines sont plus simples, et d'autres plus complexes. Je les ai ordonnées ici d'une façon plutôt chronologique. Parmi les fonctions analysées dans ces 6 cryptanalyses, il y en a deux qui n'ont pas été choisies pour le premier tour, trois qui ont été choisies pour le premier tour mais pas pour le deuxième, et une qui est passée au deuxième tour. Nous pouvons voir un résumé de ces attaques dans le tableau 6.1. Les couleurs représentent les types d'attaques tels qu'ils sont définis par le SHA-3 zoo [ECRb] :

Fonction	1er tour	2ème tour	Type d'attaque
MCSSHA-3	oui	non	2ème antécédent
Ponic	non	non	2ème antécédent
Cubehash	oui	oui	antécédent
Maraca	non	non	collision interne
LANE	oui	non	collision semi libre
ESSENCE	oui	non	collision

Tab. 6.1 – Résumé des principales attaques réalisées

6.1 MCSSHA-3

Ceci est un travail commun avec Jean-Philippe Aumasson présenté à [ANP09]. MCSSHA-3 est une proposition à la compétition SHA-3 du NIST par Mikhail Maslennikov qui a été choisie pour le premier tour. Elle est rapide et simple, mais nous avons trouvé des attaques sur la première version. En réponse à nos attaques, l'auteur a proposé les nouvelles versions MCSSHA-4/5/6. Jusqu'à maintenant, notre attaque générale est applicable sur toutes les versions, et deux attaques spécifiques portent sur MCSSHA-3 et MCSSHA-4. Nous allons d'abord voir les spécifications de MCSSHA-3, ensuite les différences introduites dans les versions suivantes, puis les attaques, et enfin la façon de réparer MCSSHA pour résister à ces attaques.

6.1.1 Description de MCSSHA-3

MCSSHA-3 est un candidat du premier tour de la compétition SHA-3, proposé par Maslennikov [Mas08]. Il est composé essentiellement d'un registre à décalage non linéaire avec des mots de 8 bits et une IV fixée. Le message est traité par blocs d'un octet. À chaque fois que nous insérons un nouveau bloc, nous faisons tourner le registre une fois, avec l'octet de message comme entrée, ensuite nous faisons tourner le registre Δ fois avec 0 comme entrée. Cette quantité Δ, est appelée retard. Dans le cas présent, on a $\Delta = 3$. Le registre est comme celui que nous pouvons voir sur le dessin, où π est une boîte-S. Nous

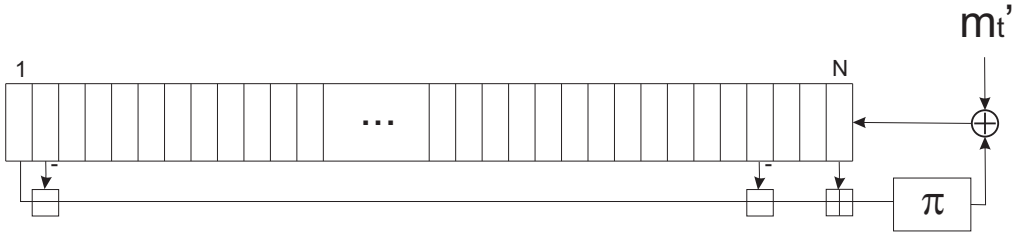

FIG. 6.1 – Représentation graphique de MCSSHA-3

allons considérer pour la suite des messages de taille multiple de huit bits, pour simplifier la description de la phase de finalisation, puisque les messages ne sont pas paddés, et si la taille du message n'est pas multiple de 8, les bits restants sont introduits pendant la phase de finalisation dans MCSSHA-3. Une fois que nous avons fini d'insérer tous les blocs de message, nous obtenons un état de N octets, où N est le nombre d'octets du registre, qui dans le cas de MCSSHA-3 vaut $N = \ell_h$, ℓ_h étant la taille du haché. L'étape de finalisation consiste maintenant en $4N$ tours du registre en utilisant comme entrée $Z\|Z\|Z\|Z$, où Z est l'état final du registre à l'étape précedente suivante, et aucun retard, processus qui n'est pas facilement inversible. Par contre, nous pouvons remarquer que l'étape d'insertion des blocs de message est complètement inversible.

6.1.2 Description de MCSSHA-4, -5, -6

Comme réponse à nos attaques, d'autres versions de MCSSHA ont été proposées. Nous allons voir les principales modifications par rapport à MCSSHA-3 :

- dans MCSSHA-4, N vaut le double de la taille du haché, $\Delta = 2$ et l'étape de finalisation est faite avec une IV fixée et un registre de taille $N/2$.
- MCSSHA-5 a le même N que MCSSHA-4, $\Delta = 3$ et un retard égal à 1 pendant la finalisation.
- MCSSHA-6 a les mêmes paramètres que MCSSHA-5 avec une finalisation plus compliquée.

6.1.3 Attaque en deuxième préimage

Le principe de cette attaque, qui fonctionne pour toutes les versions proposées de MCSSHA, est représenté dans le dessin suivant

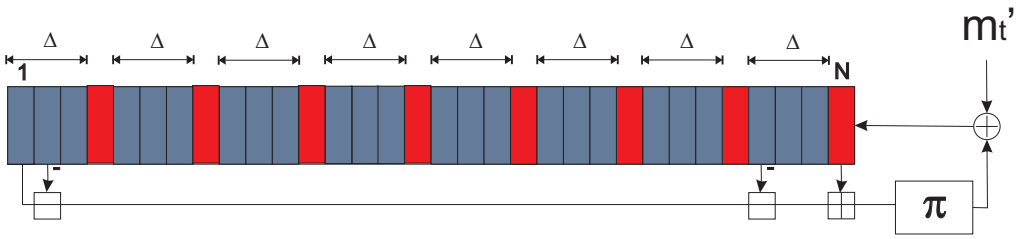

FIG. 6.2 – Représentation des mots controlés du registre

où les octets rouges sont les octets contrôlés par l'attaquant par l'insertion du bloc de message aux instants correspondants, et les octets gris, sont les octets non contrôlés à cause du retard. Le nombre d'octets non contrôlés est

$$N - \lceil N/(\Delta + 1) \rceil.$$

Comme l'étape de finalisation n'est pas inversible, nous allons rechercher des deuxièmes antécédents, et nous pouvons ainsi oublier le processus de finalisation, car nous allons effectuer une attaque par le milieu (MITM) à un instant situé entre l'état initial et l'état du registre juste avant de commencer le processus de finalisation. Dans les attaques par recherche d'un deuxième antécédent nous connaissons déjà un état avant l'étape de finalisation qui nous mène au haché voulu, et c'est pour ceci que nous pouvons faire des attaques indépendantes de la dernière étape, qui n'est pas la même pour toutes les propositions. Nous pouvons voir dans le dessin de la figure 6.3 un schéma de la procédure de l'attaque. L'attaque fonctionne de la façon suivante : d'abord, nous introduisons un nombre de blocs de message aléatoire à partir de l'état initial de façon à obtenir $2^{8\frac{N\Delta}{2(\Delta+1)}}$ états aléatoires. À partir de chacun de ces états, nous introduisons les $\frac{N}{\Delta+1}$ blocs de message nécessaires pour avoir une valeur choisie et fixée dans les octets rouges. Nous pouvons faire ceci, comme

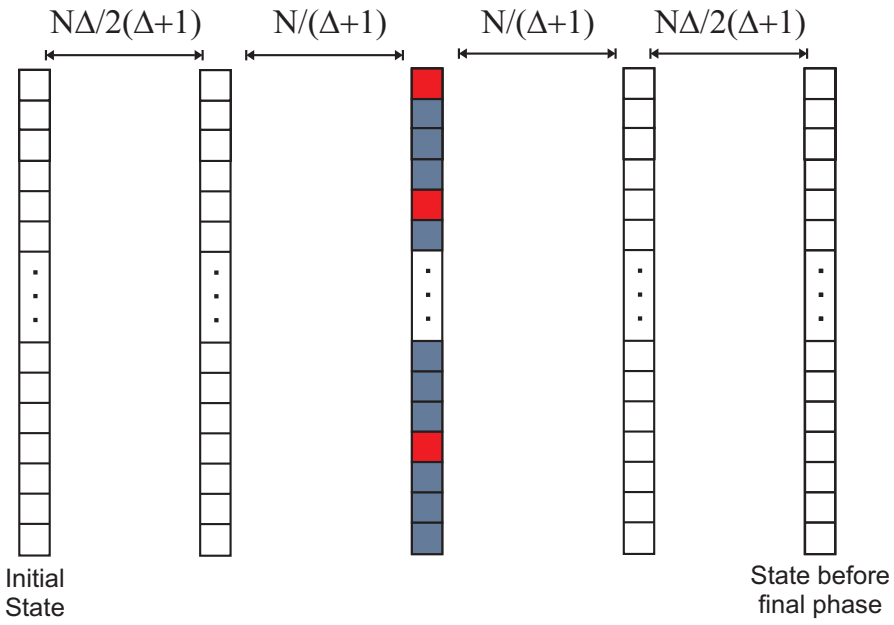

FIG. 6.3 – Principe de l'attaque sur MCSSHA

nous avons expliqué précédemment. À partir de l'état avant la phase finale, déduit en hachant le message pour lequel nous cherchons un deuxième antécédent, nous calculons en arrière et faisons l'opération inverse de l'insertion de blocs de message aléatoires de façon à avoir $2^{8\frac{N\Delta}{2(\Delta+1)}}$ états aléatoires différents obtenus, comme nous avons fait dans le sens direct. Maintenant, nous introduisons les $\frac{N}{\Delta+1}$ blocs de message nécessaires pour avoir la même valeur choisie et fixée dans les octets rouges. De cette façon, nous assurons la collision sur les octets rouges de tous les messages aléatoires, et nous cherchons alors une collision aussi sur les octets gris, qui correspondent à une taille de

$$N - \lceil N/(\Delta+1) \rceil = 8\frac{N\Delta}{\Delta+1}$$

bits. Nous avons $2^{8\frac{N\Delta}{(\Delta+1)}}$ possibilités, ce qui veut dire que nous allons trouver une collision aussi dans l'état gris, avec toutes les couples que nous avons. Avec la méthode de van Oorschot-Wiener [vOW99] qui utilise une quantité de mémoire négligeable, nous pouvons effectuer l'attaque avec une complexité en temps de $2^{4\frac{N\Delta}{(\Delta+1)}}$ et une mémoire négligeable. Nous pouvons voir dans la table 6.2 les complexités en temps pour les différentes versions de MCSSHA.

Nous allons remarquer ici, que, d'après le tableau précédent, on peut utiliser cette technique pour faire des attaques en collision moins coûteuses que les attaques génériques contre la version MCSSHA-3.

Version		N	Δ	Complexité
MCSSHA-3	256	32	3	2^{96}
	512	64	3	2^{192}
MCSSHA-4	256	64	2	2^{168}
	512	128	2	2^{340}
MCSSHA-5	256	64	3	2^{192}
	512	128	3	2^{384}
MCSSHA-6	256	64	3	2^{192}
	512	128	3	2^{384}

TAB. 6.2 – Complexité en temps de l'attaque sur les différentes versions de MCSSHA

6.1.4 Attaque par recherche d'antécédent sur MCSSHA-4

Nous allons tout d'abord voir comment l'étape de finalisation fonctionne dans la version MCSSHA-4. Ensuite nous montrerons comment nous pouvons l'exploiter pour trouver un antécédent dans un cas particulier. Nous n'allons pas donner une attaque qui trouve un antécédent pour n'importe quel haché, mais pour des certaines valeurs particulières.
Dans l'étape de finalisation nous faisons tourner un registre de taille $N/2$ octets $4N$ fois, où N est la taille du registre de l'étape d'insertion de message, et est le double de la taille du haché. L'état initial de ce registre, contrairement à MCSSHA-3, est une valeur fixée, S_0. C'est pour cette valeur S_0, pour tout S_0 possible, que nous allons trouver des antécédents. Pendant les $4N$ fois que nous faisons tourner le registre, la séquence $(z_t)_{0\leq t<4N}$ est introduite, définie par $z\|z\|z\|z$, où z est l'état final du registre de taille N octets utilisé dans l'étape précédente une fois que tout le message a été introduit.

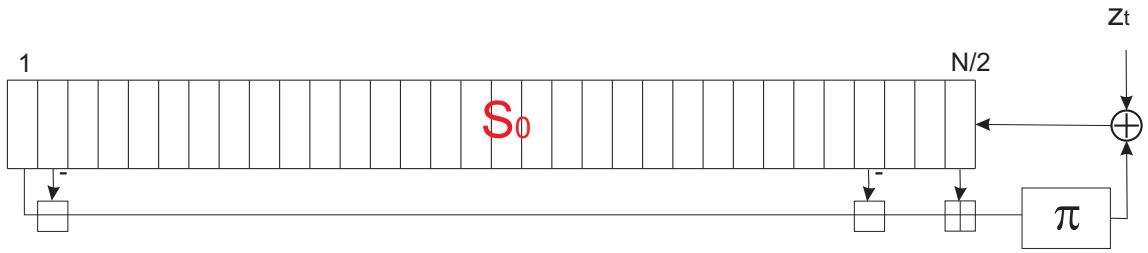

FIG. 6.4 – Finalisation pour MCSSHA-4

Dans cette attaque nous utilisons l'observation suivante sur la finalisation :
- À partir de l'état initial S_0 qui est indépendant du message, nous pouvons l'actualiser avec les octets
$$z_1, \ldots, z_{64}, z_1, \ldots, z_{64}.$$
- Si nous trouvons des
$$z_1, \ldots, z_{64}$$
qui transforment S_0 en lui-même, après quatre insertions de
$$z_1, \ldots, z_{64},$$

S_0 sera encore transformé en lui-même.

Utilisant cette observation et les points fixes ainsi définis, nous pouvons trouver un antécédent de S_0 pour tout S_0. La procédure sera la suivante : pendant l'insertion des blocs de message, nous choisissons un préfixe qui va nous mener vers un état du registre de taille $2\ell_h$ $y_1, \ldots, y_{64}$ quand $\ell_h = 256$ (ou jusqu'à y_{128} si $\ell_h = 512$, pour plus de simplicité nous allons regarder le cas 256, celui de 512 étant analogue). Maintenant nous voulons trouver $y'_1, \ldots, y'_{33}$ pour que $y_{34}, \ldots, y_{64}, y'_1, \ldots, y'_{33}$ envoie S_0 sur lui-même. Ceci est facile, puisque dans l'étape de finalisation il n'y aura pas de retard. Nous devons trouver le suffixe de message qui nous donne $y_{34}, \ldots, y_{64}, y'_1, \ldots, y'_{33}$ comme état final du registre de l'étape d'insertion de message. Nous ne pouvons contrôler qu'un tiers de celui-ci, c'est-à-dire, $33/3 = 11$ y'_i. Le reste des valeurs va être obtenu avec une probabilité $2^{-8\times 22} = 2^{-176}$. Ceci veut dire que nous pouvons trouver des antécédents de S_0 pour tout S_0 avec une complexité 2^{176} et mémoire négligeable au lieu de 2^{256}, qui est la complexité de l'attaque générique. Dans le cas $\ell_h = 512$, la complexité sera de 2^{344} au lieu de 2^{512}.

6.1.5 Réparer MCSSHA contre ces attaques

L'auteur, après chaque attaque, a modifié principalement l'étape de finalisation, mais cela ne règle pas le problème précédent, et c'est pour cela que les attaques ont été applicables contre toutes les versions. Le principal problème qui est la taille du registre, trop petite, peut être résolu en donnant une borne inférieure sur cette taille, qui dépend de la valeur de ℓ_h et de Δ, déterminée par

$$4\left(N - \left\lceil \frac{N}{\Delta + 1} \right\rceil\right) \geq \ell_h.$$

Nous avons donc
- $\ell_h = 256$ et $\Delta = 2 \rightarrow N \geq 96$,
- $\ell_h = 512$ et $\Delta = 2 \rightarrow N \geq 192$,
- $\ell_h = 256$ et $\Delta = 3 \rightarrow N \geq 86$,
- $\ell_h = 512$ et $\Delta = 3 \rightarrow N \geq 172$,

où nous rappelons que N est la taille du registre en octets.

6.1.6 Conclusion

Nous avons proposé des attaques par recherche d'un deuxième antécédent contre toutes les versions proposées de MCSSHA, une attaque en collision sur MCSSHA-3 et une attaque qui trouve des antécédents pour des valeurs spécifiques sur MCSSHA-4. Nous avons proposé un critère pour la conception des algorithmes de cette famille pour éviter les attaques par recherche d'un deuxième antécédent. MCSSHA est simple et rapide, mais il a besoin d'un état plus grand pour être sûr, au moins pour résister à nos attaques. C'est probablement pour ceci que MCSSHA n'a pas été sélectionné pour le deuxième tour de la compétition SHA-3.

6.2 Ponic

Ponic [SN08] est un algorithme qui a été soumis par Peter Schmidt-Nielsen pour la compétition SHA-3 mais qui n'a pas été accepté au premier tour. L'attaque décrite ici a été rendue publique avant la décision du NIST. C'est une attaque assez simple qui trouve des deuxièmes antécédents. Elle exploite essentiellement l'inversibilité de la procédure d'insertion du message.

6.2.1 Description de Ponic

Ponic est une fonction basée sur des registres à décalage. Nous allons décrire les caractéristiques de Ponic que nous utilisons dans notre attaque. L'état interne de Ponic est formé de 6 registres circulaires de 128 bits chacun. Les blocs de message sont composés de 256 bits. L'insertion de message s'effectue par un XOR des 128 bits les moins significatifs du bloc de message au registre 0 et des 128 bits les plus significatifs au registre 3. Après l'insertion de chaque message, la fonction de tour est appliquée un nombre ROUNDS de fois. Nous n'allons pas détailler ici cette fonction, puisque la seule caractéristique que nous allons utiliser est qu'elle est facilement inversible, le reste n'étant pas utilisé pour notre attaque.

Une fois que tous les blocs de message ont été insérés, la fonction de tour est appliquée POST-ROUNDS fois. Ensuite, la sortie de cette fonction est utilisée pour calculer le haché.

6.2.2 Cryptanalyse de Ponic

Nous allons décrire maintenant une attaque qui trouve un deuxième antécédent de manière très simple et nécessite 2^{265} appels à la fonction de compression. Cette attaque est applicable à toutes les versions de Ponic. Elle est donc meilleure que l'attaque générique pour toutes les versions avec $\ell_h \geq 256$. Comme nous l'avons déjà dit, l'attaque exploite l'inversibilité de la fonction de tour.

Soit M un message donné et h la valeur de son haché. À partir de l'état initial, nous insérons 2^{256} préfixes de messages différents, composés de 3 blocs. L'insertion de chacun de ces blocs nous donne 3 états internes. Les 3×2^{256} états internes ainsi générés vont tous être stockés dans un ensemble $\mathcal{S}_1$.

Maintenant, un calcul similaire peut être fait dans le sens l'inverse. Avec le message donné M, nous calculons l'état interne avant les derniers POST-ROUNDS tours. Ensuite, nous pouvons inverser ROUNDS tours jusqu'à arriver à l'instant où un bloc de message a été inséré. Alors, nous choisissons 2^{256} suffixes de messages aléatoires de 3 blocs chacun. Comme précédemment, ceci nous produit un ensemble $\mathcal{S}_2$ d'états internes, de taille 3×2^{256}.

Ce que nous cherchons est un couple (S_1, S_2) d'états internes dans $\mathcal{S}_1 \times \mathcal{S}_2$ qui collisionnent sur tous les registres qui ne sont pas contrôlés par l'insertion de message, c'est-à-dire, sur tous les registres sauf le registre 0 et le registre 3. La taille de ces 4 registres est de $128 \times 4 = 512$ bits, ce qui veut dire que nous allons trouver une collision entre les deux ensembles. Soient M_1 et M_2 le préfixe et le suffixe de message qui nous donnent

ces 2 états internes qui collisionnent sur les 4 registres voulus. Nous pouvons maintenant forcer la collision sur les deux registres restants en choisissant de façon adéquate le bloc de message m à insérer entre M_1 et M_2. Ceci veut dire que les 128 bits les moins significatifs de m (respectivement, les plus significatifs) doivent être égaux à la valeur de $S_1 \oplus S_2$ dans le registre 0 (resp. registre 3). Si, comme nous avons fait dans l'attaque de MCSSHA, nous appliquons la méthode de van Oorschot-Wiener, le coût de l'attaque sera de 2^{265} appels à la fonction de compression, avec une mémoire négligeable.

Alors, le message $M_1||m||M_2$ est un deuxième antécédent du message donné M.

6.3 CubeHash

CubeHash [Ber08d, Ber08c, Ber08b, Ber08a] est une fonction de hachage proposée pour la compétition SHA-3 par Daniel J. Bernstein et qui est admise au deuxième tour. Comme nous allons le voir plus tard, pour définir une fonction parmi la famille de CubeHash, il faut fixer deux paramètres : la taille en octets b du bloc de message inséré à chaque appel de la fonction de compression et le nombre de fois r que nous appliquons la fonction de tour dans la fonction de compression. Les versions de CubeHash (ℓ_h mis à part) sont donc notées CubeHashr/b. La première fonction soumise acceptée au premier tour était CubeHash8/1. Elle est très lente (de l'ordre de 20 fois plus lente que SHA-1 pour les messages longs). L'auteur avait annoncé que, en cas de passage au deuxième tour, il allait proposer comme modification (le NIST permet de faire une petite modification aux fonctions qui passent au deuxième tour) de remplacer les paramètres par d'autres, moins conservateurs : CubeHash16/32 est la fonction proposée pour le deuxième tour, celle-ci étant 16 fois plus rapide que la première proposition. Sur cette nouvelle proposition, les attaques génériques par recherche d'antécédent, ainsi que leurs améliorations que nous allons voir plus tard et que nous avons proposées dans notre papier, ont une complexité en temps de 2^{384} avec une mémoire négligeable. Ceci ne respecte pas la sécurité demandée par le NIST pour la résistance à la recherche d'antécédent dans le cas de $\ell_h = 512$, qui devrait être de 2^{ℓ_h} bits. Dans un travail commun avec Jean-Philippe Aumasson, Eric Brier, Willi Meier et Thomas Peyrin [ABM+09], nous avons fait la première analyse de CubeHash où nous améliorons l'attaque présentée dans la documentation [Ber08d, Ber08c, Ber08b, Ber08a] de CubeHash : nous montrons comment faire des multicollisions, nous proposons des attaques par recherche d'antécédent qui exploitent une propriété sur les symétries de la fonction de tour de CubeHash ainsi que des différentielles tronquées. Notre cryptanalyse a été la première publiée, mais d'autres ont été présentées plus tard, par exemple [Aum08, Dai08, BP09, BKMP09].

6.3.1 Description de CubeHash

CubeHashr/b-ℓ_h est une famille de fonctions de hachage, ℓ_h étant la taille de sortie. Pour la compétition du NIST, ℓ_h peut prendre des valeurs parmi les suivantes : $\{224, 256, 384, 512\}$. L'état interne de CubeHash a 1024 bits et est vu comme un hypercube à 5 dimensions. Comme nous l'avons déjà expliqué, la première soumission à la compétition SHA-3 était CubeHash8/1-ℓ_h.

Pour calculer le haché d'un message avec CubeHash, nous devons :

- Introduire une IV qui dépend de (ℓ_h, b, r) dans l'état interne de 1024 bits.
- Padder le message avec un 1 et autant de zéros que nécessaire pour que la longueur finale soit multiple de $8b$ bits.
- Pour chaque bloc de message de b octets :
  - XORer le bloc aux b premiers octets de l'état,
  - transformer l'état avec r tours de la fonction de tour T.
- XORer le bit 1 au bit en 993ème position de l'état.

– Transformer l'état avec $10r$ tours de T.
– Les ℓ_h premiers bits de l'état sont le haché.

Les trois derniers pas correspondent à l'étape de finalisation.

Soit $x[0], \ldots, x[31]$ un vecteur de mots de 32 bits qui représente l'état interne. La fonction de transformation T est appliquée r fois, où chaque itération T fait (voir aussi Fig. 6.5) :

for $i = 0, \ldots, 15$: $x[i+16] = x[i+16] + x[i]$
for $i = 0, \ldots, 15$: $y[i \oplus 8] = x[i]$
for $i = 0, \ldots, 15$: $x[i] = y[i] \lll 7$
for $i = 0, \ldots, 15$: $x[i] = x[i] \oplus x[i+16]$
for $i = 0, \ldots, 15$: $y[i \oplus 2] = x[i+16]$
for $i = 0, \ldots, 15$: $x[i+16] = y[i]$
for $i = 0, \ldots, 15$: $x[i+16] = x[i+16] + x[i]$
for $i = 0, \ldots, 15$: $y[i \oplus 4] = x[i]$
for $i = 0, \ldots, 15$: $x[i] = y[i] \lll 11$
for $i = 0, \ldots, 15$: $x[i] = x[i] \oplus x[i+16]$
for $i = 0, \ldots, 15$: $y[i \oplus 1] = x[i+16]$
for $i = 0, \ldots, 15$: $x[i+16] = y[i]$

6.3.2 Attaque générique améliorée

L'auteur de CubeHash a présenté dans [Ber08a] l'attaque « standard par recherche d'antécédent » suivante :

– À partir de (ℓ_h, b, r) nous calculons l'état initial S_0.
– À partir du haché de ℓ_h bits et de $(1024 - \ell_h)$ bits aléatoires, nous inversons $10r$ tours et le XOR d'un bit. Nous obtenons ainsi un état S_f qui précède l'étape de finalisation.
– Maintenant il faut trouver deux séquences de n blocs qui transforment S_0 vers l'avant, et S_f vers l'arrière, en deux états qui coïncident sur les $(1024 - 8b)$ derniers bits.

Il y a 2^{nb} messages de n blocs possibles et nous cherchons une collision sur $(1024 - 8b)$ bits. Pour avoir une probabilité de réussite de $1 - 1/e \approx 0.63$ nous avons besoin de faire 2^{512-4b} requêtes dans chaque direction, il faut donc $2nb > 1024 - 8b$, i.e., $n > 512/b - 4$. Le nombre total d'évaluations de T est approximativement de

$$2r \times \left(\frac{512}{b} - 4\right) \times 2^{512-4b} \approx 2^{522-4b-\log b+\log r} .$$

De plus, [Ber08a] considère que chaque tour de T coûte 2^{11} opérations binaires. Nous obtenons alors un coût en nombre d'opérations binaires de $2^{533-4b-\log b+\log r}$.

Nous avons proposé une accélération de l'attaque précédente en cherchant une collision non seulement sur les états générés après l'insertion de n blocs, mais sur les autres états atteints, c'est-à-dire, aussi sur les états intermédiaires. Ceci est possible grâce à l'absence de la longueur du message dans son padding. Chaque appel à T nous donne un nouveau

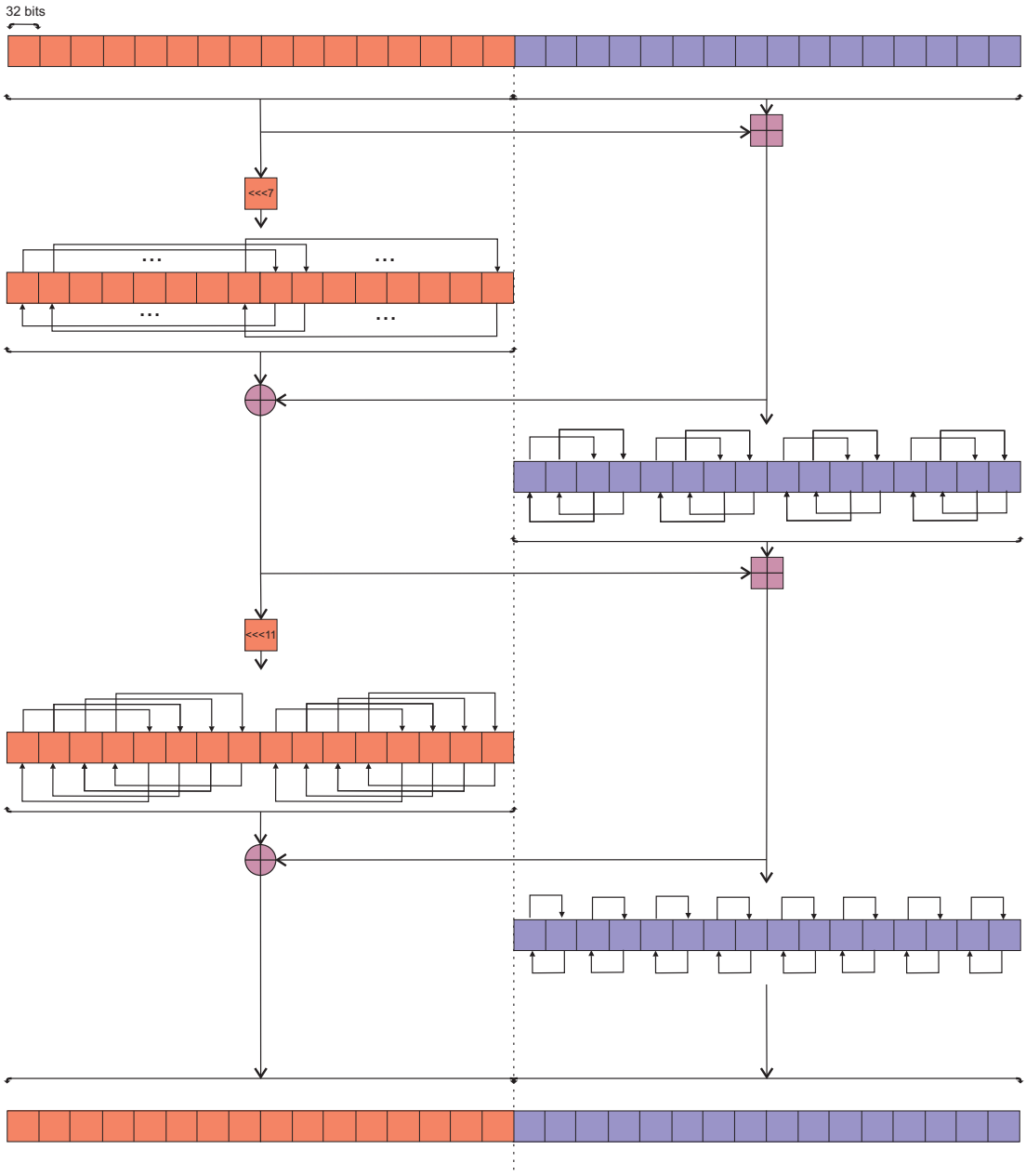

FIG. 6.5 – La fonction de tour T de CubeHash

candidat pour la recherche de collisions. De cette façon nous éliminons le facteur $(512/b-4)$ du coût, ce qui nous donne un coût total de

$$2r \times 2^{512-4b} = 2^{513-4b+\log r}$$

évaluations de T, c'est-à-dire, $2^{524-4b+\log r}$ opérations binaires.

La version proposée de CubeHash-512 a comme paramètres $(\ell_h, b, r) = (512, 1, 8)$, donc notre attaque coûte 2^{523} opérations binaires, contre 2^{532} qui est le coût de l'attaque proposée dans [Ber08a]. Si $r = 8$, notre attaque a besoin de $b > 3$ pour coûter moins de 2^{512} opérations binaires, contre $b > 5$ dans l'attaque originale.

Nous pouvons utiliser la même idée pour réduire le coût de l'attaque en collision. Le coût en nombre d'évaluations de T sera $2^{512-4b+\log r}$ à la place de $2^{521-4b-\log b+\log r}$.

6.3.3 Multicollisions narrow-pipe

À partir des attaques « narrow-pipe » de [Ber08b], nous présentons une attaque par multicollision plus rapide que celles des méthodes [Jou04] ou [DM89, STKT06] (pour des valeurs de b grandes). Notre attaque a besoin du même nombre de calculs que les collisions « narrow-pipe ». Elle exploite le fait que l'état nul est un point fixe pour la fonction de compression T, ce qui est indépendant du paramètre r, et aussi que le padding du message ne prend pas en compte la longueur du message. Depuis un état initial S_0 qui dépend de (ℓ_h, b, r), nous pouvons trouver deux séquences m et m' de n blocs qui transforment S_0 vers l'avant et l'état 0 vers l'arrière en deux états qui collisionnent sur les $(1024 - 8b)$ derniers bits. Nous pouvons trouver une relation de la forme

$$\begin{aligned}
S_0 \oplus m_1 &\xrightarrow{T} S_1 \\
S_1 \oplus m_2 &\xrightarrow{T} \cdots \\
&\cdots \\
\cdots &\xrightarrow{T} S'_1 \\
S'_1 \oplus m'_2 &\xrightarrow{T} 0 \oplus m'_1
\end{aligned}$$

Une fois que nous avons trouvé le chemin vers l'état tout à zéro, nous pouvons rajouter un nombre quelconque de blocs à zéro pour continuer à l'état zéro. Les messages qui collisionnent sont de la forme

$$m \| m' \| 0 \| 0 \| \ldots \| 0 \| \bar{m},$$

où $\bar{m}$ est une séquence arbitraire fixée de blocs.

Si nous utilisons la technique du §6.3.2, cette attaque par multicollision coûte approximativement 2^{513-4b} évaluations de T. Ceci est plus rapide que par exemple, une attaque par paradoxe des anniversaires qui cherche une k-collision, qui coûterait $(k! \times 2^{n(k-1)})^{1/k}$ requêtes, ou que l'attaque de Joux [Jou04] qui coûteraient $\log k \times 2^{4(128-b)}$. Par exemple,

avec $\ell_h = 512$ et $b = 112$, notre attaque trouve 2^{64} collisions en 2^{65} appels à T, contre un nombre supérieur à 2^{512} pour une attaque par paradoxe des anniversaires ou 2^{70} pour l'attaque de Joux.

6.3.4 Comment exploiter les symétries de l'état

Dans la documentation de CubeHash [Ber08d, p.3] l'existence de symétries dans T est mentionnée, et il est dit que l'initialisation a été conçue pour éviter ces symétries. Mais dans [Ber08d] aucun détail n'est donné sur ces symétries. Dans cette section, nous décrivons toutes les classes de symétrie existantes, 15 au total, chacune contenant 2^{512} états, et nous montrons comment exploiter ces symétries pour construire des attaques par recherche d'antécédent.

Les classes de symétrie. Si un état de 32 mots x satisfait $x[0] = x[1]$, $x[2] = x[3]$, ..., $x[30] = x[31]$, alors cette propriété est gardée tout au long de la transformation T, avec une probabilité de 1, pour tout r. Nous pouvons représenter cette symétrie avec un motif, où chaque lettre représente un mot de 32 bits :

```
AABBCCDD  EEFFGGHH  IIJJKKLL  MMNNOOPP .
```

Au total, nous avons trouvé 15 classes de symétrie :

```
C1 :  AABBCCDD  EEFFGGHH  IIJJKKLL  MMNNOOPP
C2 :  ABABCDCD  EFEFGHGH  IJIJKLKL  MNMNOPOP
C3 :  ABBACDDC  EFFEGHHG  IJJIKLLK  MNNMOPPO
C4 :  ABCDABCD  EFGHEFGH  IJKLIJKL  MNOPMNOP
C5 :  ABCDBADC  EFGHFEHG  IJKLJILK  MNOPNMPO
C6 :  ABCDCDAB  EFGHGHEF  IJKLKLIJ  MNOPOPMN
C7 :  ABCDDCBA  EFGHHGFE  IJKLLKJI  MNOPPONM
C8 :  ABCDEFGH  ABCDEFGH  IJKLMNOP  IJKLMNOP
C9 :  ABCDEFGH  BADCFEHG  IJKLMNOP  JILKNMPO
C10 : ABCDEFGH  CDABGHEF  IJKLMNOP  KLIJOPMN
C11 : ABCDEFGH  DCBAHGFE  IJKLMNOP  LKJIPONM
C12 : ABCDEFGH  EFGHABCD  IJKLMNOP  MNOPIJKL
C13 : ABCDEFGH  FEHGBADC  IJKLMNOP  NMPOJILK
C14 : ABCDEFGH  GHEFCDAB  IJKLMNOP  CDABKLIJ
C15 : ABCDEFGH  HGFEDCBA  IJKLMNOP  PONMLKJI
```

Comme nous l'avons déjà dit, chaque classe contient 2^{512} états. Si un état appartient à plusieurs classes, alors son image après T appartient aussi à ces classes ; par exemple, si $S \in (C_i \cap C_j)$, alors $T(S) \in (C_i \cap C_j)$. L'intersection de deux classes vérifie

$$|C_i \cap C_j| \leq 2^{256}.$$

Nous avons donc $|\cup_{i=1}^{15} C_i| \geq 15 \times 2^{512} - 105 \times 2^{256} \approx 2^{515.9} \approx 2^{516}$ états symétriques différents. La symétrie n'est pas préservée dans l'étape de finalisation de CubeHash, à cause de la dissymétrie introduite par le XOR avec 1.

Trouver toutes les classes de symétrie. Maintenant, nous allons prouver que nous avons trouvé toutes les classes de symétrie possibles. Si nous regardons le dessin de la fonction de tour T, nous pouvons avoir une idée des cas où une symétrie va être préservée à travers T. Tout d'abord, il est assez évident que pour préserver une symétrie, il faut avoir la même dans les deux moitiés de l'état, sinon les $\boxplus$ et les XORs détruiraient ces symétries. Donc, les symétries qui vont se conserver à travers T sont celles qui sont conservées après chaque opération de permutation des mots.

Pour le définir plus formellement, une classe de symétrie est un ensemble de contraintes d'égalité qui sera vérifié après l'application d'un tour de T. Nous pouvons représenter un tel ensemble par un ensemble de relations d'égalité entre mots de 32 bits de l'état interne $x[i]$. Soit (i, j) la représentation simplifiée de $x[i] = x[j]$. Une classe de symétrie comme définie précédemment est un ensemble de relations. Par exemple, la classe C_1 est représentée par :

(0,1)	(2,3)	(4,5)	(6,7)	(8,9)	(10,11)	(12,13)	(14,15)
(16,17)	(18,19)	(20,21)	(22,23)	(24,25)	(26,27)	(28,29)	(30,31)

Nous voulons qu'une classe de symétrie soit conservée pendant un tour de T avec une probabilité 1. Comme nous l'avons dit précédemment, pour que ceci se vérifie, il faut que les symétries des deux moitiés de l'état interne soient les mêmes. Nous allons donc parler seulement d'un côté (les 16 premiers mots), les symétries nécessaires dans les 16 derniers mots étant aussi de cette façon aussi définies, puisqu'elles doivent être égales. Autrement dit, (i, j), avec $0 \leq i \leq j \leq 15$, implique $(i + 16, j + 16)$.

Pour trouver toutes les classes de symétrie, nous allons commencer par forcer une relation d'égalité : $(0, k)$, où $1 \leq k \leq 15$. Nous allons voir que, pour que ceci se conserve à travers les 4 opérations de substitution de mots, il n'y a qu'une combinaison possible de symétries pour chaque valeur de k, c'est-à-dire, 15 classes de symétrie au total.

Pour simplifier, nous allons appeler les substitutions $\int_1, \int_2, \int_3, \int_4$ selon leur apparition dans le dessin, du haut vers le bas.

Donc, une fois que nous avons fixé $(0, k)$, tout le reste est déterminé. Il est facile de vérifier que nous pouvons toujours trouver un chemin tel que $\int_1$ nous rajoute une relation d'égalité, $\int_2$ nous rajoute 2 relations (une pour chaque relation connue précédemment), la troisième rajoute 4 relations d'égalité de la même façon, et $\int_4$ rajoute 8 relations d'égalité, ce qui fait 16 au total, de telle façon qu'il ne nous reste plus de degrés de liberté et tout le reste est alors fixé à partir de k.

Nous arrivons donc à la conclusion que les seules classes de symétrie existantes sont celles de la forme :

$$(i, k \oplus i), \text{ pour tout } i \in [0, ..., 15]$$

où k prend des valeurs entre 1 et 15. Nous obtenons donc 15 classes de symétrie, chacune de taille 2^{512}. Par exemple, $k = 1$ définit la classe C_1, et en général, $k = i$ correspond à C_i.

Comment exploiter les états symétriques pour construire des antécédents. En utilisant les classes de symétrie, nous pouvons effectuer une attaque par recherche d'antécédent très similaire à celle de la section 6.3.2, mais avec un coût plus petit pour certaines valeurs de b qui vont nous simplifier la connexion du « meet in the middle ». L'attaque fonctionne de la façon suivante :

- à partir de l'état initial, nous arrivons à un état symétrique (d'une classe quelconque) en utilisant $2^{1024-516-8} = 2^{500}$ blocs de message (pour $b = 1$) ;
- à partir de l'état avant la finalisation, nous arrivons, en allant vers l'arrière, à un autre état symétrique, qui n'est pas nécessairement de la même classe ;
- à partir de ces deux états symétriques des classes C_i et C_j, nous utilisons des blocs de message à zéro dans les deux directions pour arriver à deux états dans $C_i \cap C_j$
- on trouve une collision d'états en essayant $\sqrt{|C_i \cap C_j|}$ messages dans chaque direction.

La complexité des étapes 1 et 2 est un peu près de 2^{501} calculs de T. Le coût des étapes 3 et 4 dépend de i et de j ; mais est borné supérieurement par 2×2^{256} opérations.

Alors, en tous cas, la complexité totale sera de l'ordre de 2^{501} appels à T. Mais cette attaque trouve des messages de taille non autorisée (plus de 2^{128} octets !).

Nous pouvons trouver des antécédents de taille raisonnable avec une variante de cette attaque pour certaines valeurs de b : nous supposons que $b > 4$. À partir de l'état initial nous arrivons à un état qui appartient à une classe C_i. Nous faisons la même chose dans le sens inverse. Pour un b donné la complexité pour arriver à un état symétrique dépend de la classe C_i considérée. Ensuite, nous cherchons une collision dans C_i en insérant des messages qui gardent la symétrie. Par exemple, si $b = 5$ et $C_i = C_1$, nous devons garder l'égalité $x[0] = x[1]$ et nous pouvons alors choisir pour ceci des blocs de message de 5 octets de la forme `X000X` (chaque chiffre représente un octet). Comme chaque C_i a 2^{512} états, le coût pour trouver une collision dans C_i est de l'ordre de 2^{256} requêtes dans chaque direction.

Nous allons détailler une classe C_i facile à atteindre, en fonction de b :

- $5 \leq b < 9$: une des meilleures est C_1, qui donne $(1024 - 2 \times 4 \times 8)/2 = 480$ équations à vérifier ;
- $9 \leq b < 17$: une des meilleures est C_2, qui donne $(1024 - 2 \times 8 \times 8)/2 = 448$ équations à vérifier ;
- $17 \leq b < 33$: une des meilleures est C_4, qui donne $(1024 - 2 \times 16 \times 8)/2 = 384$ équations à vérifier ;
- $33 \leq b < 65$: une des meilleures est C_8, qui donne $(1024 - 2 \times 32 \times 8)/2 = 256$ équations à vérifier.

Si nous devons vérifier n équations, le coût pour arriver à un état symétrique est de l'ordre de 2^n évaluations de T. En comparaison avec l'attaque par recherche d'antécédent de la section 6.3.2, la meilleure amélioration obtenue à partir d'un C_i donné est quand $b = 4d+1$, où d est le nombre de mots de 32 bits qui sépare la première répétition de deux mots.

Pour mieux voir comment cette attaque peut fonctionner, nous allons voir plus en détail les cas suivants :

- si $b \equiv 0 \bmod 8$, il y a $(1024 - 8b)/2 = 512 - 4b$ équations à vérifier, donc nous avons besoin de 2^{512-4b} appels à T,
- si $b \equiv 4 \bmod 8$, il n'y a que $(1024 - 8b - 32)/2 = 496 - 4b$ équations à vérifier, parce que nous pouvons toujours forcer la vérification des relations du premier bloc non XORé avec le bloc de message ;
- en général, quand $b \bmod 8 \leq 4$, nous avons besoin de $2^{512-4(b+(b \bmod 4))}$ appels à T ;
- si $b \bmod 8 > 4$, il y a $(1024 - 8b - 32 + 8(b \bmod 4))/2$ équations à vérifier, ce qui nous donne un coût de $2^{496-4(b-(b \bmod 4))}$.

La formule générale pour obtenir le nombre d'équations est

$$512 - 32\lfloor b/8 \rfloor - 32\lfloor (b \bmod 8)/4 \rfloor - [(\lfloor (b \bmod 8)/4 \rfloor + 1) \bmod 2] \times 8(b \bmod 4) \ .$$

Dans le meilleur cas, ($b \equiv 4 \bmod 8$), l'attaque est 2^{15} fois plus rapide que l'attaque de la section 6.3.2 (dans le cas le pire, $b \equiv 0 \bmod 8$, la complexité sera la même). Quand $b = 5$, l'attaque a besoin de2^{481} appels à T, contre 2^{493} par l'attaque de la section 6.3.2.

6.3.5 Différentielles tronquées sur T

Cette section met en évidence un biais statistique de la transformation définie par 10 tours de T. La fonction de compression de la soumission était composée de 8 tours de T et dans la documentation l'auteur suggérait que faire 10 tours serait inutile car excessif, une prédiction infirmée par nos résultats.

En calculant vers l'arrière à partir d'une différence de poids 1, nous pouvons gagner 3 tours, ce qui nous fait commencer par une différence de poids 64.

Nous calculons vers l'arrière à partir de la différence `80000000` dans $x[16]$. Ce mot a été choisi parce que $x[16] \cdots x[31]$ diffuse moins dans les premiers tours que $x[0] \cdots x[15]$. La différence est dans le bit de poids fort pour minimiser la propagation des différences avec la retenue.

On obtient alors le chemin suivant si nous introduisons la différence de poids 64 :

```
18000000 10000000 08000000 30000000
00000040 00000080 00000000 00000000
00400000 00000000 00400000 01000404
00000003 80802002 00000001 81802004
40000000 08000000 00000000 E8020600
00000000 00000100 00000080 41F001C0
00400008 00000008 00400000 01000404
00000005 80802002 00000001 8080200C
```

Différence après un tour (poids 26) :

```
000E0000 00000000 00000000 00000000
00000000 00000040 00000080 00000040
01000004 00000000 00000004 00000000
00000000 00000000 00002000 00000000
800E0200 00000000 00000000 00000000
00000000 000000C0 00000080 000001C0
00000000 00000004 00000000 00000004
00000000 00002000 0000C000 00000000
```

Différence après deux tours (poids 9) :

```
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 01000000
00000000 00002000 00000000 00002000
00000000 00000000 00000000 80000000
00000000 00000000 00000000 00100000
00000000 00000000 00000000 03000000
00000000 00002000 00000000 00002000
```

Différence après trois tours (poids 1) :

```
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
80000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
```

après un autre tour, cela nous donne avec probabilité 1 la différence

```
80000000 00000000 80000000 00000000
00000400 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 80000000 00000000 80000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
```

Cette différentielle se vérifie avec une petite probabilité. Mais elle se vérifie si l'entrée est la suivante :

```
DFB7AA11 7B2872F1 2848B142 64CB0AF9
17DA36E7 320A7AB2 27621CD8 B6E23031
3BCE90DB 0E496C61 AF4156BD 0B4D857F
4379D4C0 D495EAC9 038BD6E5 72A114CC
29065395 824774C3 F0923C34 28F3B2DD
74251DF6 1A562265 BD8EE5E3 DEFDD839
2804D3BE 89417DC3 F001CE4A 6A5328A8
2BEC024E B2306F17 1F2A7C6C 14BC37B6
```

Si nous considérons 32 bits aléatoires dans $x[25]$ et $x[26]$ (ceux dans les positions $4, \ldots, 19$ pour les deux), la différentielle se vérifie avec une probabilité de 0.985. En utilisant la différentielle précédente, nous avons cherché empiriquement des différentielles tronquées de grande probabilité, basées sur la différence en entrée de poids 64. Pour cela nous avons appliqué à chaque bit de sortie un test de fréquence similaire à celui du NIST [NIS01], avec seuil de décision 0.001 et 2^{20} échantillons. Nous avons trouvé, après 10 tours, 4 bits de sortie sur lequels nous observons une « p-value »[1] inférieure à 0.001 (ce qui montre le caractère non aléatoire), dans les positions 579, 778, 841 et 842. En résumé, pour notre différence de poids 1, nous trouvons les biais décrits après 7 tours, et nous gagnons 3 tours en calculant en arrière et en commençant par la différence de poids 64. Nous n'avons trouvé aucun biais pour plus de 10 tours. Ces observations nous disent donc que 10 tours de T ne se comportent pas comme une permutation aléatoire. Mais cette analyse n'est pas faite dans un scénario réaliste.

Il est intéressant de remarquer ce qu'il se passe dans le modèle linéaire, quand les plus modulo 2^{32} sont remplacés par des XORs :

– un chemin différentiel qui commence par la différence de poids 1 `80000000` dans $x[16]$, retombera sur elle après 47 tours de T ;

[1] http ://csrc.nist.gov/groups/ST/toolkit/rng/documents/nissc-paper.pdf

– avec une recherche exhaustive nous avons montré que, si nous introduisons en entrée une différence de poids 1 quelconque, après un nombre r de tours, les différences influenceront quelques bits des positions paires, ou bien quelques bits des positions impaires de l'état, suivant la différence introduite, mais un bit sur deux restera de façon sûre sans aucune différence. Autrement dit, une différence en entrée dans le modèle linéaire ne peut atteindre que la moitié des bits (ceux des positions paires ou ceux des positions impaires), mais jamais les autres, quel que soit le nombre de tours de T que nous appliquons.

6.3.6 Conclusion

Nous avons réalisé plusieurs analyses de la fonction de hachage proposée pour la compétition du NIST, CubeHash. La version proposée dans un premier temps, CubeHash8-1, a des attaques par recherche d'antécédent à la limite des contraintes du NIST, comme le décrit sa propre documentation. Pour d'autres versions avec des valeurs particulières de b, on peut gagner par rapport à l'attaque générique, avec la technique que nous avons décrite et qui exploite les symétries de l'état. La meilleure attaque pour la nouvelle version proposée, CubeHash16/32 est celle décrite dans sa documentation, mais en tous cas, elle ne respecte pas les critères de résistance à la recherche d'antécédent demandés par le NIST.

6.4 Maraca

Maraca est une nouvelle fonction de hachage paramétrée par une clé, qui a été soumise à la compétition SHA-3 [Jen08] par Robert J. Jenkins. Elle utilise une fonction de tour qui est basée sur une permutation. Cette fonction de tour est appliquée à l'état interne de 1024 bits. La principale innovation de cet algorithme est que chaque bloc de message est inséré 4 fois, à différents instants dans un intervalle de 46 tours. Ceci implique qu'une attaque différentielle devra regarder ce qu'il se passe au moins pendant 46 tours de la fonction. Nous allons présenter ici un nouveau type d'attaque par collision, qui va nous mener vers la collision de deux états internes dans Maraca. Notre attaque a besoin de 2^{237} appels à la fonction de compression, *i.e.* de l'ordre de 2^{24} fois moins que l'attaque générique par collision pour 512 bits de haché. La complexité en temps est aussi plus faible que celle de l'attaque générique. Casser Maraca-512 n'a pas eu d'influence sur la compétition SHA-3, puisque Maraca n'a pas été acceptée au premier tour. Toutefois, notre attaque met en évidence une nouvelle propriété différentielle des permutations qui peut introduire certaines faiblesses inattendues. De plus, nous montrons ici que la résistance à notre attaque est en contradiction avec la résistance classique aux attaques différentielles, et que trouver une bonne permutation pour Maraca fait apparaître certains problèmes intéressants reliés à la construction de fonctions vectorielles booléennes ayant des bonnes propriétés cryptographiques. Une autre attaque a été publiée contre Maraca après la nôtre, par Indesteege et Preneel [IP09], qui exploite la faible non-linéarité de la boîte S. C'est donc une attaque particulière à la permutation Perm choisie dans la version soumise de Maraca, qui n'est pas généralisable car elle ne fonctionne pas pour d'autres permutations de plus grand degré. Par contre, nous montrons qu'une autre boîte S aurait rendu notre attaque plus performante, et qu'aucune boîte S n'aurait pu, avec les contraintes de performances imposées, faire résister Maraca à notre attaque.

Après une courte description de Maraca, nous présenterons notre attaque de façon générale, c'est-à-dire indépendamment de la permutation choisie. En plus de l'attaque générale, nous proposons une amélioration basée sur une méthode de crible, qui a une complexité plus faible que l'attaque générale, mais n'est applicable que si la permutation a une structure spéciale. La permutation Perm utilisée dans Maraca vérifie ces conditions, et nous pouvons donc utiliser notre attaque améliorée. Dans la dernière section nous allons étudier les propriétés de la permutation interne qui lui permettent de résister à notre attaque, et montrer que ces propriétés sont reliées aux propriétés différentielles de la permutation, et qu'il existe un compromis entre les deux propriétés. En fait, nous remarquons que d'autres choix naturels pour la permutation Perm, comme la permutation basée sur la boîte S d'AES, rendraient notre attaque encore plus performante.

Ceci est un travail commun avec Anne Canteaut [CNP09b, CNP09c].

6.4.1 Description de Maraca

Maraca prend comme entrées un message et une clé, et produit un haché de taille h. Le message original est paddé de la façon suivante : la clé de 1024 bits est insérée au début

du message et le nouveau message ainsi obtenu est paddé avec une valeur dépendant de la clé et de la longueur du message pour lui donner une longueur multiple de 1024 bits. Notre attaque par collision utilise des messages possédant tous la même longueur et avec la même clé.

L'état interne de Maraca est de taille 1024 bits et les blocs de messages insérés à chaque tour sont de la même longueur. Chaque bloc du message M_i est inséré quatre fois, aux instants i, $(i+21-6(i \bmod 4))$, $(i+41-6((i+2) \bmod 4))$ et $(i+46)$. Plus précisément, la valeur originale de M_i est insérée au tour i, et des versions de M_i ayant subi une rotation sont insérées aux trois autres tours, avec des rotations de 128 bits, 3×128 bits et 6×128 bits respectivement. Nous allons appeler ces versions avec rotation de M_i : M'_i, M''_i et M'''_i. Le bloc M_i n'est donc pas utilisé au-delà du tour $i + 46$.

La fonction de tour à l'instant i peut être représentée comme suit :

- le nouveau bloc de message M_i qui arrive est inséré pour la première fois en le XORant à l'état interne ;
- une permutation de 1024 bits, Perm, est appliquée à l'état interne ;
- $(M'_{i-3-6((i+2) \bmod 4)} \oplus M''_{i-23-6(i \bmod 4)} \oplus M'''_{i-46})$ est XORé à l'état interne ;
- on applique 2 itérations de Perm à l'état interne.

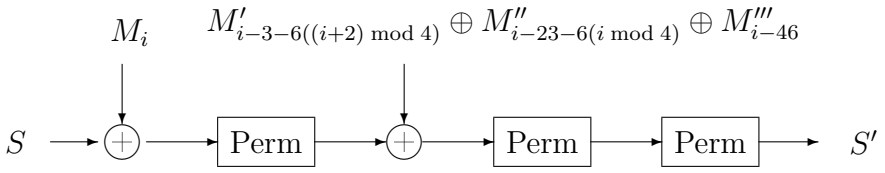

FIG. 6.6 – Tour i dans Maraca

Nous sommes alors prêts pour commencer le tour suivant et introduire le bloc de message M_{i+1}, s'il existe. Si le message est terminé, on utilise le bloc tout à zéro. L'étape d'insertion se termine quand tous les blocs de message ont été utilisés quatre fois, ce qui signifie que pour un message de ℓ blocs, l'étape d'insertion de message sera formée par $(\ell + 46)$ tours. Le haché de h bits est finalement extrait de l'état interne après 30 tours à blanc de Perm.

Comme l'état interne de Maraca a $n = 1024$ bits, l'attaque générique pour trouver une collision interne aura besoin de hacher $2^{\frac{n}{2}}$ messages, donc de faire au moins 46×2^{512} appels à la fonction de tour. En fait, à cause du padding et du fait que chaque bloc de message est inséré à quatre instants différents, nous ne pouvons pas chercher des collisions entre des états internes qui appartiennent à des tours différents. L'attaque générique par collision pour des tailles de haché de h bits nécessite de hacher de l'ordre de $2^{\frac{h}{2}}$ messages, et a besoin d'au moins $46 \times 2^{\frac{h}{2}}$ appels à la fonction de compression. Sa complexité est approximativement de $46 \times 2^{\frac{h}{2}}$ appels à la fonction de compression.

6.4.2 Cryptanalyse de Maraca

Principe général de l'attaque par collision sur l'état interne.

Le principe de notre attaque contre Maraca est de trouver deux messages paddés de la même longueur qui nous mènent au même état interne de 1024 bits. L'attaque exploite le fait que les blocs de message ont la même taille que l'état interne complet. C'est cette propriété qui permet à l'attaquant de contrôler tout l'état interne. Nous allons maintenant décrire le principe général de l'attaque ainsi que la propriété de la permutation interne que nous allons utiliser. Nous montrerons plus tard que la complexité de cette attaque générale, en temps ou en mémoire, peut être parfois plus grande que celle de l'attaque générique. Ceci peut être résolu dans certains cas, comme celui de Maraca, en exploitant la structure algébrique de la permutation interne. Le premier cas intervient quand l'ensemble de différences en entrée D que nous considérons dans l'attaque contient un grand sous-espace affine. Le deuxième cas, qui sera celui utilisé dans Maraca, intervient quand il existe un grand sous-espace linéaire ou affine dont la plupart des éléments appartiennent à D. Ce deuxième cas permet à l'attaquant d'utiliser une phase de crible qui va réduire la complexité en temps de l'attaque générale.

Comment construire deux ensembles qui nous mènent à une collision interne.

Nous allons considérer deux ensembles de messages paddés, avec une même clé donnée K de 1024 bits. Comme tous les messages que nous allons considérer sont composés de 49 blocs de 1024 bits *avant padding*, ils seront tous paddés avec la même valeur pad qui dépend seulement de K et de la longueur du message. Cette valeur n'aura aucune importance dans l'attaque, puisqu'elle sera la même pour tous les messages, et n'apparaîtra dans le calcul du haché qu'après que nous aurons trouvé une collision sur les états internes. Les deux ensembles de messages paddés sont définis comme suit :

$$\mathcal{A} = \{\mathcal{M}_a = (K, a, 0^{47}, m, \text{pad}),\ a \in \{0,1\}^{1024}\}$$

et

$$\mathcal{B} = \{\mathcal{M}_b = (K, b, 0, x, 0^{45}, m, \text{pad}),\ b \in \{0,1\}^{1024}\}$$

où x et m sont deux blocs fixés de 1024 bits qui seront définis plus bas et où 0^i est la suite formée par i fois le bloc de 1024 bits tout à zéro. Pour la suite, les blocs de message seront M_i, où une première valeur de i sera 0, *i.e.*, $M_0 = K$ pour tous les messages que nous allons considérer.

Soit S_a (resp. S_b) l'état interne obtenu au début du tour 49 quand $\mathcal{M}_a$ (resp. $\mathcal{M}_b$) est haché. Nous allons chercher la collision de l'état interne au tour 49, avant la deuxième application de Perm, comme nous pouvons le voir à la figure 6.7. Le tour 49 pour $\mathcal{M}_a$ (resp. $\mathcal{M}_b$) est formé par les opérations suivantes :

- XORer m à l'état interne ;
- appliquer Perm à l'état interne ;
- XORer 0 (resp. x''') à l'état interne ;

– appliquer Perm deux fois.

Ceci vient du fait que tous les blocs de message M_i, $3 \leq i \leq 48$, de $\mathcal{M}_a$ sont nuls, ce qui implique qu'on insère le bloc tout à zéro après la première application de Perm au tour 49. Tous les blocs de message M_i, $3 \leq i \leq 48$, dans $\mathcal{M}_b$ sont nuls sauf $M_3 = x$, ce qui veut dire que x''', qui est x après rotation de 6×128 bits, est XORé à l'état interne après la première application de Perm au tour 49. Ensuite, tous les blocs de message qui seront insérés après le tour 49 sont égaux pour les deux ensembles de messages. Donc, la collision interne arrive quand nous sommes capables de trouver trois blocs de message a, b et m qui vérifient

$$\mathrm{Perm}(S_a \oplus m) = \mathrm{Perm}(S_b \oplus m) \oplus x'''. \tag{6.1}$$

Il est important de remarquer que S_a et S_b sont indépendants de m.

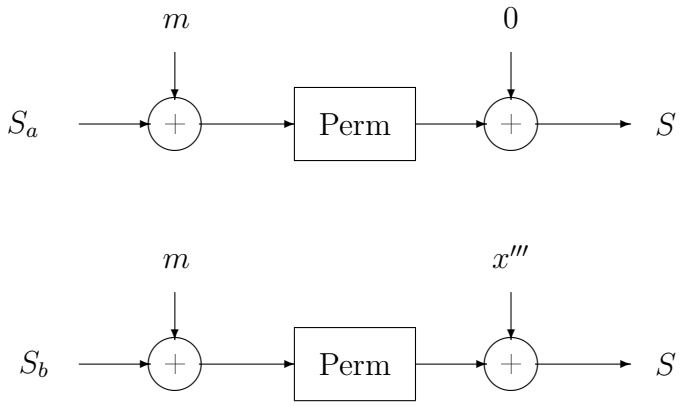

FIG. 6.7 – Début du tour 49 pour $\mathcal{M}_a$ (haut) et $\mathcal{M}_b$ (bas)

Spécificité de la permutation Perm**.**

Nous allons à présent nous intéresser à la propriété de Perm qui rend Maraca vulnérable à l'attaque décrite précédemment. Pour la suite, nous allons décrire l'attaque d'une façon générale qui nous permettra d'analyser la sécurité pour d'autres choix de Perm. Nous allons le faire en fonction de la taille de l'état interne n et de la longueur du haché, h.

L'équation (6.1) avec $v = m \oplus S_a$ nous montre que trouver une collision pour les deux ensembles de messages $\mathcal{A}$ et $\mathcal{B}$ est possible si nous trouvons un couple d'états internes (S_a, S_b) dans $\mathbf{F}_2^n$ tel que

$$\exists v \in \mathbf{F}_2^n, \ \ \mathrm{Perm}(v \oplus S_a \oplus S_b) \oplus \mathrm{Perm}(v) = \delta, \tag{6.2}$$

pour une valeur fixée δ choisie par l'attaquant. Soit $D(\delta)$ l'ensemble de tous les différences en entrée telles que l'équation (6.2) se vérifie, *i.e.*,

$$D(\delta) = \{\alpha \in \mathbf{F}_2^n, \ \ \exists v \in \mathbf{F}_2^n, \ \ \mathrm{Perm}(v \oplus \alpha) \oplus \mathrm{Perm}(v) = \delta\}.$$

En cas d'ambiguïté, nous noterons cet ensemble avec la fonction considérée en indice, par exemple $D_{\mathrm{Perm}}(\delta)$.

L'attaque consiste donc à trouver une couple d'états internes (S_a, S_b) tels que $(S_a \oplus S_b) \in D(\delta)$. En comparaison, l'attaque générique par le paradoxe des anniversaires pour trouver une collision interne cherche à trouver un couple d'états internes (S_a, S_b) dans $\mathbf{F}_2^n$ tel que $S_a \oplus S_b = 0$.

Donc, si nous choisissons d'une façon aléatoire $N_a = N_b = 2^{\frac{n}{2}}|D(\delta)|^{-1/2}$ messages dans $\mathcal{A}$ et dans $\mathcal{B}$, nous trouverons un couple d'états internes (S_a, S_b) au début du tour 49 avec $S_a \oplus S_b \in D(\delta)$. La complexité en données de l'attaque, *i.e.* le nombre d'appels à la fonction de hachage, sera plus petit que la complexité en données de l'attaque générique pour une collision interne s'il existe une différence en sortie δ telle que $|D(\delta)| > 1$. Dans le cas où la taille de l'état interne n est plus grande que la taille du haché h, comme dans le cas de Maraca, notre attaque nous donne une complexité en données plus faible que l'attaque générique par collision s'il existe une différence δ telle que $|D(\delta)| > 2^{n-h}$. Il faut remarquer que, dans notre attaque, chaque appel à la fonction de hachage correspond à 49 appels à la fonction de tour car les premiers 49 blocs de chaque message $\mathcal{M}_a$ et $\mathcal{M}_b$ doivent être utilisés mais le bloc de message 0 est constant et ne doit être évalué qu'une fois. En comparaison, l'attaque générique par collision a besoin d'au moins 46 appels à la fonction tour (et de 30 appels additionnels à Perm) pour chaque message haché.

Complexité en temps de l'attaque générale.

Si l'ensemble de différences en entrée $D(\delta)$ n'a pas une structure particulière qui nous aide à déterminer si deux états internes vérifient que $S_a \oplus S_b \in D(\delta)$, cette tâche peut être très coûteuse. La seule stratégie générale qui pourrait avoir une complexité en temps plus faible que $2^{\frac{n}{2}}$ est d'enregistrer toutes les N_a valeurs de S_a et tous les N_b valeurs de S_b dans deux tables. Alors, toutes les $N_a N_b$ différences doivent être calculées et comparées aux éléments dans $D(\delta)$. La complexité en temps de ceci est

$$N_a N_b \log(|D(\delta)|) = 2^n \frac{\log(|D(\delta)|)}{|D(\delta)|}.$$

L'attaque sera donc plus rapide que l'attaque générique par collision interne seulement si $|D(\delta)| > 2^{\frac{n}{2}}$, et plus rapide que l'attaque par collision seulement si $|D(\delta)| > 2^{n-\frac{h}{2}}$. Mais, en général, comparer les différences $S_a \oplus S_b$ avec les éléments de $D(\delta)$ nécessite de stocker $D(\delta)$, ce qui requiert une complexité en mémoire très grande. Toutefois, cette quantité de mémoire peut être plus petite dans certains cas : par exemple, si Perm correspond à la concaténation de plusieurs copies d'une plus petite boîte S $\ell \times \ell$ (qui peut être ensuite suivie d'une permutation affine), alors, l'attaquant doit stocker les éléments de

$$D_P(\delta') = \{\alpha \in \mathbf{F}_2^\ell, \ \ \exists m \in \mathbf{F}_2^\ell, \ \ P(m \oplus \alpha) \oplus P(m) = \delta'\}$$

seulement pour certains $\delta' \in \mathbf{F}_2^\ell$.

Exploiter la structure algébrique de $D(\delta)$.

Déterminer si $S_a \oplus S_b \in D(\delta)$ pour tous les (S_a, S_b) est beaucoup plus facile quand $D(\delta)$ a une structure algébrique simple. Le cas le plus simple est quand $D(\delta)$ est un sous-espace affine de dimension d (on peut remarquer que $D(\delta)$ ne peut jamais être ou contenir un sous-espace linéaire car il ne contient pas 0 quand $\delta \neq 0$). Alors, nous pouvons l'exprimer comme $D(\delta) = c + \langle e_1, \ldots e_d \rangle$ où $(e_1, \ldots, e_d)$ est une base de l'espace linéaire correspondant et c est une vecteur constant dans $\mathbf{F}_2^n$. Soient $(e_{d+1}, \ldots, e_n)$ $(n-d)$ vecteurs dans $\mathbf{F}_2^n$ tels que $e_1, \ldots e_n$ forment une base de $\mathbf{F}_2^n$. Alors, un élément $x \in \mathbf{F}_2^n$ appartient à $D(\delta)$ si et seulement si, pour tout i, $d+1 \leq i \leq n$,

$$x \cdot e_i = c \cdot e_i,$$

où $x \cdot y$ désigne le produit scalaire. Donc toutes les paires (S_a, S_b) avec $S_a \oplus S_b \in D(\delta)$ peuvent être trouvées en enregistrant la table des mots s_a de $(n-d)$ bits composés des $(n-d)$ coordonnées $(S_a \cdot e_i)$, $d+1 \leq i \leq n$. Alors, pour chaque S_b, l'attaquant calcule les mots de $(n-d)$ bits $s_b = (S_b \cdot e_i)_{d+1 \leq i \leq n}$ et vérifie si $s_b \oplus c$ appartient à la table où c est la constante qui définit l'espace affine. Donc, quand $D(\delta)$ est un sous-espace affine de dimension d, sa taille est 2^d, ce qui implique que la complexité en temps de l'attaque est $2(n-d)N_a = 2(n-d)2^{\frac{n-d}{2}}$. Ceci nécessite une table de $(n-d)2^{\frac{n-d}{2}}$ bits. La complexité en temps de cette attaque est toujours plus petite que celle de l'attaque générique par collision interne, et elle améliore l'attaque générique par collision si $d > n - h$.

Il est important de remarquer que l'attaque utilise seulement le fait que tout élément dans le sous-espace considéré appartient à $D(\delta)$. Donc, la même attaque peut être réalisée si $D(\delta)$ contient un sous-espace affine V de dimension d.

Utiliser une phase de crible.

Dans le cas où le plus grand sous-espace affine inclus dans $D(\delta)$ est de dimension $d \leq n-h$, la complexité en temps de notre attaque sera plus grande que la complexité en temps de l'attaque générique par collision. Alors, l'existence d'un sous-espace linéaire ou affine V plus grand, de dimension d, qui contient plusieurs éléments de $D(\delta)$ peut être utilisée comme crible pour sélectionner les couples (S_a, S_b) qui ont des différences qui appartiennent à $D(\delta)$. L'attaque trouve alors un couple (S_a, S_b) tels que $(S_a \oplus S_b) \in (D(\delta) \cap V)$. La complexité en données a maintenant augmenté et vaut

$$N_a = N_b = 2^{\frac{n}{2}} |D(\delta) \cap V|^{-1/2}$$

ce qui améliore l'attaque générique par collision si

$$|D(\delta) \cap V| > 2^{n-h}.$$

Mais la complexité en temps est beaucoup plus petite. En fait, une fois que la liste beaucoup plus petite des couples avec des différences dans V a été obtenue, toutes les différences

$(S_a \oplus S_b)$ de cette liste peuvent être calculées exhaustivement jusqu'à trouver une différence dans $D(\delta) \cap V$. La phase de crible sélectionne

$$\frac{N_a N_b}{2^{n-d}} = 2^d \frac{1}{|D(\delta) \cap V|}$$

couples (S_a, S_b) parmi les $2^n \frac{1}{|D(\delta)\cap V|}$ couples possibles. La complexité en temps totale est maintenant

$$2(n-d)2^{\frac{n}{2}}(|D(\delta) \cap V|)^{-\frac{1}{2}} + 2^d \log_2(|D(\delta) \cap V|)(|D(\delta) \cap V|)^{-1},$$

où le dernier terme est le coût pour vérifier si une différence dans les listes précédentes appartient à $D(\delta) \cap V$. L'attaque est alors plus rapide que l'attaque générique par collisions si la proportion d'éléments dans V qui appartiennent à $D(\delta)$, *i.e.* si $2^{-d}|D(\delta) \cap V| = |V|^{-1}|D(\delta) \cap V|$ est plus grande que $2^{-\frac{h}{2}}$.

Application à la permutation interne utilisée dans Maraca.

Structure de la permutation interne Perm**.** La permutation interne Perm utilisée dans Maraca est formée par 128 applications parallèles de la même permutation de 8 bis vers 8 bits P, qui a ses trois premières sorties linéaires :

$$\begin{aligned} P_1(x_0, \ldots, x_7) &= (x_0 \oplus x_4 \oplus x_5 \oplus x_7) \\ P_2(x_0, \ldots, x_7) &= (x_1 \oplus x_2 \oplus x_3 \oplus x_5) \\ P_3(x_0, \ldots, x_7) &= (x_1 \oplus x_3 \oplus x_4 \oplus x_5) \end{aligned}$$

et les autres cinq de plus grand degré. Une constante est alors rajoutée aux 1024 bits de l'état interne et une permutation des bits est appliquée aux 1024 bits de sortie que nous avons obtenus. Perm peut être vue comme une fonction qui s'applique à un mot de 128 octets $(b_1, \ldots, b_{128})$, et qui donne en sortie

$$\sigma(P(b_1), \ldots, P(b_{128}))$$

où σ est une permutation des 1024 bits de l'état interne, *i.e.*,

$$\sigma(x_1, \ldots, x_{1024}) = (x_{\pi(1)}, \ldots, x_{\pi(1024)})$$

avec π une permutation de $\{1, \ldots, 1024\}$.

Propriétés différentielles de Perm**.**

Nous allons maintenant regarder la table des différences de la boîte S 8×8 P utilisée dans Perm. Cette table des différences nous permet de déterminer pour chaque différence en sortie non nulle δ l'ensemble des différences en entrée qui peuvent mener à δ, *i.e.*,

$$D_P(\delta) = \{\alpha \in \mathbf{F}_2^8,\ \exists x \in \mathbf{F}_2^8,\ P(x \oplus \alpha) \oplus P(x) = \delta\}.$$

Comme les trois premières coordonnées de P, P_i, $1 \leq i \leq 3$, sont linéaires, nous obtenons que tout $\alpha \in D_P(\delta)$ doit vérifier

$$(P_1(\alpha), P_2(\alpha), P_3(\alpha)) = (\delta_1, \delta_2, \delta_3). \tag{6.3}$$

Alors, $D_P(\delta)$ est inclus dans le sous-espace affine de dimension 5 défini par

$$(\delta_1, \delta_2, \delta_3, 0, 0, 0, 0, 0) \oplus \langle e_4, \ldots, e_8 \rangle$$

où $e_1, \ldots, e_8$ est la base canonique de $\mathbf{F}_2^8$.

Maintenant, nous cherchons la différence en sortie δ pour laquelle la taille de $D_P(\delta)$ est maximale. La plus grande valeur qui peut être obtenue pour cette taille est 21, et peut être atteinte pour 20 différences en sortie δ. Un exemple d'une telle différence de sortie est $\delta = \texttt{0x3}$.

Attaque sur Maraca-512.

Nous allons maintenant décrire l'attaque concrète sur Maraca. Si nous utilisons la notation définie dans les sections précédentes, nous choisissons le bloc de message$x$ tel que sa version après rotation x''' est égale au mot de 128 octets $\sigma(\delta, \ldots, \delta)$ où δ est la différence en sortie pour P qui peut être obtenue à partir de 21 différences en entrée, *e.g.* $\delta = \texttt{0x3}$. Il s'ensuit que toute différence en entrée dans

$$D = \{(\alpha_1, \ldots, \alpha_{128}),\ \alpha_i \in D_P(\delta)\}$$

peut mener vers une différence en sortie x''', c'est-à-dire, pour chaque couple d'états internes (S_a, S_b) tels que $S_a \oplus S_b$ appartient à D, il existe un bloc de message m tel que

$$\mathrm{Perm}(m \oplus S_a) = \mathrm{Perm}(m \oplus S_b) \oplus x'''.$$

Ici, $|D| = (21)^{128}$, ce qui implique que $N_a = N_b = 2^{230.9}$.

Ensuite nous utilisons la structure particulière de P pour cribler les couples (S_a, S_b) : l'ensemble de différences en entrée est inclus dans un sous-espace affine V de dimension 640 (il est important de remarquer que ceci est un cas particulier de l'attaque décrite auparavant où nous pouvions permettre à certains éléments de $D(\delta)$ de ne pas appartenir à V). En utilisant ce sous-espace nous sommes capables de trouver tous les couples (S_a, S_b) dont la différence appartient à V. Le nombre moyen de ces couples (S_a, S_b) est

$$\frac{N_a N_b}{2^{384}} = 2^{78}.$$

Maintenant, pour ces 2^{78} couples favorables d'états internes, nous devons vérifier si $(S_a \oplus S_b)$ appartient à D. Ceci arrive avec une probabilité

$$\frac{|D|}{2^{5\times 128}} = 2^{-78}.$$

Une fois qu'un tel couple a été trouvé, nous pouvons choisir une valeur de m qui rende possible la transition de la différence en entrée $S_a \oplus S_b$ à la différence en sortie voulue. Un tel m peut être construit comme un mot de 128 octets $(\mu_1, \ldots, \mu_{128})$ défini par

$$P(\mu_i \oplus (S_a)_i) \oplus P(\mu_i \oplus (S_b)_i) = \delta$$

où $(S_a)_i$ (resp. $(S_b)_i$) est le i ème octet de S_a (resp. S_b).

Cette méthode nous donne un couple de messages $\mathcal{M}_a \in \mathcal{A}$ et $\mathcal{M}_b \in \mathcal{B}$ tels que

$$\mathrm{Perm}(S_a \oplus m) = \mathrm{Perm}(S_b \oplus m) \oplus x''',$$

i.e., une collision interne après le tour 49. Comme les blocs qui doivent être insérés dans les tours suivants sont les mêmes pour les deux messages, nous pouvons voir clairement que nous obtenons une collision interne qui se maintient à partir du tour 49. L'attaque a donc besoin de moins de $2^{232} \times 49 = 2^{237.5}$ appels à la fonction de compression. La complexité en mémoire est de $2^{239.5}$ bits. À partir de l'analyse de la page 135, nous pouvons déduire que la complexité totale en temps est de $2^{240.5}$ opérations, ce qui est clairement moins que celle de l'attaque générique par collision quand la longueur du haché est plus supérieure ou égale à 512. Il faut remarquer que la complexité de la dernière étape de l'attaque, *i.e.* après le crible, est négligeable.

Résistance de la permutation interne à l'attaque.

Comme notre attaque n'exploite aucune faiblesse classique de Perm, nous pouvons nous demander si tout cela vient d'un mauvais choix pour la permutation Perm et si nous pourrions trouver des Perm plus résistantes à notre attaque.

Une permutation est très vulnérable à notre attaque (dans le sens où notre attaque améliore l'attaque générique par collision) s'il existe une différence en sortie δ telle qu'une des conditions suivantes se vérifie :

1. $|D(\delta)| > 2^{n-\frac{h}{2}}$;
2. il existe un sous-espace (affine) V tel que $|D(\delta) \cap V| > 2^{n-h}$ et la proportion d'éléments dans V qui appartient à $D(\delta)$ est plus grande que $2^{-\frac{h}{2}}$.

La condition 1 correspond à l'attaque sans crible. La condition 2 correspond à l'attaque quand les différences dans le sous-espace V de dimension d sont sélectionnées en premier. D'après l'analyse de la page 135, la complexité en données de l'attaque est alors $2^{\frac{n}{2}}|D(\delta) \cap V|^{-\frac{1}{2}}$, ce qui doit être moins que $2^{\frac{h}{2}}$. Le coût de trouver les différences dans $D(\delta)$ après le crible est alors proportionnel à $|V|/|D(\delta) \cap V|$, ce qui implique que la complexité en temps est plus petite que $2^{\frac{h}{2}}$ si la condition sur la proportion des éléments de $D(\delta)$ dans V est vérifiée.

Il faut aussi remarquer que la condition 2 inclut le cas où $D(\delta)$ contient un sous-espace affine de dimension au moins $n - h$; ceci correspond au cas où la proportion d'éléments de $D(\delta)$ dans V est égale à 1. Nous allons maintenant regarder ces conditions. Il faut remarquer que toutes les deux nous donnent une borne inférieure sur la taille de $D(\delta)$ car la condition 2 implique que $|D(\delta)| > 2^{n-h}$.

Taille de $D(\delta)$ et lien avec la cryptanalyse différentielle.

Comme une petite $|D(\delta)|$ est une condition nécessaire pour résister à notre attaque, nous regardons d'abord sa valeur maximale pour une permutation F sur $\mathbf{F}_2^n$. On note D_F le paramètre

$$D_F = \max_{\delta \in \mathbf{F}_2^n} |D_F(\delta)|.$$

Une permutation adéquate F pour Maraca doit avoir un D_F petit. Toutefois, nous pouvons montrer qu'il existe un compromis entre un petit D_F et une bonne résistance à la cryptanalyse différentielle.

Il est connu que la cryptanalyse différentielle [BS91] exploite le fait que les fonctions non-linéaires utilisées dans une primitive sont telles que les différences entre les images de deux entrées dont la différence est fixée prend la même valeur avec une probabilité élevée. Alors, la résistance d'une fonction $F : \mathbf{F}_2^n \to \mathbf{F}_2^n$ à cette attaque est quantifiée par le paramètre suivant [NK93, NK95],

$$\Delta_F = \max_{\alpha,\beta \in \mathbf{F}_2^n,\, \alpha \neq 0} \Delta(\alpha,\beta) \text{ avec } \Delta(\alpha,\beta) = |\{x \in \mathbf{F}_2^n, F(x \oplus \alpha) \oplus F(x) = \beta\}|.$$

Une fonction avec $\Delta_F = \Delta$ est dite différentiellement Δ-uniforme. Ce paramètre Δ doit être le plus petit possible. Comme toute équation

$$F(x \oplus \alpha) \oplus F(x) = \beta$$

a un nombre pair de solutions x, la valeur minimale de Δ est 2 et les fonctions pour lesquelles $\Delta = 2$ sont nommées « almost perfect nonlinear » (APN). Toutefois, comme l'existence des permutations APN pour un nombre pair de variables était un problème ouvert (récemment, une APN avec $n = 6$ a été donnée par Dillon [Dil09]), mais c'est le seul exemple connu. Donc, la plupart des permutations utilisées en cryptographie symétrique sont différentiellement 4-uniformes ; l'exemple le plus connu est la fonction inverse sur $\mathbf{F}_{2^n}$ utilisée dans l'AES.

Maintenant, la relation entre les deux quantités D_F et Δ_F vient de l'observation suivante

Proposition 6.1. *Soit F une permutation sur $\mathbf{F}_2^n$. Pour tout $\delta \in \mathbf{F}_2^n$, nous avons que :*

$$\begin{aligned} D(\delta) &= \{\alpha \in \mathbf{F}_2^n, \exists x \in \mathbf{F}_2^n \text{ avec } F(x \oplus \alpha) \oplus F(x) = \delta\} \\ &= \{F^{-1}(x \oplus \delta) \oplus F^{-1}(x), \quad x \in \mathbf{F}_2^n\}. \end{aligned}$$

Preuve : Soit $x \in \mathbf{F}_2^n$ une solution de

$$F(x \oplus \alpha) \oplus F(x) = \delta.$$

Avec $y = F(x)$, cette équation peut aussi être écrite comme

$$y \oplus \delta = F(x \oplus \alpha)$$

ce qui signifie

$$F^{-1}(y \oplus \delta) = F^{-1}(y) \oplus \alpha.$$

On en déduit que l'ensemble $D(\delta)$ est formé par toutes les valeurs $(F^{-1}(y \oplus \delta) \oplus F^{-1}(y))$ quand y varie dans $\mathbf{F}_2^n$. □

À partir de cette expression plus simple de $D(\delta)$, on déduit que toute permutation F avec un petit Δ_F a un grand D_F.

Théorème 6.2. *Soit F une permutation sur $\mathbf{F}_2^n$. Si F est différentiellement Δ-uniforme, alors, pour tout $\delta \in \mathbf{F}_2^n$, $\delta \neq 0$, on a*

$$|D(\delta)| \geq \frac{2^n}{\Delta}$$

avec égalité si et seulement si pour tout $\alpha \in \mathbf{F}_2^n$, les équations

$$F(x \oplus \alpha) \oplus F(x) = \delta,$$

ont 0 ou Δ solutions.

Preuve : Soit $\Delta(\delta, \alpha)$ un nombre de solutions $x \in \mathbf{F}_2^n$ de

$$F^{-1}(x \oplus \delta) \oplus F^{-1}(x) = \alpha.$$

Alors on a

$$\sum_{\alpha \in \mathbf{F}_2^n} \Delta(\delta, \alpha) = 2^n$$

et

$$\sum_{\alpha \in \mathbf{F}_2^n} \Delta(\delta, \alpha) \leq \max_{\alpha} \Delta(\delta, \alpha) |D(\delta)| \leq \Delta_{F^{-1}} |D(\delta)|,$$

avec égalité si et seulement si

$$\forall \alpha \in \mathbf{F}_2^n,\ \Delta(\delta, \alpha) \in \{0, \Delta_{F^{-1}}\}.$$

En utilisant le fait que $\Delta(\delta, \alpha)$ est aussi le nombre de solutions x de $F(x \oplus \alpha) \oplus F(x) = \delta$, on obtient notamment

$$\Delta_{F^{-1}} = \max_{\delta \neq 0, \alpha} \Delta(\delta, \alpha) = \Delta_F,$$

et on déduit que, pour tout $\delta \neq 0$,

$$\Delta_F |D(\delta)| \geq 2^n.$$

□

Nous pouvons alors déduire directement le corollaire suivant.

Corollaire 6.3. *Soit F une permutation sur $\mathbf{F}_2^n$. Alors*

$$D_F = \max_{\delta \in \mathbf{F}_2^n} |D_F(\delta)| = 1$$

si et seulement si F a degré 1.

Preuve : Toute fonction vérifie que $\Delta_F \leq 2^n$ avec égalité si et seulement si F a degré 1. □

Ce corollaire implique que, pour tout choix de Perm (avec l'exception du cas trivial et pas réaliste où la fonction de hachage est linéaire), notre attaque a besoin de moins d'appels à la fonction de compression que l'attaque générique par collision sur l'état interne.

Nous allons maintenant étudier quelques choix qui peuvent paraître naturels pour Perm et leur impact sur la complexité de notre attaque. Pour des raisons évidentes d'implémentation, on suppose que Perm est formée par la concaténation de plusieurs copies de la même petite boîte S P, éventuellement suivie d'une application linéaire comme dans la fonction d'origine.

Exemple 6.4. Comme aucune permutation APN avec un nombre pair de variables n'est connue (à part, récemment, une pour 6 variables, qui a été donnée par Dillon [Dil09]), nous pouvons réduire un peu la taille de l'état interne, $n = mk$ avec m impair et choisir pour P une permutation APN sur $\mathbf{F}_2^m$. Par exemple, $m = 9$ et $k = 128$ pourrait être un choix approprié. À partir du théorème 6.2, on peut déduire que, pour tout $\delta \in \mathbf{F}_2^m$ non nul, $|D_P(\delta)| = 2^{m-1}$ car P^{-1} est APN, *i.e.* tous les équations $P^{-1}(x \oplus \delta) \oplus P^{-1}(x)$ ont ou 0 ou 2 solutions. Il s'ensuit que, pour tout $\delta = (\delta_1, \ldots, \delta_k) \in (\mathbf{F}_2^m)^k$ avec les $\delta_i \neq 0$,

$$|D_{\mathrm{Perm}}(\delta)| = 2^{k(m-1)}.$$

Notre attaque (sans crible) a besoin de hacher alors

$$N_a = N_b = 2^{\frac{k}{2}}$$

messages de $\mathcal{M}_a$ et $\mathcal{M}_b$, où k est le nombre de copies de P. Toutes les 2^k différences $(S_a \oplus S_b)$ peuvent être calculées et comparées aux éléments de $D_{\mathrm{Perm}}(\delta)$. À cause de la structure concaténée de Perm, vérifier si chaque $(S_a \oplus S_b)$ appartient à $D_{\mathrm{Perm}}(\delta)$ coûte au plus

$$\sum_{i=1}^{k} \log_2(|D_P(\delta_i)|) = k(m-1) \text{ opérations},$$

ce qui nous donne une complexité totale plus petite ou égale à $k(m-1)2^k$. Ceci améliore l'attaque générique par collision si la longueur du haché est plus grande que $2(k+\log(n-k))$ où n est la taille de l'état interne et k est le nombre de copies de P dans Perm. Pour $k = 128$ et $n = 9\times128$, ceci correspond à $h \geq 276$. La complexité en mémoire correspond au stockage de tous les états internes S_a et S_b et de tous les éléments de $D_P(\delta_i)$ (ce qui nécessite $m2^{m-1}$ bits car l'attaquant peut choisir la même valeur pour tous les δ_i).

Exemple 6.5. Si nous voulons garder les paramètres originaux, *i.e.* $k = 128$ et $m = 8$, un choix naturel pour P est la fonction inverse sur $\mathbf{F}_{2^8}$ comme dans l'AES, ou une permutation linéairement équivalente. Il est bien connu que la fonction inverse sur $\mathbf{F}_{2^m}$, avec m pair, est différentiellement 4-uniforme et que l'équation

$$(x+\delta)^{-1} + x^{-1} = \gamma,\ \delta \neq 0$$

a 4 solutions x si et seulement si $\gamma = \delta^{-1}$ [Nyb93]. Alors, quand x décrit $\mathbf{F}_{2^m}$ et est différent de ces 4 solutions, $((x+\delta)^{-1} + x^{-1})$ prend exactement $(2^{m-1} - 2)$ valeurs différentes car chaque valeur est obtenue pour exactement 2 éléments x. En utilisant la proposition 6.1, on peut déduire que, quand P correspond à la fonction inverse sur $\mathbf{F}_{2^m}$, pour tout $\delta \in \mathbf{F}_{2^m}$ non nul, $|D_P(\delta)| = (2^{m-1} - 2) + 1 = 2^{m-1} - 1$. Alors, avec nos paramètres, $|D_{\mathrm{Perm}}(\delta)| = (2^7 - 1)^{128} = 2^{894.5}$. Notre attaque (sans crible) a besoin de hacher

$$N_a = N_b = 2^{64.7}$$

messages dans $\mathcal{M}_a$ et dans $\mathcal{M}_b$. Même sans aucun crible, l'attaque sera plus rapide que l'attaque générique par collision car examiner toutes les différences $(S_a \oplus S_b)$ nécessite

$$128 \times 895 \times 2^{129.4} = 2^{146} \text{ opérations}$$

et le coût en mémoire est de l'ordre de 2^{76} bits. Alors, si P est remplacée par la fonction inverse dans Maraca, notre attaque devient beaucoup plus efficace et sa complexité en temps est plus petite que la complexité de l'attaque générique par collision quand la longueur du haché est plus grande que 292.

Structure algébrique de $D(\delta)$.

Dans le cas où Perm est telle que $|D_{\mathrm{Perm}}|$ est plus grand que $2^{n-\frac{h}{2}}$, *i.e.* $D_P \leq 2^{8-\frac{h}{2}}$ quand Perm est la concaténation de 128 copies d'une permutation P sur $\mathbf{F}_2^8$, une attaque efficace nécessite que $D(\delta)$ ait une structure particulière, comme expliqué dans les sections précédentes. Par exemple, la proposition 6.1 implique qu'un cas particulier où $D_P(\delta)$ est un sous-espace affine est celui où P^{-1} est quadratique.

Proposition 6.6. *Soit F une permutation sur $\mathbf{F}_2^n$ tel que F^{-1} est de degré 2. Alors, pour tout $\delta \in \mathbf{F}_2^n$, $\delta \neq 0$, $D(\delta)$ est un sous-espace affine.*

Il faut remarquer que ceci n'est pas applicable à la permutation P utilisée dans Maraca car nous avons $\deg(P) = 2$ est non $\deg(P^{-1}) = 2$.

D'une façon plus générale, le structure de $D(\delta)$ est un problème ouvert qui a été énoncé par [BdF98, vDdF00] dans le cas des sous-espaces de codimension 1 : les permutations F^{-1} telles que tous les ensembles $D(\delta)$, $\delta \neq 0$ sont des hyperplans affines sont appelées *fonctions crooked*, et elles correspondent toutes à des fonctions almost bent [CC03, Lemma 5], qui sont un cas particulier des fonctions APN. Toutefois, les seuls exemples connus de fonctions crooked jusqu'à présent sont de degré 2 [BdF98] ; elles correspondent au cas étudié dans la proposition 6.6. Ici, la recherche d'une bonne permutation pour Maraca nous conduit à introduire la notion plus générale suivante.

Définition 6.7. Une fonction F de $\mathbf{F}_2^n$ dans $\mathbf{F}_2^n$ est crooked de codimension d si, pour tout δ non nul de $\mathbf{F}_2^n$, $\{F(x+\delta) \oplus F(x), x \in \mathbf{F}_2^n\}$ est un sous-espace affine de codimension d. En particulier, les fonctions crooked de codimension 1 correspondent aux fonctions crooked classiques définies dans [BdF98].

De la même manière, on peut définir les fonctions faiblement crooked de codimension d comme celles pour lesquelles, pour tout $\delta \neq 0$, $\{F(x+\delta) \oplus F(x), x \in \mathbf{F}_2^n\}$ est inclus dans un sous-espace affine de codimension d. Bien sûr, une fonction faiblement crooked de codimension d l'est aussi pour toute codimension $d' \leq d$. Donc, le paramètre important ici est le plus grand entier d tel que F est faiblement crooked de codimension d. Avec ces notions plus générales, on peut donc poser le problème ouvert suivant, qui généralise la question posée d'habitude dans le cas des fonctions crooked classiques.

Problème ouvert 6.8. *Existe-t-il des permutations F sur $\mathbf{F}_2^n$ avec $\deg(F^{-1}) > 2$ telles que F soit crooked de codimension d pour un certain $d \geq 1$?*

On peut répondre à ce problème dans le cas où F est une permutation monôme $x \mapsto x^s$, où $\mathbf{F}_2^n$ est identifié au corps fini à 2^n éléments, en généralisant les résultats de Kyureghyan [Kyu07].

Proposition 6.9. *Une permutation monôme $x \mapsto x^s$ sur $\mathbf{F}_{2^n}$ est faiblement crooked de codimension d pour un certain $d \geq 1$ si et seulement si elle est de degré 2.*

Conclusions. Nous avons présenté une attaque par collision interne sur Maraca, avec une complexité plus petite que celle de l'attaque générique. À part cette cryptanalyse concrète, l'intérêt principal de notre attaque est qu'elle met en évidence des propriétés différentielles de la permutation interne qui mènent à des faiblesses et qui, à notre connaissance, n'avaient pas été exploitées avant nous. De plus, ces propriétés différentielles sont dans un certain sens, en contradiction avec le critère de sécurité qui correspond à la cryptanalyse différentielle classique. Par exemple, remplacer la permutation originale de Maraca par une boîte-S communément utilisée augmente la vulnérabilité à notre attaque. Trouver une permutation qui résiste à notre attaque pour des hachés de 512 bits n'est pas facile ; il s'agit d'une question liée à certains problèmes ouverts très intéressants sur les fonctions booléennes.

6.5 LANE

LANE [Ind08] est une fonction de hachage proposée à la compétition du NIST par Sebastiaan Indesteege *et al.* Elle fut acceptée au premier tour, mais pas au deuxième, le NIST ayant eu connaissance de nos attaques avant de prendre cette décision. Elle utilise un mode opératoire itératif basé sur le modèle Merkle-Damgård [Dam89, Mer89]. LANE existe en deux versions, une avec $\ell_h = 256$ et une autre avec $\ell_h = 512$. En collaboration avec Krystian Matusiewicz, Ivica Nikolić, Yu Sasaki et Martin Schläffer [MNPN+09], nous avons trouvé des collisions semi-libres sur les deux versions de LANE, c'est-à-dire des collisions sur la fonction de compression. La technique utilisée est une amélioration de l'attaque par rebond publiée par Mendel *et al.* [MRST65]. Nous construisons des collisions sur la fonction de compression LANE-256 avec 2^{96} appels à la fonction de compression et en utilisant un espace mémoire de 2^{88}, et sur de LANE-512 avec 2^{224} appels et un espace mémoire de 2^{128}. Ces collisions ne constituent pas une attaque sur la fonction de hachage, mais elles invalident la réduction de la preuve de Merkle, Damgård et Andreeva [And08]. Il existe un autre résultat sur l'analyse de LANE par Wu *et al.* indépendant du nôtre, utilisant également l'attaque par rebond, mais il ne parvient à cryptanalyser que 3 tours [WFW09].

6.5.1 Description de LANE

LANE [Ind08] est une fonction cryptographique itérative proposée à la compétition SHA-3 du NIST [Nat]. Elle supporte des longueurs de haché de 224, 256, 384 et 512 bits. Nous donnons ici une description du processus global de hachage, sans aller dans les détails, que nous examinerons plus loin.

- D'abord, la valeur initiale H_{-1}, de taille 256 bits pour LANE-{224,256}, et 512 bits pour LANE-{384,512}, est fixée à une valeur dépendant de la taille du haché ℓ_h et, en option, du sel S.
- Le message est ensuite paddé et séparé en blocs M_i de taille 512 bits pour LANE-{224,256}, et 1024 bits pour LANE-{384,512}. Une fonction de compression f est appliquée de façon itérative pour lire les blocs de message un par un, de la façon suivante : $H_i = f(H_{i-1}, M_i, C_i)$, où C_i est un compteur indiquant le nombre de bits de message qui ont été lus jusqu'à présent.
- Finalement, une fois que tous les blocs de message ont été lus, le haché final est obtenu à partir de la dernière valeur de chaînage, de la longueur du message et du sel par un appel supplémentaire à la fonction de compression.

La fonction de compression.

Dans LANE-{224,256} (resp. LANE-{384,512}), la fonction de compression transforme 256 bits (resp. 512) de la valeur de chaînage et 512 bits (resp. 1024) du bloc de message pour générer une nouvelle valeur de chaînage de 256 bits (resp. 512 bits). Elle utilise aussi un compteur C_i. La structure de la fonction de compression est représentée à la figure 6.8. Tout d'abord, la valeur de chaînage et le bloc de message sont traités par la fonction d'expansion

du message, qui produit un état interne de taille double. Ensuite, cet état étendu est traité en deux niveaux. Le premier niveau est composé de six fils, sur chacun on applique une permutation dépendant du numéro de fil, P_0,...,P_5, en parallèle. Le deuxième niveau est formé par deux chemins auxquels on applique aussi deux permutations différentes, Q_0, Q_1, en parallèle.

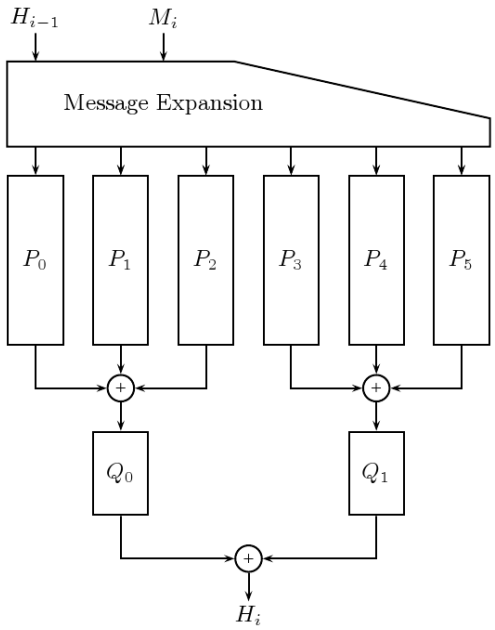

FIG. 6.8 – La fonction de compression de LANE.

L'expansion de message.

L'expansion de message de LANE prend un bloc de message M_i et une valeur de chaînage H_{i-1} et produit l'entrée des six permutations du niveau 1 P_0,...,P_5. Dans LANE-{224,256}, le bloc de message de 512 bits M_i est séparé en quatre blocs de 128 bits chacun : m_0, m_1, m_2, m_3, et la valeur de chaînage de 256 bits H_{i-1} est séparée en deux mots de 128 bits : h_0, h_1, de la façon suivante $m_0||m_1||m_2||m_3 \leftarrow M_i$, $h_0||h_1 \leftarrow H_{i-1}$. Maintenant nous pouvons calculer les six mots de 128 bits suivants qui vont servir pour calculer les entrées pour les six permutations du niveau 1 : $a_0, a_1, b_0, b_1, c_0, c_1$.

$$\begin{aligned} a_0 &= h_0 \oplus m_0 \oplus m_1 \oplus m_2 \oplus m_3\,, \quad a_1 = h_1 \oplus m_0 \oplus m_2\ , \\ b_0 &= h_0 \oplus h_1 \oplus m_0 \oplus m_2 \oplus m_3\,, \quad b_1 = h_0 \oplus m_1 \oplus m_2\ , \\ c_0 &= h_0 \oplus h_1 \oplus m_0 \oplus m_1 \oplus m_2\,, \quad c_1 = h_0 \oplus m_0 \oplus m_3\ . \end{aligned} \tag{6.4}$$

Les valeurs $a_0||a_1$, $b_0||b_1$, $c_0||c_1$, $h_0||h_1$, $m_0||m_1$, $m_2||m_3$ seront les entrées des six permutations $P_0, \ldots, P_5$ que nous décrirons plus bas. Pour les autres versions de LANE, l'expansion de message est la même, mais en doublant toutes les tailles.

Les permutations.

Chaque permutation P_i est appliquée sur un état de 256 bits, qui peut être représenté comme deux états d'AES (2×128 bits) dans le cas de LANE-{224,256} ou quatre états d'AES, (4×128 bits) dans le cas de LANE-{384,512}. La permutation réutilise les transformations SubBytes, MixColumns et ShiftRows de l'AES qui sont appliquées en parallèle aux 2 ou 4 états d'AES qui forment l'état interne. Rappelons ce qu'elles font :

- SubBytes est une fonction non-linéaire qui opère indépendamment sur chaque octet. Elle change la valeur de l'octet selon une table de substitution. Dans le cas de l'AES, cette table est définie par la fonction inverse (suivie par une fonction linéaire).
- MixColumns transforme chaque octet en une combinaison linéaire des 4 octets de sa colonne.
- ShiftRows effectue des décalages à gauche sur les lignes. Le nombre de bits de décalage dépend de la ligne (0 pour la ligne 0 jusqu'à 3 pour la ligne 3).

En plus de ces opérations classiques, LANE utilise trois nouvelles transformations :

- AddConstant somme une valeur différente à chaque colonne d'un fil, c'est-à-dire, aux 8 (resp. 16) colonnes des 2 (resp. 4) états d'AES dans chaque fil de LANE-$\{224, 256\}$ (resp. LANE-{384,512}).
- AddCounter somme une partie du compteur C_i à l'état. Notre attaque ne dépendant pas de ces deux nouvelles transformations, nous n'allons pas les décrire plus en détail.
- SwapColumns (SC) est utilisée pour mélanger les états d'AES parallèles. Soient $x_0||\ldots||x_3||x_4||\ldots||x_7$ les 8 colonnes d'un état d'un fil (pour LANE-{224,256}). Alors SwapColumns échange les deux colonnes les plus à droite de l'état d'AES de gauche avec les deux colonnes les plus à gauche de l'état d'AES de droite :

  $$SC(x_0||x_1||\ldots||x_7) = x_0||x_1||x_4||x_5||x_2||x_3||x_6||x_7 \ .$$

  Pour des valeurs de ℓ_h plus grandes, les changements sont plus complexes et définis par

  $$SC(x_0||x_1||\ldots||x_{15}) = x_0||x_4||x_8||x_{12}||x_1||x_5||x_9||x_{13}||x_2||x_6||x_{10}||x_{14}||x_3||x_7||x_{11}||x_{15} \ .$$

  Comme nous allons le voir plus tard, c'est principalement cette transformation qui introduit les faiblesses que nous exploitons dans notre attaque.

L'application séquentielle de ces 6 transformations constitue un tour interne de la permutation P_i. Le dernier tour interne appliqué omet les transformations AddConstant et AddCounter. Les permutations Q_i sont une légère variante des P_i, que nous ne décrirons pas en détail, notre attaque ne les utilisant pas. Dans LANE-{224,256} (resp. LANE-{384,512}), chaque permutation P_i est formée par 6 tours internes complets (resp. 8) et les permutations Q_i par 3 tours (resp. 4).

6.5.2 Principe de la cryptanalyse de LANE

Ce travail a été commencé dans le groupe de travail sur la cryptanalyse des candidats à la compétition SHA-3, organisé par le réseau d'excellence européen ECRYPT à Graz

(Autriche). Notre attaque cherche une collision après le premier niveau, ce qui explique pourquoi nous nous intéressons peu aux permutations Q_i. Nous avons commencé à analyser la version de 256 bits. Nous avons d'abord réussi à attaquer la fonction de compression pour 4 tours de P_i. Nous avons ensuite réalisé que cette attaque devenait plus facile pour la version complète de 512 bits, puisqu'on dispose alors de plus de degrés de liberté. Nous avons donc réussi à attaquer 5 tours de P_i et, finalement, les 6 tours de la version complète.

Nous allons tout d'abord décrire ici la technique de l'attaque par rebond, puisqu'elle va être la base de nos attaques. Ensuite, nous allons expliquer le principe général de nos attaques sur LANE, et conclure avec les attaques particulières sur les deux versions. Nous parlerons uniquement des attaques sur les versions complètes, celles sur les versions réduites étant très ressemblantes et moins intéressantes.

L'attaque par rebond.

L'attaque par rebond a été publiée par Mendel *et al.* [MRST65]. Elle a tout d'abord été appliquée à des version réduites des fonctions de compression des fonctions de hachage Whirlpool [BR] et `Grøstl` [GKM+08].

Dans l'attaque par rebond on utilise des différentielles tronquées [Knu94] et il existe un lien étroit avec l'attaque de Peyrin [Pey07] sur la fonction de hachage Grindahl [KRT07]. L'idée principale de l'attaque par rebond est d'utiliser les degrés de liberté disponibles pour vérifier des parties des chemins différentiels avec une faible probabilité. Elle est constituée d'une partie intérieure et d'une partie extérieure. L'étape intérieure est une attaque par le milieu qui exploite les degrés de liberté disponibles pour vérifier une partie du chemin différentiel ayant une faible probabilité. Dans l'étape extérieure les valeurs obtenues pendant l'étape précédente sont utilisées pour calculer en arrière et en avant pour essayer de vérifier le reste du chemin différentiel. Nous répétons normalement l'étape intérieure plusieurs fois afin d'obtenir suffisamment de paires pour qu'au moins une d'entre elles vérifie l'étape extérieure.

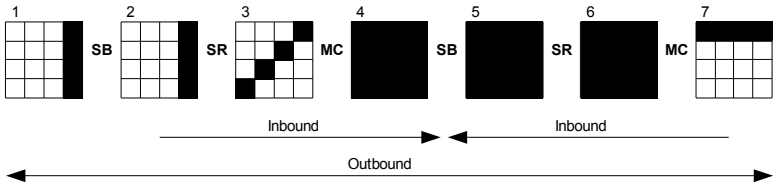

FIG. 6.9 – Exemple d'attaque par rebond sur l'AES.

Dans la figure 6.9 nous pouvons voir un petit exemple sur un état d'AES. Chaque case représente un octet. Les octets en noir représentent des différences. L'idée de l'étape intérieure est la suivante. Nous choisissons d'abord le chemin différentiel, défini par les octets en noir, et ensuite nous attaquons par le milieu avec des valeurs des différences à l'état 1 et à l'état 7 qui vont se retrouver entre les états 4 et 5, qui sont séparés par un SubBytes. Nous savons que pour la boîte S de l'AES, chaque différence en sortie peut être obtenue à

partir de 127 différences en entrée (voir page 141).

Cela signifie que, pour un octet, nous avons une probabilité 1/2 de pouvoir relier la différence dans l'état 4 à la différence dans l'état 5 en choisissant des valeurs particulières pour que, en passant par SubBytes, la différence de l'état 4 se transforme en la différence de l'état 5. Pour les 16 octets, nous avons donc une probabilité de $\frac{1}{2^{16}}$ que toutes les différences à l'état 4 puissent être transformées en les différences à l'état 5.

Pour chaque couple (α, β) de différences aux états 4 et 5 qui peuvent être reliées, il existe au moins 2^{16} valeurs x de l'état 4 qui donnent un état 5 qui diffère de β avec l'état 5 obtenu avec $x \oplus \alpha$. Ces 2^{16} valeurs sont obtenues « gratuitement » dès qu'un couple (α, β) valide a été trouvé. Le coût pour trouver 2^{16} valeurs qui vérifient un chemin différentiel de la forme montrée à la figure 6.9 est donc égal au coût pour trouver un couple (α, β) valide. Ceci nécessite l'essai de 2^{16} couples de différences aux états 2 et 7. Le coût amorti pour obtenir un couple de valeurs vérifiant le chemin différentiel trouvé est donc 1.

À présent, l'étape extérieure se charge de trouver une vraie attaque à partir de ces valeurs : une fois qu'on obtient des valeurs qui vérifient la partie du chemin associée à l'étape intérieure, il faut calculer les états en amont et en aval pour que le reste du chemin soit cohérent. Cette partie est moins générale et dépend plus du système qu'on est en train d'analyser.

L'attaque par rebond sur Lane. À cause de l'expansion de message dans Lane, il y a au moins 4 fils actifs dans une attaque différentielle. Nous cherchons des collisions semi-libres, donc il n'y aura pas de différence dans la valeur de chaînage (h_0, h_1). Ceci signifie que le fil P_3 ne sera pas actif. Comme nous l'avons dit précédemment, nous allons chercher une collision à la fin du premier niveau. Cela va être possible en créant une collision sur les différences de deux fils actifs dans chacune des deux grandes branches de ce niveau. Nous avons donc besoin d'un autre fil non actif dans l'autre branche. Nous avons choisi le fil P_1, (b_0, b_1) n'ont donc pas de différence. Les fils actifs seront donc P_0, P_2, P_4 et P_5. Le chemin différentiel tronqué que nous allons utiliser pour Lane-256 est représenté à la figure 6.13. Notons que ce chemin est similaire à celui montré dans la spécification de Lane [Fig. 4.2, page 33]. Le chemin différentiel tronqué pour Lane-512 décrit à la figure 6.14 est aussi le même que dans la spécification de Lane [Fig. 4.3, page 34]. Rappelons que nous cherchons une collision juste après les fils P, et que nous n'avons pas besoin de considérer les fils Q.

L'idée principale de l'attaque par rebond sur Lane est que nous pouvons appliquer l'étape intérieure sur plusieurs parties du chemin grâce aux degrés de liberté et à la diffusion plus tardive causée par les états d'AES en parallèle et SwapColumns. Comme nous pouvons trouver de nombreuses solutions pour cette étape intérieure de façon indépendante, nous allons essayer de les fusionner plus tard. Nous les stockons donc dans des listes. L'étape extérieure sera chargée de fusionner ces solutions et devra faire en sorte que l'expansion de message de Lane soit compatible avec les états initiaux de tous les chemins.

Pour obtenir la meilleure complexité pour l'attaque, nous fusionnons les listes dans un ordre optimisé, pour réduire les tailles des listes le plus tôt possible et trouver une solution

qui vérifie le chemin différentiel et l'expansion de message et nous donne une collision à la fin. Nous fusionnons donc d'abord les listes pour connecter les étapes intérieures, ensuite nous fusionnons les listes de deux fils dans la même branche pour vérifier l'expansion et obtenir une collision. Finalement, nous fusionnons les deux listes restant entre les deux branches différentes, celle des fils (P_0, P_2) et celle des fils (P_4, P_5), pour vérifier le reste des conditions de l'expansion de message. Nous devons garder la complexité de génération de chaque liste en deçà de $2^{n/2}$, tout en nous assurant d'obtenir une collision à la fin.

Nous allons décrire ici une application de base de l'étape intérieure sur les deux versions de LANE, étape décrite à la figure 6.10 ainsi que les probabilités associées. Dans l'étape intérieure nous allons chercher des différences et des valeurs qui vérifient le chemin différentiel tronqué pour LANE-256 ou LANE-512, montré à la figure 6.10, où les octets actifs sont ceux marqués en noir et les octets en jaune sont ceux dont la valeur est déterminée par l'étape intérieure (en plus de la valeur des octets noirs). Nous n'allons décrire qu'une application de l'étape. Dans l'exemple de la figure 6.10, il y a 16 boîtes S actives entre l'état 4 et l'état 5. À partir de la propriété MDS de MixColumns, ce chemin a au moins un octet actif dans chacune des 4 colonnes de l'état précédant le premier MixColumns et de l'état suivant le deuxième MixColumns, c'est-à-dire, à l'état 2 et à l'état 7.

Dans ces deux états, les octets actifs pourraient aussi être placés dans une autre ligne de la même colonne.

Pendant l'étape intérieure, nous choisissons d'abord de façon aléatoire les différences dans au moins 4 octets après le deuxième MixColumns (l'état 7). Ces différences sont linéairement propagées en amont jusqu'aux 16 octets actifs à la sortie du niveau SubBytes précédent (l'état 5). Ensuite, nous prenons des différences aléatoires dans au moins 4 octets avant le premier MixColumns (l'état 2) et nous les propageons linéairement en aval jusqu'à 16 octets actifs à l'entrée de SubBytes (l'état 4). Enfin, nous cherchons une correspondance entre les entrées et les sorties des 16 boîtes S actives. Pour une seule boîte S, la probabilité qu'une différentielle aléatoire de la boîte S existe est 1/2, comme nous l'avons vu dans la section précédente. Ceci peut être vérifié facilement, plus de détails sont donnés dans [MRST65].

Pour chaque boîte S pour laquelle on relie les différences des états 4 et 5, nous obtenons au moins deux valeurs d'octets possibles vérifiant la différentielle voulue. Nous obtenons donc au moins 2^{16} valeurs pour un état d'AES complet telles que le chemin différentiel pour les différences choisies dans l'état 2 et l'état 7 se vérifie. En d'autres termes, après avoir essayé 2^{16} différences non nulles dans l'état 2 et l'état 7, on obtient au moins 2^{16} solutions pour le chemin différentiel tronqué entre l'état 2 et l'état 7. Donc, la complexité amortie pour obtenir une solution de l'étape intérieure (des différences et des valeurs qui vérifient la différentielle) est 1. Ceci est vrai pour LANE-256 et LANE-512.

Une fois que nous avons trouvé les solutions des étapes intérieures des différentes parties où on les applique, nous devons connecter ces parties du chemin et les propager en aval et en amont dans l'étape extérieure. En amont, nous devons vérifier l'expansion de message, et en aval nous devons trouver une collision des 2 fils P qui appartiennent à la même branche,

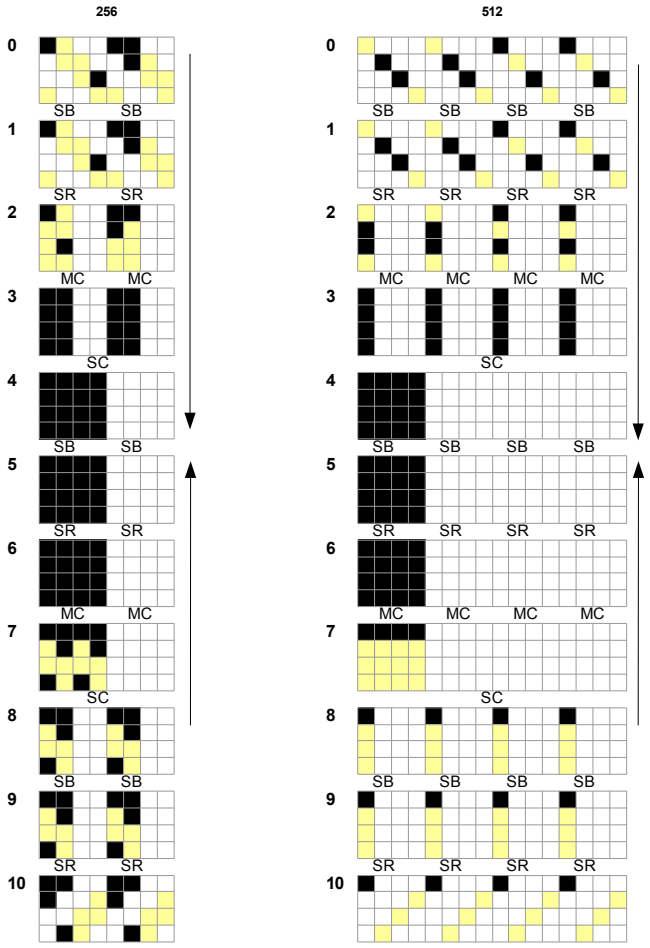

FIG. 6.10 – L'étape intérieure pour LANE-256 et LANE-512. Les octets noirs sont ceux avec des différences, les jaunes sont ceux dont la valeur est déterminée pendant cette étape.

pour chacune des deux branches. Nous allons à présent décrire les conditions permettant d'effectuer l'étape extérieure.

L'expansion de message.

Comme nous allons trouver des collisions semi-libres, on n'aura pas des différences sur (h_0, h_1), comme nous l'avons déjà dit. Donc le fil P_3 ne sera pas actif, et nous avons choisi le fil P_1 c'est-à-dire, (b_0, b_1), comme deuxième fil inactif. Les différences étant nulles dans (h_0, h_1) et (b_0, b_1), nous obtenons, à partir de l'expansion de message pour le fil P_1 (voir équation (6.4)) :

$$\Delta b_0 = 0 = \Delta m_0 \oplus \Delta m_2 \oplus \Delta m_3\,, \quad \Delta b_1 = 0 = \Delta m_1 \oplus \Delta m_2$$

Nous vérifions donc la relation suivante pour les différences dans m_0, m_1, m_2, et m_3 :

$$\Delta m_1 = \Delta m_2 = \Delta m_0 \oplus \Delta m_3. \tag{6.5}$$

En utilisant l'équation (6.4) on obtient, pour les différences dans les mots (a_0, a_1) et (c_0, c_1) du message étendu :

$$\Delta a_0 = \Delta m_1\,,\ \Delta a_1 = \Delta m_3\,,\ \Delta c_0 = \Delta m_0\,,\ \Delta c_1 = \Delta m_2$$

et donc, les relations suivantes entre a_0, a_1, c_0, et c_1 :

$$\Delta a_0 = \Delta c_1 = \Delta a_1 \oplus \Delta c_0. \tag{6.6}$$

C'est avec ceci que nous avons choisi une combinaison possible optimale des octets actifs au début des chemins différentiels, pour LANE-256 comme pour LANE-512. C'est-à-dire, d'un côté il faut prendre en compte le fait qu'à l'état 4 nous voulons avoir tous les octets actifs. Pour ceci, il faut avoir à l'état 0 au moins un octet actif dans chacune des colonnes qui va aboutir dans la partie de gauche de l'état 4, en prenant en compte l'effet de ShiftRows. Le minimum pour ceci est 4 octets dans chaque fil actif. Mais ceci n'est pas possible pour tous les fils si on doit vérifier aussi les conditions précédentes. Nous avons aussi pris en compte le fait que le nombre d'octets actifs à l'état 0 influence la complexité de l'étape qui fusionne les fils, et que cette complexité est déterminée, entre autres, par le nombre maximal de différences pour un fil à l'état 0. C'est pour ceci qu'à la place des autres configurations où ne nous pouvons pas avoir le même nombre de différences par fil, ici nous avons pris un motif ayant le plus petit nombre d'octets actifs qui soit le même pour tous les fils, ce qui nous donne la meilleure complexité. Donc les poids des différences de chaque fil actif pour les deux versions de LANE est 5. Dans un premier temps, nous avions des poids plus grands pour la version LANE-512, mais pendant la rédaction de cette thèse, nous avons remarqué que les différences que nous utilisions n'étaient pas optimales, contrairement à celles présentées à la figure 6.12. Les différences choisies pour LANE-256 sont représentées à la figure 6.11.

Nous tenons à remarquer que ces différences sont optimales mais pas uniques ; nous pouvons choisir n'importe quelle combinaison parmi les octets coloriés en gris qui respecte les conditions précédentes.

En plus des différences, nous devons aussi faire correspondre les valeurs de l'expansion de message. Comme nous l'avons déjà dit, on cherche des collisions semi-libres. Dans ce cas, nous pouvons librement choisir la valeur de la valeur de chaînage (h_0, h_1) de façon à ce que les conditions sur l'entrée (a_0, a_1) soient vérifiées :

$$h_0 = a_0 \oplus m_0 \oplus m_1 \oplus m_2 \oplus m_3\,, \quad h_1 = a_1 \oplus m_0 \oplus m_2$$

Ceci signifie qu'il reste seulement des conditions sur (c_0, c_1), que nous devons satisfaire avec les mots de message m_0, m_1, m_2 et m_3. Comme nous pouvons faire varier les fils P_0,P_2 et P_4,P_5 d'une façon indépendante dans nos attaques, nous pouvons vérifier ces conditions

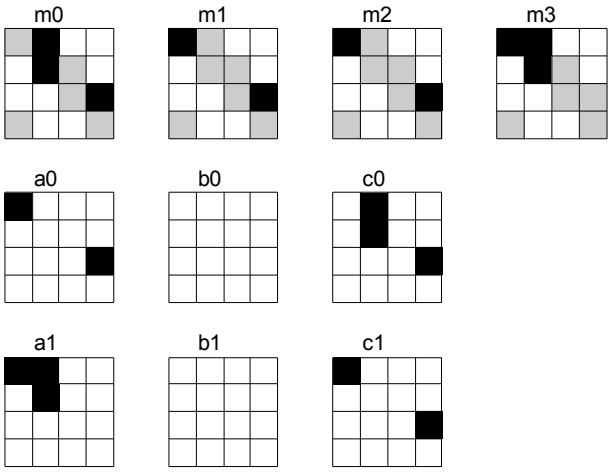

FIG. 6.11 – Différences en entrée choisies pour LANE-256.

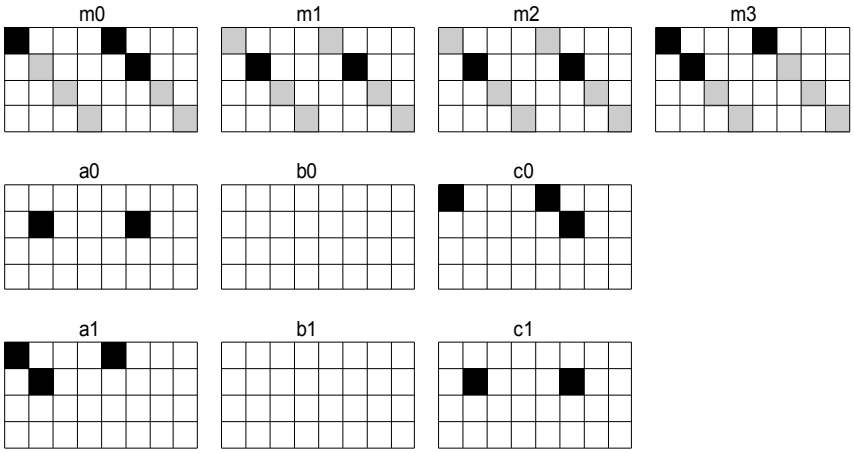

FIG. 6.12 – Différences en entrée choisies pour LANE-512.

en fusionnant les résultats des deux côtés. En utilisant les équations de l'expansion de message, on obtient que (c_0, c_1) à partir des valeurs de (a_0, a_1) :

$$c_0 = a_0 \oplus a_1 \oplus m_0 \oplus m_2 \oplus m_3\,, \quad c_1 = a_0 \oplus m_1 \oplus m_2$$

Nous pouvons réordonner ces équations de façon à avoir tous les termes qui correspondent à P_0,P_2 du côté gauche et tous les termes qui correspondent à P_4,P_5 du côté droit :

$$m_0 \oplus m_2 \oplus m_3 = c_0 \oplus a_0 \oplus a_1\,, \quad m_1 \oplus m_2 = c_1 \oplus a_0 \tag{6.7}$$

Pour fusionner les deux côtés, nous allons calculer, stocker et comparer les valeurs suivantes de chaque liste :

$$v_1 = c_0 \oplus a_0 \oplus a_1\,, \quad v_2 = c_1 \oplus a_0\,, \qquad v_3 = m_0 \oplus m_2 \oplus m_3\,, \quad v_4 = m_1 \oplus m_2. \tag{6.8}$$

Trouver une collision sur les fils P. Quand nous allons vers l'avant dans l'étape extérieure, nous devons trouver une collision sur les différences de P_0 et P_2 : $\Delta P_0 \oplus \Delta P_2 = 0$, et des différences dans P_4 et P_5 telles que $\Delta P_4 \oplus \Delta P_5 = 0$. Nous tenons à remarquer ici que nous pouvons changer de place le dernier MixColumns et le XOR des trois fils P de chaque branche, car ces deux transformations, ainsi que le SwapColumns qui se trouve entre elles, sont linéaires. Pour cette raison nous n'avons qu'à trouver une collision sur les différences après le dernier SubBytes. Si on réussit, nous allons également avoir une collision après le dernier SwapColumns. Nous remarquons aussi que les octets bleus dans la figure 6.13 de LANE-256, ou les octets rouges, bleus et jaunes dans la figure 6.14 de LANE-512, sont indépendants de l'étape intérieure. Nous pouvons donc utiliser les degrés de liberté fournis par ces octets pour trouver une collision des fils P.

6.5.3 Collisions semi-libres sur LANE-256

Dans l'attaque par rebond sur LANE-256, nous construisons une collision semi-libre pour la fonction de compression complète, c'est-à-dire, pour les 6 tours, utilisant 2^{96} appels à la fonction de compression et avec des besoins en mémoire de 2^{88} bits. Nous allons utiliser le chemin différentiel tronqué sur 6 tours donné à la figure 6.13. Nous cherchons une collision après les fils P de LANE et nous utilisons le même chemin différentiel tronqué pour les 4 fils actifs P_0, P_2, P_4 et P_5. Comme nous n'avons pas de différences dans h_0 et h_1, le résultat sera une collision semi-libre. L'attaque sur LANE-256 est constituée des parties principales suivantes, qui sont détaillées à la table 6.3 :

	Type	Match de	#valeurs	#différences	#couples qui le vérifient
1	E. Intérieure	*#0 à #8*	*0*	*1*	$2^{88} = 2^{-8}2^{96}$
2	E. Intérieure	*#9 à #16*	*0*	*0*	2^{96}
3	Fusion des E. Intérieures	*#0 à #16*	*8*	*8*	$2^{56} = 2^{-128}2^{88}2^{96}$
4	Fusion des Fils	*L0-L2,L4-L5*	*0*	*4*	$2^{80} = 2^{-32}2^{56}2^{56}$
5	Expansion de Message	*L02-L45*	*16*	*4*	$1 = 2^{-160}2^{80}2^{80}$
6	Collision	*L0-L2,L4-L5*	*0*	*8*	$2^{64} = 2^{-64}2^{64}2^{64}$
7	Expansion de Message	*L02-L45*	*16*	*0*	$1 = 2^{-128}2^{64}2^{64}$

TAB. 6.3 – Résumé de l'attaque sur LANE-256.

1. Première étape intérieure : on applique l'étape intérieure au début du chemin différentiel, de l'état 2 à l'état 7 pour chaque fil P_0, P_2, P_4, P_5 de façon indépendante.
2. Deuxième étape intérieure : on applique l'étape intérieure au milieu du chemin différentiel, de l'état 10 à l'état 15.
3. Fusion des étapes intérieures : on fusionne les solutions des deux étapes précédentes.
4. Fusion des fils : on fusionne les deux fils voisins P_0,P_2 et P_4,P_5 pour vérifier l'expansion de message correspondant.

5. Expansion de message : on fusionne les deux branches (P_0, P_2) et (P_4, P_5) en vérifiant les conditions correspondant de l'expansion de message.
6. Trouver des collisions : nous allons exploiter les degrés de liberté qui n'ont pas été utilisés pour trouver une collision pour chaque branche, (P_0, P_2) et (P_4, P_5), de façon indépendante.
7. Expansion de message : on fusionne les deux côtés, (P_0, P_2) et (P_4, P_5), pour vérifier les conditions de l'expansion de message pour les octets restant.

Dans le tableau 6.3, nous pouvons voir où chaque étape intervient, sur quelles couleurs des octets de la figure 6.13, combien de différences et de valeurs on veut associer, combien de couples le vérifient, et dans les cas où on commence une liste, de quelle couleur sont les octets que nous faisons varier pour obtenir les couples, ou bien des octets qui interviennent dans l'étape.

Première étape intérieure. L'attaque sur LANE-256 commence par appliquer la première étape intérieure à chacun des 4 fils actifs P_0, P_2, P_4, P_5 d'une façon indépendante. Nous commençons, comme nous l'avons déjà vu, avec 5 octets actifs dans l'état 2 et 8 octets actifs dans l'état 7 et nous choisissons 2^{96} différences aléatoires non nulles pour ces 13 octets. Il faut remarquer que nous pourrions choisir jusqu'à 2^{104} différences. Nous propageons les différences vers l'avant et vers l'arrière jusqu'aux 16 octets actifs de l'état 4 et de l'état 5, avec la transformation SubBytes entre les deux. Nous obtenons au moins 2^{96} associations pour cette étape avec une complexité de 2^{96} (voir section 6.5.2). Pour chaque solution, les octets jaunes et noirs dans la figure 6.13 sont complètement déterminés. Nous pouvons déjà vérifier une condition sur les différences en entrée pour chaque fil en calculant vers l'arrière jusqu'à l'état 0 :

$$P_0 : \Delta a_0[0,0] = \Delta a_1[0,0]\,, \qquad P_4 : \Delta m_0[2,3] = \Delta m_1[2,3]$$
$$P_2 : \Delta c_0[2,3] = \Delta c_1[2,3]\,, \qquad P_5 : \Delta m_2[0,0] = \Delta m_3[0,0]$$

La condition pour chacun de ces octets est vérifiée avec une probabilité 2^{-8} et on stocke les 2^{88} associations valides pour chaque fils P_0, P_2, P_4 et P_5 dans des listes L_0, L_2, L_4 et L_5.

Deuxième étape intérieure. Ensuite nous appliquons à nouveau l'étape intérieure pour faire correspondre les différences reliées par SubBytes entre l'état 12 et l'état 13. Nous commençons avec 2^{64} différences dans les 8 octets actifs de l'état 10 et 2^{32} différences dans les quatre octets actifs de l'état 15. Ainsi, on obtient au moins 2^{96} associations pour la deuxième étape intérieure avec une complexité de 2^{96}. Pour chaque solution, les valeurs des octets rouges et noirs dans la figure 6.13 entre l'état 7 et l'état 18 sont déterminées. Les solutions pour la deuxième étape intérieure pour chaque fil sont gardées dans les listes L'_0, L'_2, L'_4 et L'_5.

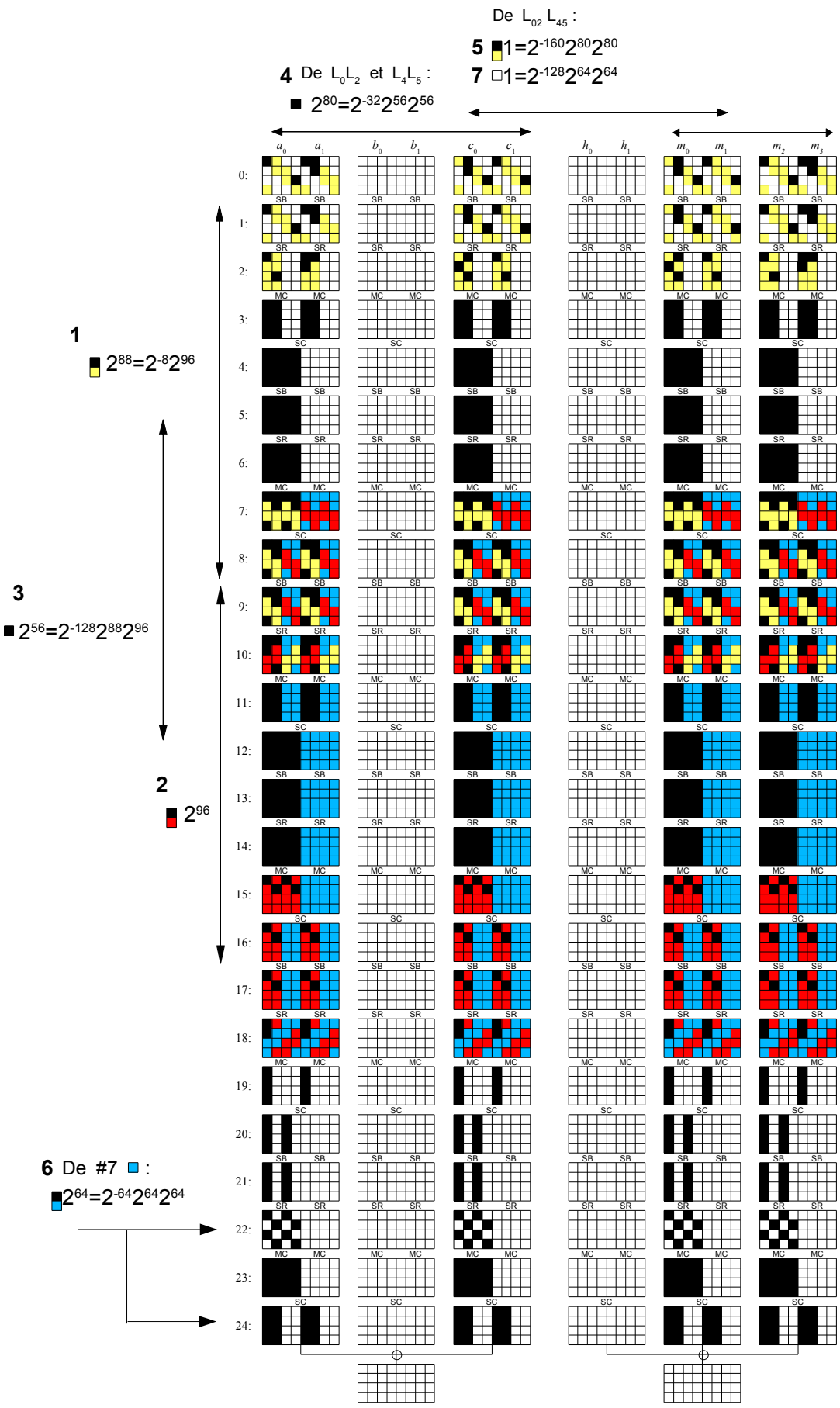

FIG. 6.13 – Le chemin différentiel pour LANE-256.

Fusionner les étapes intérieures. Les deux phases précédentes se chevauchent sur 8 octets actifs entre l'état 7 et l'état 10. Comme nous devons faire coïncider les valeurs autant que les différences de ces octets actifs, nous obtenons 128 conditions binaires. Si on fusionne les 2^{88} solutions de la première étape intérieure et les 2^{96} solutions de la deuxième étape intérieure on obtient $2^{88} \times 2^{96} \times 2^{-128} = 2^{56}$ solutions pour chaque fil. Nous sauvegardons ces résultats dans des listes L_0, L_2, L_4 et L_5 à nouveau. Pour chacune de ces solutions, les différences sont fixées entre l'état 0 et l'état 20 (les octets noirs), et les valeurs entre l'état 0 et l'état 18 (les octets jaunes, rouges et noirs). Chacune de ces solutions vérifie le chemin différentiel de l'état 0 à l'état 24 pour chaque fil.

Fusionner les fils. Ensuite, nous devons fusionner les fils suivant l'expansion de message. D'abord on combine les solutions des fils P_0 et P_2 en fusionnant les listes L_0 et L_2. Pour fusionner ces listes conformément à l'expansion de message, nous devons vérifier les conditions suivantes sur les différences des 4 octets suivants :

$$\begin{aligned} \Delta m_1[0,0] = \Delta m_2[0,0]\,, \quad & \Delta m_0[0,1] = \Delta m_3[0,1] \\ \Delta m_0[1,1] = \Delta m_3[1,1]\,, \quad & \Delta m_0[2,3] = \Delta m_2[2,3]. \end{aligned}$$

Comme cette condition est vérifiée avec une probabilité de 2^{-32} et qu'on fusionne deux listes de taille 2^{56}, on obtient $2^{56} \times 2^{56} \times 2^{-32} = 2^{80}$ associations valides, qu'on sauvegarde dans L_{02}. On répète la même procédure pour les fils P_4 et P_5 en fusionnant les listes L_4 et L_5. On obtient aussi 2^{80} associations stockées dans la liste L_{45}, car nous devons vérifier les conditions sur les différences pour les 4 octets suivants :

$$\begin{aligned} \Delta a_0[0,0] = \Delta c_1[0,0]\,, \quad & \Delta a_1[0,1] = \Delta c_0[0,1] \\ \Delta a_1[1,1] = \Delta c_0[1,1]\,, \quad & \Delta a_0[2,3] = \Delta c_0[2,3]. \end{aligned}$$

Expansion de message. Pour tous les éléments des listes L_{02} et L_{45}, les valeurs sur 32 octets et les différences sur 10 octets de chacun des (a_0, a_1, c_0, c_1) et (m_0, m_1, m_2, m_3) ont été fixées (les octets jaunes et noirs dans l'état 0 de la figure 6.13). Comme les conditions sur les différences de chaque côté séparément ont déjà été satisfaites, nous n'avons plus besoin que d'associer les conditions sur les 4 octets de différences qui se trouvent des deux côtés, en (P_0, P_2) et (P_4, P_5) :

$$\begin{aligned} \Delta a_0[0,0] = \Delta m_1[0,0]\,, \quad & \Delta a_1[0,1] = \Delta m_0[0,1] \\ \Delta a_1[1,1] = \Delta m_0[1,1]\,, \quad & \Delta a_0[2,3] = \Delta m_0[2,3]. \end{aligned}$$

Comme nous l'avons déjà dit, nous pouvons choisir librement (h_0, h_1) pour vérifier les conditions de l'expansion de message sur les valeurs des 16 octets de (a_0, a_1). Pour vérifier les conditions sur les 16 octets de (c_0, c_1) nous devons trouver des correspondances pour les valeurs et différences suivantes, en utilisant les listes L_{02} et L_{45} (voir page 152) :

- 8 octets de v_1 (L_{02}) avec v_3 (L_{45}),
- 8 octets de v_2 (L_{02}) avec v_4 (L_{45}),

– 4 octets de différences dans L_{02} et dans L_{45}.

Comme nous avons 2^{80} éléments dans chaque liste et des conditions sur 160 bits, nous allons trouver $2^{80} \times 2^{80} \times 2^{-160} = 1$ solution. Cette solution est compatible avec l'expansion de message pour tous les fils et est une solution du chemin différentiel tronqué pour tous les fils actifs entre l'état 0 et l'état 18. Toutefois, nous n'obtenons pas encore une collision à la fin des fils P.

Trouver des collisions. Dans cette étape, nous cherchons une collision à la fin des fils P (P_0, P_2) et (P_4, P_5) en utilisant les degrés de liberté qui restent dans la deuxième moitié de l'état. Si nous regardons la figure 6.13, on obtient, pour les valeurs dans l'état 7 :

– les octets noirs, jaunes et rouges sont des valeurs qui ont déjà été déterminées par les étapes précédentes de l'attaque.
– Les octets en bleu sont des valeurs pas encore déterminées et peuvent être utilisés pour faire bouger les différences dans l'état 22.

Donc, pour trouver une collision entre deux fils, nous pouvons encore choisir 2^{64} valeurs pour les octets bleus dans l'état 7 pour chaque fil, et sauvegarder les résultats obtenus dans les listes L_0, L_2, L_4 et L_5.

Mais, il n'y a que 4 octets bleus de l'état 18 (2 dans la première et 2 dans la cinquième colonne) qui influencent les octets actifs de l'état 22. Donc, c'est pour ceci que nous ne pouvons itérer que sur 2^{32} différences dans l'état 22 pour chaque fil. Toutefois, ceci est assez pour trouver une différence qui donne une collision pour chaque branche, car nous cherchons une collision sur 8 octets, et cela ferait $2^{32} \times 2^{32} \times 2^{-64} = 1$. Comme nous faisons varier les bits bleus de l'état 7 2^{64} fois pour chaque fil, nous attendons $2^{64} \times 2^{64} \times 2^{-64} = 2^{64}$ solutions ce qui donne une collision pour chaque liste fusionnée L_{02} et L_{45}.

Expansion de message. Finalement, nous devons satisfaire les conditions de l'expansion de message pour les 32 octets restant de chaque côté. Ainsi, nous allons répéter la procédure que nous avons réalisée dans la cinquième étape pour trouver la moitié du l'état 0, mais maintenant nous n'avons que les valeurs de 32 octets à faire correspondre, sans aucune différence. Nous pouvons à nouveau utiliser les octets restants de (h_0, h_1) pour vérifier les conditions des 16 octets de (a_0, a_1). Comme nous avons 2^{64} solutions dans chacune des listes L_{02} et L_{45}, nous pouvons espérer trouver $2^{64} \times 2^{64} \times 2^{-128} = 1$ couple qui donne une collision pour (c_0, c_1) et donc, une collision pour la fonction de compression complète de Lane-256.

Complexité. La complexité totale est déterminée par la première étape. La complexité totale pour trouver une collision semi-libre pour les 6 tours de Lane-256 est approximativement de 2^{96} appels à la fonction de compression et 2^{88} en mémoire.

6.5.4 Collisions semi-libres sur LANE-512

Dans notre attaque sur LANE-512, nous construisons une collision semi-libre pour les 8 tours de la fonction de compression complète, avec 2^{224} appels à la fonction de compression et une mémoire de 2^{128} bits. Nous utilisons le chemin différentiel décrit à la figure 6.14. De la même façon que sur LANE-256, nous cherchons une collision avant le début du deuxième niveau, exactement à la fin des fils P, et nous utilisons le même chemin différentiel tronqué pour les quatre fils actifs P_0, P_2, P_4 et P_5. L'attaque sur LANE-512 peut être décrite dans les grandes lignes par les parties suivantes :

1. Première étape intérieure : on applique l'étape intérieure au début du chemin différentiel tronqué (de l'état 2 à l'état 7) pour chaque fils P_0, P_2, P_4, P_5 de façon indépendante.
2. Fusion des fils : on fusionne les deux fils voisins (de la même branche) P_0,P_2 et P_4,P_5 et nous vérifions que les différences vérifient les conditions de l'expansion de message.
3. Expansion de message : on fusionne les deux côtés (P_0, P_2) et (P_4, P_5), et on fait en sorte que les conditions de l'expansion de message soient vérifiées, pour les différences et les valeurs.
4. Deuxième étape intérieure : on applique l'étape intérieure au milieu du chemin différentiel tronqué (de l'état 10 à l'état 15).
5. Fusion des étapes intérieures : on fusionne les solutions des deux étapes intérieures.
6. Points de départ : on choisit des valeurs aléatoires pour les octets marrons dans l'état 7 pour avoir assez de points de départ pour les étapes suivantes.
7. Fusion des fils : on fusionne les valeurs des points de départ pour les deux fils voisins P_0,P_2 et P_4,P_5.
8. Expansion de message : on fusionne les deux côtés (P_0, P_2) et (P_4, P_5) en vérifiant les conditions de l'expansion de message pour les valeurs de départ.
9. Troisième étape intérieure : on applique l'étape intérieure à la fin de chaque chemin différentiel tronqué (de l'état 18 à l'état 23).
10. Fusion des étapes intérieures : on fusionne les résultats de la troisième étape intérieure avec le reste en utilisant les degrés de liberté disponibles.
11. Recherche de collisions : on fusionne les deux fils de chaque branche pour trouver une collision pour chaque côté (P_0, P_2) et (P_4, P_5) de façon indépendante.
12. Expansion de message : on fusionne les deux branches (P_0, P_2) et (P_4, P_5) en vérifiant les conditions de l'expansion de message.

Première étape intérieure. Nous commençons l'attaque sur LANE-512 en appliquant la première étape intérieure à chacun des quatre fils actifs P_0, P_2, P_4, P_5 de façon indépendante. Dans chaque fil, nous commençons avec 5 octets actifs à l'état 2 et 4 octets actifs à l'état 7, et nous choisissons 2^{64} différences aléatoires non nulles pour ces 12 octets (nous

aurions pu choisir jusqu'à 2^{72} différences). Nous propageons vers l'arrière et vers l'avant jusqu'aux 16 octets actifs à l'état 4 et à l'état 5, c'est-à-dire, en entrée et sortie de SubBytes. Nous obtenons au moins 2^{64} correspondances pour l'étape intérieure avec une complexité de 2^{64} (voir la section 6.5.2). Pour chaque solution, les octets gris et noirs de la figure 6.14 sont déterminés. Ainsi, nous pouvons déjà vérifier une condition imposée par l'expansion de message sur un octet de différence en entrée pour chaque fil, en calculant vers l'arrière jusqu'à l'état 0 :

$$\begin{aligned} P_0 &: \ \Delta a_0[1,1] = \Delta a_1[1,1] \\ P_2 &: \ \Delta c_0[1,5] = \Delta c_1[1,5] \\ P_4 &: \ \Delta m_0[1,5] = \Delta m_1[1,5] \\ P_5 &: \ \Delta m_2[1,1] = \Delta m_3[1,1] \end{aligned}$$

Les conditions sur chaque fil sont vérifiées avec une probabilité de 2^{-8} et nous sauvegardons les 2^{56} associations valides pour chaque fil P_0, P_2, P_4 et P_5 dans une liste correspondant L_0, L_2, L_4 et L_5.

Fusionner les fils. Ensuite, nous continuons par la fusion des solutions de chaque fil en prenant en compte l'expansion de message. Nous combinons d'abord les résultats des fils P_0 et P_2 en fusionnant les listes L_0 et L_2. Quand on fusionne ces listes, nous avons besoin de vérifier les conditions de l'expansion de message sur les différences des 4 octets suivants :

$$\begin{aligned} \Delta a_1[0,0] = \Delta c_0[0,0]\,, \quad \Delta a_1[0,4] = \Delta c_0[0,4] \\ \Delta a_0[1,1] = \Delta c_1[1,1]\,, \quad \Delta a_0[1,5] = \Delta c_0[1,5]. \end{aligned}$$

Comme cette condition sera vérifiée avec une probabilité 2^{-32} et que nous fusionnons deux listes de taille 2^{56}, on obtient $2^{56} \times 2^{56} \times 2^{-32} = 2^{80}$ associations valides, que nous allons sauvegarder dans L_{02}. Nous faisons la même chose pour P_4 et P_5 en fusionnant les listes L_4 et L_5. On obtient 2^{80} associations pour la liste L_{45}, car nous devons aussi vérifier les conditions sur les différences de 4 octets :

$$\begin{aligned} \Delta m_0[0,0] = \Delta m_3[0,0]\,, \quad \Delta m_0[0,4] = \Delta m_3[0,4] \\ \Delta m_1[1,1] = \Delta m_2[1,1]\,, \quad \Delta m_0[1,5] = \Delta m_2[1,5]. \end{aligned}$$

Expansion de message. Pour toutes les entrées des listes L_{02} et L_{45}, les valeurs sur 32 octets et les différences sur 10 octets de chaque (a_0, a_1, c_0, c_1) et (m_0, m_1, m_2, m_3) ont été fixées (les octets gris et noirs dans l'état 0 de la figure 6.14). Comme les conditions sur

les différences de chaque côté ont déjà été vérifiées, nous n'avons besoin que de vérifier les 4 octets de différences qui interviennent des deux côtés, (P_0, P_2) et (P_4, P_5) :

$$\Delta a_1[0,0] = \Delta m_0[0,0]\,, \quad \Delta a_1[0,4] = \Delta m_0[0,4]$$
$$\Delta a_0[1,1] = \Delta m_1[1,1]\,, \quad \Delta a_0[1,5] = \Delta m_0[1,5].$$

Là encore, nous pouvons choisir librement les valeurs (h_0, h_1) pour vérifier les conditions de l'expansion de message sur les 16 octets correspondants de (a_0, a_1). Pour vérifier les conditions des 16 octets de (c_0, c_1) nous devons trouver des correspondances pour les valeurs et les différences suivantes, en utilisant les listes L_{02} et L_{45} :

- 8 octets de v_1 (L_{02}) avec v_3 (L_{45}),
- 8 octets de v_2 (L_{02}) avec v_4 (L_{45}),
- 4 octets de différences dans L_{02} et dans L_{45}.

Comme nous avons 2^{80} éléments dans chaque liste et 160 conditions binaires, nous espérons trouver $2^{80} \times 2^{80} \times 2^{-160} = 1$ solution. Cette solution vérifie l'expansion de message dans tous les fils et est une solution pour le chemin différentiel tronqué de chaque fil actif de l'état 0 à l'état 10.

Deuxième étape intérieure. Ensuite, nous appliquons à nouveau l'étape intérieure pour associer les différences en entrée et en sortie de SubBytes entre l'état 12 et l'état 13. Après la première étape intérieure, les valeurs de 16 octets de l'état 10 (les octets gris et noirs), et les différences dans 16 octets (1er bloc d'AES) de l'état 12 (les octets noirs) ont déjà été fixées. Ainsi, nous pouvons commencer avec 2^{32} différences dans 4 octets possibles à l'état 15, calculer en arrière jusqu'à l'état 13, et associer les différences en entrée et sortie de SubBytes. Nous espérons trouver au moins 2^{32} solutions pour cette deuxième étape intérieure (voir section6.5.2).

Fusionner les étapes intérieures. Comme résultat de la deuxième étape intérieure, nous avons 2^{32} valeurs pour les 16 octets dans l'état 10 (les octets verts et noirs). À partir de la première étape intérieure, nous avons obtenu une solution pour les 16 octets dans l'état 10 (les octets gris et noirs) aussi. Parmi ces 16 octets, les valeurs de 4 octets actifs (les octets noirs) se chevauchent entre les deux étapes intérieures et la probabilité pour trouver une correspondance est donc de 2^{-32} car nous devons faire coïncider seulement les valeurs sur ces 4 octets. Parmi les 2^{32} solutions de la deuxième étape intérieure, on espère en trouver une qui correspond sur les valeurs de l'état 10. Une fois que nous avons trouvé une correspondance, nous pouvons calculer les valeurs des 12 octets dans l'état 7 qui viennent d'être déterminés, marqués par des octets verts dans la figure 6.14.

Points de départ. Dans cette étape de l'attaque, nous allons calculer un nombre de points de départ qui vont être nécessaires pour les étapes suivantes. Pour chaque fil, nous choisissons des valeurs aléatoires pour 12 octets de l'état 7 (marqués par des octets marrons

à la figure 6.14) et nous calculons les 16 octets correspondants dans l'état 0. Nous répétons ceci 2^{64} fois et sauvegardons les résultats dans les listes correspondantes L'_0, L'_2, L'_4 ou L'_5.

Fusionner les fils. Ensuite, nous fusionnons les listes L'_0 et L'_2 pour obtenir la liste L'_{02}, qui est formée par 2^{128} valeurs pour les 32 octets qui viennent d'être déterminés de (a_0, a_1, c_0, c_1) (les octets marrons dans l'état 0 des fils P_0 et P_2). Plus tard, nous fusionnons les listes L'_4 et L'_5 pour obtenir la liste L'_{45} de taille 2^{128} qui contient les valeurs de 32 octets (m_0, m_1, m_2, m_3).

Expansion de message. Finalement, nous vérifions les conditions de l'expansion de message dans (a_0, a_1) en utilisant les valeurs de (h_0, h_1), et nous utilisons les listes L'_{02} et L'_{45} pour vérifier les conditions dans (c_0, c_1). Comme nous devons faire correspondre 16 octets de (c_0, c_1) et que nous avons 2^{128} éléments dans les deux listes, nous espérons $2^{128} \times 2^{128} \times 2^{-128} = 2^{128}$ couples qui correspondent, et que nous allons stocker dans la liste L_s. Nous utiliserons ces valeurs plus tard dans l'attaque.

Troisième étape intérieure. Maintenant nous augmentons le chemin différentiel tronqué vérifié en appliquant une troisième fois l'étape intérieure entre l'état 18 et l'état 23 pour chaque fil actif. Les valeurs de 16 octets de l'état 18 (les octets verts et noirs), et des différences dans 16 octets (1er bloc d'AES) de l'état 20 (les octets noirs) ont déjà été fixées à cause de la deuxième étape intérieure. Comme nous l'avons fait à la deuxième étape intérieure, nous commençons avec 2^{32} différences de 4 octets dans l'état 23 et on calcule en arrière jusqu'à l'état 21 pour obtenir une association dans SubBytes. Comme nous avons 2^{32} différences de départ, nous espérons trouver 2^{32} solutions pour la troisième phase intérieure, avec des valeurs et différences fixées pour les 16 octets dans l'état 15 (les octets violets et noirs).

Fusion des étapes intérieures. Les valeurs des deuxième et troisième étapes intérieures se chevauchent sur 4 octets actifs (les octets noirs) à l'état 18. Comme nous avons 2^{32} solutions pour la troisième étape intérieure, nous espérons trouver une solution après fusion des deux étapes. Une fois que nous avons trouvé une correspondance, nous pouvons calculer les valeurs des 12 octets de l'état 15 qui viennent d'être déterminés, marqués par des octets violets dans la figure 6.14. Ensuite, nous devons connecter toutes les étapes intérieures. Pour toutes les valeurs possibles des 8 octets de l'état 10 marqués par des octets rouges, nous calculons les 16 octets correspondants dans l'état 15 (2ème bloc d'AES). Si les valeurs calculées sont correctes sur les 4 octets de l'état 15 marqués en violet, on stocke le résultat de chaque fil dans les listes correspondantes L^a_0, L^a_2, L^a_4 et L^a_5. Au total, nous obtenons $2^{64} \cdot 2^{-32} = 2^{32}$ éléments dans chaque liste. Nous répétons la même chose pour les octets marqués en bleu et jaune, et on génère les listes L^b_i et L^c_i pour chacun des fils actifs, avec les indices $i \in \{0, 2, 4, 5\}$. Pour chaque fil, on fusionne trois listes L^a_i, L^b_i et L^c_i et on sauvegarde les 2^{96} résultats dans les listes L^*_i. Pour chaque entrée de ces listes, nous pouvons déterminer toutes les valeurs et différences du fil correspondant.

Trouver des collisions. Dans cette étape de l'attaque, on cherche une collision à la fin des fils P, (P_0, P_2) et (P_4, P_5) en utilisant les éléments des listes L_i^*. Pour trouver une collision à la fin de ces fils, nous devons faire correspondre les différences sur 16 octets de l'état 32 des deux fils actifs pour que $\Delta(P_0 \oplus P_2) = 0$ et $\Delta(P_4 \oplus P_5) = 0$. Nous pouvons vérifier ces conditions de façon indépendante pour chaque côté (P_0, P_2) et (P_4, P_5). Comme nous devons faire correspondre 128 bits et que nous avons 2^{96} éléments dans chaque liste L_i^*, nous espérons trouver $2^{96} \cdot 2^{96} \cdot 2^{-128} = 2^{64}$ collisions pour chaque côté. Nous stockons les éléments correspondants (a_0, a_1, c_0, c_1) pour la collision entre les fils P_0 et P_2 dans la liste L_{02}^* et les éléments (m_0, m_1, m_2, m_3) pour la collision entre les fils P_4 et P_5 dans la liste L_{45}^*.

Expansion de message. Finalement, nous avons besoin de vérifier les contraintes de l'expansion de message pour les 32 octets restant de chaque côté. Ainsi, nous répétons la même procédure que pour les autres parties de l'état 0, et ici nous n'avons qu'à associer des valeurs sur 32 octets et non des différences. À nouveau, nous utilisons les valeurs de (h_0, h_1) pour vérifier les conditions sur (a_0, a_1). Ensuite, on fait correspondre les valeurs des 32 octets dans (c_0, c_1). Comme nous n'avons que 2^{64} entrées dans les deux listes, L_{02}^* et L_{45}^*, la probabilité de succès pour une solution sera $2^{64} \cdot 2^{64} \cdot 2^{-256} = 2^{-128}$. Toutefois, nous pouvons recommencer à partir de l'étape numéro 6 en utilisant un point de départ différent, parmi ceux stockés dans la liste L_s. Comme nous avons 2^{128} éléments dans la liste L_s, nous pouvons répéter les étapes précédentes 2^{128} fois. Ainsi, nous espérons trouver une association valide pour l'expansion de message et donc une collision pour la fonction de compression complète de LANE-512.

Complexité. La complexité totale de l'attaque par rebond sur LANE-512 est déterminée par l'étape de fusion après la troisième étape intérieure. Cette étape a une complexité de 2^{96} appels à la fonction de compression, et est répétée 2^{128} fois. Les besoins en mémoire sont déterminés par les listes les plus grandes, qui sont L'_{02} et L'_{45} (ou L_s) avec une taille de 2^{128}. Ainsi, la complexité totale pour trouver une collision semi-libre sur LANE-512 est de l'ordre de $2^{128} \cdot 2^{96} = 2^{224}$ appels à la fonction de compression et 2^{128} bits en mémoire.

6.5.5 Conclusion

LANE était une des fonctions de hachage favorites pour passer au deuxième tour de la compétition du NIST. Nous avons présenté ici une application de l'attaque par rebond à LANE. Dans l'attaque on utilise un chemin différentiel tronqué avec des différences presque uniquement sur une partie des fils. À cause de la diffusion relativement faible entre les états d'AES paralleles avec SwapColumns, nous pouvons considérer des parties des fils d'une façon indépendante. Tout d'abord nous cherchons des différences et des valeurs (pour des parties de l'état) qui correspondent au chemin différentiel tronqué et aussi qui sont compatibles avec l'expansion de message. Ensuite, nous choisissons des valeurs qui peuvent être modifiées pour que le chemin différentiel soit toujours vérifié ainsi que les conditions de

l'expansion de message. Avec ces degrés de liberté supplémentaires, nous pouvons trouver une paire de messages qui collisionnent à la fin des fils, en même temps qu'ils respectent le chemin différentiel et l'expansion de message.

Dans notre attaque sur LANE, nous trouvons des collisions semi-libres pour les versions complètes de LANE-224 et LANE-256 avec 2^{96} appels à la fonction de compression et une mémoire de 2^{80} bits, et aussi pour la version complète de LANE-512 avec une complexité de 2^{224} appels à la fonction de compression et une mémoire de 2^{128}. Ces collisions sur la fonction de compression ne signifient pas qu'il existe une attaque sur la fonction de hachage, mais elles invalident les réductions de preuves de sécurité de Merkle et Damgård, ou de Andreeva dans le cas de LANE. Une des modifications possibles pour essayer d'invalider ces attaques est de remplacer l'opération SwapColumns par une autre qui diffuse mieux entre les états d'AES parallèles.

Comme travail futur, je pense qu'on pourrait essayer de réduire les besoins en mémoire ou le nombre d'appels à la fonction de compression. Je pense en effet que nous n'avons pas épuisé tous les possibilités pour ce type d'attaque. Par contre, essayer de transformer la collision semi-libre en une collision, me semble plus difficile.

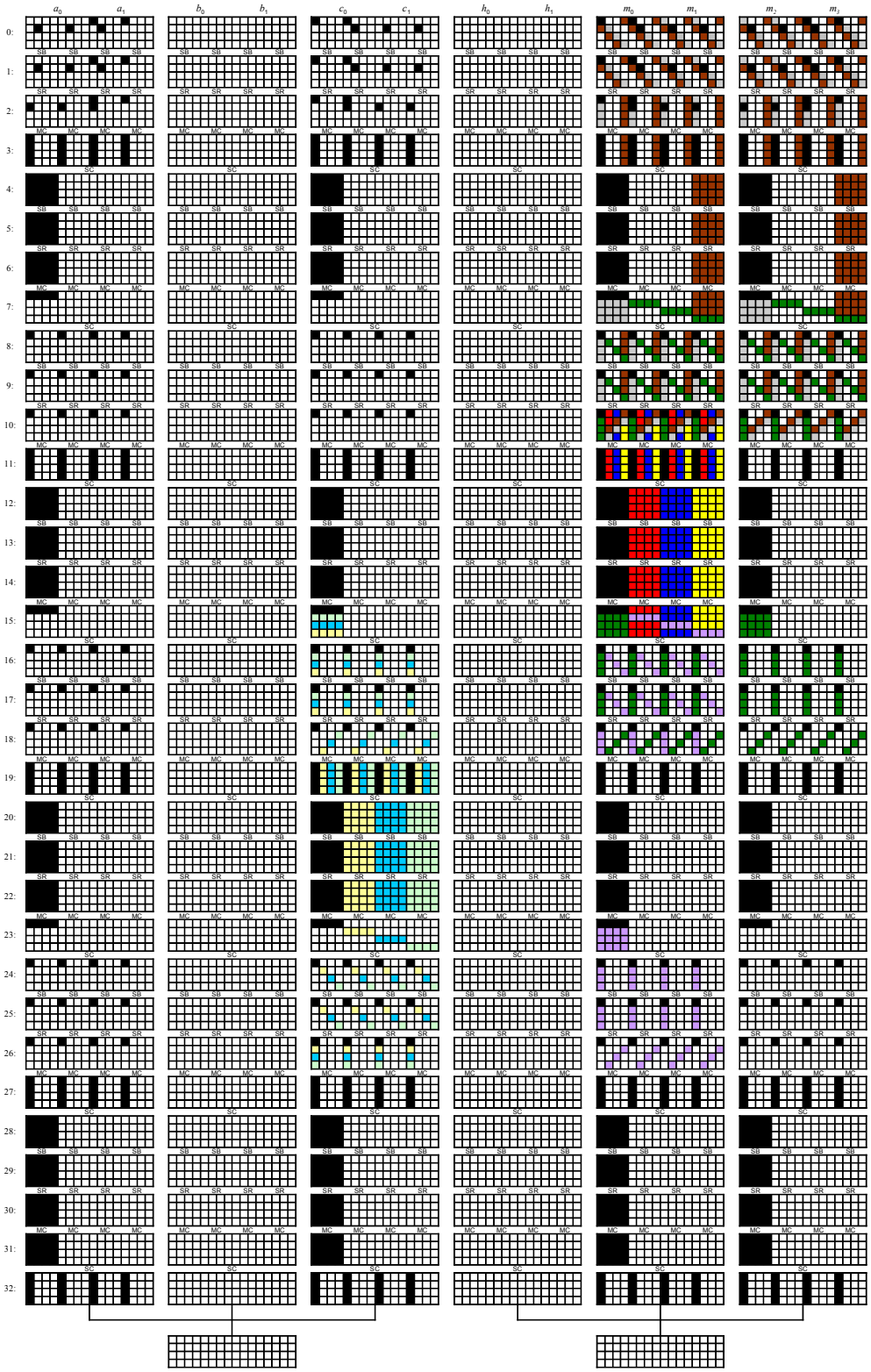

FIG. 6.14 – Le chemin différentiel tronqué pour 8 tours de LANE-512. Le fil P_0 montre le chemin différentiel simple, le fil P_2 d'autres chemins possibles, et les fils P_4 et P_5 montrent l'attaque.

6.6 ESSENCE

ESSENCE [Mar08a, Mar08b] est une soumission de Jason Worth Martin à la compétition SHA-3 qui a été acceptée au premier tour. Elle est composée de deux versions, une avec $\ell_h = 256$, ESSENCE-256, qui travaille sur des mots de 32 bits, et une autre avec $\ell_h = 512$, ESSENCE-512, qui travaille sur des mots de 64 bits. Le mode opératoire est en arbre, et utilise comme fonction de compression un algorithme très simple et élégant basé sur 2 registres à décalage non-linéaires, dont l'un agit sur l'autre. Elle a de bonnes performances. C'est sûrement par sa simplicité et son élégance qu'elle a éveillé l'intérêt de la communauté.

Dans un travail commun avec Andrea Roeck, Jean-Philippe Aumasson, Yann Laigle-Chapuy, Gaëtan Leurent, Willi Meier et Thomas Peyrin, nous avons présenté des attaques par collision [NPRA+09] sur les deux fonctions ESSENCE et sur la construction HMAC [BCK96, NIS02] utilisée avec ESSENCE. Nous allons présenter aussi ici un distingueur sur la fonction de compression. Avec les attaques par collision nous montrons que ESSENCE ne respecte pas le prérequis de sécurité demandé par le NIST pour la compétition SHA-3. D'autres analyses ont été faites sur ESSENCE [MTPT09, MSA+09], mais nos attaques sont les seules connues jusqu'à présent sur les fonctions complètes de ESSENCE, c'est-à-dire, sur les 32 tours. Après la publication de nos attaques, le NIST a publié la liste de candidats au deuxième tour et ESSENCE n'y figurait pas.

6.6.1 Description de ESSENCE

Nous allons ici décrire ESSENCE d'une façon peu détaillée mais suffisante pour comprendre nos attaques. Pour la spécification complète, on peut se reporter à [Mar08a, Mar08b].

Mode opératoire.

ESSENCE traite le message en arbre. Elle construit un arbre binaire équilibré de profondeur limitée, où les feuilles correspondent à des appels à la fonction de compression qui prennent en entrée des blocs de message. Nous n'allons pas détailler ici davantage le mode opératoire, puisque ceci n'a pas d'importance pour notre attaque, et nous renvoyons à [Mar08a, Mar08b].

Dans le message paddé, après les blocs du message, un dernier bloc est rajouté aux données à hacher. Ce bloc contient les paramètres de la fonction et des informations sur le message.

La fonction de compression.

La fonction de compression de ESSENCE, notée G dans [Mar08b], prend en entrée une valeur de chaînage de huit mots et un bloc du message ayant aussi huit mots. Dans le cas de ESSENCE-256 les mots sont de 32 bits, et dans ESSENCE-512 les mots sont de 64 bits.

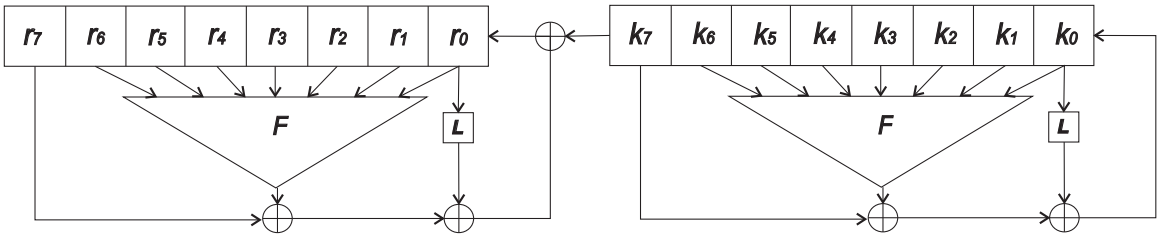

FIG. 6.15 – Fonction de tour de ESSENCE.

La fonction de compression utilise deux NFSRs, chacun de taille 8 mots :
– $r = (r_0, \ldots, r_7)$ sont les valeurs des mots de la valeur de chaînage, et
– $k = (k_0, \ldots, k_7)$ sont les valeurs des mots du bloc de message.

À la figure 6.15 nous pouvons voir comment fonctionne un tour de ESSENCE. À chaque tour, les registres tournent une fois. Le registre avec le bloc de message comme valeur initiale influence celui qui a la valeur de chaînage comme valeur initiale. La partie non-linéaire est introduite par la fonction non-linéaire F, qui est bit-à-bit et qui est décrite à la table 6.4. Elle prend en entrée sept mots, et donne en sortie un mot. La fonction linéaire L mélange les bits des différentes positions dans un même mot. Un tour de cette fonction est une permutation. Dans la documentation d'ESSENCE, au moins 24 tours sont recommandés dans la fonction de compression. La valeur choisie pour la soumission ESSENCE est de 32 tours [Mar08b, §4].

La fonction de compression prend en entrée la valeur de chaînage et le bloc de message, applique 32 tours de cette fonction de tour, applique une transformation un Davies-Meyer sur la valeur de sortie du registre de la valeur de chaînage, et renvoie cette dernière valeur comme nouvelle valeur de chaînage. Autrement dit, la nouvelle valeur de chaînage est le XOR de l'ancienne valeur de chaînage avec l'état du registre r après les 32 tours.

On note x_i le i-ème bit du mot x, et x_0 le bit le moins significatif (LSB). En particulier, $k_{n,i}$ et $r_{n,i}$ sont les i-èmes bits des mots k_n et r_n des registres, $n = 0, \ldots, 7$.

6.6.2 Distingueur

Nous allons présenter ici un distingueur pour la fonction de compression complète de ESSENCE. Il fonctionne tant pour ESSENCE-256 que pour ESSENCE-512. Le distingueur a une complexité de 2^{26} pour la version de 256 bits, et 2^{51} pour la version de 512 bits. Il est basé sur un distingueur très simple sur 30 tours qui a une complexité très petite : 2^7. Nous allons d'abord présenter ce distingueur sur 30 tours pour l'étendre après à la fonction de compression complète, sur 32 tours.

Distingueur sur 30 tours avec complexité 2^7.

Ce distingueur fonctionne de la même façon pour les deux versions d'ESSENCE. Il utilise le chemin différentiel que nous pouvons voir sur la figure 6.16, et qui est le même chemin que celui utilisé pour le distingueur sur les 32 tours.

$$
\begin{aligned}
F(a,b,c,d,e,f,g) = \quad & abcdefg \oplus abcdef \oplus abcefg \oplus acdefg \oplus \\
& abceg \oplus abdef \oplus abdeg \oplus abefg \oplus \\
& acdef \oplus acdfg \oplus acefg \oplus adefg \oplus \\
& bcdfg \oplus bdefg \oplus cdefg \oplus \\
& abcf \oplus abcg \oplus abdg \oplus acdf \oplus adef \oplus \\
& adeg \oplus adfg \oplus bcde \oplus bceg \oplus bdeg \oplus cdef \oplus \\
& abc \oplus abe \oplus abf \oplus abg \oplus acg \oplus adf \oplus \\
& adg \oplus aef \oplus aeg \oplus bcf \oplus bcg \oplus bde \oplus \\
& bdf \oplus beg \oplus bfg \oplus cde \oplus cdf \oplus def \oplus \\
& deg \oplus dfg \oplus \\
& ad \oplus ae \oplus bc \oplus bd \oplus cd \oplus \\
& ce \oplus df \oplus dg \oplus ef \oplus fg \oplus \\
& a \oplus b \oplus c \oplus f \oplus 1,
\end{aligned}
$$

TAB. 6.4 – La fonction non-linéaire F de ESSENCE

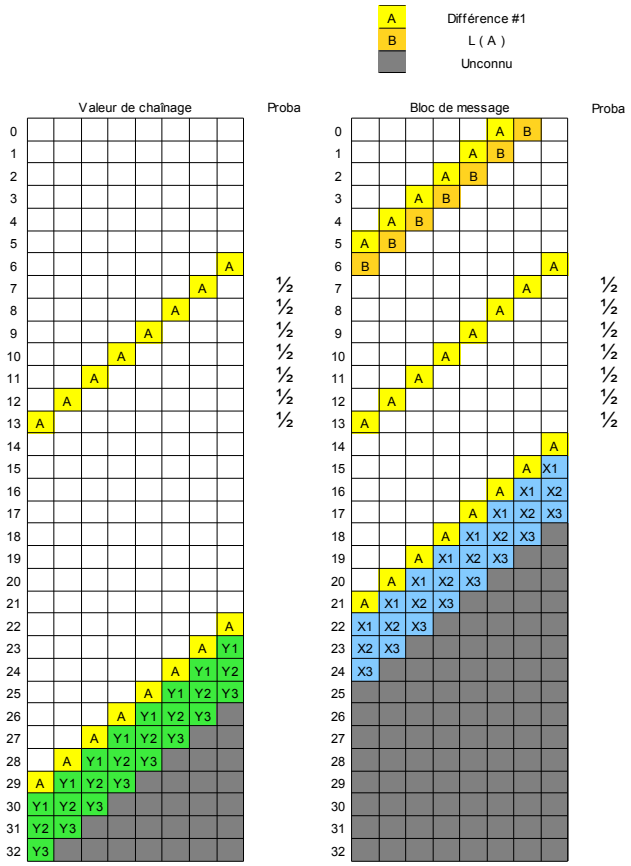

FIG. 6.16 – Chemin différentiel pour le distingueur

Comme nous pouvons le voir, A est une différence de poids 1 et $B = L(A)$. La procédure principale est :

- d'abord, nous trouvons 2^7 paires de messages qui vérifient les conditions différentielles pour les instants # 0 à # 6. Une technique que nous pouvons utiliser pour le faire sera décrite dans la section suivante sur les collisions pour un cas plus compliqué, ceci étant simplement dû au petit poids des différences. Comme nous allons le voir plus tard, ceci peut être fait avec un coût négligeable. Parmi ces 2^7 paires, nous allons trouver une paire qui vérifie le chemin différentiel jusqu'à l'instant # 14, car la probabilité que ceci arrive est approximativement 2^{-7} pour une paire qui vérifie déjà les étapes #0 à #6.
- Une fois que nous avons trouvé cette paire de messages, pour les 2^7 valeurs de chaînage que nous allons utiliser pour insérer la paire de blocs trouvée, nous espérons qu'une parmi elles va avoir une différence Y_1 dans le premier mot de l'état # 30, où $Y_1 = 0$ ou $Y_1 = A$.
- Comme nous n'avions aucune différence sur la valeur de chaînage en entrée, nous aurons toujours la même différence Y_1 dans le premier mot après l'application de la transformation Davies-Meyer.

Nous pouvons remarquer que ce chemin différentiel mène à une attaque par collision sur 21 tours avec une complexité 2^7. En effet, une fois que nous avons trouvé une paire de messages qui vérifie les conditions voulues, si nous introduisons les deux messages pour la même valeur de chaînage, pour chacune des 2^7 valeurs de chaînage que nous essayons, nous allons trouver une collision à la sortie de la fonction de hachage qui est une version réduite de ESSENCE avec seulement 21 tours dans la fonction de compression.

Étendre le distingueur pour ESSENCE complet (2 tours de plus).

Pour distinguer ESSENCE avec ces 32 tours, nous devons regarder ce qu'il se passe avec X_1, X_2, X_3, Y_1, Y_2 et Y_3 (voir figure 6.16). Nous allons calculer la probabilité que $Y_3 = 0$, car ceci va déterminer la complexité du distingueur. Si à la place de 0, nous calculons la probabilité d'obtenir une autre différence fixée, le distingueur fonctionnera également. Cette probabilité va être différente selon que nous regardons ESSENCE-256 ou ESSENCE-512.

Comment calculer la probabilité que $Y_3 = 0$. Tout d'abord, nous allons introduire les notations que nous allons utiliser dans cette section.

Notations. Nous allons travailler avec des ensembles d'indices. Nous utilisons :

- $I(w) = \{i : w_i = 1\}$: ensemble de tous les indices i du mot w tel que le i-ème bit de w est 1.
- $\mathcal{A}, \mathcal{B}, \ldots$: nous notons les ensembles d'indices par de lettres majuscules et calligraphiques.
- $1_{\mathcal{A}}$: Soit $\mathcal{A}$ un ensemble d'indices. Alors $1_{\mathcal{A}}$ est le mot avec des 1 dans tous les indices dans $\mathcal{A}$. En particulier, $1_{I(w)} = w$.

- $1_{\subseteq \mathcal{A}}$: un mot avec des 1 dans certains indices de l'ensemble $\mathcal{A}$, *i.e.* il existe un ensemble $\mathcal{B} \subseteq \mathcal{A}$ tel que $1_{\subseteq \mathcal{A}} = 1_{\mathcal{B}}$.
- R : mot aléatoire.
- $\vee$: ou bit-à-bit.

ESSENCE-256. Les relations suivantes sont toujours vraies :

$$\begin{array}{lcllcl} X_1 & = & L(A) \oplus 1_{\subseteq I(A)} & Y_1 & = & 1_{\subseteq I(A)} \\ X_2 & = & L(X_1) \oplus 1_{\subseteq I(A \vee X_1)} & Y_2 & = & L(Y_1) \oplus 1_{\subseteq I(A)} \oplus X_2 \\ X_3 & = & L(X_2) \oplus 1_{\subseteq I(A \vee X_1 \vee X_2)} & Y_3 & = & L(Y_2) \oplus 1_{\subseteq I(A \vee Y_2)} \oplus X_3 \end{array}$$

Nous supposons que $1_{\subseteq I(A)} = 0$ dans les équations de Y_1 et Y_2, ce qui arrive avec une probabilité approximativement de 2^{-2}. Alors, nous avons que $Y_2 = X_2$ et nous pouvons écrire

$$Y_3 = 1_{\subseteq I(A \vee X_1 \vee X_2)} \oplus 1_{\subseteq I(A \vee X_2)}. \tag{6.9}$$

Dans le cas de ESSENCE-256, après avoir fait une recherche exhaustive, nous avons trouvé que les indices optimaux pour la position du bit de A sont $18, 19$ et 20. Pour chacun d'entre eux, nous avons une probabilité approximativement de 2^{-17} de trouver, dans l'équation (6.9), $Y_3 = 0$. Ceci nous donne une probabilité totale pour les trois étapes de 2^{-19}.

ESSENCE-512. Pour calculer la probabilité que $Y_3 = 0$ dans ESSENCE-512, nous suivons la même méthode que pour ESSENCE-256. Maintenant nous trouvons que la meilleure position pour la différence de A est la position 56. Pour ce A, la probabilité d'avoir $Y_3 = 0$ dans l'équation (6.9) est 2^{-42}, ce qui nous donne une complexité pour les trois tours de 2^{-44}.

La procédure du distingueur.

Elle est la même que pour celui sur 30 tours, mais maintenant, à la place de 2^7 valeurs de chaînage pour trouver la différence zéro à la sortie du premier mot, nous devons utiliser 2^{26} valeurs pour ESSENCE-256 et 2^{51} pour ESSENCE-512.

6.6.3 Collisions

La figure 6.17 représente le chemin différentiel que nous allons utiliser pour trouver des collisions sur la fonction de compression de ESSENCE. La même caractéristique est utilisé pour ESSENCE-256 et ESSENCE-512. Ce chemin a été trouvé à la main, c'est-à-dire, sans l'assistance d'une recherche automatique. Comme nous pouvons le voir sur la figure, il n'y a pas de différence dans la valeur de chaînage en entrée. Le chemin peut donc être directement utilisé pour chercher des collisions entre blocs de message à partir de la même valeur de chaînage. L'attaque par collision fonctionne de la façon suivante :

1. Trouver une paire de blocs de message qui satisfait le chemin différentiel de la partie droite, qui est indépendant de la partie gauche (mais la partie gauche n'est pas indépendante de la partie droite).
2. Une fois la paire de blocs trouvée, essayer des valeurs de chaînage jusqu'à en trouver une qui vérifie aussi le chemin différentiel de la partie gauche.

Pour la deuxième phase, nous pouvons obtenir différentes valeurs de chaînage par insertion d'un premier bloc aléatoire, et ensuite vérifier les différences après l'insertion des deuxièmes blocs, qui sont ceux qui vérifient le chemin différentiel de droite. Quand nous trouvons la collision à la sortie, le message aléatoire que nous avions inséré pour arriver à cette valeur de chaînage pseudo-aléatoire, sera le préfixe des deux messages qui donnent une collision.

Nous allons maintenant décrire les détails de l'attaque :

Le paragraphe suivant explique comment fonctionne la caractéristique. Puis je présente une méthode efficace pour trouver une paire de blocs de message qui vérifie le chemin différentiel. Les sections 6.6.4 et 6.6.5 calculent la complexité de l'attaque pour chaque version de ESSENCE ; nous n'utilisons pas les probabilités basées sur le poids de Hamming des différentielles : nous calculons les complexités d'une façon beaucoup plus précise, qui utilise les probabilités exactes des chemins différentiels binaires correspondant à chaque position dans un mot, calculées par une recherche exhaustive.

Nous allons utiliser les notations suivantes : $\vee$ pour le OU logique entre deux bits (ou mots) ; $\wedge$ le ET logique ; $\neg$ est la négation bit-à-bit ; $|w|$ est le poids de Hamming du mot w ; w_i est le i-ème bit du mot w, $0 \leq i < 32$ pour ESSENCE-256, et $0 \leq i < 64$ pour ESSENCE-512.

La caractéristique.

La caractéristique différentielle de la figure 6.17 commence avec une différence dans le bloc de message, et pas de différence dans la valeur de chaînage. Pour vérifier la caractéristique, l'hypothèse que nous faisons est que la fonction non-linéaire F va parfois absorber des différences (pendant la plus grande partie du chemin), et va parfois les conserver (dans le tour #11). C'est pour ceci que la probabilité qu'une entrée aléatoire vérifie la différentielle sera très reliée au poids de Hamming des différences A et $B = L(A)$. Nous allons voir plus en détail ce qu'il faut vérifier pour que le chemin soit satisfait.

- À l'instant #0, A revient sur la cellule la plus à droite du registre de droite par un XOR, et elle ne rentre pas dans F, contrairement à B. Pour assurer qu'aucune différence ne va apparaître à la sortie de F, nous avons besoin que tous les bits de différences de B soient absorbés, ce qui devrait arriver avec une probabilité de $2^{-|B|}$. Comme nous l'avons dit précédemment, cette estimation ne doit pas être utilisée pour calculer la complexité de l'attaque. Nous l'utilisons ici pour nous faire une idée de l'attaque et de la complexité que nous allons trouver, mais nous la calculerons plus exactement plus tard.
- À l'instant #1, nous utilisons la relation $B = L(A)$ pour que les différences introduites dans la cellule la plus à droite soient nulles. Ceci réussit toujours, mais nous avons

aussi besoin que A ne rajoute pas de différence, ce qui veut dire que F doit absorber les $|A|$ différences de A, et ceci va arriver avec une probabilité approximative de $2^{-|A|}$ pour chaque registre.
- Entre les instants #2 et #7, nous voulons aussi que les $|A|$ différences soient absorbées.
- À l'instant #8, les deux différences A s'annulent entre les deux registres et ne reviennent pas dans celui de gauche, mais A rentre à droite du registre de droite.
- À l'instant #9, contrairement à l'instant #1, A introduit une différence $L(A) = B$, qui va se propager du tour 11 au tour 17.
- À l'instant #10, pour ne pas introduire de nouvelles différences, nous voulons que la sortie de F ait comme différence $L(B)$, pour que les différences disparaissent pendant l'opération de rétroaction. Ceci sera seulement possible si $A \vee B \vee L(B) = A \vee B$. Comme nous allons le voir plus tard pour pas trouver des impossibilités dans les chemins différentiels, nous devons aussi vérifier la condition $L(B) \wedge A \wedge \neg B = 0$.
- Entre les instants #16 et #24, la caractéristique est la même que entre #0 et #8.
- De #25 à #32, la caractéristique dans le registre du message ne nous intéresse plus puisqu'elle ne sera pas utilisée.

Une fois que nous avons trouvé la caractéristique générique, nous devons chercher les A qui minimisent le coût de l'attaque ainsi qu'une méthode pour trouver une paire de blocs de message qui vérifient la partie gauche du chemin.

Trouver une paire de blocs adéquate.

Une fois que nous avons trouvé un A et un B de petit poids tels que

$$\begin{aligned} A \vee B \vee L(B) &= A \vee B \\ \text{et} L(B) \wedge A \wedge \neg B &= 0 \ , \end{aligned}$$

la complexité pour trouver un bloc qui vérifie le chemin différentiel de droite par des requêtes successives est approximativement

$$2^{15|A|+2|B|+6|A \vee B|} \ .$$

Cette complexité est plus grande que la borne donnée par l'attaque générique de $2^{\ell_h/2}$ pour toutes les différences possibles. Pour réduire ce coût, nous utilisons une technique qui ressemble à l'attaque par rebond [MRST65], c'est-à-dire, nous commençons par trouver des paires qui vérifient la partie au milieu qui a une petite probabilité, ce que nous pouvons faire en utilisant les degrés de liberté. Ensuite nous regardons si elles vérifient aussi les différentielles plus probables dans les deux directions. La section du milieu correspond aux tours #10 à #17 inclus. Plus précisément, nous allons

1. Trouver plusieurs valeurs qui vérifient la section du milieu (étapes #10 à #17) ;
2. Chercher parmi ces valeurs, une qui vérifie aussi le chemin différentiel de #0 à #10 et de #17 à #23, puisqu'avec ceci nous sommes sûrs qu'il va suivre toute la caractéristique.

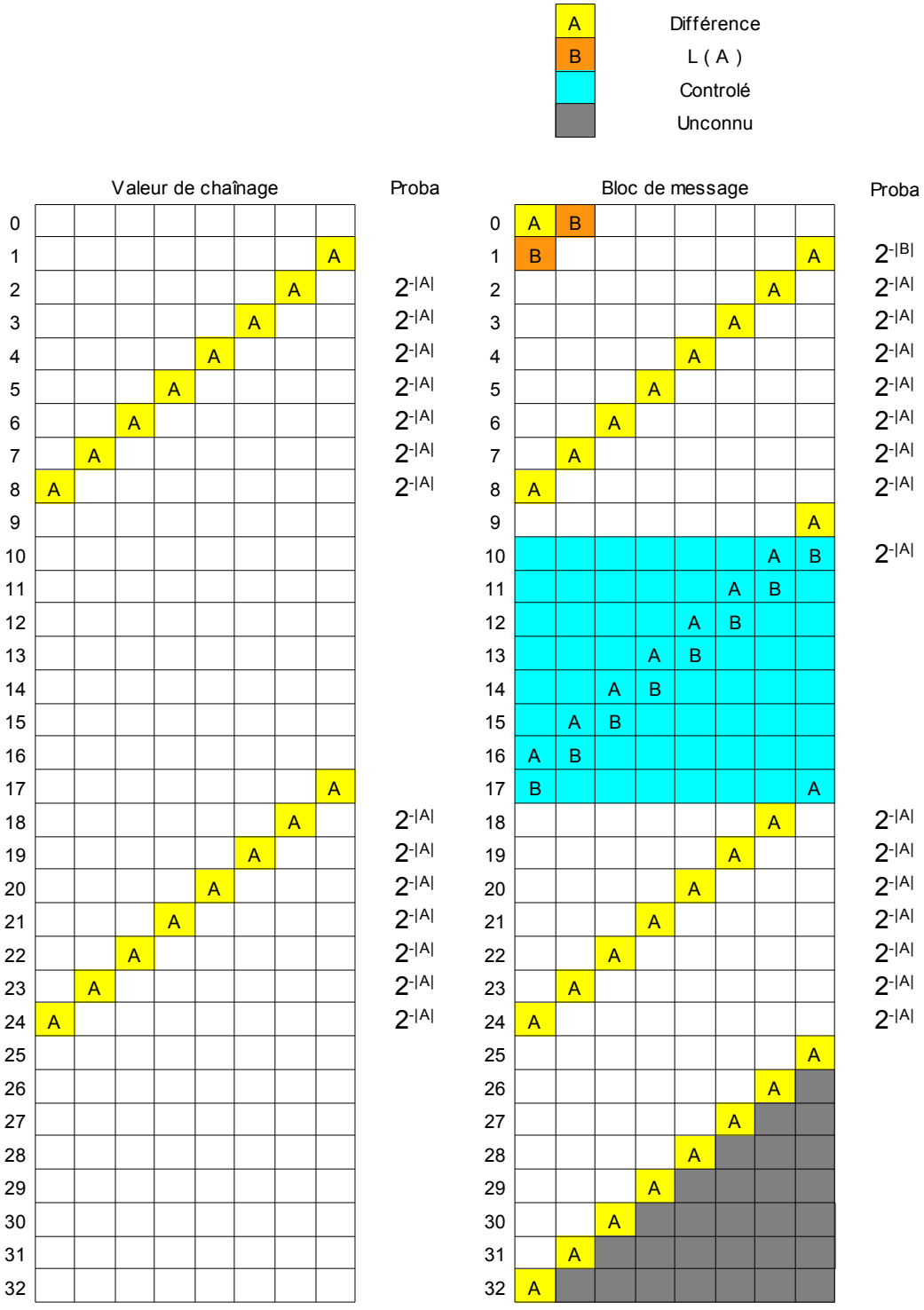

FIG. 6.17 – Le chemin différentiel utilisé sur ESSENCE

Nous avons besoin de trouver approximativement $2^{15|A|+|B|}$ messages dans la première étape, pour avoir parmi eux un message qui vérifie le reste avec une grande probabilité. Nous allons voir comment trouver efficacement de nombreuses valeurs dans la partie du milieu.

Nous allons chercher des messages qui vérifient le chemin entre #10 et #17. La table 6.5 décrit l'état pendant ces étapes. Chaque valeur correspond à un mot de 32 ou de 64 bits, selon la version utilisée.

TAB. 6.5 – Partie du message dans les tours 10-17.

10	x_0	x_1	x_2	x_3	x_4	x_5	$x_6 \oplus A$	$x_7 \oplus B$
11	x_1	x_2	x_3	x_4	x_5	$x_6 \oplus A$	$x_7 \oplus B$	x_8
12	x_2	x_3	x_4	x_5	$x_6 \oplus A$	$x_7 \oplus B$	x_8	x_9
13	x_3	x_4	x_5	$x_6 \oplus A$	$x_7 \oplus B$	x_8	x_9	x_{10}
14	x_4	x_5	$x_6 \oplus A$	$x_7 \oplus B$	x_8	x_9	x_{10}	x_{11}
15	x_5	$x_6 \oplus A$	$x_7 \oplus B$	x_8	x_9	x_{10}	x_{11}	x_{12}
16	$x_6 \oplus A$	$x_7 \oplus B$	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}
17	$x_7 \oplus B$	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	$x_{14} \oplus A$

Nous notons $\mathcal{S}$ l'ensemble de tous les indices où $A \vee B$ n'est pas zéro, c'est-à-dire,

$$\mathcal{S} = \{i, 0 \leq i < 32, A_i \vee B_i = 1\} \quad \text{pour ESSENCE-256,}$$
$$\mathcal{S} = \{i, 0 \leq i < 64, A_i \vee B_i = 1\} \quad \text{pour ESSENCE-512.}$$

Nous écrivons comme $s = |A \vee B| = |\mathcal{S}|$ le cardinal de $\mathcal{S}$. Par exemple, si $A =$ `80000000` et $B =$ `00000004`, alors $A_{31} = B_2 = 1$, donc $\mathcal{S} = \{2, 31\}$ et $s = 2$. Nous notons aussi ℓ la longueur des mots (32 ou 64, selon la version de ESSENCE).

D'abord, nous regardons un chemin binaire, pour la position i, et nous comptons le nombre de tuplets possibles $(x_{1,i}, \ldots, x_{13,i})$ qui vérifient le chemin entre les états #10 et #17. Avec $C = L(B)$, et l'exemple $(A_i, B_i, C_i) = (1, 0, 0)$, ceci correspond à tous les tuplets qui vérifient le système d'équations suivant

$$\begin{aligned}
F(x_{1,i}, x_{2,i}, x_{3,i}, x_{4,i}, x_{5,i}, x_{6,i}, x_{7,i}) &= F(x_{1,i}, x_{2,i}, x_{3,i}, x_{4,i}, x_{5,i}, x_{6,i} \oplus 1, x_{7,i}) \\
F(x_{2,i}, x_{3,i}, x_{4,i}, x_{5,i}, x_{6,i}, x_{7,i}, x_{8,i}) &= F(x_{2,i}, x_{3,i}, x_{4,i}, x_{5,i}, x_{6,i} \oplus 1, x_{7,i}, x_{8,i}) \\
F(x_{3,i}, x_{4,i}, x_{5,i}, x_{6,i}, x_{7,i}, x_{8,i}, x_{9,i}) &= F(x_{3,i}, x_{4,i}, x_{5,i}, x_{6,i} \oplus 1, x_{7,i}, x_{8,i}, x_{9,i}) \\
F(x_{4,i}, x_{5,i}, x_{6,i}, x_{7,i}, x_{8,i}, x_{9,i}, x_{10,i},) &= F(x_{4,i}, x_{5,i}, x_{6,i} \oplus 1, x_{7,i}, x_{8,i}, x_{9,i}, x_{10,i},) \\
F(x_{5,i}, x_{6,i}, x_{7,i}, x_{8,i}, x_{9,i}, x_{10,i}, x_{11,i}) &= F(x_{5,i}, x_{6,i} \oplus 1, x_{6,i}, x_{8,i}, x_{9,i}, x_{10,i}, x_{11,i}) \\
F(x_{6,i}, x_{7,i}, x_{8,i}, x_{9,i}, x_{10,i}, x_{11,i}, x_{12,i}) &= F(x_{6,i} \oplus 1, x_{7,i}, x_{8,i}, x_{9,i}, x_{10,i}, x_{11,i}, x_{12,i}) \\
F(x_{7,i}, x_{8,i}, x_{9,i}, x_{10,i}, x_{11,i}, x_{12,i}, x_{13,i}) &= F(x_{7,i}, x_{8,i}, x_{9,i}, x_{10,i}, x_{11,i}, x_{12,i}, x_{13,i})
\end{aligned}$$

Pour chaque position où $C_i = 1$, nous devons produire une différence à la sortie de F pour annuler C_i. La table 6.6 nous montre le nombre de solutions pour les x_i selon les valeurs de (A_i, B_i, C_i). Il faut préciser ici que le cas (1,0,1) mène à une impossibilité pour le chemin complet.

TAB. 6.6 – Nombre de solutions pour $(x_0, \ldots, x_{13})$ par rapport aux différences en entrée.

C_i	(A_i, B_i)		
	$(0,1)$	$(1,0)$	$(1,1)$
0	24	56	58
1	32	64	82

Pour chaque position $i \in \mathcal{S}$, nous fixons un de ces tuplets et nous calculons les bits qui manquent. Le nombre de possibilités pour fixer les tuplets pour $i \in \mathcal{S}$ est

$$N_A = 56^{|A \wedge \neg B \wedge \neg C|} \times 58^{|A \wedge B \wedge \neg C|} \times 24^{|\neg A \wedge B \wedge \neg C|} \times 82^{|A \wedge B \wedge C|} \times 32^{|\neg A \wedge B \wedge C|}$$

Pour vérifier le chemin différentiel, les équations suivantes doivent être vérifiées :

$$L(\overbrace{x_7}^{s \text{ bits fixés}}) = x_0 \oplus \overbrace{x_8 \oplus L(B) \oplus F(x_1, x_2, x_3, x_4, x_5, x_6 \oplus A, x_7 \oplus B)}^{s \text{ bits fixés}} \quad (6.10)$$

$$L(\overbrace{x_8}^{s \text{ bits fixés}}) = \overbrace{x_1 \oplus x_9 \oplus F(x_2, x_3, x_4, x_5, x_6 \oplus A, x_7 \oplus B, x_8)}^{s \text{ bits fixés}} \quad (6.11)$$

$$L(\overbrace{x_9}^{s \text{ bits fixés}}) = \overbrace{x_2 \oplus x_{10} \oplus F(x_3, x_4, x_5, x_6 \oplus A, x_7 \oplus B, x_8, x_9)}^{s \text{ bits fixés}} \quad (6.12)$$

$$L(\overbrace{x_{10}}^{s \text{ bits fixés}}) = \overbrace{x_3 \oplus x_{11} \oplus F(x_4, x_5, x_6 \oplus A, x_7 \oplus B, x_8, x_9, x_{10})}^{s \text{ bits fixés}} \quad (6.13)$$

$$L(\overbrace{x_{11}}^{s \text{ bits fixés}}) = \overbrace{x_4 \oplus x_{12} \oplus F(x_5, x_6 \oplus A, x_7 \oplus B, x_8, x_9, x_{10}, x_{11})}^{s \text{ bits fixés}} \quad (6.14)$$

$$L(\overbrace{x_{12}}^{s \text{ bits fixés}}) = \overbrace{x_5 \oplus x_{13} \oplus F(x_6 \oplus A, x_7 \oplus B, x_8, x_9, x_{10}, x_{11}, x_{12})}^{s \text{ bits fixés}} \quad (6.15)$$

$$L(\overbrace{x_{13}}^{s \text{ bits fixés}}) = x_{14} \oplus \overbrace{x_6 \oplus F(x_7 \oplus B, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13})}^{s \text{ bits fixés}} \quad (6.16)$$

Les bits fixés dans $x_1, \ldots, x_{13}$ sont ceux des tranches $i \in \mathcal{S}$. Chacune des équations (6.11)-(6.15) correspond à un système de ℓ équations binaires où $2s$ variables sont fixées et $2(\ell - s)$ sont libres. Si nous faisons un choix aléatoire des $2s$ variables fixées, chaque système aura une solution avec une probabilité approximativement de $2^{\ell-2s}$. Donc, pour un choix donné de $(x_{1,i}, \ldots, x_{13,i})$, $i \in \mathcal{S}$, nous avons une probabilité de $2^{5(\ell-2s)}$ d'obtenir une solution valide dans les équations (6.11)-(6.15).

Une solution nous donnera des valeurs qui satisfont le milieu du chemin du milieu. Pour résoudre ce système de 7 équations, nous pouvons faire comme suit :

1. Nous fixons s bits dans chaque $x_1, \ldots, x_{13}$ à une des N_a valeurs admissibles ;
2. On résout le système (6.11) à (6.15) pour les bits des s positions, en déterminant les $(\ell - s)$ valeurs qui restent de $x_8, x_9, x_{10}, x_{11}, x_{12}$. Cette étape réussit avec une

probabilité $2^{5(\ell-2s)}$, comme nous l'avons expliqué précédemment. Trouver une solution nécessite approximativement $2^{5(2s-\ell)}$ requêtes.

Une fois que nous avons trouvé une solution, toutes les valeurs qui rentrent dans L et toutes les positions de bits dans $\mathcal{S}$ sont fixées.

Une fois que nous avons trouvé une solution pour ces cinq équations, tous les bits de $x_8, x_9, x_{10}, x_{11}, x_{12}$ sont fixés, pour tous les $i \notin \mathcal{S}$ nous connaissons les valeurs de

$$
\begin{aligned}
&x_{1,i} \oplus x_{9,i} \oplus F(x_{2,i}, x_{3,i}, x_{4,i}, x_{5,i}, x_{6,i}, x_{7,i}, x_{8,i})_i \\
&x_{2,i} \oplus x_{10,i} \oplus F(x_{3,i}, x_{4,i}, x_{5,i}, x_{6,i}, x_{7,i}, x_{8,i}, x_{9,i})_i \\
&x_{3,i} \oplus x_{11,i} \oplus F(x_{4,i}, x_{5,i}, x_{6,i}, x_{7,i}, x_{8,i}, x_{9,i}, x_{10,i})_i \\
&x_{4,i} \oplus x_{12,i} \oplus F(x_{5,i}, x_{6,i}, x_{7,i}, x_{8,i}, x_{9,i}, x_{10,i}, x_{11,i})_i \\
&x_{5,i} \oplus x_{13,i} \oplus F(x_{6,i}, x_{7,i}, x_{8,i}, x_{9,i}, x_{10,i}, x_{11,i}, x_{12,i})_i
\end{aligned}
$$

3. Comme nous ne considérons que $i \notin \mathcal{S}$, les différences A et B n'apparaissent pas dans ces équations. Nous pouvons maintenant fixer les $\ell - s$ bits dans x_5, x_6, x_7 d'une façon gratuite, et nous obtenons ainsi les bits manquants de $x_{13}, x_4, x_3, x_2, x_1$. Maintenant, avec les équations (6.10) et (6.16), nous pouvons obtenir les mots x_0 et x_{14}. Alors, pour chaque solution valide de (6.10)-(6.16) nous trouvons $2^{3(\ell-s)}$ solutions gratuites. Nous obtenons au total de l'ordre de $N_A \cdot 2^{5(\ell-2s)} \cdot 2^{3(\ell-s)}$ paires possibles qui vérifient le chemin de l'instant #10 au #17.

En fait, nous obtenons au total de l'ordre de $N_\alpha \cdot 2^{5(\ell-2s)} \cdot 2^{3(\ell-s)} \cdot 2^{-1}$ couples possibles qui vérifient le chemin de #10 à #17. Le facteur 2^{-1} vient du fait que nous avons compté chaque couple possible deux fois. Tout ceci a été vérifié empiriquement, et des exemples ont été trouvés avec des complexités plus petites que celles détaillées ici, comme nous allons voir dans la conclusion.

Comment bien estimer la probabilité du chemin différentiel.

Comme nous l'avons dit avant, estimer la probabilité du chemin différentiel à partir des poids de Hamming ne nous donne pas des estimations exactes. Les probabilités d'absorption de bits d'un instant au suivant ne sont pas indépendantes. Par exemple, une seule différence est absorbée pendant 7 tours avec une probabilité $2^{-8.41}$, ce qui est significativement plus petit que la probabilité donnée par une approche basée sur les poids de Hamming qui nous donnerait 2^{-7}. D'un autre côté, pour les différentielles que nous utilisons, la dépendance entre bits de positions différentes semble être négligeable. Nous allons donc utiliser des probabilités des chemins différentiels par position, calculées empiriquement, pour calculer la probabilité du chemin complet. Ceci signifie que nous calculons la probabilité du chemin comme celle de 32 (ou 64) chemins binaires indépendants. Nous avons donc été capables de calculer la probabilité exacte de chaque position possible. Nous avons trouvé que, si dans une position nous avons la configuration suivante, $A = 1$, $B = 0$ and $L(B) = 1$, ceci mène à une impossibilité, c'est-à-dire que la différentielle ne sera jamais satisfaite pour un A qui

implique cette configuration. C'est pour ceci que nous avons besoin de vérifier la condition

$$L(B) \wedge A \wedge \neg B = 0 \ .$$

Les complexités du chemin complet utilisées pour calculer le coût de l'attaque sont dérivées directement de ces probabilités empiriques des chemins binaires, ce qui donne un résultat beaucoup plus précis que si nous avions calculé les complexités avec les poids de Hamming.

Si nous réutilisons les notations A, B, C de la section précédente, nous donnons ici les probabilités qu'une position donnée i vérifie le chemin sur les 32 tours, ce qui dépend de $(A_i, B_i, C_i) \in \{0,1\}^3$. Nous n'allons pas inclure le cas qui mène à une impossibilité.

- $(0,1,0) : 2^{-9.5}$
- $(0,1,1) : 2^{-9.0}$
- $(1,0,0) : 2^{-24.4}$
- $(1,1,0) : 2^{-23.0}$
- $(1,1,1) : 2^{-26.0}$

Donc, la probabilité qu'une entrée aléatoire vérifie le chemin est le produit de ces valeurs où chacune est élevée à une puissance égale au nombre de positions où ce cas apparaît. Pour les A que nous allons utiliser dans nos attaques, nous obtenons 2^{-236} et 2^{-461}, respectivement pour ESSENCE-256 et ESSENCE-512.

Quand nous prenons en compte la partie du milieu, nous avons aussi calculé la vraie probabilité du chemin binaire une fois que cette partie du milieu est vérifiée. Nous obtenons les probabilités suivants :

- $(0,1,0) : 2^{-1}$
- $(0,1,1) : 2^{-1}$
- $(1,0,0) : 2^{-17.2}$
- $(1,1,0) : 2^{-15.9}$
- $(1,1,1) : 2^{-19.4}$

Avec ces valeurs, nous trouvons que la probabilité que des valeurs qui vérifient la partie du milieu vérifient aussi tout le chemin est de 2^{-91} pour ESSENCE-256 et $2^{-168.2}$ pour ESSENCE-512.

Il y a deux façons de calculer le nombre total de messages qui vont vérifier le chemin complet. Comme nous obtenons le même résultat dans les deux cas, nous pouvons vérifier ainsi que nos résultats sont corrects. Soient $\rho_0, \ldots, \rho_{\ell-1}$ les probabilités pour chaque position $0, \ldots, \ell - 1$ de vérifier le chemin complet, c'est-à-dire, chaque ρ_i appartient à $\{2^{-9.5}, 2^{-9}, 2^{-24.4}, 2^{-23}, 2^{-26}\}$; et soient $\tau_0, \ldots, \tau_{\ell-1}$ les probabilités conditionnées pour chaque position différentielle, une fois que la partie du milieu a été satisfaite. Maintenant, les deux façons équivalentes de calculer le nombre de messages qui vérifient le chemin sont :

1. La probabilité du chemin complet est $\prod_{i=0}^{\ell-1} \rho_i$, donc le nombre de couples de messages qui vérifient le chemin est
$$2^{8\ell} \cdot \prod_{i=0}^{\ell-1} \rho_i \ ,$$
où 8ℓ est la longueur du haché.

2. La probabilité du chemin complet une fois que la partie du milieu a été vérifiée est $\prod_{i=0}^{\ell-1} \tau_i$. Si N est le nombre de couples qui vérifient la partie du milieu, le nombre de couples de messages qui vérifient tout le chemin est alors :

$$N \cdot \prod_{i=0}^{\ell-1} \tau_i \ .$$

Nous avons vérifié que ces deux façons de calculer le nombre de couples de messages donne le même résultat, ce qui confirme l'exactitude de nos calculs.

6.6.4 Collisions sur ESSENCE-256

Pour ESSENCE-256, nous avons pu effectuer une recherche exhaustive sur tous les A possibles, trouvant que la valeur optimale est $A = \texttt{80102040}$, pour laquelle $|A| = 4$, $|B| = 18$, et $|A \vee B| = s = 19$. Heuristiquement, nous avons besoin de l'ordre de $2^{15\times4+18} = 2^{78}$ couples de messages qui vérifient la partie du milieu pour en trouver au moins un qui vérifie le chemin complet de la partie de droite. La complexité calculée avec les probabilités les plus précises est de l'ordre de 2^{91}.

Résoudre la partie de droite.

Pour ce A, nous avons au total

$$24^6 \times 56^1 \times 32^9 \times 82^3 \approx 2^{97.7}$$

possibilités pour fixer les bits de $\mathcal{S}$. Nous avons une probabilité de $2^{5(32-2\times19)} = 2^{-30}$ de trouver une solution au système défini par la partie du milieu. Nous obtenons de l'ordre de $2^{67.7}$ solutions. Pour chaque solution, nous obtenons 2^{39} solutions additionnelles en changeant les valeurs des bits dans les positions i qui n'appartiennent pas à $\mathcal{S}$, obtenant au total jusqu'à $2^{106.7}$ solutions.

Pour chaque couple de messages trouvé, nous devons vérifier s'il satisfait aussi le reste du chemin. Comme nous l'avons montré dans la section précédente, nous avons besoin de 2^{91} valeurs qui vérifient la partie du milieu pour en trouver une parmi elles qui suit aussi le reste du chemin. Nous allons détailler le coût pour trouver ces messages.

Pour chaque solution de (6.11)-(6.15) nous obtenons 2^{39} couples de messages en plus gratuitement, donc nous devons trouver seulement $2^{91-39} = 2^{52}$ solutions pour le système. Comme chaque solution coûte à peu près 2^{30} requêtes, nous devons essayer $2^{52} \times 2^{30} = 2^{82}$ choix différents pour les s tuplets fixés. Nous obtenons donc une complexité de 2^{82} pour trouver 2^{91} couples, et une complexité de 2^{91} pour vérifier s'il y a en un qui vérifie le chemin complet.

Résoudre la partie de gauche.

Une fois que nous avons trouvé un couple de messages qui vérifie le chemin différentiel de droite, nous devons essayer 2^{64} valeurs de chaînage différentes pour trouver une collision

(pour voir la différence, heuristiquement nous obtiendrions $2^{14\times 4} = 2^{56}$). Sans augmenter la complexité totale de l'attaque, nous pouvons trouver $2^{91-64} = 2^{27}$ collisions, alors que ceci coûte 2^{142} requêtes avec l'attaque générique. Notre attaque peut être faite avec une mémoire négligeable, puisque nous n'avons pas besoin de stocker les 2^{91} messages qui vérifient la partie du milieu, car, dès que nous en avons un, nous testons s'il vérifie tout le chemin.

6.6.5 Collisions sur ESSENCE-512

Pour ESSENCE-512, la meilleure différence trouvée est $A = \texttt{8408400000480082}$, ce qui conduit à $|A| = 8$, $|B| = 35$, et $|A \vee B| = s = 39$. Comme nous l'avons dit précédemment, nous avons besoin d'approximativement $2^{168.2}$ solutions pour la partie du milieu pour trouver une solution pour le chemin complet de la partie de droite (alors que nous trouvons 2^{155} avec les estimations qui dépendent du poids de Hamming).

Résoudre la partie de droite.

Pour notre A nous avons

$$24^{14} \times 56^4 \times 58^3 \times 32^{17} \times 82^1 \times \approx 2^{196.4}$$

possibilités pour les tuplets dans les indices $i \in \mathcal{S}$ et une probabilité de 2^{-70} approximativement de trouver une solution pour la partie du milieu. Nous attendons donc de l'ordre de $2^{126.4}$ solutions. En utilisant les degrés de liberté qui restent, nous pouvons obtenir pour chaque solution $2^{3\times(64-39)} = 2^{79}$ solutions additionnelles de façon gratuite. Au total, il y a $2^{205.4}$ solutions, ce qui est assez pour en trouver une parmi elles qui vérifie le chemin complet, puisqu'il suffit d'en essayer $2^{168.2}$.

Pour calculer les $2^{168.2}$ couples, nous devons trouver $2^{168.2-79} = 2^{89.2}$ solutions du système linéaire de la partie du milieu. Chaque solution a besoin de 2^{70} requêtes, donc nous devons tester $2^{159.2}$ tuplets. Nous obtenons un coût total de moins de $2^{168.2}$ évaluations de la fonction de compression.

Résoudre la partie de gauche.

Maintenant nous avons un couple de messages qui vérifient le chemin différentiel. La probabilité pour une valeur de chaînage aléatoire de vérifier le chemin est de 2^{-128}. Donc, une fois que nous avons trouvé la paire de messages, nous pouvons trouver une collision avec une complexité de 2^{128} requêtes. Sans augmenter la complexité de $2^{168.2}$, qui est le coût pour trouver la paire de messages, nous pouvons trouver $2^{168.2-128} = 2^{40.2}$ collisions, alors que ceci devrait coûter $2^{276.6}$ requêtes.

6.6.6 Conclusion

Nous avons présenté des attaques par collision sur ESSENCE-256 et ESSENCE-512 de complexité correspondant respectivement à 2^{91} et $2^{168.2}$ appels à la fonction de compression. Plus précisément, ces valeurs sont des bornes supérieures de la complexité des

attaques. L'implémentation de nos attaques peut être faite avec une mémoire négligeable. Ces attaques sont applicables également aux versions de ESSENCE avec 224 et 384 bits de haché.

À la fin de la rédaction de cette thèse, l'attaque a été considérablement améliorée. N'ayant pas le temps de l'expliquer en détail, je dirais seulement que la complexité peut finalement être limitée à celle de la partie de gauche, c'est-à-dire 2^{64} et 2^{128} pour ESSENCE-256 et ESSENCE-512 respectivement. La validité de l'attaque a aussi été prouvée puisque nous avons trouvé une paire de messages qui satisfait le chemin différentiel pour les 28 premiers tours.

Comme réparation possible, je crois qu'il serait intéressant de regarder ce qu'il se passe si on applique une transformation linéaire $L' \in \mathbf{F}_2^\ell$ à k_7 et r_7 avant de les XORer à la sortie de F. Ceci permettrait de garantir que la fonction de tour est une permutation mais éviterait l'existence de chemins différentiels simples comme ceux que nous avons trouvés.

Conclusions.

Dans cette thèse nous avons présenté dans la première partie les cryptanalyses sur un chiffrement à flot, Achterbahn ; nous avons aussi présenté des travaux plus théoriques sur le biais des relations de parité utilisées dans ce type attaques ; et nous avons relié ces deux chapitres pour donner des critères pour améliorer et faciliter tant la conception que la cryptanalyse des chiffrements à flot par combinaison de registres. Dans la deuxième partie, nous avons d'abord introduit Shabal, l'algorithme de hachage que nous avons proposé à la compétition SHA-3 du NIST. Nous avons montré qu'il est un des plus intéressants et qu'il est sûr, en même temps qu'il reste un des plus rapides. C'est pour tout ceci que le NIST a retenu sa candidature, et Shabal est passé au deuxième tour de la compétition. Ensuite, nous avons présenté des cryptanalyses concrètes sur plusieurs candidats à cette compétition, qui est très importante pour la communauté cryptographique. Analyser la sécurité des 14 algorithmes sélectionés au deuxième tour est donc une tâche indispensable. En effet, mieux comprendre les candidats est essentiel pour être sûrs, à la fin de la compétition, d'avoir choisi un nouveau standard solide. Ces propositions ont des constructions très diverses et parfois novatrices, et de nouvelles techniques de cryptanalyse sont réquis pour bien les analyser. Dans la suite, je compte continuer à étudier ces propositions et donc continuer à travailler sur des fonctions de hachage, mais je souhaite aussi m'intéresser aux autres deux branches de la cryptographie symétrique, les chiffrements à flot et les chiffrements par blocs. En effet, il me semble que ces trois catégories de primitives symétriques sont très liées quand on laisse de côté les applications et qu'on regarde les algorithmes hors contexte. Toute avancée dans l'un de ces champs aidera très certainement de plusieurs manières à comprendre les autres.

Bibliographie

[ABM+09] J.-Ph. Aumasson, E. Brier, W. Meier, M. Naya-Plasencia et T. Peyrin. « Inside the hypercube ». Dans *Australasian Conference on Information Security and Privacy - ACISP 2009*, Lecture Notes in Computer Science. Springer, 2009. A paraître. (document), 2, 6.3

[AMM09] J.-Ph. Aumasson, A. Mashatan et W. Meier. « More on Shabal's permutation ». NIST mailing list, official comment, 2009. http://131002.net/data/papers/AMM09.pdf. 5.5.3

[And08] E. Andreeva. « On LANE modes of Operation ». COSIC Technical Report, 2008. COSIC, Université de Leuven, Belgique. 6.5

[ANP09] J.-Ph. Aumasson et M. Naya-Plasencia. « Second preimages on MCSSHA-3 ». Dans *Western European Workshop on Research in Cryptology - WEWoRC 2009*, Graz, Autriche, 2009. Disponible sur le SHA-3 Zoo - http://131002.net/data/papers/AN08.pdf. (document), 1, 6.1

[Aum08] J.Ph. Aumasson. « Collision for CubeHash2/120-512 ». NIST mailing list, 2008. http://ehash.iaik.tugraz.at/uploads/a/a9/Cubehash.txt. 6.3

[Aum09] J.Ph. Aumasson. « On the pseudorandomness of Shabal's keyed permutation ». Disponible sur http://131002.net/data/papers/Aum09.pdf, 2009. 5.5.3

[BC04] E. Biham et R. Chen. « Near-Collisions of SHA-0 ». Dans *Advances in Cryptology - CRYPTO 2004*, volume 3152 de *Lecture Notes in Computer Science*, pages 290–305. Springer, 2004. 4.2.2

[BCCM+08] E. Bresson, A. Canteaut, B. Chevallier-Mames, C. Clavier, T. Fuhr, A. Gouget, T. Icart, J.Misarsky, M. Naya-Plasencia, P. Paillier, T. Pornin, J. Reinhard, C. Thuillet et M. Videau. « Shabal, a submission to NIST's cryptographic hash algorithm competition ». Soumission à la compétition SHA-3 du NIST, 2008. http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/documents/Shabal.zip. 4.1.1

[BCCM+09a] E. Bresson, A. Canteaut, B. Chevallier-Mames, C. Clavier, T. Fuhr, A. Gouget, T. Icart, J.Misarsky, M. Naya-Plasencia, P. Paillier, T. Pornin, J. Reinhard, C. Thuillet et M. Videau. « Shabal ». Dans *The first SHA-3 candidate conference*, Leuven, Belgique, 2009. (document)

[BCCM+09b] E. Bresson, A. Canteaut, B. Chevallier-Mames, C. Clavier, T. Fuhr, A. Gouget, T. Icart, J.-F. Misarsky, M. Naya-Plasencia, P. Paillier, T. Pornin, J.-R. Reinhard, C. Thuillet et M. Videau. « Indifferentiability with Distinguishers : Why Shabal Does Not Require Ideal Ciphers ». Cryptology ePrint Archive, Report 2009/199, 2009. http://eprint.iacr.org/2009/199.pdf. 5.1.1

[BCJ+05] E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet et W. Jalby. « Collisions of SHA-0 and Reduced SHA-1 ». Dans *Advances in Cryptology - EUROCRYPT 2005*, volume 3494 de *Lecture Notes in Computer Science*, pages 36–57. Springer, 2005. 4.2.2

[BCK96] M. Bellare, R. Canetti et H. Krawczyk. « Keying Hash Functions for Message Authentication ». Dans *Advances in Cryptology - CRYPTO'96*, volume 1109 de *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996. 6.6

[BD07] E. Biham et O. Dunkelman. « A Framework for Iterative Hash Fonctions - HAIFA ». Rapport Technique CS-2007-15, 2007. Technion - Computer Science department. 4.1.1

[BDAP06] G. Bertoni, J. Daemen, G. Van Assche et M. Peeters. « RADIOGATUN, a belt-and-mill hash fonction ». Dans *Second Cryptographic Hash Workshop*, Santa Barbara, USA, août 2006. http://radiogatun.noekeon.org/. 5.2.1

[BdF98] T. Bending et D. Fon der Flass. « Crooked functions, bent functions, and distance regular graphs ». *Electron. J. Combin.*, 5(1), 1998. R34. 6.4.2, 6.7

[BDPA07] G. Bertoni, J. Daemen, M. Peeters et G. Van Assche. « Sponge fonctions ». Dans *ECRYPT hash workshop*, Barcelone, Espagne, Mai 2007. http://sponge.noekeon.org/. 5.1.2, 5.2.1

[BDPA08] G. Bertoni, J. Daemen, M. Peeters et G. Van Assche. « On the Indifferentiability of the Sponge Construction ». Dans *Advances in Cryptology - EUROCRYPT 2008*, volume 4965 de *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008. 4.2.3

[Ber07] C. Berbain. « *Analyse et conception d'algorithmes de chiffrement à flot* ». PhD thesis, Université Paris 7, 2007. 2.4.3, 2.4.5

[Ber08a] D. J. Bernstein. « CubeHash appendix : complexity of generic attacks ». Soumission à la compétition SHA-3 du NIST, 2008. http://cubehash.cr.yp.to/. 6.3, 6.3.2

[Ber08b] D. J. Bernstein. « CubeHash attack analysis (2.B.5) ». Soumission à la compétition SHA-3 du NIST, 2008. http://cubehash.cr.yp.to/. 6.3, 6.3.3

[Ber08c] D. J. Bernstein. « CubeHash expected strength (2.B.4) ». Soumission à la compétition SHA-3 du NIST, 2008. http://cubehash.cr.yp.to/. 6.3

[Ber08d] D. J. Bernstein. « CubeHash specification (2.B.1) ». Soumission à la compétition SHA-3 du NIST, 2008. http://cubehash.cr.yp.to/. 6.3, 6.3.4

[BG09] C. Blondeau et B. Gérard. « On the Data Complexity of Statistical Attacks Against Block Ciphers ». Dans *Workshop on Coding and Cryptography - WCC 2009*, Mai 2009. 2.2.4

[BGM06] C. Berbain, H. Gilbert et A. Maximov. « Cryptanalysis of Grain ». Dans *Fast Software Encryption - FSE 2006*, pages 15–29, 2006. 2.4.3

[BJV04] T. Baignères, P. Junod et S. Vaudenay. « How far can we go beyond linear cryptanalysis ? ». Dans *Advances in Cryptology - ASIACRYPT 2004*, volume 3329 de *Lecture Notes in Computer Science*, pages 432–450. Springer-Verlag, 2004. 2.2.4, 2.6

[BKMP09] E. Brier, S. Khazaei, W. Meier et T. Peyrin. « Attack for CubeHash-2/2 and collision for CubeHash-3/64 ». NIST mailing list, 2009. http://ehash.iaik.tugraz.at/uploads/3/3a/Peyrin_ch22_ch364.txt. 6.3

[Bla85] R. E. Blahut. « *Fast Algorithms for Digital Signal Processing* ». Addison Wesley, 1985. 2.4.3

[BP09] E. Brier et T. Peyrin. « Cryptanalysis of CubeHash ». Disponible sur http://thomas.peyrin.googlepages.com/BrierPeyrinCubehash.pdf, 2009. 6.3

[BR] P. S. L. M. Barreto et V. Rijmen. « The WHIRLPOOL Hashing Function ». Soumission à NESSIE, Septembre 2000. Revisée en mai 2003. Disponible sur http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html. 6.5.2

[BS91] E. Biham et A. Shamir. « Differential Cryptanalysis of DES-like cryptosystems ». *Journal of Cryptology*, 4(1) :3–72, 1991. 6.4.2

[Can06] A. Canteaut. « *Analyse et conception de chiffrements à clef secrète* ». Mémoire d'habilitation à diriger des recherches, Université Paris 6, 2006. http://www-rocq.inria.fr/codes/Anne.Canteaut/canteaut-hdr.pdf. 1

[CC03] A. Canteaut et P. Charpin. « Decomposing Bent Functions ». *IEEE Transactions on Information Theory*, 49(8) :2004–19, août 2003. 6.4.2

[CCCF00] A. Canteaut, C. Carlet, P. Charpin et C. Fontaine. « Propagation characteristics and correlation-immunity of highly nonlinear Boolean functions ». Dans *Advances in Cryptology - EUROCRYPT'2000*, volume 1807 de *Lecture Notes in Computer Science*, pages 507–522. Springer-Verlag, 2000. 3.3

[CHJ02] D. Coppersmith, S. Halevi et C. Jutla. « Cryptanalysis of stream ciphers with linear masking ». Dans *Advances in Cryptology - CRYPTO 2002*, volume 2442 de *Lecture Notes in Computer Science*. Springer-Verlag, 2002. 2.1.1

[CJ98] F. Chabaud et A. Joux. « Differential Collisions in SHA-0 ». Dans *Advances in Cryptology - CRYPTO'98*, volume 1462 de *Lecture Notes in Computer Science*, pages 56–71. Springer, 1998. 4.2.2

[CJM02] P. Chose, A. Joux et M. Mitton. « Fast correlation attacks : an algorithmic point of view ». Dans *Advances in Cryptology - EUROCRYPT 2002*, volume

2332 de *Lecture Notes in Computer Science*, pages 209–221. Springer-Verlag, 2002. 1.3, 2.1.1, 2.4.3

[CJS00] V. Chepyshov, T. Johansson et B. Smeets. « A simple algorithm for fast correlation attacks on stream ciphers ». Dans *Fast Software Encryption 2000*, volume 1978 de *Lecture Notes in Computer Science*, pages 124–135. Springer-Verlag, 2000. 1.3, 2.1.1

[CM03a] N. Courtois et W. Meier. « Algebraic attacks on stream ciphers with linear feedback ». Dans *Advances in Cryptology - EUROCRYPT 2003*, volume 2656 de *Lecture Notes in Computer Science*, pages 345–359. Springer-Verlag, 2003. 1.3, 2.1.1

[CM03b] N. Courtois et W. Meier. « Algebraic attacks on stream ciphers with linear feedback ». Dans *Advances in Cryptology - EUROCRYPT 2003*, volume 2656 de *Lecture Notes in Computer Science*, pages 345–359. Springer-Verlag, 2003. 2.1.1

[CNP09a] A. Canteaut et M. Naya-Plasencia. « Computing the bias of parity-check relations ». Dans *IEEE International Symposium on Information Theory - ISIT 2009*, Séoul, Corée, 2009. IEEE Press. (document), 3

[CNP09b] A. Canteaut et M. Naya-Plasencia. « Internal collision attack on Maraca ». Dans *Seminar 09031, Symmetric Cryptography*, Dagstuhl Seminar Proceedings, 2009. http://drops.dagstuhl.de/opus/volltexte/2009/1953/pdf/09031.NayaPlasenciaMaria.Paper.1953.pdf. 6.4

[CNP09c] A. Canteaut et M. Naya-Plasencia. « Structural weaknesses of differentially uniform mappings ». Dans *International conference on finite fields and their applications - Fq9*, Dublin, Irlande, juillet 2009. (document), 3, 6.4

[Cou03] N. Courtois. « Fast algebraic attacks on stream ciphers with linear feedback ». Dans *Advances in Cryptology - CRYPTO 2003*, volume 2729 de *Lecture Notes in Computer Science*, pages 176–194. Springer-Verlag, 2003. 1.3, 2.1.1

[CT00] A. Canteaut et M. Trabbia. « Improved Fast Correlation Attacks Using Parity-Check Equations of W.ght 4 and 5 ». Dans *Advances in Cryptology - EUROCRYPT 2000*, volume 1807 de *Lecture Notes in Computer Science*, pages 573–588. Springer-Verlag, 2000. 1.3, 2.1.1, 2.3.3, 3.15, 3.3.4

[Dai08] W. Dai. « Collisions for CubeHash1/45 and CubeHash2/89 ». Available online, 2008. http://www.cryptopp.com/sha3/cubehash.pdf. 6.3

[Dam89] I. Damgård. « A Design Principle for Hash Functions ». Dans *Advances in Cryptology - CRYPTO'89*, volume 435 de *Lecture Notes in Computer Science*, pages 416–427. Springer, 1989. 4.2.1, 6.5

[Did07] F. Didier. « *Codes de Reed-Muller et cryptanalyse du registre filtré.* ». PhD thesis, École Polytechnique, 2007. 3.3.4

[Dil09] J.F. Dillon. « APN polynomials : an update ». Dans *International Conference on Finite fields and applications - Fq9*, Dublin, Irlande, 2009. http://mathsci.ucd.ie/~gmg/Fq9Talks/Dillon.pdf. 6.4.2, 6.4

[DM89] P. Diaconis et F. Mosteller. « Methods for studying coincidences ». *Journal of the American Statistical Association*, 84(408) :853–861, 1989. 6.3.3

[ECRa] ECRYPT - European Network of Excellence in Cryptology. « The eSTREAM Stream Cipher Project ». http://www.ecrypt.eu.org/stream/. 1.5, 2.1.2

[ECRb] ECRYPT II- European Network of Excellence in Cryptology II. « The SHA-3 Zoo ». http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo. 4.3, 6

[EJ05] H. Englund et T. Johansson. « A new simple technique to attack filter generators and related ciphers ». Dans *Selected Areas in Cryptography - SAC 2004*, volume 3357 de *Lecture Notes in Computer Science*, pages 39–53. Springer-Verlag, 2005. 2.1.1, 3.3.4

[FA03] J.-C. Faugère et G. Ars. « An algebraic cryptanalysis of nonlinear filter generators using Gröbner bases ». Rapport technique RR-4739, INRIA, 2003. ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-4739.pdf. 1.3

[GG07] R. Göttfert et B. Gammel. « On the frame length of Achterbahn-128/80 ». Dans *Proceedings of the 2007 IEEE Information Theory Workshop on Information Theory for Wireless Networks*, pages 1–5. IEEE, 2007. 2.3.2, 3.1, 3.2.4

[GGK05a] B. Gammel, R. Göttfert et O. Kniffler. « The Achterbahn stream cipher ». Soumission à eSTREAM, 2005. http://www.ecrypt.eu.org/stream/. 2.1.2

[GGK05b] B. M. Gammel, R. Gottfert et O. Kniffler. « Improved Boolean Combining Functions for Achterbahn ». eSTREAM, ECRYPT Stream Cipher Project, Report 2005/072, 2005. http://www.ecrypt.eu.org/stream/papersdir/072.pdf. 2.3.1

[GGK06a] B. Gammel, R. Göttfert et O. Kniffler. « Achterbahn-128/80 ». Soumission à eSTREAM, 2006. http://www.ecrypt.eu.org/stream/. 2.4.1

[GGK06b] B. Gammel, R. Göttfert et O. Kniffler. « Status of Achterbahn and Tweaks ». Dans *Proceedings of SASC 2006 - Stream Ciphers Revisited*, 2006. http://www.ecrypt.eu.org/stream/papersdir/2006/027.pdf. 2.3.1

[GGK07] B. Gammel, R. Göttfert et O. Kniffler. « Achterbahn-128/80 : Design and analysis ». Dans *Proceedings of SASC 2007 - Stream Ciphers Revisited*, 2007. http://www.ecrypt.eu.org/stream/papersdir/2007/020.pdf. 2.4.5

[GH04] H. Gilbert et H. Handschuh. « Security Analysis of SHA-256 and Sisters ». Dans *Selected Areas in Cryptography - SAC 2003*, volume 3006 de *Lecture Notes in Computer Science*, pages 175–193. Springer, 2004.

[GKM+08] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer et S. S. Thomsen. « Grøstl – a SHA-3 candidate ». Soumission à la compétition SHA-3 du NIST, 2008. http://www.groestl.info. 6.5.2

[GM00] H. Gilbert et M. Minier. « A Collision Attack on 7 Rounds of Rijndael ». Dans *AES Candidate Conference*, pages 230–241, 2000.

[Gol96] J. Golic. « On the Security of Nonlinear Filter Generators ». Dans *Fast Software Encryption 1996*, volume 1039 de *Lecture Notes in Computer Science*, pages 173–188. Springer-Verlag, 1996. 2.1.1

[HJ06] M. Hell et T. Johansson. « Cryptanalysis of Achterbahn-Version 2 ». Dans *SAC 2006 - Selected Areas in Cryptography*, volume 4356 de *Lecture Notes in Computer Science*, pages 45–55. Springer, 2006. 2.2.4, 2.3.1, 2.3.2, 2.3.2, 2.3.2, 2.4.1, 2.4.3

[HJ07] M. Hell et T. Johansson. « Cryptanalysis of Achterbahn-128/80 ». *IET Information Security*, 1(2) :47–52, juin 2007. 2.2.4, 2.3.2, 2.4.2, 2.4.3, 2.4.4, 2.4.5, 3.2.4

[Ind08] S. Indesteege. « The LANE hash function ». Soumission à la compétition SHA-3 du NIST, 2008. http://www.cosic.esat.kuleuven.be/publications/article-1181.pdf. 6.5, 6.5.1

[IP09] S. Indesteege et B. Preneel. « Practical Preimages for Maraca ». Disponible sur http://homes.esat.kuleuven.be/~sindeste/maraca.html, 2009. 6.4

[Jen08] R. J. Jenkins Jr.. « Maraca - algorithm specification ». Soumission à la compétition SHA-3 du NIST, 2008. http://burtleburtle.net/bob/crypto/maraca/nist/Supporting_Documentation/specification.pdf. 6.4

[JJ99a] T. Johansson et F. Jönsson. « Fast correlation attacks based on turbo code techniques ». Dans *Advances in Cryptology - CRYPTO'99*, volume 1666 de *Lecture Notes in Computer Science*, pages 181–197. Springer-Verlag, 1999. 1.3, 2.1.1

[JJ99b] T. Johansson et F. Jönsson. « Improved fast correlation attack on stream ciphers via convolutional codes ». Dans *Advances in Cryptology - EUROCRYPT'99*, volume 1592 de *Lecture Notes in Computer Science*, pages 347–362. Springer-Verlag, 1999. 1.3, 2.1.1

[JJ00] T. Johansson et F. Jönsson. « Fast correlation attacks through reconstruction of linear polynomials ». Dans *Advances in Cryptology - CRYPTO'00*, volume 1880 de *Lecture Notes in Computer Science*, pages 300–315. Springer-Verlag, 2000. 1.3, 2.1.1

[JMM06] T. Johansson, W. Meier et F. Muller. « Cryptanalysis of Achterbahn ». Dans *Fast Software Encryption - FSE 2006*, volume 4047 de *Lecture Notes in Computer Science*, pages 1–14. Springer, 2006. 2.1.2, 2.2.2, 2.2.2, 2.2.4, 2.3.1, 2.3.3, 2.4.1, 2.4.3, 2.4.4

[Jou04] A. Joux. « Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions ». Dans *Advances in Cryptology - CRYPTO 2004*, volume 3152 de *Lecture Notes in Computer Science*. Springer, 2004. 6.3.3

[Jou09] A. Joux. « *Algorithmic Cryptanalysis* ». Chapman & Hall/CRC, 2009. 2.1.1, 2.4.3

[KMT09] L. R. Knudsen, K. Matusiewicz et S. S. Thomsen. « Observations on the Shabal keyed permutation ». NIST mailing list, official comment, 2009. http://www.mat.dtu.dk/people/S.Thomsen/shabal/shabal.pdf. 5.5.3

[Knu94] L. R. Knudsen. « Truncated and Higher Order Differentials ». Dans *Fast Software Encryption - FSE'94*, volume 1008 de *Lecture Notes in Computer Science*, pages 196–211. Springer, 1994. 6.5.2

[KRT07] L. R. Knudsen, C. Rechberger et S.S. Thomsen. « The Grindahl Hash Functions ». Dans *Fast Software Encryption - FSE 2007*, volume 4593 de *Lecture Notes in Computer Science*, pages 39–57. Springer, 2007. 6.5.2

[KS05] J. Kelsey et B. Schneier. « Second preimages on n-bit hash functions for much less than 2^n work ». Dans *Advances in Cryptology - EUROCRYPT 2005*, volume 3494 de *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005. 5.2.1

[Kyu07] G. Kyureghyan. « Crooked maps in $\mathbb{F}_{2^n}$ ». *Finite Fields and their applications*, 13(3) :713–726, 2007. 6.4.2

[LC09] Y. Laigle-Chapuy. « *Polynômes de permutation et applications en cryptographie. Cryptanalyse de registres combinés.* ». PhD thesis, Université Pierre et Marie Curie, 2009. 3.3.4

[Mar08a] J.W. Martin. « ESSENCE : A Candidate Hashing Algorithm for the NIST Competition ». Soumission à la compétition SHA-3 du NIST, 2008. 6.6, 6.6.1, 6.6.1

[Mar08b] J.W. Martin. « ESSENCE : A Family of Cryptographic Hashing Algorithms ». Soumission à la compétition SHA-3 du NIST, 2008. 6.6, 6.6.1, 6.6.1, 6.6.1, 6.6.1

[Mas08] M. Maslennikov. « Secure Hash Algorithm MCSSHA-3 ». Soumission à la compétition SHA-3 du NIST, 2008. http://registercsp.nets.co.kr/MCS{S}{H}{A}/MCS{S}{H}{A}-3.pdf. 6.1.1

[Mer89] R. C. Merkle. « One Way Hash Functions and DES ». Dans *Advances in Cryptology - CRYPTO'89*, volume 435 de *Lecture Notes in Computer Science*, pages 428–446. Springer, 1989. 4.2.1, 6.5

[MH04] H. Molland et T. Helleseth. « An improved correlation attack against irregular clocked and filtered generator ». Dans *Advances in Cryptology - CRYPTO 2004*, volume 3152 de *Lecture Notes in Computer Science*, pages 373–389. Springer-Verlag, 2004. 2.1.1, 3.3.4

[MNPN+09] K. Matusiewicz, M. Naya-Plasencia, I. Nikolic, Y. Sasaki et M. Schlaffer. « Rebound Attack on the full LANE Compression Function ». Dans *Advances in Cryptology - ASIACRYPT 2009*, Lecture Notes in Computer Science. Springer, 2009. A paraître. (document), 4, 6.5

[MP06] M.Naya-Plasencia. « Cryptanalysis of Achterbahn-128/80 ». Rapport eSTREAM, 2006. http://www.ecrypt.eu.org/stream/papersdir/2006/055.pdf. 2.2.4

[MRST65] F. Mendel, C. Rechberger, M. Schläffer et S. S. Thomsen. « The Rebound Attack : Cryptanalysis of Reduced Whirlpool and Grøstl ». Dans *Fast Software Encryption - FSE 2009*, volume 1008 de *Lecture Notes in Computer Science*. Springer, 5665. 6.5, 6.5.2, 6.5.2, 6.6.3

[MS88] W. Meier et O. Staffelbach. « Fast correlation attacks on stream ciphers ». Dans *Advances in Cryptology - EUROCRYPT'88*, volume 330 de *Lecture Notes in Computer Science*, pages 301–314. Springer-Verlag, 1988. 1.3, 2.1.1, 3.3.4

[MS89] W. Meier et O. Staffelbach. « Fast correlation attack on certain stream ciphers ». *Journal of Cryptology*, pages 159–176, 1989. 1.3, 2.1.1

[MSA+09] N. Mouha, G. Sekar, J.Ph. Aumasson, T. Peyrin, Søren S. Thomsen, M. S. Turan et B. Preneel. « Cryptanalysis of the ESSENCE Compression Function ». Disponible sur http ://www.nickymouha.be/papers/Essence-MouhaSekar.pdf, 2009. 6.6

[MTPT09] N. Mouha, S. S. Thomsen, B. Preneel et M. S. Turan. « Observations of non-randomness in the ESSENCE compression function ». The First SHA-3 candidate Conference - Rump session, 2009. 6.6

[MvOV97] A.J. Menezes, P.C. van Oorshot et S.A. Vanstone. *Handbook of applied cryptography*. CRC Press, 1997. Disponible sur http://www.cacr.math.uwaterloo.ca/hac/. 5.2.1

[Nat] National Institute of Standards and Technology. « Cryptographic Hash Algorithm Competition ». http://csrc.nist.gov/groups/ST/hash/sha-3/index.html. 4.3, 6.5.1

[Nat07] National Institute of Standards and Technology. « Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family ». Federal Register Notice, November 2007. Disponible sur http://csrc.nist.gov. 4.3

[NIS01] NIST. « SP 800-22, A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications », 2001. 6.3.5

[NIS02] NIST. « FIPS 198 – The Keyed-Hash Message Authentication Code (HMAC) », 2002. 6.6

[NK93] K. Nyberg et L.R. Knudsen. « Provable security against differential cryptanalysis ». Dans *Advances in Cryptology - CRYPTO'92*, volume 740 de *Lecture Notes in Computer Science*, pages 566–574. Springer-Verlag, 1993. 6.4.2

[NK95] K. Nyberg et L.R. Knudsen. « Provable Security Against a Differential Attack ». *Journal of Cryptology*, 8(1) :27–37, 1995. 6.4.2

[NP07a] M. Naya-Plasencia. « Cryptanalysis of Achterbahn-128/80 ». Dans *Fast Software Encryption - FSE 2007*, volume 4593 de *Lecture Notes in Computer Science*, pages 73–86. Springer, 2007. (document), 2.4.4, 2.4.5, 2.4.5, 3.2.4

[NP07b] M. Naya-Plasencia. « Cryptanalysis of Achterbahn-128/80 ». Dans *SASC 2007 - ECRYPT Workshop on stream ciphers*, pages 139–151, Bochum, Germany, 2007. (document)

[NP08] M. Naya-Plasencia. « Cryptanalysis of Achterbahn-128/80 with a New Keystream Limitation ». Dans *WEWoRC 2007 - Second Western European Workshop in Research in Cryptology*, volume 4945 de *Lecture Notes in Computer Science*, pages 142–152. Springer, 2008. (document), 2.4.5, 2.4.5

[NPRA+09] M. Naya-Plasencia, A. Röck, J.-Ph. Aumasson, Y. Laigle-Chapuy, G. Leurent, W. Meier et T. Peyrin. « Cryptanalysis of ESSENCE ». 2009. Disponible sur le SHA-3 Zoo http://www.131002.net/data/papers/NRALLMP09.pdf. 5, 6.6

[Nyb93] K. Nyberg. « Differentially uniform mappings for cryptography ». Dans *Advances in Cryptology - EUROCRYPT'93*, volume 765 de *Lecture Notes in Computer Science*, pages 55–64. Springer-Verlag, 1993. 6.5

[Pey07] T. Peyrin. « Cryptanalysis of Grindahl ». Dans *Advances in Cryptology - ASIACRYPT 2007*, volume 4833 de *Lecture Notes in Computer Science*, pages 551–567. Springer, 2007. 6.5.2

[RB08] M. Robshaw et O. Billet, éditeurs. *New stream cipher designs - The eSTREAM finalists*, volume 4986 de *Lecture Notes in Computer Science*. Springer, 2008. 1.5

[Rog06] P. Rogaway. « Formalizing Human Ignorance ». Dans *VIETCRYPT*, pages 211–228, 2006. 4.1

[RS04] P. Rogaway et T. Shrimpton. « Cryptographic Hash-Function Basics : Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance ». Dans *Fast Software Encryption - FSE 2004*, volume 3017, pages 371–388. Springer, 2004. 4.1

[Sie84] T. Siegenthaler. « Correlation-immunity of nonlinear combining functions for cryptographic applications ». *IEEE Transactions on Information Theory*, 30(5) :776–780, 1984. 3.2.4

[Sie85] T. Siegenthaler. « Decrypting a class of stream ciphers using ciphertext only ». *IEEE Transactions on Computers*, C-34(1) :81–84, 1985. 1.3, 2.1.1, 2.2.4, 3, 3.2.4

[SN08] P. Schmidt-Nielsen. « The Ponic Hash Function ». Soumission à la compétition SHA-3 du NIST, 2008. http://ehash.iaik.tugraz.at/uploads/3/3c/PonicSpecification.pdf. 6.2

[STKT06] K. Suzuki, D. Tonien, K. Kurosawa et K. Toyota. « Birthday Paradox for Multi-collisions ». Dans *Information Security and Cryptology - ICISC 2006*, volume 4296 de *Lecture Notes in Computer Science*. Springer, 2006. 6.3.3

[vDdF00] E.R. van Dam et D. Fon der Flass. « Codes, graphs, and schemes from nonlinear functions ». Rapport technique, Research memorandum, FEW 790, Tilburg University, The Netherlands, May 2000. 6.4.2

[vOW99] P. C. van Oorschot et M. J. Wiener. « Parallel Collision Search with Cryptanalytic Applications ». *Journal of Cryptology*, 12(1) :1–28, 1999. 6.1.3

[WFW09] S. Wu, D. Feng et W.Wu. « Cryptanalysis of the LANE hash fonction ». Dans *SAC 2009 - Selected Areas in Cryptography*, Lecture Notes in Computer Science. Springer, 2009. A paraître. 6.5

[WY05] X. Wang et H. Yu. « How to Break MD5 and Other Hash Functions ». Dans *Advances in Cryptology - EUROCRYPT 2005*, volume 3494 de *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005. 4.2.2, 4.3

[WYY05] X. Wang, Y. Lisa Yin et H. Yu. « Finding Collisions in the Full SHA-1 ». Dans *Advances in Cryptology - CRYPTO 2005*, volume 3621 de *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005. 4.2.2, 4.3

[ZZ99] Y. Zheng et X.-M. Zhang. « Plateaued functions ». Dans *Information and Communication Security, ICICS'99*, volume 1726 de *Lecture Notes in Computer Science*, pages 224–300. Springer-Verlag, 1999. 3.3, 3.17

Table des figures

1.1 Chiffrement à flot synchrone additif 4
1.2 Registre à décalage avec rétroaction F. 5

2.1 Générateur pseudo-aléatoire par combinaison de FSRs 8
2.2 Relation entre les $P_{\text{fa},i_{max}}$ et $P_{\text{nd},i_{max}}$ pour des i_{max} différents. 36

4.1 Vision générale de la construction de Merkle-Damgård 74

5.1 Le mode opératoire : tours d'insertion du message 79
5.2 Tours finaux : première représentation 79
5.3 Tours finaux : deuxième représentation 80
5.4 Structure principale de la permutation paramètrée $\mathcal{P}$ utilisée dans Shabal. . 83
5.5 Ancienne version numéro 1 du mode opératoire de Shabal 85
5.6 Mode général incluant les éponges et l'ancienne version numéro 1 du mode de Shabal 86
5.7 Procédure de l'attaque 87
5.8 Recherche d'un deuxième antécédent quand F n'est pas inversible 87
5.9 Représentation de l'attaque avec compteur 88
5.10 Ancienne version 2 du mode opératoire 89
5.11 Ancienne version 3 du mode opératoire 89
5.12 Représentation graphique générale de la permutation $\mathcal{P}$ 96
5.13 Représentation graphique de l'attaque avec $p = 1$ 99
5.14 Représentation graphique de l'attaque avec $p = 2$ 101
5.15 Représentation graphique du distingueur de $\mathcal{P}$ avec $p = 3$ sur $B[15]$. 106
5.16 Représentation graphique du distingueur sur B_{15} et $\mathcal{R}$ 108

6.1 Représentation graphique de MCSSHA-3 112
6.2 Représentation des mots controlés du registre 113
6.3 Principe de l'attaque sur MCSSHA 114
6.4 Finalisation pour MCSSHA-4 115
6.5 La fonction de tour T de CubeHash 121
6.6 Tour i dans Maraca 131
6.7 Début du tour 49 pour $\mathcal{M}_a$ (haut) et $\mathcal{M}_b$ (bas) 133
6.8 La fonction de compression de Lane. 145

6.9 Exemple d'attaque par rebond sur l'AES. 147
6.10 L'étape intérieure pour LANE-256 et LANE-512. 150
6.11 Différences en entrée choisies pour LANE-256. 152
6.12 Différences en entrée choisies pour LANE-512. 152
6.13 Le chemin différentiel pour LANE-256. 155
6.14 Le chemin différentiel tronqué pour 8 tours de LANE-512. 164
6.15 Fonction de tour de ESSENCE. 166
6.16 Chemin différentiel pour le distingueur 167
6.17 Le chemin différentiel utilisé sur ESSENCE 172

Liste des algorithmes

1 Attaque utilisant une restriction linéaire de la fonction de combinaison . . 12
2 Attaque par distingueur exploitant une approximation linéaire de la fonction de combinaison 17
3 Attaque sur Achterbahn-128 : algorithme pour trouver les états initiaux des registres R_1 et R_2 29
4 Calcul du biais exact de $PC_{f,\mathcal{T}}$ 52
5 Réprésentation schématique de Shabal. 81
6 La permutation paramètrée $\mathcal{P}$. 82

Liste des tableaux

2.1 Complexités des attaques pour retrouver la clé, publiées sur les différentes versions d'Achterbahn 39

3.1 Exemple des groupes G_i formés avec des approximations 59
3.2 Exemple de configuration des groupes avec $m = 3$. 60

5.1 Relations de dépendance entre les mots de B en sortie de $\mathcal{R}^{-1}$ et les mots de M avec $p = 2$ dans Weakinson-NoFinalUpdateA 103
5.2 Relations de dépendance entre les mots de B en sortie de $\mathcal{P}^{-1}$ et les mots de M avec $p = 2$ 103
5.3 Relations de dépendance entre les mots de B en sortie de $\mathcal{R}^{-1}$ et les mots de M avec $p = 3$ dans Weakinson-NoFinalUpdateA 103
5.4 Relations de dépendance entre les mots de B en sortie de $\mathcal{P}^{-1}$ et les mots de M avec $p = 3$ 104

6.1 Résumé des principales attaques réalisées 111
6.2 Complexité en temps de l'attaque sur les différentes versions de MCSSHA . 115
6.3 Résumé de l'attaque sur Lane-256. 153
6.4 La fonction non-linéaire F de ESSENCE 167
6.5 Partie du message dans les tours 10-17. 173
6.6 Nombre de solutions pour $(x_0, \ldots, x_{13})$ par rapport aux différences en entrée. 174

Table des matières

Remerciements iv

Overview x

Introduction générale 1

I Chiffrement à flot 1

1 Introduction au chiffrement à flot 3
- 1.1 Chiffrement à flot . . . 3
- 1.2 Générateurs pseudo-aléatoires . . . 3
- 1.3 Les principales attaques . . . 4
- 1.4 FSRs . . . 5
- 1.5 Le projet eSTREAM . . . 6

2 Cryptanalyse de Achterbahn 7
- 2.1 Le générateur utilisé . . . 7
  - 2.1.1 Modèle général et notations . . . 7
  - 2.1.2 Achterbahn version 1 . . . 8
- 2.2 Attaques de base sur les combinaisons de FSRs . . . 9
  - 2.2.1 Relations de parité . . . 9
  - 2.2.2 Attaque exploitant une restriction linéaire de la fonction de combinaison . . . 11
  - 2.2.3 Attaque exploitant l'existence de certaines structures linéaires pour la fonction de combinaison . . . 13
  - 2.2.4 Attaque exploitant une approximation linéaire de la fonction de combinaison . . . 15
- 2.3 Premières améliorations de l'attaque . . . 19
  - 2.3.1 Achterbahn version 2 . . . 19
  - 2.3.2 Amélioration en décimant par la période d'un registre . . . 20
  - 2.3.3 Amélioration de l'attaque avec l'utilisation d'approximations linéaires 23
- 2.4 Attaques exploitant des approximations linéaires . . . 24

2.4.1 Description d'Achterbahn-80/128 . . . 24
2.4.2 Attaque par distingueur sur Achterbahn-80 . . . 26
2.4.3 Amélioration de l'attaque avec un algorithme qui accélère la recherche exhaustive . . . 27
2.4.4 Retrouver la clé . . . 30
2.4.5 Attaque avec contrainte sur la longueur de suite . . . 32

3 Relations de parité **41**
3.1 Nouvelle vision des relations de parité . . . 42
3.2 Calculer le biais des relations de parité efficacement . . . 46
3.2.1 Calculer le biais des relations de parité en fixant des variables . . . 46
3.2.2 Calculer le biais des relations de parité avec la transformée de Fourier 52
3.2.3 Combiner les deux méthodes : fixer des variables et la transformée de Fourier . . . 54
3.2.4 Quand f est résiliente . . . 54
3.3 Quand f est une fonction plateau t-résiliente . . . 57
3.3.1 Comment calculer le biais efficacement (quand $n = t + 2$) . . . 58
3.3.2 Comment construire les relations de parité pour avoir un meilleur biais 59
3.3.3 Exemples . . . 62
3.3.4 Conclusion . . . 64

Conclusion **68**

II Fonctions de hachage **69**

4 La compétition SHA-3 **71**
4.1 Fonctions de hachage cryptographiques . . . 71
4.1.1 Résistance au sens strict . . . 71
4.1.2 Résistance au sens faible . . . 72
4.1.3 Résistances à d'autres attaques . . . 73
4.2 Les grandes constructions de fonctions de hachage . . . 73
4.2.1 Construction de Merkle-Damgård . . . 73
4.2.2 La famille SHA . . . 74
4.2.3 La construction éponge . . . 74
4.3 Le concours SHA-3 lancé par le NIST . . . 74

5 Conception d'une fonction de hachage : Shabal **77**
5.1 Description de Shabal . . . 78
5.1.1 Description du mode opératoire . . . 78
5.1.2 L'initialisation et la permutation interne . . . 80
5.2 Le mode opératoire et ses versions successives . . . 84

5.2.1 Ancien mode numéro 1 et attaques par recherche d'un deuxième antécédent . . . 84
5.2.2 Evolutions vers le mode actuel . . . 89
5.3 La permutation paramétrée $\mathcal{P}$ . . . 90
5.4 Proposition pour inclure un sel . . . 94
5.5 Analyse de la sécurité de Shabal et des versions réduites . . . 95
5.5.1 Attaques sur la version sans la mise à jour finale . . . 98
5.5.2 Automatiser la recherche des dépendances . . . 102
5.5.3 Distingueurs sur $\mathcal{P}$ . . . 104
5.6 Conclusion . . . 109

6 Cryptanalyse de fonctions de hachage 111
6.1 MCSSHA-3 . . . 112
6.1.1 Description de MCSSHA-3 . . . 112
6.1.2 Description de MCSSHA-4, -5, -6 . . . 113
6.1.3 Attaque en deuxième préimage . . . 113
6.1.4 Attaque par recherche d'antécédent sur MCSSHA-4 . . . 115
6.1.5 Réparer MCSSHA contre ces attaques . . . 116
6.1.6 Conclusion . . . 116
6.2 Ponic . . . 117
6.2.1 Description de Ponic . . . 117
6.2.2 Cryptanalyse de Ponic . . . 117
6.3 CubeHash . . . 119
6.3.1 Description de CubeHash . . . 119
6.3.2 Attaque générique améliorée . . . 120
6.3.3 Multicollisions narrow-pipe . . . 122
6.3.4 Comment exploiter les symétries de l'état . . . 123
6.3.5 Différentielles tronquées sur T . . . 126
6.3.6 Conclusion . . . 129
6.4 Maraca . . . 130
6.4.1 Description de Maraca . . . 130
6.4.2 Cryptanalyse de Maraca . . . 132
6.5 Lane . . . 144
6.5.1 Description de Lane . . . 144
6.5.2 Principe de la cryptanalyse de Lane . . . 146
6.5.3 Collisions semi-libres sur Lane-256 . . . 153
6.5.4 Collisions semi-libres sur Lane-512 . . . 158
6.5.5 Conclusion . . . 162
6.6 ESSENCE . . . 165
6.6.1 Description de ESSENCE . . . 165
6.6.2 Distingueur . . . 166
6.6.3 Collisions . . . 169
6.6.4 Collisions sur ESSENCE-256 . . . 177

6.6.5 Collisions sur ESSENCE-512 178
6.6.6 Conclusion 178

Conclusion **182**

Bibliographie **192**

Table des figures **194**

Liste des algorithmes **195**

Liste des tableaux **197**