Cryptanalyse de Trivium par *Cube Attack* et *Cube Tester*

Léo Barré

1er septembre 2017

Introduction

Le chiffrement symétrique est la plus ancienne de toutes les formes de cryptographie (jusqu'en 2000 av. J.-C.). Elle est généralement scindée en deux types : les chiffrements par bloc et ceux à flot. Ces derniers, auparavant appréciés pour leur simplicité et efficacité, ont commencé à perdre de leur notoriété en 2000, lorsque toutes les primitives de chiffrement par flot proposées pour le projet ECRYPT ont été éliminées dès les premiers tests, et 3 ans plus tard avec les toutes nouvelles attaques algébriques [1] qui ont fragilisé la plupart des chiffrements connus (se basant presque tous, à l'époque, sur l'utilisation de LFSRs). En 2004 le projet européen eSTREAM [2] est créé, afin d'encourager de nombreux cryptologues à tenter de concevoir de nouvelles primitives de chiffrement par flot, efficaces et résistantes. Le projet est achevé en 2008 et publie un portfolio, dernièrement mis à jour en 2012, dans lequel figurent plusieurs chiffrements par flot qui n'ont jamais encore été attaqués dans leur configuration complète : HC, Rabbit, Salsa20 et SOSEMANUK, reconnus pour leur rapidité, et Grain, MICKEY et Trivium [3], pour leur faible consommation en ressources.

Ce dernier, conçu par De Cannière et Preneel, est décrit par ses créateurs comme un excellent alliage entre simplicité et robustesse, l'algorithme n'ayant d'attaques existantes que contre des versions simplifiées de celui-ci. De toutes les attaques tentées contre celui-ci durant la phase de cryptanalyse de l'eSTREAM, les plus efficaces étaient toutes basées sur des attaques différentielles sur les IVs [4, 5, 6]. Elles ont plus tard inspiré un nouveau type d'attaque encore plus efficace : les *cube attacks* [7], pour la récupération de clé, et les *cube testers* [8], pour l'extraction de distingueurs qui ont donné lieu par la suite à de nombreuses attaques [9, 10, 11, 12, 13]. Cette famille d'attaques a permis de pousser la récupération de clé jusqu'à 799 tours de chauffe [11] et l'extraction de distingueur jusqu'à 839 [12], voire 961 pour une certaine famille de clés [10], sur les 1152 que Trivium requiert.

Durant ce stage, je me suis intéressé à tout ce contexte, de la création du projet Trivium lui-même, ses premières cryptanalyses jusqu'aux dernières attaques par cube, et je passerai en revue dans ce rapport chacun de ces points pour finir sur les contributions apportées au cours de ce stage, à savoir quelques

résultats sur des attaques algébriques classiques par SAT-solver, les derniers datant de 2007 [14], ainsi que quelques cubes distingueurs, dont ceux tirés de l'article de Knellwolf et al [10] avec des résultats améliorés, permettant de trouver un distingueur, pour un échantillon de 2^{26} clés faibles, jusqu'à 977 tours de chauffe.

Table des matières

1 Chiffrements par flot

1.1 Principe des chiffrements par flot additifs binaires

Les chiffrements par flot sont, avec les chiffrements par bloc, l'un des principaux types de chiffrements symétriques (où les deux correspondants possèdent tous deux la clé de chiffrement). Si on se réfère à la définition dans [15] : *Les chiffrements par flot chiffrent chaque caractère individuel (généralement des bits) d'un texte clair, l'un après l'autre, via une fonction chiffrante qui varie au cours du temps. En comparaison, les chiffrements à bloc vont chiffrer simultanément des groupes de caractères du message clair via une fonction chiffrante.* [traduction non contractuelle]

En général, les chiffrements par flot sont initialisés à l'aide d'une clé $k \in \mathbb{F}_2^m$, connue uniquement par les correspondants donc la même durant toute la conversation, et d'un IV (*Initial Vector*) $x \in \mathbb{F}_2^n$, choisi juste avant un chiffrement. Ces éléments permettent d'initialiser l'état interne de l'algorithme, c'est-à-dire un ensemble d'informations qui changeront au cours du temps et qui permettront de générer des bits de suite chiffrante à chaque tour.

Les chiffrements par flot additifs binaires reposent sur l'invulnérabilité cryptanalytique du *one-time pad* (ou masque jetable) : soient $M = M_1 M_2 \dots M_m$, le message à chiffrer de taille n bits, $K \in \mathbb{F}_2^n$, une clé de chiffrement, et $RNG_m : \mathbb{F}_2^n \to \mathbb{F}_2^m$, un générateur pseudo-aléatoire. Les chiffrement et déchiffrement de M fonctionnent de la même façon :

— encryption : $C = M \oplus RNG_m(K)$
— decryption : $M = C \oplus RNG_m(K)$

Ainsi, la sécurité d'un tel chiffrement par flot est reliée à sa capacité à générer de l'aléatoire.

Pour représenter l'état interne d'un chiffrement par flot, on utilise souvent le modèle des LFSR (*Linear Feedback Shift Register*), des registres de bits qui sont mis à jour à chaque tour. Les modèles les plus simples de LFSRs possède un polynôme de rétroaction $P(X) \in \mathbb{F}_2[X]$, dont les puissances de X présentes fixaient quels éléments du registre seront utilisés pour mettre celui-ci à jour. Généralement on souhaite que P soit un polynôme primitif (i.e. que le groupe $\left(\frac{\mathbb{F}_2[X]}{P(X)}\right)^*$ soit engendré par X) de degré ℓ, ℓ étant la taille du registre. Ainsi la suite chiffrante générée possède une taille maximale, sans cycle, de $2^\ell - 1$ bits. La figure 1 présente un tel exemple.

Un LFSR ne peut cependant pas être utilisé seul pour faire du chiffrement : il suffit d'obtenir suffisamment de bits de sortie et on peut retrouver la clé via l'algorithme de Berlekamp-Massey. Il existe toutefois diverses façons de les incorporer dans un chiffrement par flot :

— rendre la fonction de rétroaction non-linéaire (en ajoutant de la multiplication), ce qui en fait un NLFSR,
— mettre plusieurs LFSRs qui se mettront à jour grâce à des éléments présents dans les autres,
— mettre à jour les LFSRs irrégulièrement.

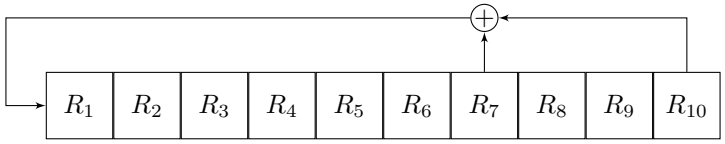

FIGURE 1 – LFSR avec $P(X) = 1 + X^7 + X^{10}$

On peut citer comme primitive utilisant de tels LFSRs le premier chiffrement utilisé dans le protocole de téléphones cellulaires GSM européen : A5/1. La primitive fonctionne avec 3 LFSRs R_1, R_2 et R_3 de polynômes de rétroaction respectifs $P_1(X) = X^{19} + X^{18} + X^{17} + X^{14} + 1$, $P_2(X) = X^{22} + X^{21} + 1$ et $P_3(X) = X^{23} + X^{22} + X^{21} + X^8 + 1$. Chaque registre est initialisé avec une clé $k \in \mathbb{F}_2^{64}$ et un IV $x \in \mathbb{F}_2^{22}$. À chaque mise à jour, un test de majorité est initié : on regarde les bits R_{1_8}, $R_{2_{10}}$ et $R_{3_{10}}$ et les registres contenant les bits en majorité sont mis à jour. Juste après l'initialisation, l'algorithme effectue 100 tours de chauffe, c'est-à-dire qu'il effectue 100 mises à jours consécutives sans générer de bits de sortie. Il existe une version de cet algorithme pour les communications hors de l'Europe : A5/2. Celui-ci possède un quatrième LFSR de polynôme de rétroaction $P_4(X) = X^{17} + X^{12} + 1$ qui est mis à jour à chaque tour et qui gère seul le test de majorité pour savoir si les autres se mettent à jour également.

Un autre chiffrement célèbre qui utilise un modèle de registre atypique est RC4 (*Rivest's Cipher 4*), autrefois le chiffrement le plus répandu, utilisé notamment pour la sécurité du Wi-Fi (protocole WEP) ou des protocoles SSL/TLS. Le principe est, cette fois, un registre S de 256 éléments de 8 bits chacun, numérotés de 1 à 256 et mélangés. À chaque tour deux indices i et j sont utilisés : d'abord les valeurs $S[i]$ et $S[j]$ sont échangées, puis les 8 bits $S[S[i] + S[j]]$ sont sortis. On est ici dans un modèle bien plus proche d'un chiffrement par bloc au niveau du schéma interne.

Il est à noter d'ailleurs que les chiffrements par bloc peuvent servir de générateur pseudo-aléatoire, et donc à créer un chiffrement par flot, selon leur mode d'opération [16], ici les modes OFB (*Output FeedBack*) et CTR (*CounTeR*) [Figure 2].

En 2003, Courtois et Meier [1] sortent un nouveau type d'attaque extrêmement efficaces contre les primitives de chiffrement par flot présentes à l'époque et utilisant largement les LFSRs comme A5/1 : les attaques algébriques. L'idée est d'exprimer chaque bit de sortie au tour i, z_i, comme un polynôme booléen en fonction de la clé $K = K_1, \ldots, K_m$ et de l'IV $X = X_1, \ldots, X_n$, c'est-à-dire :

$$z_i(K, X) \in \mathbb{B}_{K,X} = \frac{\mathbb{F}_2[K, X]}{\langle K_1^2 + K_1, \ldots, K_m^2 + K_m, X_1^2 + X_1, \ldots, X_n^2 + X_n \rangle}, \forall i \geq 0$$

ce qu'on appelle la forme ANF (*Algebraic Normal Form*). Ainsi on obtient un système d'équations qu'on peut résoudre avec des techniques classique de résolution, comme le pivot de Gauss (de complexité $\mathcal{O}(\ell^3)$ avec ℓ le nombre d'inconnues, si on suppose qu'on a autant d'équations que d'inconnues). Au début considéré uniquement pour des primitives à mise à jour régulière des LFSRs,

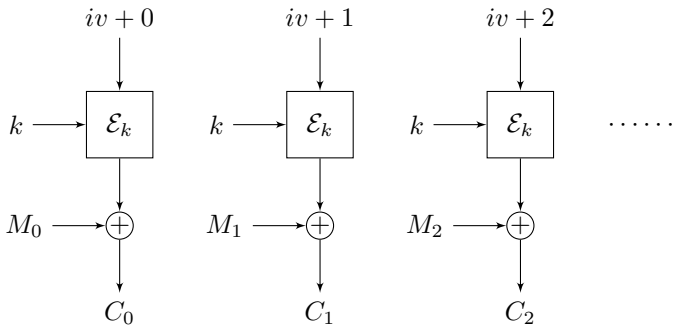

FIGURE 2 – *Block cipher*, mode *CounTeR*

comme TOYOCRYPT qui repose sur un registre de 128 bits, ce premier modèle a mené à des attaques plus dévastatrices et notamment sur des algorithmes à mise à jour irrégulière, comme par exemple une sur A5/2 [17] rapportant la difficulté de trouver la clé à celle de deviner le contenu du registre R_4, et qui pu ensuite être transposée vers A5/1 pour une attaque en temps réel, mettant fin à sa carrière.

À côté, RC4 résistera à ce genre d'attaque mais ne sera pas épargné par de nombreuses tentatives de le mettre à mal. En 2014 il est finalement retiré du protocole TLS suite à une preuve de vulnérabilité. Il est remplacé par ChaCha20, un chiffrement par flot créé par Daniel J. Bernstein et inspiré d'une autre primitive du même auteur et au nom semblable. Ces deux algorithmes font parties de la nouvelle génération de chiffrements par flot introduite en 2004 et résistant à tout type d'attaque existante encore aujourd'hui.

1.2 ECRYPT et le projet eSTREAM

En 2000 la commission européenne lance le projet NESSIE (*New European Schemes for Signatures, Integrity and Encryption*) dont le but est de favoriser l'émergence de nouvelles primitives cryptographiques dans divers domaines : hachage, signature électronique, chiffrements asymétrique et symétrique. La compétition s'est terminée en 2003 et a vu passer et récompensé des primitives aujourd'hui connues telles que AES, SHA-2 ou ECDSA. Cependant toutes les six primitives de *stream cipher* proposées ont échoué aux tests de cryptanalyse, ce qui fait que leur catégorie a été retirée de la compétition (seuls les *block ciphers* représentaient les chiffrement symétriques).

À Asiacrypt 2004, A. Shamir tint une conférence [18] dans laquelle il annonce la future disparition des *stream ciphers* au profit des *block ciphers*, ces derniers ayant des règles de sécurité mieux comprises par les cryptologues. De plus la meilleure rapidité et moindre consommation qui donnaient l'avantage aux chiffrements par flot est maintenant un facteur moins critique, les outils de calculs étant bien plus performants qu'alors. Cela se voyait notamment, selon lui, dans le remplacement de certains chiffrements par flot par des chiffrements

par bloc dans certains domaines : A5/1 faisant place à A5/3 (ou KASUMI) pour le GSM et AES préféré à RC4 pour le standard du WEP.

C'est toujours en 2004, et en réponse aux résultats de NESSIE, que le réseau d'excellence ECRYPT (*European Network of Excellence in Cryptology*) commence ses activités et lance le "*Stream Cipher Project*", ou eSTREAM [2]. Le but de ce projet est de rassembler des primitives de *stream ciphers* efficaces qui pourraient être utilisées dans un cadre officiel. En somme, il s'agît d'un projet NESSIE restreint aux chiffrements par flot.

Le projet sépare les primitives en deux types :

1. celles pour applications "software" nécessitant un haut débit,
2. celles pour applications "hardware" avec des ressources limitées (portes logiques, consommation d'énergie,...).

De plus, chaque profil possédait à la base une sous-catégorie A (1A et 2A) qui s'occupait des primitives proposant de l'authentification, mais aucune de celles-ci n'a été conservée.

Durant tout le projet, d'une durée de quatre ans, découpé en trois phases, les différents concurrents sont passés à travers divers tests de performance et études cryptanalytiques, voire même une comparaison à des primitives déjà existantes comme SNOW, un des six candidats du projet NESSIE, ou AES en mode CTR. La dernière mise à jour du protfolio officiel du projet est sortie en 2012 et y figurent les chiffrements ayant validé la troisième phase :

"*software*"	"*hardware*"
HC-128	Grain v1
Rabbit	MICKEY v2
Salsa20/12	Trivium
SOSEMANUK	

Tous les chiffrements "*software*" ont 128 bits de sécurité (i.e. $k \in \mathbb{F}_2^{128}$), et tous ceux de "*hardware*" en possèdent 80, qui sont les tailles minimum respectives pour les deux profils. De plus, HC et Salsa20 proposent également une version 256 bits officielle et MICKEY possède une version 128 bits. Il existe aussi Grain-128a, créé après les preuves de vulnérabilité de l'ancienne version 128 bits, mais elle ne fait pas partie du portfolio car jugée moins efficace que son grand frère.

Chacun de ces algorithmes a subi de nombreuses cryptanalyses et aucune n'a réussi à les casser entièrement, seulement des versions plus faibles, soit algorithmiquement, soit en terme de nombre de tours de chauffe. On va maintenant parler plus en détail du dernier représentant de la gamme "*hardware*" en terme d'ordre alphabétique.

1.3 Trivium [3]

Lorsque De Cannière et Preneel ont commencé à travailler sur le *stream cipher* qu'ils allaient proposer, ils se sont inspirés des caractéristiques de sécurité

des *block ciphers* actuels, pour le transposer à un chiffrement par flot. Trivium prend la forme d'un registre séparé en 3 NLFSRs (*Non Linear Feedback Shift Register*, c'est-à-dire comme des LFSRs mais où la fonction de mise à jour est au moins quadratique) où chaque fonction de mise à jour des sous-registres utilise également des éléments d'autres sous-registres.

Les trois sous-registres sont composés respectivement de 93, 84 et 111 bits, pour un état interne de 288 bits au total. Il nécessite d'être initialisé à l'aide d'une clé $k \in \mathbb{F}_2^{80}$ et un IV $x \in \mathbb{F}_2^{80}$:

$$\begin{aligned}(S_1, \ldots, S_{93}) &\leftarrow (k_1, \ldots, k_{80}, 0, \ldots, 0)\\ (S_{94}, \ldots, S_{177}) &\leftarrow (x_1, \ldots, x_{80}, 0, \ldots, 0)\\ (S_{178}, \ldots, S_{288}) &\leftarrow (0, \ldots, 0, 1, 1, 1)\end{aligned}$$

et, au tour i, l'état interne est mis à jour :

$$\begin{aligned}t_1 &\leftarrow S_{66} + S_{93}\\ t_2 &\leftarrow S_{162} + S_{177}\\ t_3 &\leftarrow S_{243} + S_{288}\\ \\ z_i &\leftarrow t_1 + t_2 + t_3\\ \\ t_1 &\leftarrow t_1 + S_{91}S_{92} + S_{171}\\ t_2 &\leftarrow t_2 + S_{175}S_{176} + S_{264}\\ t_3 &\leftarrow t_3 + S_{286}S_{287} + S_{69}\\ (S_1, \ldots, S_{93}) &\leftarrow (t_3, S_1, \ldots, S_{92})\\ (S_{94}, \ldots, S_{177}) &\leftarrow (t_1, S_{94}, \ldots, S_{176})\\ (S_{178}, \ldots, S_{288}) &\leftarrow (t_2, S_{178}, \ldots, S_{287})\end{aligned}$$

et le bit z_i est sorti. Une représentation graphique de l'algorithme peut être trouvée à la figure 3. Comme pour A5/1, l'algorithme effectue ensuite $4 \times 288 = 1152$ tours de chauffe pour augmenter l'aléa.

Il est à noter que l'implémentation officielle du portfolio range la clé et l'IV à l'envers dans l'état interne du *cipher* ($(k_{79}, \ldots, k_0, 0, \ldots, 0)$). Cette variation, proposée par Tim Good et Paul Crowly, rend la mise à jour du registre plus naturelle car plus proche d'un FIFO (*First In First Out*). Ce n'est cependant pas celle utilisée dans les différents articles, nous considérerons donc uniquement la première.

Comme toutes les primitives du portfolio, il n'y a aucune attaque ni cryptanalyse connue contre Trivium avec 1152 tours de chauffe. Il en existe cependant contre des versions réduites.

2 Cryptanalyse de Trivium

Dans la suite nous allons montrer plusieurs types d'attaques, toutes à clair connu : on suppose qu'on connaît l'IV x et qu'on cherche la clé k. On suppose

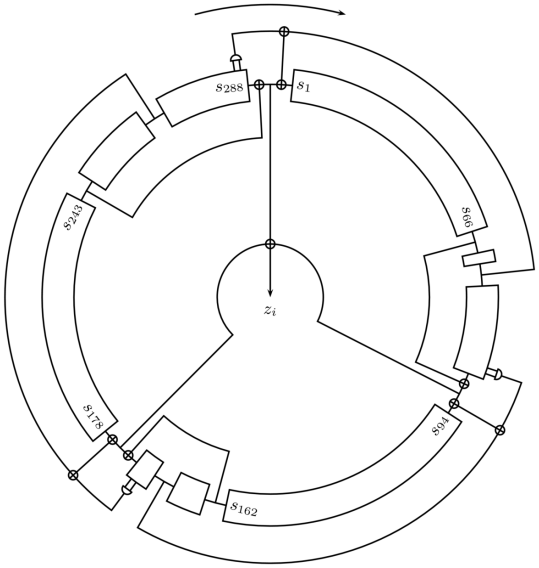

FIGURE 3 – Trivium [3, Fig. 6]

également qu'on a accès à une implémentation de Trivium pour faire nos tests. On séparera ces attaques en deux types :

— récupération : le but est de récupérer entièrement la clé, ou partiellement : les bits restant pouvant être devinés.
— distinction : le but est de trouver des preuves que l'algorithme ne se comporte pas entièrement comme un générateur d'aléatoire.

Dans les deux cas, on cherche à avoir une attaque de complexité inférieure à 2^{80}, qui est celle d'une récupération de clé par *brute force* (on teste toutes les clés jusqu'à ce que ça marche).

Il existe également un troisième type d'attaque : le *non-randomness*. Le principe est d'étendre l'attaque de distinction en ajoutant des bits de clé dans le calcul de l'attaque. Cependant nous ne nous y intéresserons pas dans la suite, ce type d'attaque étant assez impratiquable (on ne peut pas contrôler la clé en temps réel), et celles sur Trivium se déroulant toutes sur le même modèle de clé (clé nulle $k = 0, \ldots, 0$).

2.1 Premières attaques

Durant le projet eSTREAM, chaque chiffrement concurrent s'est vu subir plusieurs études cryptanalytiques, afin de s'assurer du bien-fondé de leur sécurité. Les attaques qui vont suivre dans cette sous-section ont été menées durant cette période.

H. Raddum [19] agrandit la famille de Trivium avec une version simplifiée de son algorithme : Bivium ne contient que les deux premiers registres d'état

interne, ceux qui accueillent la clé k et l'IV x :

$$\begin{aligned} t_1 &\leftarrow S_{66} + S_{93} \\ t_2 &\leftarrow S_{162} + S_{177} \\ z_i &\leftarrow t_1 + t_2 \\ t_1 &\leftarrow t_1 + S_{91}S_{92} + S_{171} \\ t_2 &\leftarrow t_2 + S_{175}S_{176} + S_{69} \\ (S_1, \ldots, S_{93}) &\leftarrow (t_2, S_1, \ldots, S_{92}) \\ (S_{94}, \ldots, S_{177}) &\leftarrow (t_1, S_{94}, \ldots, S_{176}) \end{aligned}$$

Cette version est appelée Bivium B. Son quasi-jumeau A retourne t_2 dans la suite chiffrante au lieu de $t_1 + t_2$. Ces deux versions, bien plus faibles que leur grand-frère, seront utilisées dans certaines attaques qui espèrent, en réussissant à les casser, dénicher quelques vulnérabilités exploitable chez Trivium :

- Raddum [19] utilise une méthode construisant un graphe à partir d'un système d'équations booléennes puis le résoud. Il arrive à récupérer les clés de Bivium A et B en respectivement un jour ($2^{16.4}$s) et 2^{56}s. Cette même attaque sur Trivium a une complexité de 2^{164}.
- Maximov et Biryukov [20] présentent un système où l'attaquant devine quelques bits de clé et de produit de bits afin d'obtenir un système d'équations linéaires. Il arrive à casser Bivium B en temps $c \cdot 2^{36.1}$ et Trivium en $c \cdot 2^{83.5}$, avec c la complexité de résolution de l'équation linéaire. Il est à noter que cette attaque se base sur une observation de l'algorithme Trivium : si on partitionne l'état interne en 3 suites enchevêtrées les unes dans les autres : Se_1, Se_2 et Se_3 avec $Se_i = \left\{ S_{i+3j} \;\middle|\; 1 \leq i + 3j \leq 288 \right\}$, à chaque tour, seuls des éléments de Se_2 seront utilisés pour construire le bit de sortie et ceux de mise à jour des registre.
- Charnes, McDonald et Pieprzyck [14] ont utilisé le SAT-solver MiniSat pour attaquer Bivium B, et ont pu récupérer la clé avec une complexité de 2^{56}.

Une autre façon de simplifier Trivium est de simplement considérer une variante de celui-ci avec moins de tours de chauffe. On appellera désormais ces versions, dites réduites, Trivium-N, avec N le nombre de tours de chauffe (ainsi le Trivium officiel est Trivium-1152). Les attaques suivantes se sont déroulées dessus :

- Englund, Johansson et Turan [4] ont développé une analyse statistique permettant de déceler un distingueur pour Trivium-736.
- Vielhaber [5] propose une attaque de récupération sur Trivium-576 (ou, comme il l'appelle, One.Fivium, à prononcer "one point five -yewm") permettant de récupérer 47 bits.
- Fischer, Khazaei et Meier [6] s'inspirent de l'attaque précédente et la combine avec des tests statistiques, leur permettant de récupérer Trivium-672 avec une complexité de 2^{55}.

Ces trois modèles utilisent toutes un même concept : on choisit un sous-ensemble

I de bits d'IVs puis calcule les suites chiffrantes pour tous les $2^{|I|}$ vecteurs possibles de I (les bits d'IV ne faisant pas partie de I sont fixés). Dans le cas de Vielhaber (et donc de Fischer et al), il somme les suites obtenues entre elles et appelle cela une AIDA (*Algebraic IV Differential Attack*).

Cette dernière a plus tard inspiré à Dinur et Shamir un modèle plus large et plus efficace d'attaque différentielle par IV.

2.2 Cubes de Dinur-Shamir [7]

Dans cette partie, nous travaillerons sur un polynôme booléen $p(x_1, \ldots, x_n) \in \mathbb{B}_{x_1,\ldots,x_n}$, un ensemble $I \subseteq \{1, \ldots, n\}$ et son terme associé dans $\mathbb{B}_{x_1,\ldots,x_n}$, $t_I = \prod_{i \in I} x_i$. Pour illustrer les définitions qui vont suivre, nous allons d'ailleurs prendre comme exemple le polynôme p et le sous-ensemble I suivant :

$$\begin{aligned} p(x_1, x_2, x_3, x_4, x_5) =& x_1x_2x_3 + x_1x_2x_4 + x_2x_4x_5 \\ &+ x_1x_2 + x_2 + x_3x_5 + x_5 + 1 \\ I =& \{1, 2\} \ (\Rightarrow t_I = x_1x_2). \end{aligned} \tag{1}$$

Définition 1. *Le superpoly de p par I est le polynôme $p_{S_I} \in \mathbb{B}_{x_1,\ldots,x_n}$ quotient de la division euclidienne de p par t_I, c'est-à-dire tel que :*

$$p(x_1, \ldots, x_n) = t_I \cdot p_{S_I} + q(x_1, \ldots, x_n), \text{ avec } t_I \nmid q.$$

On a ainsi dans le cas de l'exemple 1 :

$$p(x_1, \ldots, x_5) = x_1x_2(x_3 + x_4 + 1) + (x_2x_4x_5 + x_2 + x_3x_5 + x_5 + 1)$$

Définition 2. *Un maxterm t_I est tel que le superpoly associé p_{S_I} est de degré 1.*

Dans (1), on a bien $deg(x_3 + x_4 + 1) = 1$, donc $t_I = x_1x_2$ est un *maxterm* de p.

Tout sous-ensemble I de taille k permet de définir un cube de Dinur-Shamir C_I de taille k, c'est-à-dire un ensemble de 2^k polynômes $p_{|v}$ où $v \subseteq I$ et $p_{|v}$ est le polynôme p dont chaque variable subit la substitution suivante :

$$x_i \xrightarrow{v} \begin{cases} 1 & \text{si } i \in v, \\ 0 & \text{si } i \in I \backslash v, \\ x_i & \text{sinon.} \end{cases}$$

On calculera par la suite $p_I = \sum_{v \subseteq I} p_{|v}$.

Avec (1), on obtient le cube :

x_1	x_2	$p_{\|v}$
0	0	$x_3x_5 + x_5 + 1$
1	0	$x_3x_5 + x_5 + 1$
0	1	$x_4x_5 + x_3x_5 + x_5$
1	1	$x_3 + x_4 + x_4x_5 + x_3x_5 + x_5 + 1$
p_I		$x_3 + x_4 + 1$

On peut d'ailleurs remarquer dans l'exemple que le théorème suivant est respecté.

Théorème 1. *Pour tout p et I, $p_I = p_{S_I}$.*

Démonstration. Chaque terme t de p peut être décomposé de manière unique en $t = t_J t'$, avec $J \subseteq I$, $t_J = \prod_{i \in J} x_i$ et $PGCD(t_I, t') = 1$. Dans tous les $p_{|v}$, seule la partie t' pourra apparaître et ne sera présente que si $t_J \xrightarrow{v} 1$. On peut donc exprimer le résultat de p_I ainsi :

$$p_I = \sum_{\substack{t \text{ est un} \\ \text{terme de } p}} \sum_{\substack{t_J \xrightarrow{v} 1 \\ v \subseteq I}} t'.$$

Autrement dit, le coefficient de t' dans p_I est le nombre de $v \subseteq I$ tel que $J \subseteq v$, à savoir $2^{|I|-|J|}$ (on peut s'en convaincre si on considère la bijection $\mathcal{P}(I) \leftrightarrow \mathbb{F}_2^{|I|}$). Or nous sommes dans un anneau de caractéristique 2, donc la seule possibilité pour que le coefficient soit non nul, c'est d'avoir

$$\begin{aligned} & 2^{|I|-|J|} = 1 \\ \Leftrightarrow \quad & |J| = |I| \\ \Leftrightarrow \quad & J = I \text{ (puisque } J \subseteq I). \end{aligned}$$

Et comme $\sum_{t_J = t_I} t' = p_{S_I}$, on a bien $p_I = p_{S_I}$. □

Au terme de la somme, on se retrouve donc avec un polynôme p_I de degré $deg(p_I) \leq deg(p) - k$, voire $deg(p_I) = 1$ si t_I est un *maxterm*.

C'est ce modèle d'attaque qui en a inspiré de nombreuses autres qui se sont, de tous les modèles existant, le plus approché d'une attaque complète de Trivium (sans toutefois l'atteindre).

2.3 *Cube attacks* et *cube testers*

Pour la suite, tous les polynômes booléens seront des éléments de l'anneau des polynômes booléens

$$\mathbb{B}_{k,x} = \frac{\mathbb{F}_2[k_1, \ldots, k_{80}, x_1, \ldots, x_{80}]}{\langle k_1^2 + k_1, \ldots, k_{80}^2 + k_{80}, x_1^2 + x_1, \ldots, x_{80}^2 + x_{80} \rangle}$$

avec $k = k_1, \ldots, k_{80}$ représentant les bits de clé et $x = x_1, \ldots, x_{80}$ pour l'IV et $\langle a_1, \ldots, a_n \rangle$ représentant l'idéal formé par les éléments $a_1, \ldots, a_n$.

La première série dde *cube attacks* a été menée par Dinur et Shamir eux-mêmes. L'idée est de récupérer un maximum de *maxterms* afin d'obtenir un système d'équation pour déduire certains bits de clé, puis deviner les bits restants. Pour ce faire ils reconstruisent l'équation des différents cubes à l'aveugle

(ou en boîte noire) grâce à deux règles simples pour tout polynôme $p(X) = a_0 + a_1X_1 + \cdots + a_nX_n$:

$$p(0) = a_0$$
$$p(e_i) + p(0) = a_i$$

avec e_i le vecteur de poids de Hamming 1 à indice i non nul. Un test de linéarité est ensuite mené pour s'assurer de la linéarité des superpolys obtenus, ici Dinur et Shamir utilisent le BLR test (*Blum-Luby-Rubinfield*) [21] s'appuyant sur le fait que, pour p linéaire, on a $p(X + Y) = p(X) + p(Y)$ avec X et Y deux vecteurs non colinéaires. Ils ont ainsi proposé trois attaques :

- une sur Trivium-672 avec 63 *maxterms* de taille 12, pouvant aller jusqu'à Trivium-685,
- une sur Trivium-735 avec 53 *maxterms* de taille 23 en moyenne, pouvant aller jusqu'à Trivium-685,
- une sur Trivium-767 avec 35 *maxterms* de taille 29 en moyenne (et certains bits d'IV sont fixés à 1 au lieu de 0), pouvant aller jusqu'à Trivium-685.

Bernstein critique cependant la méthode de Dinur et Shamir [22], disant que ce genre d'attaque de récupération sur polynôme a déjà été introduit auparavant [23] et est limité car ne marchant que sur des polynômes booléens de bas degré, sachant que celui des bits de sortie de Trivium augmente exponentiellement.

Ainsi, Aumasson, Dinur, Meier et Shamir [8] proposent une variation des *cube attacks* : les *cube testers*. L'idée est, cette fois, de ne pas rechercher la forme ANF des superpolys, mais simplement de regarder si ces superpolys peuvent être considérés comme aléatoire, c'est-à-dire que chacun des $2^{2^{80}}$ monômes possibles a 50% de chances d'apparaître dans le polynôme. En effet si on construit des cubes sur des polynômes aléatoires, on peut s'attendre à ce que leurs superpolys le soient également. Les *cube testers* servent donc à mener des attaques de distinction. Quelques propriétés distinguantes sont proposées, mais une semble particulièrement adaptée pour Trivium : le test de neutralité. Il s'agît de regarder, étant donné le superpoly p, si on veut vérifier la neutralité du bit j, si on a :

$$p(v) + p(v + e_j) = 0$$

avec v un vecteur quelconque. On fait le test suffisamment de fois afin d'obtenir une statistique de neutralité qu'on compare à la valeur uniforme ($\frac{1}{2}$). Il est à noter d'ailleurs que, dans le cas des cubes, on peut voir le calcul de la neutralité d'un superpoly comme le calcul d'un superpoly de degré moindre, donc d'un cube de taille $\ell + 1$ sur l'ensemble $J = I \cup \{j\}$. Ainsi on parlera plutôt de neutralité du superpoly du cube en J. Également on parlera, dans le cas d'un pourcentage de neutralité à 0% ou 100%, respectivement de neutralité ou de dépendance totale, ce qui signifie que la variable associée au bit en question est, respectivement, absente du polynôme booléen ou présente et de degré 1. Également ils mettent en avant une propriété qui découle des observations de Biryukov et Maximov [20] sur le décalage de 3 : les cubes dont les bits d'IV

concernés sont espacés de 3 en 3 (ou multiple de 3), sont en moyenne plus efficaces que les autres. Avec cette méthode, les auteurs de l'article parviennent à construire un distingueur sur Trivium-772 avec un cube de taille 24 et un sur Trivium-790 avec un cube de taille 30.

Stankovsky [9], quant à lui, propose une autre méthode pour trouver un cube : on définit de manière déterministe et progressive un cube d'une certaine taille ℓ en ajoutant petit à petit des éléments à ce cube. L'élément à ajouter est choisi après comparaison de la neutralité des superpolys issus de tous les cubes possibles après ajout. Il appelle cette méthode de construction gloutonne le GreedyBitSet. L'auteur choisit de comparer le nombre maximale de neutralités totales consécutives, c'est-à-dire quel cube génère la plus longue chaîne de superpolys consécutifs à 0% de neutralité en partant du superpoly du tour 0. Il trouve donc un distingueur pour Trivium-806 avec un cube de taille 44.

Par la suite Knellwolf, Meier et Naya-Plasencia [10] proposent une attaque différentielle d'ordre supérieur classique : un modèle d'attaque où on choisit n vecteurs de différence $d_1, \ldots, d_n$ et où on calcule la dérivée de p à l'ordre $n^{\text{ème}}$:

$$\delta^{(n)}_{v_1,\ldots,v_n} p(X) = \sum_{d \in \langle d_1,\ldots,d_n \rangle} (p(X+d)).$$

Les résultats sont apparemment meilleurs lorsque toutes les différences sont de poids de Hamming 1, ce qui s'apparente à un cube. Cela leur a permis, non seulement de tirer un modèle de cubes de taille 25 dont les tests de neutralité permettent un distingueur sur Trivium-798, mais également un ensemble de 2^{26} clés faibles pour lesquelles ce même modèle permet de pousser jusqu'à Trivium-961 et un de 2^{31} clés faibles pour lesquelles on trouve un biais pour Trivium-868.

Fouque et Vannet [11] proposent une amélioration des résultats de Dinur et Shamir en terme de récupération, ainsi que de la méthode de calcul elle-même : ils travaillent en plus sur des polynômes quadratiques (en remarquant qu'il y a beaucoup de chance que les monômes de degré 2 soient sur des éléments consécutifs) et utilisent la transformée de Moebius (servant à déterminer la forme ANF d'une table de vérité) pour calculer tous les sous-cubes d'un cube plus gros C_I, c'est-à-dire tous les cubes C_J tels que $J \subset I$. Ils proposent également une amélioration de l'utilisation du BLR test, ce qui accélère les calculs. Ils obtiennent ainsi en calculant des cubes de taille 38 et 40 plusieurs sous-cubes (de tailles 30 et 37) leur permettant d'exécuter une attaque de récupération sur, respectivement, Trivium-784 et -799, qui sont pour l'instant les attaques de récupération les plus efficace sur Trivium.

Liu, Lin et Wang [12] proposent quant à eux de construire d'abord tous les cubes d'une taille moindre ℓ_{min} (grâce à la méthode de Fouque et Vannet), de les comparer entre eux pour choisir les plus "intéressants", puis de les fusionner pour obtenir des cubes efficaces. Ils obtiennent ainsi plusieurs cubes, dont un de taille 37 qui permet de distinguer jusqu'à Trivium-839, ce qui en fait le distingueur le plus efficace à ce jour. De plus, ils utilisent le test du χ^2 pour qualifier la pertinence des biais trouvés. Ce test calcule la différence entre les probabilités mesurées lors d'un expérience et les probabilités qu'on s'attend à

obtenir. Il se formule ainsi dans notre cas, c'est-à-dire où il n'y a que 2 valeurs atteignables dans chaque test : 0 ou 1 (on dit qu'on a $2 - 1 = 1$ degré de liberté) et qu'on compare les statistiques mesurées sur un échantillon restreint par rapport à la loi uniforme :

$$\chi^2 = 2\frac{\left(n - \frac{N}{2}\right)^2}{\frac{N}{2}}$$

avec N le nombre de tests menés et $\frac{n}{N}$ la probabilité mesurée lors de l'expérience.

Enfin Sarkar, Maitra et Baksi [13] améliorent les résultats de Stankovsky en utilisant l'algorithme de construction du GreedyBitSet en y changeant toutefois le comparateur, préférant les cubes avec un plus grand nombre possible de superpolys totalement neutres, ou bien un superpoly totalement neutre le plus tard possible dans l'algorithme. Ils ont également étudié l'algorithme sur des cubes déjà trouvés dans le passé, notamment un par Dinur et Shamir dans leur attaque de récupération sur Trivium-672. La qualité des cubes est, quant à elle, directement vérifiée avec un test de neutralité. Ils obtiennent ainsi, avec un cube seulement de taille 13, un distingueur sur Trivium-710. Ils en trouvent d'ailleurs un pour Trivium-829 avec un cube de taille 27 avec le fameux espacement de 3 mis en lumière plus tôt.

On l'aura remarqué, la première problématique des *cube attacks* et *cube testers* est la recherche de cubes. En effet, il n'existe pas encore de méthode déterministe pour distinguer efficacement un bon cube d'un mauvais cube, encore moins en faisant fi du nombre exponentiel de calculs à traiter pour le vérifier.

3 Contributions

Dans cette partie on parlera du travail effectué durant le stage. Chacune des implémentations et la plupart des résultats indiqués ci-dessous peuvent être retrouvés sur le git à l'adresse suivante.

`https://github.com/Shinigamileo/StageMaster17.git`

3.1 Implémentation des *cube attacks*

La première tâche incombée fût d'implémenter directement, Trivium dans un premier temps, le principe des *cube attacks* ensuite.

Un premier modèle fût d'abord conçu sur Sagemath, l'idée étant d'exploiter le modèle des anneaux booléens en PolyBori [24] afin de pouvoir générer des polynômes booléens directement, mais cette implémentation fût vite abandonnée : le degré augmentant bien trop vite sur les polynômes des bits de sortie, et donc le temps d'exécution.

Cette implémentation aura tout de même été utile pour la deuxième en C. Celle-ci devait servir à vérifier les résultats préalablement obtenus sur les *cube attacks* des articles de Dinur et Shamir [7] et Fouque et Vannet [11], ainsi qu'à

auteur(s)	ℓ	N	clés
récupération			
Dinur, Shamir	12	672 - 685	all
	23	735 - 747	all
	29	767 - 774	all
Fouque, Vannet	30 (38)	784	all
	37 (40)	799	all
distinction			
Aumasson, Dinur, Meier, Shamir	24	772	all
	30	790	all
Stankovski	44	806	all
Knellwolf, Meier, Naya-Plasencia	25	798	all
	25	868	2^{31}
	25	961	2^{26}
Liu, Lin, Wang	31	812	all
	34	824	all
	37	839	all
Sarkar, Maitra, Baksi	13	710	all
	20	792	all
	21	801	all
	22	810	all
	27	829	all
Mon stage	13	724	all
	25	808	all
	25	968	2^{27}
	27	817	all
	27	969	2^{26}
	27	977	2^{26}

FIGURE 4 – Récapitulatif des *cube attacks* et *cube testers* sur cubes de taille ℓ menés sur Trivium-N jusqu'à présent

mieux cerner le principe des cubes eux-mêmes et les méthodes développées pour les calculer.

Le programme a été intégralement codé en C et utilise l'implémentation officielle de Trivium de l'eSTREAM figurant sur leur site [1]. Il suit entièrement les conseils donnés par Dinur, Fouque et al pour la construction d'un cube en I en boîte noire pour tous les bits de sortie de chaque tour i considéré :

— obtention de la constante ($p_{S_I i}(0)$)
— obtention de tous les monômes de degré 1 ($p_{S_I i}(e_j) \oplus p_{S_I i}(0)$)
— vérification du degré (BLR)

La possibilité d'utiliser la transformée de Moebius et ainsi obtenir les mêmes résultats pour tous les sous-cubes de celui en I.

Les tests ont validé les polynômes figurant dans les articles pour les cubes donnés, et donc leurs attaques par récupération. Le code utilisé est présent dans la partie cubatak du dossier principal.

3.2 Passage par les SAT-solvers

L'une des idées de ce stage était de voir s'il pouvait y avoir un contact entre les *cube attacks* et l'utilisation de SAT-solver. Déjà nous allons rappeler rapidement ce qu'est un SAT-solver et sur quoi il se base.

Rapides rappels

Un SAT-solver est un outil servant à résoudre, le plus rapidement possible, les instances du problème SAT, c'est-à-dire les problèmes introduits ainsi :

Contexte :	$f(b_1, \ldots, b_n)$ est une formule booléenne
Question :	*existe-t-il une évaluation de chaque booléen* b_1 *à* b_n *telle que la formule* f *soit vraie ?*

une formule booléenne étant une formule avec, comme variables, des booléens (pouvant prendre comme valeur 0 :False ou 1 :True) et, comme opérations, le AND($\wedge$), le OR($\vee$) et le NOT($\neg$) (avec éventuellement le XOR($\oplus$), le IMPLIES($\rightarrow$) et d'autres).

Ce problème est NP-complet, c'est-à-dire qu'il peut exprimer tous les problèmes NP, à savoir tous les problèmes où la vérification d'une solution se fait en temps polynomial. Ainsi un outil permettant de les résoudre est très utile pour la résolution de nombreux problèmes dits NP, comme la coloration d'un graphe.

Les formules doivent être données à l'outil sous forme CNF, c'est à dire une

1. On a utilisé l'implémentation de Trivium de la version complète du portfolio.

conjonction de clauses disjonctives :

$$f(b_1, \ldots, b_n) = \bigwedge_{i=1}^{m} C_i$$

$$C_i = \bigvee_{j=1}^{s_i} l_j, \text{ avec } s_i \text{ la taille de la clause}$$

$$\forall j, \exists k \in \{1, \ldots, n\} : l_j = b_k \text{ ou } l_j = \neg b_k$$

et codées dans un fichier selon l'écriture conventionnelle DiMACS CNF [25]

Tests sur SAT

L'idée de base était de faire intéragir le problème de choix d'un cube et celui du SAT, afin de pouvoir choisir des cubes à l'aide d'un SAT-solver. Cependant, le problème majeur dans cette situation, c'est que toutes les méthodes existantes pour choisir un cube demandent forcément de les construire (même partiellement). Donc, pour l'instant, la vérification de la qualité d'un cube n'est pas polynomiale, et donc le problème n'est pas NP. Il ne peut donc pas encore y avoir de réel lien entre la construction d'un cube et la résolution du problème SAT, du moins pas tant qu'on ne trouve pas de nouvelles propriétés pour les cubes.

On a tout de même voulu voir comment avaient progressé les SAT-solvers en terme d'efficacité sur une attaque de récupération, les derniers résultats datant de 2007 [14] et ayant été menés sur Bivium seulement à l'aide de MiniSAT (qui a été détrôné depuis). Les résultats qui vont suivre, et qui n'ont rien à voir avec les cubes, proviennent de petits tests menés en parallèle d'autres (de la section suivante). Il est à noter cependant que, la compétition SAT 2017 ne se terminant qu'en septembre, les constats évoqués par la suite seront probablement obsolètes au moment où ce rapport sera lu.

On a d'abord implémenté en Python un petit ensemble de classes permettant de construire un fichier CNF, qu'on a utilisé dans un deuxième programme du même langage qui se charge de, étant donnés l'IV x et la suite chiffrante composée des bits de sortie z, construire une version booléenne de Trivium basée sur la descritpion alternative de Trivium proposé par Bernstein, avec 3 rubans d'état interne a, b et c et le ruban de sortie z :

$$(a_{-1}, \ldots, a_{-93}) \leftarrow (k_0, \ldots, k_{79}, 0, \ldots, 0)$$
$$(b_{-1}, \ldots, b_{-84}) \leftarrow (x_0, \ldots, x_{79}, 0, \ldots, 0)$$
$$(c_{-1}, \ldots, c_{-111}) \leftarrow (0, \ldots, 0, 1, 1, 1)$$
pour tout $i \geq 0$:
$$a_i = c_{i-66} \oplus c_{i-111} \oplus (c_{i-110} \cdot c_{i-109}) \oplus a_{i-69}$$
$$b_i = a_{i-66} \oplus a_{i-93} \oplus (a_{i-92} \cdot c_{i-91}) \oplus b_{i-78}$$
$$c_i = b_{i-69} \oplus b_{i-84} \oplus (b_{i-83} \cdot c_{i-82}) \oplus c_{i-87}$$
$$z_i = c_{i-66} \oplus c_{i-111} \oplus a_{i-66} \oplus a_{i-93} \oplus b_{i-69} \oplus b_{i-84}$$

Le programme reconstruit donc l'état interne de Trivium avec des booléens. Les XORs figurant dans les calculs s'expriment en CNF de la façon suivante :

$$x \oplus y \oplus z = 0 \Leftrightarrow \begin{cases} & x \vee y \vee \neg z \\ \wedge & x \vee \neg y \vee z \\ \wedge & \neg x \vee y \vee z \\ \wedge & \neg x \vee \neg y \vee \neg z \end{cases}$$

ce qui prend, pour un XOR à n termes, 2^{n-1} clauses. Heureusement, les XORs dans ce constructeur ne dépassent jamais les 4 termes (limite préconisée par Mate Soos : 5). Les ANDs servant à augmenter le degré du polynôme sont simulés en créant une nouvelle variable exprimée ainsi via la transformation de Tseytin :

$$x = x_1 x_2 \Leftrightarrow \begin{cases} & x \vee \neg x_1 \vee \neg x_2 \\ \wedge & \neg x \vee x_1 \\ \wedge & \neg x \vee x_2 \end{cases}$$

dont la création est linéaire.

On a voulu regarder l'efficacité de quelques gagnants de la compétition SAT 2016 à retrouver la clé k avec 0, 50 et 100 tours de chauffe. Nous nous sommes focalisés sur les modèles capables de gérer du pivot de Gauss :

	Trivium-0	Trivium-50	Trivium-100
CryptoMiniSAT [26]	68ms	102ms	2 749ms
CryptoMiniSAT *'x mode'* [26]	39ms	59ms	2 755ms
Lingeling [27]	65ms	120ms	1 650ms
Plingeling [27]	66ms	103ms	757ms
Treengeling [27]	66ms	126ms	1 274ms
Riss [28]	67ms	95ms	2 545ms

le 'x mode' signifiant que l'outil est capable de lire des clauses contenant des XORs (ou χlauses), ce qui permet d'éviter la première recherche de XORs pour la création de la matrice à pivot. Les résultats ci-dessus indiqueraient que Plingeling, la version parallèle de Lingeling, aurait un avantage.

On s'est ensuite davantage intéressé à Plingeling, ainsi qu'à Coprocessor, le coprocesseur de Riss, qui peut être utilisé pour simplifier les formules. Pour ce dernier, on s'est inspiré d'un script conçu par son créateur qui s'occupe de simplifier, résoudre (avec le SAT-solver précisé) puis donner le résultats du problème considéré, qu'on a un peu changé (et débuggé). On a ensuite tenté d'obtenir une attaque de récupération sur Trivium-N avec N le plus grand possible en un minimum de temps en régulant les paramètres suivants :

— le nombre d'IVs, et de rubans de sortie, à considérer pour une même clé,
— la taille du ruban à donner,
— les paramètres du coprocesseur (qui dispose d'un large pannel d'outils de simplification)

On a récupéré quelques attaques de récupération sur des versions très réduites, toutes avec les mêmes paramètres de coprocesseur, réglé pour faire du pivot de

Gauss, ainsi que d'autres technique de résolution de formule SAT, et menées sur un ordinateur 4 cœurs à 3.20 GHz avec des IVs choisis aléatoirement :

Version	Nb d'IVs	Bits de sortie	Temps
Trivium-224	6	226	3s
Trivium-256	10	176	17s
Trivium-272	15	128	40s
Trivium-280	20	120	75s
Trivium-284	22	116	1 500s
Trivium-288	20	172	4 800s

On remarque d'ailleurs que la complexité augmente drastiquement à partir de Trivium-284.

Le code utilisé pour ces tests se trouve dans le dossier satsolver. CNFmaker.py renferme les classes de construction d'une formule en DiMACS-CNF, et satrivium.py sert à construire les instances de Trivium en booléen. Le script pour utiliser Coprocessor (nécessite donc d'avoir installé Riss au préalable) est appelé donau (soit Danube en allemand, en référence au ruisseau Riß qui s'y jette).

3.3 Tests de neutralité

Avant-propos

Je me suis finalement intéressé davantage aux résultats obtenus par Knellwolf et al [10], son modèle de cubes à calculer et son modèle à 2^{26} clés faibles. J'ai donc mené plusieurs expériences de calcul de neutralité de divers modèles, plus ou moins proches de celui de Knellwolf et al, voire même de certains cubes trouvés dans d'autres articles.

Pour mener les calculs, j'ai implémenté un programme en C qui calcule la neutralité des superpolys par rapport aux modèles précisés : une clé aléatoire est générée puis transformée pour ressembler au modèle demandé, puis on calcule la neutralité des bits d'IV requis dans le superpoly issu du cube construit via les bits d'IV concernés. On répète l'expérience autant de fois que nécessaire avec une nouvelle clé aléatoire à chaque fois, et on ajoute les résultats entre eux pour calculer un vrai pourcentage de neutralité. De plus j'utilise, comme Liu, Lin et Wang [12], le test du χ^2 pour quantifier la qualité du biais obtenu par rapport à l'uniformité.

Pour la suite, nous introduirons une notation pour caractériser le comportement de chaque bit du modèle d'IV et de clé étudié pour chaque expérience. La

notation est la suivante pour chaque $i^{\text{ème}}$ bit :

$$x_i : \begin{cases} \texttt{0} & \Rightarrow x_i \text{ prend la valeur fixe 0} \\ \texttt{1} & \Rightarrow x_i \text{ prend la valeur fixe 1} \\ \texttt{c} & \Rightarrow x_i \text{ est dans l'ensemble du cube} \\ \texttt{n} & \Rightarrow \text{on regarde la neutralité de } x_i \end{cases}$$

$$k_i : \begin{cases} \texttt{0} & \Rightarrow k_i \text{ prend la valeur fixe 0} \\ \texttt{1} & \Rightarrow k_i \text{ prend la valeur fixe 1} \\ \texttt{r} & \Rightarrow k_i \text{ prend une valeur aléatoire} \end{cases}$$

Par exemple si je considère le cube avec $I = 1, 3, 10, 12, 14, 38, 45, 48, 50, 69, 75, 79$ et que je veux calculer la neutralité du bit 23 sur des clés totalement aléatoires, le modèle d'attaque sera le suivant :

$$\begin{cases} x : \texttt{c0c000000c0c0c00000000n00000000000000c000000c00c0c0000000000000000000c00000c000c0} \\ k : \texttt{rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr} \end{cases}$$

Également les différents résultats obtenus seront notés ainsi :

	j_1	j_2
i	P_1	P_2
	χ_1^2	χ_2^2

avec bits d'IV j_1 et j_2 sur lesquels sont fait le tests de neutralité, i le tour, P_1 et P_2 les pourcentages de neutralité et χ_1^2 et χ_2^2 les χ^2 obtenus sur ces pourcentages respectifs.

Le code utilisé pour les tests menés est disponible dans le répertoire neutrality. Le programme issu de neutrality.c sert à faire les tests et afficher les résultats et le script neutralisee.py sert à lire les informations contenues dans les fichiers de résultats correctement. Ces fichiers, d'ailleurs, se trouvent dans le répertoire results dans le même dossier et leurs noms suivent ce modèle :

⟨modèle d'IV⟩_⟨modèle de clé⟩_⟨nombre de tours max⟩`.kw`

Ce nom est ensuite "hashé" grâce au script neutralihash.py afin d'être plus court. Le vrai nom peut néanmoins être récupéré en réutilisant le script sur le nom "hashé" (le script ls présent dans results affiche également les vrais noms des fichiers de résultats). Le script neutralineutrali présent avec eux sert à pouvoir exécuter le programme du dossier précédent sur plusieurs cœurs à la fois.

Résultats

Tous les résultats qui vont suivre sont issus de tests menés sur des ordinateurs avec des processeurs de fréquence entre 2.00 et 3.20 GHz. Chaque fois une expérience sur 1 000 clés était d'abord menée, avec des résultats transitoires toutes les 100 clés puis une plus grande sur 20 000 clés en cas de biais petit mais intéressant (voire plus si les cubes étaient suffisamment petits pour que

les tests soient rapides et/ou pour vérifier plus intensément un biais). Enfin nous considérerons un risque de non-uniformité à 0.1%, c'est-à-dire que seront affichés uniquement les pourcentages dont le χ^2 est supérieur à 10.828, à savoir la valeur critique de la distribution du χ^2 associée avec 1 degré de liberté (car chaque résultat n'a que 2 valeurs possibles).

Les premiers cas étudiés ont été les paires proposées par l'article de Knellwolf et al, avec et sans clés faibles, à savoir, respectivement :

$$\begin{cases} x : \texttt{c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00n00n00n0} \\ k : \texttt{r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r0} \end{cases}$$

et

$$\begin{cases} x : \texttt{c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00n00n00n0} \\ k : \texttt{rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr} \end{cases}$$

Pour les clés faibles de Knellwolf, des résultats similaires à ceux de l'article concerné ont été trouvé, à savoir :

	72	75	78
953	0.00 20 000	0.00 20 000	0.00 20 000
961	100.00 20 000	50.37 1.066	0.00 20 000

Cependant, pour la partie à toute clé, une légère avancée à été trouvée : en effet l'article de Knellwolf et al s'arrêtait à 798 tours pour la découverte de biais mais peut être poussée plus loin :

	72	75	78
798	44.83 213.83	41.71 549.13	34.25 1 985.8
805	47.87 36.30	49.36 3,328	47.35 56.39
808	49.56 1.549	48.74 12.70	48.80 11.52

On peut expliquer que la première équipe n'ait pas remarqué ces résultats par le fait qu'ils se focalisaient sur la recherche de biais importants en terme de pourcentage : en effet, même si le χ^2 est satisfaisant, les pourcentages obtenus au tour 808 sont très proche de l'uniformité. Également les résultats de l'autre équipe reposent sur 100 clés tirées au hasard, là où ceux affichés ici sont tirés de tests sur 20000 clés. Nous avons également voulu considérer un nouveau modèle de clés faibles, dites de Knellwolf "régularisées" :

$$\begin{cases} x : \texttt{c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00n0} \\ k : \texttt{r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r0} \end{cases}$$

c'est-à-dire que les bits 67 et 68, fixés respectivement à 0 et 1 sur les clés faibles de Knellwolf, prennent cette fois une valeur aléatoire pour le premier, et 0 pour le second, ce qui nous donne des clés avec un bit aléatoire tous les indice multiples de 3 (d'où "régularisées"). On a obtenu sur 20000 tests :

	72	75	78
952	24.69 5 124.8	24.92 5 034.1	74.99 4 994.0
968	24.44 5 228.6	24.86 5 056.2	24.98 5 010.0

Le biais n'est pas aussi gros que celui trouvé par Knellwolf et al, mais permet de gagner 7 tours pour un jeu de 2^{27} clés faibles, entièrement différent de celui de Knellwolf.

En s'insipirant de la caractéristique de Trivium observée par Biryukov et Maximov [20] et utilisée par Aumasson et al [8], on a également considéré un prolongement du modèle de cube découvert par Knellwolf et al en construisant le cube sur tous les bits d'IV 3 à 3 en commençant par le bit 1. On obtient ainsi les deux modèles :

$$\begin{cases} x : \texttt{c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00n0} \\ k : \texttt{rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr} \end{cases}$$

puis, pour les clés faibles de Knellwolf,

$$\begin{cases} x : \texttt{c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00n0} \\ k : \texttt{r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00010r00r00r00r0} \end{cases}$$

et comme indice de neutralité :

toute clé	
798	20.11 7 860.1
808	39.13 1 040.3
818	48.76 13.45

clés faibles de Knellwolf	
961	0.00 20 000
962	0.00 20 000
977	0.00 20 000

sur, respectivement, 22 000 et 20 000 tests. On gagne donc 10 tours pour toute clé, et 16 pour les clés faibles.

On a ensuite voulu considérer le modèle de cube de taille 27 à espacement 3 à 3 de l'article de Sarkar et al [13], c'est-à-dire qu'ici le premier bit est le $2^{\text{ème}}$ bit d'IV :

$$\begin{cases} x : \texttt{c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00n0} \\ k : \texttt{rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr} \end{cases}$$

On vérifie aussi pour les clés de Knellwolf

$$\begin{cases} x : \texttt{c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00n0} \\ k : \texttt{r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00010r00r00r00r0} \end{cases}$$

et un nouveau modèle de clé faible, dite de Knellwolf régularisée inversée

$$\begin{cases} x : \texttt{c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00n0} \\ k : \texttt{0rr0rr0rr0rr0rr0rr0rr0rr0rr0rr0rr0rr0rr0rr0rr0rr0rr0rr0rr0rr0rr0rr0rr0rr0rr0rr0rr0rr0r} \end{cases}$$

où les bits aléatoires des clés faibles régularisées sont fixées à 0 et inversement. Pour toute clé, nos calculs n'ont réussi qu'à trouver un biais sur Trivium-818,

après 20 000 tests, avec un indice de neutralité de 48.72%, là où Sarkar et al en trouve un sur Trivium-823 (voire -829) avec un indice de 49.7% qu'ils détectent avec une complexité de calcul de $2^{42.74}$. Avec clés faibles de Knellwolf, une maigre progression jusqu'à Trivium-846 mais un biais très faible, incomparable à ceux déjà mesurés avec le modèle de cube de Knellwolf. Pour les clés inversées, on obtient les résultats suivants :

clés inversées	
818	40.33 766.50
823	41.15 642.07
832	47.59 47.81

Soit un léger progrès pour un ensemble de 2^{64} clés faibles. Enfin, vu que ce modèle de cube était semblable à celui de Knellwolf et al avec un décalage de 1 bit, on a également considéré des clés faibles de Knellwolf régularisées avec ce même décalage :

$$\begin{cases} x : \texttt{c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00c00n0} \\ k : \texttt{0r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r00r} \end{cases}$$

ce qui nous a donné les résultats suivants :

clés régularisées décalées	
953	0.00 1 000.0
954	0.00 1 000.0
969	0.00 1 000.0

donannt un autre modèle de clés faibles dont le biais est comparable à ceux déjà évoqués.

Les dernières clés qui ont été testées étaient des revérifications des cubes proposés par Sarkar et al (cubes de taille 13 et 20 par exemple) ainsi que des confirmations de la règle de 3 de Trivium (en plaçant quelques `c` en respectant la règle de 3, puis en plaçant des `n` aléatoirement pour vérifier lesquels donnent les meilleurs résultats). Les résultats obtenus confirment les deux idées suscitées. Durant le deuxième type de tests d'ailleurs le modèle suivant a été mis en avant :

$$\begin{cases} x : \texttt{0c00n00c00c00c00c00c00c00c0000000000000000000c00c00c00000000000c000000000000000} \\ k : \texttt{rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr} \end{cases}$$

donnant les résultats suivants après 100 000 tests :

710	33.30 11 157
722	44.51 1 206.0
725	49.15 28.76

Ce résultat, trouvé au hasard de tests, est cité uniquement car il dépasse en terme de qualité de distingueur celui de même taille trouvé par Sarkar et al et inspiré du papier originel [7] (distingueur jusqu'à Trivium-710).

Le problème majeur de ces tests est qu'on tombe régulièrement sur des faux positifs, la faute au fait qu'on doive se référer à de petits échantillons pour faire nos statistiques : en moyenne on fait dans les 20 000 ($\approx 2^{14}$) tests, une toute petite partie des 2^{80} clés existantes, il est donc très possible qu'on tombe sur un ensemble de clés avec quelques faiblesses ponctuelles (par exemple : il a fallu plus de 500 000 tests pour faire disparaître un biais sur le $906^{\text{ème}}$ bit sur les tests d'un des cubes de Sarkar et al).

Conclusion

Durant ce stage j'ai passé beaucoup de temps à assimiler les concepts et les idées qui s'étaient développés autour du projet eSTREAM, de Trivium et des attaques par cube, à implémenter certains concepts pour mieux me les approprier.

Le problème principal des *cube attacks* et *testers* est la recherche de cubes : encore aujourd'hui il n'existe aucune science exacte permettant de déterminer à coup sûr si un cube sera efficace ou pas. Encore moins sans passer par les 2^{ℓ} calculs nécessaires à sa construction, ce qui rend compliqué le passage par des outils de résolution comme le SAT-solver. Une bonne partie de mon stage m'a fait tester plusieurs modèles de cubes basés sur des observations sur le schéma interne de Trivium et, à part une amélioration de l'attaque sur clés faibles de Knellwolf et al, aucun de mes résultats ne fait progresser le nombre de tours "conquis" de Trivium.

Néanmoins les *cube attacks* sont les attaques les plus efficaces jamais menées sur Trivium (récupération de Trivium-799 et distingueur sur Trivium-839), on peut donc espérer que celles-ci s'amélioreront davantage au fil du temps.

Également, la partie interaction avec SAT-solver reste très intéressante, mais inutilisable pour l'instant. Une idée serait d'essayer de trouver des caractéristiques d'un "bon" cube qui seraient visible en temps polynomial, tâche que je me suis incombé pendant une partie de mon stage, mais en vain. Si un jour on parvient à déceler une qualité de cube reconnaissable en temps polynomial, les SAT-solvers seraient un atout indéniable dans la recherche de cubes, et donc dans les analyses de la sécurité de Trivium.

Références

[1] Nicolas Courtois and Willi Meier. Algebraic attacks on stream ciphers with linear feedback. In Eli Biham, editor, *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, volume 2656 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2003.

[2] ECRYPT. Stream cipher project estream, 2004-2008. `http://www.ecrypt.eu.org/stream/`.

[3] Christophe De Cannière and Bart Preneel. Trivium. In Matthew J. B. Robshaw and Olivier Billet, editors, *New Stream Cipher Designs - The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 244–266. Springer, 2008.

[4] Håkan Englund, Thomas Johansson, and Meltem Sönmez Turan. A framework for chosen IV statistical analysis of stream ciphers. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *Progress in Cryptology - INDOCRYPT 2007, 8th International Conference on Cryptology in India, Chennai, India, December 9-13, 2007, Proceedings*, volume 4859 of *Lecture Notes in Computer Science*, pages 268–281. Springer, 2007.

[5] Michael Vielhaber. Breaking ONE.FIVIUM by AIDA an algebraic IV differential attack. *IACR Cryptology ePrint Archive*, 2007 :413, 2007.

[6] Simon Fischer, Shahram Khazaei, and Willi Meier. Chosen IV statistical analysis for key recovery attacks on stream ciphers. In Serge Vaudenay, editor, *Progress in Cryptology - AFRICACRYPT 2008, First International Conference on Cryptology in Africa, Casablanca, Morocco, June 11-14, 2008. Proceedings*, volume 5023 of *Lecture Notes in Computer Science*, pages 236–245. Springer, 2008.

[7] Itai Dinur and Adi Shamir. Cube attacks on tweakable black box polynomials. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, volume 5479 of *Lecture Notes in Computer Science*, pages 278–299. Springer, 2009.

[8] Jean-Philippe Aumasson, Itai Dinur, Willi Meier, and Adi Shamir. Cube testers and key recovery attacks on reduced-round MD6 and trivium. In Orr Dunkelman, editor, *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, volume 5665 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2009.

[9] Paul Stankovski. Greedy distinguishers and nonrandomness detectors. In Guang Gong and Kishan Chand Gupta, editors, *Progress in Cryptology - INDOCRYPT 2010 - 11th International Conference on Cryptology in*

India, Hyderabad, India, December 12-15, 2010. Proceedings, volume 6498 of *Lecture Notes in Computer Science*, pages 210–226. Springer, 2010.

[10] Simon Knellwolf, Willi Meier, and María Naya-Plasencia. Conditional differential cryptanalysis of trivium and KATAN. In Ali Miri and Serge Vaudenay, editors, *Selected Areas in Cryptography - 18th International Workshop, SAC 2011, Toronto, ON, Canada, August 11-12, 2011, Revised Selected Papers*, volume 7118 of *Lecture Notes in Computer Science*, pages 200–212. Springer, 2011.

[11] Pierre-Alain Fouque and Thomas Vannet. Improving key recovery to 784 and 799 rounds of trivium using optimized cube attacks. In Shiho Moriai, editor, *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*, volume 8424 of *Lecture Notes in Computer Science*, pages 502–517. Springer, 2013.

[12] Meicheng Liu, Dongdai Lin, and Wenhao Wang. Searching cubes for testing boolean functions and its application to trivium. In *IEEE International Symposium on Information Theory, ISIT 2015, Hong Kong, China, June 14-19, 2015*, pages 496–500. IEEE, 2015.

[13] Santanu Sarkar, Subhamoy Maitra, and Anubhab Baksi. Observing biases in the state : case studies with trivium and trivia-sc. *Des. Codes Cryptography*, 82(1-2) :351–375, 2017.

[14] Cameron McDonald, Chris Charnes, and Josef Pieprzyk. An algebraic analysis of trivium ciphers based on the boolean satisfiability problem. *IACR Cryptology ePrint Archive*, 2007 :129, 2007.

[15] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[16] Morris J. Dworkin. Sp 800-38c. recommendation for block cipher modes of operation : The ccm mode for authentication and confidentiality. Technical report, Gaithersburg, MD, United States, 2004.

[17] Elad Barkan, Eli Biham, and Nathan Keller. Instant ciphertext-only cryptanalysis of GSM encrypted communication. *J. Cryptology*, 21(3) :392–429, 2008.

[18] Adi Shamir. Stream ciphers : Dead or alive ? conference ASIACRYPT, 2004.

[19] Håvard Raddum. Cryptanalytic results on trivium, 2007. eSTREAM, ECRYPT Stream Cipher Project.

[20] Alexander Maximov and Alex Biryukov. Two trivial attacks on trivium. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *Selected Areas in Cryptography, 14th International Workshop, SAC 2007, Ottawa, Canada, August 16-17, 2007, Revised Selected Papers*, volume 4876 of *Lecture Notes in Computer Science*, pages 36–55. Springer, 2007.

[21] Manuel Blum, Michael Luby, and Ronitt Rubinfeld. Self-testing/correcting with applications to numerical problems. *J. Comput. Syst. Sci.*, 47(3) :549–595, 1993.

[22] Daniel J. Bernstein. Why haven't cube attacks broken anything. http://cr.yp.to/cubeattacks.html.

[23] Xuejia Lai. Higher order derivatives and differential cryptanalysis. In Richard E. Blahut, Daniel J. Costello, Ueli Maurer, and Thomas Mittelholzer, editors, *Communications and Cryptography : Two Sides of One Tapestry*, pages 227–233, Boston, MA, 1994. Springer US.

[24] Michael Brickenstein and Alexander Dreyer. Polybori : A framework for gröbner-basis computations with boolean polynomials. *Journal of Symbolic Computation*, 44(9) :1326 – 1345, 2009. Effective Methods in Algebraic Geometry.

[25] Domagoj Babic. Satisfiability suggested format. http://www.domagoj-babic.com/uploads/ResearchProjects/Spear/dimacs-cnf.pdf.

[26] Mate Soos. Wonderings of a SAT geek. https://www.msoos.org/.

[27] Armin Biere. Lingeling, plingeling and treengeling. http://fmv.jku.at/lingeling/.

[28] Norbert Manthey. The SAT-solving package riss. http://tools.computational-logic.org/content/riss.php.