Charles Bouillaguet

Les attaques cryptographiques sont-elles toujours meilleures que la force brute ?

(réponse courte : ça dépend)

— Habilitation à Diriger des Recherches —

17 mars 2022

Sorbonne université — CNRS — LIP6

Avant-propos

> MÉPHISTOPHÉLÈS : Mon bon ami, toute théorie est sèche, et l'arbre précieux de la vie est fleuri.
>
> *Faust*, J. W. von Gœthe, traduit par G. de Nerval, 1877

Au lieu de faire une présentation exhaustive de mes travaux passés et de tenter de trouver une suite logique ou un fil conducteur qui les relie, ce manuscrit présente un long raisonnement sur le coût des attaques cryptographiques. Il n'est à mon avis pas facile de condenser le faisceau de présomptions, d'indices ou de raisonnements présentés ici — mais rédiger une HDR était une bonne occasion de les exposer et surtout de mettre mes idées au clair par la même occasion.

La conclusion de tout ceci, présentée dans le chapitre 5, est que certaines attaques cryptographiques considérées comme théoriquement valides ne peuvent probablement pas être exécutées plus efficacement que des techniques très naïves (telles que la recherche exhaustive) dans le monde dans lequel nous vivons. Du coup, ceci pose un certain nombre de questions : qu'est-ce qu'une « attaque » cryptographique ? Quels critères doivent permettre de juger que certaines sont meilleures que d'autres ? Est-il utile de les programmer ou bien la simple menace de leur existence est-elle suffisante ?

J'ai été amené à cette conclusion par un cheminement, dont le point de départ se trouve dans mes tentatives d'implanter des attaques cryptographiques ou des algorithmes pour résoudre des problèmes calculatoires qui jouent un rôle en cryptographie. Cela m'a fait prendre conscience qu'il y a parfois une grosse différence entre la « théorie » et la « pratique ». Je peux essayer d'illustrer ceci en décrivant sommairement mon parcours.

Théorie et pratique. La communauté des chercheurs en algorithmique a un positionnement assez centré sur la théorie. Pour ne donner qu'un exemple, il y a une quinzaine d'années, j'ai découvert la décomposition modulaire des graphes (qui généralise la décomposition en composantes connexes) dans un cours de master de Michel Habib. Une des diapos de ce cours montrait les 36 articles de recherches consacrés au calcul de cette décomposition, entre 1972 et 2008 (et encore, la liste n'était pas présentée comme exhaustive). En 1972, le calcul se fait en $\mathcal{O}\left(n^4\right)$. En 1978, on arrive à $\mathcal{O}\left(n^3\right)$. En 1989, 7 publications plus tard, à $\mathcal{O}\left(n^2\right)$. En 1992, 4 publications après, à $\mathcal{O}\left(n + m\alpha(n, m)\right)$. Enfin, en 1994, 4 publications après, à $\mathcal{O}\left(n + m\right)$. Enfin, en 2008, 21 publications plus tard, toujours $\mathcal{O}\left(n + m\right)$ mais avec un algorithme nettement plus simple.

Des algorithmes sont souvent considérés comme « meilleurs » que les précédents s'ils les améliorent sur le papier. En tout cas, ils ne sont pas toujours programmés ni essayés en pratique. Ils sont publiés dans des colloques ou des journaux respectés. Lorsque j'ai commencé la recherche et que j'ai commencé à étudier l'état de l'art, je me suis tout naturellement « coulé dans le moule » en faisant mienne l'idée que si un algorithme est meilleur sur le papier, alors c'est un progrès scientifique qui mérite d'être communiqué.

J'ai notamment commencé par étudier des attaques génériques, c'est-à-dire des attaques dont la complexité est si grande qu'il serait farfelu de chercher à les programmer. Mes premiers travaux étaient

donc de nature relativement théorique. Inventer un algorithme, puis démontrer qu'il est efficace, suffit à « casser » un mécanisme cryptographique, donc constitue une analyse de sécurité pertinente. L'existence de l'algorithme (et de la preuve de sa complexité), est en soi un certificat de faiblesse du mécanisme cryptographique en question.

Cependant, comme j'ai fait de la cryptanalyse, en plus de chercher des attaques « sur le papier », j'ai souvent été travaillé par la volonté de résoudre des problèmes concrets. Pour casser tel jeu de paramètres, il faudrait résoudre un système de 64 polynômes quadratiques booléens en 64 variables. Est-ce concrètement possible ? Réfléchir à des « défis calculatoires » donne l'occasion de voir les choses autrement, car les performances théoriques des algorithmes passent au second plan devant leur efficacité concrète.

J'ai alors été frappé par le fait que beaucoup d'algorithmes généralement considérés comme étant à l'état de l'art sont complètement inutilisables. *A contrario*, des algorithmes théoriquement inférieurs peuvent fonctionner de manière satisfaisante. C'est notamment le cas dans un des domaines où j'ai été actif, la résolution des systèmes quadratiques booléens.

En fait, ce décalage entre la théorie et la pratique est un phénomène bien documenté. Il y a un exemple archi-connu avec la multiplication de matrices (le meilleur algorithme n'est pas utilisable). Mais du coup, se poser des problèmes concrets amène à regarder d'un autre œil la masse de savoir « théorique » publiée : on est rapidement amené à considérer qu'une bonne partie ne peut être d'aucune utilité pour affronter des problèmes concrets. C'est pour moi une source de frustration.

Une quinzaine d'années après avoir suivi le cours de master sur l'algorithmique des graphes dont il est question ci-dessus, j'ai réalisé qu'il n'existe à peu près aucune implantation publique des algorithmes décrits dans les 36 publications citées. La décomposition modulaire a de nombreuses applications, est-il écrit, mais aucune qui justifie son usage dans le monde réel.

Raisonner uniquement dans des modèles théoriques n'est bien sûr pas une mauvaise chose en soi, mais cela peut aboutir à des concepts assez éloignés de la réalité matérielle, voire même qui n'y sont plus reliés du tout. Don Knuth a écrit il y a précisément 30 ans quelque chose de radical sur le sujet [133] :

> *When theory becomes inbred — when it has grown several generations away from its roots, until it has completely lost touch with the read world — it degenerates and becomes sterile.*

Recherche et ingénierie.

> I can tell you, I don't have money.
> But what I do have
> are a very particular set of skills.
> Skills I've acquired
> over a very long career.
>
> Bryan Mills dans le film *Taken*, 2008

L'informatique est à la fois une science et un art. Affirmer ceci est une grosse banalité ; on pourrait le dire de toutes les disciplines qui ont un caractère scientifique, en particulier de la médecine. L'informatique a une dimension théorique (la « science ») et une dimension technique (« l'art »).

J'ai cependant le sentiment que des scientifiques, surtout ceux qui sont principalement des théoriciens, jettent parfois un regard condescendant envers l'*ingénierie*, qui est l'utilisation de principes scientifiques pour concevoir et construire des machines, des ouvrages d'art ou tout autre chose. Ils peuvent parfois y opposer la *recherche*, c'est-à-dire les travaux qui tendent à la découverte de connaissances nouvelles.

Dans l'excellent article « *SHA-1 is a Shambles* » [148], Leurent et Peyrin expliquent qu'ils ont implanté leur incroyable attaque en collision avec préfixe choisi sur SHA-1, et précisent :

> *This attack is extremely technical, contains many details, various steps, and requires a lot of* engineering *work.* [souligné par nous]

Je suis tenté, en faisant preuve d'un peu de mauvaise foi, de penser que les auteurs voulaient dire : *mener à bien ces travaux de recherche a nécessité que nous fassions un gros travail d'ingénierie. Ce*

dernier nous a pris beaucoup de temps et beaucoup d'énergie. Globalement, ça nous a bien cassé les pieds, mais c'était obligatoire.

Les travaux de recherche en cryptographie contiennent parfois une part d'ingénierie (mise en œuvre d'attaques, implantations de mécanismes de chiffrement/signature, de protocoles, ...). Cependant, aux yeux d'une (bonne) partie de notre communauté de recherche, un programme informatique vaut moins qu'un bon théorème. Je pense que ceci est en train d'évoluer, et que les contributions sous forme de logiciel sont de mieux en mieux reconnues, mais la tendance est toujours présente.

Un point de vue légèrement caricatural consisterait à affirmer que l'ingénierie ne consiste qu'à mettre en œuvre des idées existantes sans devoir en apporter de nouvelles, alors qu'*a contrario* ce serait l'essence même de la recherche. Mais si on regarde dans les détails, on peut se rendre compte que c'est beaucoup plus compliqué. Il y a des travaux de recherche qui se contentent d'utiliser des idées existantes autrement, et des réalisations concrètes qui ont nécessité une approche radicalement nouvelle.

La programmation, c'est un art [132] (de l'ingénierie). À ce titre, les scientifique pourraient penser que la partie « recherche » du travail (difficile, donc valorisante) est d'inventer les idées algorithmiques. Les mettre en œuvre concrètement serait comparativement facile et donc moins valorisant, et pourrait être délégué à des ingénieurs. Peu de conférences de cryptologie publieraient un article décrivant dans les détails une implantation de haute qualité d'un algorithme déjà connue.

En réalité, traduire la vague description d'un algorithme en code efficace nécessite souvent de se poser des problèmes algorithmiques cachés dans un modèle sensiblement plus complexe que celui de la « recherche théorique ». Il faut avoir une démarche expérimentale (scientifique), oublier ses préjugés, etc. Pour citer à nouveau le même texte de Knuth : « *Software is hard !* ». Seuls ceux qui n'ont jamais essayé peuvent penser que passer de la théorie à la pratique est facile.

En informatique, l'art et la science sont les deux facettes de la même pièce. Je ne prend pas beaucoup de risques en répétant l'opinion de Don Knuth [133], mais à mon avis, opposer recherche et ingénierie est contre-productif. J'ai été convaincu, par mon expérience, que raisonner sur des problèmes concrets, tenter de programmer des algorithmes, les voir fonctionner « en vrai », est une fructueuse source de connaissance et de questions. Mais j'ai le sentiment que ce point de vue est, pour l'instant, minoritaire.

Du coup, ce texte présente des raisonnements (théoriques !) pour tenter de montrer que la manière actuelle de faire de la cryptanalyse « sur le papier » est assez éloignée de la réalité matérielle, ce que d'ailleurs personne ne conteste. Ce manuscrit décrit aussi mes tentatives, plus ou moins fructueuses, de valider en pratique une partie de ces raisonnements. À assez peu de choses près, tout ce qui est exposé ici est inédit.

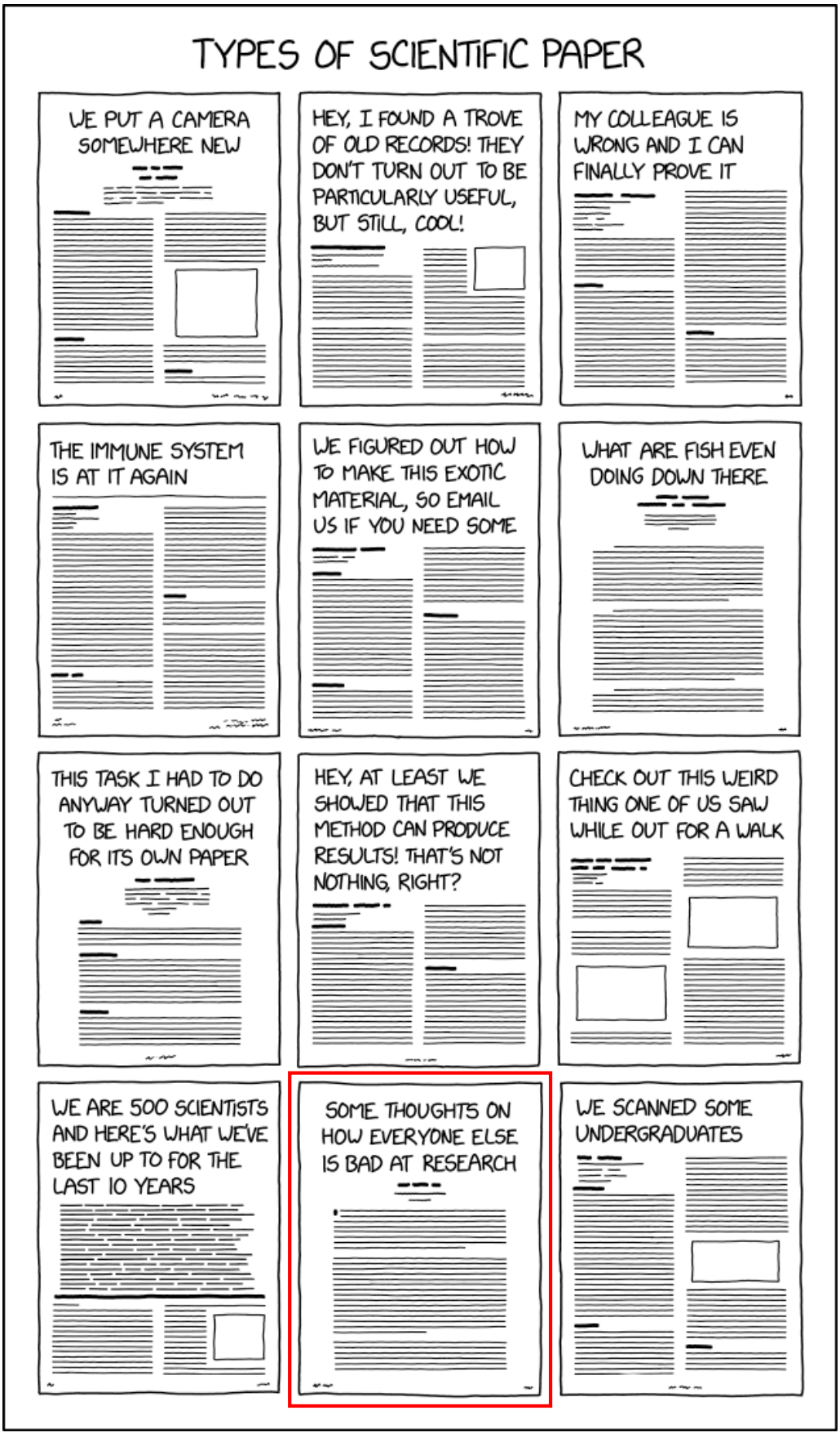


FIGURE 1 – XKCD (`https://xkcd.com/2456/`)

Table des matières

Chapitre 1

Introduction

La cryptographie joue un rôle de premier plan dans la sécurité des systèmes d'information. Un petit nombre de mécanismes cryptographiques sont largement déployés, et garantissent la sécurité de la plupart des échanges électroniques. Par exemple, la signature RSA sert à garantir l'authenticité de toutes les cartes de crédit européennes, et permettent d'échanger du courrier électronique sécurisé (c.a.d. confidentiel et authentifié). La signature RSA, combinée avec l'échange de clef Diffie-Hellman, sont présents dans l'écrasante majorité des connections sécurisée sur internet, car ce sont des composants de la couche TLS. Ils sécurisent des connections à des boites mail, permettent aux navigateurs web d'authentifier des sites bancaires, de e-commerce ou de services publics ; ils permettent des connections à des réseaux privés virtuels (VPN), etc. En France, la signature de Rabin est largement déployée puisqu'elle est présente dans les badges VIGIK qui donnent accès à des halls d'immeubles.

Les mécanismes cryptographiques sont utilisés pour garantir des propriétés de sécurité, telles que la confidentialité, l'intégrité et l'authenticité des communications. Face à cela, on suppose que des adversaires tentent de briser les garanties offertes. S'ils y parviennent, on dit qu'ils *cassent* le mécanisme utilisé.

La cryptographie à clef publique repose sur la difficulté de problèmes calculatoires bien définis. Concrètement, la sécurité de RSA repose sur la difficulté de factoriser des grands entiers, tandis que celle de l'échange de clef Diffie-Hellman repose sur la difficulté du calcul du logarithme discret (que ce soit sur le groupe multiplicatif des entiers modulo p ou bien sur le groupe des points d'une courbe elliptique).

La période récente a vu l'essor du déploiement et de l'usage des mécanismes cryptographiques basés sur les courbes elliptiques, qui sont maintenant presque systématiquement utilisés lors des connections sécurisés (avec ECDHE et ECDSA). Ils sont en effet standardisés dans la dernière version du standard TLS (TLS 1.3). Ils offrent de meilleures performances et des tailles de clefs réduites par rapport à RSA et à l'échange de clef Diffie-Hellman classique.

1.1 Cryptographie « post-quantique »

Depuis la mise au point de l'algorithme de Shor en 1994, il est connu que des ordinateurs quantiques suffisamment gros, s'ils pouvaient être construits, marqueraient la fin du règne des mécanismes à clef publique les plus courants aujourd'hui : l'algorithme de Shor casserait en temps quasi-linéaire le chiffrement RSA, la signature RSA, la signature DSA, l'échange de clef Diffie-Hellman, ainsi que les variantes sur courbes elliptiques. Cette perspective a poussé le NIST (une agence civile du gouvernement américain) à lancer en 2016 une compétition publique dans le but de faire émerger de nouvelles primitives cryptographiques « post-quantiques ». La compétition est entrée dans sa phase finale, à l'heure de la rédaction de ce texte.

Les mécanismes cryptographiques « post-quantiques » reposent sur des problèmes calculatoires différents (parfois associés à un problème de décision NP-difficile), qui ne seraient pas facilement résolus même avec de gros ordinateurs quantiques. Ces problèmes incluent le décodage des codes linéaires

aléatoires, la résolution des systèmes polynomiaux multivariés sur un corps fini, les problèmes difficiles dans les réseaux euclidiens (SVP, CVP, ...) et ceux qui s'y ramènent (LWE, NTRU, ...), les marches dans des graphes d'isogénies, etc.

Ces problèmes calculatoires « post-quantiques » sont relativement récents, comparés à Diffie-Hellman (1976) et RSA (1979). À la notable exception du chiffrement de McEliece, qui date lui de 1978, la plupart des autres n'ont pas été proposés avant le milieu des années 1990.

1.2 Sécurité, attaques et algorithmes

La sécurité des mécanismes cryptographiques, surtout à clef publique, repose sur la difficulté supposée de résoudre certains problèmes calculatoires bien précis. En effet, il existe des algorithmes dont l'exécution révélerait la solution d'instances de ces problèmes, et donc briserait les propriétés de sécurité correspondantes. Il y a par exemple des algorithmes de factorisation des grands entiers, dont l'exécution complète, si elle était possible, permettrait de casser des instances de RSA.

En général, les primitives cryptographiques sont considérées comme sûres lorsque l'exécution de ces algorithmes est concrètement impossible : il faut ou bien qu'ils soient trop lents, ou bien que leur probabilité de succès soit trop faible. Un algorithme suffisamment efficace qui brise une propriété de sécurité offerte par un mécanisme cryptographique est une *attaque* contre le mécanisme en question.

Si la cryptographie est la science du secret, la *cryptanalyse* est l'art de casser des mécanismes cryptographiques. Il s'agit d'en exploiter des défauts ou des propriétés imprévues dans le but d'accéder à des secrets, d'usurper des identités, ou d'altérer l'intégrité des communications.

La « sécurité prouvée » est une tendance forte dans le monde de la cryptographie, qui consiste à obtenir des preuves mathématiques que la difficulté du problème calculatoire sous-jacent est non seulement nécessaire pour la sécurité, mais aussi suffisante. Ceci se présente sous la forme de théorèmes affirmant qu'il est à peu près aussi difficile de briser la propriété de sécurité que de résoudre le problème calculatoire.

La sécurité prouvée est importante, car elle garantit qu'il n'est pas possible de « contourner l'obstacle » que constitue le problème calculatoirement difficile. En d'autres termes, il s'agit de prouver qu'il n'y a pas de faiblesse exploitable par les cryptanalystes. Par conséquent, la sécurité des mécanismes ne sera pas brisée sans mobiliser des ressources calculatoires hors du commun, ou bien sans une percée algorithmique aussi spectaculaire qu'improbable.

Dans tous les cas, au final la sécurité se réduit à des hypothèses de complexité algorithmique, c'est-à-dire dans la conviction que telle ou telle tâche calculatoire, par exemple la factorisation des grands entiers, est impraticable. Dans ce cas précis, cette conviction est renforcée car le problème a une formulation particulièrement simple, et que malgré quelques siècles d'efforts, l'humanité n'a pas encore trouvé d'algorithme efficace pour factoriser de très grands entiers. Il est par conséquent crédible que ces algorithmes ou bien n'existent pas, ou bien sont très difficiles à trouver. Dans les deux cas, il semble improbable qu'ils soient découverts prochainement.

Il n'y a cependant pas de garantie absolue que des algorithmes efficaces de factorisation des grands entiers n'existent pas. En réalité, il n'y a pas, à ma connaissance, de preuve mathématique de l'existence d'une fonctions à sens unique, quelle qu'elle soit.

1.3 La cryptanalyse est un service public

Il s'ensuit qu'il est nécessaire de réévaluer en permanence la difficulté concrète de certains problèmes calculatoires qui sont devenus, de fait, des « hypothèses de sécurité ». Ceci est le sujet de ce qu'on pourrait appeler la *cryptanalyse algorithmique*, pour reprendre le titre du livre de Joux [125].

En suivant les progrès de la recherche publique en cryptanalyse, des éditeurs de logiciels qui ne sont pas des experts en cryptographie peuvent maintenir leurs produits à jour, tandis que les citoyens lambda

peuvent mieux comprendre quel niveau de confiance avoir dans les outils cryptographiques qu'ils sont sommés d'utiliser, qu'ils utilisent à leur insu, ou dont on leur interdit l'utilisation. Dans tous les cas, la recherche en cryptanalyse est un service public utile à tous, y compris aux entreprises commerciales qui n'ont pourtant, de leur point de vue, aucune raison d'y investir quelque ressource que ce soit.

La recherche en cryptanalyse est de nature à la fois théorique et pratique. Sur le plan théorique, des faiblesses dans des schémas cryptographiques peuvent être découvertes et exposées « sur papier » : il s'agit de décrire une attaque (c'est-à-dire un algorithme), dont l'exécution, si on disposait de ressources suffisantes pour la mener à bien, devrait briser des propriétés de sécurité plus efficacement que prévu. L'attaque découverte en 2011 par Isobe [120] contre le système de chiffrement par bloc soviétique GOST (dont les spécifications ont été publiées en 1994) entre dans cette catégorie : elle retrouve la clef secrète de 256 bits après un calcul plus ou moins équivalent à 2^{224} évaluations du dispositif, ce qui est donc sensiblement plus rapide que la recherche exhaustive.

Ce genre d'attaques théoriques (les anglophones parlent de *certificational attacks*) est important, par exemple pour les organismes de standardisation qui doivent choisir un dispositif ayant vocation à être largement déployé et utilisé pendant une longue période. Autant en choisir un qui n'a *aucun* défaut, même uniquement théorique et d'une importance pratique négligeable.

D'un autre côté, lorsqu'un mécanisme cryptographique est cassé *en pratique* et pas seulement sur le papier, les conséquences sont généralement plus dramatiques, surtout lorsqu'il est largement déployé. Ceci met bien plus de pression sur les acteurs industriels pour qu'ils mettent à jour leur produit, et ceci peut avoir des coûts importants.

Le DES est un système de chiffrement par bloc conçu par IBM sous l'égide du gouvernement américain (la NSA a joué un rôle dans la finalisation de ses spécifications) et publié en 1975. C'est devenu un standard *de facto* et il a été abondamment étudié. Ses clefs ne font que 56 bits, un choix surprenant et instantanément critiqué, qui n'a depuis jamais été justifié par ses concepteurs. Dans les années 1970, casser le DES par une recherche exhaustive des 2^{56} clefs possibles ne semblait pas faisable. Toutefois, l'augmentation naturelle de la puissance de calcul disponible a fini par rendre cette attaque par force brute réaliste. Pour faire face à cette éventualité, le gouvernement américain a décidé en 1995 de mettre en place des solutions avec des clefs plus longues. Il a d'abord suggéré le triple DES pour augmenter la taille de la clef à 112 ou 168 bits à peu de frais, en recyclant le DES, puis il a lancé une compétition publique (la première du genre) pour choisir un nouveau système de chiffrement par bloc, l'AES. Il était temps : en 1997 des amateurs, utilisant un système de calcul distribué sur internet, ont cassé une clef du DES par force brute, en quelques heures. De nos jours, une machine spécialisée, contenant des FPGAs, capable de casser le DES en quelques jours, coûte le même prix qu'une petite voiture [137].

Le cas de la fonction de hachage SHA-1, également conçue par la NSA en 1995, et largement déployé ensuite, est également très intéressant. Les premiers résultats de cryptanalyse « théoriques », dûs à Wang *et al* ont été publiés en 2005 [199]. SHA-1 a continué d'être largement utilisé. Lorsque (l'inévitable) collision réelle a été produite par une équipe de recherche universitaire en 2017 [180], avec le soutien technique de Google, le niveau de sécurité offert par SHA-1 a été instantanément réduit à zéro, et la dépréciation de la fonction devenait brusquement une urgence... très urgente.

La situation des mécanismes à clef publique qui reposent sur des problèmes de la théorie des nombres (factorisation, logarithme discret) est assez différente. Les algorithmes de factorisation sont publics [1] et leur complexité est connue. Cela permet en principe d'extrapoler la quantité de calcul nécessaire pour casser une clef RSA d'une taille donnée. Le processus est néanmoins assez imprécis, à cause de la nature précise des calculs à effectuer ; cela peut conduire à des sous-estimations ou des surestimations de la difficulté réelle du problème.

La publication régulière de records de calcul facilite le processus, en l'alimentant avec des points de données réalistes. En 2009, un groupe international d'équipes de recherches universitaires a factorisé une clef RSA de 768 bits, en utilisant les ressources calculatoires à leur disposition dans les universités [128]. Ceci a été suivi en 2017 par le calcul d'un logarithme discret modulo un nombre premier de

1. Enfin, on l'espère...

768 bits [129]. Plus récemment, en 2019 et 2020, une factorisation et un logarithme discret de 795 bits ont été calculés avec le logiciel libre CADO-NFS [40], suivis de près par une factorisation de 830 bits.

Ces records ont plusieurs conséquences pratiques. Premièrement, le *Oakley Group 1* (modulo un nombre premier de 768 bits), standardisé dans la RFC 2412, est cassé dans la pratique, de même que le système des badges VIGIK (qui, à ma connaissance, utilise toujours des clefs de 768 bits). Deuxièmement, suivre cette évolution permet d'extrapoler les capacités d'agences gouvernementales dotées de moyens importants. Ces travaux pratiques ont donné de la crédibilité à l'hypothèse, énoncée par Edward Snowden en 2013, que la NSA aurait calculé des logarithmes discrets de 1024 bits (par exemple, en cassant le *Oakley Group 2*), ce qui lui aurait permis de déchiffrer une portion importante du trafic internet « sécurisé » [1].

1.4 Défis et implantations de référence

Pour permettre au public de suivre les progrès de la recherche en cryptanalyse, plusieurs solutions sont envisageables. La première consiste à disposer d'un jeu de défis de difficulté variable. Les cryptanalystes peuvent alors publier la solution des défis lorsqu'ils parviennent à les résoudre. En 1991, l'entreprise *RSA labs* a publié une série de clefs RSA de tailles variées [202] — toutes les clefs de 830 bits ou moins ont été cassées à ce jour. L'entreprise Certicom a publié des défis consistant à calculer un logarithme discret sur une série de courbes elliptiques en 1997 [71] — seules les instances « jouet » de 112 bits ou moins ont été cassées. Les « *Fukuoka MQ Challenges* » [205] rassemble des systèmes polynomiaux aléatoires, significatifs pour la cryptographie post-quantique. La « *Lattice Challenge collection* » de l'université technique de Darmstadt [66] héberge quant à elle une collection de défis difficiles sur les réseaux euclidiens, eux aussi significatifs pour la cryptographie post-quantique. Enfin, le site web « *Decoding Challenge* » [10] héberge dans le même esprit une collection d'instance de problèmes de décodage de codes linéaires aléatoires.

Ces collections de défis sont utiles, car elles permettent à des groupes de recherche concurrents de démontrer leurs progrès sans même devoir expliquer comment ils font — simplement en publiant la solution. Cependant, des implantations *open-source* à l'état de l'art des attaques cryptographiques sont préférables, car elles peuvent être étudiées et améliorées par d'autres chercheurs. Il existe de tels programmes libres, mais pas dans tous les domaines. `CADO-NFS` [183] est l'un d'entre eux, pour la factorisation des grands entiers et le calcul des logarithmes discrets dans certains groupes finis. `fplll` [82] est aussi une librairie à l'état de l'art pour la réduction de réseau, tandis que la librairie `G6K` [3] contient des algorithmes de crible sur les réseaux.

Malheureusement, de telles implantations de référence, suffisamment faciles à utiliser et dotées d'une documentation exploitable, ne sont pas disponibles pour tous les problèmes intéressants d'un point de vue cryptographique, et c'est bien dommage. La caractère publiquement accessible de CADO-NFS a été l'un des ingrédients qui a permis le succès de la preuve de concept de l'attaque LOGJAM [1]. Cette dernière fonctionne en rabaissant le niveau de sécurité à une instance faible du problème du logarithme discret, puis en résolvant ce dernier avec `CADO-NFS`. Ce calcul est faisable, mais non-trivial.

1.5 Cryptanalyse « sur le papier »

> La menace est plus forte que l'exécution.
>
> Ce grand principe des échecs, souvent attribué à tort à Aaron Nimzowitsch (1886–1935), a été formulé pour la première fois par Karl Eisenbach (1836–1894), secrétaire de la Société viennoise des échecs.

La sécurité des mécanismes cryptographiques peut se quantifier en estimant leur résistance face à des adversaires dotés de pouvoir plus ou moins étendus : ils peuvent faire chiffrer/signer/authentifier les données qu'ils veulent, ou bien peut-être seulement observer des paires clair-chiffré aléatoires ; ils peuvent ou pas observer des canaux auxiliaires ; ils peuvent ou pas corrompre une partie des participants à un protocole.

Pour raisonner de manière rigoureuse et obtenir des garanties fiables sur la sécurité, il est logique d'être *pessimiste* et d'exagérer les pouvoirs des adversaires. Par exemple, l'hypothèse qu'un adversaire peut faire chiffrer et déchiffrer ce qu'il veut par un oracle n'est pas très facile à traduire en un scénario réaliste dans le monde réel. Mais si les mécanismes cryptographiques résistent « sur le papier » à des adversaires extraordinairement puissants, alors ils devraient résister aussi à des adversaires plus réalistes.

Il est par conséquent habituel de se placer dans des scénarios *très favorables* aux adversaires. D'un point de vue calculatoire, les attaques cryptographiques sont généralement décrites (souvent de façon assez informelle) dans un modèle de calcul qui les favorise vraiment : mémoire illimitée, absence de contraintes pratiques (réseaux, parallélisme, ...), pas de coûts énergétiques, pas de temps perdu dans les communications, pas de pannes, etc. Le principal modèle de calcul utilisé est celui de la *Random Access Machine* dont il sera largement question dans le chapitre 2.

Dans ce modèle, on fixe une limite à la puissance de calcul dont les adversaires disposent, sinon la plupart des mécanismes cryptographiques deviennent invalides. Dans le cadre de la cryptographie symétrique, on impose généralement que l'adversaire n'ait pas les moyens d'effectuer une recherche exhaustive sur tous les bits de la clef secrète. Concrètement, cela veut dire une puissance de calcul limitée à 2^{128} opérations en 2022. Si aucun algorithme connu ne parvient à briser une propriété de sécurité plus vite que prévu dans ce modèle de calcul favorable, alors on peut raisonnablement supposer qu'aucun programme réel s'exécutant sur de vrais ordinateurs n'y parviendra non plus.

Par la force des choses, les attaques cryptographiques décrites dans ce modèle calculatoire favorable aux adversaires sont potentiellement des objets « théoriques », c'est-à-dire des algorithmes décrits plus ou moins précisément, qui n'ont jamais été concrètement traduit en code exécutable et qui donc *a fortiori* n'ont jamais été exécutés. Leurs mérites respectifs sont donc comparés dans un modèle de calcul abstrait, sans recourir à des expériences sur du vrai matériel. L'attaque sur GOST d'Isobe dont il est question ci-dessus nécessite 2^{224} évaluations du système de chiffrement par bloc ; ceci a été amélioré à 2^{192} par Dinur, Dunkelman et Shamir un an plus tard [92]. La deuxième attaque est meilleure que la première car elle obtient le même résultat avec moins de ressources (moins de temps, toutes choses égales par ailleurs). Tenter d'implanter ces attaques n'a pas vraiment de sens, car l'humanité ne dispose pas de la puissance de calcul nécessaire à leur exécution.

Dans une certaine mesure, la cryptanalyse peut donc être une *théorie* où des informaticiens décrivent des algorithmes (qui ne sont souvent pas implantables) et comparent leurs efficacités respectives dans un modèle de calcul potentiellement irréaliste. La « dangerosité » des attaques est évaluée en estimant le nombre d'opérations nécessaire à leur exécution, ce qui est la principale mesure de complexité dans le modèle de calcul. Ces complexités calculatoires peuvent être données asymptotiquement, en fonction d'un paramètre de sécurité (« L'attaque casse HMAC en $2^{3n/4}$ opérations » [169]), ou parfois avec des nombres d'opérations « concrets » lorsque des primitives spécifiques sont visées (c'est le cas des attaques sur GOST déjà citées).

Dans la plupart des circonstances, il y a des attaques *génériques* qui peuvent s'appliquer sur de vastes classes de mécanismes cryptographiques : reconstituer les secrets en testant toutes les possibilités, inverser une fonction à sens unique en testant des antécédents aléatoires jusqu'à ce que ça marche, trouver une collision par la méthode *rho*, etc. Dans les mécanismes cryptographiques raisonnables, les tailles de paramètres sont choisies pour offrir une résistance suffisante à ces tentatives directes. Par conséquent, une attaque cryptographique digne de ce nom doit faire mieux : on s'attend en particulier à ce qu'elle nécessite moins d'opérations dans le modèle de calcul usuel, moins de mémoire, l'accès à moins de « données », l'accès à un oracle moins puissant, etc.

Une attaque, décrite sur le papier, ne nécessitant qu'un nombre d'opérations dangereusement faible dans le modèle de calcul « casse » donc le mécanisme en question, qui doit alors être mis au rebut, même si l'attaque est encore loin de se concrétiser en pratique. La sécurité revient alors à l'absence d'attaques efficaces dans le modèle de calcul. Ce point de vue est fiable : si la meilleure attaque connue dans le modèle de calcul favorable aux adversaires nécessite 2^x opérations, et que x est trop grand,

alors elle ne sera jamais réalisable dans le monde réel.

Ce raisonnement est concrètement utilisé par les concepteurs de mécanismes à clef publique qui sont confrontés au problème concret de devoir choisir des tailles de paramètres offrant une résistance suffisante aux attaques. Le schéma de signature GeMSS [70, §8], actuellement en « liste complémentaire » du 3ème tour de la compétition post-quantique du NIST, repose sur la difficulté de résoudre des systèmes polynomiaux en plusieurs variables. Ses concepteurs suivent cette approche prudente : l'algorithme `BooleanSolve` de [16] était, lors du début de la compétition, celui qui nécessitait (asymptotiquement) le moins d'opérations. Sa complexité est $\tilde{\mathcal{O}}\left(2^{0.792n}\right)$, donc les concepteurs de GeMSS ont choisi $n = 162$; l'exécution de l'algorithme, qui permettrait de forger une signature, nécessite alors 2^{128} opérations au moins (en supposant que la constante vaille un). Malheureusement pour eux, un meilleur algorithme a été développé par Dinur en 2021, de complexité $\tilde{\mathcal{O}}\left(2^{0.694n}\right)$ [89], ce qui invalide le raisonnement ayant conduit à ce choix de n. Du point de vue de la sécurité, ce n'est cependant pas du tout un problème, car il suffit d'augmenter n très légèrement pour être hors de la zone de danger.

La cryptanalyse peut donc, dans le fond, se résumer à un jeu mathématique se déroulant dans un modèle abstrait largement déconnecté de la réalité. Elle peut jouer un rôle utile pour la sécurité, sans pour autant que ceux qui s'y adonnent aient à s'embarrasser avec de vrais ordinateurs ni à savoir comment les programmer.

1.6 Cryptanalyse avec de vrais ordinateurs

De fait, seule une (petite) partie des attaques cryptographiques publiées dans la littérature ont été programmées et ont pu être complètement exécutées jusqu'au bout sur des machines réelles. Ce sont celles-là qui sont « pratiques ». Certaines ont eu un gros impact car elles s'appliquent à des schémas cryptographiques largement déployés, ou bien potentiellement prometteurs. La petite liste qui suit n'est pas du tout exhaustive (les attaques pratiques exploitant les faiblesses de RC4 sont notoirement absentes) :

- En 2003, l'inversion dans les faits d'un challenge HFE censé offrir 80 bits de sécurité par Faugère et Joux [103] a contribué à convaincre les cryptologues que HFE était cassé, même si la complexité asymptotique de l'attaque était alors inconnue. En réalité, des paramètres sûrs pour HFE existent, mais pas ceux de départ.
- En 2007, Dubois, Fouque et Shamir ont pu forger des signatures SFLASH en quelques minutes [93], alors que SFLASH était en voie de standardisation par le consortium européen NESSIE.
- En 2007, Stevens, Lenstra and de Weger [181] ont créé une collision pour MD5 avec deux préfixes choisis. Ceci a permis la création d'une autorité de certification pirate pour les certificats SSL. Par contrecoup, ceci a accéléré la dépréciation de MD5. Le code source de l'attaque est public.
- En 2008, Bernstein, Lange et Peters ont cassé le chiffrement McEliece [25], en tout cas les paramètres proposés initialement par McEliece en 1978 [158] et censés offrir 64 bits de sécurité. Canteaut, Chabaud et Sendrier avaient annoncé 10 ans plus tôt que les paramètres en question n'offraient pas la sécurité voulue [68, 69]. L'article [25] affirme que ses auteurs « prévoient » de publier le code source de l'attaque ; au moment de la rédaction de ce manuscrit, donc 13 ans plus tard, ce n'est pas le cas et Bernstein a refusé de me le transmettre.
- En 2010, la factorisation d'une clef RSA de 768-bit par une équipe internationale [128] a été un marqueur clair dans les progrès des algorithmes de factorisation. Celui-ci a été complémenté en 2017 par le calcul d'un logarithme discret modulo un nombre premier de 768 bits [129]. Les programmes utilisés n'étaient pas publics. En 2020, des factorisation de 795 et 830 bits ainsi qu'un log. discret modulo un nombre premier p de 795 bits ont été calculés tous les deux par le même logiciel libre, `CADO-NFS` [40].
- En 2012, Bos, Kaihara, Kleinjung, Lenstra et Montgomery [37] ont calculé un log. discret sur la courbe elliptique `secp112r1`, c'est-à-dire modulo $\left(2^{128} - 3\right)/(11 \times 6949)$, un nombre premier de 112 bits. C'est le record actuel, et il a un « *cool factor* » assez élevé puisque le calcul a été

effectué sur un *cluster* de consoles de jeu vidéo (playstation 3). Le code source n'est pas public et Bos m'a confirmé qu'il ne le serait pas.

— En 2017, Stevens, Bursztein, Karpman, Albertini et Markov ont créé la première collision pour la fonction de hachage SHA-1 [180], ce qui n'a fait que renforcer le besoin de la déprécier à son tour. Il faut noter que Wang, Yin et Yu avaient découvert la possibilité « théorique » d'une attaque 15 ans auparavant [199]. Le code source de l'attaque est public. Ceci a été étendu ensuite à des collisions avec préfixes imposés par Leurent et Peyrin [148] en 2019. Le code n'est, cette fois, pas public.

— En 2020, Ding, Deaton, Schmidt, Vishakha et Zhang ont publié une attaque contre LUOV, un candidat au second tour de la compétition *post-quantique* du NIST reposant sur la difficulté de la résolution de systèmes polynomiaux. L'attaque ramène la contrefaçon d'une signature à la résolution de systèmes polynomiaux modulo deux de taille modeste. L'attaque a pu être implantée et fonctionne en 210 minutes [84] avec une recherche exhaustive sur FPGAs.

— En 2022, Beulens a publié une attaque contre un jeu de paramètre pour Rainbow, un candidat au troisième tour de la compétition *Post-Quantique* du NIST reposant lui aussi sur la difficulté de la résolution de systèmes polynomiaux. L'attaque ramène la contrefaçon d'une signature à la résolution de systèmes de 64 polynômes quadratiques *et* 64 équations linéaires en 96 variables sur $\mathbb{F}_{16}$ (sans les équations linéaires ce serait infaisable). Le code source est public et l'attaque a pu être menée a bien en « un week-end », grâce à la disponibilité d'un logiciel de résolution d'équations dû à Niederhagen et Chou.

Il est frappant de constater que nombre d'attaques cryptographiques « pratiques » ne sont pas accompagnées des programmes correspondants. Parfois, elles ont un résultat vérifiable (par exemple en produisant le résultat d'un calcul). Mais parfois, il faut se contenter de la parole des auteurs et d'un tableau qui détaille les temps d'exécution des différentes phases du calcul. La tendance actuelle semble cependant être à contraindre les chercheurs à publier leurs programmes, au nom de l'intégrité scientifique et de la vérifiabilité/reproductibilité des résultats.

Certains chercheurs ont essayé de se restreindre à trouver des attaques pratiques. Par exemple, au lieu d'essayer de casser le plus de tours de l'AES en faisant moins d'opérations que la recherche exhaustive, Bar-On, Dunkelman, Keller, Ronen et Shamir ont essayé de casser le plus de tours possible en pratique [14]. Ils affirment avoir vérifié une partie de leurs attaques en les ayant programmées.

En fait, une partie des attaques cryptographiques, pourtant importantes, n'ont jamais été programmées et n'ont pas vraiment vocation à l'être, par exemple parce que leur complexité est hors de portée des capacités calculatoires de l'humanité.

Dans d'autres cas, implanter les attaques et tenter de les exécuter est le seul moyen de trancher la question de leur faisabilité pratique. C'est notamment le cas de la factorisation des grands entiers, où il est difficile d'estimer de façon fiable le temps nécessaire à la factorisation de (disons) RSA-1024 uniquement à partir de la complexité asymptotique en $L_{1/3}$, sans avoir effectué plusieurs factorisations plus petites (512, 768, 896, etc.). En effet, la phase d'algèbre linéaire n'est pas « massivement parallèle » et nécessite un réseau de communication performant.

1.7 Transition de phase

Une attaque, décrite sur le papier, ne nécessitant qu'un nombre d'opérations dangereusement faible dans le modèle de calcul, se traduit-elle par une menace concrète ? Si elle ne nécessite que 2^x opérations, et que x est dans la plage où le calcul est envisageable en pratique (disons $x \leq 64$ en étant optimiste), que va-t-il se passer ? Autrement dit : sera-t-il possible de l'implanter et de l'exécuter sur de vraies machines ? Le moins qu'on puisse dire, c'est que « *ça dépend* ». Une partie du problème repose sur le fait que le modèle de calcul est favorable aux adversaires et que passer d'une description de haut niveau d'un algorithme dans ce modèle simplifié à du code exécutable sur une vraie machine recèle son lot de problèmes et de surprises.

La compétition « post-quantique » du NIST fourni un exemple intéressant. Le schéma de signature multivarié LUOV a été cassé par Ding, Deaton, Vishakha et Yang [85]. Forger une signature pour le jeu de paramètres censé offrir 128 bits de sécurité se ramène à la résolution de deux systèmes (sous-déterminés) de 57 équations quadratiques modulo 2. Ceci est faisable en pratique et les auteurs l'ont en effet réalisé avec recherche exhaustive sur des FPGAs qui met un peu moins de 4 heures. Pourquoi n'ont-ils pas utilisé l'un des nombreux algorithmes asymptotiquement plus efficaces (discutés section 4.6) ? La réponse courte est : parce qu'ils sont inutilisables en pratique.

Pour commencer, lorsqu'on passe de la description de haut niveau d'un algorithme à du code véritablement exécutable, les constantes et même parfois les facteurs logarithmiques dissimulés dans les notations $\mathcal{O}(...)$ apparaissent. Ceux-ci dépendent de la qualité de l'implantation et il est en fait *difficile* d'écrire des implantations efficaces. Les nombres d'opérations concrets des attaques « améliorées » peuvent être asymptotiquement plus faibles que ceux des attaques naïves, tout en étant concrètement plus élevés pour de petites valeurs des paramètres qui sont parfois les seules attaquables en pratique.

Le cas de la multiplication de matrice est emblématique. En 1987, Coppersmith et Winograd ont proposé un algorithme capable de multiplier deux matrices en temps $\mathcal{O}\left(n^{2.3755}\right)$. Ceci a été récemment amélioré à $\mathcal{O}\left(n^{2.3728596}\right)$ par Alman et Vassilevska Williams [5]. Alors qu'il y aurait pourtant un enjeu considérable du point de vue du calcul scientifique, ces algorithmes n'ont jamais été implanté et ils sont inutilisable en pratique tant la constante est grande. Ce sont des joujoux théoriques.

L'algorithme `BooleanSolve` de [16] dont il a déjà été question est un autre exemple plus proche de la cryptographie. Il n'a jamais été implanté et ses auteurs estiment qu'il n'est meilleur que la recherche exhaustive que pour des systèmes en plus de 200 variables. La recherche exhaustive nécessite alors 2^{200} opérations, ce qui est complètement hors de portée de l'humanité. L'algorithme n'a donc a priori aucun intérêt pratique.

Jusque-là théorie et pratique ne se contredisent pas. L'étude théorique des algorithmes est souvent imprécise, même si on peut parfois expliciter les constantes ; Knuth le fait par exemple pour les algorithmes de tri classiques dans [134]. Pour rester sur le problème de la résolution des systèmes d'équations polynomiales modulo 2, Dinur a présenté en 2021 un algorithme qui nécessite moins de $n^2 2^{0.815n}$ opérations [88] — ici la borne est « concrète », il n'y a pas pas de $\mathcal{O}(...)$. En fait, la réalité est un peu plus compliquée, car en réalité il y a une constante *et* un facteur polynomial caché — Dinur prétend que les deux sont proches de 1.

Mais il peut y avoir d'autres problèmes dont les causes sont plus profondes. Si une attaque nécessite 2^x opérations et que x est vraiment faible ($x \approx 30$), alors elle sera implantable et pratiquement réalisable. Mais si x est plus grand (disons $x \geq 50$), alors ça n'est pas du tout évident, et toutes les considérations « pratiques » qui sont ignorées par le modèle calculatoire favorable aux adversaires vont alors faire une entrée fracassante. Il faut alors se mettre à discuter de la vitesse du réseau, de la taille des caches, de stratégies de parallélisation, de mécanismes de détection et de correction des pannes, de mémoire limitée, de la latence des entrées-sorties, etc.

Le cas du double-DES est intéressant : il a été considéré comme « cassé » depuis le milieu des années 1970, car dans le modèle abstrait il n'est pas plus sûr que le simple-DES. En 2022, casser le simple-DES est à la portée du grand public (cf. ci-dessus), mais casser le double-DES ne l'est pas du tout ! C'est *peut-être* faisable avec les moyens des Etats les plus riches (la section 4.2.6 discute des détails).

Si l'attaque A s'exécute en temps 2^n, sans mémoire, et si l'attaque B nécessite $2^{0.875n}$ opérations et autant de bits de mémoire, laquelle est la meilleure ? Autrement dit, laquelle est « la plus dangereuse » ? En tant que collectivité, les cryptologues répondent « B ». Dans le modèle favorable aux adversaires, elle nécessite moins de temps, donc elle est plus proche d'être « faisable ».

Il est frappant de regarder ce qui se passe lorsque $n = 64$. Des attaques nécessitant de l'ordre de 2^{64} opérations ont déjà été menées à bien en pratique par des équipes de recherche ayant accès à des ressources calculatoires importantes. D'un autre côté, la consommation en mémoire de l'attaque B (8Po) tue dans l'œuf toute velléité de réalisation pratique. Il est évident, pour quiconque a un minimum

d'expérience, que l'attaque B va être plus difficile à exécuter que l'attaque A. Même si elle est « moins rapide », l'attaque A est concrètement « plus dangereuse », car elle, au moins, peut potentiellement être menée à bien si n est suffisamment faible.

Cet exemple peut sembler abstrait, mais il existe dans la réalité, dans le contexte de la résolution de systèmes polynomiaux modulo 2 :

A : Recherche exhaustive
B : *Beating Brute Force for Systems of Polynomial Equations over Finite Fields* [153]

A a déjà été implanté (notamment sur des FPGAs par Cheng, Chou, Nierderhagen, Yang et moi-même [42]) et exécuté pour $n = 64$. De son côté, B n'a jamais été implanté et ne le sera sûrement jamais, car cela n'aurait rigoureusement aucun intérêt. Sur des machines existantes, la quantité de mémoire disponible rend impossible de dépasser $n \approx 45$. La théorie favorise l'attaque B mais la pratique favorise l'attaque A. On pourrait citer de nombreux autres exemples.

Cette situation pose question : est-ce un problème que la théorie usuellement pratiquée soit très déconnectée de la réalité ? On a déjà répondu par la négative à la fin de la section 1.5. Cependant, les contradictions qu'on peut observer en passant de la théorie à la pratique ont un côté frustrant. Ne pourrait-on pas malgré tout disposer d'un modèle théorique plus proche de la réalité matérielle ?

Au passage, si on rajoute dans la balance une attaque intermédiaire, qui nécessite $2^{5n/6}$ opérations et une mémoire de taille $2^{n/2}$, alors le problème devient encore plus compliqué.

L'approche « théorique-prudente » est par ailleurs davantage problématique lorsque des schémas cryptographique ont une marge de sécurité faible, et où des attaques peuvent donc être proches de la faisabilité pratique. On pourrait bien sûr objecter que cette situation n'est pas souhaitable, cependant elle peut exister pour des raisons de coût. Les cartes bleues contiennent des clefs RSA de taille moyenne (la mienne a une clef de 1152 bits) ; leur taille est choisie pour qu'elles ne puissent pas être factorisées en pratique, mais le « niveau de sécurité » offert est bien inférieur à celui d'une clef de l'AES-128. On peut également s'attendre à ce que des dispositifs de chiffrement déployés sur des objets à bas coût aient une marge de sécurité plus faible.

Dans toutes ces situations, décider si une attaque décrite sur le papier est une menace concrète ou pas nécessite alors une maîtrise de l'art de la programmation des ordinateurs, en particulier des grosses machines parallèles. Cela nécessite d'être capable de faire la différence entre des attaques « pratiques » et des attaques « théoriques ». Essayer d'implanter des attaques cryptographiques permet dans une certaine mesure d'apprendre cela.

Dans cette perspective, la conclusion à laquelle je suis arrivé est que, même si cela n'épuise pas le sujet, raisonner sur le *coût* des attaques cryptographiques est bien plus efficace que de raisonner sur leur complexité en nombre d'opérations. C'est plus réaliste, mais surtout : a) cela donne une meilleure compréhension de ce qui se passerait si on essayait de programmer l'attaque et b) le coût est mieux corrélé avec la difficulté concrète d'exécuter un calcul donné.

La notion du coût des calculs est détaillée au chapitre 3. Le chapitre 4 discute de quelques algorithmes qui jouent un rôle dans la cryptanalyse (jointure, décodage des codes linéaires, résolution de systèmes polynomiaux, 3XOR). Le chapitre 5 discute de certaines attaques génériques dans le modèle du coût. Du point de vue de la cryptographie, la substantifique mœlle de ce document se situe dans les sections 4.5, 4.6 et dans tout le chapitre 5.

Chapitre 2

Modèles de calcul classiques

> The theory of computation is valid over a synthetic domain : its formal models have relevance only if they correspond to possible computational systems.
>
> C. D. Thompson, 1979 [186]

L'étude rigoureuse, « scientifique » des algorithmes nécessite un cadre formel dans lequel leur comportement peut être décrit de manière quantitative. Historiquement, la *machine de Turing* et le *lambda-calcul* ont été les premiers modèles formels de l'exécution d'un programme par une machine. Ils sont toujours largement utilisés dans le cadre de la théorie de la complexité et de l'étude des langages fonctionnels, respectivement. Cependant, ils sont très éloigné de la réalité matérielle d'un ordinateur contemporain, et à ce titre ils ne sont pas très adaptés pour décrire le comportement concret d'un algorithme.

Le modèle de calcul idéal doit être simultanément le plus simple possible et suffisamment fidèle pour que le comportement « théorique » d'un algorithme dans le modèle ait quelque chose à voir avec le comportement « pratique » de son implantation sur une vraie machine. Par exemple, on en attend que des opérations arithmétiques sur de petits entiers (64 bits) s'effectuent en temps constant. Par contre on s'attend à ce qu'effectuer des opérations arithmétiques sur des entiers arbitrairement grands ne prenne pas un temps constant, car ce n'est le cas sur aucune machine réelle.

Concrètement, l'étude des algorithmes et de leur complexité se déroule généralement dans le modèle de la *Random Access Machine* (RAM) — ou dans une de ses variantes. Celui-ci a été introduit par Cook et Reckhow [75] en 1972–73 pour modéliser une architecture de Von Neumann.

De manière informelle, il s'agit d'un processeur séquentiel relié à une mémoire arbitrairement grande. Lire et écrire la mémoire est une « opération élémentaire », tout comme effectuer des opérations arithmétiques ou des comparaisons sur des variables de type entier/flottant. L'étude des algorithmes revient alors peu ou prou à déterminer des bornes suffisamment précises sur le nombre d'opérations élémentaires effectuées, ou bien compter combien de fois certaines opérations particulières sont effectuées (lorsqu'on étudie les algorithmes de tri par comparaison, on compte souvent seulement les comparaisons).

Maintenant, commençons à couper les cheveux en quatre. Donner un modèle formel de la Random Access Machine expose des problèmes « amusants ». Pour cela, ouvrons le *Handbook of Theoretical Computer Science* et lisons le chapitre écrit par Van Emde Boas [192].

Une *Random Access Machine* possède un nombre fini d'états qui encodent son « programme » (comme les machines de Turing), un ou plusieurs registres, un pointeur d'instruction et une séquence infinie $\mathtt{M}[0], \mathtt{M}[1], \ldots$ de cases mémoire. Les registres et les cases mémoire sont de taille *non-bornée*. Les instructions exécutables par la machine incluent :

1. Flot de contrôle : **halt**, **goto**, **if** condition **then goto**. Une condition est un test de la forme `registre` $= 0$ ou `registre` ≥ 0.
2. Entrée/Sortie : **read** `registre` et **print** `registre`
3. Transfert de données : `registre` $\leftarrow i$, `registre` $\leftarrow M[$`registre`$]$, $M[$`registre`$] \leftarrow$ `registre`.
4. Opérations booléennes et arithmétiques entre registres.

Le fait que la taille de la mémoire ne soit pas bornée semble nécessaire pour pouvoir traiter des problèmes arbitrairement gros et donc parler de complexité asymptotique. Rappelons que les machines de Turing elles aussi ont un ruban de taille non-bornée. Seule une quantité finie de mémoire est utilisée dans un calcul donné, et on peut donc supposer qu'une valeur spéciale (« blanc ») est écrite partout où la mémoire n'a pas été utilisée.

Dans le cas des machines RAM, les accès indirects à la mémoire (`registre` $\leftarrow M[$`registre`$]$) nécessitent que les registres puissent contenir des entiers arbitrairement grand, sinon il serait impossible d'accéder à toute la mémoire.

Les problèmes commencent lorsqu'on doit choisir quelles opérations arithmétiques entre registres sont autorisées. Si on autorise la multiplication (`r1` $\leftarrow$ `r2` $\times$ `r3`), alors le modèle formel obtenu permet l'exécution *en temps linéaire* de l'algorithme dû à Shamir [178] qui factorise de grands entiers. Ceci n'est clairement pas raisonnable car aucune machine réelle n'en est capable.

A contrario, Papadimitriou démontre dans son livre *Computational Complexity* [167] que si une *Random Access Machine* qui peut faire l'addition, la soustraction, la division par deux et la comparaison à zéro (sur des entiers arbitrairement grands), calcule une certaine fonction en temps $T(n)$, alors une machine de Turing à 7 bandes peut calculer la même fonction en $T(n)^3$ opérations. Ceci exclut donc la factorisation des grands entiers en temps polynomial.

Il faut donc introduire des restrictions : ne pas autoriser la multiplication ; autoriser les opérations arithmétiques seulement si l'un des opérandes est inférieur à 2^{64} ; autoriser uniquement l'incrémentation ; etc. Aucune de ces solutions n'est complètement satisfaisante, car aucune ne modélise correctement l'intuition que les programmeurs ont des machines concrètes auxquelles ils ont affaire.

On pourrait aussi changer la mesure de complexité pour contourner le problème : compter uniquement les comparaisons, les opérations de groupe, les invocations d'une permutation publique, etc. Le risque étant alors d'obtenir un résultat qui a un rapport plus lointain avec le comportement d'une véritable implantation. Knuth, dans le tome 4 de *The Art of Computer Programming*, compte le nombre d'accès à la mémoire pour comparer l'efficacité pratique d'algorithmes combinatoires sur des instances données, de manière à la fois réaliste et indépendante du matériel.

Alors, dans quel modèle étudier les algorithmes ? Et que diraient nos étudiants s'ils savaient qu'on leur ment ou qu'on les induit en erreur dès la 2ème année de licence ? Plus sérieusement, comment est-ce que les auteurs « classiques » se sortent de ce problème ?

Knuth, dans *The Art of Computer Programming* [135], décrit dans les détails une architecture matérielle réaliste mais fictive nommée MMIX, qui idéalise des processeurs RISC contemporains. Il s'agit d'une machine qui possède des registres de 64 bits. Les opérations arithmétiques, y compris la multiplication, s'exécutent donc en temps constant entre registres. Mais par la force des choses, sa mémoire est limitée à 2^{64} octets. Knuth ignore le problème et donne joyeusement des complexités asymptotiques. Cette option est très satisfaisante pour étudier des problèmes *pratiques*. En effet, aucune machine ne possède, à ce jour, plus de 2^{64} octets de mémoire. L'inconvénient théorique de cette solution est que, comme la machine est de taille finie, il existe une borne supérieure sur le temps d'exécution de n'importe quel programme (qui termine) s'exécutant dans cette machine fictive. Les complexités sont donc toutes $\mathcal{O}(1)$, dans le fond, et la machine n'est pas adaptée à des raisonnements asymptotiques. Pas question de définit la sécurité par « l'absence d'attaque en temps polynomial », par exemple. Ça n'a en effet pas de sens

Cormen, Leiserson, Rivest et Stein, dans *Introduction to Algorithms* [76], décrivent un modèle comparable à la *Random Access Machine*, mais où les registres et les cases de la mémoire ne peuvent contenir

que $c \cdot \log n$ bits, où $c > 0$ est une constante et n est la taille de l'entrée de la machine. Les opérations arithmétiques sont toutes « élémentaires », donc de coût constant. Il s'agit donc de renoncer à l'idée d'une machine universelle, mais d'adapter la taille de la machine à la taille de l'instance qu'on veut traiter. Il semble que cette idée ait été introduite par Fredman et Willard dans [110] puis dans [111], où elle explicitement nommée : c'est le « *Transdichotomous*[1] *model* ». Le deuxième article cité, qui traite d'algorithmes pour les graphes, précise :

> Examining the criteria for evaluating such an algorithm reveals in particular the assumption that arithmetic operations can be performed in constant time for operands of size commensurate with that of the individual input values (e.g., edge weights). Not allowed, however, are computations that achieve hidden parallelism by doing operations on "long words", conforming to the reality that computers have fixed, bounded word length. We take as our model of computation the random access machine with word size b, where b is assumed to be only large enough to hold the edge weights (assumed to be integers) and also the number of vertices n of the input graph. (Thus we assume that $b \geq \log n$.) We allow the normal arithmetic operations as well as bitwise Boolean operations. We assume that when computing the product of two b-bit numbers the result consists of two words, one word consisting of the least significant b bits and a second word consisting of the most significant b bits. We refer to the second word as the significant portion of the product. We measure space in terms of words of memory required, and time in terms of machine operations on words.

Ce point de vue semble être adopté, de manière implicite, par une grande partie de la littérature cryptographique ; en tout cas, c'est le cas de toutes les publications traitant du paradoxe des anniversaires généralisé et de presque toutes celles qui traitent d'attaques sur des systèmes de chiffrement symétriques : lorsqu'il s'agit de trouver des « collisions » sur des fonctions aléatoires de n bits, on suppose que manipuler des blocs de n bits est une opération élémentaire, et que la mémoire est adressable par des pointeurs de n bits.

Il existe une extension parallèle du modèle, la « *Parallel Random Access Machine (PRAM*[2]*)* ». Dans cette machine, un nombre arbitrairement élevé de processeurs opèrent de manière synchrone, et ont tous accès à une mémoire partagée de taille non-bornée. Tous les processeurs exécutent le même programme, mais chaque processeur possède un identifiant entier distinct (son *rang*). Plusieurs variantes de PRAM existent là aussi, avec des comportements différents lors d'accès conflictuels à la mémoire (deux processeurs accèdent à la même case mémoire au même moment, et l'un des deux accès au moins est une écriture). Le plus couramment accepté, la PRAM CREW (« *Concurrent Read / Exclusive Write* ») interdit purement et simplement les accès conflictuels à la mémoire.

2.1 Critique du modèle de la *Random Access Machine*

Ce modèle de la *Random Access Machine* est simple et pratique. Il est raisonnablement proche de la façon dont un (petit) ordinateur séquentiel se présente à un programmeur, tout en faisant abstraction des complications pénibles (présence de caches, etc.). L'idée générale, c'est que le *temps d'exécution* est approximativement proportionnel au *nombre d'opérations élémentaires dans le modèle*. Le modèle est universellement enseigné (souvent implicitement) dans tous les cursus d'informatique de la planète. Tous les informaticiens ont été éduqués à l'algorithmique dans l'idée qu'il faut optimiser la complexité dans ce modèle, c'est-à-dire réduire le nombre d'opérations.

Au cours de mes travaux de recherche, je me suis forgé l'opinion suivante : le modèle de la *Random Access Machine* est bien adapté à l'étude d'algorithmes *de faible complexité*, mais cesse d'être utile pour discuter de *gros* calculs. C'est un peu comme faire l'hypothèse que la terre est plate : cela simplifie

1. « [...] because the dichotomy between the machine model and the problem size is crossed in a reasonable manner. » [18]

2. Il me semble que l'acronyme PRAM est surchargé et possède des significations variables selon les auteurs. La description qui suit est tirée de [123], mais [151] décrit quelque chose de légèrement différent.

les raisonnements et c'est tout à fait valable à petite échelle, mais ce n'est pas un modèle fiable si on envisage des voyages intercontinentaux.

Voici une liste non exhaustive des problèmes que pose le modèle de la *Random-Access Machine* pour décrire de gros calculs, et en particulier des attaques cryptographiques :

- Le modèle de calcul est séquentiel. Les attaques cryptographiques qui sont vraiment implantées le sont sur des machines parallèles.
- Depuis le début des années 1980, les machines qui ont une très grosse mémoire ne possèdent jamais un seul processeur, car cela serait extraordinairement inefficace. De telles machines (un processeur + grosse mémoire) n'existent pas dans la réalité.
- Le modèle de la *Parallèle Random-Access Machine* est tout aussi irréaliste : les grandes machines parallèles ne sont pas à mémoire partagée. Les nœuds de calculs possèdent leur propre mémoire et communiquent entre eux via un réseau. Des modèles de calculs plus réalistes prennent en compte la topologie de ce réseau, ainsi que la durée et la latence des communications.
- Dans tous les cas, l'hypothèse qu'une mémoire arbitrairement grande peut être accédée en temps constant semble s'opposer au fait que l'information se déplace à une vitesse finie. La latence des accès à la mémoire ne peut qu'augmenter avec la taille de celle-ci, or le modèle l'ignore.

Dans le fond, ces critiques pourraient se résumer en disant : « le modèle manque de réalisme ». Les problèmes tournent tous plus ou moins autour des accès à la mémoire, qui sont supposés de coût constant alors que c'est empiriquement faux et incompatible avec les lois connues de la physique.

Lorsque des gros calculs doivent être effectués en pratique, le modèle RAM prédit assez mal le comportement pratique d'une implantation concrète. Le « nombre d'opérations élémentaires » dans le modèle RAM ou PRAM ne donne qu'une *borne inférieure* sur la difficulté réelle d'exécution du calcul, car le modèle néglige trop de problèmes concrets (absence de mémoire partagée, complexité de communication, etc.)

La prévalence du modèle de la *Random Access Machine* engendre un autre problème, peut-être plus subtil. Dans la mesure où le modèle postule qu'effectuer une opération arithmétique et lire une case en mémoire sont des opérations de coût équivalent, il n'y a pas de raison spéciale de privilégier la première par rapport à la seconde. Comme la complexité dans le modèle RAM est mesurée en nombre d'opérations, les informaticiens sont poussés à concevoir des algorithmes (séquentiels) qui font le moins d'opérations possibles, même si pour cela il doivent occuper beaucoup de mémoire. Cela tend à produire des algorithmes « théoriques » qui ne peuvent pas être implantés de manière réaliste, ni exécutés pour résoudre des instances concrètes des problèmes concernés.

2.2 Confrontation à la réalité #1 : machines existantes

L'un des reproches adressés au modèle de la *Random Access Machine* dans la section précédente est qu'il décrit une abstraction d'un petit ordinateur séquentiel, sur lequel l'exécution de gros calculs tels que des attaques cryptographiques est impossible. À quel point les machines « haute-performance » ressemblent-elles au modèle ?

Le tableau 2.1 montre les machines les plus puissantes du monde sur les 30 dernières années. Une première constatation est que la machine la plus puissante du monde lors de la rédaction de ce manuscrit, `fugaku`, possède 5 peta-octets de mémoire (environ 2^{55} bits), mais que cette mémoire est distribuée entre 150 000+ nœuds qui n'ont que 32Go chacun. De manière générale, la quantité de mémoire disponible sur un seul nœud de calcul est toujours relativement limitée. Obtenir une puissance de calcul « de classe mondiale » nécessite un très grand niveau de parallélisme, et chaque processeur n'a que peu de mémoire.

Il existe certes des machines qui ont 2 voir 4 tera-octets de RAM, mais leur puissance est sensiblement plus limitée. L'hypothèse d'une très grande mémoire partagée monolithique est donc clairement déraisonnable d'un point de vue pratique. En tout état de cause, les programmeurs doivent affronter le fait que la mémoire est distribuée.

Machine	Année	nodes	M/node	M	$R_{\max}$	$R_{\max}/M$
Fugaku	2020	158 976	32G	5 087T	430P	84.53
Summit	2018	4 608	608G	2 802T	150P	53.54
Sunway TaihuLight	2016	40 960	32G	1 310T	93P	70.96
Tiahne-2A	2013	16 000	88G	2 277T	62P	26.98
Titan	2012	18 688	38G	710T	18P	24.77
Sequoia	2012	98 304	16G	1 573T	17P	10.92
K computer	2011	82 944	16G	1 410T	11P	7.45
Tiahne-1A	2010	7 168	32G	229T	2 566T	11.19
Jaguar	2009	18 688	16G	598T	1 941T	3.25
Roadrunner	2008	3 240	32G	104T	1 105T	10.66
BlueGene/L	2004	106 496	512M	74T	478T	6.48
Earth Simulator	2002	640	16G	10T	39T	3.76
ASCI White	2000	512	16G	6 144G	7226G	1.19
ASCI Red	1997	4 649	256M	1 292G	2379G	1.84
CP-PACS/2048	1996	2 048	64M	128G	368G	2.88
Numerical Wind Tunnel	1993	166	25M	42G	124G	2.99
CM-5	1993	1 024	3M	32G	60G	1.87

FIGURE 2.1 – Machines occupant la première place au classement TOP500 des machines les plus puissantes du monde. On indique l'année d'arrivée à la tête du classement et le nombre de nœuds de la machine parallèle. M désigne la quantité de mémoire de la machine (en octets), et $R_{\max}$ désigne sa performance de crête (en FLOPS) lors de l'exécution du traditionnel benchmark `linpack`.

Une deuxième constatation est que la puissance de calcul augmente plus vite que la quantité totale de mémoire disponible ($R_{\max}/M$ semble croître). Ceci va un peu à rebours de la tendance au « compromis temps-mémoire », qui vise à utiliser plus de mémoire pour réduire le temps.

2.3 Confrontation à la réalité #2 : coût des accès à la mémoire

> [...] we estimate the cost of each access to a bit within N bits of memory as the cost of $\sqrt{N}/2^5$ "bit operations".
>
> The NTRU Prime Team, NTRU Prime : NIST Round 3 submission, 2020.

L'objection « on ne peut pas accéder à une mémoire arbitrairement grande en temps constant à cause de la vitesse finie de la lumière », qui figure section 2.1 est-elle de nature purement théorique ou bien a-t-elle une quelconque valeur pratique ?

Dans les processeurs récents, par exemple les AMD EPYC 7742 (« Rome »), les cœurs peuvent être distants de 6cm (cf. Fig 2.2). Cela signifie que, dans le meilleur des cas, l'information mettrait 0.3ns pour parcourir cette distance (à 200 000km/s, soit la vitesse de propagation des ondes électromagnétiques dans le cuivre, ce qui est *très* optimiste. En réalité ça doit prendre bien plus longtemps car il faut traverser aussi des portes logiques). À 3.4Ghz, la fréquence maximum de la puce, la durée d'un cycle d'horloge est de 0.29ns. Les délais de propagation de l'information ne sont donc pas négligeables. Atteindre une barrette de RAM située hors du CPU a donc un coût mesurable. Plus cette mémoire est loin, plus ce coût est important. Pour cette raison, des cœurs distincts qui partagent la même mémoire n'ont pas forcément en permanence une vue cohérente de son état (il serait trop coûteux de maintenir la consistance séquentielle), et les processeurs prévoient des instructions de *barrière mémoire* pour les synchroniser explicitement.

Il est bien connu dans la communauté du calcul haute-performance que la *latence* des accès à la mémoire augmente rapidement quand la mémoire à laquelle on accède est de plus en plus éloignée des

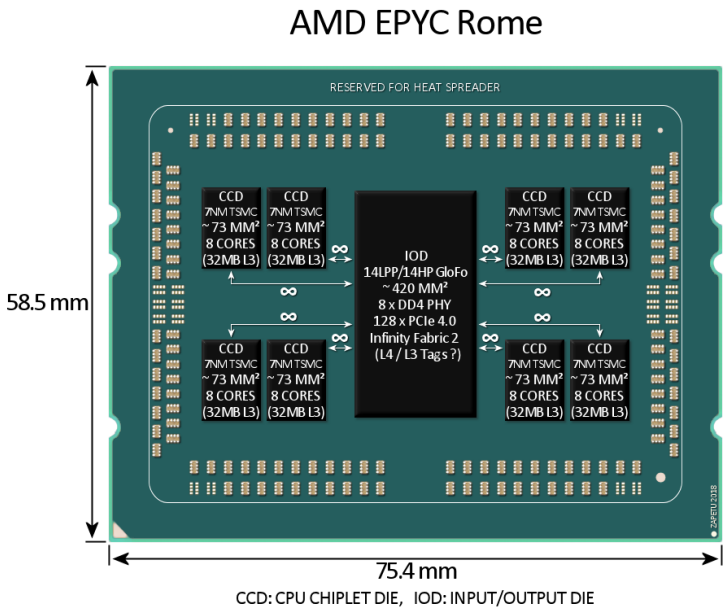


FIGURE 2.2 – Un processeur AMD EPYC Rome, taille réelle. Il y a 8 « chiplets » de 8 cœurs chacun, physiquement séparés de plusieurs centimètres.

unités d'exécution. Ceci est facile à vérifier expérimentalement. On alloue un tableau T de taille N, et on l'initialise préalablement avec une permutation pseudo-aléatoire des entiers $\{0, 1, \ldots, N-1\}$ [3]. Ensuite, on mesure le temps d'exécution du bout de code suivant :

```
int x = 0;
for (int i = 1000000000; i != 0; i--)
        x = T[x];
```

Dans le modèle de la *Random Access Machine*, son temps d'exécution est censé être constant, alors qu'on observe en pratique qu'il dépend de N à cause des « détails pratiques » de l'architecture des ordinateurs. Ce micro-benchmark (le « *pointer-chasing* ») permet d'observer la latence des accès à la mémoire car l'exécution ne peut pas progresser tant que la prochaine valeur de x n'a pas été obtenue.

La mesure a lieu sur un nœud du cluster `grvingt` de Grid'5000, équipé de deux processeurs Xeon Gold 6130 (« Skylake ») et de 192Go de mémoire DDR4-2933. La figure 2.3 montre les résultats. La forme de ces courbes et la différence entre les deux mettent en évidence plusieurs phénomènes architecturaux dont on ne va pas discuter ici : fautes de cache, de TLB, taille des pages, effets NUMA, etc.. Il faut noter que les deux courbes montrent le temps nécessaire à l'exécution du même petit bout de code ci-dessus. Obtenir la meilleure courbe nécessite de comprendre le fonctionnement du matériel et du système d'exploitation (Linux dans ce cas) : utiliser `malloc` pour allouer la mémoire puis lancer le programme normalement donne la courbe « débutant ». Pour obtenir la courbe « expert », il faut :

a) Forcer le noyau à préparer des *huge pages* avant le lancement du programme ; en tant que `root`, exécuter

   ```
   echo 70000 > /proc/sys/vm/nr_hugepages
   ```

   Il n'y a pas de moyen fiable d'obtenir des *huge pages* sans être `root`. En particulier, `madvise` avec le drapeau `MADV_HUGEPAGE` ne fonctionne pas forcément, ou pas sur toute la zone de la mémoire concernée.

b) Allouer la mémoire avec

   ```
   int flags = MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB | MAP_LOCKED;
   T = mmap(NULL, allocsize, PROT_READ | PROT_WRITE, flags, -1, 0);
   ```

3. On vérifie que le cycle de la permutation qui commence en zéro est au moins de taille $N/3$, sinon on change la permutation. Ça n'arrive pas très souvent : la longueur moyenne du cycle auquel appartient un élément aléatoire dans une permutation aléatoire de N éléments est $(N+1)/2$ (cf. [135, §1.3.3, exercice 24]).

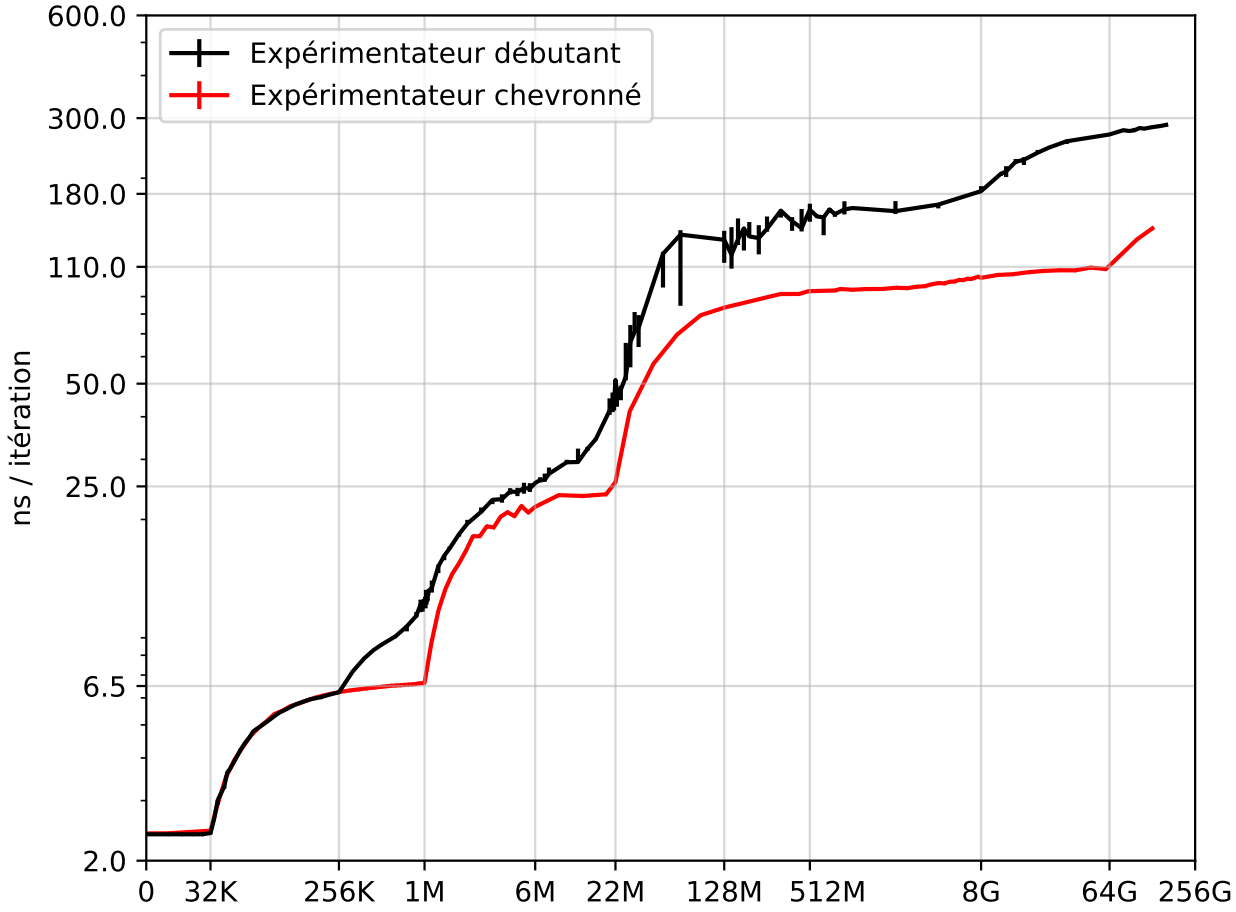


FIGURE 2.3 – Résultat du *pointer chasing* benchmark sur un nœud du cluster `grvingt` de Grid'5000. Le processeur a 32Ko / 1Mo / 22Mo de cache L1/L2/L3. Le TLB couvre 64/32 pages de 4Ko/2Mo au premier niveau et 1536 pages au 2ème niveau. Avec des pages de 4Ko, on a donc une faute de TLB1 à partir de 256Ko et un *page walk* par itération à partir de 6Mo. Avec des pages de 2Mo, cela monte à 64Mo / 3Go. Au-delà de 96Go, il y a des effets NUMA.

c) Forcer l'allocation de la mémoire sur la bonne partition NUMA en lançant le programme avec
`numactl --cpunodebind=0 --preferred=0 -- ./pointer_jumping`

Le résultat obtenu n'est pas spécifique à l'architecture de la machine utilisée pour les tests. On peut reproduire les résultats sur des processeurs qui ont des jeux d'instructions différents (des PowerPC et des ARM). La figure 2.4 l'illustre.

Le message à retenir est qu'aller lire aléatoirement dans une structure de données de quelques centaines de méga-octets est environ 100× plus lent que d'accéder à quelques kilo-octets de données en cache. Même à l'intérieur d'un seul nœud de calcul, plus un algorithme a besoin d'une grande quantité de mémoire, plus il risque de passer du temps à attendre ses données. De nombreuses attaques cryptographiques sont écrites en utilisant de manière libérales d'énormes tables de hachage, et en négligeant ces problèmes pratiques.

Un autre message intéressant à retenir est le suivant : à quelques exceptions près, évaluer un système de chiffrement par bloc sur 128 bits est plus rapide que de lire *un* mot de 64 bits situé à un emplacement aléatoire dans un tableau de 128Mo, ce qui est pourtant modeste. Voire les tableau 2.5 et 2.6.

Tout ceci est d'ailleurs bien connu (d'une partie) de la communauté cryptographique ! En effet, des fonctions *memory-bound* telles qu'Argon2 [33] ont été conçues dans le but exprès d'être *lentes* en

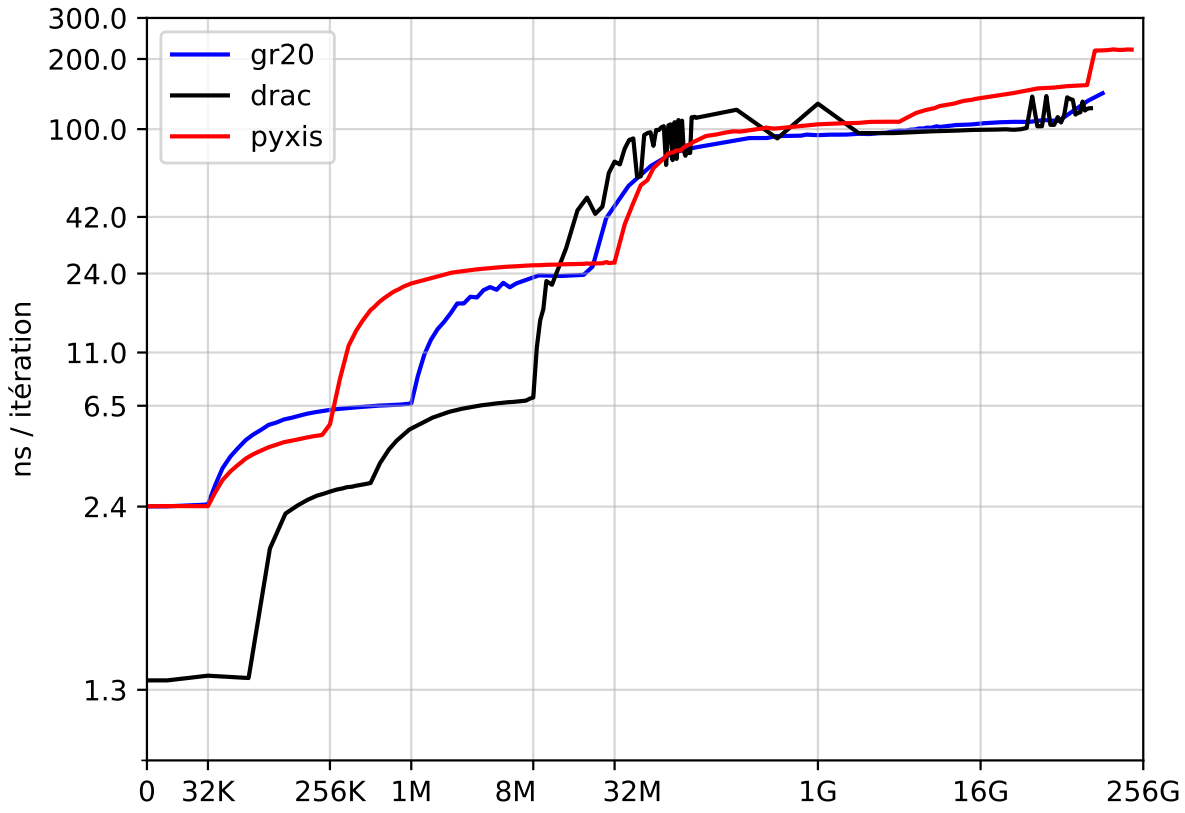


FIGURE 2.4 – Résultat du *pointer chasing* benchmark sur des nœuds des clusters `grvingt` (Intel Xeon Gold 6130), `drac` (IBM POWER8+) et `pyxis` (Cavium/ARM ThunderX2 99xx) de Grid'5000.

exploitant le fait que des accès aléatoires à la mémoire sont particulièrement coûteux. Ces fonctions peuvent être utilisées pour fournir des preuves de travail [96] et leur usage est recommandé pour hacher des mots de passe.

2.3.1 Quelques idées venant de la communauté du HPC

> This brief note points out something obvious — something the authors "knew" without really understanding. With apologies to those who did understand, we offer it to those others who, like us, missed the point.
>
> W. A. Wulf et S. A. McKee, 1994 [203]

Le fait que les accès à la mémoire sont en réalité très coûteux est un phénomène bien connu de la communauté du calcul haute-performance ; Wulf et McKee l'ont baptisé le « *Memory Wall* » en 1994 [203]. Le problème est par ailleurs considérablement aggravé quand il est nécessaire de passer sur plusieurs nœuds de calcul... justement pour bénéficier de plus de mémoire.

Au passage, la communauté du HPC a défini depuis longtemps l'« intensité arithmétique » ou l'« intensité opérationnelle » comme étant le nombre d'opérations arithmétiques (flottantes) effectuées par octet transféré depuis la mémoire ; le folklore affirme que plus cette dernière est faible, plus il est

Fonction	Latence (ns)	Implantation
$x \mapsto \mathsf{AES128}_0(x)$	11	Instructions `AES-NI`
$x \mapsto \mathsf{AES128}_x(0)$	42	Instructions `AES-NI`
$x \mapsto \mathsf{Speck128/128}_0(x)$	27	Code C de référence
$x \mapsto \mathsf{Speck128/128}_x(0)$	31	Code C de référence

FIGURE 2.5 – Latence de l'évaluation de certains systèmes de chiffrement par bloc sur un nœud de `grvingt` (Xeon Gold 6130 2.1GHz, *turbo boost* désactivé), avec ou sans la diversification de clef. En comparant avec la figure 2.3, on voit que les latences sont comparable avec celles entraînées par un accès au cache L3 après une faute de cache L2.

Fonction	machine	Latence (ns)
$x \mapsto \mathsf{AES128}_0(x)$	`grvingt` (x86-64)	44
	`drac` (POWER)	71
	`pyxis` (ARM)	108
$x \mapsto \mathsf{AES128}_x(0)$	`grvingt` (x86-64)	85
	`drac` (POWER)	139
	`pyxis` (ARM)	185
$x \mapsto \mathsf{Speck128/128}_0(x)$	`grvingt` (x86-64)	27
	`drac` (POWER)	45
	`pyxis` (ARM)	63
$x \mapsto \mathsf{Speck128/128}_x(0)$	`grvingt` (x86-64)	31
	`drac` (POWER)	51
	`pyxis` (ARM)	64

FIGURE 2.6 – Latence de l'évaluation de certains systèmes de chiffrement par bloc, avec ou sans la diversification de clef. Code C de référence.

difficile d'atteindre les performances maximales du matériel, car les unités de calcul vont passer leur temps... à attendre que les données arrivent. Le produit matrice-matrice (et donc la factorisation LU et la résolution de systèmes linéaires) a une intensité arithmétique de $\mathcal{O}(n)$. Correctement implantée, elle peut s'approcher assez près des performances de crête de grandes machines parallèles. La transformée de Fourier rapide (FFT) a une intensité arithmétique de $\mathcal{O}(\log n)$, plus faible. De fait, les codes de FFT ont des performances moindres, mesurées en nombre d'opérations arithmétiques flottantes par seconde (FLOPS), car le coût des accès à la mémoire est proportionnellement plus élevé.

Ces deux opérations font partie des *benchmarks* du HPC Challenge (HPCC) qui ont été utilisés pour mesurer les performances de grosses machines parallèles [154]. Par exemple, sur la machine `mira` (BlueGene/Q à 48 *racks*, donc 49 152 nœuds), la résolution d'un système dense $Ax = b$ peut se faire à 5.7 PFLOPS, mais la FFT à 226 TFLOPS seulement. Le maximum théorique est 10 PFLOPS, donc la FFT ne fonctionne qu'à 2.3% des performances maximales, pénalisée par le coût des accès à la mémoire.

Le HPC Challenge contient un autre *benchmark* intéressant, nommé `RandomUpdate`. Il s'agit d'allouer un tableau d'entiers de 64 bits de taille $N = 2^n$, où n est la plus grande valeur possible telle que N occupe moins de la moitié de la mémoire disponible sur la machine. Le tableau est initialisé avec un contenu quelconque, puis une séquence de $4N$ nombres aléatoires de 64 bits $x_0, x_1, \ldots$ est générée avec un LFSR. Pour chacun d'entre eux, le tableau est mis à jour avec $T[x_i \ggg 64-n] \leftarrow T[x_i \ggg 64-n] \oplus x_i$. Les performances sont mesurées en GUPS (*Giga Update Per Second*). Ceci ressemblerait assez fort à ce qui se passerait si on essayait de programmer un crible distribué, dans le contexte de la factorisation des grands entiers par exemple.

Pour simuler une mémoire partagée de la bonne taille, le tableau T est nécessairement réparti sur les nœuds de la machine, et les mises à jour du tableau sont transmises par le réseau. On peut calculer le

Machine	Année	CPU	1 nœud		Toute la machine			R	FLOP/GUP
			RAM	GUPS	# CPUs	RAM	GUPS		
BG/L	2005	PPC 440	512Mo	0.00652	65536	32To	33.0	12.95	7.8K
BG/P	2008	PPC 450	2Go	0.00969	32768	64To	103.1	3.08	1.7K
Jaguar	2009	Opteron 2435	16Go	0.03324	37376	292To	36.7	33.0	39K
Endeavour	2010	Xeon X5670	2Go	0.03537	4320	8.6To	10.8	14.2	4K
K	2012	Sparc VIIIfx	16Go	0.10707	82944	1296To	471.9	18.81	21K
BG/Q	2014	PPC A2	16Go	0.00745	49152	768To	417	0.88	13.6K

FIGURE 2.7 – Résultats du *benchmark* `RandomUpdate` pour plusieurs grosses machines parallèles qui caracolaient en tête du classement. La colonne R est calculée par $R := \mathrm{GUPS}_1/\mathrm{GUPS}_{\mathrm{all}} \times \#\mathrm{CPU}$.

ralentissement R (qui est l'inverse de l'*efficacité*) : c'est le ratio entre le rythme auquel un processeur effectue les mises à jour lorsqu'il est seul, ou bien quand il est accompagné de tous les autres. Le tableau 2.7 montre certains résultats. Il faut garder à l'esprit que ce sont des machines ultra-haut-de-gamme, avec des réseaux de communications spéciaux et particulièrement performants. Malgré tout, le coût des accès à la mémoire se dégrade fortement lorsqu'il faut passer par le réseau. Un accès mémoire est équivalent à plusieurs milliers ou dizaines de milliers d'opérations arithmétiques.

2.3.2 Conclusion

Pour conclure, tirer un trait d'égalité entre un accès à la mémoire (au coût très variable, potentiellement très élevé, surtout avec une grosse mémoire) et une opération arithmétique (au coût relativement constant), est une simplification grossière. Accéder à un gros volume de données n'est pas gratuit, et accéder à une grosse mémoire partagée entraîne des *coûts de communication cachés*, qui ne sont pas pris en compte par le modèle de la *Random Access Machine* — alors que ça l'est dans... les machines de Turing.

2.4 Tendance à la consommation excessive de mémoire

> The memory requirement is probably not an issue for an attacker that can spend time $2^{5n/6}$.
>
> G. Leurent et L. Wang, 2015 [150]

Le modèle de la *Random Access Machine* pousse les concepteurs d'algorithmes à la consommation de mémoire, surtout si cela leur permet de diminuer le nombre total d'opérations effectuées.

La tendance au « compromis temps-mémoire » est particulièrement frappante dans le cas des algorithmes de décodage des codes linéaires aléatoires, comme l'illustre le tableau 2.4. On constate que chaque nouvel algorithme prend moins longtemps mais occupe plus de mémoire que son prédécesseur. Tous ces algorithmes sont facilement parallélisables, car ils consistent à répéter une procédure probabiliste relativement simple tant qu'une solution n'a pas été trouvée, et ces répétitions peuvent être faites en parallèle. La section 4.5 discute plus en détail de ces algorithmes.

L'algorithme de Prange / Lee-Brickell est facile à implémenter. Une variante de l'algorithme de Stern a été utilisée en 2008 par Bernstein, Lange et Peters [25] pour casser le jeu de paramètres proposé initialement par McEliece en 1978, et censé offrir 64 bits de sécurité (un fait curieusement assez peu connu). Cela revient à décoder un code linéaire aléatoire de paramètres $[1024, 524, 101]$. À ma connaissance, aucun des autres algorithmes n'a été testé à grande échelle.

Les « praticiens » savent que les contraintes d'espace sont plus dures à gérer que les contraintes de temps — attendre que le calcul termine, c'est une chose ; mais s'il n'y a pas assez de mémoire pour qu'il démarre, c'en est une autre. Quand des chercheurs essayent d'implanter des attaques cryptographiques, ils sont souvent réduits à une autre sorte de « compromis temps-mémoire » : les algorithmes doivent parfois être modifiés (d'une manière qui les *ralentit*) pour réduire leur complexité en espace. L'implan-

Algorithme	Half-distance			McEliece		
	T	M	ρ	T	M	ρ
Lee-Brickell [142]	0.0575	0	0	0.0409	0	0
Stern [179]	0.0556	0.0135	0.24	0.0391	0.0123	0.31
Ball-collision [26]	0.0556	0.0148	0.27	?	?	?
MMT [156]	0.0536	0.0216	0.40	0.0376	0.0179	0.47
BJMM [17]	0.0493	0.0306	0.62	0.0341	0.0240	0.70
May-Ozerov [157]	0.0473	0.0346	0.73	0.0331	0.0251	0.76

TABLE 2.1 – La complexité de résoudre les problèmes de décodage dans le pire cas est $2^{nF(R)}$, où n est la longueur du code et son débit. Chaque case du tableau montre une borne supérieure sur $F(R)$. La colonne « Half-distance » correspond au fait de décoder (par syndrome) un mot de code ayant $(d-1)/2$ erreurs, où d est la distance minimale du code (donnée par la borne de Gilbert-Varshamov). La colonne « McEliece » correspond au problème du décodage au maximum de vraisemblance d'un code aléatoire de distance relative $D = 0.04$ et de débit $R = 0.75$, comme proposé dans [25]. Ce tableau est tiré de [17] ; la dernière colonne du tableau dans [17] est fausse (elle correspond à une distance relative deux fois trop grande). La complexité en mémoire de l'algorithme BJMM était également fausse. Tout ceci a été correctement recalculé pour ce document, tout comme la dernière ligne. Les complexités données sont des ordres de grandeur et ne sont pas à prendre au pied de la lettre.

tation de l'algorithme de Wagner par Bernstein, Lange, Niederhagen, Peters et Schwabe [24], dans le but d'attaquer des versions réduites de la fonction de hachage FSB, en est un exemple caractéristique.

Leurent et Wang décrivent dans [150] une attaque qui nécessite $2^{n/3}$ bits de mémoire et $2^{5n/6}$ opérations (dont on reparlera section 5.4.2). Au détour de leur article, ils écrivent la phrase citée au début de cette section et qui m'a beaucoup fait réfléchir :

> The memory requirement is probably not an issue for an attacker that can spend time $2^{5n/6}$.

Autrement dit, si une attaque cryptographique nécessite un temps T, alors ce n'est « probablement pas un problème » qu'elle consomme $\mathcal{O}\left(T^{0.4}\right)$ bits de mémoire. Dans une communication privée d'avril 2021, Gaudry affirme un point de vue comparable :

> Moi, ma théorie « rule-of-thumb », c'est que tant que la complexité en mémoire ne dépasse pas en gros la racine carrée de la complexité en temps, on va peut-être en baver pour programmer le truc, mais une fois qu'on s'est bien battu et qu'on a adapté les algos, le point bloquant pour faire tourner des calculs toujours plus gros sur des machines standards (cluster HPC), ça sera le temps. (je suis conscient de tous les biais et les incohérences de cette « définition »)

Ceci suggère de définir le « ratio temps-mémoire » d'une attaque cryptographique par $\rho = \log M / \log T$ — c'est en fait une sorte de version logarithmique de l'intensité arithmétique du monde du HPC. Dans la littérature, de nombreuses attaques cryptographiques ont un « ratio temps-mémoire » plus élevé que les seuils suggérés ci-dessus, et celui-ci est à mon avis un bon critère pour juger du caractère réaliste d'une attaque. Je propose donc de formuler la

Proposition 1 (« Loi du bon sens pratique »)**.** *Considérons une attaque cryptographique et notons ρ son ratio temps-mémoire.*

1. *(Version optimiste) Si $\rho \leq \frac{1}{2}$, alors il sera raisonnablement possible d'implanter l'attaque et de l'exécuter sur du matériel commun. Le calcul sera vraisemblablement* compute-bound*.*
2. *(Version pessimiste) Si $\rho \geq \frac{3}{4}$, alors il est quasiment certain que l'attaque ne sera pas exécutable en pratique, sauf pour des instances jouets.*
3. *(Version normande) Si $\frac{1}{2} \leq \rho \leq \frac{3}{4}$, alors une réflexion plus poussée est nécessaire pour déterminer le caractère pratique ou pas de l'attaque.*

Appliquons ce résultat aux algorithmes de décodage des codes aléatoires listés dans le tableau 2.4. L'algorithme de Stern, qui a donc vraiment été utilisé lors d'un calcul à grande échelle contre une instance réaliste du système McEliece, possède un ratio de 0.31. Le code de l'attaque n'est malheureusement pas disponible, mais l'article compagnon, qui discute abondamment de détails pratiques, suggère que la complexité en espace est de l'ordre de quelques Méga-octets.

Par contre, l'algorithme BJMM, qui a 10 ans, n'a, lui, jamais été utilisé ; il a un ratio de 0.62. Ce n'est pas pour rien qu'il n'est pas utilisé ; sa consommation en mémoire est certainement problématique. Ce simple calcul permet de rendre ce jugement avec un bon niveau de confiance.

Dinur a récemment publié une version améliorée de l'algorithme de Lokshtanov pour résoudre les systèmes polynomiaux dans un article intitulé « *Cryptanalytic Applications of the Polynomial Method for Solving Multivariate Equation Systems over GF(2)* » [88]. Sa complexité est de $T = n^2\ 2^{0.815n}$ opérations sur des bits, avec une mémoire de $M = n^2\ 2^{0.63n}$ bits. Il a donc un « ratio temps-mémoire » de $\rho = 0.77$, ce qui le classe d'après la proposition 1 dans la catégorie des algorithmes inutilisables. En effet, même si on disposait de `fugaku`, la machine la plus puissante du monde d'après le tableau 2.1, avec ses 2^{55} bits de mémoire, on ne pourrait exécuter l'algorithme que jusqu'à $n = 68$ (après la mémoire déborde). La machine coûte presque un milliard de dollars et consomme environ 30 MW d'électricité. Comme l'atteste le site des *Fukuoka MQ challenges*, une instance avec $n = 66$ (type IV) a été résolue en 2015 par une recherche exhaustive implantée sur FPGA [42]. Le calcul a été mené à bien par Chou, Niederhagen, et Yang en 93 heures de calcul et avec des moyens qui n'ont rien à voir (128 FGPAs de type Spartan-6). Il est tout à fait raisonnable de penser que $n = 68$ est faisable par la recherche exhaustive, en quinze jours avec le même matériel, qui coûte le prix d'une voiture. Il faut noter que l'algorithme de Dinur nécessite plus de 2^n opérations lorsque $n < 66$. Au-delà, il nécessite au moins 1.5Po de mémoire ; mais heureusement, il est « *concretely efficient* »— c'est l'abstract de [88] qui le dit !

Le problème 3XOR peut être formulé de la façon suivante[4] : étant donné trois fonctions aléatoires $f, g, h : \{0,1\}^n \to \{0,1\}^n$, trouver un triplet (x, y, z) tel que $f(x) \oplus g(y) \oplus h(z) = 0$. Ceci peut être effectué de différentes manières, mais une possibilité particulièrement intéressante est la suivante : fixer $z = 0$, poser $f'(x) = f(x) \oplus h(0)$, et chercher une collision $f'(x) = g(y)$. Ceci peut se faire sans mémoire et en temps $\approx 2^{n/2}$ par une variante des algorithmes de recherche de cycle habituels (cf. section 4.2). Joux, dans son ouvrage « *Algorithmic cryptanalysis* » [125], en présente un autre qui résout le problème en temps et en espace $2^{n/2}/\sqrt{n}$. Passer d'un algorithme pratique, raisonnablement implantable et parallélisable, à un algorithme qui ne peut pas être exécuté en pratique à cause de sa consommation mémoire trop élevée est présenté par Joux comme une « amélioration incrémentale » [125, §8.3.3.1]. Avec $n = 96$, le calcul est réalisable même avec des moyens de calculs modestes [44]... à condition de ne *pas* utiliser les algorithmes « améliorés » qui consomment $2^{n/2}$ unités de mémoire.

2.5 Des algorithmes uniquement séquentiels

> There is no concensus among researchers on a model that takes memory complexity into account.
>
> I. Dinur, 2021 [88]

Les algorithmes décrits dans le modèle de la *Random Access Machine* sont nécessairement séquentiels. Il sont parfois trivialement parallélisables (recherche exhaustive), mais ce n'est pas systématiquement le cas. Je me suis également forgé la conviction que cette focalisation exclusive sur le nombre d'opérations dans un modèle de calcul séquentiel conduit à des aberrations d'un point de vue pratique.

Je vais tâcher d'illustrer cette opinion sur un exemple. En 2017, Lokshtanov, Paturi, Tamaki, Williams et Yu ont publié à SODA (l'une des principales conférences d'algorithmique) un article intitulé « *Beating Brute Force for Systems of Polynomial Equations over Finite Fields* » [153] dont on a déjà discuté.

4. La notation $\oplus$ désigne le XOR.

Le titre fait allusion au fait que la complexité asymptotique de l'algorithme est de $\mathcal{O}\left(2^{0.8765n}\right)$. Ce travail est remarquable, car c'est le premier algorithme capable de résoudre des systèmes quadratiques sur $\mathbb{F}_2$ avec complexité inférieure à 2^n *dans le pire des cas*, sans nécessiter d'hypothèse algébrique supplémentaire sur les polynômes de départ — contrairement à d'autres travaux antérieurs, notamment l'algorithme `BooleanSolve` de Bardet, Faugère, Salvy et Spaenlehauer [16] dont on a déjà parlé aussi.

Ce n'est pas indiqué dans l'article, mais la complexité en mémoire de l'algorithme de Lokshtanov *et al.* est asymptotiquement identique à sa complexité en temps. Il est significatif qu'une partie importante des informaticiens en général et des cryptologues en particulier considère qu'un algorithme séquentiel dont la complexité temporelle et spatiale est $\mathcal{O}\left(2^{0.875n}\right)$ est « meilleur » qu'une recherche exhaustive aisément parallélisable nécessitant 2^n opérations et zéro mémoire. Il semble pourtant difficile d'échapper à l'argument suivant [5] :

- L'exécution de l'algorithme nécessite une machine de taille $2^{0.875n}$ (à cause de sa consommation en mémoire).
- Si on pouvait construire une machine de cette taille, alors on pourrait aussi construire une machine de la même taille capable de faire la recherche exhaustive en parallèle.
- Sur cette machine alternative, l'instance du problème serait résolue en $2^{0.125n}$ opérations parallèles par la recherche exhaustive.
- Par conséquent, pour « battre la force brute » en pratique, il faudrait que l'algorithme de départ puisse être parallélisé de telle sorte à s'exécuter en moins de $2^{0.125n}$ opérations parallèles sur une machine de taille $2^{0.875n}$.

Est-ce le cas ? La question n'a fait l'objet d'aucun travail de recherche (publié). C'est très improbable à première vue, et dans le fond ce n'est pas le problème. En effet, une machine de taille $2^{0.875n}$, disposée dans l'espace, contient forcément des composants à distance $\Omega\left(2^{0.291n}\right)$ les uns des autres, et en particulier de l'utilisateur qui doit récupérer la solution à la fin. Le temps que va mettre l'information à se propager à travers la machine semble donc minoré par cette borne. Au contraire, une machine de taille $2^{0.75n}$ (plus petite, donc), est capable de faire la recherche exhaustive *et de communiquer le résultat* en temps $\mathcal{O}\left(2^{0.25n}\right)$, ce qui est meilleur.

Que vaut l'argument « on pourrait remplacer des circuits qui stockent des bits par des circuits qui calculent » ? Une puce de mémoire HBM2 (haut de gamme, standardisée en 2016) de 8Go occupe 7.75 mm × 11.87 mm, soit environ 92 mm^2. Par comparaison, en 2012, Gürkaynak *et al.* [119] ont produit un ASIC qui peut calculer la fonction de hachage Keccak[512] à 2.75Go/s. Le circuit occupe 0.067mm^2, avec une technologie nettement inférieure (65 nm, pas d'empilement 3D). Par conséquent, avec la surface nécessaire à 1Go de RAM moderne, on peut avoir un circuit qui évalue la permutation interne de Keccak au moins 3.5 milliards de fois par seconde (3.7 To/s).

Pour la préparation de ce manuscrit, j'ai demandé à Dinur ce qu'il pensait du raisonnement précédent (« est-ce vraiment meilleur que la recherche exhaustive ? ») appliqué à son algorithme récent [88] dont il est question dans les sections 1.7 et 2.4 de ce manuscrit. Sa réponse a été essentiellement la suivante :

> Les *security claims* des constructions cryptographiques sont souvent formulées en disant qu'un certain nombre (très élevé) d'opérations est nécessaire pour casser la sécurité. Par conséquent, un algorithme qui fait moins d'opérations, y compris dans un modèle séquentiel irréaliste, contredit les prétentions des concepteurs et constitue alors un résultat scientifique valide, indépendamment du fait qu'il ait une quelconque valeur pratique.
>
> [...]
>
> Formuler des *security claims* dans un modèle simple (qui ne prend pas en compte la mémoire/le parallélisme) a l'avantage que cela permet une analyse de sécurité plus simple et plus propre.

D'autre part, l'article en question [88] explique :

5. Si ma mémoire ne me trompe pas, je l'ai lu sous pour la première fois sous la plume de Dan Bernstein ; il figure noir sur blanc un article non publié de 2005 [20]. Il est aussi repris dans l'article « *Too Much Crypto* » d'Aumasson [12].

> [...] there is no concensus among researchers on a model that takes memory complexity into account, and the formal security claims of the Picnic (and LowMC) designers only involve time complexity.

Ce point de vue a d'ailleurs été le mien dans le passé. J'ai proposé durant ma thèse en 2010, dans un travail conjoint avec Derbez, Dunkelman, Keller, et Fouque [47] des attaques contre des versions réduites de l'AES. Ces attaques « *guess-and-determine* » ont la particularité de ne nécessiter aucune mémoire et d'être aisément parallélisables. L'article en question décrit en particulier une attaque contre 4 tours de l'AES avec seulement deux paires clair-chiffré choisies qui nécessite 2^{104} opérations et zéro mémoire. Un peu plus tard, en 2011, j'ai été très fier de publier avec Derbez et Fouque un autre article [48] qui « améliore » certaines de ces attaques. En particulier, ce nouvel article affirme également l'existence d'une attaque contre 4 tours de l'AES avec deux paires clair-chiffré choisies qui nécessiterait, elle, 2^{80} opérations et 2^{80} « mémoire » (sans qu'on sache forcément très bien à quoi ça correspond). L'article prétend que cette nouvelle attaque est supérieure à l'ancienne car elle nécessite moins d'opérations. Heureusement pour mon CV, les rapporteurs n'ont rien trouvé à y redire à l'époque. Le raisonnement exposé ci-dessus exigerait pourtant que l'on demande : « sur une machine de taille 2^{80}, il est possible d'exécuter l'ancienne attaque en temps 2^{24}. Est-il possible de paralléliser la nouvelle attaque pour qu'elle s'exécute plus vite, sachant qu'elle nécessite une machine de taille 2^{80} quoi qu'il arrive ? ». Il est difficile de répondre à cette question car... l'attaque en question n'a jamais été décrite ! On peut dire ce qu'on veut à propos des attaques qui n'existent « que sur le papier », mais celle-ci n'atteint même pas ce statut-là.

Chapitre 3

Modèle du « coût »

Plusieurs chercheurs ont proposé l'idée de s'intéresser au « *coût* » des calculs en général et des attaques cryptographiques en particulier, en tentant de proposer une alternative plus satisfaisante au modèle de la *Random Access Machine*. C'est notamment le cas de Bernstein, qui en discute dans plusieurs articles, dont par exemple [19, 23, 21]. C'est également le cas de Wiener, qui en fait le sujet principal de son article « *The Full Cost of Cryptanalytic Attacks* » [200].

Les chercheurs qui ont exécuté en pratique des attaques cryptographiques nécessitant beaucoup de calculs discutent généralement de leur coût concret. Les auteurs de la factorisation en 2009 d'une clef RSA de 768 bits [128] estiment que le calcul a nécessité une quantité d'énergie totale de 500MWh (ceci est une autre mesure de coût intéressante en pratique : quelqu'un doit bien payer la note !). Ceci est suffisant pour évaporer deux piscines olympiques à 20°C. Stevens, Bursztein, Karpman, Albertini et Markov ont produit en 2017 la première collision sur SHA-1 [180], un résultat pratique majeur de la cryptanalyse. Ils ont estimé qu'exécuter les calculs sur le *cloud* d'Amazon aurait coûté $560 000.

À partir de 2012, Kleinjung, Lenstra, Page et Smart ont proposé de quantifier le « coût pratique » de certaines attaques (factorisation, recherche exhaustive, log discret...) en utilisant la somme d'argent qu'il faudrait verser à un opérateur de *cloud computing* (tel qu'Amazon) pour pouvoir exécuter l'attaque sur son infrastructure [130]. Ces coûts ont été réévalués de manière triennale [81]. Kuo, Schneider, Dagdelen, Reichelt, Buchmann, Cheng et Yang discutent du « coût Amazon » de la résolution pratique de certaines instances du problème du plus court vecteur dans des réseaux euclidiens [138].

Le problème principal du « coût Amazon » est qu'il évolue dans le temps, ce qui le rend peu pratique comme mesure de complexité « toutes choses égales par ailleurs ». De plus, Amazon lui-même, comme tout ce qui vit en ce bas monde, a une espérance de vie finie. Pour pouvoir énoncer des vérités éternelles aussi bien que les théoriciens qui se placent dans le modèle de la *Random Access Machine*, il faudrait une notion abstraite du coût qui puisse se prêter à des raisonnement asymptotique. L'idée qui a fait consensus jusque-là est la suivante.

Définition 1. *Le* coût d'un calcul *est le produit de la taille de la machine sur lequel il s'est exécuté et du temps nécessaire à son exécution.*

$$[\text{Coût}] = [\text{Taille de la machine}] \times [\text{Temps d'exécution}].$$

Bernstein l'appelle le « *Price-Performance ratio* » ; Wiener [200] l'appelle le « *full cost* » ; Lenstra, Shamir, Tomlinson et Tromer [144] l'appellent le « *throughput cost* ». Je l'appelle le *coût*.

La notion du coût « facture » au programmeur son usage de la mémoire. Pour contenir n bits de mémoire, une machine doit être de taille $\Omega(n)$. Un algorithme séquentiel qui s'exécute en temps T et qui utilise M bits d'espace coûte donc $\Omega(TM)$. À ce titre, la notion du coût évite une partie des problèmes exposés par le modèle de la *Random Access Machine*.

Le nombre d'opérations nécessaires à l'exécution du meilleur algorithme séquentiel dans le modèle de la *Random Access Machine* donne une *borne inférieure* sur le coût. Si jamais le calcul ne nécessitait

qu'une quantité de mémoire constante, il pourrait s'exécuter sur une machine de taille $\mathcal{O}(1)$. Ceci tend à confirmer le point de vue, déjà affirmé en conclusion de la section 2.1, que la complexité en temps est une borne inférieure sur la difficulté de l'exécution d'un calcul, mais que cette dernière ne s'y résume pas.

Il est parfois possible de concevoir des implantations parallèles dont le coût correspond à la complexité dans le modèle de la *Random Access Machine*. On peut alors parler de coût optimal. C'est notamment le cas de la recherche exhaustive : en effet, s'il y a N combinaisons à tester et qu'on dispose d'une machine qui a P processeurs (on peut supposer qu'elle est de taille $\mathcal{O}(P)$), alors toutes les combinaisons peuvent être testées en temps N/P.

3.1 Coût et parallélisme

Il y a des exemples de situations où des machines parallèles coûtent moins que des machines séquentielles. Cet argument a été mis en avant par Bernstein [19] avec l'exemple du tri : en 1977, Thompson et Kung ont proposé une architecture systolique à deux dimensions [1] capable de trier n^2 nombres en temps $\mathcal{O}(n)$ [187]. Ceci coûte $\mathcal{O}\left(n^3\right)$, ce qui est à comparer à $\Omega\left(n^4\right)$ pour un processeur unique relié à une mémoire de taille n^2. Observons au passage que le tri de n nombres coûte $\mathcal{O}\left(n^{1.5}\right)$ avec cette machine, ce qui est asymptotiquement plus que le nombre d'opérations nécessaires.

La multiplication de matrices en est un autre exemple. Une autre architecture systolique « bien connue », à deux dimensions, peut multiplier deux matrices $n \times n$ en temps $\mathcal{O}(n)$, avec un coût de $\mathcal{O}\left(n^3\right)$, contre $\Omega\left(n^5\right)$ pour un processeur séquentiel relié à une mémoire de taille n^2.

Bernstein, toujours dans [19], a montré qu'une architecture systolique en deux dimensions de taille $\tilde{\mathcal{O}}(N)$ peut effectuer le produit matrice-vecteur en temps $\tilde{\mathcal{O}}\left(N^{1/2}\right)$, en le ramenant au tri — ici N désigne le nombre total de coefficients non-nuls dans la matrice et le vecteur. Wiener a ensuite fait remarquer [200] que passer en 3D permet à une machine de la même taille de trier en temps $\tilde{\mathcal{O}}\left(N^{1/3}\right)$, donc de faire le produit matrice-vecteur dans le même temps. Ceci se traduit par une borne supérieure de $\tilde{\mathcal{O}}\left(N^{7/3}\right)$ sur le coût de la résolution d'un système linéaire très creux avec l'algorithme de Wiedemann, qui nécessite $\mathcal{O}(N)$ produits matrice-vecteur. Ceci suppose que la matrice a $\tilde{\mathcal{O}}(N)$ coefficients, soit $\mathcal{O}\left(\log^k N\right)$ coefficients par ligne, ce qui est typique de situations cryptographiques (comme les algorithmes sous-exponentiels de factorisation).

Le fait que des machines parallèles coûtent moins cher que des machines séquentielles de sémantique équivalente peut s'expliquer de la manière suivante : dans une machine séquentielle (raisonnable), à chaque étape, seul un nombre constant de bits de la mémoire peuvent être lus ou écrits. Par conséquent, la plupart d'entre eux sont « inactifs » presque tout le temps, mais leur présence contribue au coût du calcul. Dans une machine parallèle, chaque composant a plus de chance de contribuer à la progression du calcul vers le résultat à chaque étape de temps. Ceci corrobore une critique déjà formulée contre le modèle de la *Random Access Machine* section 2.1 : des machines avec *un seul* processeur relié à une *énorme* mémoire n'existent pas dans la réalité, car elles seraient terriblement inefficaces.

Pour que le coût d'un algorithme qui a besoin de mémoire puisse s'approcher sensiblement du nombre d'opérations qui est nécessaire à son exécution (sa « complexité en temps » dans le modèle de la *Random Access Machine*), alors il est *nécessaire* que cet algorithme soit parallèle.

3.2 Le coût est une mesure plus intéressante que le nombre d'opérations

Je me suis convaincu que le coût est une meilleure mesure de la difficulté pratique d'exécuter une attaque cryptographique que le simple nombre d'opérations « élémentaires » qui sont nécessaires.

Cependant, la notion de coût a cependant un côté ennuyeux : elle transforme potentiellement des

1. C'est-à-dire une grille $n \times n$ de processeurs possédant chacun une quantité de mémoire constante ; chaque processeur est relié à ses quatre voisins Nord, Sud, Est, Ouest.

« attaques », c'est-à-dire des algorithmes qui sont suffisamment rapides pour casser quelque chose, en « non-attaques », c'est-à-dire en algorithmes trop coûteux pour casser quoi que ce soit.

La discussion de la section 2.5 (« la recherche exhaustive est-elle plus efficace ? ») se reformule de manière simple avec la notion de coût. Le coût de la recherche exhaustive sur n bits est 2^n ; le coût des attaques « améliorées » est-il inférieur ? La réponse n'est pas évidente : ce sont des algorithmes séquentiels, or pour qu'ils puissent avoir un coût compétitif, il faudrait en étudier des versions parallèles. Paralléliser certaines attaques peut être non-trivial. Pour ne citer qu'un exemple : la méthode rho de Pollard a été publiée en 1975 [170] mais parallélisée efficacement des années plus tard [172, 193]. S'intéresser au coût des attaques, passées ou futures, ouvre des perspectives de recherche intéressantes.

On pourrait argumenter qu'il y a des situations où la notion de coût, telle qu'elle est formulée ci-dessus, n'a pas vraiment d'intérêt même pour un cryptanalyste qui envisage réellement d'exécuter une attaque cryptographique en pratique. S'il dispose d'une machine concrète, son problème n'est pas le « ratio prix-performance » mais le temps d'exécution sur sa machine. Cependant, si la machine en question est une machine parallèle pas trop petite, alors essayer de minimiser le temps d'exécution réel de l'attaque pose les mêmes problèmes que d'essayer de minimiser le coût asymptotique sur le papier : il faut là aussi prendre en compte les « difficultés de la vraie vie » (communications, I/O, latence de la mémoire et du réseau, etc.) qui sont absentes du modèle de la *Random Access Machine*.

Les cryptanalystes n'ont généralement pas le loisir de faire construire des machines parallèles adaptées à leurs besoins (dans le but de minimiser le coût), et il leur faut bien se débrouiller avec ce qu'ils ont. Les jolis réseaux de tri décrits par Bernstein [19] n'ont, à ma connaissance, jamais connu de réalisation pratique. En fait, la plupart des attaques cryptographiques ayant nécessité de gros calculs dont j'ai eu connaissance se sont déroulées sur des infrastructures de calcul mutualisées [2]. Les machines peuvent appartenir à l'opérateur d'un *cloud* public (cas de la collision sur SHA-1 [180]). Elle peuvent aussi appartenir à un centre de calcul national ou universitaire (cas des records de factorisation/log discret [128, 129, 40]). Dans ce contexte où les machines sont partagées, les programmeurs doivent choisir le nombre de processeurs qu'ils veulent « réserver » et parfois s'engager sur une durée d'exécution maximale. Même s'ils ne peuvent pas faire construire une machine parallèle *ad hoc*, ils peuvent choisir dans une certaine mesure le degré de parallélisme qu'ils sont capables d'exploiter. Ils sont généralement « facturés » le nombre d'heures-CPU utilisées... ce qui est très précisément le coût tel qu'il est défini ci-dessus.

De plus, il semble crédible que le coût soit corrélé proportionnellement avec la consommation énergétique du calcul, donc avec un véritable coût pécuniaire (quelqu'un doit bien payer la facture). En 2013, Giridhar *et al* [115] estimaient que dans les supercalculateurs du futur, 100Po de RAM consommeraient une puissance d'environ 4.7MW [3]. Avoir de la mémoire coûte cher ! Dans les machines actuelles, la RAM consomme 30% de l'énergie totale. Le nombre d'opérations dans le modèle de la *Random Access Machine* néglige complètement cet aspect des choses.

D'autre part, lire un mot de 64 bits depuis la mémoire centrale d'un ordinateur contemporain (avec une faute de cache) nécessite 10 fois plus d'énergie que d'effectuer une multiplication flottante double-précision entre les registres du processeur [194, appendix A]. Et plus les données sont éloignées des unités de calcul, plus la facture énergétique augmente. Plus précisément, avec des processeurs gravés en 22nm, il faut 6.4 pJ pour effectuer une multiplication flottante double-précision, tandis qu'il faut 11.2 pJ pour déplacer un flottant double-précision de 5mm [95].

3.3 Bornes inférieures sur le coût

Une notion comparable au coût était en vogue au début des années 1980, quand la recherche sur l'intégration à très grande échelle (« *Very Large Scale Integration* », VLSI) battait son plein. Des bornes inférieures ont été sur établies le produit aire-temps de n'importe quel circuit VLSI effectuant

2. Il y a bien sûr des exceptions [24, 181].

3. Avec une technologie de mémoire 3D qui commence à être utilisée à l'heure actuelle ; avec de la simple DDR4, cela consommerait 25MW ; c'est le point de l'article en question.

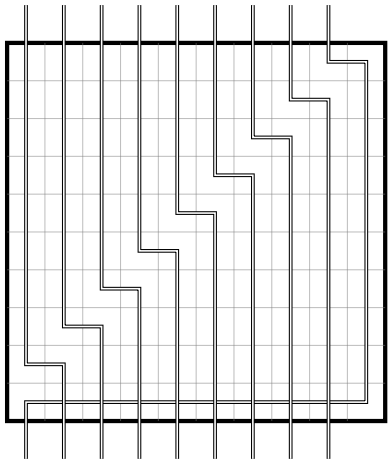


FIGURE 3.1 – Une réalisation possible d'un circuit VLSI qui effectue la rotation d'un bit vers la gauche. La surface est de l'ordre de n^2, où n désigne le nombre d'entrées. Les croisements ne peuvent avoir lieu qu'à angle droit et un seul fil peut occuper chaque case dans le même sens.

des calculs courants : rotation et convolution [197], tri [31], multiplication entière [64], transformée de Fourier [186], produit de matrices [176], etc. Toutes ces bornes inférieures ont été complétées par des circuits qui les atteignent, donc elles sont optimales.

Ces bornes lient entre elles l'aire A d'un circuit (sa taille) et le temps T nécessaire à son exécution. Elles sont généralement de la forme « $AT \geq n^{1.5}$ » ou bien « $AT^2 \geq n^2$ ». La première découle de la seconde lorsque $A \geq n$, ce qui est souvent le cas car les entrées doivent souvent être lues entièrement avant que la sortie ne puisse être produite. Ce sont des bornes sur le « rapport prix-performance », donc sur le coût du calcul : on peut avoir ou bien un petit circuit (pas cher mais lent) ou bien un gros circuit (cher mais rapide).

Ces bornes inférieures impliquent que le coût de certaines opérations élémentaires de la cryptanalyse (tri, multiplication d'entiers/polynômes/matrices, ...) est *asymptotiquement supérieur* au nombre d'opérations qui est nécessaire à leur réalisation. Ceci provient du fait qu'elle prennent en compte la complexité de communication, ce que le modèle de la *Random Access Machine* ignore complètement. Ces bornes permettent parfois de donner des bornes inférieures sur le coût de certaines attaques, et de démontrer qu'elle sont plus coûteuse que la recherche exhaustive.

Ces bornes reposent de manière cruciale sur le fait que les circuits sont *plats*. Une des techniques de preuve consiste à montrer que si le circuit est coupé en deux moitiés à peu près égales, alors une certaine quantité d'information doit nécessairement franchir la coupe. Par ailleurs, la coupe « tranche » un certain nombre de fils par lesquels cette information doit nécessairement transiter. Ces fils occupent une certaine surface. Plus le temps d'exécution du circuit est faible, plus le flot de données qui doit franchir la coupe est rapide, donc plus le nombre de fils coupés est important, et plus la surface totale du circuit est importante.

Cette manière d'appréhender la complexité des calculs donne un rôle majeur au coût des communications : déplacer des données est coûteux. La figure 3.1 montre un circuit qui n'effectue aucun calcul (il effectue une rotation des n bits en entrée), mais dont la surface est de l'ordre de n^2. L'opération en question est soumise à la borne $AT^2 \geq n^2$, donc la construction ci-dessus est optimale.

Le caractère planaire des circuits joue un rôle crucial dans les preuves de ces bornes, mais est-ce une restriction fondamentale ? À ma connaissance, ces bornes n'ont pas été généralisées en trois dimensions. Il est néanmoins tentant de conjecturer qu'elles se généralisent en « Volume $\times$ Temps $= \Omega\left(n^{4/3}\right)$ ». Par exemple, une architecture systolique (« mesh ») à d dimensions de taille n^d peut trier n^d nombres en temps $\mathcal{O}(n)$ [187]. Pour $d = 2$, on retombe sur $AT = n^{3/2}$, tandis que pour $d = 3$ on obtient $VT = n^{4/3}$. De plus, le théorème 1 ci-dessous va dans le même sens pour une large classe d'algorithmes.

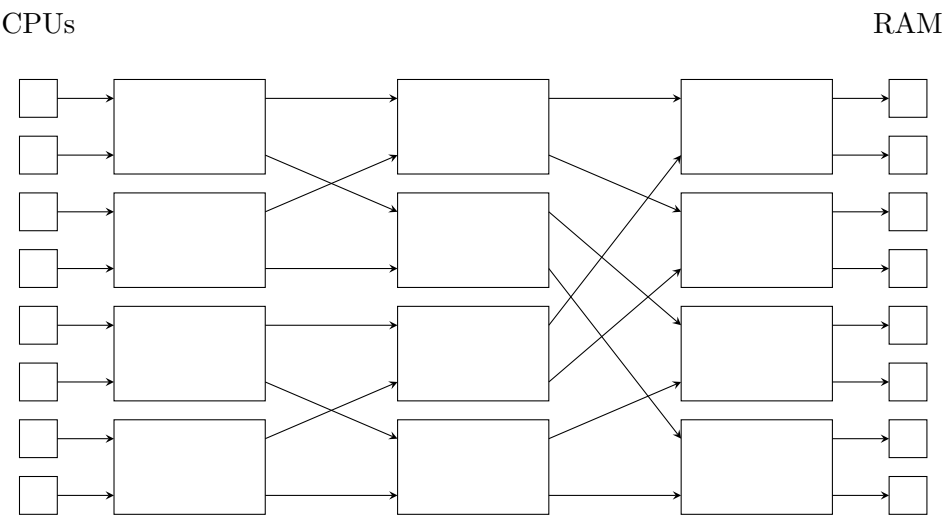


FIGURE 3.2 – Un *Clos network* pour relier ensemble des processeurs et de la mémoire.

3.4 Quelle est vraiment la taille de la machine ?

> A vast body of theoretical research has focused [...] on overly simplistic models of parallel computation, notably the PRAM
>
> Culler, Karp, Patterson, ..., 1993, [80]

La plupart des problèmes discutés jusque-là tournent autour de ce qui se passe lorsque des processeurs accèdent à la mémoire, surtout lorsqu'elle est grande. Cela met en réalité en jeu un réseau de communication « caché », ce qui induit des coûts non-triviaux, des délais, etc. Dans son article « *The Full Cost of Cryptanalytic Attacks* » [200], Wiener aboutit à la même conclusion, et il la quantifie. Le raisonnement suivant en est tiré.

Pour relier n processeurs à n éléments de mémoire, on peut par exemple utiliser un *Clos network* inventé par Clos en 1953 [73], représenté figure 3.2. Il faut utiliser $n \log n$ commutateurs, donc ces derniers dominent en fait asymptotiquement la taille de la machine, qui n'a que n processeurs et n éléments de mémoire. Qu'en est-il de la longueur des câbles ? Entre les deux dernières couches de commutateurs, il y a n câbles de longueur moyenne $n/2$. La longueur totale de câble est donc n^2, et c'est donc la longueur de ces derniers qui domine la taille de la machine ! Au passage, le cuivre et la fibre optique ne sont pas gratuits.

La disposition des processeurs et des éléments de mémoire sur une ligne est sous-optimale. En plaçant les éléments de mémoire en 2D sur un carré de taille $\sqrt{n} \times \sqrt{n}$, on pourrait réduire la longueur moyenne des câbles entre les deux dernières couches à $\approx \sqrt{n}$. La longueur totale, donc le volume occupé par ces câbles serait donc $n^{1.5}$; il tiendraient donc tout juste dans un cube de côté $\sqrt{n}$. Cf. figure 3.3.

Un petit raisonnement géométrique montre qu'on ne peut pas réduire le coût total du câblage sous la barre de $n^{1.5}$, quelle que soit la disposition des composants. Supposons que n processeurs fassent des accès aléatoires à n éléments de mémoire. Supposons aussi que la longueur totale de câble nécessaire soit n^{1+z} pour un certain $z \geq 0$, et que tous les composants sont enfermés dans un cube. Le volume des câbles impose que ce cube soit de dimension au moins $n^{(1+z)/3}$. Supposons que les processeurs, la mémoire et les commutateurs soient bien répartis dans le cube, de telle sorte qu'on peut découper le cube en trois régions A, B et C par deux plans parallèles à distance $\Theta\left(n^{(1+z)/3}\right)$, comme représenté par la figure 3.4.

Si les accès mémoire sont aléatoires, alors environ un tiers des requêtes émises à chaque unité de temps par les processeurs de la zone A doivent être conduits vers la mémoire de la zone C. Ceci nécessite que $\Omega(n)$ câbles traversent la zone B. La longueur totale (donc le coût) de ces câbles est donc $n^{(4+z)/3}$.

FIGURE 3.3 – Une armoire d'un Cray XC40.

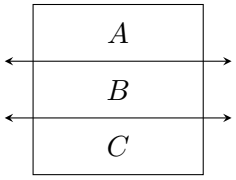


FIGURE 3.4 – Illustration du raisonnement géométrique. Si les processeurs de la zone A accèdent à la mémoire de la zone C, alors de nombreux câbles doivent traverser la zone B.

On avait initialement supposé que ce coût était n^{1+z}, donc la seule solution est $z = 1/2$. Le câblage coûte donc bien $n^{1.5}$.

Wiener prouve (en éliminant une partie des hypothèses ci-dessus) le résultat suivant.

Théorème 1 ([200], légèrement reformulé)**.** *Dans une machine où p processeurs peuvent effectuer des accès aléatoires à une mémoire partagée de taille M, avec une fréquence r, la longueur totale des fils reliants les éléments est $\Omega\left((pr)^{1.5}\right)$. Cette borne est optimale car on peut réaliser une machine qui l'atteint.*

(La fréquence est le débit moyen, en bits/cycle, entre un processeur et la mémoire).

Ceci ne nécessite pas d'hypothèse sur les positions relatives des processeurs et des éléments de mémoire. Par contre, ceci suppose que le calcul se déroule en « état stationnaire » pendant suffisamment longtemps.

En d'autres termes, si beaucoup de processeurs doivent accéder fréquemment et de manière arbitraire à une grande mémoire partagée, alors il y a un coût caché (le réseau de communication). Et ce réseau a une taille qui peut même dominer asymptotiquement la taille de la machine ! Il s'agit dans le fond

d'une généralisation à trois dimensions des bornes AT de l'époque VLSI, et les arguments utilisés sont comparables.

Ceci implique qu'un calcul qui nécessite T unités de temps sur p processeurs, en accédant aléatoirement à une mémoire M avec une fréquence r coûte $T(p + M + (pr)^{1.5})$.

Le théorème admet un corollaire assez simple dans le cas où $r = \mathcal{O}(1)$, c'est-à-dire dans la situation courante où les processeurs accèdent aléatoirement à la mémoire en permanence (une situation courante dans les attaques cryptographiques). Dans ce cas précis, le coût de l'exécution d'un algorithme qui nécessite T opérations, dont $\Omega(T)$ accès à une mémoire de taille M est $\Omega\left(TM^{1/3}\right)$. Cette borne est atteinte s'il est possible de paralléliser le calcul (avec efficacité optimale) sur $p = M^{2/3}$ processeurs.

Ceci suggère l'utilisation de machines parallèles où chaque nœud possède une mémoire de l'ordre de $M^{1/3}$. La mémoire totale est donc le cube de la mémoire propre à chaque nœud. Ceci n'est pas très facile à comparer avec grosses machines parallèles décrites dans la figure 2.1.

Il faut noter que si on s'intéresse à des machines qui sont confinées dans un espace à deux dimensions seulement, alors la borne du théorème devient $\Omega\left((pr)^2\right)$. Alors dans la situation précédente, le coût optimal est $TM^{1/2}$, atteint avec $P = M^{1/2}$.

Le théorème 1 affirme somme toute quelque chose d'assez banal : une machine nécessaire à l'exécution d'un algorithme qui demande des accès intensifs et arbitraires à une grosse mémoire partagée est plus grosse, et donc plus coûteuse, qu'une machine avec des processeurs qui n'ont pas besoin de communiquer entre eux.

Il y a d'autres exemples documentés de « coûts cachés des communications ». Supposons par exemple que 2^n processeurs soient reliés entre eux sur un hypercube de dimension n ; chaque nœud est connecté à n voisins — cette architecture est un grand classique des cours de parallélisme. Le tri bitonique est simple à implanter sur ce genre de machines : il permet de trier 2^n éléments (un par nœud) en $n^2/2$ étapes, où chaque étape nécessite une communication entre un nœud et un de ses voisins, sans aucune contention sur les liens du réseau. Par conséquent, trier 2^n éléments prendrait un temps $\mathcal{O}\left(n^2\right)$ sur une machine de taille $\mathcal{O}(2^n)$. Ceci semble battre les bornes inférieures du type $AT = N^{3/2}$ ou $VT = N^{4/3}$ énoncées dans la section 3.3.

Mais on triche, et il y a un coût caché. Vitányi [196] a démontré que la longueur totale des fils nécessaire pour connecter entre eux les nœuds d'un hypercube dans notre monde bassement tridimensionnel est $\Omega\left(2^{4n/3}\right)$. Par conséquent, la « taille de la machine » est en fait plus grande qu'annoncé. Et encore ! Ceci suppose que les fils ont un volume nul. Si les fils occupaient un certain espace, cela repousserait les nœuds de la machine plus loin, ce qui nécessiterait des fils plus longs encore... Il est également connu que des variantes d'hypercube (les « *cube-connected cycles* ») peuvent être disposés sur une grille en trois dimensions avec un volume total $2^{3n/2}$, et il semble que ceci soit optimal (en tout cas, on retombe sur le théorème 1).

3.5 Modèles de calcul plus réalistes

> Recent studies confirm that sorting continues to account for roughly one-fourth of all computer cycles. [...] It is well documented that the bottleneck in external sorting is the time for input/output (I/O) between internal memory and secondary storage.
>
> A. Aggarwal et J. S. Vitter, 1988, [2]

Si on cherche à discuter sur le papier du coût de telle ou telle attaque cryptographique sans essayer de la programmer, on pourrait vouloir faire attention aux hypothèses implicites ou aux « coûts cachés »

dont on vient de discuter. Ceci a conduit de nombreux chercheurs à proposer des modèles alternatifs. Mais la quête du « bon » modèle de calcul n'est pas terminée.

Aggarwal et Vitter discutent du coût du tri, de la FFT, de la transposition de matrices dans le « modèle Entrée/Sortie » (*IO model*) [2]. Il s'agit d'un modèle générique où un processeur est relié à une mémoire *interne* (de taille M, petite), elle-même reliée à une mémoire *externe* (non-bornée). Les données se transfèrent entre les deux mémoires par *blocs* de taille B. Jusqu'à P blocs peuvent être transférés en parallèle. Dans ce modèle, la mesure de complexité est le nombre de transferts entre les deux mémoires. Ce modèle correspond à la situation où on doit traiter des données qui ne tiennent pas en cache, ou bien qui ne tiennent pas en mémoire et doivent être stockées sur des disques. Le tri et la FFT de taille N nécessitent alors

$$\frac{N}{PB}\frac{\log(1+N/B)}{\log(1+M/B)}$$

transferts. En 1972 déjà, Floyd discutait d'un cas simple du même problème [106].

Dans la même veine, Frigo, Leiserson, Prokop et Ramachandran ont inventé les algorithmes « inconscients du cache » (*cache-oblivious*), qui minimisent le nombre de fautes de cache, sans connaître sa taille ni ses caractéristiques [113]. Ils discutent d'un modèle de calcul où la complexité est donnée à la fois par le nombre d'opérations et par le nombre de fautes de cache. L'idée est de garder le second au minimum sans que le premier n'augmente trop par rapport au modèle de la *Random Access Machine* qui ignore complètement le problème.

Culler, Karp, Patterson, Sahay, Schauser, Santos, Subramonian et von Eicken définissent le modèle LogP pour le calcul parallèle. Il s'agit de considérer P processeurs fonctionnant de manière asynchrone, qui communiquent sur un réseau « réaliste ». Ce réseau est décrit par plusieurs paramètres :

- L : sa *latence*, qui est le temps nécessaire au transfert d'un mot, ou de quelques mots, entre deux processeurs.
- o : le surcoût (*overhead*), qui décrit le temps que passe un processeur à envoyer ou à recevoir un message, pendant lequel il ne peut rien faire d'autre.
- g : le temps minimal (*gap*) qui s'écoule entre deux transmissions consécutives sur le même processeur. L'inverse de g donne la bande passante disponible au processeur.

Dans le cadre de la compétition « post-quantique » du NIST, certaines équipes font la différence entre nombre d'opérations et nombre d'accès à la mémoire. Par exemple, le document de soumission de NTRU Prime distingue les attaques dans des modèles de calculs dits locaux, où l'information se propage à vitesse finie (machine de Turing à une seule bande, circuits VLSI, ...) et les autres. Le même document (notamment sa section 6.6 : « *Interlude : memory, parallelization, and the cost of sorting*) discute en condensé des mêmes problèmes que ceux que soulèvent ce manuscrit. Ces arguments ont ensuite été repris par une l'équipe des concepteurs de Rainbow [184]. Cela conduit les auteurs à tirer un trait d'égalité entre le coût d'un accès à la mémoire et le fait d'effectuer un certain nombre d'opérations de calcul (« *we estimate the cost of each access to a bit within N bits of memory as the cost of $N^{0.5}/2^5$ bit operations.* »). On parlera ici du « modèle NIST » pour désigner le fait qu'un accès à une mémoire de taille M doit être facturé un coût de $M^{1/3}$.

Sans rentrer dans les détails, on peut noter que cette idée est à manier avec prudence car il y a des situations où... elle est fausse. Par exemple, si p processeurs accèdent à *la même* case mémoire, ils peuvent le faire avec un coût en énergie correspondant à celui d'*un* accès à la mémoire.

Le corollaire simple du théorème 1 donne un coût de $TM^{1/3}$ pour des algorithmes qui accèdent tout le temps à la mémoire de façon aléatoire. Le modèle de la *Random Access Machine* donne une complexité de TM^0. On pourrait généraliser et dire que le coût du calcul est TM^β, pour un certain exposant $0 \leq \beta$ qui fixe le « prix de la mémoire ». Sur la base de considérations géométriques, Wiener postule que $\beta = \frac{1}{3}$. Mais on pourrait aussi envisager d'autres valeurs, plus élevées (rares sont les machines qui s'étendent de la même manière dans les trois dimensions) ou plus faibles.

Pour raisonner asymptotiquement et sans trop se compliquer la vie dans le modèle du coût, quelques solutions raisonnables peuvent être envisagées.

1. Modèle « *Cloud* ». On utilise un nombre arbitrairement grand de nœuds d'un cluster, possédant chacun une mémoire de taille M (indépendante de la taille de l'instance du problème [4] considéré). Ces nœuds sont reliés entre eux par un réseau idéal de débit fixé, sans congestion et non-bloquant.
2. Modèle « *Mesh* ». On peut utiliser une grille 1D, 2D ou 3D de taille arbitraire de nœuds d'un cluster, possédant chacun une mémoire de taille M (indépendante de la taille de l'instance du problème). Chaque nœud est relié à ses 2, 4 ou 6 voisins par des liens qui supportent un débit fixé. Si on utilise N nœuds pendant un temps T, le coût est NTM. Le réseau peut aussi être torique sans surcoût.
3. Modèle « pas de modèle ». On exécute du vrai code sur une vraie machine, et on mesure le temps d'exécution sur une horloge murale.

Le modèle *Mesh 3D* semble implantable sans surcoût caché dans le monde matériel ; le réseau est complètement spécifié et la longueur des fils est proportionnelle au nombre de nœuds.

Plusieurs grandes machines de calcul scientifique ont des réseau de topologie comparable. Les IBM BlueGene/L et IBM BlueGene/P ont un réseau en tore 3D. Les BlueGene de toutes les générations sont formées de d'armoires contenant contenant 32 cartes, contenant 32 processeurs avec leur propre RAM (cela fait donc 1024 nœuds par armoire). Une BlueGene/L a tenu la tête du classement des machines les plus puissantes de novembre 2004 (avec 16 384 nœuds) jusqu'à juin 2008 (avec 106 496 nœuds), en passant par une étape intermédiaire avec 65 536 nœuds. Une BlueGene/P (`intrepid`) a eu 40 960 nœuds.

Beaucoup de machines Cray (`jaguar`, 7 832 nœuds pour la partie XT4 plus 18 688 nœuds pour la partie XT5 ; `titan`, un XK7 à 18 688 nœuds) ont également un réseau en tore 3D.

Le `K computer` (Riken Advanced Institute for Computational Science, Japan) possède 88 128 nœuds (864 armoires de 96 nœuds disposées sur une grille 24×36 ; les processeurs sont donc disposés sur une grille 3D de taille $48 \times 72 \times 24$) avec un réseau maison « Tofu ». Les nœuds sont disposé dans un grand tore 3D, composé de petits tores 3D de dimensions $2 \times 3 \times 2$. Son successeur `fugaku` fonctionne sur le même principe, avec 158 976 nœuds (384 par rack avec 396 racks complets et 36 demi-racks).

Il faut noter qu'il n'y a pas nécessairement une correspondance parfaite entre la disposition physique des processeurs dans la machine et la dimension du tore « virtuel » exposé aux programmeurs. Par exemple, une BlueGene/L, un demi-rack de 512 nœuds expose un tore de dimension $8 \times 8 \times 8$ aux programmeurs, alors que la disposition physique des processeurs est $2 \times 8 \times 16$; la machine complète à 65536 nœuds expose un tore de dimension $64 \times 32 \times 32$ aux programmeurs, alors que la disposition physique des processeurs est $16 \times 64 \times 32$. Il y a donc de « longs fils » qui se baladent.

La machine `sequoia` (Lawrence Livermore National Laboratory, USA) est une IBM BlueGene/Q de 98 304 nœuds. Elle est composée de 96 racks disposés sur une grille 8×12. Dans l'espace physique, les CPUs sont donc disposés sur une grille 3D de taille $16 \times 192 \times 32$. Les nœuds sont reliés par un réseau en tore 5D de dimension $16 \times 12 \times 16 \times 16 \times 2$ (il y a donc de « longs fils »). Plus de détails peuvent être trouvés dans [77].

Tout ceci suggère néanmoins que la topologie en tore 3D (ou des variantes avec quelques dimensions de plus) passe bien à l'échelle et supporte un très grand nombre de nœuds.

Qu'en est-il du modèle « *Cloud* » ? Le problème potentiel est que l'architecture du réseau n'est pas décrite et qu'elle pourrait entraîner un coût caché. L'hypothèse d'un réseau idéal est-elle réaliste à grande échelle ?

Certaines grosses machines de calcul scientifique (`summit` par exemple) ont un réseau en *fat tree* qui permet des échanges non-bloquants (n'importe quel port peut être connecté à n'importe quel autre

4. En fait, si on ne fait pas attention, on risque de retomber sur une variante du problème discuté section 2. Il faut que la quantité de mémoire de chaque nœud soit au moins $\log_2 N$, où N désigne la taille de l'instance. En effet, il faut bien que chaque nœud puisse connaître son *numéro*, or ceci va occuper $\mathcal{O}(\log_2 N)$ bits. Le modèle le plus simple est donc « transdichotomique » : chaque nœud possède un nombre constant de cases mémoire de taille au moins $\log N$.

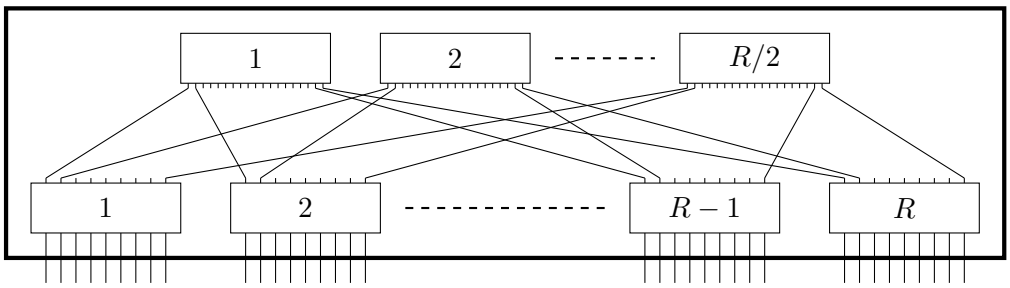


FIGURE 3.5 – Le réseau en *fat tree* à deux étages. On forme un maxi-commutateur à $N^2/2$ ports à partir de $3N/2$ commutateurs à N ports.

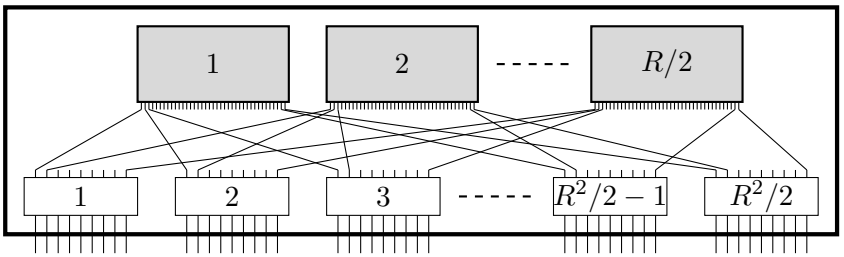


FIGURE 3.6 – Le réseau en *fat tree* à trois étages utilisé dans `summit`. Les boites grises représentent les maxi-commutateurs à deux étages de la figure 3.5.

port libre, sans interférence du reste du trafic). La technique a été proposée par Leiserson en 1985 [143]. À partir de $1.5N$ petits commutateurs non-bloquants à N ports, on peut fabriquer un commutateur non-bloquant à $N^2/2$ ports, comme l'illustre la figure 3.5. On peut ensuite recommencer récursivement, et former un ultra-commutateur à k étages qui connecte $2(N/2)^k$ ports. Dans la machine `summit`, des commutateurs InfiniBand standard à $N = 36$ ports sont utilisés dans un arbre à $k = 3$ étages, ce qui offre 11 664 ports au final (la machine a 9 216 processeurs et tous ont leur propre interface réseau). En théorie, le nombre de composants matériels augmente linéairement avec le nombre de nœuds connectés (le nombre de câbles, le nombre de commutateurs, etc.).

En pratique, c'est une autre affaire ; le coût et surtout la longueur des fils augmentent de manière prohibitive, et à la fin on obtient quelque chose qui ressemble à la figure 3.3. En tout cas, il n'est pas réaliste de supposer que les opérateurs de *clouds* publics mettent en place une telle topologie de réseau. Et de toute façon, on ne peut pas échapper au théorème 1 sur le coût du câblage : ce genre de machine satisfait les hypothèses du théorème, donc il y *a* bien un surcoût caché.

Il faut noter que les deux modèles *mesh* et *cloud* ne sont pas équivalents : il est possible de réaliser certaines opérations (tri, FFT, ...) plus rapidement dans le *cloud* que dans un *mesh* 3D — la raison principale, c'est que lorsque tous les nœuds doivent échanger des données avec tous les autres, cela prend plus longtemps sur un tore 3D que dans le réseau idéal du *cloud*.

En effet, supposons que chacun des n nœuds possède M données et doivent envoyer une tranche de taille M/n à chacun des $n-1$ autres nœuds. Un certain nombre de calculs très communs font intervenir ce motif de communication particulier : le tri, la transposition de matrice, la transformée de Fourier, etc.

Avec le réseau idéal du modèle « Cloud » (ou avec un *fat tree*), ceci devrait prendre un temps constant, indépendamment de n.

Par contre, avec un réseau en tore 3D, la durée de ce « *all-to-all shuffle* » devrait augmenter en $\Theta\left(n^{1/3}\right)$. Si le tore est de dimension $u \times v \times w$ et que $u \geq v, w$, alors on considère le plan d'équation $x = u/2$. Il partitionne les nœuds de la machine en deux moitiés de taille égale. Chaque nœud du côté gauche doit envoyer la moitié de ses données du côté droit et réciproquement. Le volume de données qui doit traverser le plan est donc $nM/2$. Or le plan offre $v \times w$ liens réseaux reliant les deux moitiés.

Le temps nécessaire au transfert est donc $\Omega\left(n/(v \times w)\right)$. Dans le cas le plus favorable, on trouve $n^{1/3}$, comme annoncé. Cette opération de communication coûte donc $n^{4/3}$, sur un réseau en tore 3D.

Cet argument, très classique, revient à dire que la *bisection bandwidth* du réseau est limitée. Cette grandeur désigne le débit maximal qui peut franchir une coupe qui sépare les n nœuds du réseau en deux paquets de taille $n/2$.

Pour toutes ces raisons, il est plus sage de décrire des algorithmes dans un *mesh* 3D. La durée des transmissions peut donc dépendre de la taille de la machine. Avec p nœuds, un message peut traverser jusqu'à $3p^{1/3}$ nœuds pour arriver à destination. Une autre situation récurrente est celle où tous les processeurs envoient simultanément un message à une destination aléatoire. Le routage peut alors être effectué par un algorithme simple dû à Valiant en 1981, en temps $\tilde{\mathcal{O}}\left(p^{1/3}\right)$ avec forte probabilité [191]. Il consiste à transférer tous les messages sur le plan x jusqu'à la bonne coordonnée, puis faire de même sur le plan y et enfin sur le plan z. Le nombre maximal de messages qui s'accumulent dans un nœud est borné par $\mathcal{O}\left(\log p\right)$ avec forte probabilité. Ceci découle de l'observation classique suivante : si on lance n balles aléatoirement dans n seaux, alors le seau le plus rempli contient environ $\log n$ balles.

Enfin, on pourrait discuter de la question de devoir accepter une entrée de taille exponentielle, ou produire une sortie de taille exponentielle. La machine démarre-t-elle avec l'entrée déjà en mémoire ? Ou bien doit-elle la lire depuis le monde extérieur ? Et à quelle vitesse ? Les mêmes questions se posent pour la sortie.

3.6 Confrontation à la réalité #3

Le tri est une brique de base importante dans beaucoup d'attaques cryptographiques. Beaucoup de chercheurs écrivent que trier une liste de N éléments aléatoires de $\{0,1\}^n$ (avec N exponentiellement grand) « coûte » $\mathcal{O}\left(N \log N\right)$ ou $\mathcal{O}\left(nN \log N\right)$.

La section précédente affirme que le tri, la multiplication, la FFT, etc. « coûtent » $\Omega\left(n^{3/2}\right)$ ou $\Omega\left(n^{4/3}\right)$ à cause de la grande quantité de communication qu'ils entraînent, c'est-à-dire sérieusement plus que $\mathcal{O}\left(n\right)$ ou $\mathcal{O}\left(n \log n\right)$. Qu'en est-il dans la réalité ? Dans la suite de cette section, on essaye de trancher cette question expérimentalement.

Mais avant tout, coupons les cheveux en quatre dans le modèle de la *Random Access Machine*. Il est connu depuis les travaux de MacLaren en 1966 [155] que des chaînes de bits aléatoires peuvent être triées en temps linéaire, c'est-à-dire en temps $\mathcal{O}\left(nN\right)$ — l'aléa de l'entrée permet de dépasser la borne inférieure en $N \log N$ des méthodes par comparaison. Voici une manière de faire qui utilise un espace additionnel $\mathcal{O}(\sqrt{N})$: effectuer deux passes de *radix sort* sur $0.5 \log N$ bits puis terminer en effectuant un tri par insertion sur l'ensemble du tableau (ceci garantit qu'il est trié). L'espace additionnel sert à maintenir un tableau de $\sqrt{N}$ compteurs correspondant au nombre de catégories possibles du *radix sort*. Après les deux passes de *radix sort* (qui s'exécutent en temps linéaire), le tableau est trié selon les premiers $\log N$ bits. Ceci réduit le nombre moyen d'inversions de $\approx N^2/4$ à $\approx N/4$ en moyenne. Par conséquent, le temps d'exécution du tri par insertion, qui est proportionnel au nombre d'inversions, est linéaire lui aussi. Tout ceci est expliqué dans les ouvrages d'algorithmiques classiques utilisés dans l'enseignement, notamment [76, 134].

3.6.1 Méthodologie

Étudier expérimentalement le coût d'une opération de calcul pose des problèmes méthodologiques intéressants. On s'intéresse à un calcul sur des données de taille n, avec un algorithme qui nécessite $T(n)$ opérations — dans le cas du tri, $T(n) = n \log n$, mais cette discussion est plus générale.

On peut mesurer facilement le temps d'exécution d'un programme qui résout le problème sur des données de plus en plus grandes. Mais le coût fait aussi intervenir la « taille de la machine » ; comment jauger cette dernière ? Est-ce la taille de la mémoire utilisée ? Le nombre de processeurs ? Et comment le choisir ? La longueur des câbles réseaux ? Un mélange de tout ceci ?

Le plus simple est d'étudier ce qui se passe à l'intérieur d'une seule machine. Pour simuler une notion de coût raisonnable, imaginons qu'on doive louer à un opérateur de *cloud* une machine permettant de traiter une instance de taille n. L'opérateur loue des machines virtuelles et propose plusieurs tailles : i cœurs et $i \cdot M$ Go de RAM, pour $1 \leq i \leq 64$ disons — en réalité, l'opérateur possède des machines physiques de taille fixe, mais il les *partitionne* et les loue « par morceaux ».

Pour avoir assez de mémoire, il faut choisir un i suffisamment grand. Le coût est proportionnel à i (donc à la quantité de mémoire nécessaire) et au nombre de secondes de la location. Plusieurs architectures matérielles sont disponibles (Intel Xeon, Amd EPYC, IBM Power8, ARM, ...), avec des tarifications à la seconde différentes [5].

Du coup, le nombre de cœurs disponibles est *proportionnel* à la mémoire disponible. La mesure (honnête) du coût nécessite donc l'emploi de codes *parallèles*. Utiliser un seul cœur aboutirait à obtenir un coût en $nT(n)$ (on sous-utilise la machine, donc on surestime le coût). Mais, plus subtilement, utiliser tous les cœurs de la machine physique, quelle que soit la taille de la mémoire nécessaire, est aussi une erreur méthodologique : on sur-utilise la machine, qui ne peut rien faire d'autre même si la quantité de mémoire consommée est très faible. La bonne méthodologie consiste donc à faire varier le nombre de processeurs en même temps que la taille de la mémoire.

Dans ce contexte, le coût est directement fonction d'une notion standard du calcul scientifique, à savoir l'*efficacité* de la parallélisation des calculs. Rappelons que si on passe de 1 à p processeurs, on s'attend à une accélération d'un facteur p dans le meilleur des mondes. L'*accélération* obtenue est $S := T_1/T_p$, où T_i désigne le temps d'exécution avec i processeurs. L'*efficacité* $E := S/p$ quantifie à quel point l'accélération obtenue dans la réalité est proche de celle qu'on aurait dans le meilleur des mondes.

Si le passage à l'échelle des calculs était parfait (accélération de i avec i processeurs), alors le temps d'exécution serait $T(n)/i$ sur une machine de taille i, et le coût serait $T(n)$, ce qui est optimal. Si le passage à l'échelle s'avère sous-optimal, alors on observera un temps d'exécution plus élevé. L'étude du coût se ramène donc à l'étude du *weak scaling* [6], une autre notion complètement standard dans le contexte du calcul haute-performance.

3.6.2 Tri interne (dans un seul nœud)

Vu la discussion qui précède, on utilise des fonctions de tri *parallèles* sur étagère :

- La fonction `parallel_sort` de la bibliothèque Intel Thread Building Block, qui implante a priori un QuickSort parallèle.
- La fonction `sort::block_indirect_sort` de la bibliothèque Boost.

On mesure le temps d'exécution ; le coût correspondant est $\Theta(T \cdot i)$. Si l'efficacité de la parallélisation était optimale, alors le temps nécessaire pour trier un volume de données $\Theta(n)$ avec $\Theta(n)$ processeurs devrait être $\mathcal{O}(\log n)$, et le coût serait alors $\mathcal{O}(n \log n)$. L'hypothèse alternative est que le coût est une fonction du type $\mathcal{O}(n^\alpha)$.

On mesure le temps d'exécution de ces fonctions sur des problèmes de plus en plus gros et donc sur des « machines » de plus en plus grosses (i cœurs et iM Go de RAM). Le tableau 3.7 décrit les machines utilisées, qui ont des architectures variées.

Le coût empiriquement observé dépend de la machine et de la qualité de la bibliothèque de tri parallèle. La figure 3.8 montre l'évolution du coût (en Go-heures) en fonction du nombre de Go de données triées (on trie des entiers de 64 bits aléatoires). Par dessus les points mesurés, des modèles en $n \log n$ et en n^α ont été ajustés. Ce dernier semble mieux coller aux mesures.

Essayons de conclure plus fermement, en essayant d'exclure l'« hypothèse nulle » H_0 suivante : le coût est $\alpha n \log n + \epsilon$, où α est un paramètre inconnu et ϵ un bruit qui suit une distribution normale de

5. Ceci est une replique tout à fait réaliste du modèle de facturation du *cloud* Amazon.

6. Le *weak scaling* quantifie l'évolution du temps de calcul lorsque le nombre de processeurs grandit en même temps que la taille des données à traiter ; le *strong scaling* mesure lui l'évolution du temps de calcul à taille de données fixées, lorsque le nombre de processeurs grandit.

Cluster	CPU	ISA	Cœurs / CPU	RAM (Go)
grue	2 x AMD EPYC 7351	x86-64	16	128
grouille	2 x AMD EPYC 7452		32	128
grvingt	2 x Intel Xeon Gold 6130		16	192
yeti	4 x Intel Xeon Gold 6130		16	768
drac	2 x IBM POWER8+	PowerPC	10	128
pyxis	2 x ThunderX2 99xx	ARM	32	256

FIGURE 3.7 – Description des machines de Grid'5000 utilisées pour l'expérience.

variance inconnue, centrée en zéro. En effet, comme on trie des données aléatoires, le temps d'exécution de la procédure est nécessairement une variable aléatoire.

Si on trace les courbes « coût / $n \log n$ », on devrait trouver $\alpha + \epsilon/n \log n$, c'est-à-dire une constante plus un bruit centré en zéro. On trouve en fait qu'elles sont systématiquement *croissantes* comme l'atteste la figure 3.9, ce qui va dans le sens d'un rejet de l'hypothèse nulle, ou en tout cas de l'existence d'un phénomène qu'elle ne prend pas en compte.

On peut calculer $\tilde{\alpha}$ comme étant l'espérance empirique de la grandeur « coût / $n \log n$ ». On peut ensuite observer ϵ en soustrayant $\tilde{\alpha} n \log n$ au coût empirique observé, puis tester si ϵ suit bien une loi normale centrée en zéro. En l'occurrence, ceci peut être établi avec un test de Student (« *one-sampled t-test* »), qui donne une p-valeur. Le tableau 3.10 montre les p-valeurs ainsi obtenues, qui sont toutes très largement inférieures au seuil de 5% qui est traditionnellement considéré comme « statistiquement significatif ». Par conséquent, les données observées pèsent dans le sens d'une très forte présomption contre l'hypothèse nulle, qu'on va rejeter dans toutes les combinaisons de machine/librairie testées.

On en conclut que le coût des routines de tri testées sur les machines testées est de la forme n^α. Le tableau 3.10 montre les valeurs de α. On remarque que, sur bon nombre de machines, la librairie Boost permet de trier avec coût environ $n^{4/3}$.

3.6.3 Tri externe sur un cluster

Lorsque la taille n des données à trier va dépasser quelques centaines de gigaoctets, il deviendra nécessaire de répartir les données sur plusieurs nœuds d'un cluster car ils ne tiendront plus dans la mémoire d'une seule machine. Et là —surprise !— les caractéristiques du réseau qui relie les machines vont faire une entrée fracassante dans l'étude du coût des calculs. Dans un cluster donné, on peut facilement utiliser un nombre i (variable) de nœuds. Dans le modèle « cloud », la taille de la machine est alors simplement proportionnelle à i.

Un algorithme de tri distribué pourrait fonctionner de la façon suivante. On utilise le nombre minimal de nœuds qui est suffisant pour contenir les données en mémoire ; chaque nœud commence par trier ses propres données ; les nœuds communiquent et s'échangent des portions triées ; chaque nœud fusionne les portions reçues et les stocke localement. À la fin du calcul, si $i < j$, alors les entiers en possession du nœud i sont tous plus petits que ceux du nœud j (et ils en ont à peu près le même nombre). Lors de la phase de communication, chaque nœud communique avec tous les autres, ce qui stresse sévèrement le réseau de communication.

Le temps d'exécution de la procédure est $T = T_{tri} + T_{comm} + T_{merge}$. Dans la mesure où tous les nœuds effectuent la première et la troisième étape en parallèle sur un volume de donnée comparable, ce temps ne dépend pas du nombre de nœuds utilisé. Seul le temps passé dans la phase de communication est susceptible de dépendre du nombre de nœuds. Le coût du tri est donc :

$$Cost = n \cdot (T_{tri} + T_{merge}) + n \cdot T_{comm}(n)$$

La question « quel est le coût du tri ? » devient en fait « combien de temps va mettre le réseau de ma machine pour faire en sorte que chacun des P nœuds transmette N/P octets à chacun des autres ? ».

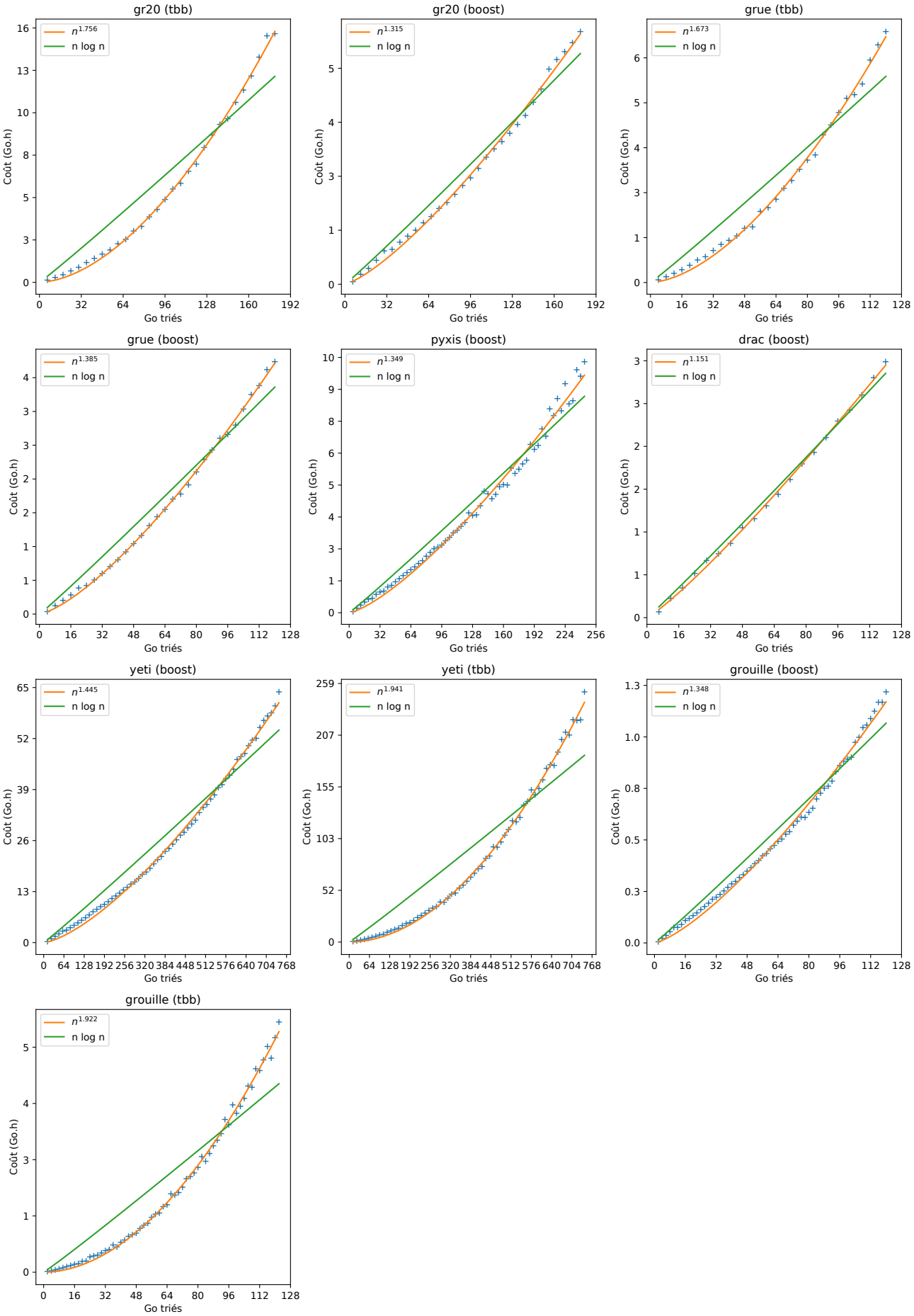


FIGURE 3.8 – Coût du tri dans un seul serveur de calcul.

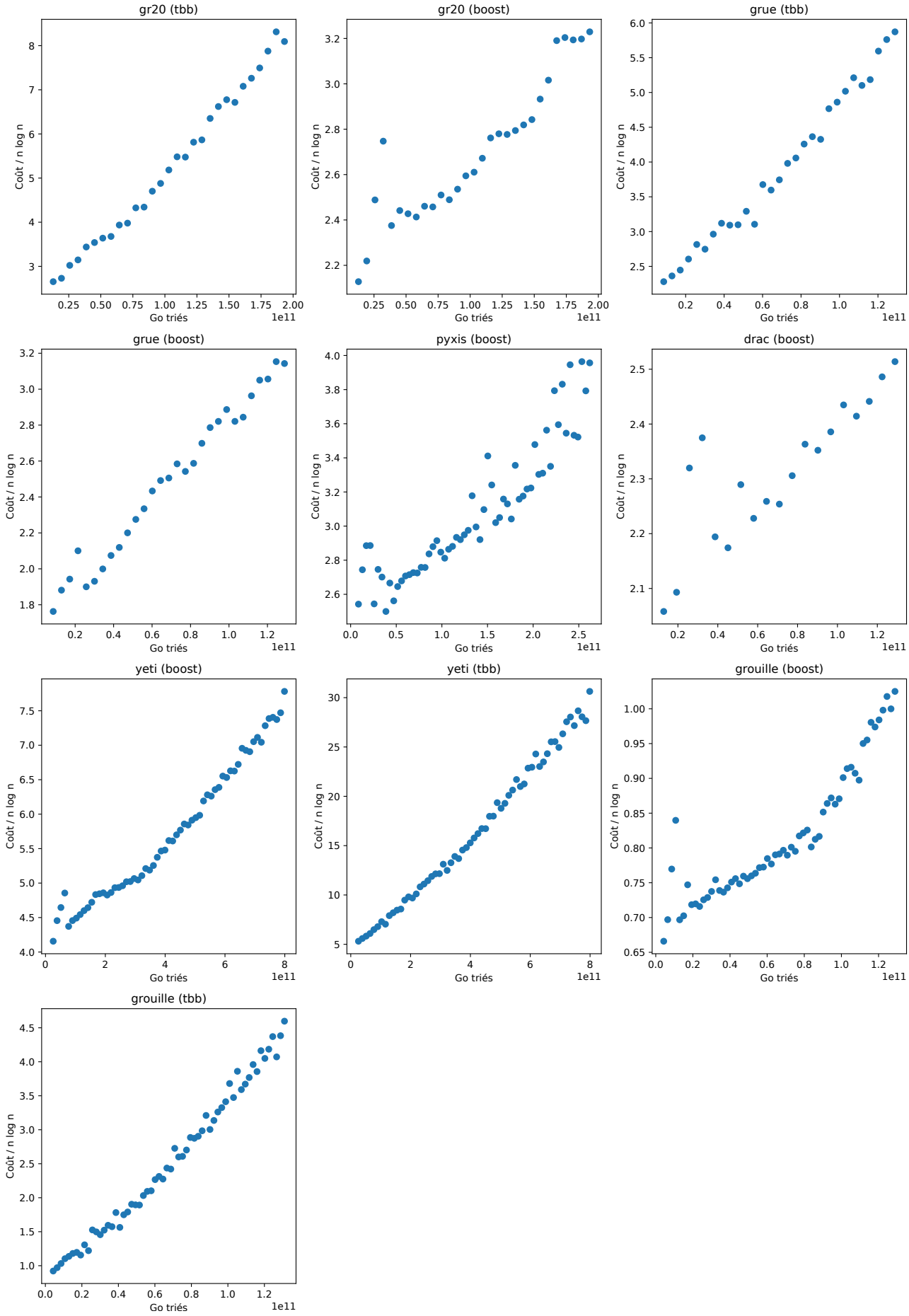


FIGURE 3.9 – « Résidus »— Coût divisé par $n \log n$.

Cluster	Code de tri	Exposant du coût	p-valeur
grue	TBB	1.673	0.0167
	Boost	1.385	0.0070
grouille	TBB	1.922	0.0004
	Boost	1.348	0.0019
grvingt	TBB	1.756	0.0149
	Boost	1.315	0.0127
yeti	TBB	1.941	0.0003
	Boost	1.445	0.0004
drac	Boost	1.151	0.0143
pyxis	Boost	1.349	0.0015

FIGURE 3.10 – Exposant observés empiriquement pour le coût du tri interne sur différentes architectures — le tri « coûte » n^α, pour les valeurs de α contenues dans la troisième colonne. La colonne p-valeur indique le résultat du test

J'ai mené une première expérience sur les 124 nœuds (avec 96Go de RAM) du cluster `gros` de Grid'5000. Ils sont reliés par un lien 25Gb ethernet à un *switch*[7]. Chaque nœud génère $N = 32$Go d'entiers de 64 bits aléatoires. Pour simuler la phase de communication, chaque nœud envoie et reçoit $32/p$ Go vers/depuis chaque autre nœud en utilisant la fonction `MPI_Alltoall` de la bibliothèque OpenMPI (p désigne le nombre de nœuds). La bande-passante effective du réseau (entre deux nœuds) est de 2480Mo/s. Le volume de données qui doit entrer dans chaque nœud donne une borne inférieure sur le temps d'exécution de $T_{\min} = 13.2\frac{p-1}{p}$ secondes.

Les résultats sont visibles figure 3.11. Il ne semble pas facile de plaquer un modèle très clair là-dessus. Les variations observées peuvent dépendre de la bibliothèque MPI utilisée comme du *switch* du réseau. Le volume total de données transporté par seconde augmente à peu près linéairement jusqu'à 55Go/s à 105 nœuds (ce qui explique le palier observé), puis il augmente plus lentement.

3.6.4 Problèmes méthodologiques

Pour essayer de creuser un peu, j'ai chronométré une autre opération soumise à une borne $AT \geq n^{1.5}$: la rotation. Les nœuds sont disposés sur un anneau ; chacun envoie 32Go de données à son successeur, et reçoit 32Go de son prédécesseur. Les résultats sont visibles figure 3.12.

Au-delà de 100 nœuds, les performances baissent un peu. En tout cas, ceci ne permet pas du tout d'affirmer que la rotation « coûte » $\Omega\left(n^{1.5}\right)$ sur ce cluster ; on observe plutôt un coût en $\mathcal{O}(n)$ — le temps est constant lorsque la taille de la machine augmente.

En fait, ceci expose une difficulté méthodologique délicate. Les clusters concrets sont équipés d'un réseau de communication *correctement dimensionné*, c'est-à-dire capable de bonnes performances même lorsque l'intégralité du cluster est utilisée. Et on ne peut pas artificiellement faire diminuer la taille du réseau ou les performances d'un commutateur (surtout dans le cas de `gros` où tous les nœuds sont reliés au même *switch*).

Ce problème peut se contourner sur certaines machines où la « taille du réseau » est proportionnelle au nombre de nœuds utilisés ; c'est le cas des machines où le réseau est un *mesh* ou un tore à plusieurs dimensions. Certaines machines de HPC très haut de gamme ont un réseau de ce style. Le problème c'est qu'elles sont rares et qu'il est difficile d'y avoir accès directement pour mener de petites expériences.

3.6.5 Tri sur IBM BlueGene/Q

Par chance, j'ai pu exécuter une simulation de la phase de communication du tri sur 4096 nœuds de `turing`, une IBM BluGene/Q située à l'IDRIS, et mise au rebut en septembre 2019. Chaque nœud

7. Les connaisseurs noteront qu'il y a *deux* interfaces réseau ; chacune est reliée à un *switch* différent. En fait, la deuxième est désactivée par défaut ; je l'ai laissée telle quelle.

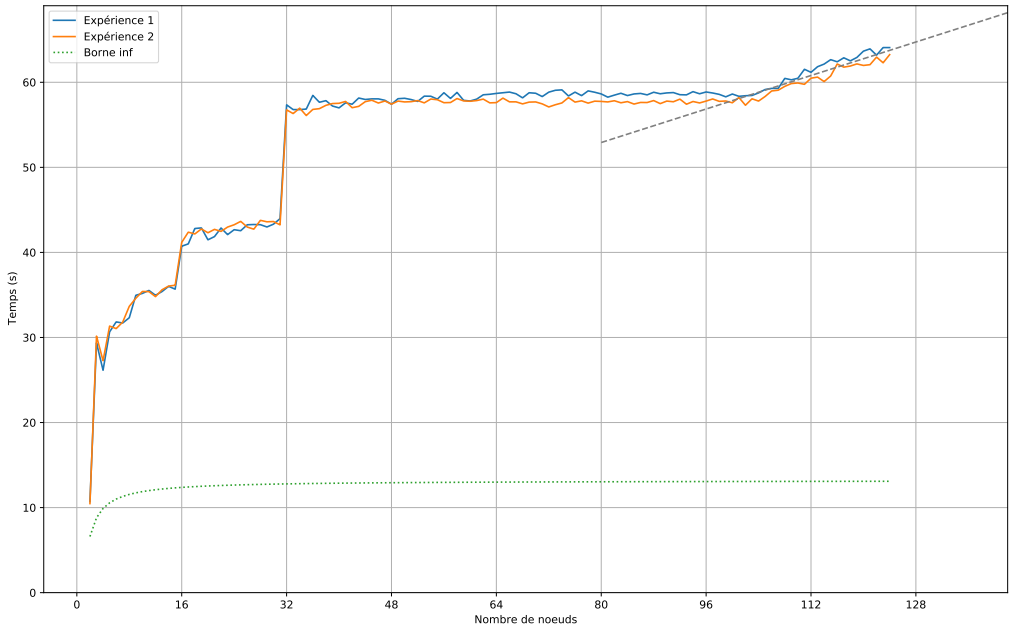


FIGURE 3.11 – Temps d'exécution de la phase de communication d'un tri sur le cluster `gros` de Grid'5000. Chaque nœud possède 32Go de données.

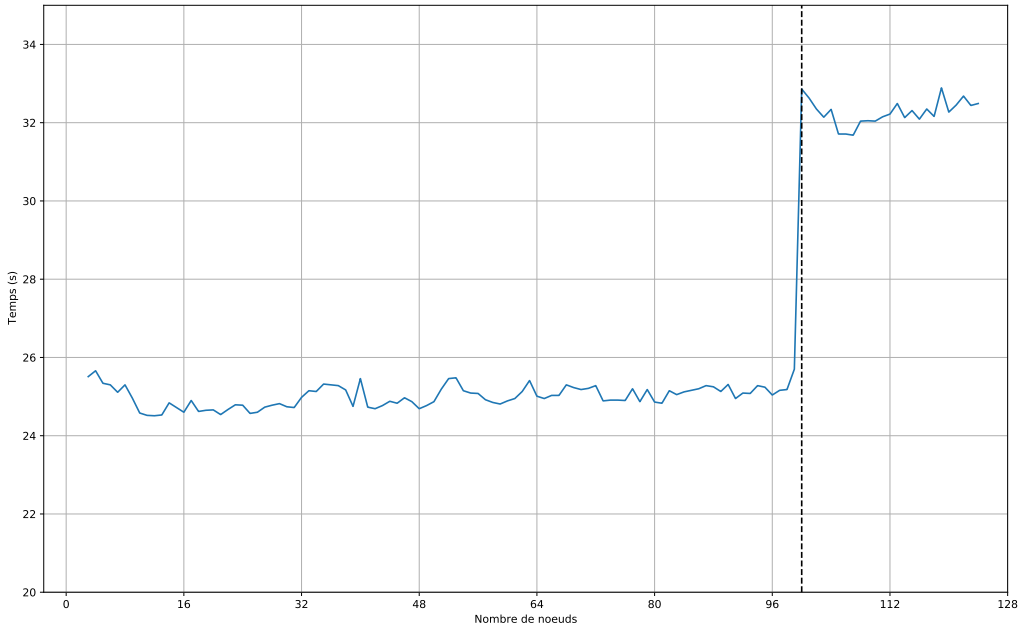


FIGURE 3.12 – Temps d'exécution de la « rotation » sur le cluster `gros` de Grid'5000. Chaque nœud envoie/reçoit 32Go. L'augmentation brutale (inexpliquée) a lieu à partir de 100 nœuds.

possède 16Go de RAM. Les nœuds sont reliés entre eux par un réseau « maison » très hautement performant, avec une topologie en tore 5D : chaque nœud a un lien à 2Go/s vers chacun de ses 10 voisins. La bibliothèque MPI, dérivée de MPICH2, est optimisée pour le réseau.

Chaque nœud traite 6.75Go de données. Il n'est possible que d'allouer un nombre de nœud qui est une puissance de deux, avec un maximum de 4096, ce qui permet de simuler le tri de $2^{41.75}$ entiers 64-bits (27.5To). Le tableau 3.13 montre les résultats. Il s'agit d'une machine de HPC de très bonne facture : le réseau transporte 1.5To/s en utilisant tous les nœuds disponibles.

Là encore, il n'est pas facile de plaquer un modèle sur ces données. Le nombre de points est trop faible, et il est bien difficile de distinguer $n \log_2 n$ de $n^{1.13}$ (c'est l'exposant qui colle le mieux).

Il est donc difficile de confirmer ou d'infirmer la borne inférieure théorique d'un coût en $N^{4/3}$. D'abord, la BlueGene/Q, avec son réseau en tore 5D, n'est-elle pas plutôt concernée par une borne en $N^{6/5}$? Mais dans ce tore, la taille de la 5ème dimension est limitée à deux, donc n'est pas plutôt un tore à 4 dimensions ? Dans tous les cas, la machine triche, car elle a de « longs fils ».

En tout cas, ceci montre bien que le modèle « Cloud » ne convient pas à cette machine : le temps que mettent les nœuds à transmettre leurs volume (constant) de données augmente avec le nombre de nœuds ; c'est normal car le réseau n'est pas entièrement connexe.

3.6.6 Tri sur `fugaku`

Mistuhisa Sato et Masahiro Nakao ont eu la gentillesse d'exécuter une petite expérience sur `fugaku` (la machine actuellement la plus puissante du monde) pour la préparation de ce manuscrit. Ils ont chronométré la fonction `MPI_Alltoall` sur des volumes de données et un nombre de nœuds variables. Ceci permet d'inférer le coût d'un tri où chaque nœud envoie et reçoit 12Mo. Jusqu'à 144Go sont ainsi triés avec 12 288 nœuds. Le tableau 3.6.6 montre les résultats. Il s'agit là encore d'une machine de HPC de classe mondiale : le réseau transporte 5To/s.

La machine a un réseau en tore 3D, donc on s'attend à voir apparaître un coût en $N^{4/3}$, et c'est bien ce qu'on observe. Dans ce cas précis, l'hypothèse sur le coût du tri semble assez bien vérifiée, mais encore une fois $N^{4/3}$ et $N \log N$ sont toujours assez proches.

3.6.7 Coût de la FFT

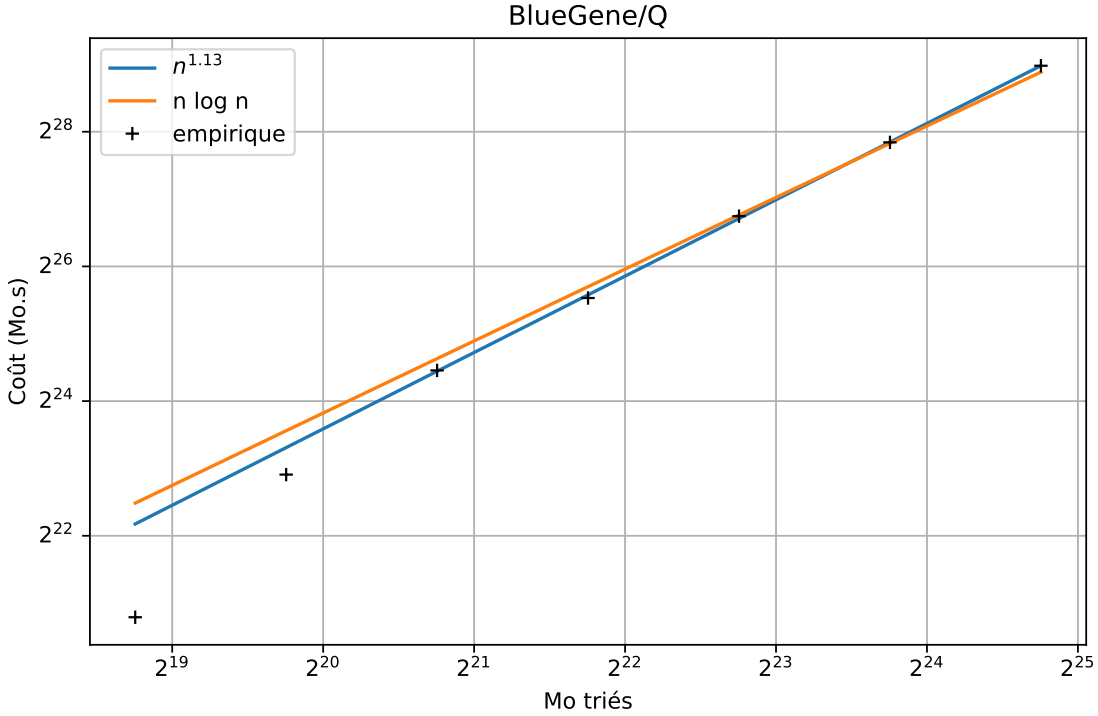
> In this paper we present scaling results of a FFT library [...] using 65536 cores of Blue Gene/P and 196608 cores of Cray XC40 supercomputers. We observe that communication dominates computation, more so on the Cray XC40.
>
> Abstract de [72], 2018

Pour bénéficier des tests effectués aux plus grandes échelles possibles, on peut utiliser les mesures de performances de la transformée de Fourier rapide menées sur les plus grandes machines du monde. En effet, cela expose à peu près le même motif de communication que le tri (tout le monde doit communiquer avec tout le monde).

La Transformée de Fourier Discrete (DFT) d'un tableau X de n nombres complexes est le tableau Y de taille n défini par

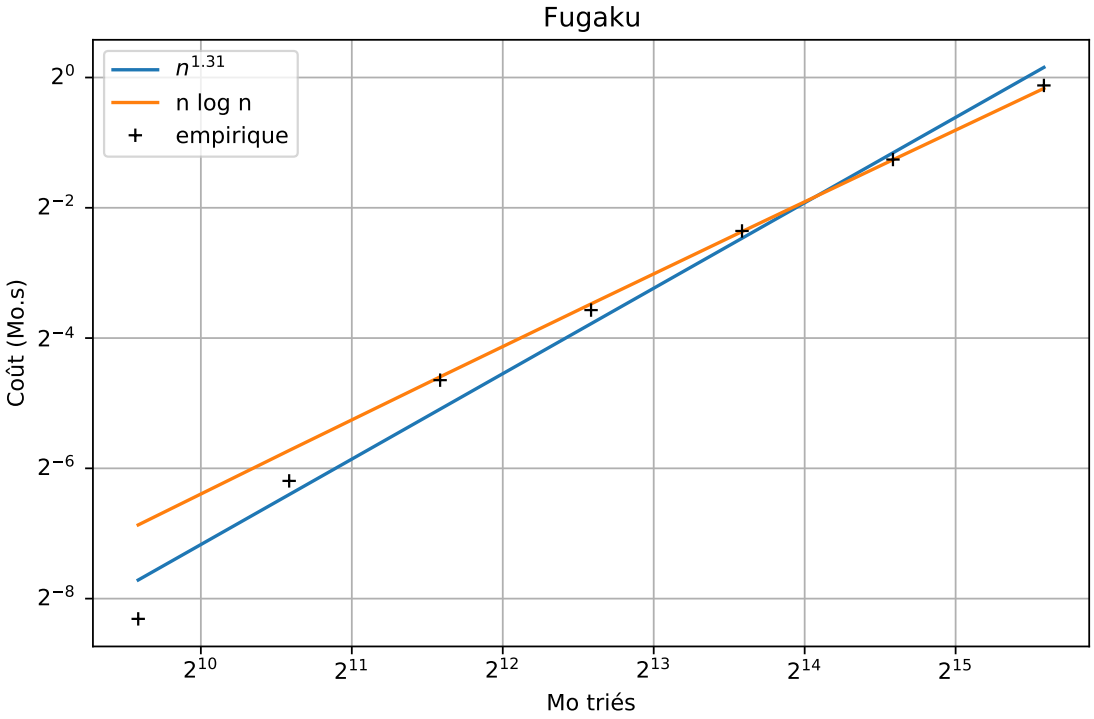
$$Y[k] = \sum_{j=0}^{n-1} X[j]\omega_n^{jk}, \qquad \text{avec} \qquad \omega_n = e^{-\frac{2i\pi}{n}}$$

Lorsque la taille se factorise en $n = n_1 \times n_2$, alors on peut poser $j = j_1 n_2 + j_2$ et $k = k_1 + k_2 n_1$ et ceci



nœuds	Donnnées (Go)	Temps (s)	dimension grille / tore
2	13.5	2.7	2
4	27	2.9	2×2
8	54	7.0	$2 \times 2 \times 2$
16	108	9.5	$4 \times 2 \times 2$
32	216	12.5	$2 \times 4 \times 2 \times 2$
64	432	4.1	$2 \times 2 \times 4 \times 2 \times 2$
128	864	8.9	$4 \times 4 \times 4 \times 2$
256	1728	13.0	$2 \times 4 \times 4 \times 4 \times 2$
512	3456	13.7	$4 \times 4 \times 4 \times 4 \times 2$
1024	6912	15.9	$4 \times 4 \times 4 \times 4 \times 2$
2048	13824	17.0	$4 \times 4 \times 8 \times 8 \times 2$
4096	27648	18.7	$8 \times 4 \times 8 \times 8 \times 2$

FIGURE 3.13 – Temps nécessaire à l'exécution de la phase de communication d'un tri simulé. La colonne "Données" indique la quantité totale d'entiers 64 bits triés, et donc véhiculés sur le réseau. Les mauvaises performances obtenues pour $8 \leq n \leq 32$ sont peut-être causées par le fait que ce sont de « petites tâches » qui sont mal adaptées à la topologie du réseau. Le réseau est toujours torique sur la dernière dimension, qui est toujours de taille 2. Sur les autres dimensions, c'est un tore s'il y a au moins 4 rangs, sinon c'est une grille.



nœuds	Données	Dimension tore	Temps (μs)
12	144M	$2 \times 3 \times 2$	1483
24	288M	$2 \times 3 \times 4$	1691
48	576M	$4 \times 3 \times 4$	1795
96	1.1G	$4 \times 6 \times 4$	1985
192	2.3G	$4 \times 6 \times 8$	2691
384	4.5G	$4 \times 6 \times 16$	5757
768	9G	$8 \times 6 \times 16$	6139
1536	18G	$8 \times 12 \times 16$	6343
3072	36G	$16 \times 12 \times 16$	7132
6144	72G	$16 \times 24 \times 16$	7790
12288	144G	$16 \times 24 \times 32$	28241

FIGURE 3.14 – Temps nécessaire à l'exécution de la phase de communication d'un tri simulé sur `fugaku`. La colonne "Données" indique la quantité totale d'entiers 64 bits triés, et donc véhiculés sur le réseau.

peut se réécrire :

$$Y[k_1 + k_2 n_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2] {\omega_{n-1}}^{j_1 k_1} \right) {\omega_n}^{j_2 k_1} \right] {\omega_{n_2}}^{j_2 k_2}.$$

Ceci donne un algorithme pour le calcul de la DFT de taille $n_1 \times n_2$: on calcule n_2 DFT de taille n_1 (la somme interne), on multiplie par les *twiddle factors* ${\omega_n}^{j_2 k_1}$ puis on effectue n_1 DFT de taille n_2 (la somme externe). On peut alors recommencer récursivement. Ceci donne l'algorithme bien connu de la transformée de Fourier rapide (FFT), aussi appelé algorithme de Cooley-Tuckey. Il se présente sous de nombreuses variantes (itératives, récursives, ...). L'un des choix les plus communément présenté consiste à poser $n_2 = 2$ (« *Radix-2 Decimation in Time* »). Sur des architectures vectorielles et/ou parallèles, de plus grandes valeurs de n_1 et/ou n_2 sont courantes.

Une difficulté importante est que la dimension n_1 correspond à des positions j_1 non-contiguës en mémoire pour X mais à des positions contiguës k_1 pour la sortie Y, et vice-versa pour n_2. Ceci limite l'efficacité des accès à la mémoire, et c'est un des problèmes majeurs de l'implantation efficace de la FFT, même sur un seul cœur (si l'on en croit les spécialistes [112]).

Une astuce consiste à utiliser $n_1 \approx n_2$; la FFT peut alors se calculer de la façon suivante (« *six-step FFT* ») :

1. Transposer les données, considérées comme une matrice $n_1 \times n_2$, en une matrice $n_2 \times n_1$.
2. Effectuer une FFT à une dimension sur chacune des n_2 lignes de taille n_1.
3. Multiplier par les *twiddle factors* (A_{ij} est multiplié par ω_n^{ij}).
4. Transposer la matrice $n_2 \times n_1$ en une matrice $n_1 \times n_2$.
5. Effectuer une FFT à une dimension sur chacune des n_1 lignes de taille n_2.
6. Transposer la matrice $n_1 \times n_2$ en une matrice $n_2 \times n_1$.

L'avantage de cette stratégie est que les FFTs effectuées dans les étapes 2 et 5 opèrent sur des données contiguës, et peuvent être réalisées en parallèle. On le voit sur la figure 3.15.

Plusieurs articles étudient le coût de la FFT (complexe, double-précision) sur de grosses machines équipées d'un réseau en tore 3D. Les auteurs étudient (entre autre) le *weak scaling*, ce qui est exactement ce qu'il nous faut : ils calculent des FFTs de taille $\mathcal{O}(n)$ avec n nœuds — chaque nœud possède donc une quantité constante de données. La « taille de la machine » augmente donc avec la taille des données et il n'y a pas d'équipement réseau de taille fixe. Les auteurs rapportent les résultats en GFLOPS (communauté HPC oblige), ce qui n'est pas très pratique car nous aurions besoin du temps d'exécution. Pour un tableau de taille totale N, le nombre total de FLOPS est estimé à $5N \log_2 N$, ce qui permet d'inférer les temps de calcul et les coûts.

Dans ce contexte, le nombre d'opérations arithmétiques que chaque nœud doit effectuer augmente en $\Theta(\log n)$, où n est le nombre de nœuds, et le temps passé dans les communications sur un tore 3D augmente en $\Theta\left(n^{1/3}\right)$. Il faut noter qu'avec un réseau en *fat tree*, le temps passé dans les communications devrait rester constant. Dans tous les cas, toutes les études sur FFT sur de grandes machines concluent que le temps passé dans les communications domine le temps passé à faire des calculs. Par contre, les études se concentrent sur la somme des deux (c'est le temps passé à effectuer la FFT), et ne se concentrent pas forcément sur la phase de communication en tant que telle.

Dans l'article « *Optimization of Fast Fourier Transforms on the BlueGene/L Supercomputer* » [175], des mesures de performances sont présentées pour une IBM BlueGene/L à 65536 nœuds. Chaque nœud possède 512Mo de RAM, et traite 4 millions d'éléments du tableau (64 Mo pour X et autant pour Y). Le tableau 3.16 montre les résultats, et encore une fois il n'est pas facile de conclure franchement. Pour essayer d'y voir plus clair, on a calculé les valeurs optimales de α et β telles que $\alpha n \log_2 n$ et $\beta n^{4/3}$ collent le mieux possible aux coûts observés *jusqu'à 16384 nœuds*. On voit que $\beta n^{4/3}$ prédit mieux le coût pour 65536 nœuds que $\alpha n \log_2 n$. La figure montre qu'il est bien difficile de décider si la série de points appartient aux courbes $\Theta(n \log n)$, $\Theta\left(n^{4/3}\right)$ ou encore $\Theta\left(n^{1.2}\right)$.

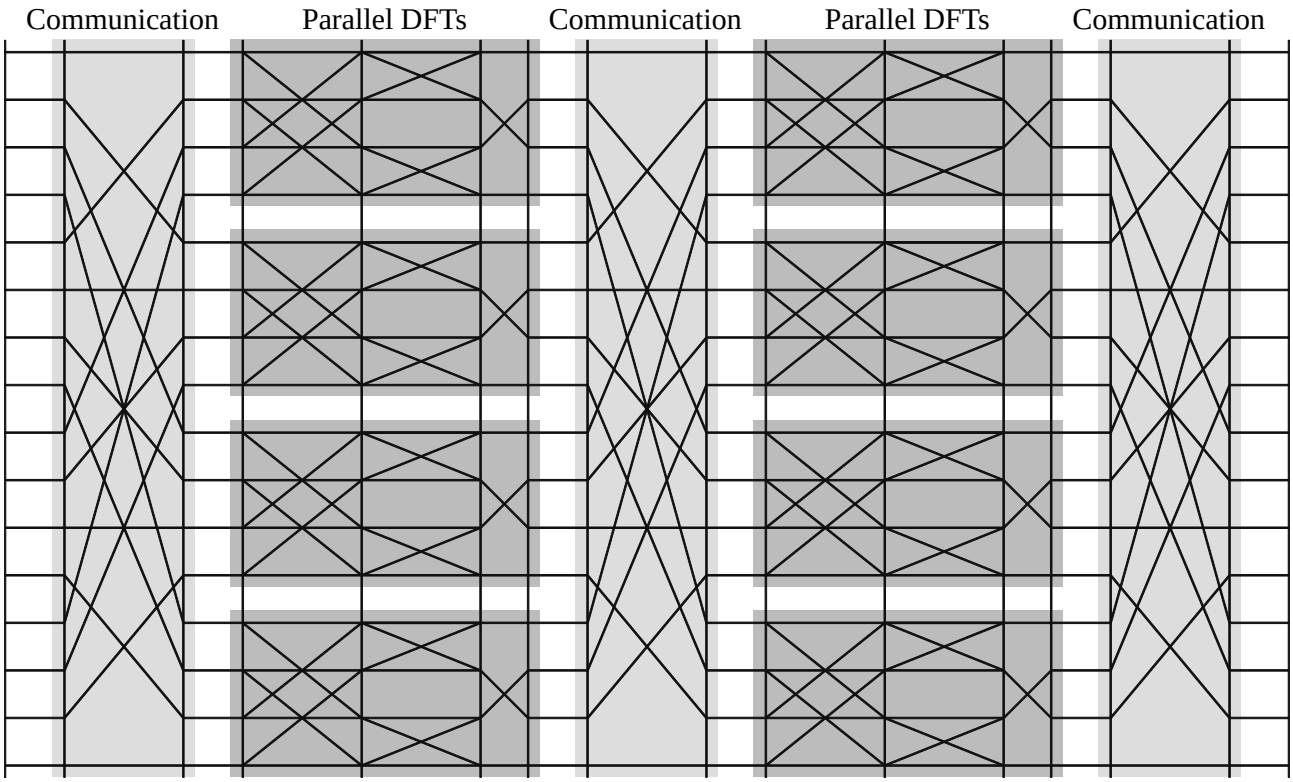


FIGURE 3.15 – Illustration de l'algorithme « *six-step FFT* ». Image :[108].

L'article « *Automatic Generation of the HPC Challenge's Global FFT Benchmark for BlueGene/P* » [108] présente aussi des résultats de performances sur une machine de la génération d'après, une IBM BlueGene/P, avec 32 768 nœuds en tore 3D. Chaque nœud possède 2Go de RAM, et traite un tableau de taille 2^{24} (512Mo pour X et autant pour Y). Après un peu de retro-conception sur la figure 5 de l'article en question, on peut obtenir les données du tableau 3.17. Il est encore difficile de conclure de manière affirmative. Les deux modèles ajustés sur tous les points sauf le dernier surestiment le coût de la plus grosse instance, mais le modèle en $n \log n$ approche plus près de la vraie valeur.

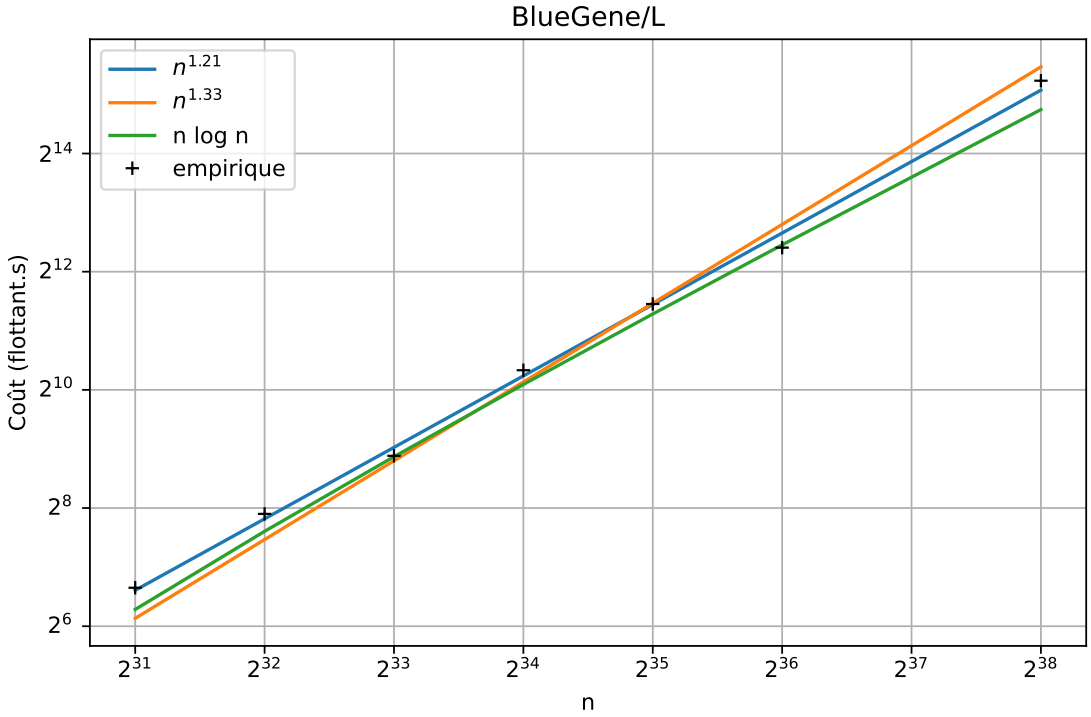
3.6.8 Conclusion

Il est bien malaisé de conclure, car un coût en $n \log n$ et un coût en $n^{4/3}$ sont finalement assez proches. Les deux courbes sont de forme proche, et en ajustant les constantes on peut donc à peu près les superposer sur de petits intervalles. Mis à part pour le tri interne où le verdict est sans appel, il est difficile de confirmer ou d'infirmer, en pratique sur de grandes machines, l'hypothèse que le tri ou la FFT coûtent $N^{4/3}$ plutôt que $N \log N$. Ceci dit, les raisonnements sur les réseaux de communication prouvent qu'ils coûtent asymptotiquement $N^{4/3}$ sur des machines en tore 3D, et ceci n'est visiblement pas contredit par l'expérience.

3.7 Comment raisonner dans le modèle du coût ?

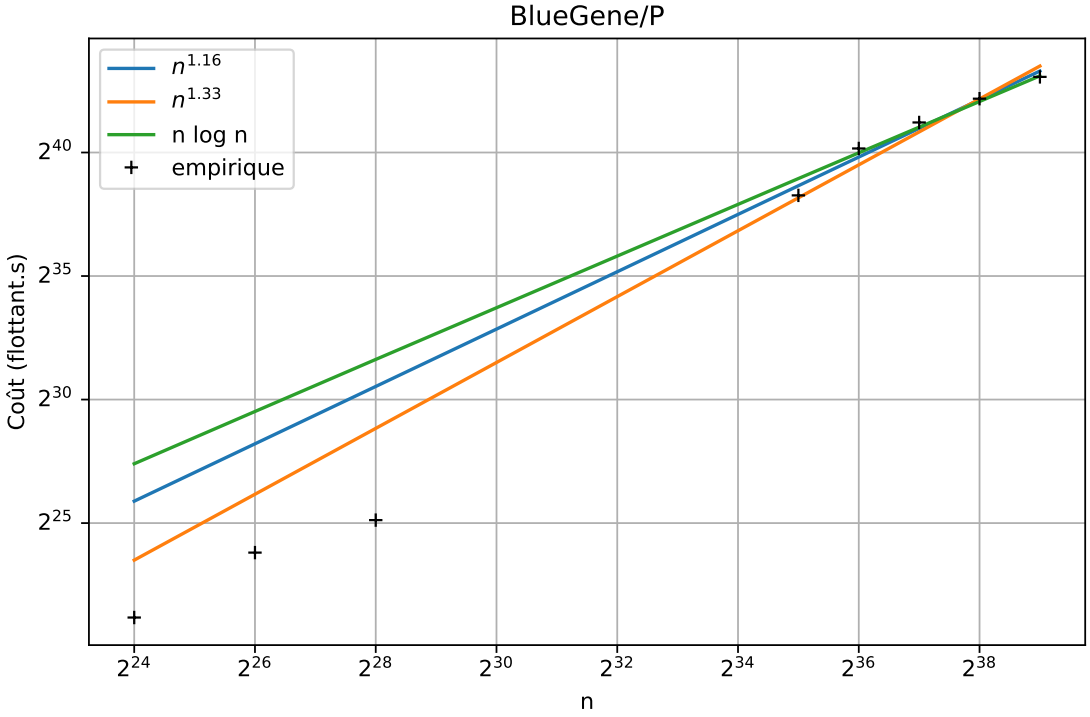
Comment estimer le coût de l'exécution d'un algorithme ? Discuter du coût nécessite de discuter d'une *machine* capable de l'exécuter ; le coût est celui de la combinaison algorithme + machine.

Néanmoins, on peut malgré tout dire quelques généralités. Si une tâche peut se découper en sous-tâches indépendantes (c'est typiquement le cas de la recherche exhaustive), alors il suffit d'estimer le coût de traitement d'une sous-tâche. En effet, supposons qu'il faut faire N fois la même chose, et que ceci prenne un temps T sur une machine de taille M. Le coût unitaire est donc TM. On utilise une meta-machine capable de traiter k instances en parallèle ; cette machine est de taille kM. Le coût total



$\log_2 n$	Taille	GFLOPS	T (s)	Coût (Go-h)	$\alpha n \log_2 n$	$\beta n^{4/3}$
9	16Go	53	6.28	0.028	0.0216	0.0156
10	32Go	92	7.44	0.066	0.0541	0.0394
11	64Go	192	7.38	0.131	0.1299	0.0992
12	128Go	290	10.07	0.358	0.3030	0.2500
13	256Go	549	10.92	0.779	0.6926	0.6300
14	512Go	1166	10.60	1.509	1.5584	1.5876
16	2048Go	2779	18.79	10.69	7.6189	10.0808

FIGURE 3.16 – Coût du calcul de la FFT sur une BlueGene/L (réseau en tore 3D $64 \times 32 \times 32$). Données tirées de [175]. Les constantes sont $\alpha = 0.0003382$ et $\beta = 0.0003876$. La figure montre également la droite de pente 1.209, car c'est celle qui correspond le mieux.



$\log_2 n$	Taille (n)	GFLOPS	T (s)	Coût (Go-h)	$\alpha n \log_2 n$	$\beta n^{4/3}$
0	0.5Go	14	0.14	-	-	-
2	2Go	40	0.22	0.00012	0.0008	0.0007
4	8Go	276	0.14	0.00030	0.0095	0.0042
11	1To	625	9.61	2.73	4.0514	2.7068
12	2To	690	17.92	10.2	8.9130	6.8208
13	4To	1367	18.59	21.1	19.4466	17.1874
14	8To	2884	18.10	41.2	42.1344	43.3095
15	16To	6400	16.69	76	90.7510	109.1330

FIGURE 3.17 – Coût du calcul de la FFT sur une BlueGene/P (réseau en tore 3D). Données tirées de [108]. Les constantes sont $\alpha = 0.00039564$ et $\beta = 0.00026226$.

est donc :

$$Cost = [\text{Temps}] \times [\text{Taille machine}] = \frac{NT}{k} \times kM = NTM.$$

Ceci est précisément le coût unitaire multiplié par le nombre de sous-tâches à traiter, indépendamment du degré de parallélisme avec lequel on les traite. Atteindre cette situation est souhaitable, car le coût varie alors proportionnellement au nombre de tâches à traiter

Si on a affaire à des algorithmes plus compliqués (tri, produit, FFT, etc.), une possibilité pour raisonner consiste à utiliser les algorithmes conçus pour des architectures de type « mesh 3D », et donc le coût est connu.

Enfin, si on essaye d'analyser le coût d'un algorithme présenté de manière séquentielle, nécessitant T opérations et M unités de mémoire, la solution consciencieuse consiste à en décrire une implantation parallèle puis à étudier son coût. En effet, la discussion de la section 3.1 implique qu'il faut que l'algorithme et la machine sur laquelle on l'exécute soient tous les deux parallèles, sinon le coût résultat sera égal au produit TM (temps $\times$ mémoire), et sera donc ridiculement élevé.

Quand on commence à discuter de machines parallèles, le théorème 1 peut intervenir. Il s'applique aux *machines* utilisées, et pas aux *algorithmes* dont on souhaite étudier le coût et encore moins au *problèmes calculatoires* qu'on essaye de résoudre. Il faut donc l'utiliser avec une certaine prudence, et discuter du choix de la machine qu'on utilise pour évaluer le coût.

Supposons qu'on ait affaire à un algorithme présenté séquentiellement qui nécessite $\mathcal{O}(T)$ opérations en tout, dont T accès à une mémoire de taille M. On peut *supposer* que son exécution parallèle nécessite une machine où chaque processeur peut accéder à n'importe quelle case mémoire de manière aléatoire. Le théorème 1 donne alors un coût de $TM^{1/3}$ si le calcul est suffisamment parallélisable.

Mais il y a des situations où le caractère structuré des accès mémoires permet de « battre » le théorème et d'obtenir un coût inférieur. C'est par exemple le cas si tous les processeurs accèdent à la même adresse simultanément en permanence. Dans ce cas, une machine de taille plus faible (cf. figure 3.18) peut faire l'affaire. Dans ce cas-là, le coût peut descendre à $\mathcal{O}(T)$. Cette situation se présente concrètement dans les implantations de la recherche exhaustive pour les polynômes quadratiques que j'ai programmées [41]. Cette observation précise permet d'obtenir de bonnes performances sur GPU.

Ceci suggère qu'un paramètre déterminant est donné par le nombre d'adresses mémoires distinctes auxquelles l'ensemble des processeurs accèdent simultanément (ça fixe la taille minimale du « pont » dans la figure 3.18). Il pourrait être intéressant d'essayer de prouver une version raffinée du théorème de Wiener, qui soit paramétrée par une distribution de probabilité sur les adresses mémoire auxquelles les processeurs accèdent.

Au passage, de nombreux algorithmes classiques (tri, FFT, ...) effectuent des accès prévisibles à la mémoire selon un motif structuré, et donc l'hypothèse que chaque processeur doit être capable d'effectuer des accès mémoires aléatoires ne semble pas nécessaire. Mais pourtant, toutes les machines connues pour effectuer le tri ou la FFT coûtent $N^{4/3}$, exactement comme ce que prédit le théorème 1.

Alternativement, il y a d'autres situations où on ne peut pas atteindre le coût annoncé par le théorème 1, car celui-ci discute de machines parallèles, or certains problèmes peuvent avoir une dimension intrinsèquement séquentielle. On peut par exemple penser à une variante de Scrypt [168] : partant d'une chaîne de bits x, initialiser $A[0] \leftarrow H(x)$ puis répéter $x \leftarrow H(x)$ et $A[i+1] \leftarrow x$ pour $0 \leq i < M-1$. Ensuite, répéter T fois : $j \leftarrow x \bmod M$ et $x \leftarrow H(x \oplus A[j])$. Enfin, renvoyer x. Le coût est TM et la sécurité de la fonction consiste en ce qu'on ne peut pas le réduire.

Pour conclure, le théorème 1 ne donne *ni* automatiquement une borne inférieure sur le coût de l'exécution d'un algorithme, *ni* automatiquement une borne supérieure.

Certains algorithmes sont étudiés pour effectuer des accès mémoire à une *fréquence réduite*. Les algorithmes basés sur l'utilisation de points distingués rentrent typiquement dans cette catégorie. Il faut utiliser le théorème 1 et déterminer le nombre optimal de processeurs qui minimise le coût. Alternativement, on peut obtenir le même résultat en considérant un *mesh 3D* de la bonne taille.

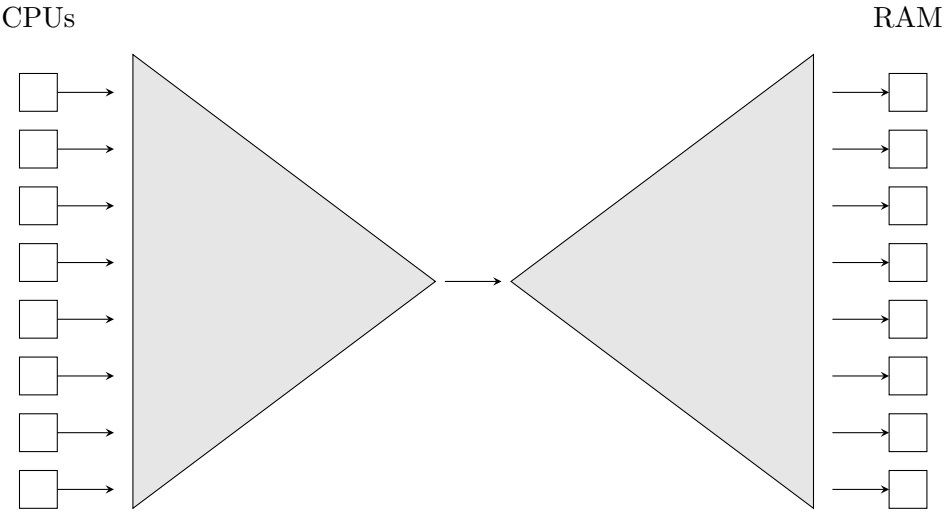


FIGURE 3.18 – Une machine où p processeurs accèdent de manière synchrone à la même case mémoire en permanence. Un arbre binaire complet relie les p processeurs au « pont » du milieu, relié aux M cases mémoire par un autre arbre binaire complet. Le pont ne constitue pas un goulet d'étranglement, car tous les processeurs accèdent à la même case mémoire. La taille de la machine, y compris les câbles, est $\mathcal{O}(P + M)$, ce qui bat le théorème 1.

Des algorithmes peuvent combiner des accès coûteux à une grosse mémoire avec des calculs encore plus coûteux. Par exemple : une phase de *meet-in-the-middle* qui effectue $2^{n/2}$ accès à une mémoire de taille $2^{n/2}$ est suivie d'une recherche exhaustive sur $3n/4$ bits. Le coût de la première phase est alors $2^{2n/3}$, et il est dominé par le coût de la deuxième phase qui est $2^{3n/4}$. Le coût est alors égal au nombre d'opérations. En réalité, deux machines différentes pourraient (et devraient) être utilisées pour les deux phases. Par conséquent, il vaut mieux décomposer l'algorithme en étapes « atomiques » puis évaluer leur coût séparément (mais attention aux transferts de données...).

Cependant, l'idée principale contenue dans le théorème 1, qu'on pourrait résumer en disant « coût $TM^{1/3}$ », même si elle ne marche pas à tous les coups, permet d'attirer l'attention sur des situations « intéressantes ». Lorsque le produit $TM^{1/3}$ atteint ou dépasse 2^n, elle suggère un examen plus approfondi.

Parfois, l'algorithme peut être modifié pour réduire la quantité de mémoire nécessaire, et/ou réduire la fréquence des accès à la mémoire. Mais je ne connais aucun algorithme qui 1) nécessite une mémoire de taille M, 2) fonctionne en temps $\mathcal{O}(M)$ et 3) puisse être implanté sur une machine parallèle avec un coût inférieur à $M^{4/3}$. Cette constatation donne du poids au « modèle NIST » qui facture le coût des accès à la mémoire. Mais on verra dans la suite deux exemples qui prouvent que les modèles ne sont pas équivalents.

Les deux chapitres suivants discutent du coût d'algorithmes variés. Ces méthodes de raisonnement y sont mises en œuvre.

Chapitre 4

Cryptanalyse algorithmique dans le modèle du coût

Après avoir critiqué le modèle de la *Random Access Machine* et avoir fait la promotion d'un modèle alternatif, celui du coût, essayons d'examiner le coût de certains algorithmes qui jouent un rôle en cryptographie.

Ce chapitre commence par des composants algorithmiques récurrents dans les attaques cryptographiques : test d'appartenance à un gros dictionnaire, recherche de collisions et *Meet-in-the-Middle*, calcul des jointures. Il s'enchaîne sur le problème 3XOR, qui sert parfois de composant dans des attaques cryptographiques complètes. Enfin, les deux dernières sections discutent de deux problèmes algorithmiques qui jouent un rôle dans la sécurité de systèmes à clef publique « post-quantiques » : le décodage des codes linéaires aléatoires et la résolution de systèmes quadratiques multivariés modulo 2.

4.1 Test d'appartenance à un dictionnaire statique

En 1984, Fredman, Komlós et Szemerédi [109] ont proposé une structure de données capable de stocker un ensemble de N objets de taille constante dans une structure de données qui peut se construire en temps (probabiliste) $\mathcal{O}(N)$ et qui permet de réaliser le test d'appartenance en temps constant dans le pire des cas. Ceci repose sur l'utilisation de familles de fonctions de hachage universelles.

Au passage, cette solution est rarement utilisée dans du code concret, et d'autres techniques, comme le hachage avec sondage linéaire ou le *cuckoo Hashing* [166] sont beaucoup plus simples à mettre en œuvre, notamment car il n'est pas pratique de devoir gérer des familles de fonctions de hachage. De plus, le *cuckoo hashing* est vectorisable.

Un motif qui revient souvent dans la cryptanalyse est le suivant : construire un gros ensemble $Y \subset \{0,1\}^n$ constitué de valeurs aléatoires, puis trouver x tel que $f(x) \in Y$, où f est une fonction aléatoire. Du fait de son caractère aléatoire, il est nécessaire de *stocker* l'ensemble Y (et en plus on ne peut pas le compresser d'après les théorèmes classiques de Shannon).

Dans le modèle de la *Random Access Machine*, la question se règle en $T = 2^n/|Y|$ opérations. Quel est le coût correspondant ? La recherche de x peut être effectuée en parallèle sur de nombreux processeurs, à condition que ces derniers soient tous reliés à une mémoire qui contient Y. Le caractère aléatoire de f suggère que cela nécessite une machine capable de faire communiquer en permanence chaque processeur avec chaque élément de mémoire, donc le coût serait $2^n/|Y|^{2/3}$ d'après le théorème 1.

L'utilisation de fonctions de hachage entraîne quasi-inévitablement des accès aléatoires, donc l'utilisation du corollaire simple du théorème 1 semble justifiée. En tout cas, tenter d'implanter avec un ensemble Y qui ne tient pas dans la mémoire d'un seul nœud de calcul va vraisemblablement donner lieu à une répartition de l'ensemble Y entre les p nœuds de la machine parallèle, donc à la mise en place d'une table de hachage distribuée (un sujet très classique). Et du coup, tester si une valeur $f(x)$

quelconque appartient à Y va nécessiter l'envoi de messages sur le réseau. Par exemple, une manière de faire pourrait être d'utiliser une fonction de hachage H qui affecte un nœud de calcul à chaque chaîne de n bits. Chaque nœud de calcul stocke les éléments de Y qui lui sont affectés. Pour tester si $x \in Y$, on transfère x jusqu'au nœud $H(x)$, et ce dernier se débrouille pour vérifier si c'est bien l'une des valeurs qui sont sous sa responsabilité.

Considérons ce qui se passe avec un réseau en Mesh 3D composé de p nœuds de calcul disposant chacun d'une mémoire $|Y|/p$. Chaque nœud est capable d'émettre une valeur de $f(x)$ par unité de temps. Ceci sature-t-il le réseau ? Une des manières d'y réfléchir fait encore intervenir une notion standard du HPC, la « *bisection bandwidth* » : si on partitionne la machine en deux moitiés égales, la coupe est de surface $p^{2/3}$ dans le pire des cas, donc $p^{2/3}$ message peuvent la franchir par unité de temps au maximum. Cette machine de taille $|Y|$ résout le problème en temps $2^n/(|Y|p^{2/3})$, donc le coût optimal est atteint pour $p = |Y|$. C'est le même que celui qu'on avait trouvé au-dessus en appliquant le théorème 1 sans réfléchir.

Un autre raisonnement aboutit au même résultat. Acheminer chaque message nécessite $p^{1/3}$ étapes de routage. Si les p nœuds produisent les messages avec un débit de r messages par unité de temps, alors la loi de Little [152] affirme que le nombre moyen de messages « en vol » à un instant donné est égal à $pr \times p^{1/3}$. Comme le réseau ne peut contenir que $\mathcal{O}(p)$ messages en vol, il est nécessaire de limiter le débit de production à $r = p^{-1/3}$.

La capacité de transport du réseau est donc clairement le facteur limitant, et les processeurs ne doivent émettre qu'une valeur de $f(x)$ toutes les $p^{1/3}$ unités de temps. Ils doivent donc rester principalement... inactifs ! Dit autrement, la bonne machine possède un réseau ultra-rapide et des processeurs lents — c'est d'ailleurs les choix qu'IBM avaient fait lors de la conception des machines BlueGene pour le calcul scientifique.

4.2 Recherche de collisions et itération dans les graphes fonctionnels

On s'intéresse ici au problème de trouver une ou des collisions entre deux fonctions aléatoires $f_0, f_1 : \{0,1\}^n \to \{0,1\}^m$ (en toute généralité, les deux fonctions pourraient avoir des domaines différents, et on pourrait adapter les techniques décrites ici). Autrement dit, on cherche x et y tels que $f_0(x) = f_1(y)$. Ceci constitue l'attaque générique de base contre les fonctions de hachage cryptographiques ou bien contre le problème du logarithme discret sur des groupes n'offrant pas de structure exploitable.

Le problème est assez différent selon qu'on cherche une collision arbitraire entre f_0 et f_1, ou bien si l'on cherche *toutes* les collisions, par exemple parce qu'on en veut une spéciale.

On s'attend à ce qu'il y ait 2^{2n-m} collisions en tout, et les trouver peut se faire avec un algorithme naïf : pour tout $x \in \{0,1\}^n$, stocker $f_0(x) \leftarrow x$ dans une table de hachage ; ensuite pour tout $y \in \{0,1\}^n$, tester si $f_1(y)$ appartient à la table de hachage. Le cas échéant, ceci révèle une paire x, y. Le coût de cette technique est de $2^{4n/3}$. Cette section discute d'une technique non-triviale pour faire baisser ce coût, analysée par van Oorschott et Wiener au tournant de l'an 2000.

4.2.1 Graphe fonctionnels

Les techniques plus sophistiquées reposent sur l'itération de fonctions $\{0,1\}^k \to \{0,1\}^k$, donc sur des propriétés de leur *graphe fonctionnel* : c'est le graphe dont les nœuds sont les chaînes de k bits et où $x \to y$ lorsque $y = f(x)$. La figure 4.1 montre un graphe fonctionnel.

Dans un tel graphe, chaque composante connexe est constituée d'un cycle sur lequel des arbres sont greffés. Partant d'un nœud x, l'itération de f dessine un chemin qui se connecte nécessairement à un cycle. La *hauteur* de x est la distance qui le sépare du cycle, ou bien zéro s'il est sur le cycle. Un nœud est de *profondeur* i si on peut l'atteindre en itérant i fois la fonction.

Certaines propriétés asymptotiques des graphes fonctionnels sont bien connues, notamment après les travaux de Flajolet et Odlyzko [105]. Voici un petit résumé, en supposant une fonction aléatoire d'un

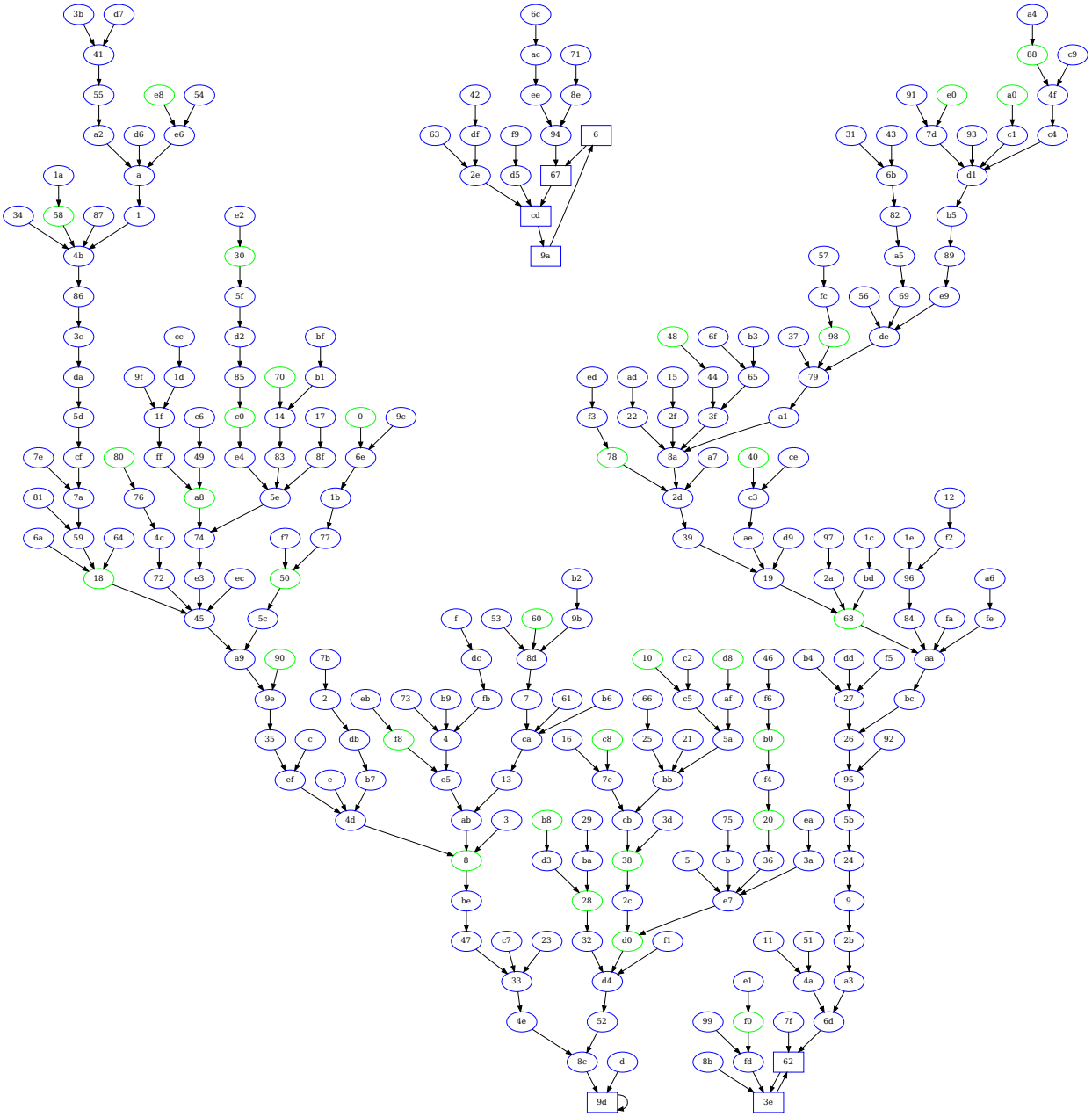


FIGURE 4.1 – Le graphe fonctionnel de SHA-256 tronqué à 8 bits. Les nœuds carrés appartiennent à un cycle. Les nœuds verts sont des « points distingués » qui ont les 3 bits de poids faible à zéro. Il y a trois composantes connexes et chacune d'entre elles est formée d'un cycle sur lequel des arbres sont greffés.

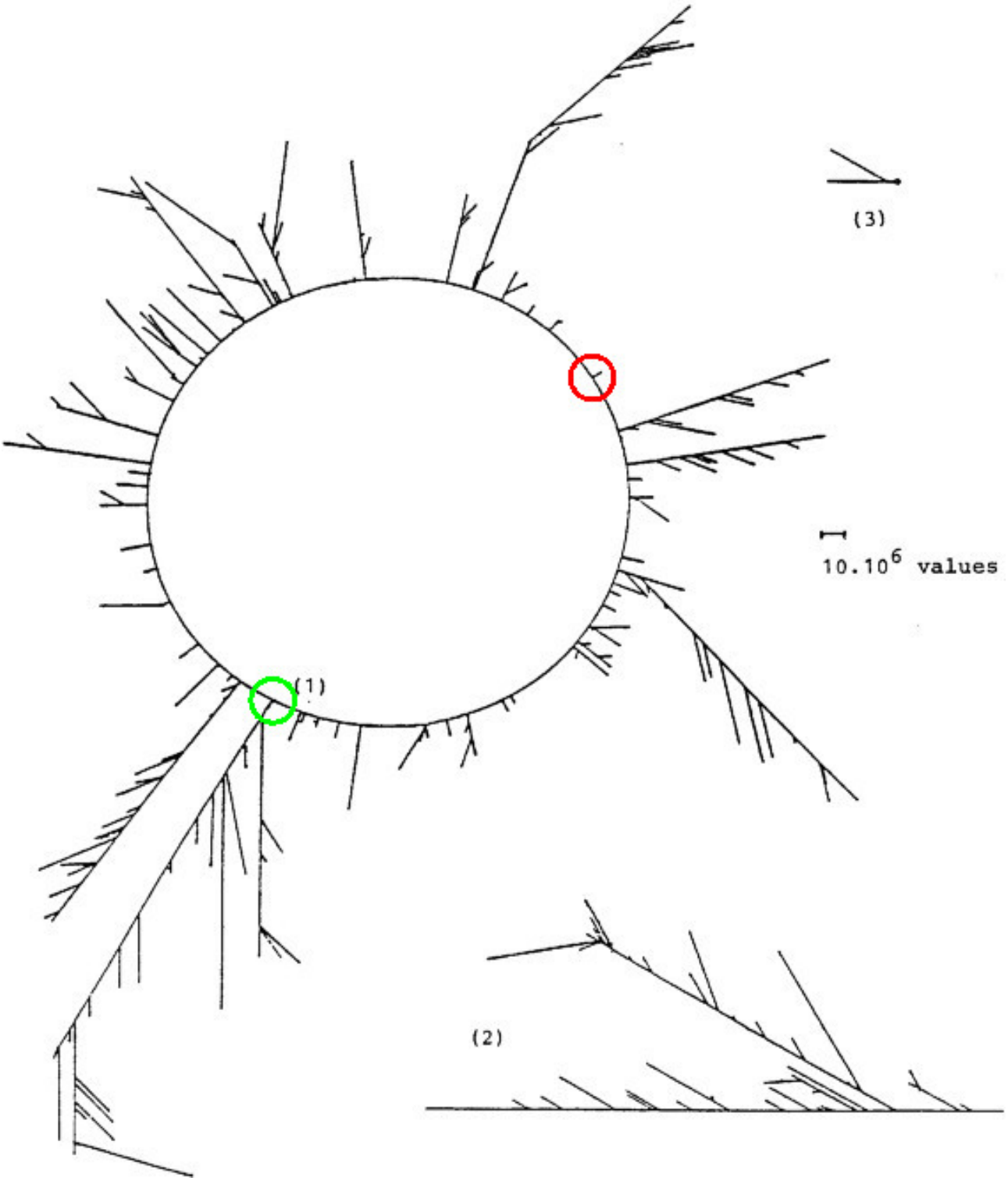


FIGURE 4.2 – Quelques composantes connexes du graphe fonctionnel du DES sous une clef fixée, calculées par Quisquater et Delescaille en 1987. Image :[172].

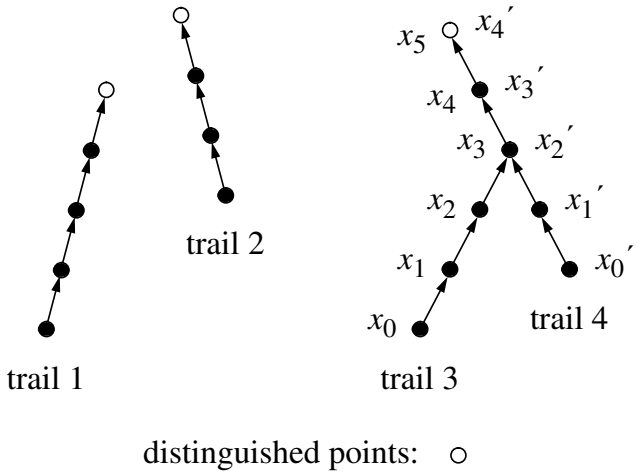


FIGURE 4.3 – L'algorithme basé sur les points distingués pour la recherche de collision parallèle. Image :[193].

ensemble A de taille N dans lui-même. L'espérance du nombre de composantes connexes est $0.5 \ln N$; celle du nombre de nœuds dans un cycle est $\sqrt{\pi N/2}$; celle du nombre de feuilles est N/e ; celle du nombre de nœuds de profondeur i est $(1-\tau_i)N$, où $\tau_0 = 0$ et $\tau_{i+1} = e^{\tau_i - 1}$. L'espérance de la hauteur d'un nœud aléatoire x est $\sqrt{\pi N/8}$; celle de la taille du cycle accessible depuis x est également $\sqrt{\pi N/8}$ (par conséquent, partant de x il faut $\sqrt{\pi N/2}$ itérations pour obtenir une collision) ; l'espérance de la taille de la composante connexe qui contient x est $2N/3$; celle du nombre d'ancêtres de x est encore $\sqrt{\pi N/8}$. Enfin, l'espérance du nombre de cycles de taille r est $1/r$ et l'espérance du nombre de nœuds qui ont r prédécesseurs est $N/(er!)$.

Pour finir, il est bien connu qu'il y a une « composante géante ». L'espérance de la taille du plus grand cycle est $0.78248\sqrt{N}$, celle de la hauteur maximale d'un nœud est $\sqrt{2\pi} \ln 2\sqrt{N} \approx 1.737\sqrt{N}$. La taille moyenne de la plus grande composante connexe est $0.75782N$.

Ces propriétés amènent une remarque : dans les graphes fonctionnels, les cycles représentent une fraction négligeable des nœuds, mais ils sont « faciles » à trouver. En effet, itérer la fonction de l'ordre de $\sqrt{N}$ depuis un point aléatoire suffit grosso modo à atterrir dans un cycle, qui peut être détecté avec le même nombre d'itérations. Par contre, ce processus semble intrinsèquement *séquentiel*.

4.2.2 Recherche d'une collision $\{0,1\}^n \to \{0,1\}^n$

Le problème se présente un peu différemment lorsque $n \neq m$, donc on va considérer dans un premier temps que les domaines d'entrée et de sortie ont la même taille. Dans ce cas, la technique classique consiste à introduire une nouvelle fonction $g : \{0,1\}^n \to \{0,1\}^n$ et un prédicat $P : \{0,1\}^n \to \{0,1\}$ en posant :

$$g : x \mapsto \begin{cases} f_0(x) & \text{si } P(x) = 0 \\ f_1(x) & \text{si } P(x) = 1 \end{cases}$$

Une collision $g(x) = g(x')$ $(x \neq x')$ révèle une collision entre f_0 et f_1 si et seulement si $P(x) \neq P(x')$. Une stratégie algorithmique pour trouver une collision entre f_0 et f_1 *sans mémoire* consiste donc à choisir un prédicat P quelconque (par exemple, un produit scalaire avec un vecteur aléatoire fixé), puis à lancer la « méthode rho » sur g, c'est-à-dire un algorithme de recherche de cycle tels que celui que Knuth attribue à Floyd, ou (mieux) celui de Brent [65]. Si la collision obtenue ne satisfait pas la condition sur le prédicat, on recommence avec un autre prédicat. En moyenne, deux essais doivent suffire. Ceci permet de trouver une collision, séquentiellement, en temps $\approx 2^{n/2}$. La mémoire nécessaire est constante, donc le coût est égal au nombre d'opérations.

En 1989, Quisquater et Delescaille ont exhibé une collision sur le DES (une même paire clair-chiffré pour deux clefs de 56 bits différentes) avec une méthode parallèle reposant sur la technique des points

distingués [173]. Cette technique a été raffinée par Van Oorschott et Wiener à la fin des années 1990 dans [193]. En voici une description.

On utilise un nouveau prédicat π qui ne renvoie 1 que sur une fraction θ de toutes les entrées possibles. Un point x est *distingué* lorsque $\pi(x) = 1$. La figure 4.1 les rend visibles.

Chaque processeur choisit un point de départ aléatoire ainsi qu'un prédicat P aléatoire, puis itère la fonction g jusqu'à atteindre un point distingué. Les points distingués sont stockés dans une liste centralisée, avec le point de départ, le prédicat P utilisé, et la longueur du *segment* qui relie le point de départ au point distingué. Lorsque le même point distingué est atteint à partir de deux points de départ différents, une collision pourra vraisemblablement être isolée (cf. figure 4.3). Les processeurs continuent à contribuer des points distingués tant qu'une collision satisfaisante n'a pas été trouvée. La longueur des *segments* suit clairement une loi géométrique de paramètre $1/\theta$. Par contre, la longueur moyenne des segments *qui collisionnent* est un peu plus élevée et la distribution est différente.

Plusieurs problèmes peuvent survenir :

— si deux *segments* collisionnent et que le point distingué qui termine le premier est précisément le point de départ du second, on ne trouvera pas de collision. Ceci pourrait être évité en s'assurant que les points de départs ne sont *pas* distingués.

— Itérer la fonction g pourrait entrer dans un cycle court qui ne contient pas de point distingué. Dans la figure 4.1, presque un quart des points de départ expose ce problème : aucun des cycles ne contient de point distingué !
Ceci peut être détecté avec la méthode de détection de cycle de Brent (au prix d'un test d'égalité par itération) ou bien simplement en notant que la probabilité de ne pas avoir atteint un point distingué après $20/\theta$ itérations est environ e^{-20}, auquel cas on peut supposer qu'on est sur un cycle. Dans ce cas-là, si le point de départ n'était *pas* sur le cycle, alors la situation est favorable : on peut localiser la collision à l'entrée du cycle avec les méthodes habituelles.

Le nombre moyen de points distingués stockés est $2^{n/2}\theta$ (à un facteur constant près). Supposons qu'on utilise p processeurs. Pour que l'ensemble ait du sens, il faut que la mémoire soit suffisante pour que chaque processeur puisse contribuer un point distingué, donc il faut que $p \leq \theta 2^{n/2}$. Le temps total nécessaire (mesuré en évaluations de la fonction g) est $T = \sqrt{\pi} 2^{n/2}/p$.

Pour étudier le coût de cet algorithme, il faut noter plusieurs aspects. Le plus important est que la fréquence à laquelle les processeurs accèdent à la mémoire commune est contrôlée par le paramètre θ. De plus, localiser la collision entre deux *segments* est un processus purement séquentiel, et donc la bonne stratégie est : 1) lancer la machine parallèle et trouver quelques paires de *segments* qui collisionnent, 2) éteindre la machine parallèle, 3) utiliser une petite machine séquentielle sans mémoire pour localiser les collisions (cette opération a un coût négligeable). Ceci peut paraître abstrait, mais si on utilisait un opérateur de *cloud* ou bien un centre de calcul partagé, c'est exactement ce qu'on serait amené à faire.

Voici une architecture possible pour réaliser le calcul dans le modèle « Mesh ». On pose $\theta = p2^{-n/2}$. Il y a donc $\theta 2^n = p2^{n/2}$ points distingués en tout. On considère une fonction ψ qui affecte chaque point distingué à l'un des p processeurs (par exemple en extrayant $\log_2 p$ bits du point distingué). La machine fonctionne de la façon suivante : chaque processeur calcule UN point distingué ; ensuite, tous les points distingués sont envoyés, via un protocole de routage, sur le *mesh* jusqu'au processeur désigné par ψ. Les processeurs qui reçoivent plusieurs points distingués émettent les collisions.

Cette machine est de taille p et elle fonctionne en temps $\frac{2^{n/2}}{p} + p^{1/3}$ (calcul et routage, respectivement). Le coût est donc $2^{n/2} + p^{4/3}$, et il est donc d'ordre $2^{n/2}$ tant que $p \leq 2^{3n/8}$ (au-delà de cette borne, la latence des communications domine le coût des calculs).

On peut noter que le modèle « Mesh » impose une restriction sur la taille de la machine qui n'existe pas dans le modèle « Cloud » (réseau idéal sans congestion). À part cela, les coûts sont les mêmes dans les deux modèles.

La recherche parallèle d'une collision a souvent été implantée, notamment pour calculer des logarithmes

discrets sur des courbes elliptiques lorsque c'est concrètement possible. L'article « *Breaking ECC2K-130* » (23 auteurs) [163] décrit une architecture client-serveur efficace pour mener ce genre de calcul à bien de manière distribuée. Bien sûr, sur des vrais ordinateurs chaque processeur disposera d'une certaine mémoire, et il faut l'utiliser pour faire baisser la complexité. Dequen, Ionica et Trimoska ont étudié les structures de données pour stocker des points distingués de manière compacte [190].

4.2.3 Recherche de toutes les collisions $\{0,1\}^n \to \{0,1\}^n$

Ce cas pose des problèmes supplémentaires et je ne connais pas d'implantation sérieuse. Il faut dire que la complexité est plus élevée, et donc les applications pratiques sont plus lointaines.

Une des difficultés est que toutes les collisions du graphe fonctionnel n'ont pas la même probabilité d'être trouvées. Dans la figure 4.2, pour trouver la collision entourée en rouge, il faut forcément tirer un point de départ sur le tout petit arbre qui touche le cycle (peu probable). Par contre, trouver la collision entourée en vert est beaucoup plus facile, car il y a beaucoup plus de prédécesseurs possibles.

Pour trouver toutes les collisions avec bonne probabilité, il faut donc *randomiser* la fonction itérée. Cela signifie par exemple qu'on peut changer aléatoirement le prédicat P utilisé dans la définition de g. Ou bien on peut simplement composer g avec une bijection quelconque, et changer de bijection de temps en temps. Le lecteur intéressé par les détails pourra consulter [193].

Une analyse heuristique montre que le nombre total d'itérations nécessaires par collision trouvée est de $\theta 2^n/M + 2/\theta$ en utilisant M mots de mémoire. Pour trouver toutes les collisions, il faut multiplier ceci par 2^n, et alors la valeur de θ qui minimise le nombre d'opérations est environ $\sqrt{M/2^n}$. Chaque collision est alors trouvée après $\sqrt{2^n/M}$ itérations (une fois que la machine est « chaude » donc que sa mémoire est remplie de points distingués). Trouver toutes les collisions revient à calculer 2^n points distingués.

Si on souhaite trouver C collisions sur une fonction aléatoire d'un ensemble de taille N dans lui-même, ceci réalise le compromis temps-mémoire $T^2M = \mathcal{O}\left(C^2N\right)$. Dinur a démontré en 2020 que ceci est optimal [87].

Voici une nouvelle possibilité de réalisation dans le modèle « *Mesh* ». On utilise p processeurs avec $\mathcal{O}(1)$ mémoire chacun. Chaque processeur choisit un point de départ aléatoire, un prédicat P et/ou une bijection supplémentaire aléatoire, puis itère g jusqu'à tomber sur un point distingué ; il route le résultat vers le processeur destination sur le réseau, puis recommence. Le processeur destination identifie et émet les collisions, puis stocke le point distingué reçu (en écrasant celui qui était déjà là).

Avec le choix de θ ci-dessus, un processeur met $\sqrt{2^n/p}$ unités de temps à produire un point distingué. Pris tous ensemble, les p processeurs produisent en moyenne $\sqrt{p^3/2^n}$ points distingués par unité de temps. Le problème consiste à vérifier que ceci ne sature pas le réseau, et pour cela on peut reprendre les raisonnements de la section 4.1. Si on coupe la machine en deux, les deux moitiés produisent des points distingués au même rythme, et la moitié d'entre eux doivent franchir la coupe, donc il faut que $\sqrt{p^3/2^n} \leq p^{2/3}$, sinon le rythme de production est plus rapide que la capacité de transport de la coupe. Ceci se traduit par $p \leq 2^{3n/5}$.

On peut remarquer que cette borne est plus permissive que dans le cas où on cherchait une seule collision : dans le cas précédent, la latence du routage était primordiale, alors qu'ici c'est une forme de bande passante qui compte.

On pose donc $p = 2^{\alpha n}$, avec $0 < \alpha < \frac{3}{5}$. L'espérance du temps d'exécution est $2^{\frac{3}{2}(1-\alpha)n}$ et la machine est de taille $2^{\alpha n}$. Le coût total est donc $2^{\frac{3-\alpha}{2}n}$. Augmenter le nombre de processeurs réduit le nombre total d'opérations nécessaires car cela augmente la taille de la mémoire et réduit les calculs inutiles.

Avec $p = 2^{3n/5}$, le coût est $2^{6n/5}$: chaque processeur calcule $2^{2n/5}$ points distingués ; calculer un point distingué nécessite $2^{n/5}$ opérations séquentielles, et le routage dans le *mesh* nécessite autant d'étapes : la machine est de taille $2^{n/5} \times 2^{n/5} \times 2^{n/5}$.

Si on s'était restreint à un *mesh* à deux dimensions, il aurait fallu choisir $\alpha < \frac{1}{2}$ et le meilleur coût possible aurait été $2^{5n/4}$. Si on disposait simplement d'une ligne de processeurs reliés à chacun de ses deux voisins, il aurait fallu choisir $\alpha < \frac{1}{3}$ et le meilleur coût possible aurait été $2^{4n/3}$, comme avec la méthode naïve.

Ceci illustre que les communications peuvent jouer un rôle crucial dans l'étude des algorithmes, et que différents modèles de communication peuvent amener à des résultats différents. Faire l'hypothèse que les communications sont gratuites permet de meilleurs résultats sur le papier, mais on risque de se heurter à des problèmes si on tente de programmer quoi que ce soit sur de vraies machines.

Machines inférieures

Voici quelques idées de machines qui donnent un coût supérieur. Utiliser M cases mémoire par processeur divise le nombre d'opérations total par $\sqrt{M}$ mais augmente la taille de la machine d'un facteur M — donc ça augmente le coût de $\sqrt{M}$. La meilleure solution est donc $M = 1$.

La recherche de toutes les collisions avec une mémoire inférieure à 2^n « gaspille » des calculs : des points distingués vont être stockés puis ré-écrits avant d'avoir pu contribuer à une collision. Pour éviter ce phénomène, c'est-à-dire pour ne jamais « oublier » un point distingué calculé, il faudrait augmenter le nombre de processeurs à $p = \theta 2^n$. Ceci ralentirait la phase de routage dans le réseau, et il faudrait diminuer θ pour garder l'équilibre entre communications et calculs. Ceci conduit à fixer $\theta = 2^{-n/4}$ avec $p = 2^{3n/4}$ processeurs. Comme il faut trouver 2^n collisions, alors il faut calculer au moins 2^n points distingués. Ceci prend un temps $2^{n/2}$, et donc coûte $2^{5n/4}$.

Il s'avère donc que s'autoriser à oublier des points distingués permet d'utiliser une machine plus petite et d'atteindre un meilleur coût.

Cas particulier des fonctions dont la description n'est pas de taille négligeable

Tous les raisonnements précédents reposent sur l'hypothèse qu'il existe un programme *compact* pour évaluer les fonctions sur lesquelles on cherche une collision, et en tout cas dont la taille ne varie pas sensiblement avec n. On s'intéresse maintenant au cas où les fonctions ont une description de taille $2^{\alpha n}$, avec $\alpha \leq 1$. Ceci modélise notamment une recherche de collision entre deux fonctions dont l'évaluation nécessite l'accès à un tableau exponentiellement grand. On se concentre sur la recherche de toutes les collisions dans des fonctions $\{0,1\}^n \to \{0,1\}^n$.

Pour commencer, il y a une technique simple : tabuler les deux fonctions (ce qui produit deux listes de taille 2^n), trier les deux listes, chercher les collisions avec une sorte de fusion. L'espace nécessaire au stockage des deux fonctions est dominé par la taille des listes construites pendant la recherche des collisions. Le théorème 1 affirme que ceci peut se réaliser avec un coût de $2^{4n/3}$. Il s'agit donc de faire mieux que cela.

Ceci peut se faire de manière directe en reprenant l'architecture en *mesh* 3D décrite ci-dessus, c'est-à-dire utiliser $p = 2^{3n/5}$ processeurs qui calculent chacun $2^{2n/5}$ points distingués ; chacun d'entre eux nécessite $2^{n/5}$ évaluations de la fonction plus un routage dans le réseau. Cependant, pour pouvoir évaluer la fonction, il est nécessaire de doter chaque processeur d'une mémoire suffisamment grosse pour contenir sa description. Le coût est alors $2^{(6/5+\alpha)n}$. Ceci est meilleur que la technique de base lorsque $\alpha < 2/15$.

On va maintenant essayer d'améliorer ceci. Supposons dans un premier temps que $\alpha \leq 3/5$. Pour éviter le grossissement de la machine, on peut répartir la description de la fonction sur tous les processeurs sans remettre en cause le fait qu'ils ont une mémoire de taille constante — par contre, l'évaluation de la fonction nécessitera alors un routage dans le réseau pour accéder à cette mémoire distribuée. On peut découper la machine en $2^{(3/5-\alpha)n}$ « partitions » de taille $2^{\alpha n}$, chacune capable de stocker la description de la fonction. Au sein d'une de ces partitions, tous les processeurs peuvent évaluer la fonction simultanément en effectuant un routage (avec l'algorithme de Valiant [191]) sur le réseau *de la partition*, ce qui nécessite un temps $2^{\alpha n/3}$. On en arrive à une machine de taille $2^{3n/5}$ qui effectue le

calcul en temps $2^{(3/5+\alpha/3)n}$. Le coût total est donc $2^{(6/5+\alpha/3)n}$, ce qui est mieux qu'avant. Ça améliore l'idée de base pour $\alpha < 2/5$.

On va maintenant améliorer encore les performances tout en éliminant la restriction sur α. Comme l'évaluation de la fonction a désormais un coût asymptotiquement significatif, il devient pertinent de réévaluer les équilibres établis au-dessus entre communications et calculs. Augmenter le nombre de processeurs, donc la mémoire totale disponible, permet de réduire le nombre total d'évaluations de la fonction, qui est de l'ordre de $\sqrt{2^{3n}/p}$. En contrepartie, acheminer les points distingués dans le réseau sera potentiellement plus lent, mais en fait il s'agit d'un rééquilibrage. On va choisir un nombre de processeurs suffisant pour que la description de la fonction puisse être stockée tout en conservant une mémoire constante par processeur. Ceci implique en particulier que $p \geq 2^{\alpha n}$.

Rappelons qu'avec $\theta = \sqrt{p/2^n}$, chaque processeur a besoin de $\sqrt{2^n/p}2^{\alpha n/3}$ unités de temps pour produire un point distingué ; les p processeurs produisent $\sqrt{p^3/2^n}2^{-\alpha n/3}$ points distingués par unité de temps ; ceci atteint la capacité de transport du réseau global de la machine lorsque $\sqrt{p^3/2^n}2^{-\alpha n/3} = p^{2/3}$. Ceci se traduit par $p = 2^{3n/5+2\alpha n/5}$, et ceci est toujours plus grand que $2^{\alpha n}$. Le calcul d'un point distingué nécessite donc un temps $2^{n/5+2\alpha n/15}$. Chacun des processeurs calcule $2^{2n/5-2\alpha n/5}$ points distingués, ce qui nécessite un temps $T = 2^{3n/5-4\alpha n/15}$. Le coût total est donc $pT = 2^{6n/5+2\alpha n/15}$, ce qui est encore mieux que ce qu'on avait trouvé avant. C'est toujours meilleur que le coût de la machine de base en $2^{4n/3}$, quelle que soit la valeur de $\alpha \leq 1$.

4.2.4 Cas des fonctions expansives $\{0,1\}^n \to \{0,1\}^m$ avec $n < m$

Pour trouver toutes les collisions entre f_0 et f_1 lorsque la sortie est plus grosse que l'entrée, on peut commencer par tronquer la sortie pour se ramener au cas $\{0,1\}^n \to \{0,1\}^n$. On peut alors trouver toutes les collisions (partielles) sur les n premiers bits de la sortie ; pour chacune d'entre elles on peut alors tester si c'est une collision complète sur les m bits. Le coût dans ce cas-là est toujours $2^{6n/5}$.

Si on cherche juste une seule collision, alors il faut réaliser que les techniques de recherche de cycle traditionnelles ne vont pas s'appliquer. Par exemple, une technique qui ne fonctionne pas est la suivante : utiliser la « méthode rho » sur la fonction $g \circ M$, où M est une fonction linéaire aléatoire de n bits dans m bits. Le problème, c'est qu'une collision aléatoire sur $g \circ M$ est avec forte probabilité une collision... sur M.

Si $m > 2n$, alors il faut trouver toutes les collisions possibles sur la fonction tronquée et les tester toutes. Si $m < 2n$, on peut se contenter d'en trouver 2^{m-n}, car alors on aura en moyenne *une* collision sur les m bits.

Le nombre d'opérations par collision est toujours $\theta 2^n/M + 2/\theta$ en utilisant M mots de mémoire. Le nombre total d'opérations est alors $\theta 2^m/M + 2^{m-n+1}/\theta$, et alors la valeur de θ qui minimise le nombre d'opérations est toujours $\sqrt{M/2^n}$. Le nombre d'opérations total est alors $2^m/\sqrt{M2^n}$.

On utilise $P = 2^{\alpha n}$ processeurs (munis d'une mémoire constante). Chaque processeur effectue donc $2^{m-n/2-3\alpha n/2}$ itérations de la fonction, et il faut que ceci soit au moins $1/\theta$ (pour que chaque processeur trouve au moins un point distingué), donc on a la contrainte $\alpha \leq m/n - 1$, et on a toujours $\alpha \leq \frac{3}{5}$ pour que les communications ne dominent pas les calculs.

Si $m \leq 8n/5$, alors on pose $\alpha n = m - n$. Le temps d'exécution de la machine est alors $2^{n-m/2}$, et le coût est $2^{m/2}$. Dans ce cas précis, il est égal au nombre d'opérations auquel on s'attendrait séquentiellement.

Si $m > 8n/5$, alors on pose $\alpha = \frac{3}{5}n$. Le temps d'exécution est alors $2^{m-7/5n}$ et le coût total est donc $2^{m-4n/5}$. C'est parce que les communications ne sont pas gratuites et que le réseau de communication que le coût n'est pas le même dans les deux situations dans le modèle « Mesh ».

4.2.5 Cas des fonctions contractantes $\{0,1\}^n \to \{0,1\}^m$ avec $n > m$

Trouver une seule collision est facile : il suffit de fixer arbitrairement les bits excédentaires de l'entrée puis de se ramener au cas d'une fonction $\{0,1\}^m \to \{0,1\}^m$.

Trouver toutes les collisions est différent, car il y a en a plus que d'habitude. Au total, on s'attend à ce qu'il y en ait 2^{2n-m} en tout.

Une solution simple consiste à répéter 2^{2n-2m} fois la recherche de toutes les collisions sur une fonction $\{0,1\}^m \to \{0,1\}^m$, en fixant $n-m$ bits de l'entrée de chacune des deux fonctions à toutes les valeurs possibles. Le coût de cette solution est $2^{2n-4m/5}$.

4.2.6 Application : le Double-DES

Le double-DES consiste à effectuer le chiffrement d'un bloc de 64 bits avec deux clefs de 56 bits chacune, en faisant :

$$\mathsf{2DES}_{k_1,k_2}(x) = \mathsf{DES}_{k_1} \circ \mathsf{DES}_{k_2}(x).$$

Plus généralement, le chiffrement double consiste à utiliser un système de chiffrement par bloc deux fois de suite avec deux clefs de n bits, ce qui donne des clefs de $2n$ bits. Le fait que le chiffrement double n'offre en réalité que « n bits de sécurité » semble être un consensus répandu [1]. En effet, une attaque « *meet-in-the-middle* » très célèbre, due à Diffie et Hellman [83] retrouve la clef de $2n$ bits avec deux paires clair-chiffré, en temps 2^n et mémoire 2^n. L'attaque fonctionne de la façon suivante : pour chaque k_1 possible, stocker l'association $E_{k_1}(P_1) \mapsto k_1$ dans un dictionnaire. Ensuite, pour chaque k_2 possible, chercher la ou les valeurs de k_1 associées à $E_{k_2}^{-1}(C_1)$ dans le dictionnaire. Finalement, tester toutes les combinaisons (k_1, k_2) obtenues de la sorte avec la deuxième paire clair-chiffré.

À première vue, il pourrait sembler que le chiffrement double n'offre pas véritablement d'avantage par rapport au chiffrement simple. Un examen plus approfondi révèle une situation plus complexe. La recherche exhaustive sur le DES a été déjà été effectuée en 1998 avec une machine parallèle nommée « *Deep Crack* » spécialement conçue pour l'occasion [107]. *A contrario*, l'attaque décrite ci-dessus a un « ratio temps-mémoire » de 1.00, ce qui la classe dans la catégorie « impraticable ». Une estimation *optimiste* suggère qu'il faut 8×2^{56} octets de mémoire pour stocker le dictionnaire, ce qui fait $\approx$ 512Po de mémoire rapide. Aucune machine au monde ne possède autant de mémoire.

Par ailleurs, on a expliqué ci-dessus que le coût de l'exécution de cette attaque simple et directe n'est pas de l'ordre de 2^n mais de $2^{4n/3}$. Casser le double-DES semble donc plus dur que prévu. Utiliser la recherche de collision parallèle (versions « toutes les collisions d'une fonction de n bits ») donne un coût de $2^{6n/5}$, comme discuté ci-dessus.

Dans tous les cas, utiliser moins de 2^n bits de mémoire va nécessiter plus de 2^n opérations. Par conséquent, briser le double-DES sur le cloud coûtera *bien plus cher* que de casser le simple DES par recherche exhaustive. Cela demandera plus de stockage, plus d'énergie, plus de ressources matérielles, plus d'effort intellectuel, etc. Toutes choses qui ne sont pas reflétées par le fait de compter le nombre d'opérations élémentaires que l'attaque nécessite.

En fait, casser le chiffrement double sur le *cloud* coûte *asymptotiquement* plus d'argent que de casser le chiffrement simple par recherche exhaustive.

Le double-DES a « depuis toujours » été décrit comme faible et n'a jamais été utilisé pour cette raison. Le calculateur `jean-zay` dont le CNRS a fait l'acquisition en 2019 possède 61 120 cœurs, donc casser RSA-1024 demanderait, dans le meilleur des cas, 6.5 ans d'exploitation exclusive de la machine. Que pourrait donner l'exécution de l'algorithme de recherche de collision parallèle pour casser le double-DES sur la même machine ? En utilisant `usuba`, le compilateur *bit-slicing* de Dagand et Mercadier [159], il semble possible de pouvoir faire 211 millions d'évaluations du DES par seconde sur un cœur de `jean-zay`. La machine peut donc casser le simple DES par recherche exhaustive en une heure et demie.

En utilisant presque toute la RAM disponible, à savoir 261 288Go, le choix réputé optimal consiste à décréter qu'un point sur 32 est distingué (en reprenant l'analyse de [193]). Alors $2^{63.6}$ évaluations du DES seraient requises. Mais il y a un hic : chaque nœud de la machine produirait alors des points

1. C'est en tout cas ce qu'affirme la page Wikipédia en anglais consacrée au triple-DES [201].

distingués au rythme de 4Go/s, or le réseau ne peut pas probablement pas transmettre un tel volume de données sans congestion.

Le choix optimal consiste probablement à distinguer un point sur 128. Les nœuds produiraient alors des points distingués au rythme de 1Go/s, ce qui est élevé mais peut-être acceptable. Le nombre total d'évaluations du DES requises est alors $2^{64.2}$. On peut alors estimer que l'attaque nécessitera 30 millions d'heures-CPU, soit à peu près 3 semaines d'utilisation exclusive de la machine.

Ces estimations au doigt mouillé mériteraient d'être affinées. Elles suggèrent que casser le double-DES est environ 300 fois plus dur que de casser le simple-DES. Ceci n'est pas prédit pas le modèle de la Random Access Machine.

4.2.7 Structure de données pour les (sous-)graphes fonctionnels

Plusieurs attaques cryptographiques récentes reposent sur une structure de données capable de représenter des morceaux du graphe d'une fonction $f : \{0,1\}^n \to \{0,1\}^n$. Plus précisément, étant donné un paramètre g, avec $n/2 \leq g$, il s'agit de représenter un sous-ensemble de 2^g nœuds (quelconques) du graphe fonctionnel de f, dont une fraction constante est de « profondeur » $\geq 2^{n-g}$. La structure de données supporte les trois opérations suivantes :

- $\mathcal{D} \leftarrow \textsc{Build}(f, g)$: initialise la structure de données.
- $\textsc{Iterate}(\mathcal{D})$: permet l'itération efficace sur les nœuds représentés par $\mathcal{D}$.
- $\textsc{Membership}(\mathcal{D}, x)$: teste si $x \in \mathcal{D}$.

Cette structure de données peut, dans sa variante la plus simple, être construite en recommençant la procédure suivante tant que 2^g nœuds n'ont pas été accumulés : choisir un nœud aléatoire x_0, puis calculer $x_{i+1} = f(x_i)$ jusqu'à ce qu'une collision soit trouvée ($x_i = x_j$ et $j > i$) ou bien que x_i fasse déjà partie des nœuds connus ; stocker tous les nœuds rencontrés jusque-là. Ceci nécessite un temps (et un un espace) 2^g.

La i-ème itération de cette boucle produit une chaîne de longueur moyenne $2^{n/2}/\sqrt{i}$, donc il devrait y avoir 2^{2g-n} itérations. Toutes les itérations pourraient être effectuées en parallèle : il faut pour cela ajouter les sommets du graphe à la structure de donnée au fur et à mesure qu'ils sont produits. Si un processeur essaye d'ajouter un sommet déjà présent, alors il se réinitialise et recommence à itérer la fonction depuis un sommet aléatoire.

Dans cette réalisation naïve, l'espace nécessaire est 2^g. Le calcul de la première chaîne nécessite $2^{n/2}$ étapes de calcul séquentiel, donc le coût est d'au moins $2^{g+n/2}$. Comme il faut que $g \geq n/2$, ceci ne peut pas être moins cher que la recherche exhaustive.

Heureusement, Dinur [86] observe qu'on peut réduire sensiblement la quantité d'espace nécessaire avec une technique de points distingués. Il s'agit de considérer qu'une proportion 2^{g-n} des nœuds est distinguée (par exemple ceux dont les $n-g$ bits de poids faibles valent zéro), et de ne stocker que le graphe des points distingués. Il faut donc choisir aussi des points distingués pour démarrer les chaînes calculées. Tester l'appartenance d'un nœud x quelconque à la structure de donnée n'est alors possible que lorsque x est un point distingué lui aussi. Par contre, énumérer tous les nœuds appartenant à la structure est facile.

La taille de la structure de données modifiée est alors 2^{2g-n}, et on admet que les complexités en temps sont les mêmes qu'avant. La construction de la structure de donnée est alors un algorithme qui effectue des accès mémoire à une fréquence réduite, de l'ordre de $r = 2^{g-n}$, en effectuant 2^g opérations et utilisant une mémoire 2^{2g-n}. À cause du caractère intrinsèquement séquentiel des calculs à effectuer (parcourir les « chaînes »), il n'est pas possible d'utiliser le nombre de processeurs qui minimiserait le coût.

Comme il y a au moins une tâche qui nécessite un temps $2^{n/2}$, il semble raisonnable de viser $p = 2^{g-n/2}$ processeurs (c'est la racine carrée du nombre de chaînes à calculer). La mémoire totale est 2^{2g-n}, et la « longueur des fils » est alors $(pr)^{3/2} = 2^{3g-9n/4}$ d'après le théorème 1. C'est la taille de la mémoire qui domine, et le coût total est donc $2^{2g-n/2}$.

Envisageons une architecture de type *Mesh* 2D pour nous convaincre que tout va bien. On considère une grille de $2^{g-n/2} \times 2^{g-n/2}$ processeurs munis d'une mémoire de taille constante. Seuls les processeurs de la première rangée effectuent des calculs, les autres ne sont que des serveurs de stockage. Lorsqu'un processeur calcule un point distingué, il l'envoie sur le *mesh* jusqu'au nœud destination, et reçoit en échange un booléen lui indiquant si ce point distingué était déjà connu. Calculer un point distingué nécessite un temps 2^{n-g}, tandis qu'effectuer le routage sur le réseau nécessite un temps $2^{g-n/2}$. Tant que $g < 3n/4$, la durée des calculs domine la durée des communications. Pris ensemble, les processeurs produisent $2^{2g-3n/2}$ points distingués par seconde, or la *bisection bandwidth* du réseau est $2^{g-n/2}$. Les processeurs ne peuvent donc pas la saturer.

On note enfin que la même structure de données permet de connaître la *hauteur* de chaque nœud assez facilement. Pour cela, il faut stocker le sommet de départ de la chaîne avec chaque point distingué (ceci permet de détecter les cycles). Il faut marquer tous les nœuds dans les cycles à hauteur zéro. Ensuite, il faut reparcourir toutes les chaînes deux fois : lorsqu'on tombe sur un nœud dont la hauteur est connue, on s'arrête. Le premier passage permet de connaître la distance entre le point de départ et le prochain sommet dont la hauteur est connue. Le deuxième passage permet de décorer tous les points distingués rencontrés avec leur hauteur.

4.3 Calcul des jointures

Étant donné deux listes $\mathcal{L}_1$ et $\mathcal{L}_1$ de paires, la *jointure* des deux listes, souvent notée $\mathcal{L}_1 \bowtie \mathcal{L}_1$ est l'ensemble des paires (b, c) telles que $(a, b) \in \mathcal{L}_1$ et $(a, c) \in \mathcal{L}_2$. Il est commode de se représenter que les deux listes contiennent en fait des paires « clef $\mapsto$ valeur », et du coup la jointure consiste à trouver toutes les paires de valeurs qui partagent la même clef. On considère qu'il suffit d'*émettre* le résultat de la jointure, et pas forcément de la stocker en mémoire (la littérature spécialisée parle de « matérialiser » la jointure).

Le concept est abondamment utilisé dans les bases de données relationnelles et toute une littérature est consacrée aux algorithmes efficaces pour les calculer dans cette communauté. Il est aussi largement utilisé dans de nombreux algorithmes en cryptographie, à commencer par les algorithmes pour le paradoxe des anniversaires généralisé. Celui de Wagner [198], qui empreinte à des travaux précédents de Vaudenay ou Camion-Patarin, est le plus connu. Enfin, il intervient plus généralement dans un grand nombre d'attaques qui ont une phase de « *meet-in-the-middle* ».

Une stratégie simple consiste à trier les deux listes par clefs croissantes. Ensuite, une procédure comparable à la fusion du « tri fusion » permet d'énumérer les éléments de la jointure, en temps linéaire en la taille de l'entrée et de la sortie. Si en plus les clefs sont aléatoires, alors le tout peut se faire en temps linéaire aussi (car on peut trier des clefs aléatoires en temps linéaire). Une autre stratégie simple consiste à initialiser un dictionnaire (par exemple une table de hachage) avec le contenu de la première liste, puis à tester la présence dans cette dernière de toutes les clefs de la deuxième liste. L'avantage est que ceci est censé fonctionner en temps linéaire quoi qu'il arrive.

Dans le modèle de la *Random Access Machine*, le problème est donc résolu, puisqu'on dispose non pas d'une, mais de deux stratégies algorithmiques de complexité optimale.

Sur le vrai matériel cependant, les deux méthodes décrites ci-dessus ne sont pas équivalentes. Utiliser une grosse table de hachage aboutit à de très mauvaises performances en raison de la faible localité des données ; chaque accès provoque une faute de cache, voire même de TLB et le tout est donc assez lent (cf. la discussion de la section 2.3). Du coup, une solution hybride semble avoir la préférence des experts : effectuer un tri sommaire, par exemple sur les k bits de poids fort des clefs des deux listes, ce qui permet de *partitionner* le calcul de la jointure initiale en 2^k calculs de jointures sur des sous-listes. L'avantage de la manœuvre, c'est que les sous-listes sont plus petites, donc les tables de hachage peuvent tenir en cache et le tout peut être efficace.

La valeur optimale de k dans le partitionnement décrit ci-dessus est typiquement 512 sur les processeurs courants, car c'est le nombre de lignes du cache L1 — cette valeur de k est la plus grande qui permet

des accès efficaces au cache. Des techniques légèrement contre-intuitives, telles que le *software write-combining buffer* [13], qui consiste à déplacer les données en mémoire de l'adresse A vers l'adresse C *via* une étape intermédiaire à une adresse B a priori en cache, permettent de minimiser le nombre de fautes de cache et de TLB. On peut alors s'approcher assez près de la performance de crête du matériel.

En pratique, l'implantation efficace des jointures lorsque les listes sont données en entrée n'est pas facile et fait intervenir à la fois des considérations algorithmiques et des considérations architecturales de bas-niveau. En effet, le type d'algorithme à utiliser dépend en partie de la taille des listes (tiennent-elles en cache ?). Souvent, en cryptographie, les deux listes sont de tailles comparables, ce qui élimine déjà une partie des problèmes. Mais la taille relative des clefs et des listes joue un rôle : le nombre moyen de valeurs par clef est-il supérieur à 1 ? Une fraction significative des clefs est-elle présente ? La taille de la sortie est-elle prévisible ? Les réponses à ces questions orientent des choix algorithmiques concrets dans les « vraies » implantations.

On s'intéresse maintenant au cas où les deux listes en entrée sont de même taille N, et on note K la taille de l'ensemble des clefs (on suppose que ces dernières sont aléatoires). Quel est le coût de l'opération ? On peut commencer par noter que le calcul des jointures permet aisément de décider si deux listes ont une intersection non-vide. Or il s'agit d'un problème central de l'étude de la complexité de communication, et des bornes inférieures sont connues : si Alice possède X et Bob possède Y (deux ensembles de taille N), alors ils doivent se communiquer $\Omega(N)$ bits pour tester si $X \cap Y = \emptyset$ [182]. Ceci risque fortement d'impliquer des bornes inférieures de type VLSI sur le coût du calcul des jointures.

Il faut noter que toutes les stratégies décrites ci-dessus coûtent au moins $N^{4/3}$, soit parce que c'est le coût du tri des listes en entrée, soit parce qu'on retombe sur ce qu'on faisait section 4.1. Dans les faits, le calcul des jointures est généralement *memory-bound* (au moins en partie), ce qui est cohérent avec cette constatation sur le coût de l'opération.

Pour être plus précis, imaginons qu'on dispose d'une machine à N nœuds organisés en *mesh* 3D. On découpe la machine en K partitions (connexes) — chaque partition est associée à une clef. Initialement, les deux listes de N éléments sont placées de manière quelconque avec un élément par nœud. La première étape consiste à transférer chaque paire clef-valeur jusqu'à un nœud (aléatoire) de la partition destination. Comme on l'a déjà vu, ceci nécessite un temps $\mathcal{O}\left(p^{1/3}\right)$ avec l'algorithme de routage de Valiant. Ensuite, chaque partition peut traiter l'ensemble des paires stockées par ses nœuds. Ceci peut se faire en N/K étapes, en faisant « tourner » une des listes dans la partition tandis que l'autre est fixe. Le coût total est alors :

$$N\left(N^{1/3} + \frac{N}{K}\right) = N^{4/3} + \frac{N^2}{K}$$

Le premier de ces deux termes représente le coût du partitionnement, tandis que le deuxième représente le coût de la production des paires dans les partitions individuelles.

La situation pourrait être différente si on devait calculer la jointure entre deux listes *dont on peut facilement calculer les entrées*, car il n'est alors pas nécessaire de les stocker et on peut se contenter d'une machine plus petite. Imaginons donc qu'on dispose de deux fonctions $f_0 : \{0,1\}^n \to \{0,1\}^k \times A$ et $f_1 : \{0,1\}^n \to \{0,1\}^k \times B$; on cherche à calculer l'ensemble des paires (i,j) telles que $f(i) = (u,x)$ et $f(j) = (v,y)$ avec $u = v$. Si les fonctions f_0 et f_1 sont suffisamment aléatoires, alors ceci peut se ramener à une recherche de toutes les collisions entre deux fonctions. Si $k = n$ par exemple, alors ceci peut se réaliser avec un coût de $2^{6n/5}$, qui est donc amélioré par rapport à la technique générique (cf. section 4.2.3).

4.4 Le problème 3XOR

Le problème 3XOR, déjà défini section 2.4, a récemment connu un regain d'intérêt. Il s'agit d'un cas particulier difficile (avec seulement trois « listes ») du problème des anniversaires généralisé d'abord étudié par Wagner [198], qui en a laissé l'étude comme un problème ouvert. Il peut être utilisé comme brique de base dans des attaques plus sophistiquées ; c'est notamment le cas de l'attaque inventée par

Algorithme	Ref.	Temps	Mémoire	Entrées
Quadratique	folklore	$2^{2n/3}$	$2^{n/3}$	Arbitraires
Collision	folklore	$2^{n/2}$	1	Choix adaptatif
Nikolić et Sasaki	[164]	$2^{n/2}/\sqrt{n/\ln n}$	$2^{n/2}/\sqrt{n/\ln n}$	Arbitraires
Joux	[125]	$2^{n/2}/\sqrt{n}$	$2^{n/2}/\sqrt{n}$	Arbitraires
Joux généralisé	[44]	$2^{2n/3}/n$	$2^{n/3}$	Arbitraires

TABLE 4.1 – Algorithmes pour la résolution du problème 3XOR avec trois fonctions aléatoires. Les quantités sont asymptotiques. La colonne « Entrées » désigne les valeurs fournies aux trois fonctions aléatoires f, g et h.

Nandi en 2015 contre le mode opératoire COPA de chiffrement authentifié [161], ainsi que de l'attaque découverte en 2019 par Leurent et Sibleyras contre le chiffrement d'Even-Mansour à deux tours et une seule clef [149] (il en sera question plus en détail section 5.3.3).

Des méthodes qui appartiennent au folklore peuvent résoudre le problème. À la suite de l'« amélioration incrémentale » de Joux en 2010 (déjà mentionnée section 2.4), une amélioration *décrémentale* a été proposée par Nikolić et Sasaki en 2014, qui utilise encore plus d'opérations. Enfin, en collaboration avec Delaplace et Fouque, j'ai proposé une généralisation de l'algorithme de Joux en 2018 [44]. Le tableau 4.1 résume ces algorithmes.

On considère d'abord la version suivante (« avec oracles ») du problème. Soient $F, G, H : \{0,1\}^n \to \{0,1\}^n$ trois fonctions aléatoires. On considère le problème de calculer un triplet 3XOR sur (F, G, H) : trouver x, y et z tels que $F(x) \oplus G(y) \oplus H(z) = 0$.

Il faut noter que, pour qu'une solution existe avec probabilité non-négligeable, le produit du nombre d'invocations des trois fonctions doit être supérieur à 2^n. Le cas équilibré consiste donc à évaluer les trois fonctions $2^{n/3}$ fois chacune. Dans ce cas, la seule méthode connue nécessite alors $\tilde{\mathcal{O}}\left(2^{2n/3}\right)$ opérations.

Il faut noter que l'*espérance* du nombre de « triplets 3XOR » qu'on peut obtenir en évaluant chaque oracle N fois est facile à calculer : c'est $N^3/2^n$. Par contre, la *probabilité* d'en obtenir au moins un en évaluant les oracles N fois chacun ne s'en déduit pas directement ; le résultat n'est pas directement disponible dans la littérature publiée. On ne peut pas l'obtenir en utilisant une inégalité de concentration simple telle que la borne de Chernoff, car les triplets ne sont pas indépendants les uns des autres. On peut l'obtenir en utilisant la méthode du deuxième moment, après un peu de calcul, pour n'importe quelle opération de groupe et beaucoup de distributions différentes sur les entrées.

D'autres algorithmes évaluent les oracles plus que nécessaire pour avoir un seul triplet 3XOR, et tirent partie de la présence de nombreux triplets pour en trouver *un* qui ait une structure particulière. Ceci donne par exemple des techniques qui nécessitent $2^{n/2}$ opérations avec $2^{n/2}$ évaluations des fonctions.

4.4.1 Algorithmes appartenant au folklore

L'algorithme le plus simple possible est probablement l'algorithme quadratique :

1. Préparer une table de hachage de taille $2^{n/3}$; stocker $A[H_k(i)] \leftarrow i$ pour tout $0 \leq i < 2^{n/3}$.
2. Tabuler les deux autres fonctions de manière identique ; pour tout $0 \leq i < 2^{n/3}$ faire : $B[i] \leftarrow F_k(i)$ et $C[i] \leftarrow G_k(i)$.
3. Chercher une solution. Pour tous $0 \leq i, j < 2^{n/3}$ faire : poser $u \leftarrow B[i] \oplus C[j]$; sonder la table de hachage A avec la clef u ; si $u \in A$, alors signaler le triplet $(i, j, A[u])$.

L'algorithme nécessite une mémoire de taille $M = 2^{n/3}$, et tel qu'il est décrit sa complexité en temps est $T = 2^{2n/3}$ opérations. Le coût de cette version séquentielle serait donc de 2^n.

Il est possible d'améliorer ceci avec une idée simple que Bernstein [21] a nommée le « *clamping*[2] » :

2. En fait, cette idée est à la base du fameux algorithme de Schrœppel-Shamir [177].

évaluer chaque oracle $2^{n/2}$ fois, mais ne stocker que les résultats qui commencent par $n/4$ bits à zéro. Ceci donne trois « listes » de chaînes de $3n/4$ bits, de taille $2^{n/4}$, qui contiennent en moyenne une unique solution. Celle-ci peut être trouvée avec l'algorithme quadratique, en temps $2^{n/2}$ et en espace $2^{n/4}$, donc avec un coût réduit à $2^{3n/4}$. Le « *clamping* » permet donc de réduire le coût, toutes choses égales par ailleurs, en augmentant le nombre de requêtes aux oracles.

La technique de recherche de collision déjà évoquée section 2.4 résout le problème séquentiellement en temps $2^{n/2}$ sans mémoire, donc elle coûte elle aussi $2^{n/2}$. L'inconvénient principal de cette méthode est qu'elle nécessite une évaluation *adaptative* de deux des trois fonctions. Il est donc impossible de les « tabuler » à l'avance sur des entrées librement choisies.

Pour obtenir une alternative compétitive à la technique de recherche de collision, il faut examiner des versions parallèles de l'algorithme quadratique. Le problème 3XOR peut être rendu massivement parallèle avec une approche « diviser-pour-régner » : commencer par tabuler les trois fonctions ; partitionner ensuite chacune des trois listes A, B et C selon le bit de poids fort des clefs. Ceci découpe le problème original en quatre sous-problèmes de taille réduite de moitié : en effet, un triplet 3XOR dans (A, B, C) appartient nécessairement à $(A_0, B_0, C_0), (A_0, B_1, C_1), (A_1, B_0, C_1)$ ou bien (A_1, B_1, C_0). Ceci est en réalité une reformulation de l'algorithme quadratique, car la complexité de cette procédure récursive est gouvernée par la récurrence $T(N) = 4T(N/2)$, qui aboutit à $T(N) = \mathcal{O}\left(N^2\right)$.

Décrivons donc une parallélisation de l'algorithme quadratique dans le modèle « cloud ». On dispose d'un stock illimité de nœuds avec une mémoire fixée de taille B. On utilise le *clamping* sur $n/4$ bits, donc on va accumuler trois listes de taille $2^{n/4}$. Notons p le plus petit entier tel que $2^{n/4-p} \leq B/3$. On utilise $N = 2^p$ nœuds, qui vont stocker une portion de taille $2^{n/4-p}$ de chaque liste. Autrement dit, $2^{n/4} \approx BN$.

Générer les listes A, B et C en parallèle nécessite un temps $2^{n/2}/N \approx 2^{n/4}B$. Les listes sont ensuite triées (temps négligeable, même avec les communications), et on effectue p étapes de « diviser-pour-régner » : ceci découpe les listes en N tranches de taille $\leq B$. Il y a donc N^2 sous-problèmes indépendants à résoudre. Chacun des N nœuds répète donc N fois : obtenir les bonnes tranches (transfert de $3B$ données sur le réseau), puis résolution séquentielle locale du sous-problème (temps B^2). Un recouvrement des communications par les calculs est envisageable. En pratique, il est également judicieux, voire indispensable, d'utiliser cette technique à l'intérieur d'un même nœud pour tirer parti de la hiérarchie mémoire (faire des sous-problèmes qui tiennent en cache L1).

La taille de la machine est $NB \approx 2^{n/4}$. Le temps nécessaire à l'exécution de cette procédure est $2^{n/2}/N + NB^2 = 2^{n/4}B$. Son coût est donc $2^{n/2}B$. Cette version parallèle de l'algorithme quadratique est donc compétitive avec la technique de recherche de collision parallèle.

4.4.2 Algorithmes plus récents

Considérons, pour simplifier, une version légèrement plus générique du problème 3XOR :

— Étant donné trois tableaux de chaînes de n bits $\mathtt{A}, \mathtt{B}$ et $\mathtt{C}$, de tailles respectives A, B et C, trouver un unique (resp. trouver tous les) triplet(s) (i, j, k) tels que $\mathtt{A}[i] \oplus \mathtt{B}[j] \oplus \mathtt{C}[k] = 0$.

Tous les algorithmes pour le problème 3XOR publiés depuis 2010 résolvent en fait ce problème particulier, et ne font pas intervenir explicitement les fonctions aléatoires. L'exécution des algorithmes débute avec les trois tableaux $\mathtt{A}, \mathtt{B}$ et $\mathtt{C}$ déjà présents en mémoire. Dans ce contexte, la question du coût des calculs se pose un peu différemment, car la machine doit être assez grosse pour contenir les listes ; des algorithmes séquentiels tels que la méthode *rho* ne sont donc plus compétitifs.

L'algorithme de Joux [125] fonctionne avec des listes de taille contrainte : $A = B = 2^{n/2}/\sqrt{n/2}$ et $C = n/2$. On commence par remarquer que $x \oplus y \oplus z = 0$ si et seulement si $xM \oplus yM \oplus zM = 0$, où M est une matrice $n \times n$ inversible. On peut choisir M librement, et travailler avec les trois listes $\mathtt{A} \times M$, $\mathtt{B} \times M$ et $\mathtt{C} \times M$. L'astuce consiste à choisir la matrice M pour mettre la petite liste $\mathtt{C} \times M$ sous la forme :

$$\mathtt{C} \times M = \begin{pmatrix} 0 & \dots & 0 & \star & \dots & \star \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & \star & \dots & \star \end{pmatrix} \Big\updownarrow n/2$$
$$\underbrace{\qquad\qquad}_{n/2} \quad \underbrace{\qquad\qquad}_{n/2}$$

Alors, trouver tous les triplets 3XOR se ramène à calculer la jointure $(\mathtt{A} \times M) \bowtie (\mathtt{B} \times M)$ sur les $n/2$ premiers bits, puis à tester si le résultat est dans $\mathtt{C} \times M$. Ceci résout le problème en temps et en espace linéaire.

En vertu de la discussion de la section 4.3, ceci coûte $2^{2n/3}$. Il semble donc que l'algorithme de Joux soit plus coûteux que les méthodes naïves, bien qu'il nécessite moins d'opérations. Ceci tend à confirmer le jugement sommaire rendu sur la base du « ratio temps-mémoire » section 2.4.

L'algorithme proposé par Delaplace, Fouque et moi-même [44] trouve tous les triplets 3XOR présents dans les listes d'entrée en temps $(A+B)C/n$, sans condition sur la taille des listes. Ceci nécessite C/n calculs de jointures entre A et B, qui peuvent être faits en parallèle. En supposant que $A = B = C = 2^{n/3}$, et en reprenant l'analyse ci-dessus, ces jointures coûtent alors au moins $2^{7n/9}$.

Il semble donc que ce nouvel algorithme est plus coûteux que celui de Joux ; en fait le problème qu'il résout est plus difficile, car les listes en entrées sont plus courtes. Avec $A = B = C = 2^{n/2}$, on peut faire le « *clamping* » sur $n/4$ bits, ce qui produit une instance avec des listes de taille $2^{n/4}$. Dans cette variante, l'algorithme de [44] résout le problème 3XOR avec un coût de $2^{7n/12}$, moindre que celui de l'algorithme de Joux, mais malgré tout plus élevé que celui de l'algorithme quadratique.

4.4.3 Discussion

Il semble qu'un algorithme simple et naïf (l'algorithme quadratique) soit moins coûteux que des algorithmes plus sophistiqués qui nécessitent un nombre d'opérations plus faible. Son avantage en terme de coût provient de ce qu'il évite des opérations dont la complexité de communication est élevée, telles que le tri ou le calcul des jointures.

Que peut-on en conclure ? Le coût asymptotique plus faible de l'algorithme quadratique suggère qu'il « passe mieux à l'échelle » sur de grandes instances. Ce raisonnement « théorique » coïncide-t-il avec l'expérience pratique ?

Quelqu'un qui essaierait de résoudre une grosse instance du problème 3XOR serait confronté au choix de devoir :

— Soit implanter une version distribuée d'un algorithme de calcul de jointure, avec des données trop grosses pour tenir dans la mémoire d'un seul nœud.
— Soit faire des étapes de diviser-pour-régner jusqu'à ce que les sous-problèmes indépendants tiennent dans la mémoire d'un seul nœud, puis utiliser n'importe quel algorithme pour résoudre les sous-problèmes.

Dans la pratique, le choix est vite fait : la première possibilité entraîne avec quasi-certitude une pénalité importante en terme de communication, et seule la seconde est raisonnable. Les travaux que j'ai menés avec Bouam, Delaplace et Fouque [44, 39] montrent que l'algorithme de Joux généralisé est 3 à 5 fois plus rapide que des implantations pourtant bien optimisées de l'algorithme quadratique, à l'intérieur d'un même nœud de calcul. En effet, à l'intérieur d'un même nœud, le nombre d'opérations à effectuer joue un rôle plus important, car les coûts de communication sont moins pénalisants.

Ceci suggère une stratégie à la fois efficace empiriquement et satisfaisante théoriquement (coût faible) : faire du diviser-pour-régner tant que les sous-instances ne tiennent pas en mémoire, puis utiliser l'algorithme de Joux généralisé à l'intérieur d'un seul nœud. J'ai utilisé cette stratégie pour calculer un triplet 3XOR sur 96 bits de SHA-256 avec des moyens modestes [44], puis sur 128-bits de SHA-256 avec des moyens plus importants [39] (en détournant de leur usage « normal » des machines à miner les bitcoins). Le résultat est visible figure 4.4. Entre les deux, Leurent avait obtenu un 3XOR sur 119

Considérons les trois chaînes de 256 bits :

$$x = \texttt{2cf9b0f0 8cf86175 1f3faad0 4fee9fec 99ac4305 69a48c7c 49d779d8 c4d34321},$$
$$y = \texttt{b9c9240a 4295ff73 fcd53d9b 559ff454 64e9feb2 2d954f9c c7f12d5c 7910bbc0},$$
$$z = \texttt{d0f7153e e6ceb465 01583208 603423b5 f0e2221b 81ccce79 5b0189d5 671bdcca}.$$

On a alors :

$$\begin{aligned}
\texttt{SHA-256}(x) &= \texttt{00000000 1a3d266d 0cce284a 21ee2b70 730f8603 62b84219 9af220b9 bdaee2a7}\\
\oplus\ \texttt{SHA-256}(y) &= \texttt{00000000 1a2dea9c 30f58ff7 24f4533a e2485711 a143b883 0db5cd0a efa96f60}\\
\oplus\ \texttt{SHA-256}(z) &= \texttt{00000000 0010ccf1 3c3ba7bd 051a784a efb83f87 a5a87be7 51873c64 aac9340b}\\
\hline
&\texttt{00000000 00000000 00000000 00000000 7effee95 6653817d c6c0d1d7 f8ceb9cc}
\end{aligned}$$

FIGURE 4.4 – Un triplet 3XOR pour les 128 premiers bits de SHA-256

bits en exploitant le fait que la blockchain du système bitcoin contient de nombreux messages dont les 64+ premiers bits de l'empreinte par SHA-256 sont égaux à zéro.

4.4.4 Version creuse

Il est possible d'obtenir des algorithmes plus efficaces dans le cas d'une distribution non uniforme en entrée. Un cas intéressant est celui des vecteurs *creux*, où le nombre de bits à zéro domine le nombre de bits à 1. On peut considérer plusieurs distributions creuses : le cas « densité faible » où chaque bit en entrée est tiré indépendamment au hasard avec un biais vers zéro, ainsi que le cas « poids faible » où chaque vecteur en entrée est de poids w avec $w < n/2$.

L'intérêt de cette dernière distribution est qu'elle est facile à échantillonner par *rejection sampling*. Par exemple, pour résoudre le problème 3XOR *dense* (normal), on pourrait procéder de la manière suivante : à partir des trois listes de départ, qu'on suppose de taille $2^{\mu n}$ avec $\mu = \frac{5}{24}\log_2 5 = 0.483...$, éliminer tous les vecteurs de poids $\neq n/4$. Ceci donne trois listes plus petites, de taille moyenne $\approx 2^{\lambda n}$, avec $\lambda = \frac{5}{24}\log_2 5 - \frac{3}{4}\log_2 3 + 1 = 0.295...$. Elles contiennent *un* triplet 3XOR en moyenne. Si cette instance « creuse » du problème 3XOR pouvait être résolue en temps $\mathcal{O}\left(N^{1.69}\right)$, alors l'instance de départ du problème 3XOR dense serait résolue en temps $\approx 2^{0.498n}$, ce qui serait une percée majeure. Malheureusement, pour ma part je ne sais résoudre ce problème creux qu'en $\mathcal{O}\left(N^{1.75}\right)$ opérations. On peut interpréter ceci comme une borne inférieure conditionnelle sur la difficulté de résoudre le problème 3XOR avec poids faible.

Ceci dit, les instances creuses du problème sont sensiblement plus faciles à résoudre. Par exemple, on peut en forger une, complètement artificielle mais « amusante ». On extrait de la base de données DBLP une liste de publications tirées des conférences CRYPTO, EUROCRYPT, ASIACRYPT, FSE, PKC, CHES, SAC et des revues TOSC, TCHES. Ceci donne un peu plus de 7700 articles de recherche. Pour chacun d'entre eux, on écrit une ligne de texte avec le noms des auteurs et le titre.

On considère ensuite la fonction (où & désigne le ET bit-à-bit) :

$$F(x_1, x_2, x_3, x_4) = \texttt{SHA-512}(x_1)\ \&\ \texttt{SHA-512}(x_2)\ \&\ \texttt{SHA-512}(x_3)\ \&\ \texttt{SHA-512}(x_4).$$

Cette fonction produit des chaines de 512 bits avec densité moyenne 1/16. On cherche trois quadruplets d'articles (tous différents) tels que

$$F(x_1, x_2, x_3, x_4) \oplus F(y_1, y_2, y_3, y_4) \oplus F(z_1, z_2, z_3, z_4) = 0$$

C'est donc une instance du problème 3XOR avec densité faible. Il y a 5775 manières de ranger 12 boules numérotées dans 3 urnes indistinguables de capacité 4, donc avec nos 7700 articles on peut assembler $2^{138.5}$ paquets de 12 publications ayant une chance de satisfaire l'équation ci-dessus. De manière équivalente, on peut former 2^{47} quadruplets de publications.

```python
from hashlib import sha512

# FSE 2011
a = "Simon Knellwolf and Willi Meier: Cryptanalysis of the Knapsack Generator. (2011)"

# ASIACRYPT 2017
b = "Ran Canetti, Oxana Poburinnaya and Mariana Raykova: Optimal-Rate Non-Committing Encryption. (2017)"

# CRYPTO 2019
c = "Muhammed F. Esgin, Ron Steinfeld, Joseph K. Liu and Dongxi Liu: Lattice-Based Zero-Knowledge \
Proofs: New Techniques for Shorter and Faster Constructions and Applications. (2019)"

# FSE 2009
d = "Martijn Stam: Blockcipher-Based Hashing Revisited. (2009)"

# EUROCRYPT 1990
e = "Cees J. A. Jansen: On the Construction of Run Permuted Sequences. (1990)"

# EUROCRYPT 2013
f = 'Charles Bouillaguet, Pierre-Alain Fouque and Amandine Véber: Graph-Theoretic Algorithms for the \
"Isomorphism of Polynomials" Problem. (2013)'

# CRYPTO 2017
g = "Prabhanjan Ananth, Arka Rai Choudhuri and Abhishek Jain: A New Approach to Round-Optimal Secure \
Multiparty Computation. (2017)"

# EUROCRYPT 2001
h = "William Aiello, Yuval Ishai and Omer Reingold: Priced Oblivious Transfer: How to Sell Digital \
Goods. (2001)"

# CRYPTO 2019
i = "Navid Alamati, Hart Montgomery and Sikhar Patranabis: Symmetric Primitives with Structured \
Secrets. (2019)"

# CRYPTO 2019
j = "Shweta Agrawal, Monosij Maitra and Shota Yamada: Attribute Based Encryption (and more) for \
Nondeterministic Finite Automata from LWE. (2019)"

# EUROCRYPT 1986
k = "Christoph G. Günther: On Some Properties of the Sum of Two Pseudorandom Sequences. (1986)"

# CRYPTO 2009
l = "Susan Hohenberger and Brent Waters: Short and Stateless Signatures from the RSA Assumption. (2009)"

def H(s : str) -> int:
    """Returns the hash (SHA-512) of the string s as a 512-bit integer."""
    return int.from_bytes(sha512(s.encode('utf8')).digest(), byteorder='big')

assert (H(a) & H(b) & H(c) & H(d)) ^ (H(e) & H(f) & H(g) & H(h)) ^ (H(i) & H(j) & H(k) & H(l)) == 0
```

FIGURE 4.5 – Une application raisonnable des algorithmes pour le problème 3XOR creux.

Évaluer F sur tous les quadruplets disponibles n'est pas un problème (ça prend 240 heures CPU). Les ennuis commencent lorsqu'on envisage de « matérialiser » la liste formée par les sorties de F en l'écrivant dans un fichier : il faudrait 9 peta-octets. Ensuite, trouver un triplet nécessiterait 2^{94} accès à une table de hachage en utilisant l'algorithme quadratique, ce qui n'est pas envisageable. La figure 4.5 montre pourtant le résultat.

Exploiter le caractère creux des valeurs produites par F rend le problème facile. En effet, le poids moyen des sorties de F est 32, mais le poids moyen des triplets de densité $1/16$ qui se somment à zéro est de ≈ 2.25. Il suffit d'évaluer F sur les 2^{47} entrées et de ne garder que celles de poids ≤ 3. Il reste 5091 quadruplets de publications candidats, qui occupent 358Ko. Chercher les solutions avec l'algorithme quadratique naïf est alors très rapide.

m	n	t	Code	D	R	Niveau de sécurité	Source
10	1024	50	[1024, 524, 101]	0.51	0.099	64	[158]
11	1632	33	[1632, 1269, 67]	0.78	0.041	80*	
11	2048	27	[2048, 1751, 55]	0.85	0.027	80	[25]
11	2960	56	[2960, 2288, 113]	0.77	0.038	128*	
12	3488	64	[3488, 2720, 129]	0.78	0.037	128	
13	4608	96	[4608, 3360, 193]	0.73	0.042	192	
13	6688	128	[6688, 5024, 257]	0.75	0.038	256	[22]
13	6960	119	[6960, 5413, 238]	0.78	0.034	256	
13	8192	128	[8192, 6528, 257]	0.80	0.031	256	

FIGURE 4.6 – Jeux de paramètres pour le chiffrement McEliece. R et D indiquent le débit et la distance relative du code, respectivement. L'étoile signifie que ce niveau de sécurité est atteint en injectant une ou deux erreurs de plus que la moitié de la distance minimale du code ; par conséquent il faut un décodage en liste et de la redondance dans le clair.

4.5 Le décodage des codes linéaires aléatoires

> Our real-world cryptanalysis shows that memory access certainly has to be taken into account when computing bit-security [...] an algorithm with time T and memory M has cost $T \log 2M$.
>
> A. Esser, A. May et F. Zweydinger, 2021 [99]

Les algorithmes de décodage des codes linéaires aléatoires sur $\mathbb{F}_2$ cherchent des solutions creuses à des systèmes linéaires sous-définis : il s'agit principalement de résoudre le *syndrome decoding problem*, c'est-à-dire trouver un vecteur e, de poids de Hamming borné (inférieur à w), tel que $He = s$. Ici, H est la matrice de parité d'un code linéaire (aléatoire) $[n, k, d]$, de taille $(n-k) \times n$, et s est un syndrome quelconque. Le débit du code, $R = \frac{k}{n}$, joue un rôle particulièrement important dans la complexité des algorithmes de décodage.

On choisit généralement $w = (d-1)/2$ (« *half-distance decoding* ») comme borne sur le nombre d'erreurs, car ceci garantit qu'on peut décoder de manière non-ambiguë. La complexité calculatoire du décodage dépend alors des trois paramètres n, k et d. Mais dans le cas des codes linéaires aléatoires, ceux-ci sont liés approximativement par la borne de Gilbert-Varshamov : $R \approx 1 - H\left(\frac{d}{n}\right)$, où H désigne la fonction d'entropie binaire [3]. En particulier, on trouve que $H^{-1}(1-R) \approx \frac{d}{n}$.

L'application cryptographique la plus directe de ces algorithmes consiste à attaquer le système de chiffrement McEliece [158]. Les paramètres initialement proposés en 1978 étaient $[1024, 524, 101]$. En effet, le code de Goppa binaire défini par un polynôme de degré t (sans facteurs multiples) sur le corps à 2^m éléments donne généralement un code $[n, n - tm, 2t + 1]$ (il faut que $n \leq 2^m$). Plusieurs jeux de paramètres ont été proposés dans le passé, et ils sont collectés dans le tableau 4.6.

Par ailleurs, le site `decodingchallenge.org` propose un défi qui consiste à trouver un mot de poids le plus faible possible dans un code de débit 1/2 et de longueur 1280 bits. La borne de Gilbert-Varshamov donne une distance minimale de 144.

3. Ceci peut se retrouver facilement grâce à l'heuristique suivante : si les 2^k mots d'un code de longueur n étaient uniformément aléatoires et indépendants, alors il y en aurait en moyenne $\binom{n}{w} 2^{k-n}$ de poids w. Cette grandeur doit être à peu près égale à un lorsque le poids est la distance minimale. Utiliser l'équation ($\star$) (ci-dessous) donne $2^{nH(d/n)} = 2^{n-k}$, donc $H(d/n) = 1 - k/n$.

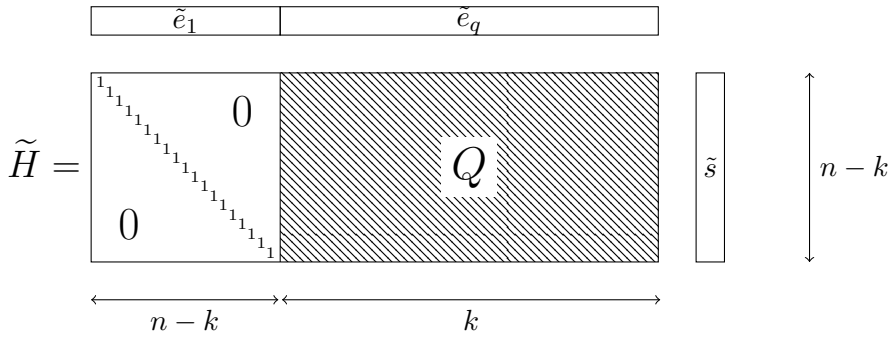


FIGURE 4.7 – Un cadre algorithmique pour l'*Information Set Decoding*.

4.5.1 Algorithme de Prange/Lee-Brickell

Tous les algorithmes efficaces fonctionnent sur la base de l'*Information Set Decoding*. Cela revient à faire une hypothèse sur la distribution des bits dans e, puis l'exploiter pour retrouver e efficacement. Effectuer une permutation aléatoire des positions de e (donc des colonnes de H) va satisfaire cette hypothèse avec une certaine probabilité, et il faudra recommencer tant que ce n'est pas le cas. Ce sont donc des algorithmes *Las Vegas*.

Les colonnes de la matrice de parité sont permutées (en multipliant à droite par une matrice de permutation P), puis le résultat est mis sous forme échelonnée réduite, en multipliant à gauche par une matrice inversible ; cela donne $\widetilde{H} = THP$. Le problème de départ se ramène alors à $\widetilde{H}\tilde{e} = \tilde{s}$ avec $s = T\tilde{s}$, $Pe = \tilde{e}$ et toujours $|\tilde{e}| \leq w$.

Le vecteur $\tilde{e}$ se découpe alors naturellement en deux parties : $\tilde{e}_1$ qui correspond aux $n-k$ premières coordonnées et $\tilde{e}_q$ qui correspond aux k dernières.

Si $\tilde{e}_q = 0$, alors $\tilde{e}_1 = \tilde{s}$ et le problème n'est soluble que si le poids de $\tilde{s}$ est inférieur ou égal à w. Ceci aboutit à un algorithme simple : permuter aléatoirement les colonnes de la matrice, mettre sous forme échelonnée réduite, tester si $|\tilde{s}| \leq w$; tant que ce n'est pas le cas, recommencer. Ceci est l'algorithme de Prange [171]. Supposons qu'il existe une solution $e^\star$; la procédure termine lorsque la permutation « pousse » tous les bits de $e^\star$ sur les $n-k$ premières colonnes. L'espérance du nombre d'itérations est donc inférieure à :

$$N = \binom{n}{w} / \binom{n-k}{w}.$$

En posant $w = d/2$, en utilisant la borne de Gilbert-Varshamov puis en exploitant l'encadrement

$$\frac{2^{nH(x)}}{\sqrt{8nx(1-x)}} \leq \binom{n}{xn} \leq \frac{2^{nH(x)}}{\sqrt{2\pi nx(1-x)}}, \qquad (0 < x < 1/2) \qquad (\star)$$

qu'on va sommairement traduire par $\binom{n}{xn} \approx 2^{nH(x)}$, on obtient finalement :

$$N \approx 2^{nF(R)}, \qquad \text{avec} \qquad F(R) = H\left(\frac{H^{-1}(1-R)}{2}\right) - (1-R)H\left(\frac{H^{-1}(1-R)}{2(1-R)}\right)$$

Cette présentation des choses, qui permet de sortir n de l'évaluation de la complexité, est extrêmement commode et sera reprise systématiquement ensuite. La fonction $F(R)$ admet un maximum de 0.0575... pour $R = 0.4687...$. Par conséquent, on peut affirmer que l'algorithme termine en moins $2^{0.0575n}$ itérations, quelle que soit la valeur de R (on retrouve la valeur annoncée pour le coût du *Half-distance decoding* avec l'algorithme de Lee-Brickell dans le tableau 2.4). En pratique, si on s'attaque à un code qui a un débit différent, la complexité asymptotique peut être plus faible. En tout cas, le nombre d'itérations est exponentiellement grand. Ces itérations peuvent être faites en parallèle, et chacune a un temps d'exécution polynomial. Le coût est donc facile à calculer, et il est identique au nombre d'opérations.

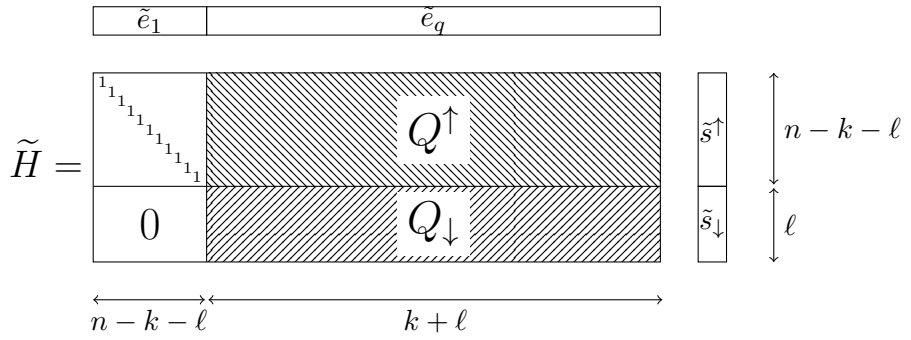


FIGURE 4.8 – Un cadre algorithmique général pour l'*Information Set Decoding*.

On peut améliorer un peu les performances en relâchant la contrainte sur e_q : au lieu d'imposer $\tilde{e}_q = 0$, on peut imposer que $\tilde{e}_q$ soit de poids p (petit) ; $\tilde{e}_1$ est alors de poids inférieur ou égal à $w - p$. Une solution potentielle s'écrit alors $\tilde{e}_1 + Q\tilde{e}_q = \tilde{s}$. En d'autres termes, on cherche une combinaison linéaire de p colonnes de Q qui coïncide *approximativement* avec $\tilde{s}$. En effet, si $|Q\tilde{e}_q + \tilde{s}| \leq w - p$, alors on peut poser $\tilde{e}_1 = Q\tilde{e}_q + \tilde{s}$ et on a bien une solution. En gros, $\tilde{e}_1$ permet de compenser la différence entre la combinaison linéaire de p colonnes de Q et le syndrome $\tilde{s}$.

La solution la plus simple pour trouver un tel $\tilde{e}_q$ consiste à faire une recherche exhaustive sur les $\binom{k}{p}$ combinaisons linéaires possibles de p colonnes de Q. Ceci ne va aboutir que si la permutation des colonnes amène bien $w - p$ erreurs dans les $n - k$ premières coordonnées et p dans les k dernières. C'est l'algorithme de Lee-Brickell [142]. En pratique, la valeur optimale de p est généralement $p = 1$ ou $p = 2$ (ce qui équilibre à peu près le coût de la recherche exhaustive avec celui de l'élimination gaussienne). L'accélération obtenue par rapport à l'algorithme de Prange n'est que polynomiale.

L'algorithme est assez simple à implanter et fonctionne sans mémoire. Une amélioration pratique due à Canteaut et Chabaud [68] mérite d'être notée : elle consiste ne pas recalculer une forme échelonnée à chaque itération, ce qui est coûteux, mais à changer *une* colonne à la fois. Dans la figure 4.5, on choisit aléatoirement *une colonne* dans Q, on trouve un indice i tel que $Q_{ij} \neq 0$, puis on permute cette colonne avec la i-ème et on ré-echelonise. Ceci augmente le nombre d'itérations attendu, mais diminue le coût des itérations.

Au passage, d'un point de vue de l'implantation, ceci a beaucoup d'effets intéressants. En effet, la re-échelonisation a pour effet... de défaire la permutation des colonnes. Il n'est du coup plus nécessaire de permuter les colonnes de la matrice (ce qui est très inefficace) et il suffit de mémoriser la permutation qui a été réalisée. De plus, la procédure d'échelonisation elle-même est particulièrement simple : poser $A_{ij} \leftarrow 0$; pour tout k, faire $A_k \leftarrow A_k + A_{kj}A_i$; enfin, poser $A_{ij} \leftarrow 1$.

4.5.2 Un cadre algorithmique général

Les algorithmes ultérieurs reposent sur une idée de Leon [145], Stern [179] et Dumer [94]. On présente ici une version légèrement améliorée dû à Finiasz et Sendrier [104] ; cette amélioration a la même complexité asymptotique que le *Ball-Collision Decoding* de Bernstein, Lange et Peters [26]. Elle utilise une structure un peu plus générale, en ne mettant que partiellement la matrice génératrice sous forme échelonnée réduite, comme illustré figure 4.8.

Comme on veut $\widetilde{H}\tilde{e} = \tilde{s}$, si on impose une contrainte $|\tilde{e}_q| = p$ sur le poids de $\tilde{e}_q$ (comme précédemment), alors le problème de départ se découpe en deux sous-problèmes indépendants :

$$\begin{aligned} \tilde{e}_1 + Q^{\uparrow}\tilde{e}_q &= \tilde{s}^{\uparrow} & |\tilde{e}_1| &\leq w - p \\ Q_{\downarrow}\tilde{e}_q &= \tilde{s}_{\downarrow} & |\tilde{e}_q| &= p. \end{aligned}$$

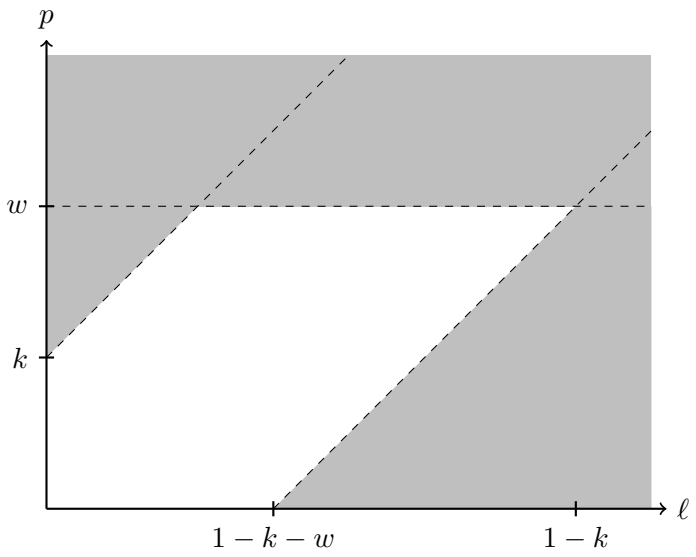


FIGURE 4.9 – Domaine admissible pour les paramètres p et ℓ dans les procédures d'ISD.

Les valeurs possibles de p et ℓ obéissent aux limites suivantes :

$$(\chi)\begin{cases} p & \leq k+\ell \\ p & \leq w, \\ \ell & \leq n-k \\ w-p & \leq n-k-\ell \end{cases}$$

En fait la 3ème est une conséquence de la 2ème et de la 4ème. Du coup, la zone admissible se présente comme dans la figure 4.9.

Ceci ouvre la possibilité suivante : trouver les solutions de $Q_{\downarrow}\tilde{e}_q = \tilde{s}_{\downarrow}$, avec $|\tilde{e}_q| = p$, puis pour chacune d'entre elle tester si $\tilde{s}^{\uparrow} + Q^{\uparrow}\tilde{e}_q$ est bien de poids inférieur ou égal à $w - p$. Le cas échéant, cela donne une valeur admissible de $\tilde{e}_1$.

Cette procédure ramène donc le fait de trouver *une* solution à l'instance initiale du *syndrome decoding problem* pour un code $[n, k]$ avec une borne w au fait de trouver *toutes* les solutions d'un nombre exponentiel d'instances sensiblement plus petites du même problème : le code est $[k+\ell, \ell]$ et la borne est p. Le débit de ces petits codes est bien plus faible, car ℓ est typiquement assez petit devant k.

Les algorithmes ultérieurs diffèrent par la manière de résoudre ces sous-instances. Effectuer une recherche exhaustive sur les $\binom{k+\ell}{p}$ solutions possibles nous ramènerait à une variante de l'algorithme de Lee-Brickell.

Le nombre d'itérations moyen pour obtenir la bonne répartition des erreurs $(w-p, p)$ dans $\tilde{e}$ est :

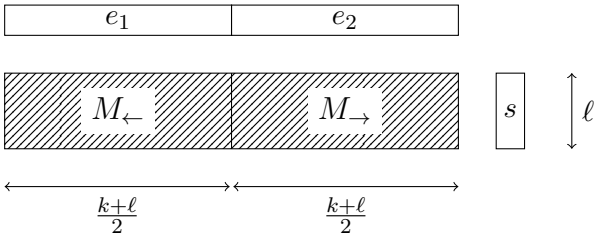
$$N \approx \frac{\binom{n}{w}}{\binom{k+\ell}{p}\binom{n-k-\ell}{w-p}}.$$

Moralement, on impose une restriction sur l'ensemble des solutions cherchées (elles doivent coïncider avec le syndrome *exactement* sur ℓ lignes et seulement approximativement sur les $n-k-\ell$ autres lignes). En échange de cette restriction, on pourra obtenir des procédures plus efficaces.

Dans la suite de cette section, on discute des moyens de résoudre l'instance suivante du *syndrome decoding problem* : trouver tous les vecteurs x tels que $Me = s$ avec $|e| \leq p$; la matrice M est de dimension $\ell \times (k+\ell)$ et supposée aléatoire. A priori, ℓ est sensiblement plus petit que k. Il faut noter qu'on s'attend à ce que le problème possède un nombre de solutions de l'ordre de $2^{-\ell}\binom{k+\ell}{p}$.

4.5.3 Algorithme de Leon/Stern/Dumer

On peut faire baisser le nombre d'opérations, comparé à la recherche exhaustive, par la technique du *collision decoding*, c'est-à-dire par un *meet-in-the-middle*. On découpe le paquet de $k+\ell$ colonnes en deux moitiés égales :



On exige ensuite que les p bits de e soit équitablement répartis entre les deux moitiés, autrement dit que $|e_1| = |e_2| = p/2$. Ceci se produit avec une probabilité assez élevée qu'on peut minorer grâce à $(\star)$:

$$\binom{(k+\ell)/2}{p/2}^2 / \binom{k+\ell}{p} \geq \sqrt{\frac{\pi}{8p}\frac{k+\ell}{k+\ell-p}}.$$

L'algorithme fonctionne comme suit :

1. Assembler la liste $\mathcal{L}_1$ des combinaisons linéaires de $p/2$ colonnes de $M_\leftarrow$:
$$\mathcal{L}_1 = \{(y \mapsto e_1) \ : \ |e_1| = p/2, y = M_\leftarrow e_1\}$$
2. Assembler la liste $\mathcal{L}_2$ des combinaisons linéaires de $p/2$ colonnes de $M_\rightarrow$.
$$\mathcal{L}_2 = \{(y \mapsto e_2) \ : \ |e_2| = p/2, y = s + M_\rightarrow e_2\}$$
3. Effectuer une jointure pour extraire la liste $\mathcal{L}$ des solutions :
$$\mathcal{L} = \mathcal{L}_1 \bowtie \mathcal{L}_2 = \{(e_1, e_2) \ : \ (x \mapsto e_2) \in \mathcal{L}_1, (y \mapsto e_2) \in \mathcal{L}_2, x = y\}$$

Le point est que si $(e_1, e_2) \in \mathcal{L}$, alors $M_\leftarrow e_1 = s + M_\rightarrow e_2$, donc $Me = s$ avec $e = (e_1, e_2)$ et $|e| = p$. Ceci est l'algorithme de Léon/Stern/Dumer [179, 145, 94].

À de petits facteurs près, les listes $\mathcal{L}_1$ et $\mathcal{L}_2$ sont de taille $\binom{(k+\ell)/2}{p/2} \approx \binom{k+\ell}{p}^{1/2}$, et on s'attend à ce que le nombre de solutions renvoyées soit de l'ordre de $2^{-\ell}\binom{k+\ell}{p}$. À de petits facteurs près, on les trouve donc toutes. Le temps total d'exécution de la procédure est grosso-modo la somme de la taille des listes et du nombre de solutions. On s'attend à ce que ceci soit à peu près minimisé lorsque

$$2^\ell \approx \binom{k+\ell}{p}^{\frac{1}{2}}.$$

La complexité en espace est donnée par la taille des listes.

Complexité de l'ISD basé sur la procédure de Leon/Stern/Dumer

> With only a bit of exaggeration, we can say that, if you formulate a practical problem as a convex optimization problem, then you have solved the original problem
>
> S. Boyd et L. Vandenberghe [63]

La complexité totale est

$$\frac{\binom{n}{w}}{\binom{n-k-\ell}{w-p}\binom{k+\ell}{p}}\left(\binom{k+\ell}{p}^{\frac{1}{2}} + 2^{-\ell}\binom{k+\ell}{p}\right)$$

Ceci néglige la durée des mises sous forme échelonnée, qui n'ont pas un coût nul en pratique. Mais asymptotiquement, c'est suffisant.

Concrètement, trouver les meilleures valeurs $p \leq w$ et $\ell \leq n - k$ ne doit pas être trop difficile, car ce sont des entiers qui ne peuvent prendre qu'un nombre limité de valeurs possibles. Pour casser les paramètres originaux du chiffrement McEliece ($n = 1024, k = 524, t = 101$), Bernstein, Lange et Peters ont utilisé $p = 4$ et $\ell = 20$.

On peut néanmoins essayer d'optimiser ceci asymptotiquement. Pour cela, il faut faire disparaître n en utilisant $(\star)$ puis en posant $\overline{p} = p/n$ et $\overline{\ell} = \ell/n$. Le nombre d'opérations est alors $2^{nF(R,\overline{w},\overline{p},\overline{\ell})}$ et l'exposant est donné par

$$F(R,\overline{w},\overline{p},\overline{\ell}) = \underbrace{I(R,\overline{w},\overline{p},\overline{\ell})}_{\#\text{ itérations}} + \underbrace{C(R,\overline{w},\overline{p},\overline{\ell})}_{\text{Coût d'une itération}}$$

Avec

$$I(R,\overline{w},\overline{p},\overline{\ell}) = H(\overline{w}) - (R+\overline{\ell})H\left(\frac{\overline{p}}{R+\overline{\ell}}\right) - (1-R-\overline{\ell})H\left(\frac{\overline{w}-\overline{p}}{1-R-\overline{\ell}}\right)$$
$$C(R,\overline{w},\overline{p},\overline{\ell}) = \max\left(\frac{R+\overline{\ell}}{2}H\left(\frac{p}{R+\overline{\ell}}\right), (R+\overline{\ell})H\left(\frac{p}{R+\overline{\ell}}\right) - \overline{\ell}\right)$$

Le problème consiste d'abord, pour R et $\overline{w}$ fixés, à trouver $\overline{p}, \overline{\ell}$ qui minimisent $F(R,\overline{w},\overline{p},\overline{\ell})$ sous les contraintes qui définissent le domaine admissible (χ). Ceci permet par exemple de remplir la colonne « McEliece » du tableau 2.4. Dans un deuxième temps, pour déterminer le pire cas de l'algorithme en *half-distance decoding*, on calcule :

$$F(R,\overline{w}) = \min_{\overline{p},\overline{\ell}} F(R,\overline{w},\overline{p},\overline{\ell})$$
$$F_{1/2}(R) = \min_{\overline{p},\overline{\ell}} F(R, 0.5H^{-1}(1-R),\overline{p},\overline{\ell})$$
$$W = \max_{R} F_{1/2}(R)$$

Ces tâches d'optimisation numérique posent des problèmes théoriques et pratiques « amusants ». On sait depuis 1968 que le problème qui consiste à tester si une fonction arbitraire peut devenir négative est indécidable, même pour des fonctions en une seule variable réelle appartenant à des classes assez restreintes [174]. De plus, la précision des nombres à virgule flottante utilisés est finie, donc on pourrait aussi avoir des problèmes de fiabilité numérique. On est donc en droit de se demander s'il existe un algorithme (efficace) capable de calculer $F(R,\overline{w})$ et à plus forte raison W. Dans le tableau 2.4, on voit que chaque algorithme améliore l'exposant du précédent de 3–10%, donc on voudrait être sûr qu'on les calcule avec une bonne précision (de l'ordre de 10^{-5} ou 10^{-6}).

Des raisonnements issus du domaine de l'*optimisation convexe* permettent de s'en sortir dans certains cas. Grosso modo, les problèmes d'optimisation convexes qui ont la forme suivante sont à peu près solubles algorithmiquement :

$$\begin{array}{rll} \text{Minimiser} & f_0(x) & \\ \text{Sous les contraintes} & f_i(x) \leq 0, & i = 1,\ldots,m \\ & Ax = b & \end{array}$$

où $f_0,\ldots,f_m : \mathbb{R}^n \to \mathbb{R}$ sont convexes. Les contraintes déterminent un ensemble admissible convexe.

Il est bien connu que H est concave. Un petit raisonnement basé sur les opérations qui préservent la convexité établit que $I(R,\overline{w},\overline{p},\overline{\ell})$ est convexe en $(R,\overline{p},\overline{\ell})$ pour un $\overline{w}$ fixé.

Malheureusement, le même raisonnement permet d'établir que $C(R, \overline{w}, \overline{p}, \overline{\ell})$ est concave. Mais il se trouve que, dans ce cas précis, la somme des deux $C(R, \overline{w}, \overline{p}, \overline{\ell}) + I(R, \overline{w}, \overline{p}, \overline{\ell})$, elle, est bel et bien convexe.

Calculer $F(R, \overline{w})$ revient donc à minimiser une fonction convexe sur le polyèdre convexe (χ), ce qui est déjà plus simple. Ceci garantit déjà qu'un minimum local est un minimum global. De plus, un algorithme classique (la méthode de l'ellipsoïde) permet de produire une séquence de points qui converge vers le minimum exponentiellement vite. Ceci fonctionne sous la seule condition que la fonction à minimiser est convexe et le domaine admissible aussi. De plus, si F est convexe et différentiable, alors $\nabla F(x) = 0$ garantit que x minimise F — on a alors une notion de *certificat* de la solution... à condition que F soit différentiable sur x, et que x soit à l'intérieur du domaine de faisabilité.

Les problèmes commencent lorsqu'on se rend compte que, dans le cas qui nous intéresse, la fonction $F(R, \overline{w}, \overline{p}, \overline{\ell})$ n'est pas différentiable au point qu'on soupçonne d'être optimal. Ceci a lieu parce que $\max(f(t), g(t)) = (f(t) + g(t) + |f(t) - g(t)|)/2$, et on voit que la dérivée n'est pas définie lorsque $f(t) = g(t)$ et $f'(t) \neq g'(t)$. Or dans notre cas, c'est précisément lorsque les deux termes dans le max sont égaux qu'on s'attend à atteindre la solution optimale. En fait, la fonction objectif n'est pas lisse et son gradient présente une grosse discontinuité à cet endroit là. Les méthodes numériques usuelles divergent parfois sur de tels exemples [74]. De plus, on peut rapidement vérifier qu'il y a des valeurs de $\overline{p}, \overline{\ell}$ situés en dehors du domaine admissible (χ) qui rendent l'exposant négatif. Tout ceci fait que la technique qui consiste à certifier la solution en examinant le gradient au point considéré n'est pas viable.

Pour tenter de certifier la solution, une possibilité consiste à se débarrasser de ce max pénible. Deux méthodes semblent possibles. La première, un peu *ad hoc*, consiste à imposer franchement que les deux termes du max soient égaux. On aura donc :

$$\ell = \frac{R + \overline{\ell}}{2} H\left(\frac{\overline{p}}{R + \overline{\ell}}\right), \quad \text{autrement dit} \quad \overline{p} = (R + \overline{\ell}) H^{-1}\left(\frac{2\overline{\ell}}{R + \overline{\ell}}\right)$$

On note que $\overline{p}$ est une fonction croissante de $\overline{\ell}$, donc la plus grande valeur de $\overline{\ell}$ autorisée est celle qui donne $\overline{p} = \overline{w}$. On peut alors se ramener à minimiser la fonction d'une seule variable

$$F_{R,\overline{w}}(\ell) = H(\overline{w}) - (1 - R - \overline{\ell}) H\left(\frac{\overline{w} - (R + \overline{\ell}) H^{-1}\left(\frac{2\overline{\ell}}{R+\overline{\ell}}\right)}{1 - R - \overline{\ell}}\right) - \overline{\ell}$$

sur l'intervalle défini par $0 \leq \overline{\ell}$ et $(R + \overline{\ell}) H^{-1}\left(\frac{2\overline{\ell}}{R+\overline{\ell}}\right) \leq \overline{w}$. On peut démontrer que la fonction est toujours (strictement) convexe, donc un algorithme simple comme la méthode du nombre d'or viendra à bout du problème d'optimisation en une seule variable.

La deuxième solution, qui est plus générale et ne nécessite pas d'hypothèse, consiste à introduire deux nouvelles variables T et M qui représentent respectivement la complexité en nombre d'opérations et en espace d'une itération (donc la valeur du max). Ces variables peuvent être forcées à la bonne valeur en ajoutant les contraintes idoines. On introduit aussi un paramètre κ qui représente le « coût de la mémoire » : on le fixe à 0 pour optimiser le nombre d'opérations, et à 1/3 pour optimiser le coût. On cherche alors à minimiser la fonction (convexe) :

$$F(R, \overline{w}, \overline{p}, \overline{\ell}, C, T) = \underbrace{I(R, \overline{w}, \overline{p}, \overline{\ell})}_{\#\text{ itérations}} + T + \kappa M$$

Sous les contraintes suivantes :

$$\begin{aligned} 2M &\geq (R+\overline{\ell})H\left(\frac{p}{R+\overline{\ell}}\right) \\ T &\geq M \\ T &\geq 2M - \overline{\ell} \\ 0 &\leq \overline{p} \leq \overline{w} \\ 0 &\leq \overline{\ell} \leq 1 - R - \overline{w} + \overline{p} \end{aligned}$$

Cette reformulation du problème a un inconvénient majeur : la première contrainte définit un ensemble admissible concave. En effet, elle affirme que M appartient à l'épigraphe de la fonction à droite de l'inégalité, or cette dernière est concave. On est donc sorti du domaine favorable de l'optimisation convexe pour entrer dans le territoire incertain de l'optimisation non-linéaire.

Ceci dit, cette reformulation du problème a des avantages. Le premier est que la fonction objectif et les fonctions qui définissent les contraintes sont lisses et différentiables. Le deuxième avantage est qu'on dispose d'un moyen de certifier les solutions, à défaut de savoir comment les trouver. En effet, on peut s'intéresser au problème dual. Si une procédure magique nous fournissait une solution x réputée optimale du problème de départ, alors on pourrait optimiser la relaxation Lagrangienne en x du problème, donc déterminer les multiplicateurs de Lagrange et vérifier que les conditions de Karush-Kuhn-Tucker sont satisfaites. Le cas échéant, cela certifie l'optimalité de la solution (le lecteur intéressé pourra consulter un ouvrage classique sur la question [63]). En pratique, la méthode magique utilisée pour trouver l'optimal sera une méthode itérative qui converge (dans les bonnes conditions) vers un optimum local. Le troisième avantage de cette formulation est sa généricité : elle est très facilement adaptable à d'autres variantes d'*information set decoding* (en changeant les contraintes qui définissent T et M).

Le problème du calcul fiable de l'exposant W du pire cas est plus compliqué. Il semble (visuellement) que $F(R)$ a une forme concave, mais je n'ai pas de preuve de cette assertion. Si c'était établi, cela garantirait qu'une procédure simple, telle que la méthode du nombre d'or, trouverait le maximum global de $F(R)$, donc l'exposant du pire cas.

Essayons de voir ce que ça donne en pratique avec scipy [195], une bibliothèque libre et populaire pour le calcul scientifique. Comme on impose des bornes sur les variables et des contraintes linéaires, ceci utiliserait par défaut une implantation efficace (en Fortran) de la méthode de l'optimisation quadratique successive écrite au tournant des années 1990 par Kraft [136]. En fait, l'utilisation de cet algorithme est un art délicat et cela pose beaucoup de problèmes pratiques. Par conséquent, il vaut mieux forcer l'utilisation d'une méthode de point intérieur [67] un peu moins efficace, mais qui certifie le résultat comme expliqué ci-dessus. Dans tous les cas, c'est un exercice délicat : l'algorithme peut ne pas converger, ou bien converger vers une solution erronée à cause d'un défaut de précision, parce que le problème est mal conditionné, parce que le point de départ est mal choisi, etc. Les algorithmes peuvent évaluer la fonction objectif et/ou les fonctions qui définissent les contraintes en dehors de leur domaine de définition, ou bien pile sur le bord du domaine admissible (où les dérivées sont peut-être mal définies), etc. Il arrive que fournir un moyen explicite de calculer les dérivées (qui est donc plus précis qu'une estimation numérique) empêche la convergence ! Tout ça pour dire que c'est un art plus qu'une science.

```
from math import log2, inf, log
from scipy.optimize import *

def H(x):
    """Entropie binaire (avec prolongement analytique)"""
    if x <= 0 or x >= 1:
        return 0
    return -(x * log2(x) + (1-x) * log2(1 - x))

def Hp(x, y):
```

```
    """Perspective de la fonction entropie"""
    return x * H(y / x)

def Hinv(y):
    """Fonction réciproque de l'entropie binaire"""
    return brentq(lambda x: H(x) - y, 0, 1/2)

def cost(args, R, w, kappa):
    """Fonction objectif du problème d'optimisation"""
    p, l, T, M = args
    return  H(w) - Hp(R + l, p) - Hp(1 - R - l, w - p) + T + kappa * M

def optimize_cost(R, w, kappa):
    """Trouve (p, l) qui optimisent le coût, à (R,w) fixés"""

    # contrainte non-linéaire
    def constraints(args):
        p, l, T, M = args
        return M - 0.5 * Hp(R + l, p)
    nlc = NonlinearConstraint(constraints, lb=0, ub=inf)

    # Contraintes linéaires
    A = [[1, -1, 0, 0],       # p <= R + l
         [-1, 1, 0, 0],       # w - p <= 1 - R - l
         [0, 0, -1, 1],       # M <= T
         [0, -1, -1, 2]]      # 2M - l <= T
    lc = LinearConstraint(A, lb=-inf, ub=[R, 1 - R - w, 0, 0], keep_feasible=True)
    bounds = Bounds(lb=0, ub=[w, inf, inf, inf], keep_feasible=True)

    x0 = (min(R, w) / 2, (1 - R - w) / 2, 2, 1)      # intérieur de la zone de faisabilité
    return minimize(cost, args=(R, w, kappa), x0=x0, bounds=bounds,
                    constraints=[nlc,lc], method='trust-constr')

def half_distance_decoding(R, kappa):
    w = 0.5 * Hinv(1 - R)                    # borne de G-V
    sol = optimize_cost(R, w, kappa)         # exposant atteint pour ce R
    assert sol.success
    return sol.fun

# pire cas, nombre d'opérations
print(minimize_scalar(lambda R: -half_distance_decoding(R, 0), bounds=(0, 1), method='bounded'))

# pire cas, coût
print(minimize_scalar(lambda R: -half_distance_decoding(R, 1/3), bounds=(0, 1), method='bounded'))
```

Il faut noter qu'on ne fournit pas explicitement à l'algorithme d'optimisation de procédures pour évaluer le gradient de la fonction objectif ou des contraintes non-linéaires (ils vont être estimés numériquement, et de toute façon ils sont lisses). On obtient un exposant de 0.055573 pour $R = 0.46565$ avec les paramètres $\overline{p} = 0.00324031$ et $\overline{\ell} = 0.01401022$, si on cherche à minimiser le nombre d'opérations. Ceci est conforme à la littérature publiée. Au passage, ces raisonnements asymptotique suggèrent $p \approx 3$ et $\ell \approx 13$ pour les paramètres originaux de McEliece $[1024, 524, 101]$, ce qui n'est pas si éloigné que ça des valeurs concrètes utilisées pour les casser ($p = 4, \ell = 20$).

Dans tous les cas, on observe que la solution optimale consiste à poser $T = M = 2M - \overline{\ell}$, donc à équilibrer les deux termes du max.

Si on cherche à optimiser le coût, alors on ne peut pas faire pire que l'algorithme de Lee-Brickell, car il suffit de poser $p = 0$ et on retombe dessus. La procédure numérique trouve un exposant de 0.0574924 dans ce cas, c'est-à-dire diminué de 0.035% comparé à l'algorithme de Prange. Les paramètres optimaux sont $\overline{p} \approx 1/5000$. En terme de coût asymptotique, il semble donc que l'algorithme de Leon/Stern/Dumer n'ait pas d'intérêt comparé à l'algorithme de Prange/Lee-Brickell. En pratique, les choses sont différentes. Pour les tailles de code qui sont potentiellement solubles en pratique, les

valeurs de ℓ qu'il serait raisonnable d'utiliser sont assez faibles, et les données tiennent facilement en RAM. Par conséquent, raisonner en nombre d'opérations est acceptable dans ce cas-là.

Pour les codes « de type McEliece » avec $R = 0.75$ et $\overline{w} = 0.02$, les paramètres optimaux sont :

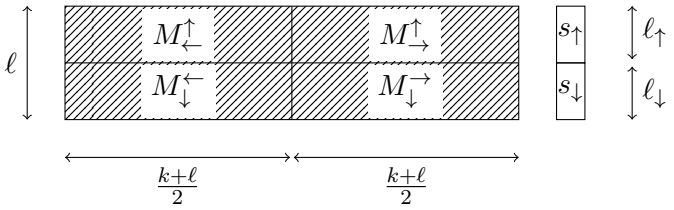
Minimisation	Nombre d'opérations	Coût
$\overline{p}$	0.002583	0.0001665
$\overline{\ell}$	0.0125	0.00119
# Itérations	0.02666	0.03969
# Opérations	0.03914	0.04055
Mémoire	0.0125	0.00119
Coût	0.0433	0.04088

Minimiser le nombre d'opération (atteindre $2^{0.03914n}$) nécessite un choix de paramètre différent de celui qui minimise le coût (qui donne, lui, $2^{0.04055n}$ opérations).

L'article de Bernstein, Lange et Peters [25] contient une description partielle de leur implantation, et fournit des détails pratiques sur la façon d'implanter la recherche de collision et le calcul de la jointure. Il existe quelques autres implantations, peut-être pas aussi compétitives, mais publiques, elles, dues à Landais [140] et Vasseur entre autres.

4.5.4 Algorithme de May-Meurer-Thomae [156]

L'algorithme MMT est une amélioration de celui de Leon/Stern/Dumer. Le schéma général est le même : on construit deux listes $\mathcal{L}_1$ et $\mathcal{L}_2$ puis on calcule leur jointure. Mais cette fois, on se débrouille pour que les deux listes soient composées de vecteurs qui coïncident déjà partiellement avec le syndrome visé. Pour pouvoir imposer cette contrainte, on relâche la définition des listes : $\mathcal{L}_i$ n'est plus limité aux combinaisons d'une seule moitié des colonnes. Ceci laisse plus de liberté pour imposer des contraintes supplémentaires qui permettent de gagner en efficacité. La présentation originale de ce raisonnement repose sur le concept des *représentations*.



Pour améliorer la procédure de recherche de collision, on découpe les ℓ lignes en deux paquets de taille $\ell^{\uparrow}$ et $\ell_{\downarrow}$. Si x est un vecteur de $\{0,1\}^{\ell}$, alors on le décompose en $x = (x^{\uparrow}, x_{\downarrow})$. L'algorithme fonctionne encore en produisant deux listes $\mathcal{L}_1$ et $\mathcal{L}_2$ dont on va calculer la jointure. On tire aléatoirement deux vecteurs u et v de taille ℓ tels que $u + v = s$, puis on pose :

$$\mathcal{L}_1 = \{(y^{\uparrow} + u^{\uparrow} \mapsto x) \mid y = Mx, |x| = p/2, y_{\downarrow} = u_{\downarrow}\},$$
$$\mathcal{L}_2 = \{(y^{\uparrow} + v^{\uparrow} \mapsto x) \mid y = Mx, |x| = p/2, y_{\downarrow} = v_{\downarrow}\}.$$

Les vecteurs de la liste $\mathcal{L}_1$ décrivent des combinaisons linéaires de $p/2$ colonnes qui se somment à $u_{\downarrow}$ sur les $\ell_{\downarrow}$ lignes du bas, tandis que ceux de $\mathcal{L}_2$ décrivent des combinaisons linéaires qui se somment à $v_{\downarrow}$ sur les mêmes lignes. Par conséquent, la somme de n'importe quels $x \in \mathcal{L}_1$ et $y \in \mathcal{L}_2$ donne un vecteur de poids inférieur ou égal à p qui se somme à $s_{\downarrow}$ sur les lignes du bas. La jointure des deux listes est :

$$\mathcal{L} = \{x + y \mid (a \mapsto x) \in \mathcal{L}_1, (b \mapsto y) \in \mathcal{L}_2, a = b\},$$

et les éléments de $\mathcal{L}$ sont bien les solutions du sous-problème de *syndrome decoding*, car la jointure garantit qu'ils se somment en $s^{\uparrow}$ sur les lignes du haut.

Dans l'algorithme de Leon/Stern/Dumer, il fallait qu'un vecteur donné de poids p se décompose en deux moitiés de poids $p/2$, or ceci n'arrive qu'avec une certaine probabilité. À contrario, ici un vecteur

de poids p peut s'écrire de $\binom{p}{p/2} \approx 2^p$ manières comme somme de deux vecteurs de poids $p/2$, ce qui permet d'imposer la contraintes que les vecteurs aient une valeur fixée sur $\approx p$ lignes. On choisira donc $\ell_\downarrow$ à cette valeur-là.

Pour former les deux listes $\mathcal{L}_1$ et $\mathcal{L}_2$, on prépare 4 listes composées de combinaisons linéaires de $p/4$ colonnes prises dans l'une des deux moitiés. Pour cela, on tire aléatoirement quatre chaînes de bits satisfaisant $a + b = u_\downarrow$ et $c + d = v_\downarrow$.

$$\begin{aligned}
\mathcal{L}_{11} &= \{(y_\downarrow + a \mapsto e_1) \mid y = M_\downarrow^{\leftarrow} e_1, |e_1| = p/4\},\\
\mathcal{L}_{12} &= \{(y_\downarrow + b \mapsto e_2) \mid y = M_\downarrow^{\rightarrow} e_2, |e_2| = p/4\},\\
\mathcal{L}_{21} &= \{(y_\downarrow + c \mapsto e_1) \mid y = M_\downarrow^{\leftarrow} e_1, |e_1| = p/4\},\\
\mathcal{L}_{22} &= \{(y_\downarrow + d \mapsto e_2) \mid y = M_\downarrow^{\rightarrow} e_2, |e_2| = p/4\}.
\end{aligned}$$

On obtient ensuite $\mathcal{L}_1$ (resp. $\mathcal{L}_2$) en effectuant la jointure de $\mathcal{L}_{11}$ et $\mathcal{L}_{12}$ (resp. $\mathcal{L}_{21}$ et $\mathcal{L}_{22}$), comme illustré par la figure 4.10.

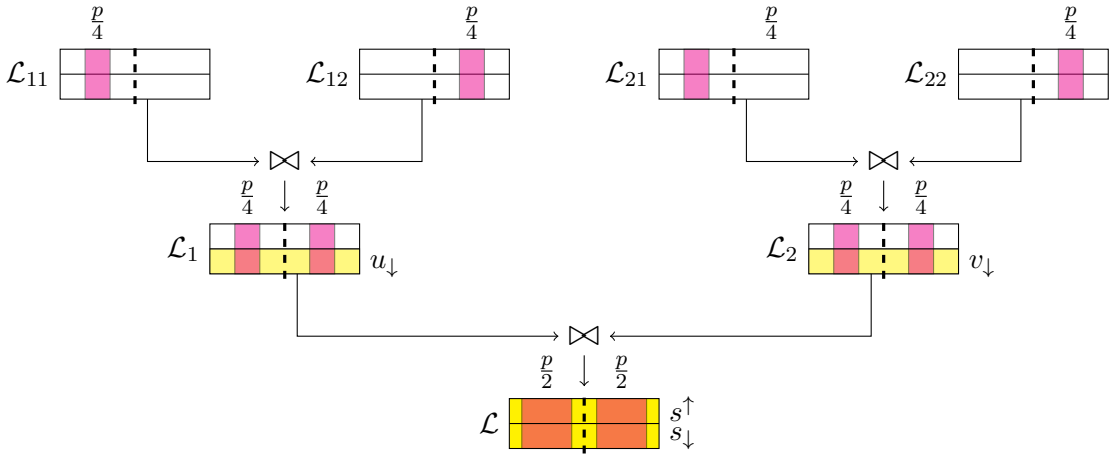


FIGURE 4.10 – Illustration de l'algorithme MMT.

Les quatre listes du premier niveau sont clairement de taille $\binom{(k+\ell)/2}{p/4} \approx \binom{k+\ell}{p/2}^{1/2}$. Les listes intermédiaires sont de taille $\approx 2^{-\ell_\downarrow}\binom{k+\ell}{p/2}$. Quant à la liste finale, qui contient des solutions de la sous-instance, elle est de taille $2^{-\ell-\ell_\downarrow}\binom{k+\ell}{p/2}^2$. Ceci n'est pas facile à comparer directement avec le nombre de solutions attendu, mais May, Meurer et Thomae prouvent, avec des arguments sur le nombre de représentations, que si $\ell_\downarrow \leq p - 2$, alors n'importe quelle solution particulière est renvoyée avec probabilité au moins $1/2$. On peut donc dire qu'en deux itérations, l'algorithme renvoie toutes les solutions. Par contre, comme les valeurs de p sont relativement faibles, les valeurs de $\ell_\downarrow$ sont très contraintes.

Le nombre d'opérations total de la procédure est donc :

$$T := \binom{k+\ell}{p/2}^{1/2} + 2^{-\ell_\downarrow}\binom{k+\ell}{p/2} + 2^{-\ell-\ell_\downarrow}\binom{k+\ell}{p/2}^2.$$

La complexité spatiale est presque la même (il n'est pas nécessaire de stocker $\mathcal{L}$).

Une implantation efficace de l'algorithme MMT a été écrite par Esser, May et Zweydinger [99] puis utilisée pour casser McEliece avec $n = 1284$ en un mois.

Complexité de l'ISD basé sur la procédure de May-Meurer-Thomae

Le nombre d'itérations est toujours le même que dans la section 4.5.2. Il suffit de réutiliser le long raisonnement qu'on a déjà fait pour l'algorithme de Leon/Stern/Dumer, mais avec des contraintes non-linéaires différentes qui définissent la complexité en temps et en espace d'une itération. On introduit une

variable supplémentaire L qui donne la taille de $\mathcal{L}_{ij}$. On a alors (en plus des inégalités qui définissent (χ) :

$$\begin{aligned}(R+\overline{\ell})H\left(\frac{\overline{p}}{2(R+\overline{\ell})}\right) &\leq 2L\\ L &\leq M\\ 2L-\overline{\ell_\downarrow} &\leq M\\ M &\leq T\\ 4L-\overline{\ell}-\overline{\ell_\downarrow} &\leq T\end{aligned}$$

Modifier le programme précédent pour optimiser ce problème est alors relativement simple. La partie importante est :

```
def optimize_cost(R, w, kappa):
    """Trouve (p, l) qui optimisent le coût, à (R,w) fixés"""
    def constraints(args):
        p, l, ldown, L, T, M = args
        return L - 0.5 * Hp(R + l, p / 2)
    nlc = NonlinearConstraint(constraints, lb=0, ub=inf, keep_feasible=True)

    # Contraintes linéaires
    A = [[-1, 1, 0, 0, 0, 0],    #  w-p    <= 1 - R - l
         [ 1,-1, 0, 0, 0, 0],    #  p      <= R+l
         [ 0,-1, 1, 0, 0, 0],    #  ldown <= l
         [-1, 0, 1, 0, 0, 0],    #  ldown <= p
         [ 0, 0, 0, 1, 0,-1],    #  L <= M
         [ 0, 0,-1, 2, 0,-1],    #  2L - ldown <= M
         [ 0, 0, 0, 0,-1, 1],    #  M <= T
         [ 0,-1,-1, 4,-1, 0]]    #  4L-l-down <= T
    lc = LinearConstraint(A, lb=-inf, ub=[1 - R - w, R, 0, 0, 0, 0, 0, 0], keep_feasible=True)
    bounds = Bounds(lb=0, ub=[w, inf, inf, inf, inf, inf], keep_feasible=True)

    l0 = (1 - R - w) / 2
    p0 = min(w, R) / 2
    x0 = (p0, l0, min(l0, p0)/2, 1, 4, 2)      # intérieur de la zone de faisabilité
    return minimize(cost, args=(R, w, kappa), x0=x0, bounds=bounds,
                    constraints=[nlc,lc], method='trust-constr')
```

Ce n'est pas un théorème, mais il semble que la bonne solution soit de choisir la plus grande valeur possible de $\ell_\downarrow$. Ceci a comme effet d'équilibrer les tailles des listes intermédiaires $\mathcal{L}_1$ et $\mathcal{L}_2$ avec celle de la liste finale $\mathcal{L}$. Les listes de départ sont plus petites. Les paramètres optimaux semblent donc satisfaire :

$$2^\ell = \binom{k+\ell}{p/2}.$$

On peut alors de nouveau se ramener à un problème en une seule variable, en posant $\overline{\ell_\downarrow} = \overline{p}$, $\overline{p} = 2(R+\overline{\ell})H^{-1}\left(\frac{\overline{\ell}}{R+\overline{\ell}}\right)$ et $T = \overline{\ell} - \overline{\ell_\downarrow}$. La durée d'une itération est alors simplement $2^{\ell^\uparrow}$.

Pour le *half-distance decoding*, le pire cas est atteint pour un débit $R = 0.4636$, avec $\overline{p} = 0.006390$, $\overline{\ell} = 0.02787$. Par exemple, pour le défi du décodage d'un code de débit 0.5 et de longueur 1280, ceci suggère $p = 8$.

Pour la colonne « McEliece » du tableau 2.4 ($R = 0.75, \overline{w} = 0.02$), on obtient les résultats suivants selon que l'on cherche à minimiser le nombre d'opérations ou bien le coût.

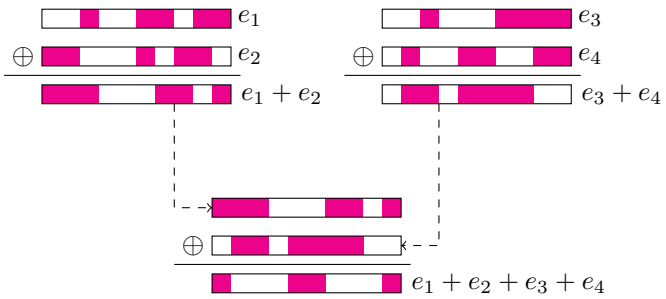


FIGURE 4.11 – Décomposition d'une solution possible dans l'algorithme BJMM.

Minimisation	Nombre d'opérations	Coût
$\overline{p}$	0.00458	0.000431
$\overline{\ell}$	0.0226	0.00285
$\overline{\ell_\downarrow}$	0.00456	0.000425
# Itérations	0.0196	0.0375
# Opérations	0.0376	0.0399
Mémoire	0.0179	0.00242
Coût	0.0436	0.0407

Ce tableau appelle plusieurs commentaires. Comparé à l'algorithme de Léon/Stern/Dumer, on choisit p presque deux fois plus grand et on fait moins d'itérations. La stratégie de minimisation du nombre d'opérations utilise une valeur de p plus élevée : elle consiste à effectuer moins d'itérations un peu plus lourdes. Au contraire, la stratégie de minimisation du coût consiste choisir p petit, puis à faire beaucoup d'itérations très légères. En effet, pour la minimisation du coût, on a $\overline{p} \approx \overline{\ell_\downarrow} \approx 1/2350$. Donc, concrètement, ceci suggère $p = 1$ tant que n n'est pas très grand. Ceci n'est pas à prendre au pied de la lettre, car c'est une estimation assez imprécise, mais le message « petit p » est clair.

Concrètement, $p = 4$ est toujours possible sur des machines classiques, $p = 8$ aboutit à des listes de quelques millions d'entrées et pourrait donc être utilisable en pratique, alors que $p = 12$ est irréaliste. Du coup, il serait intéressant de comparer les choix $p = 4$ et $p = 8$ avec leurs équivalents dans l'algorithme de Stern.

4.5.5 Algorithme de Becker-Joux-May-Meurer [17]

L'algorithme BJMM améliore l'algorithme MMT en autorisant encore plus de liberté dans la construction des deux listes $\mathcal{L}_1$ et $\mathcal{L}_2$. Précédemment, des éléments des deux listes *pouvaient* avoir des bits aux mêmes positions : ils s'éliminent lors de la jointure et le poids total baisse en dessous de la borne.

L'algorithme BJMM rend cet évènement obligatoire : les deux listes seront constituées de vecteurs de poids $\frac{p}{2} + \epsilon$, et seules les paires de vecteurs qui se chevaucheront sur ϵ coordonnées donneront lieu à une somme de poids exactement p.

En fait, cette technique est appliquée deux fois récursivement. La solution recherchée e satisfaisant $Me = s$ et $|e| = p$ est, comme dans MMT, décomposée en quatre parts $e = (e_1 + e_2) + (e_3 + e_4)$ avec les contraintes suivantes :

— $(e_1 + e_2)$ et $(e_3 + e_4)$ sont de poids $p_1 = \frac{p}{2} + \epsilon_1$.
— Tous les e_i sont de poids $p_2 = \frac{p_1}{2} + \epsilon_2 = \frac{p}{4} + \epsilon'$.

Tout ceci est résumé par la figure 4.11. Par conséquent, avec $\epsilon_1 = \epsilon_2 = 0$, on retombe sur l'algorithme MMT. Au passage, pour que ça fonctionne, il faut que $p + \epsilon_1 \leq k + \ell$ et $p + \epsilon_1/2 + \epsilon_2 \leq k + \ell$.

On introduit de nouveau une décomposition aléatoire du membre droit $s = u + v$, et on pose comme

dans MMT :

$$\mathcal{L}_1 = \{(y + u \mapsto e) \mid y = Me, |e| = p_1, y_\downarrow = u_\downarrow\},$$
$$\mathcal{L}_2 = \{(y + v \mapsto e) \mid y = Me, |e| = p_1, y_\downarrow = v_\downarrow\}.$$

Il y a

$$R_1 = \binom{p}{p/2}\binom{k+\ell-p}{\epsilon_1}$$

manières d'écrire un vecteur e de poids p comme somme de deux vecteurs e_1 et e_2 de poids $p_1 = p/2+\epsilon_1$. Par conséquent, on peut imposer une contrainte sur $\approx \log R_1$ coordonnées de Me_1 et Me_2 et espérer préserver une « représentation » de e. Par conséquent, on fixe que la taille de la tranche du bas soit

$$\ell_\downarrow \approx \log_2 R_1 \approx p + (R + \overline{\ell} - \overline{p})H\left(\frac{\epsilon_1}{R + \overline{\ell} - \overline{p}}\right).$$

On étend l'opérateur de jointure avec une contrainte sur le poids de la sortie :

$$\mathcal{L}_1 \bowtie_p \mathcal{L}_2 = \{(e_1 + e_2) \ : \ (x \mapsto e_2) \in \mathcal{L}_1, (y \mapsto e_2) \in \mathcal{L}_2, x = y, |e_1 + e_2| = p\}$$

Calculer cette jointure se fait en évaluant la jointure « classique » puis en filtrant les sorties qui n'ont pas le bon poids. Ceci implique que la complexité du calcul de cette jointure n'est plus linéaire en la taille de l'entrée plus celle de la sortie (elle peut être plus grande). La jointure des deux listes $\mathcal{L}_1$ $\mathcal{L}_2$, avec cette contrainte sur le poids, donne toutes les solutions du problème :

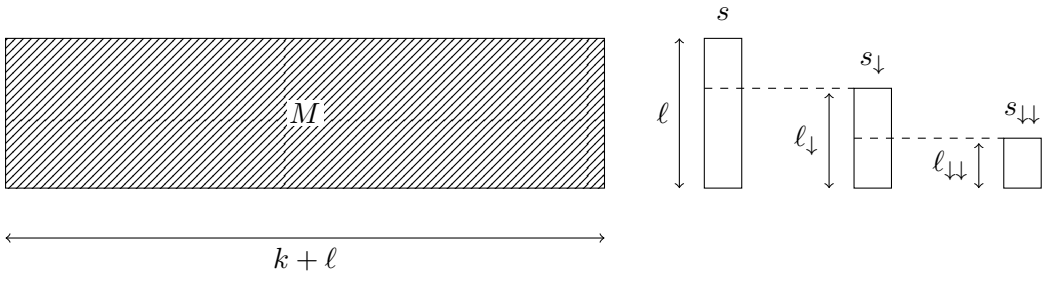
$$\mathcal{L} = \mathcal{L}_1 \bowtie_p \mathcal{L}_2.$$

Pour construire les deux listes $\mathcal{L}_1$ et $\mathcal{L}_2$, on utilise la même technique récursivement. Chaque vecteur de poids p_1 va s'écrire comme la somme de deux vecteurs de poids p_2. Comme il y a encore un certain nombre R_2 de représentations, avec

$$R_2 = \binom{p_1}{p_1/2}\binom{k+\ell-p_1}{\epsilon_2},$$

on peut imposer des contraintes sur $\ell_{\downarrow\downarrow}$ lignes, avec

$$\ell_{\downarrow\downarrow} \approx \log_2 R_2 \approx p_1 + (R + \overline{\ell} - \overline{p_1})H\left(\frac{\epsilon_2}{R + \overline{\ell} - \overline{p_1}}\right).$$



On prépare de nouveaux 4 listes composées de combinaisons linéaires de $p_2 = p_1/2 + \epsilon_2 = p/4 + \epsilon'$ colonnes prises dans l'une des deux moitiés. Pour cela, on tire aléatoirement quatre chaînes de bits satisfaisant $a + b = u$ et $c + d = v$ et on pose :

$$\mathcal{L}_{11} = \{(y_\downarrow + a_\downarrow \mapsto e) \mid y = Me, |e| = p_2, y_{\downarrow\downarrow} = a_{\downarrow\downarrow}\},$$
$$\mathcal{L}_{12} = \{(y_\downarrow + b_\downarrow \mapsto e) \mid y = Me, |e| = p_2, y_{\downarrow\downarrow} = b_{\downarrow\downarrow}\},$$
$$\mathcal{L}_{21} = \{(y_\downarrow + c_\downarrow \mapsto e) \mid y = Me, |e| = p_2, y_{\downarrow\downarrow} = c_{\downarrow\downarrow}\},$$
$$\mathcal{L}_{22} = \{(y_\downarrow + d_\downarrow \mapsto e) \mid y = Me, |e| = p_2, y_{\downarrow\downarrow} = d_{\downarrow\downarrow}\}.$$

Puis on obtient $\mathcal{L}_1 = \mathcal{L}_{11} \bowtie_{p_1} \mathcal{L}_{12}$ et $\mathcal{L}_2 = \mathcal{L}_{21} \bowtie_{p_1} \mathcal{L}_{22}$.

Il ne reste plus qu'à construire les quatre listes $\mathcal{L}_{11}, \ldots, \mathcal{L}_{22}$. Pour cela, on utilise un simple « *meet-in-the-middle* » : pour chacune d'entre elles, on partitionne aléatoirement les $k+\ell$ colonnes en deux sous-ensembles de taille $(k+\ell)/2$, on produit les deux listes de vecteurs e de poids $p_2/2$, dont les coordonnées non-nulles sont toutes dans le premier (resp. le second) sous-ensemble, et on effectue une jointure pour forcer la somme à avoir la bonne valeur sur les $\ell_{\downarrow\downarrow}$ premières lignes. Ceci induit une certaine probabilité de perdre des solutions, pour chacune des 4 listes.

La taille des listes de départ est $S_3 = \binom{(k+\ell)/2}{p_2/2} \approx \binom{k+\ell}{p_2}^{1/2}$. La taille de leur jointure (donc la taille de $\mathcal{L}_{ij}$), en moyenne, est $S_2 = \binom{k+\ell}{p_2} 2^{-\ell_{\downarrow\downarrow}}$. Enfin, la taille de $\mathcal{L}_i$ est $S_1 = \binom{k+\ell}{p_1} 2^{-\ell_{\downarrow}}$ — et en principe on doit avoir $S_1 \leq S_2^2 2^{\ell_{\downarrow\downarrow} - \ell_{\downarrow}}$ car la jointure se fait avec contrainte sur le poids de la sortie. La complexité spatiale est donc $\max(S_1, S_2, S_3)$.

Le calcul des jointures aux trois niveaux nécessitent des temps respectifs $T_3 = \max(S_3, S_3^2 2^{-\ell_{\downarrow\downarrow}}), T_2 = \max(S_2, S_2^2 2^{\ell_{\downarrow\downarrow} - \ell_{\downarrow}})$ et $T_1 = \max(S_1, S_1^2 2^{\ell_{\downarrow} - \ell})$. La complexité en temps est finalement $T = T_1 + T_2 + T_3$.

Il faut noter que le choix de se limiter à deux étages dans la construction des listes est partiellement arbitraire (en fait, en ajouter un troisième ne changerait pas grand-chose).

À ce que je sache, l'algorithme BJMM n'a jamais été utilisé (toutes les implantations ont $\epsilon_1 = \epsilon_2 = 0$).

Complexité de l'ISD basé sur la procédure de Becker-Joux-May-Meurer

Le nombre d'itérations est le même que dans les algorithmes de MMT et de Léon/Stern/Dumer, et on réutilise la même démarche, qui montre ici toute sa souplesse. Les paramètres de l'algorithme sont p, ℓ, ϵ_1 et ϵ_2. On définit les fonctions suivantes des paramètres :

$$
\begin{aligned}
\overline{p_1} &= 0.5\overline{p} + \overline{\epsilon_1} \\
\overline{p_2} &= 0.5\overline{p_1} + \overline{\epsilon_2} \\
\overline{\ell_{\downarrow}} &= \overline{p} + (R + \overline{\ell} - \overline{p}) H(\overline{\epsilon_1}/(R + \overline{\ell} - \overline{p})) \\
\overline{\ell_{\downarrow\downarrow}} &= \overline{p_1} + (R + \overline{\ell} - \overline{p_1}) H(\overline{\epsilon_2}/(R + \overline{\ell} - \overline{p_1})) \\
S_3 &= \frac{R + \overline{\ell}}{2} H\left(\frac{\overline{p_2}}{R + \overline{\ell}}\right) \\
S_2 &= 2S_3 - \overline{\ell_{\downarrow\downarrow}} \\
S_1 &= (R + \overline{\ell}) H\left(\frac{\overline{p_1}}{R + \overline{\ell}}\right) - \overline{\ell_{\downarrow}}
\end{aligned}
$$

Et on a les contraintes :

$$
\begin{aligned}
\overline{p} + \overline{\epsilon_1} &\leq R + \overline{\ell} \\
\overline{p_1} + \overline{\epsilon_2} &\leq R + \overline{\ell} \\
0 \leq \overline{\ell_{\downarrow\downarrow}} &\leq \overline{\ell_{\downarrow}} \leq \overline{\ell} \\
S_3 &\leq M \\
S_2 &\leq M \\
S_1 &\leq M \\
M &\leq T \\
2S_2 + \overline{\ell_{\downarrow\downarrow}} - \overline{\ell_{\downarrow}} &\leq T \\
2S_1 + \overline{\ell_{\downarrow}} - \overline{\ell} &\leq T
\end{aligned}
$$

Numériquement, ce problème d'optimisation est plus dur que les autres. Il est plus facile d'écrire des programmes qui ne fonctionnent pas. Certaines formulations du problèmes permettent la résolution

numérique, et d'autres non. Par exemple, mettre $\overline{\ell_\downarrow}$ et $\overline{\ell_{\downarrow\downarrow}}$ dans les variables à optimiser conduit à l'échec.

```
def optimize_cost(R, w, kappa):
    A = np.array([[-1,  1, 0, 0, 0, 0],    # l <= 1 - R - w + p
                  [ 1, -1, 0, 0, 0, 0],    # p <= R + l
                  [ 1, -1, 1, 0, 0, 0],    # p + e1 <= R + l
                  [0.5,-1, 1, 1, 0, 0],    # p1 + e2 <= R + l
                  [ 0,  0, 0, 0,-1, 1],    # M - T <= 0
                 ])
    lc = LinearConstraint(A, -inf, [1 - R - w, R, R, R, 0])
    def nlf(x):
        p, l, epsilon1, epsilon2, T, M = x
        p1 = p/2 + epsilon1
        p2 = p1/2 + epsilon2
        ldown = p + Hp(R + l - p, epsilon1)
        ldowndown = p1 + Hp(R + l - p1, epsilon2)
        S3 = 0.5 * Hp(R + l, p2)
        S2 = 2 * S3 - ldowndown
        S1 = Hp(R + l, p1) - ldown
        c1 = ldowndown
        c2 = ldown - ldowndown
        c3 = l - ldown
        c4 = M - S3
        c5 = M - S2
        c7 = M - S1
        c6 = T - (2*S2 + ldowndown - ldown)
        c8 = T - (2*S1 + ldown - l)
        return [c1, c2, c3, c4, c5, c6, c7, c8]
    nlc = NonlinearConstraint(nlf, 0, inf)
    x0 = [0.01, 0.01, 0.001, 0.0001, 1, 1]
    bounds = Bounds(lb=0, ub=np.array([w, inf, inf, inf, inf, inf]))
    return minimize(cost, args=(R, w, kappa),
                    x0=x0, bounds=bounds, constraints=[lc, nlc], method='trust-constr')
```

On constate que dans les solutions renvoyées, le temps nécessaire à la réalisation d'une itération est atteint par le calcul des deux dernièrs étages de jointures, c'est-à-dire la production de $\mathcal{L}_1$ et $\mathcal{L}_2$ puis le calcul de leur jointure. Le calcul de $\mathcal{L}_i$ est "équilibré", c'est-à-dire que la taille de l'entrée et le nombre de vecteurs à examiner pour produire la sortie sont égaux ($T = S_2 = 2\overline{S_2} + \overline{\ell_{\downarrow\downarrow}} - \overline{\ell_\downarrow}$). Le nombre d'éléments dans la sortie ne domine pas la complexité ($S_1 < T$), par contre le temps nécessaire au calcul de la jointure $\mathcal{L}_1 \bowtie_p \mathcal{L}_2$ est lui aussi égal à T.

Dans le cas du *half-distance decoding*, le pire cas est atteint pour $R = 0.45897$, et la complexité est en $2^{0.0493n}$ avec $\overline{p} = 0.01408$, $\overline{\ell} = 0.07374$, $\overline{\epsilon_1} = 0.003492$ et $\overline{\epsilon_2} = 0.0001661$. Un premier commentaire s'impose : $\overline{\epsilon_2} \approx 1/6000$, donc tant que n n'est pas absurdement grand, il n'y aura pas de « chevauchement » au deuxième étage de la construction. Ceci suggère d'envisager en pratique une version « allégée » de l'algorithme BJMM avec $\epsilon_2 = 0$ d'entrée de jeu. Pour le défi du code avec $R = 0.5$ et $n = 1280$, ceci suggère $p = 16$, $\ell = 96$, $\epsilon_1 = 4$, $\epsilon_2 = 0$ ($p = 4$ et $\ell = 30$ pour minimiser le coût).

Les paramètres optimaux (pour le coût et pour le temps, pour le *half-distance decoding* ou pour McEliece) équilibrent la durée de construction des listes aux trois étages. Les listes de base sont plus courtes que les listes $\mathcal{L}_{ij}$ et $\mathcal{L}_i$, qui sont de la même taille. Chaque itération fonctionne en temps linéaire en l'espace occupé.

Pour la colonne « McEliece » du tableau 2.4 ($R = 0.75, \overline{w} = 0.02$), on obtient les résultats suivants selon que l'on cherche à minimiser le nombre d'opérations ou bien le coût.

Minimisation	Nombre d'opérations	Coût
$\overline{p}$	0.008622	0.002671
$\overline{\ell}$	0.05531	0.02117
$\overline{\epsilon_1}$	0.002307	0.0007932
$\overline{\epsilon_2}$	0.00005086	0.000007823
# Itérations	0.01009	0.02721
# Opérations	0.03409	0.03670
Mémoire	0.02400	0.009481
Coût	0.04209	0.03985

Comparé à l'algorithme MMT, on choisit des valeurs de p deux fois plus grandes et on fait moins d'itérations. Par contre, les valeurs optimales de ϵ_2 dans les deux cas sont ridiculement faibles (1/20000 voire moins), ce qui suggère que dans des instances concrètes ce paramètre sera fixé à une valeur très proche de zéro. Encore une fois, minimiser le coût revient à faire plus d'itérations légères.

On peut enfin noter qu'une itération ne produit la bonne solution qu'avec une probabilité (polynomialement) faible, qu'on n'a pas cherché à quantifier. De faire, le caractère pratique de l'algorithme n'est pas vraiment établi, et ce serait probablement intéressant à trancher sur de vraies machines.

4.5.6 Algorithme de May-Ozerov [157]

L'idée principale de May et Ozerov [157] consiste à observer que le problème du décodage peut se ramener au « problème des plus proches voisins » (*Nearest Neighbor* ou *Bichromatic Closest Pair*). Celui-ci est par ailleurs similaire au *Light Bulb Problem*. Les deux ont connu des progrès récents [4, 98].

On a déjà dit que l'*Information Set Decoding* consiste dans le fond à trouver une combinaison linéaire de p colonnes de Q qui coïncide *approximativement* avec $\tilde{s}$: il faut que $|Q\tilde{e}_q + \tilde{s}| \leq w - p$ (cf. figure 4.7). Ceci peut se faire de la façon suivante, inspirée de l'algorithme de Leon/Stern/Dumer : énumérer tous vecteurs e_1, e_2 de poids $p/2$ à coordonnées dans la première (resp. la deuxième) moitié des colonnes ; construire deux listes $\mathcal{L}_1$ et $\mathcal{L}_2$ de produits Qe_1 et $Qe_2 + \tilde{s}$; chercher les paires $(u, v) \in \mathcal{L}_1 \times \mathcal{L}_2$ telles que $|u + v| \leq w - p$. Ceci est une instance du problème des plus proches voisins, et peut se résoudre naïvement par un algorithme quadratique en $|\mathcal{L}_1| \times |\mathcal{L}_2|$. En plus, il faut noter que cette instance du problème est « spéciale » dans le sens où on s'attend à ce que la sortie soit vide (en effet, il suffit de trouver *un* élément dans la sortie, et c'est gagné). La combinaison algorithmique résultante n'améliore même pas l'algorithme de Prange.

Dans notre contexte, le problème des plus proches voisins se formule de la façon suivante. Étant donné deux listes A et B de taille $2^{\lambda m}$ composées de chaînes de m bits, le problème (m, γ, λ)-NN consiste à produire les paires (u, v) de $A \times B$ avec $|u + v| \leq \gamma m$.

Si on remplace les deux listes par des fonctions, alors le problème revient à trouver une collision partielle (« *near collision* ») entre deux fonctions. Un certain nombre de techniques ont été développées pour ce problème : collisions sur une version tronquée de la fonction, utilisation de codes de recouvrement [139], combinaisons de ces deux approches, utilisation de *locality sensitive hashing*, etc.

Des techniques inspirées de l'ISD pourraient s'appliquer au problème des plus proches voisins. Par exemple, on peut « deviner » un ensemble de positions où u et v coïncident, puis noter que (u, v) appartient à la jointure entre A et B sur les positions en question. On peut donc calculer cette jointure et tester si la distance de Hamming des paires résultantes est assez faible. C'est relativement équivalent à l'algorithme de Stern.

Un ensemble de αm colonnes aléatoire est correct (c'est-à-dire que u et v coïncident dessus) avec probabilité $\binom{(1-\gamma)m}{\alpha m} / \binom{m}{\alpha m}$. La taille moyenne de la sortie de la jointure est $2^{(2\lambda - \alpha)m}$. Il n'est donc pas très utile de choisir $\alpha > 2\lambda$, car ça surcontraint le problème inutilement. La complexité de cette procédure est $2^{\mu m}$ avec $\mu = H(\alpha) - (1 - \gamma) H\left(\frac{\alpha}{1-\gamma}\right) + \max(\lambda, 2\lambda - \alpha)$.

On verra ci-dessous que les cas qui nous intéressent sont ceux où $2\lambda - \gamma \geq \lambda$, donc on peut éliminer

le max et se ramener à

$$\mu = H(\alpha) - (1-\gamma)H\left(\frac{\alpha}{1-\gamma}\right) + 2\lambda - \alpha$$

Si on fait le choix « simple » $\alpha = \lambda$ (sortie des jointures de taille équivalente à celle des listes), alors l'exposant dans la complexité devient $\mu = H(\lambda) - (1-\gamma)H\left(\frac{\lambda}{1-\gamma}\right) + \lambda$.

Par exemple, avec $\gamma = 0.0517$ et $\lambda = 0.1123$ (le choix de ces valeurs précises est discuté ci-dessous), alors l'algorithme s'exécute en $2^{0.1214m}$, donc en temps $N^{1.082}$ par rapport à la taille des listes en entrée, ce qui est bien sous-quadratique.

Il faut noter que poser $\alpha = 2\lambda$ garantit que la taille moyenne de la jointure est de 1. On pourrait donc se contenter de calculer *un* élément dans la jointure et tester si c'est le bon (si le choix des colonnes est correct, alors ce sera le bon avec probabilité non-négligeable). Dans le cas où on a affaire à deux fonctions plutôt qu'à deux listes, ceci permet d'utiliser un algorithme de recherche de collision sans mémoire. Leurent s'en est servi pour trouver une 10-quasi collision sur MD5 [146].

May et Ozerov sont partis de cette même idée et l'ont améliorée pour obtenir un algorithme plus efficace asymptotiquement. Pour n'importe quel $\epsilon > 0$ et $\lambda < 1 - H(\gamma/2)$, leur algorithme résout le problème en temps $\tilde{\mathcal{O}}\left(2^{(e+\epsilon)m}\right)$ avec

$$e = (1-\gamma)\left(1 - H\left(\frac{H^{-1}(1-\lambda) - \gamma/2}{1-\gamma}\right)\right).$$

La complexité est sous-quadratique, mais elle cache un facteur polynomial important (en $n^{\mathcal{O}(\log 1/\epsilon)}$) et May-Ozerov posent comme un problème ouvert de le réduire. Dans la suite, admettons pour simplifier que l'algorithme fonctionne avec $\epsilon = 0$. La complexité en espace est dominée par la taille des listes en entrée, c'est-à-dire $2^{\lambda m}$.

En utilisant cet algorithme, May et Ozerov prouvent qu'on peut faire marcher la variante de l'algorithme de Léon/Stern/Dumer décrite ci-dessus en temps $\mathcal{O}\left(2^{0.055n}\right)$ avec $\overline{p} = 0.003839$ et $\overline{\ell_\downarrow} = 0$ dans le cas du *half-distance decoding*, ce qui améliore un tout petit peu l'original. Mais en fait, la même stratégie fonctionne en $\mathcal{O}\left(2^{0.053n}\right)$ opérations avec $\overline{p} = 0.00577662$ et $\overline{\ell_\downarrow} = 0.002357$, donc ça améliore même l'algorithme MMT. C'est dire si la technique est efficace.

Et ce n'est pas tout, car elle s'applique aussi à l'algorithme BJMM. Dans ce cas, on construit deux listes $\mathcal{L}_1$ et $\mathcal{L}_2$ composées de vecteurs de poids $p/2 + \epsilon_1$ et telles que $M_\downarrow(u+v) = \tilde{s}_\downarrow$. Au lieu de faire une jointure entre $\mathcal{L}_1$ et $\mathcal{L}_2$ pour réduire le nombre de vecteurs à tester sur le reste du syndrome, on cherche les plus proches voisins entre $\mathcal{L}_1$ et $\mathcal{L}_2$. Le reste de l'algorithme est identique, et on peut l'analyser avec $\ell = \ell_\downarrow$.

Voici une technique simplifiée, inspirée de l'algorithme de May-Ozerov pour résoudre le problème des plus proches voisins. Tirer un entier r selon une loi de Poisson de paramètre $2(1-\gamma)n$ puis répéter r fois la procédure suivante : tirer au hasard un entier $k \in \{1, \dots, n\}$, tirer un bit b (on fait l'hypothèse que $u[k] = b$ et $v[k] = b$), former les sous-listes $A' = \{x \in A \ : \ x[k] = b\}$ et $B' = \{y \in B \ : \ y[k] = b\}$, répéter la procédure récursivement sur A', B' (on peut stopper la récursion lorsque les deux listes sont suffisamment petites). La distribution du nombre de *bons* choix effectués à chaque étage de la récursion est la loi de Poisson de paramètre 1 (indépendamment de γ et n). L'arbre formé par les bons choix est alors un arbre de Galton-Watson critique, et il est facile à étudier. La probabilité qu'un « chemin de bons choix » atteigne la profondeur h est au moins $1/h$.

Il faut noter qu'en pratique, il n'est pas du tout évident que l'algorithme de May-Ozerov améliore sensiblement des techniques assez naïves, notamment celles qui sont décrites ci-dessus.

Complexité de l'ISD basé sur la procédure de Becker-Joux-May-Meurer avec amélioration de May-Ozerov

Là encore, on peut réutiliser la même démarche pour la résolution numérique du problème d'optimisation, sauf que c'est encore plus dur. Notre algorithme d'optimisation préféré ne s'avère plus utilisable

dans ce cas-là (diverge et/ou évalue les fonctions sur des points où elles ne sont pas définies). On en revient donc à la méthode de l'optimisation quadratique successive.

```
def optimize_cost(R, w, kappa):
    bounds = Bounds(lb=0, ub=np.array([w, inf, inf, inf, inf, inf]))
    A = np.array([[-1,    1, 0, 0, 0, 0],    #  ldown <= 1 - R - w + p
                  [ 1,   -1, 0, 0, 0, 0],    #  p <= R - ldown
                  [ 1,   -1, 1, 0, 0, 0],    #  p + e1 <= R - ldown
                  [ 0.5, -1, 1, 1, 0, 0],    #  p1 + e2 <= R - ldown
                  [ 0,    0, 0, 0,-1, 1]])   #  M <= T
    lc = LinearConstraint(A, -inf, [1 - R - w, R, R, R, 0])

    def nlf_ineq(x):
        p, ldown, epsilon1, epsilon2, T, M = x
        p1 = p/2 + epsilon1
        p2 = p1/2 + epsilon2
        ldowndown = p1 + Hp(R + ldown - p1, epsilon2)
        S3 = 0.5 * Hp(R + ldown, p2)
        S2 = 2 * S3 - ldowndown
        S1 = Hp(R + ldown, p1) - ldown
        c1 = ldowndown
        c2 = ldown - ldowndown
        c4 = M - S3
        c5 = M - S2
        c7 = M - S1
        c6 = T - (2*S2 + ldowndown - ldown)
        m = (1 - R - ldown)
        lamb = S1 / m
        gamma = (w - p) / m
        c8 = T - m * nn(gamma, lamb)     # active
        return c1, c2, c4, c5, c6, c7, c8

    def nlf_eq(x):
        p, ldown, epsilon1, epsilon2, T, M = x
        p1 = p/2 + epsilon1
        p2 = p1/2 + epsilon2
        return ldown - (p + Hp(R + ldown - p, epsilon1))

    nlc_ineq = NonlinearConstraint(nlf_ineq, 0, inf)
    nlc_eq = NonlinearConstraint(nlf_eq, 0, 0)
    x0 = [0.01, 0.01, 0.001, 0.0001, 0.01, 0.01]
    return minimize(cost, args=(R, w, kappa), x0=x0, bounds=bounds,
                    constraints=[lc, nlc_ineq, nlc_eq], method='SLSQP')
```

Dans le cas du *half-distance decoding*, le pire cas est atteint pour $R = 0.463$, et la complexité est en $2^{0.0472n}$ avec $\overline{p} = 0.01666$, $\overline{\ell} = 0.06343$, $\overline{\epsilon_1} = 0.005953$ et $\overline{\epsilon_2} = 0.001469$. Tous les paramètre sauf $\overline{\ell}$ sont plus grands que pour l'algorithme BJMM « pur ». En particulier, ϵ_2 est raisonnablement grand (sauf si on veut minimiser le coût, avec $\overline{\epsilon_2} \approx 1/2000$). Cette fois-ci, les listes ont des tailles toutes différentes aux trois étages (c'est $\mathcal{L}_{ij}$ les plus grosses dans tous les cas considérés). Le problème des plus proches voisins se présente avec $\gamma = 0.09429$ et $\lambda = 0.066$. L'algorithme de May-Ozerov pour les plus proches voisins fonctionne en $\mathcal{O}\left(N^{1.108}\right)$ en la taille de $\mathcal{L}_i$, ce qui est fortement sous-quadratique.

Les paramètres optimaux (pour le coût et pour le temps, pour le *half-distance decoding* ou pour McEliece) équilibrent le temps qu'il faut pour produire $\mathcal{L}_{ij}$, produire $\mathcal{L}_i$ et chercher les plus proches voisins. Ceci est vrai aussi si on cherche à minimiser le coût. Chaque itération fonctionne encore en temps linéaire en l'espace occupé.

Pour la colonne « McEliece » du tableau 2.4 ($R = 0.75, \overline{w} = 0.02$), on obtient les résultats suivants selon que l'on cherche à minimiser le nombre d'opérations ou bien le coût.

Algorithme	Half-distance			McEliece		
	T	M	Coût	T	M	Coût
Lee-Brickell [142]	0.0575	0	0.0575	0.0409	0	0.0409
Stern [179]	0.0556	0.0135	0.0601	0.0391	0.0123	0.0432
Ball-collision [26]	0.0556	0.0148	0.0605	?	?	?
MMT [156]	0.0536	0.0216	0.0608	0.0376	0.0179	0.0436
BJMM [17]	0.0493	0.0331	0.0589	0.0340	0.0240	0.0421
May-Ozerov [157]	0.0473	0.0346	0.0588	0.0331	0.0251	0.0415
MMT	0.0564	0.0029	0.0573	0.0399	0.00242	0.0407
BJMM	0.0525	0.0115	0.0564	0.0367	0.00948	0.0399
May-Ozerov	0.0507	0.0159	0.0560	0.0355	0.00121	0.0396

TABLE 4.2 – Nouvelle version du tableau 2.4 qui montre les coûts. Les entrées du bas sont optimisées pour le coût plutôt que le nombre d'opérations.

Minimisation	Nombre d'opérations	Coût
$\overline{p}$	0.009061,	0.003484
$\overline{\ell_\downarrow}$	0.03825,	0.01793
$\overline{\epsilon_1}$	0.003101,	0.001367
$\overline{\epsilon_2}$	0.0004493,	0.0002012
γ	0.0517	0.0712
λ	0.1123	0.0485
exposant NN	1.0578	1.0786
# Itérations	0.007937	0.02337
# Opérations	0.03307	0.03552
Mémoire	0.02514	0.01214
Coût	0.04145	0.03957

L'algorithme de May-Ozerov pour les plus proches voisins est utilisé dans un régime où il est presque linéaire. Par rapport à l'algorithme BJMM classique, on peut utiliser des valeurs de $\ell_\downarrow$ plus faibles, donc faire moins d'itérations et utiliser un p légèrement plus grand. Là encore, le caractère pratique de l'algorithme est relativement flou et nécessiterait des investigations plus poussées.

4.5.7 Discussion

Ces algorithmes passent l'essentiel de leur temps à effectuer des jointures. Du coup, leur coût est relativement simple à étudier et il est de la forme $TM^{1/3}$. À l'exception des algorithmes de Prange/Lee Brickell, tous consistent à répéter une procédure qui a un « ratio temps-mémoire » égal à un. À partir du tableau 2.4, on peut donc calculer les coûts, et on peut voir dans le tableau 4.2 que... au fur-et-à-mesure des travaux de recherche successifs, le coût des algorithmes a augmenté (!) jusqu'à l'algorithme de Becker-Joux-May-Meurer. La réduction substantielle du nombre d'opérations fait alors baisser le coût... mais jamais au point d'améliorer le vénérable algorithme de Prange.

Mais en fait, cette comparaison est biaisée, car ces algorithmes ont été conçus avec l'objectif de minimiser le nombre d'opérations à l'exclusion de tout autre critère. Optimiser les paramètres pour minimiser le coût montre finalement qu'il n'y a pas de régression au fur et à mesure que la recherche progresse. Chaque nouvel algorithme permet de réduire le coût, et ce ne sont pas juste des compromis temps-mémoire. Ce fait était déjà connu, mais le raisonnement basé sur le coût donne un nouvel argument dans ce sens-là.

Enfin, il faut noter que toutes ces discussions sont assez éthérées, et que l'absence de code exécutable pour les algorithmes réputés les meilleurs, qui n'ont pas été testés sur des *challenges* raisonnables, limite l'intérêt pratique de ces discussions.

Par exemple, le jeu de paramètre original du chiffrement McEliece a été cassé avec l'algorithme de

Stern avant l'invention des algorithmes MMT, BJMM, MO, etc. qui sont asymptotiquement supérieurs. Mais le calcul est-il plus facile maintenant qu'on dispose d'algorithmes « supérieurs » ?

4.6 Résolution de systèmes polynomiaux modulo 2

La résolution de systèmes polynomiaux en plusieurs variables sur un corps fini est un problème NP-complet simple et naturel. Une des versions les plus critiques concerne les systèmes sur $\mathbb{F}_2$, et tout particulièrement ceux qui sont quadratiques.

On considère des systèmes de m polynômes quadratiques $f_1, \ldots, f_m$ en n variables. Ils vivent dans l'anneau $\mathbb{F}_2[x_1, \ldots, x_n]$ quotienté par l'idéal des « équations de corps » $\langle x_1^2 - x_1, \ldots, x_n^2 - x_n \rangle$. On note $x = (x_1, \ldots, x_n)$ le vecteur des variables.

Pour fixer les idées, le schéma de signature GeMSS [70] expose dans sa clef publique un système de $m = 162$ polynômes quadratiques booléens en $n = 192$ variables.

Les systèmes très surdéterminés ($m \geq 0.5n^2$) peuvent être résolus efficacement par linéarisation. Les systèmes très sous-déterminés ($n \geq m^2$) peuvent aussi être résolus en temps polynomial par une technique due à Courtois, Goubin, Meier et Tacier [78]. La méthode a ensuite été étendue au cas des systèmes légèrement sous-determinés, où elle permet de faire disparaître quelques variables, par Thomae et Wolf [185] puis par Furue, Nakamura et Takagi [114].

Bon nombre d'algorithmes font une recherche exhaustive sur une partie des variables, qui sont donc partitionnées deux sous-ensembles. On pose $x = (y, z)$ avec $y = (y_1, \ldots, y_{n_1}), z = (z_1, \ldots, z_{n_2})$ et $n_1 + n_2 = n$. Pour résoudre le système initial $\{f_i(y, z)\}_{i=1}^m$, on peut utiliser la procédure non-déterministe suivante : « deviner » les valeurs de y, puis résoudre le système restant pour trouver z. Autrement dit : énumérer toutes les chaînes de n_1 bits ; pour chaque $\hat{y} \in \{0,1\}^{n_1}$, résoudre le système « spécialisé en $\hat{y}$ » , qu'on note $\{P_i(z) := f_i(\hat{y}, z)\}_{i=1}^m$.

4.6.1 Recherche exhaustive et variantes

L'approche combinatoire la plus directe pour résoudre des systèmes modulo 2 est sans conteste la recherche exhaustive.

Elle peut être implantée de manière particulièrement efficace, comme je l'ai démontré avec Chen, Cheng, Chou Niederhagen, Shamir et Yang [41]. Sa complexité en temps est $\mathcal{O}(2^n)$, mais la constante peut être rendue extraordinairement faible. J'en ai écrit une version capable de tester 37.5 milliards de solutions par seconde sur un cœur d'un Xeon Gold 6130 CPU @ 2.10GHz — lorsqu'on utilise ce code sur tous les cœurs, ils fonctionnent à 2.4GHz. Ceci fait 15.6 candidats testés *par cycle d'horloge* sur un seul cœur. Un serveur avec deux processeurs de ce type teste 1200 milliards de solutions par seconde, et peut donc résoudre $n = 48$ en un peu moins de 4 minutes. Atteindre ce niveau de performance n'est pas facile : la boucle critique écrite en C fait moins de 10 lignes, mais les compilateurs ne la traduisent pas efficacement en code exécutable. Il faut donc y pallier en générant la séquence d'instructions du processeur directement, avec un script python qui génère le code assembleur, effectue l'allocation de registres, etc.

Ce niveau de performance implique que « battre la force brute » *en pratique* est difficile ; y parvenir est une réussite notable.

Dans la famille des algorithmes appartenant au folklore, on trouve aussi la transformée de Mœbius, décrite par exemple dans [125]. Il s'agit d'une sorte de transformée de Fourier adaptée au cas des fonctions booléennes. Une fonction booléenne peut être représentée ou bien par sa table de vérité, ou bien par sa forme normale algébrique, c'est-à-dire par un polynôme multivarié. Ces deux représentations occupent 2^n bits, et la transformée de Mœbius, qui est involutive, permet de passer de l'une de ces deux représentations à l'autre en $n2^n$ opérations sur des bits (en place).

Le coût de la recherche exhaustive est très clairement 2^n. Dans la mesure où la transformée de Mœbius est une sorte de FFT, les résultats classiques cités section 3.3 laissent supposer que son coût est de

l'ordre de $2^{4n/3}$.

En pratique, la complexité spatiale de la transformée de Mœbius la rend inexploitable telle quelle (ratio temps-espace = 1...). Cependant, elle a l'avantage de pouvoir fonctionner avec des polynômes de n'importe quel degré, et d'être facile à programmer. Ceci la rend pratiquement avantageuse, combinée avec une recherche exhaustive : on énumère les n_1 premières variables, ce qui transforme le système de départ en 2^{n_1} sous-systèmes indépendants en 2^{n_2} variables (avec $n = n_1 + n_2$). Si n_1 est choisi suffisamment grand, les sous-systèmes peuvent être suffisamment petits pour être traités avec la transformée de Mœbius. En pratique, on se débrouillera pour que ça tienne dans les caches des processeurs.

Cas des systèmes de degré élevé

Le cas de la recherche exhaustive appliquée à des polynômes de degré élevé mérite une discussion. En particulier, la « méthode polynomiale » pour la conception d'algorithme tend à produire des systèmes en λn variables de degré $(1-\lambda)n$, pour une certaine constante λ. Pour fixer les idées, on peut imaginer des polynômes de degré $d = n/4$, où n désigne le nombre de variables. Un polynôme de degré d possède de l'ordre de $\binom{n}{d}$ coefficients. Comme la taille de la machine nécessaire augmente exponentiellement avec d (et donc n), il faut envisager un moyen de paralléliser le calcul pour éviter une explosion du coût.

À titre d'exemple, on pourrait se poser le problème de résoudre un système de 48 polynômes en 48 variables de degré 16. Les polynômes occupent 22.5 To.

La transformée de Mœbius s'applique quel que soit le degré, nécessite $n2^n$ accès à une mémoire de taille 2^n. Elle est complètement parallélisable, et comme c'est une sorte de FFT, elle coûte $2^{4n/3}$. En réalité, elle nécessite une machine de taille 2^n et réalise n phases de communications parallèles qui nécessitent chacune un temps $2^{n/3}$ (cf. discussion de la section 3.6.7).

La recherche exhaustive « naïve » est complètement parallélisable ; l'évaluation d'un polynôme sur chacun des 2^n candidats nécessite $\binom{n}{d}$ opérations et autant d'accès à une mémoire de la même taille. L'avantage, c'est que quel que soit le nombre de processeurs, s'ils opèrent de façon synchrone, ils peuvent tous lire la même adresse mémoire en même temps (il suffit de *broadcaster* la description du système autant de fois que nécessaire. Ceci diminue le coût des accès à la mémoire comme discuté section 3.7. Autrement dit, on peut imaginer une machine avec une mémoire de taille $\binom{n}{d}$ qui *broadcaste* la description du polynôme. p processeurs y sont reliés, reçoivent les coefficients du polynôme et procèdent à son évaluation au fur-et-à-mesure sur des entrées distinctes. Il faut $2^n/p$ cycles pour compléter le processus ; chaque cycle a une durée de $\binom{n}{d}$. La taille de la machine est $\binom{n}{d} + p$. Le coût total est donc $2^n\binom{n}{d}\left(\binom{n}{d}/p + 1\right)$, ce qui, avec $p = \binom{n}{d}$, donne un total de l'ordre de $2^{n(1+H(d/n))}$.

Il faut noter que, pour ce cas précis, le modèle « NIST » donnerait un coût strictement supérieur de $2^{n(1+\frac{4}{3}H(d/n))}$. Ceci prouve donc que les deux modèles ne sont pas équivalents.

L'algorithme d'énumération rapide que j'ai conçu avec les auteurs cités ci-dessus [41] ramène le nombre d'opérations à $d2^n$, mais il est intrinsèquement séquentiel. *A contrario*, le problème lui-même est en fait « intrinsèquement parallélisable » : en devinant les n_1 premières variables, on partitionne le problème de départ en 2^{n_1} sous-problèmes indépendants en n_2 variables avec $n_1 + n_2 = n$. Chacun de ces sous-problèmes nécessite le stockage de $\binom{n_2}{d}$ coefficients. On pourrait donc imaginer la machine suivante : on utilise 2^{n_1} processeurs, munis chacun d'une mémoire locale de taille $\binom{n_2}{d}$, plus une mémoire globale qui contient le système initial. Au début du calcul, le système initial est *broadcasté* à tous les processeurs, qui calculent à la volée les coefficients de leur système local, puis le résolvent. Le temps nécessaire est $T = \binom{n}{d} + 2^{n_2}$ et la taille de la machine est $\binom{n}{d} + 2^{n_1}\binom{n_2}{d}$. Une petite séance d'optimisation numérique révèle que ceci est meilleur que la transformée de Mœbius lorsque $d/n < 0.09424$. La solution optimale consiste à choisir n_1 et n_2 tels que $\binom{n}{d} = 2^{n_2}$, donc $n_2 = nH(d/n)$. La taille de la machine est alors dominée par celle des sous-systèmes. Le coût total est alors $2^{\mu n}$, avec $\mu = 1 + H\left(\frac{d}{n}\right) H\left(\frac{d}{n}\frac{1}{H\left(\frac{d}{n}\right)}\right)$.

Dinur décrit dans [88] un algorithme inspiré de la transformée de Mœbius pour évaluer un polynôme

de degré d sur les 2^n entrées possibles. En fait, il s'agit de faire la transformée de Mœbius via un parcours en profondeur, avec un arrêt anticipé. L'algorithme a une description récursive, et démarre avec le polynôme initial en mémoire :

— [*cas de base*] Si le polynôme a k variables, avec $2^k \leq \binom{n}{d}$, alors utiliser la transformée de Mœbius originale pour l'évaluer puis terminer.
— [*cas récursif*] Sinon, fixer $x_k \leftarrow 0$ et $k \leftarrow k-1$; invoquer récursivement l'algorithme ; fixer ensuite $x_k \leftarrow 1$; invoquer récursivement l'algorithme ; restaurer l'état initial et terminer.

La quantité de mémoire nécessaire est au pire 2 fois la taille du polynôme de départ, car toutes les mises à jour du polynôme peuvent être effectuées en place. Pour effectuer la transformée de Mœbius, on copie le polynôme dans un nouveau tableau et l'algorithme fonctionne encore en place. Le nombre d'opérations est de l'ordre de 2^n. Mais l'algorithme a alors une présentation essentiellement séquentielle, et déterminer son coût n'est plus si évident que ça.

Si on s'y prend bien, obtenir le polynôme spécialisé avec $x_n \leftarrow 0$ est gratuit : il suffit d'ignorer les monômes contenant x_n, et si on dispose de leurs coefficients dans l'ordre lexicographique inverse, alors ils sont tous à la fin. Mais obtenir le polynôme spécialisé en $x_n \leftarrow 1$ nécessite $\sum_{i=0}^{d-1} \binom{n-1}{d-1}$ opérations arithmétiques (qui peuvent être faites en parallèle) : le coefficient du monôme $m \cdot x_n$ doit être ajouté à celui du monôme m. Cela nécessite la mise à jour d'une fraction environ d/n des coefficients.

Imaginons donc la machine suivante : un *Mesh* 3D de $\binom{n}{d}$ nœuds où chaque nœud a une mémoire de taille constante. Initialement, chaque nœud stocke un coefficient du polynôme. La spécialisation d'une variable nécessite une phase de communication qui prend un temps $\binom{n}{d}^{1/3}$. Chaque transformée de Mœbius nécessite elle aussi un petit nombre de phases de communications qui ont la même durée. Il y a $2^{n(1-H(d/n))}$ spécialisations et autant de transformées de Mœbius. La durée totale du calcul est donc $2^{n(1-2/H(d/n)/3)}$. Vu la taille de la machine, le coût total est finalement : $2^{n(1+H(d/n)/3)}$

Cette dernière solution est donc, quoi qu'il arrive asymptotiquement supérieure aux autres.

4.6.2 Survol de certains autres algorithmes

De nombreux autres algorithmes ont été proposés, et il serait trop long de tous les énumérer ici. La plupart d'entre eux n'ont jamais connu de succès pratique. Toutefois, quelques méthodes issues de la communauté cryptographique se distinguent car elles ont été appliquées avec plus ou moins de succès : le calcul des bases de Gröbner, l'algorithme XL et l'utilisation de SAT-solvers.

Les algorithmes de calcul de base de Gröbner, notamment F4 [100] et F5 [101], conçus Faugère, ont des implantations efficaces, que ce soit dans le système de calcul formel MAGMA ou bien dans la bibliothèque FGb programmée par Faugère [102] (dont le code source n'est pas public) et plus récemment dans la bibliothèque open-source msolve [27] de Berthomieu, Eder et Safey El Din. L'étude de la complexité de ces algorithmes est notoirement difficile ; les résultats connus figurent dans la thèse de Bardet et dans [15], où il est affirmé que l'algorithme F5 nécessite $2^{4.3n}$ opérations sur le corps pour résoudre un système de n polynômes quadratiques en n variables. Ceci rend l'algorithme avantageux si la taille du corps est importante (supérieure à 20), car il bat alors la recherche exhaustive.

Ces algorithmes sont plus efficaces sur des systèmes surdéterminés (plus de polynômes que de variables), et du coup la « méthode hybride » est avantageuse : énumérer toutes les valeurs possibles des n_1 premières variables, puis résoudre les 2^{n_1} systèmes de n équations en $n_2 = n - n_1$ variables. Il y a un nombre de variables n_1 optimal à deviner, qui est a priori linéaire en n : Yang et Chen [204], suivis de Bettale, Faugère et Perret [29] discutent de sa valeur.

Dans la communauté cryptographique, les algorithmes de calcul de base de Gröbner ont été découverts ou redécouverts sous l'appellation de méthode XL par Courtois, Klimov, Patarin et Shamir [79]. Il a ensuite été établi que XL, F4 et F5 sont de proches cousins [11]. De toutes façons, ils dérivent tous d'une idée de Lazard datant de 1983 [141] et qu'on pourrait résumer en disant : mettre sous forme échelonnée réduite des matrices de Macaulay suffisamment grandes.

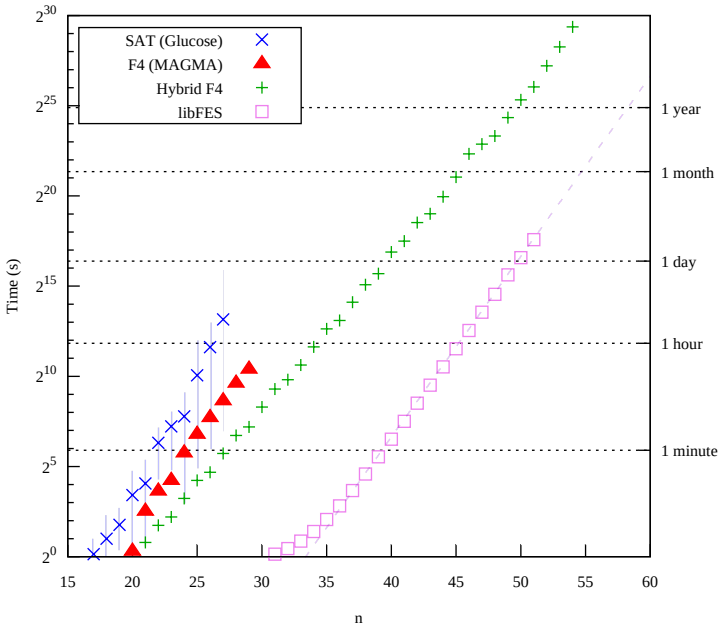


FIGURE 4.12 – Résolution de systèmes quadratiques aléatoires modulo 2.

Quelques résultats pratiques sont présentés figure 4.12. Voici des précisions sur ce qui a été chronométré :

libFES Recherche exhaustive efficacement implantée (`libfes`).

SAT Les polynômes sont convertis en CNF, et l'instance résultante résolue avec Glucose 2.2.

F4 Utilise l'implémentation de l'algorithme F4 de calcul des bases de Gröbner présente dans MAGMA [38] (V2.19-1). Avec 74Go de RAM, impossible de dépasser $n = 29$.

Hybrid-F4 Énumère n_1 variables, puis résout les 2^{n_1} systèmes de n équations en $n - n_1$ variables avec F4. La valeur empiriquement optimale de n_1 a été utilisée.

Un certain nombre de travaux, pour certains très récents, proposent des méthodes toujours exponentielles pour les systèmes de polynômes quadratiques sur $\mathbb{F}_2$, mais avec un exposant réduit :

- En 2012, Bardet *et al.* ont proposé l'algorithme BooleanSolve [16] déjà mentionné au chapitre 1, qui fonctionne de manière probabiliste en temps $2^{0.792n}$, en faisant des hypothèses algébriques sur les polynômes en entrée (qui ont l'air bien satisfaites par des polynômes aléatoires).
- En 2017, Lokshtanov *et al.* ont proposé l'algorithme basé sur la « méthode polynomiale » [153], lui aussi déjà mentionné au chapitre 1. Sa complexité en temps est de $2^{0.8765n}$. Il est remarquable que cet algorithme ne nécessite pas d'hypothèse spéciale sur les polynômes en entrée.
- En 2019, Björklund, Kaski et Williams proposent une amélioration de la méthode polynomiale [34] dont la complexité en temps baisse à $2^{0.804n}$.
- En 2021, Dinur améliore encore la méthode polynomiale [89]. La complexité en temps baisse encore à $2^{0.6943n}$.

À ma connaissance, aucun de ces algorithmes n'a été implanté et ils n'ont jamais résolu aucun système polynomial. Les auteurs de [16] conjecturent que leur algorithme bat la recherche exhaustive lorsque $n \geq 200$, ce qui signifierait qu'il est inutilisable (ce seuil est à prendre avec prudence en l'absence d'implantation). En 2021, Dinur a publié une version simplifiée et « concrètement efficace » [88] (d'après l'abstract) de la « méthode polynomiale » dont il a déjà été question plusieurs fois, dans les sections 1.7,

2.4 et 2.5. Il démontre des bornes concrètes (c'est-à-dire non-asymptotiques) sur la complexité en temps lorsque des polynômes aléatoires sont fournis en entrée : l'algorithme effectue moins de $n^2 2^{0.815n}$ opérations arithmétiques sur des bits et nécessite moins de $n^2 2^{0.63n}$ bits de mémoire.

Dans le modèle de la *Random Access Machine*, il semble que ces algorithmes puissent améliorer la recherche exhaustive, au moins pour de grandes valeurs de n. Mais combien coûtent-ils ? Répondre à cette question nécessite de plonger dans leurs descriptions car, dans la plupart des cas, leur complexité en espace n'est pas explicitée.

Il faut noter aussi l'algorithme « *crossbred* » de Joux-Vitse [126]. Sa complexité asymptotique n'est pas connue précisément, mais c'est le seul algorithme qui bat la recherche exhaustive en pratique (ce qui est remarquable). L'implantation originale n'est pas publique, mais Niederhagen, Ning et Yang en proposent une version qui utilise des GPUs [162].

4.6.3 L'algorithme le plus simple qui « bat la force brute »

> Up to now, the best complexity bound was reached by [...] exhaustive search.
>
> M. Bardet, J.-C. Faugère, B. Salvy, P.-J. Spaenlehauer, 2012 [16]

Un algorithme très simple permet (heuristiquement) de faire moins de 2^n opérations : il s'agit de se débrouiller pour rendre les sous-problèmes tellement surdéterminés qu'on peut les résoudre par linéarisation. Autrement dit : fixer $n_2 = \lfloor \sqrt{2m} - 2 \rfloor$, deviner les valeurs de $\hat{y}$ (rappelons qu'il s'agit des valeurs des n_1 premières variables), puis résoudre le système quadratique restant (en z) par linéarisation. Ceci signifie considérer chaque monôme quadratique du système « spécialisé en $\hat{y}$ » comme une variable indépendante ; trouver toutes les solutions du système linéaire (ce qui révèle les valeurs des variables de z) et vérifier qu'elles donnent bien une solution du système quadratique initial. Cet algorithme simple nécessite $\tilde{\mathcal{O}}\left(2^{n-\sqrt{2m}}\right)$ opérations, est facilement parallélisable et sa consommation en mémoire est nulle.

Ceci est (à peu de choses près) l'algorithme « *crossbred* » de Joux-Vitse [126] avec $D = 2$; c'est aussi la variante FXL de l'algorithme XL de Courtois-Klimov-Patarin-Shamir [79] avec $D = 2$; c'est un cas particulier de la « méthode hybride » [28] ; il est également décrit (de manière légèrement implicite, §8.2.2) dans la thèse de Trimoska [189]. Il est frappant de constater que cette technique d'une simplicité désarmante n'a jamais été décrite telle quelle, et que des méthodes asymptotiquement plus efficaces mais plus complexes ont été inventées avant.

Le facteur polynomial caché est majoré par m^3, mais on peut le faire baisser jusqu'à $m^{1.5}$. En effet, notons $\mathcal{V}$ le sous-espace de $\mathbb{F}_2[x]$ constitué des polynômes ne contenant *pas* de produit $z_i z_j$. Le point est qu'un polynôme dans $\mathcal{V}$ devient linéaire lorsqu'on fixe les valeurs des variables de y.

On part de l'espace vectoriel engendré par les polynômes de départ $\langle f_1, \ldots, f_m \rangle$ puis on calcule une base $g_1, \ldots, g_\ell$ de son intersection avec $\mathcal{V}$. Le choix de n_2 garantit que $\ell \geq 5n_2/2$. Si on fixe la valeur des variables $y_1, \ldots, y_{n_1}$, alors on se retrouve avec un système linéaire en n_2 variables, qu'on peut résoudre en temps $\mathcal{O}\left(m^{1.5}\right)$. En utilisant des ruses algorithmiques inspirées des implantations efficaces de la recherche exhaustive [41], on peut aussi évaluer les coefficients de ce système linéaire en temps $\mathcal{O}(m)$.

Le tout se prête à une implantation simple et efficace avec du *bitslicing* : supposons qu'on dispose d'un *circuit* qui prend en entrée une matrice A et un vecteur b sur $\mathbb{F}_2$ et qui décide si $Ax = b$ a une solution. Alors sur une machine qui a des registres de w bits, on peut exécuter efficacement w copies du circuit en parallèle (on a $w \geq 256$ sur la plupart des processeurs contemporains).

Un circuit peut être décrit par un programme écrit dans un sous-ensemble très restreint des langages de programmation habituels, dans lequel seules les boucles dont le nombre d'itérations est connu à l'avance sont autorisées et où les instructions conditionnelles sont interdites.

Algorithme G (*pivot de Gauss*). Sous l'hypothèse que la matrice A, de taille $\ell \times n_2$, est de rang plein, décide si le système linéaire $Ax = b$ possède une solution. Les opérations sur les lignes sont à effectuer dans la matrice A et le vecteur b simultanément.

G1. [Boucle sur les colonnes] Pour $1 \leq j \leq n_2$, faire :
G2. [Recherche pivot] Faire $i \leftarrow j$. Tant que $i \leq \ell$ et $A[i, j] = 0$, incrémenter i.
G3. [Non trouvé ?] Si $i > \ell$, s'arrêter (le système a un défaut de rang).
G4. [Permuter] Échanger les lignes i et j.
G5. [Boucle sur les lignes] Pour $j < i \leq \ell$, faire :
G6. [Élimination] Si $A[i, j] = 1$, ajouter la j-ème ligne à la i-ème.
G7. [Incohérent ?] Pour $n_2 < i \leq \ell$, faire : si $b[i] = 1$, renvoyer « incohérent ».
G8. [Cohérent] Renvoyer « cohérent ».

FIGURE 4.13 – Un vénérable algorithme : une version simple du pivot de Gauss. Exercice : en tirer un *circuit*.

Vu la faible taille de A, la manière naturelle d'attaquer le problème serait le pivot de Gauss. On peut le simplifier un peu (donc l'accélérer) en faisant en sorte que la matrice soit généralement de rang plein. Elle est de taille $\ell \times n_2$, et on s'est débrouillé exprès pour que $\ell \geq \frac{5}{2}n_2$. Si on suppose que la matrice est aléatoire, alors elle est de rang plein avec probabilité

$$\prod_{i=\ell+1-n_2}^{\ell} \left(1 - 2^{-i}\right) \geq \left(1 - 2^{n_2-\ell}\right)^{n_2} \approx 1 - n_2 2^{n_2-\ell},$$

ce qui est raisonnablement proche de 1. L'algorithme qu'il semble donc naturel d'implanter est donné figure 4.13 . Mais pour en faire un circuit, il faudrait éliminer le « tant que » de l'étape G2 ainsi que les « Si » des étapes G3, G6 et G7. Bref, il faut tout changer. Il est possible d'effectuer cette transformation automatiquement (voire par exemple les travaux à FUN de Elmasry et Katajainen [97]). À la fin, on se retrouve avec environ $3{n_2}^2\ell$ portes logiques.

Ceci aboutit à un peu moins de 1000 lignes de C. La recherche exhaustive s'exécute en temps $\alpha T(n)$ (pour une certaine constante α qui dépend de la machine), tandis que cet algorithme s'exécute en temps $P(m)2^{n-\sqrt{2m}}$ pour un certain polynôme P qui dépend lui aussi de la machine. Il s'ensuit que cette technique est fatalement plus rapide que la recherche exhaustive pour de grandes valeurs de m, et déterminer le seuil est facile. Sur mon laptop, c'est $m = 48$, une valeur surprenamment basse, alors que l'algorithme est *sensiblement plus facile* à programmer que la recherche exhaustive.

Ceci confirme que l'algorithme Crossbred de Joux-Vitse est, empiriquement, la meilleure solution actuelle pour résoudre en pratique des systèmes quadratiques modulo 2.

L'algorithme peut se généraliser de plusieurs manières. Le véritable algorithme Crossbred de Joux-Vitse s'obtient notamment en calculant l'intersection de $\mathcal{V}$ avec l'espace vectoriel engendré par les *multiples* tf_i des polynômes de départ, où t parcourt tous les monômes d'un degré fixé. Le record obtenu par Joux et Vitse (résolution d'un système aléatoire de 148 équations en 74 inconnues) a été obtenu avec des monômes t de degré 2.

Alternativement, l'algorithme FXL s'obtient en partant des multiples tf_i, en fixant une partie des variables puis en résolvant le système restant par linéarisation. C'est aussi la technique à la base de l'algorithme de la prochaine section.

4.6.4 Algorithme BooleanSolve de Bardet, Faugere, Salvy et Spaenlehauer [16]

> We also give a rough estimate for the actual threshold between our method and exhaustive search, which is as low as 200, and thus very relevant for cryptographic applications.
>
> M. Bardet, J.-C. Faugère, B. Salvy, P.-J. Spaenlehauer, 2012 [16]

Cet algorithme utilise des manipulations algébriques dans le style des bases de Gröbner pour résoudre les sous-systèmes. L'algorithme effectue une recherche exhaustive sur les $n_1 = (1-\gamma)n$ premières variables, où γ est un paramètre qui sera déterminé plus tard.

Pour résoudre les sous-systèmes $\{P_i(z)\}_{i=1}^m$ en n_2 variables, commencer par vérifier si 1 peut s'écrire comme une combinaison polynomiale de $P_1, \ldots, P_m$. Si c'est le cas, alors le système « spécialisé en $\hat{y}$ » n'a pas de solution en z d'après le théorème des zéros de Hilbert. Dans le cas contraire, chercher z par recherche exhaustive et terminer l'algorithme.

Déterminer si 1 peut s'écrire comme une combinaison polynomiale des $\{P_i\}_{i=1}^m$ peut se décider en résolvant un système linéaire sur $\mathbb{F}_2$: il existe des polynômes $g_1, \ldots, g_m$ de degré inférieur ou égal à d tels que $g_1P_1 + \cdots + g_mP_m = 1$ si et seulement s'il existe une combinaison *linéaire* des tP_i dans laquelle tous les monômes s'annulent, à l'exception de 1, où t parcourt l'ensemble des monômes de degré $\leq d-2$ en les variables z.

Le problème consiste à choisir la bonne valeur de d. Sous l'hypothèse usuelle de semi-régularité des polynômes en entrée, il est suffisant de prendre pour d l'indice du premier coefficient négatif de la série $\frac{(1+t)^{\gamma n}}{(1-t)(1+t^2)^n}$. En utilisant des techniques d'analyse complexe, une estimation asymptotique peut être dérivée :

$$d \sim n\gamma M\left(\frac{1}{\gamma}\right), \quad M(x) = -x + \frac{1}{2} + \frac{1}{2}\sqrt{2x^2 - 10x - 1 + 2(x+2)\sqrt{x(x+2)}}.$$

Pour la bonne valeur de d, le système linéaire à résoudre s'écrit $Ax = 0$, où la plus grande dimension de la matrice est $N = \tilde{\mathcal{O}}\left(2^{\gamma n H(M(1/\gamma))}\right)$, et où H désigne la fonction d'entropie binaire. Cette matrice est très creuse, avec moins de n^2 coefficients par ligne. Résoudre un tel système par élimination gaussienne coûterait $\mathcal{O}\left(N^3\right)$; mais on peut exploiter son caractère creux avec une méthode itérative comme l'algorithme de Wiedemann, avec une complexité de l'ordre de $\tilde{\mathcal{O}}\left(N^2\right)$. On peut noter que la matrice A admet une représentation compacte : ses coefficients sont aisément recalculables à la volée à partir de d et des polynômes. Par conséquent, exécuter l'algorithme de Wiedemann nécessite principalement le stockage de quelques vecteurs de taille N.

Sous des hypothèses de généricité additionnelles sur le système à résoudre (« *strong semi-regularity* »), la phase de recherche exhaustive ne sera jamais effectuée en vain : elle trouvera toujours une solution, donc il suffit de la faire une seule fois. La complexité en temps de l'algorithme est donc :

$$2^{n[1-\gamma+2\gamma H(M(1/\gamma))]} + 2^{\gamma n}.$$

Minimisation du temps d'exécution

Il suffit de minimiser l'exposant $1 - \gamma + 2\gamma H(M(1/\gamma)$. Cette expression atteint un minimum de 0.792 lorsque $\gamma = 0.55...$, et la phase finale de recherche exhaustive est alors négligeable. Ceci établit le temps d'exécution annoncé. Notons que les valeurs des $n_1 = 0.45n$ premières variables sont énumérées, ce qui aboutit à $2^{0.45n}$ sous-problèmes indépendants, qui nécessitent la résolution d'un système linéaire exponentiellement gros.

Objectif	Nombre d'opérations	Coût
Itérations $(1-\gamma)$	0.45	0.58
Espace	0.171	0.108
# Opérations	0.792	0.801
Coût	0.963	0.837

FIGURE 4.14 – Caractéristiques des choix optimaux de paramètres pour l'algorithme BooleanSolve.

Minimisation du coût

Avec $\gamma = 0.55...$, on trouve que l'algorithme nécessite $2^{0.171n}$ bits de mémoire (pour stocker un vecteur de taille N). Si ces systèmes étaient résolus par une machine séquentielle (en temps quadratique), alors le coût de la résolution serait cubique, et le coût total de l'algorithme serait $2^{0.963n}$, ce qui est légèrement meilleur que la recherche exhaustive.

Mais cette valeur de γ avait été choisie pour minimiser le nombre d'opérations. La valeur qui minimise le coût est $\gamma = 0.275...$. Le coût est alors $2^{0.8875n}$, la consommation en mémoire est $2^{0.054n}$ et le nombre d'opérations est $2^{0.833n}$.

Ceci peut être amélioré en remplaçant l'algorithme séquentiel de résolution de systèmes linéaires creux par la version parallèle discutée section 3.1. L'exposant du coût de la résolution du système linéaire passe de 3 à $\frac{7}{3}$, et minimiser le coût de BooleanSolve revient alors à minimiser $1 - \gamma + \frac{7}{3}\gamma H(M(1/\gamma)$. Le coût diminue jusqu'à $2^{0.837n}$, et il est atteint pour $\gamma = 0.416...$, tandis que le nombre d'opérations augmente quant à lui à $2^{0.801n}$ et que la consommation de mémoire diminue à $2^{0.108n}$. Cette fois, les valeurs des $n_1 = 0.58n$ premières variables sont énumérées. La figure 4.14 résume ces choix.

Discussion

Faire baisser le coût revient à faire plus de recherche exhaustive et moins de choses « intelligentes ». Il est notable que chercher à minimiser le coût aboutit à choisir des paramètres différents de celui de minimiser le nombre d'opérations.

Il est probable qu'une tentative d'implantation de l'algorithme aboutirait à choisir la plus petite valeur de γ (supérieure à 0.45) qui permet que les systèmes linéaires tiennent dans la mémoire d'un seul nœud, afin d'éviter les surcoûts et la complexité d'une implantation parallèle de l'algorithme de Wiedemann.

4.6.5 Algorithme de Lokshtanov *et al* [153]

Essayons de résoudre la question posée section 2.5, à savoir : cet algorithme peut-il « battre la force brute » ? On va voir que ce n'est pas le cas et qu'on se heurte à plusieurs des obstructions relevées section 3.3.

Tout d'abord, voici un petit échauffement. Partant d'un système $\{f_i(x)\}_{i=1}^m$, on considère le polynôme :

$$F(x) = (1 + f_1(x))(1 + f_2(x)) \dots (1 + f_m(x)).$$

Une chaîne de bits $\hat{x}$ est solution du système de départ si et seulement si $F(\hat{x}) = 1$. On peut voir F comme la fonction indicatrice de l'ensemble des solutions. F est de degré $2m$, et calculer ses coefficients a un coût prohibitif. Mais F peut être *approximé* par un polynôme de degré plus faible. On tire au hasard une matrice A de taille $k \times n$ puis on pose :

$$R_i(x) = \sum_{j=1}^{m} A_{ij} f_j(x),$$

$$\widetilde{F}(x) = \prod_{i=1}^{\ell} (1 + R_i(x))$$

Chacun des $R_i(x)$ est une combinaison linéaire aléatoire des polynômes de départ. Le point est que $F(\hat{x}) = 1$ signifie que $\hat{x}$ est une solution, et ceci entraine $\widetilde{F}(\hat{x}) = 1$. Dans le cas contraire, si $F(\hat{x}) = 0$, alors on ne peut avoir $\widetilde{F}(\hat{x}) = 1$, qu'avec probabilité $2^{-\ell}$. En effet, pour avoir une « erreur », il faudrait que $\hat{x}$ soit une solution accidentelle du système $\{R_i(x)\}_{i=1}^{\ell}$. En attendant, $\widetilde{F}$ est de degré 2ℓ.

Voilà pour l'échauffement. Maintenant, passons aux choses sérieuses. On pose

$$G(y) = 1 + \prod_{\hat{z}\in\{0,1\}^{n_2}} (1 + F(y, \hat{z}))\,.$$

Et le « préfixe » $\hat{y}$ peut se compléter en une solution complète $\hat{x} = (\hat{y}, \hat{z})$ si et seulement si $G(\hat{y}) = 1$. Lorsqu'un tel préfixe $\hat{y}$ avec $G(\hat{y}) = 1$ est trouvé, il suffit alors de faire une recherche exhaustive sur les n_2 variables restantes pour obtenir une solution complète, puis terminer l'algorithme.

L'idée algorithmique principale consiste à éviter de travailler avec G qui est un polynôme monstrueux (il peut avoir 2^n termes), mais d'en utiliser à la place une approximation de degré plus faible :

$$\widetilde{F}_{\hat{z}}(y) = \prod_{i=1}^{n_2+2} \left(1 - \sum_{j=1}^{n} A_{j\hat{z}} f_j(y, \hat{z})\right),$$
$$\widetilde{G}(y) = \sum_{\hat{z}\in\{0,1\}^{n_2}} b_{\hat{z}} \widetilde{F}_{\hat{z}}(y).$$

où les coefficients b_z et A_{zj} sont choisis uniformément au hasard dans $\{0, 1\}$. Comme au-dessus, $\widetilde{F}_{\hat{z}}(y)$ approxime $F(y, \hat{z})$, la seule différence étant que cette fois $\hat{z}$ est fixé. Si $\hat{z}$ peut se compléter en une solution $(\hat{y}, \hat{z})$ du système de départ, alors $\widetilde{F}_{\hat{z}}(\hat{y}) = 1$. Dans le cas contraire, $\widetilde{F}_{\hat{z}}(\hat{y}) = 0$ avec probabilité $1 - 2^{-n_2-2}$. L'erreur obtenue sur $\widetilde{G}$ est plus importante, à cause de la somme sur les 2^{n_2} valeurs possibles de $\hat{z}$. Plus précisément, on a :

$$G(y) = 1 \qquad \Longrightarrow \qquad \Pr\left(\widetilde{G}(y) = 1\right) \geq \frac{1}{2},$$
$$G(y) = 0 \qquad \Longrightarrow \qquad \Pr\left(\widetilde{G}(y) = 1\right) \leq \frac{1}{4}.$$

L'algorithme fonctionne alors de la façon suivante. Répéter $100n$ fois : tirer au hasard les $b_{\hat{z}}$ et les $A_{j\hat{z}}$; calculer le polynôme $\widetilde{G}$; évaluer $\widetilde{G}(\hat{y})$ sur l'ensemble des $\hat{y} \in \{0, 1\}^{n_1}$; pour chaque $\hat{y}$, compter le nombre de fois où on trouve $\widetilde{G}(\hat{y}) = 1$. Un $\hat{y}$ pour lequel ce compteur excède $40n$ peut être complété en une solution du système de départ avec très forte probabilité.

Évaluer $\widetilde{G}(\hat{y})$ sur tous les $\hat{y}$ possibles peut se faire en temps (et en espace) $\tilde{\mathcal{O}}\left(2^{n_1}\right)$ en utilisant par exemple la transformée de Mœbius.

Optimisation du temps d'exécution

Soit $0 < \delta < 1/2$ un paramètre dont on déterminera la valeur plus tard. On fixe $n_1 = (1 - \delta)n$ et $n_2 = \delta n$.

Augmenter le paramètre δ accélère l'évaluation de $\widetilde{G}(y)$ (ça fait diminuer le nombre de variables), mais cela peut augmenter le coût de la construction des $\widetilde{F}_{\hat{z}}(y)$ et de $\widetilde{G}(y)$: ce sont des polynômes en n_1 variables de degré $2n_2 + 4$. Par conséquent, il peuvent avoir environ $\binom{(1-\delta)n}{2\delta n}$ termes de degré $2\delta n$. Ceci atteint un maximum de $2^{0.81n}$ pour

$$\delta = \frac{1}{3}\left(\frac{4}{529}\right)^{\frac{1}{3}}\left(3\sqrt{69} - 23\right)^{\frac{1}{3}} - \frac{23\left(\frac{4}{529}\right)^{\frac{2}{3}}}{3\left(3\sqrt{69} - 23\right)^{\frac{1}{3}}} + \frac{1}{3} \approx 0.177$$

Le nombre des termes de degré $2\delta n$ domine asymptotiquement le nombre de termes de degrés inférieurs tant que $\delta < 1/5$. En utilisant l'approximation classique des coefficients binomiaux, on trouve que ces polynômes possèdent environ $2^{n(1-\delta)H\left(\frac{2\delta}{1-\delta}\right)}$ termes, où H désigne encore la fonction d'entropie binaire.

Calculer directement le polynôme $\widetilde{G}$ se fait en sommant les $2^{\delta n}$ polynômes $\widetilde{F}_{\hat{z}}(y)$, qui sont eux-mêmes le produit de δn polynômes quadratiques en $(1-\delta)n$ variables. Admettons (de manière un peu optimiste) que le temps de calcul de cette multiplication est proportionnel à la taille de la sortie. Par conséquent, le temps nécessaire au calcul de $\widetilde{G}$ est environ $2^{n\left[\delta+(1-\delta)H\left(\frac{2\delta}{1-\delta}\right)\right]}$.

Minimiser le nombre d'opérations revient à équilibrer la complexité du calcul et de l'évaluation de $\widetilde{G}$. La valeur optimale de δ est donc la solution de

$$\delta + (1-\delta)H\left(\frac{2\delta}{1-\delta}\right) = 1-\delta,$$

ce qui donne $\delta = 0.12375....$ On obtient donc la complexité annoncée de $2^{0.877n}$. La complexité spatiale (c'est-à-dire l'espace occupé par le polynôme $\widetilde{G}$) est la même.

Optimisation du coût

L'algorithme a un « ratio temps-mémoire » égal à un, donc il accède à la mémoire de manière intensive, ce qui est inquiétant. On avait déjà conclu, section 2.5, qu'il était inférieur à la recherche exhaustive sur la base du caractère fini de la vitesse de la lumière .

Le coût de cet algorithme peut-il passer sous la barre fatidique des 2^n en ajustant le paramètre δ ? En tout cas, les deux phases qui dominent son nombre d'opérations (la multiplication de polynômes et la transformée de Mœbius) sont toutes les deux sujettes à une borne inférieure VLSI de type $AT \geq n^{3/2}$ comme indiqué section 3.3, et on peut donc supposer que leur coût est minoré par $n^{4/3}$.

Le coût du calcul de $\widetilde{G}$ est donc minoré par $2^{\delta n+\frac{4}{3}n(1-\delta)H\left(\frac{2\delta}{1-\delta}\right)}$, et le coût de l'énumération de toutes les valeurs de $\widetilde{G}$ est également minoré par $2^{\frac{4}{3}n(1-\delta)}$. Équilibrer le coût des deux opérations revient alors à résoudre :

$$\delta + \frac{4}{3}(1-\delta)H\left(\frac{2\delta}{1-\delta}\right) = \frac{4}{3}(1-\delta),$$

et la valeur optimale de δ est maintenant $\delta = 0.1319....$ Le coût résultant est $2^{1.157n}$.

Pour tenter d'améliorer un peu les choses, on peut imaginer d'utiliser la version améliorée de la transformée de Mœbius de la section 4.6.1 pour énumérer tous les zéros de $\widetilde{G}$. Rappelons que c'est un polynôme en $(1-\delta)n$ variables de degré $2\delta n$. Mais dans ce cas, le coût des deux phases devient une fonction croissante de δ, donc on a complètement intérêt à choisir $\delta = 0$. Cela revient donc à faire la recherche exhaustive.

Pour conclure, dans le modèle du coût, l'algorithme de Lokshtanov *et al.* ne bat *pas* la force brute.

4.6.6 Algorithme « concrètement efficace » de Dinur [88]

Il s'agit d'une extension de l'algorithme de Lokshtanov. On utilise toujours le polynôme indicateur des solutions :

$$F(x) = (1+f_1(x))(1+f_2(x))\dots(1+f_m(x)).$$

Une solution $(\hat{y},\hat{z})$ est dite « isolée » si $(\hat{y},\hat{z}')$ n'est pas une solution pour tout $\hat{z}' \neq \hat{z}$. Si on suppose que les polynômes ont été générés aléatoirement, alors une solution donnée est isolée avec probabilité $1-2^{n_2-m}$.

On pose

$$V_0(y) = \sum_{\hat{z}\in\{0,1\}^{n_2}} F(y,\hat{z}),$$
$$V_i(y) = \sum_{\hat{z}\in\{0,1\}^{n_2-1}} F(y,\hat{z}_1,\dots,\hat{z}_{i-1},0,\hat{z}_{i+1},\dots,\hat{z}_{n_2}).$$

Le point, c'est que si une solution $(\hat{y}, \hat{z})$ est isolée, alors $V_0(\hat{y}) = 1$ et on a de plus $\hat{z}_i = V_i(\hat{y})$. On peut donc non seulement détecter mais aussi récupérer les solutions isolées en évaluant V_0 et les V_i sur tous les $\hat{y}$ possibles. Pour gagner en efficacité, on remplace F par son approximation $\widetilde{F}$ qui est de degré plus faible. On tire aléatoirement une matrice A de taille $(n_2+1) \times m$, puis on pose :

$$\begin{aligned}
R_i(x) &= \sum_{j=1}^{m} A_{ij} f_j(x), \\
\widetilde{F}(x) &= (1 + R_1(x))(1 + R_2(x)) \dots (1 + R_{n_2+1}(x)), \\
\widetilde{V}_0(y) &= \sum_{\hat{z} \in \{0,1\}^{n_2}} \widetilde{F}(y, \hat{z}), \\
\widetilde{V}_i(y) &= \sum_{\hat{z} \in \{0,1\}^{n_2-1}} \widetilde{F}(y, \hat{z}_1, \dots, \hat{z}_{i-1}, 0, \hat{z}_{i+1}, \dots, \hat{z}_{n_2}).
\end{aligned}$$

Évaluer les $\widetilde{V}_i$ sur les 2^{n_1} entrées possibles révèle l'ensemble des solutions isolées de $\{R_i\}_{i=1}^{n_2+1}$. On a déjà vu qu'une solution $\hat{x}$ de $\{f_i\}_{i=1}^{m}$ a de fortes chances d'être isolée. C'est nécessairement aussi une solution du système condensé $\{R_i\}_{i=1}^{n_2+1}$, mais comme il y a des solutions parasites, $\hat{x}$ n'est isolée dans $\{R_i\}_{i=1}^{n_2+1}$ qu'avec probabilité 1/2.

Pour trouver les solutions du système de départ, on répète donc un petit nombre de fois la procédure suivante : générer les R_i aléatoirement, construire les $\widetilde{V}_i$, évaluer les $\widetilde{V}_i$ sur tous les $\hat{y}$ possibles, tester les solutions suggérées sur le système de départ.

Les polynômes $\widetilde{V}_i$ sont de degré $n_2 + \mathcal{O}(1)$ en n_1 variables. Ils ont donc $\binom{n_1}{n_2}$ termes de degré maximal $\approx n_2$. Au lieu de les construire directement en effectuant des multiplications comme dans l'algorithme de Lokshtanov *et al.*, on obtient leurs coefficients par une procédure d'interpolation qui repose sur la transformée de Mœbius. Pour calculer les coefficients des $\widetilde{V}_i$, on les évalue sur l'ensemble des $\hat{y}$ de poids de Hamming inférieur ou égal à n_2. Et pour cela, on trouve (par recherche exhaustive) les solutions (en z) de $\{R_i(\hat{y}, z)\}_{i=1}^{n_2+1}$ pour tout $\hat{y}$ de poids inférieur à n_2. La parité du nombre de solutions donne la valeur souhaitée.

La construction des $\widetilde{V}_i$ nécessite donc $\tilde{\mathcal{O}}\left(2^{n_2}\binom{n_1}{n_2}\right)$ opérations. L'évaluation des $\widetilde{V}_i$ sur tous les $\hat{y}$ possibles nécessite $\tilde{\mathcal{O}}(2^{n_1})$ opérations, que ce soit par la transformée de Mœbius ou par l'algorithme d'énumération rapide de [41].

Optimisation du temps d'exécution

Posons $n_1 = \lambda n$ et $n_2 = (1-\lambda)n$. Le nombre d'opérations s'écrit alors :

$$2^{(1-\lambda)n} 2^{\lambda n H((1-\lambda)/\lambda)} + 2^{\lambda n}$$

Et pour l'optimiser on résout

$$1 - \lambda + \lambda H\left(\frac{1-\lambda}{\lambda}\right) = \lambda$$

La solution est $\lambda = 0.8149...$, et cela donne un temps d'exécution de $2^{0.815n}$, comme annoncé. On admet que l'espace nécessaire est juste celui du stockage des coefficients des $\widetilde{V}_i$, c'est-à-dire $2^{n\lambda H((1-\lambda)/\lambda)}$. Encore une fois, on trouve bien $2^{0.63n}$ comme annoncé.

Optimisation du coût

Du point de vue du coût, l'opération critique est la construction du polynôme $\widetilde{V}_0$ par interpolation. Cette opération est « FFT-like », donc son coût est de $\binom{n_1}{n_2}^{4/3}$. Les phases de recherche exhaustive sur des polynômes quadratiques peuvent s'effectuer avec un coût proportionnel au nombre d'opérations, mais la recherche des solutions de V_0 par recherche exhaustive coûte potentiellement $N^{4/3}$ (avec la transformée de Mœbius), car ce polynôme est de degré $\Omega(n)$.

On utilise $p = \binom{n_1}{n_2}$ processeurs disposant chacun d'une mémoire constante. Chaque processeur effectue une recherche exhaustive sur n_2 bits pour évaluer $\widetilde{V}_i$ sur un point, puis ensemble ils effectuent une interpolation. On admet qu'évaluer $\widetilde{V}_i$ sur les 2^{n_1} valeurs possibles de $\hat{y}$ coûte $2^{n_1}\binom{n_1}{n_2}^{1/3}$ comme discuté section 4.6.1. On admet aussi que le coût de l'interpolation est identique. Le coût est donc minoré par $2^{\mu n}$ avec

$$1 - \lambda + \lambda H\left(\frac{1-\lambda}{\lambda}\right) \leq \mu, \qquad \text{solutions de } \{R_i(\hat{y}, z)\}_{i=1}^{n_2+1}$$

$$\lambda + \frac{1}{3}\lambda H\left(\frac{1-\lambda}{\lambda}\right) \leq \mu. \qquad \text{Interpolation et énumération des } \widetilde{V}_i$$

Une solution optimale est fournie par $\lambda = 0.712$ et $\mu = 0.9625$. Le coût est donc très légèrement meilleur que celui de la recherche exhaustive, et ce n'est pas si mal. Par contre, on a déjà argumenté qu'en pratique l'algorithme était inutilisable.

Chapitre 5

Attaques génériques dans le modèle du coût

> [...] this attack leads to a data complexity of $2^{7n/8}$ chosen plaintexts, an online phase with time complexity of $2^{3n/4}$ operations, and a memory complexity of $2^{7n/8}$ [...]
>
> J. Guo, J. Jean, I. Nikolić et Y. Sasaki, 2016, [117]

Les attaques génériques en cryptographie symétrique sont dans le fond des attaques sur les *modes opératoires*. Un mode opératoire réalise un mécanisme cryptographique à partir d'une primitive cryptographique plus simple. Par exemple, les réseaux de Feistel réalisent une bijection pseudo-aléatoire P^F sur $2n$ bits à partir d'une fonction pseudo-aléatoire F sur n bits. Dans ce contexte-là, une attaque générique casse P^F même si F est un composant *idéal*, par exemple une fonction aléatoire ou une permutation aléatoire.

Certaines attaques génériques ont été des surprises lors de leur publication, et certaines sont sophistiquées. On peut citer par exemple les attaques contre le mode opératoire de Merkle-Damgård, notamment l'attaque en multi-collision de Joux en 2004 [124] et l'attaque en seconde préimage de Kelsey et Schneier [127] en 2005.

Ces attaques exploitent souvent le paradoxe des anniversaires, et par conséquent leur complexité est généralement trop élevée pour qu'elles puissent être réalisables en pratique avec des tailles cryptographiques raisonnables ($n = 128$). En effet, dans les mécanismes cryptographiques usuels, les tailles sont choisies pour résister aux attaques génériques. De plus, pour bénéficier de « l'effet paradoxe des anniversaire », il faut souvent beaucoup de mémoire. Les chercheurs qui travaillent sur ces attaques se placent donc de manière affirmée dans le champ de la théorie. Les attaques produites sont considérées comme valides si elles sont plus rapide que la recherche exhaustive (et éventuellement si elles n'ont pas besoin de faire chiffrer tous les clairs possibles). Comme l'écrivent les auteurs de l'article « *Meet-in-the-Middle Attacks on Classes of Contracting and Expanding Feistel Constructions* » cité au début de ce chapitre :

> [...] without reaching the time complexity bound limited by exhaustive key search and the data complexity bound limited by the code book. [1]

Une exception notable au raisonnement précédent a été donnée par Bhargavan et Leurent en 2016 [30]. En exploitant des collisions sur des blocs de 64-bits lorsqu'un système de chiffrement par bloc de cette taille est utilisé en mode CBC, ils parviennent à récupérer des informations significatives, telles que des *cookies* de session, à travers le chiffrement. Le mécanisme de chiffrement est sémantiquement sûr... jusqu'à la borne du paradoxe des anniversaires, qu'on peut ici atteindre en pratique.

1. Je suggère d'ajouter qu'il serait souhaitable que la machine nécessaire à l'attaque puisse fonctionner suffisamment longtemps pour pouvoir lire toute son entrée et lire/écrire toute sa mémoire au moins une fois.

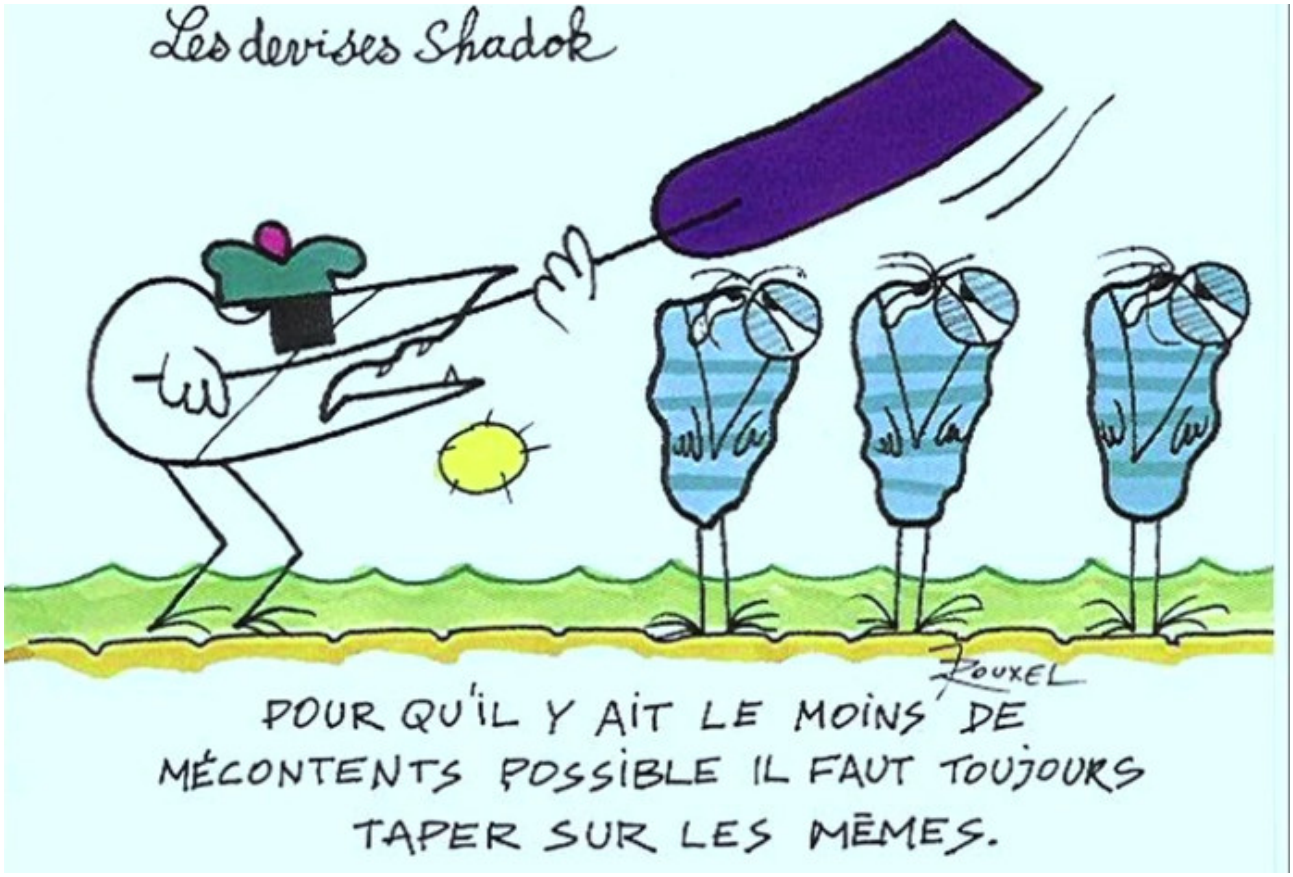


FIGURE 5.1 – Les Shadocks, Jacques Rouxel

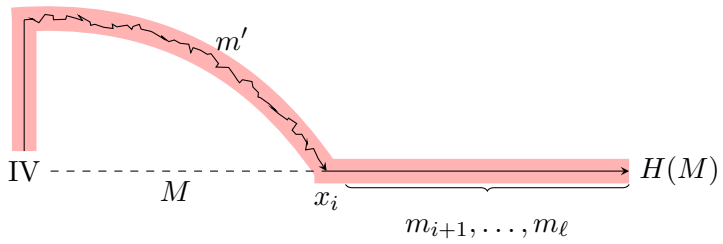


FIGURE 5.2 – La « *long message attack* ».

Le point, c'est que ces attaques peuvent être étudiées asymptotiquement en faisant grandir le paramètre de sécurité. Il est donc particulièrement intéressant de les examiner d'un point de vue asymptotique et de comparer leur coût à celui des attaques naïves qu'elles sont censées améliorer. Dans beaucoup de cas, leur coût est facile à étudier car elles effectuent $\mathcal{O}(N)$ accès aléatoires à une mémoire de taille $\mathcal{O}(N)$, donc le théorème 1 suggère que leur coût est de l'ordre de $N^{4/3}$. Par conséquent, si une attaque nécessite plus de $2^{3n/4}$ opérations et plus de $2^{3n/4}$ bits de mémoire, alors elle mérite un examen particulier.

5.1 Échauffement : la *long-message attack*

Une fonction de hachage itérée H^f calcule l'empreinte d'un message M en le découpant en une séquence de blocs $M = m_0 \| m_1 \| \ldots \| m_\ell$ de taille fixe, puis en itérant une fonction de compression f à partir d'un vecteur d'initialisation :

$$\begin{aligned} x_{-1} &= IV, \\ x_i &= f(x_{i-1}, m_i). \end{aligned}$$

L'empreinte de M est x_ℓ. Ceci nécessite d'appliquer un *bourrage* dans le cas où la taille du message n'est pas un multiple de la taille des blocs. Le schéma de bourrage le plus connu est celui de Merkle-Damgård, qui consiste à ajouter la taille du message dans le bourrage, car ceci étend la résistance aux collisions de f à la construction itérée H^f.

Le schéma de bourrage le plus simple possible consiste à ajouter des bits à zéro. Il est bien connu que ceci n'est pas résistant aux collisions (M et $M\|0$ collisionnent), mais ce n'est pas non plus résistant aux secondes préimages à cause de l'attaque des longs messages, représentée figure 5.2 et dont voici la description. Pour calculer une seconde préimage d'un message M, commencer par hacher M, ce qui donne une séquence d'états internes $x_0, x_1, \ldots, x_\ell$. Ensuite, répéter tant que ça n'a pas abouti ($2^n/\ell$ fois en moyenne) : choisir un bloc de message m' aléatoire et vérifier si $f(x_{-1}, m') = x_i$ pour $0 \leq i < \ell$. Alors $M' = m' \| m_i \| \ldots \| m_\ell$ est une seconde préimage de M.

L'attaque nécessite $\ell + 2^n/\ell$ évaluations de la fonction de compression ; elle fonctionne mieux sur de longs messages M, d'où son nom. Le nombre d'opérations minimal est atteint avec $\ell = 2^{n/2}$.

Discutons maintenant du coût. Cette attaque simple permet encore d'illustrer la différence entre le modèle « NIST » et le modèle du coût.

Dans le modèle « NIST », où chaque accès à une mémoire de taille M coûte $M^{1/3}$, le coût de l'attaque est $\ell^{4/3} + 2^n/\ell^{2/3}$. Le premier terme correspond au stockage des états intermédiaires de la fonction de hachage, le second à la recherche d'une « connections » entre m' et M. La solution optimale est toujours $\ell = 2^{n/2}$ et le coût total est $2^{2n/3}$.

Dans le modèle du coût, une fois que l'ensemble des états internes $x_0, \ldots, x_\ell$ a été calculé, on retombe exactement sur le problème du test d'appartenance étudié section 4.1. Par conséquent, le coût de la recherche du bloc de message m' qui « connecte » le vecteur d'initialisation au message ciblé est $2^n/\ell^{2/3}$. Là-dessus vient se greffer un problème amusant : la machine nécessaire est de taille au moins ℓ, or la construction de la séquence d'états internes est un processus nécessairement séquentiel. La

production des x_i coûte donc ℓ^2. Le coût total est donc $\ell^2 + 2^n/\ell^{2/3}$. Le coût minimal est $2^{3n/4}$, atteint avec $\ell = 2^{3n/8}$. L'attaque reste meilleure que la recherche exhaustive, même en « payant le prix de la mémoire ».

Le bourrage de Merkle-Damgård empêche cette attaque simple (M' n'a plus la même taille que M, donc les bourrages sont différents). L'attaque de Kelsey-Schneier [127] contourne ce problème au moyen d'un message expansif. Le coût est le même (il est dominé par la phase de « connexion » du message expansif au message cible).

Un compromis temps-mémoire utilisant des points distingués, à ce jour inédit, permettrait de faire baisser le coût.

5.2 Réseaux de Feistel

On s'intéresse ici à des attaques en récupération de clef sur des réseaux de Feistel. On considère dans un premier temps des réseaux de Feistel équilibrés dans le modèle le plus général, où les fonctions de tours sont indexées par des sous-clefs de $n/2$ bits. On considère que les sous-clefs sont indépendantes, et qu'elles sont générées à partir d'une clef maîtresse de $2n$ bits, où n désigne la taille du bloc chiffré. On s'intéresse au problème de retrouver toutes les sous-clefs, ce qui permet d'effectuer le chiffrement ou le déchiffrement légitime, même si la clef « maîtresse » dont ces dernières sont dérivées reste inconnue. Plus la clef maîtresse originale est grande, plus le temps dont dispose l'adversaire est important.

De nombreuses attaques dues à Knudsen [131], Todo [188], Guo, Jean, Nikolic et Sasaki [116, 117], etc. ne sont pas discutées ici.

5.2.1 Attaque *meet-in-the-middle* de base

En premier lieu, une attaque *meet-in-the-middle* appartenant au folklore est facile à monter sur 7 tours. Partant de quatre paires clair-chiffré, l'adversaire doit :

1. Énumérer les $2^{1.5n}$ valeurs de k_0, k_1, k_2. Pour chacune d'entre elle : effectuer le chiffrement partiel pour calculer les 4 valeurs de R_3 puis stocker l'association « $R_3 \mapsto k_i$ » dans un dictionnaire.
2. Énumérer les $2^{1.5n}$ valeurs de k_4, k_5, k_6. Pour chacune d'entre elle : effectuer le déchiffrement partiel pour calculer les 4 valeurs de R_3 puis interroger le dictionnaire. Pour chaque triplet (k_0, k_1, k_2) suggéré :
   (a) Effectuer une recherche exhaustive sur k_3 et tenter le chiffrement complet des 4 clairs connus pour tester les sous-clefs suggérées.

La complexité en espace est de $2^{1.5n}$ blocs de n bits, et la complexité en temps est également de l'ordre de $2^{1.5n}$ — en effet, l'étape 2.a n'est effectuée que 2^n fois en moyenne. Si on supposait un peu trop vite que ceci nécessite une machine permettant à chaque processeur de faire des accès aléatoires à toute la mémoire à chaque instant, alors on en conclurait (sur la base du théorème 1) que le coût est $TM^{1/3}$. Ceci signifierait que l'attaque coûterait 2^{2n} et n'améliorerait donc pas la recherche exhaustive.

Mais ce raisonnement est faux. En fait, il s'agit de trouver toutes les collisions entre deux fonctions expansives $\{0,1\}^{1.5n} \to \{0,1\}^{2n}$, puis d'effectuer un « post-traitement » (parallélisable) sur chaque collision trouvée. On a vu section 4.2.4 que trouver ces collisions coûte $2^{9n/5}$, et le post-traitement coûte $2^{3n/2}$ en tout, donc le coût total de l'attaque est $2^{9n/5}$. C'est donc meilleur que la recherche exhaustive.

5.2.2 Amélioration de Dunkelman, Keller, Dinur et Shamir [90]

Dunkelman, Keller, Dinur et Shamir [90] ont amélioré ceci avec une idée inspirée de leur technique de la « dissection ». On note $X^{(i)}$ une valeur obtenue lors du chiffrement de la i-ème paire clair-chiffré.

Il s'agit de commencer par deviner la valeur de $R_3^{(0)}$. Ceci permet de réduire de $2^{1.5n}$ à 2^n le nombre de triplets (k_0, k_1, k_2) et (k_4, k_5, k_6) qui conviennent. L'attaque fonctionne alors de la façon suivante. Pour chacune des $2^{n/2}$ valeurs possibles de $R_3^{(0)}$, faire :

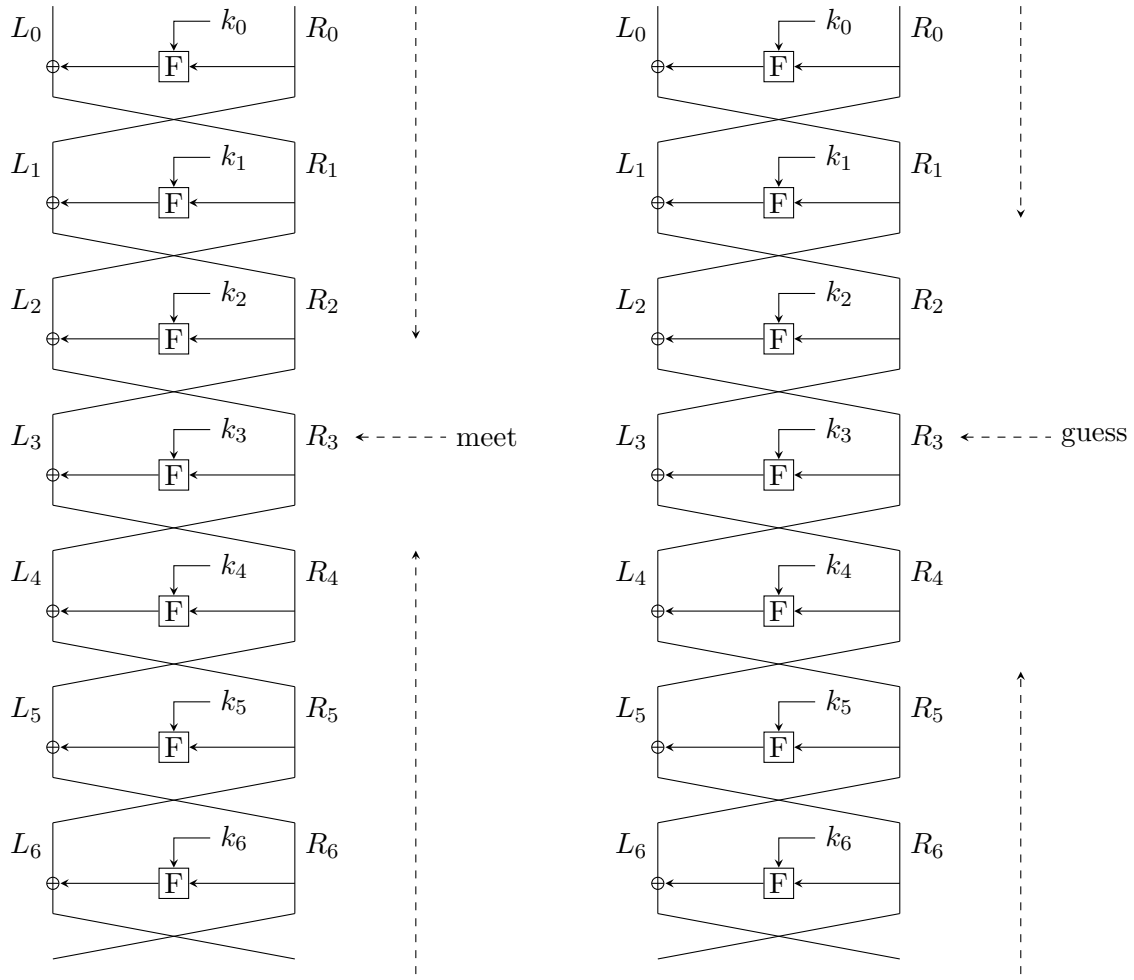


FIGURE 5.3 – Attaque directe en *meet-in-the-middle* sur 7 tours de réseau de Feistel (à gauche) et sa version améliorée par Dinur, Dunkelman, Keller et Shamir dans [90] (à droite).

1. Énumérer les 2^n valeurs de $R_2^{(0)}$ et k_2 ; pour chacune d'entre elles : calculer $L_2^{(0)}$ puis stocker l'association « $L_2^{(0)}, R_2^{(0)} \mapsto k_3$ » dans un dictionnaire $H^{\uparrow}$.
2. Énumérer les 2^n valeurs de $R_4^{(0)}$ et k_4. Pour chacune d'entre elles : calculer $L_5^{(0)}$ et $R_5^{(0)}$ puis stocker l'association « $L_5^{(0)}, R_5^{(0)} \mapsto k_4$ » dans un dictionnaire $H_{\downarrow}$.
3. Énumérer les 2^n valeurs de k_0 et k_1 ; pour chacune d'entre elles : effectuer le chiffrement partiel pour calculer $L_2^{(0)}, R_2^{(0)}$. Interroger $H^{\uparrow}$ pour obtenir éventuellement une valeur de k_2. Le cas échéant, poursuivre le chiffrement partiel pour obtenir une suggestion de R_3 pour chaque paire clair-chiffré (on obtient automatiquement la bonne valeur pour la première). Insérer l'association « $R_3 \mapsto k_0, k_1, k_2$ » dans un dictionnaire H.
4. Énumérer les 2^n valeurs de k_5, k_6. Pour chacune d'entre elle : effectuer le déchiffrement partiel pour calculer $L_5^{(0)}, R_5^{(0)}$. Interroger $H_{\downarrow}$ pour obtenir éventuellement une valeur de k_4. Le cas échéant, poursuivre le déchiffrement partiel pour calculer les 4 valeurs de R_3 puis interroger le dictionnaire H. Pour chaque triplet (k_0, k_1, k_2) suggéré :
   (a) Effectuer une recherche exhaustive sur k_3 et tenter le chiffrement complet des 4 clairs connus pour tester les sous-clefs suggérées.

La complexité en mémoire est réduite à 2^n et la complexité en temps est inchangée à $2^{1.5n}$ — l'étape 4.*a* est effectuée $2^{n/2}$ fois par itération de la boucle externe. Ceci améliore donc strictement l'attaque directe décrite ci-dessus.

Du point de vue du coût, chaque itération de la boucle externe fonctionne en temps et en espace 2^n. Admettons un instant que cela nécessite une machine où chaque processeur doit être capable d'effectuer

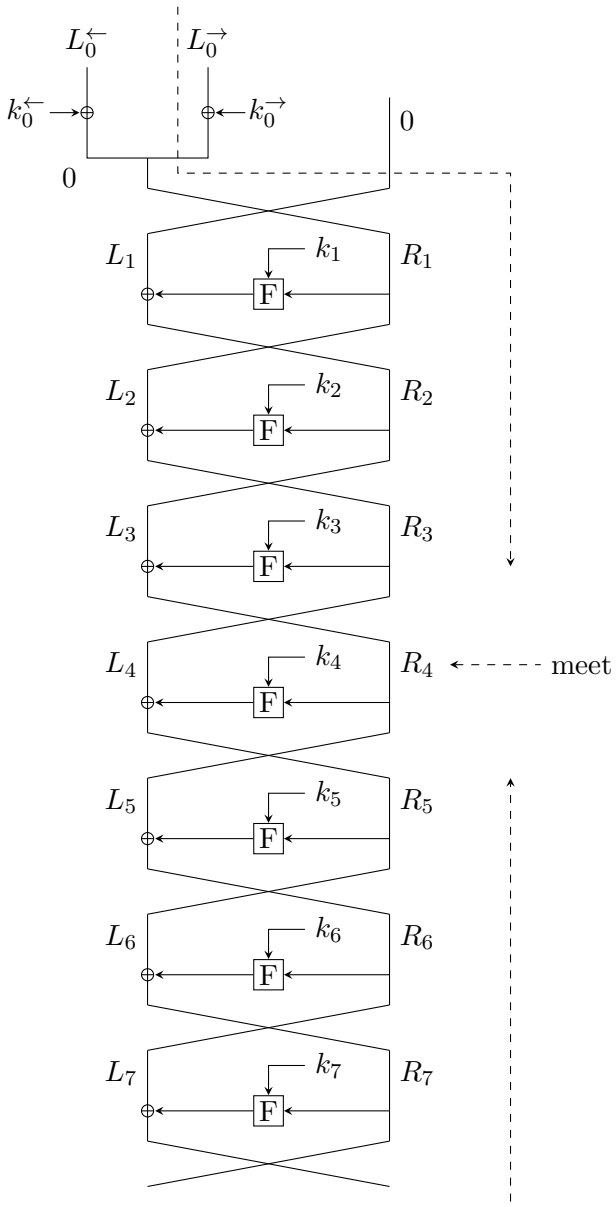


FIGURE 5.4 – Amélioration d'Isobe et Shibutani sur 8 tours de réseau de Feistel.

des accès aléatoires à une mémoire de taille 2^n ; sous cette hypothèse, le théorème 1 implique que l'attaque coûte $2^{11n/6}$, c'est-à-dire *davantage* que l'attaque de base.

Peut-on implanter cette attaque avec un coût réduit ? Une itération de la boucle interne cherche toutes les collisions entre deux fonctions qui prennent en argument (k_0, k_1) et (k_5, k_6), respectivement. Mais ces deux fonctions ont maintenant une description de taille 2^n, matérialisée par les deux dictionnaires $H^{\uparrow}$ et $H_{\downarrow}$. La discussion de la section 4.2.3 (cas des fonctions qui ont une grosse description) montre qu'on ne sait pas faire mieux que $2^{11n/6}$.

Ceci illustre que raisonner du point de vue du coût recèle des surprises, et qu'on ne peut pas anticiper la situation en se basant uniquement sur le nombre d'opérations et la taille de la mémoire nécessaire. L'attaque améliorée nécessite le même temps et *moins* de mémoire en tout que l'attaque de base, mais pourtant elle coûte *plus* cher !

5.2.3 Attaques d'Isobe et Shibutani [121]

Isobe et Shibutani [121] décrivent une technique qui permet d'attaquer un tour supplémentaire (cf. figure 5.4). Ceci repose sur deux idées :

— On choisit des messages clairs avec $R_0 = 0$ (ou n'importe quelle autre valeur fixée). Du coup, le premier tour revient à XORer une clef de blanchiment sur L_0.
— On utilise la technique du « *splice-and-cut* » d'Aoki et Sasaki [9] pour découper cette clef en deux et inclure chacune des deux moitiés de $n/4$ bits dans les deux côtés du *meet-in-the-middle*. Pour cela, le premier tour est « recollé » au dernier : on impose que les $n/4$ bits de gauche de L_0 valent zéro, on part d'une valeur de $k_0^{\leftarrow}$, on forme le clair correspondant, on récupère le chiffré associé et on poursuit le déchiffrement partiel.

Plus précisément, l'attaque —qui n'est tout simplement pas décrite dans [121]— fonctionne de la façon suivante :

1. Obtenir le chiffré des $5 \cdot 2^{n/4}$ messages de la forme $L_0 = i \| P_j$ et $R_0 = 0$, où i prend toutes les valeurs possibles dans $\{0,1\}^{n/4}$ tandis que les P_j (pour $0 \leq j < 5$) sont arbitraires.
2. Énumérer les $2^{1.75n}$ valeurs de $k_0^{\rightarrow}, k_1, k_2, k_3$; pour chacune d'entre elle : effectuer le chiffrement partiel de $L_1 = 0$ et $R_1^{(j)} = 0 \| P_j \oplus k_0^{\rightarrow}$ pour calculer les 5 valeurs de R_4. Stocker l'association « $R_4 \mapsto k_0^{\rightarrow}, k_1, k_2, k_3$ » dans un dictionnaire H.
3. Énumérer les $2^{1.75n}$ valeurs de $k_0^{\leftarrow}, k_4, k_5, k_6$; pour chacune d'entre elles : récupérer les cinq chiffrés $C^{(j)}$ des messages clairs avec $L_0^{(j)} = k_0^{\leftarrow} \| P_j$. Effectuer le déchiffrement partiel pour obtenir les 5 valeurs de R_4 puis interroger le dictionnaire H. Pour chaque triplet $(k_0^{\rightarrow}, k_1, k_2, k_3)$ suggéré :
   (a) Effectuer une recherche exhaustive sur k_4 et tenter le chiffrement complet de 4 clairs connus pour tester les sous-clefs suggérées.

La complexité en temps et en espace est $2^{1.75n}$. Essayons de déterminer le coût. Comparé à la version à 7 tours de la section 5.2.1, la différence est que bien qu'on cherche toujours toutes les collisions entre deux fonctions expansives sur $1.75n$ bits, mais l'une de ces deux fonctions possède maintenant une description de taille exponentielle (l'accès aux $2^{n/4}$ chiffrés est nécessaire). On peut donc reprendre le raisonnement de la section 4.2.3 avec $\alpha = 1/7$, ce qui donne un coût de $2^{32n/15}$, qui est maintenant plus élevé que celui de la recherche exhaustive !

Au passage, même si on oubliait le fait que la fonction est de taille exponentielle, la recherche de toutes les collisions sur des fonctions de $1.75n$ bits coûte $2^{21n/10}$, ce qui est déjà plus grand que 2^{2n}...

5.3 Chiffrement d'Even-Mansour à deux tours et une seule clef

Le Chiffrement d'Even-Mansour à deux tours et une seule clef transforme deux permutations aléatoires P_1, P_2 sur n bits en un système de chiffrement par bloc sur n bits avec des clefs k de n bits aussi :

$$E(x) = P_2(P_1(x \oplus k) \oplus k) \oplus k$$

Une série de travaux par Nikolic, Wang, et Wu [165], Dinur, Dunkelman, Keller et Shamir [91], Isobe et Shibutani [122], ainsi que Leurent et Sibleyras [149] présentent des attaquent en récupération de clef sur cette construction. Toutes ces attaques effectuent au moins $2^n/n$ opérations et ont comme objectif de minimiser le nombre de requêtes effectuées à l'oracle de chiffrement. Ceci est loin de la meilleure borne inférieure prouvée, qui est de $2^{2n/3}$ requêtes aux oracles [35].

Les caractéristiques des attaques connues sont résumées par le tableau 5.5. Elles utilisent toutes exponentiellement plus de n unités de mémoire, donc il est très vraisemblable qu'elles coûtent toutes plus cher que la recherche exhaustive. Il faut noter que ces attaques ressemblent aux algorithmes pour le problème 3XOR : celle de Nikolic, Wang et Wu [165] ressemble à l'algorithme de Nikolic et Sasaki [164] ; celle de Dinur, Dunkelman et Shamir [91] ressemble à l'algorithme de Joux [125]. Ceci n'est pas un hasard : Leurent et Sibleyras [149] ont démontré qu'on peut retrouver la clef en résolvant une instance assez générique du problème 3XOR.

Ref.	Données		Temps	Mémoire
trivial	1	KP	2^n	1
[165]	$2^n \ln n/n$	KP	$2^n \ln n/n$	$2^n \ln n/n$
[91]	$2^n/\lambda n$	CP	$2^n/\lambda n$	$2^{\lambda n}$
[122]	$2^{\lambda n}$	CP	$2^n \ln n/n$	$2^n \ln n/n$
[149]	λn	KP	$2^n/\lambda n$	$2^{\lambda n}$

FIGURE 5.5 – Caractéristiques des attaques publiées contre le chiffrement d'Even-Mansour à deux tours et une seule clef. Le paramètre $0 < \lambda < 1$ peut être choisi librement. KP : Known plaintext ; CP : Chosen plaintext.

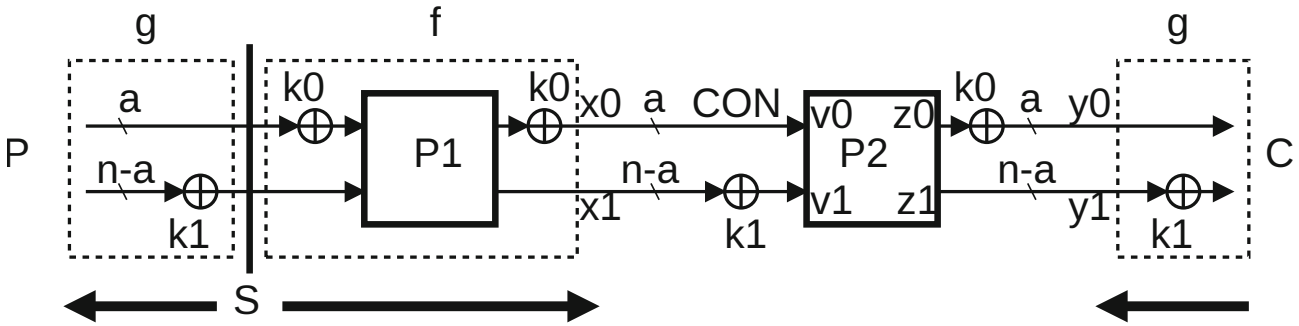


FIGURE 5.6 – Représentation schématique de l'attaque d'Isobe et Shibutani sue le chiffrement d'Even-Mansour à deux tours et une seule clef. Image : [122].

5.3.1 Attaque d'Isobe et Shibutani [122]

> We do not claim that our time-optimized attacks sufficiently reduce the time complexity required for the previously known key recovery attacks
>
> T. Isobe et K. Shibutani, 2017, [122]

L'article « *New Key Recovery Attacks on Minimal Two-Round Even-Mansour Ciphers* » d'Isobe et Shibutani [122] décrit une « attaque de base » dont les complexités en temps et en espace sont toutes les deux égales à 2^{n-a}, sous la contrainte que $a2^a \leq n$. Le plus grand choix valide de a est inférieur à $\log_2 n$.

Pour $n = 256$, l'article donne des complexités « concrètes » : avec $a = 5$, 2^{251} accès à une mémoire de taille 2^{251} et 2^{251} requêtes à l'oracle de chiffrement. Le même article continue avec une « attaque optimisée en temps », qui, en réalité, vise à minimiser le nombre d'évaluations « privées » des permutations. Pour $n = 256$, la complexité de cette attaque améliorée est décrite par « 2^{208} évaluations de la permutation, 2^{254} accès à une mémoire de taille 2^{205} et 2^{254} accès à l'oracle de chiffrement ». Nous en concluons que l'attaque « optimisée en temps » effectue *plus* d'accès à une mémoire (de taille réduite) et *plus* de requêtes à l'oracle de chiffrement que l'attaque « de base ». Les auteurs se justifient :

> Basically, it is very hard to compare the cost of encryption and memory access because it strongly depend (*sic*) on the execution environments, the size of table and the underlying permutation.

Puisque tout est dans tout, et réciproquement, alors tout va bien !

L'attaque réutilise la technique du *splice-and-cut* dans un *Meet-in-the-Middle* décrit par la figure 5.6. La fonction f, qui englobe P_1 dépend d'une partie de la clef (k_0) tandis que la fonction g qui englobe le recollement entre le clair et le chiffré (par l'oracle de chiffrement) dépend de k_1.

Dans un précalcul, identifier une entrée S de f qui laisse constants (à une valeur `CON`) les a bits de la

sortie notés x_0 quelle que soit la clef k_0. Enumérer toutes les $k_0 \in \{0,1\}^a$ et stocker $k_0 \mapsto f(S)$ dans un (petit) dictionnaire H. Énumérer toutes les entrées possibles de P_2 dont les a premiers bits (v_0 sur la figure) ont la valeur `CON`, puis stocker $z_1 \mapsto v_1$ dans un autre (gros) dictionnaire D.

Pour la phase « en ligne » de l'attaque, énumérer toutes les 2^{n-a} valeurs de k_1, invoquer l'oracle de chiffrement pour évaluer g^{-1}, tester si la valeur de y_1 obtenue est dans le gros dictionnaire D. Si oui, cela suggère une valeur de v_1, donc de x_1. Tester si elle appartient au petit dictionnaire H. Le cas échéant, cela suggère une clef potentielle.

On a donc une procédure qui effectue 2^{n-a} recherche dans un dictionnaire de taille 2^{n-a}. En vertu de ce qui a été discuté section 4.1, ceci coûte $2^{\frac{4}{3}(n-a)}$, donc davantage que la recherche exhaustive vu la contrainte qui pèse sur a.

Il en va de même d'une attaque assez similaire de Nikolic, Wang et Wu [165].

5.3.2 Attaque de Dinur, Dunkelman, Keller et Shamir [91]

L'article (antérieur) « *Key Recovery Attacks on Iterated Even–Mansour Encryption Schemes* » [91] de Dinur, Dunkelman, Keller et Shamir ne tombe pas dans ces travers-là. L'attaque qu'il décrit répète $D = 2^{(1-\lambda)n}/\lambda n$ fois une procédure qui consiste à :

1. Évaluer $S = 2^{\lambda n}$ fois une des permutations publiques ; effectuer un produit matrice-vecteur sur n bits ; stocker le résultat dans une liste.
2. Trier la liste de S blocs de n bits ainsi obtenus.
3. Évaluer S fois la construction complète sur des clairs choisis ; effectuer un produit matrice-vecteur sur n bits ; chercher si la valeur est présente dans la liste (par recherche dichotomique) ; si oui ceci suggère une valeur possible de la clef.

Effectués naïvement avec des algorithmes en $\mathcal{O}(N \log N)$, les tris sembleraient nécessiter 2^n opérations, et les recherches dichotomiques dans le tableau trié aussi, donc l'attaque nécessiterait le même nombre d'opérations que la recherche exhaustive. Les auteurs commentent :

> A delicate issue which we face in the analysis of our attacks is the complexity of sorting. As most papers in the field, our analysis assumes that this complexity is negligible compared to the complexity of cipher evaluations performed by our attacks.

L'article explique que réaliser l'un de ces tris nécessite $nS \log S$ opérations sur des bits, ce qui donne donc, vu leur taille $n^2 2^{\lambda n}$ opérations. Comme ceci est répété D fois, on effectue en tout $n2^n$ opérations de bits pour trier. Les auteurs s'en sortent en affirmant (sans justification) qu'évaluer une permutation sur n bits de qualité cryptographique nécessiterait $\Omega\left(n^2\right)$ opérations sur des bits. Par conséquent, la recherche exhaustive, qui nécessite 2^n évaluations du dispositif de chiffrement, nécessiterait $\Omega\left(n^2 2^n\right)$ opérations sur des bits, et serait donc plus lente.

Ce raisonnement mériterait d'être évalué. Miles et Viola ont montré en 2015 [160] qu'une généralisation de l'AES à des blocs de n bits peut être évaluée en $\tilde{\mathcal{O}}(n)$ opérations sur des bits, tout en offrant une résistance exponentielle aux cryptanalyses linéaires et différentielles. Ceci semble contredire, au moins en partie, le raisonnement qui « prouve » que l'attaque nécessite moins d'opérations que la recherche exhaustive. Mais bon, si on ne peut plus introduire les hypothèses qui nous arrangent pour justifier que notre travail puisse être publié au *Journal of Cryptology*, où va-t-on ! Notons que toutes ces contorsions auraient pu être évitées en utilisant un dictionnaire statique à la Fredman et al. [109] (cf. section 4.1). Dans le fond, l'attaque est bel et bien valide dans le modèle de la *Random Access Machine*.

Pour ce qui est du coût total de l'attaque, on commence par noter qu'il suffit d'évaluer le coût d'une itération de la procédure ci-dessus. Celle-ci effectue S accès à une mémoire de taille S, donc on pourrait commencer par imaginer, sur la base du théorème 1, qu'elle va coûter $S^{4/3}$. Mais en fait, si on y regarde à deux fois, on trouve qu'il s'agit de calculer une jointure entre deux fonctions (la première est décrite par l'étape 1, la seconde par l'étape 3). On pourrait donc partir sur l'hypothèse que cela coûte $S^{6/5}$.

Mais en plus cela soulève un autre problème : évaluer la fonction de l'étape 3 nécessite de faire appel à l'oracle de chiffrement. Dans une machine parallèle, comment est-ce que ça se passe ? Tous

les processeurs sont-ils reliés, par un moyen magique, à l'oracle (et quelle longueur de cable est-elle nécessaire ?) ? Peuvent-ils l'évaluer en parallèle à un rythme non-borné ?

De toute façon, même si ces problèmes étaient résolus dans un sens favorable à l'adversaire, le coût de l'attaque est $DS^{6/5} = 2^{n(1+\lambda/5)}$. Elle est donc plus coûteuse que la recherche exhaustive.

5.3.3 Attaque de Leurent et Sibleyras [149]

Leurent et Sibleyras ont démontré en 2019 qu'attaquer la construction d'Even-Mansour à deux tours et une seule clef se ramène à la résolution d'une instance du problème 3XOR. En effet, on a :

$$E(x) = P_2(P_1(x \oplus k) \oplus k) \oplus k$$

Notons $y \leftarrow x \oplus k$ et $z \leftarrow P_1(y) \oplus k$. On a alors

$$k = x \oplus y = P_1(y) \oplus z = P_2(z) \oplus E(x)$$

Et donc x, y et z satisfont :

$$\begin{pmatrix} x \\ x \oplus E(x) \end{pmatrix} \oplus \begin{pmatrix} y \oplus P_1(y) \\ y \end{pmatrix} \oplus \begin{pmatrix} z \\ P_2(z) \end{pmatrix} = 0$$

Une solution possible consiste donc à assembler ces trois listes de chaînes de $2n$ bits, puis trouver un triplet x, y, z valide : ceci révèle k. La première attaque décrite dans [149] consiste à appliquer directement l'algorithme de calcul de 3XOR de Delaplace, Fouque et moi-même [44]. La deuxième attaque décrite dans [149] consiste à appliquer directement l'algorithme pour le problème 3SUM de Baran, Demaine et Pătraşcu qui a été adapté pour le problème 3XOR par Delaplace, Fouque et moi-même [44].

La troisième attaque décrite par les auteurs, nommée **LD**, est la meilleure. Elle utilise l'algorithme de Joux pour le problème 3XOR de manière répétée, avec du *clamping*. Ceci est possible car l'attaquant peut construire librement les trois listes de l'instance du problème 3XOR. L'idée générale consiste à obtenir λn paires clair-chiffré (donc des valeurs de x et $E(x)$), qui forment la première liste ; choisir une matrice M de taille $2n \times 2n$, inversible, qui annule les $(2-\lambda)n$ bits de poids forts de la première liste ; puis répéter $2^{(1-\lambda)n}/(\lambda n)$ fois la procédure suivante :

- [*Clamping*] Fixer les $(1-\lambda)n$ bits de poids forts de $y.M$ et $P_2(z).M$ à une valeur aléatoire commune.
- [*Assemblage*] Former la deuxième et la troisième liste en énumérant toutes les valeurs possibles des λn bits de poids faibles de $y.M$ et $P_2(z).M$, respectivement.
- [*3XOR*] Calculer une jointure entre la deuxième et la troisième liste, sur les n premiers bits ; tester chaque paire issue de la jointure vis-à-vis de la première liste.

(certains détails ont été laissés de côté par souci de simplification, mais l'idée générale est là).

Le nombre moyen de triplets 3XOR trouvés lors d'une itération est $\lambda n 2^{\lambda n - n}$. Par conséquent, l'espérance du nombre total de triplets trouvés est de 1.

Le gros du calcul consiste à effectuer une jointure (sur n bits) entre deux listes de $2^{\lambda n}$ chaînes de $2n$ bits. Si les listes doivent être stockées, alors le coût du calcul des jointures est $N^{4/3}$. Par contre, dans ce cas précis, il s'agit d'une « jointure entre fonctions » qu'on peut en fait calculer sans stocker les listes, avec un coût de $N^{6/5}$ (cf. discussion de la section 4.3). Par conséquent, le coût de cet algorithme est environ $2^{(1+\lambda/5)n}/(\lambda n)$, ce qui est toujours plus grand que 2^n.

5.3.4 Expériences

À ma connaissance, aucune de ces attaque n'a été implantée. Les articles [91, 149], dont la consommation mémoire est plus faible, discutent du cas « concret » $n = 64$, et donnent des quantités précises pour le « temps » et la « mémoire ». Il serait intéressant de prendre une permutation publique sur 64

bits, puis de comparer, avec du code concret, le temps d'exécution de ces attaques avec celui de 2^{65} évaluations de la permutation. Monter cette expérience prend quelques minutes, mais les auteurs de ces attaques n'ont pas jugé utile de le faire.

Dans le cas de [91], les auteurs discutent explicitement du cas $n = 64$, et ils proposent $\lambda = 1/4$. La complexité en mémoire est alors très faible et l'attaque serait implantable en pratique, au moins à des fins de chronométrage. En effet, la question de l'efficacité de l'attaque se ramène concrètement au problème de savoir si trier 2^{16} entiers de 64 bits puis effectuer 2^{16} recherches dichotomiques dans le résultat est plus rapide que d'évaluer la permutation 2^{21} fois.

Ceci est facile à vérifier : 2^{21} évaluations de Speck64/128 nécessitent 63ms sur un cœur d'un laptop moderne. Le tri de 2^{16} entiers 64 bits, les produits matrice-vecteurs, les recherches dichotomiques dans le tableau trié prennent 13ms. L'attaque est donc visiblement 5 fois plus rapide que la recherche exhaustive. Avec une table de hachage à sondage linéaire, le code est sensiblement plus simple et l'attaque est 9.7 fois plus rapide. Conclusion : mieux vaut étudier soigneusement le tome 3 (*Searching and Sorting*) de *The Art Of Computer Programming* pour implanter des attaques cryptographiques.

Quant à l'article [149], son abstract affirme crânement :

> For instance, with $n = 64$ and $\lambda = 1/2$, the memory requirement is practical, and we gain a factor 32 over brute-force search.

Cette affirmation serait exacte si réaliser une jointure sur 64 bits entre deux listes de 2^{32} entrées de 128 bits était plus rapide que d'évaluer la permutation 2^{33} fois. Pour améliorer la recherche exhaustive, il faudrait que ce soit plus rapide que 2^{38} évaluations de la permutation.

Ces deux listes occupent donc 128Go, donc un laptop ne suffit plus à vérifier la véracité de cette affirmation. Sur un nœud du cluster `grvingt` de Grid'5000 (qui a 192Go de RAM), évaluer Speck64/128 2^{38} fois nécessite 307 secondes (sans faire d'efforts, avec l'implantation en C de référence). Implanter assez naïvement un équivalent de l'attaque de [149] (produit matrice-vecteur optimisé et jointure réalisée en triant les deux listes avec une routine de tri parallèle sur étagère) nécessite moins de 90s. L'attaque est donc 3.5× fois plus rapide que la recherche exhaustive, et non pas 32 fois. Mais ça n'est déjà pas si mal !

L'implantation de l'attaque pourrait sûrement être améliorée. Mais le calcul efficace des jointures est un art délicat et cela nécessiterait des semaines de travail. En outre, il y aurait le risque que tous ces efforts soient réduits à néant, car la recherche exhaustive pourrait elle aussi être améliorée. Avec de la vectorisation notamment, on pourrait espérer l'accélérer d'un facteur compris entre 8 et 16 sur des processeurs modernes (avec AVX2 ou AVX512).

5.3.5 Discussion

Toutes ces attaques coûtent (asymptotiquement) plus cher que la recherche exhaustive sur n bits. Elles ont toutes une complexité en temps de l'ordre de $2^n/n$ et une complexité en espace exponentielle en n. Si le « prix de la mémoire » n'est pas nul, aucune de ces attaques ne peut battre la force brute en terme de coût. Autrement dit : ces attaques n'améliorent asymptotiquement la recherche exhaustive *que* dans un modèle de calcul qui ignore complètement le coût de la mémoire et des communications.

Pourtant, on vient de voir que les attaques dont la consommation en mémoire est modulable peuvent être plus rapides que la recherche exhaustive, bien qu'elles soient plus coûteuse !

Il n'y a pas de paradoxe, et on a déjà observé ce phénomène, par exemple pour les algorithmes de calcul de 3XOR section 4.4. Commençons par noter que le matériel nécessaire à l'exécution des attaques est « gros » et plus coûteux que celui qui est nécessaire à la recherche exhaustive. Tant que la quantité de mémoire nécessaire à l'exécution des attaques est inférieure à celle de la machine dont on dispose, alors le nombre d'opérations est une mesure raisonnable de ce qui devrait se passer.

Mais si on ne disposait que d'une mémoire très réduite, par exemple si on voulait implanter tout ceci sur un FPGA, la situation serait très différente. L'avantage des attaques sophistiquées par rapport

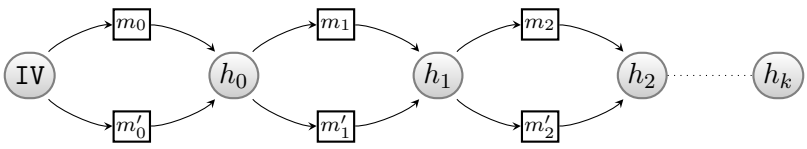


FIGURE 5.7 – Multicollision sur une fonction de hachage itérée. Les nœuds ronds désignent les états internes de la fonction de hachage, tandis que les rectangles désignent des blocs de message (ou des séquences de tels blocs).

à la recherche exhaustive fondrait comme neige au soleil : qui dit mémoire réduite, dit valeur de λ faible, donc gain réduit. Le fond de l'affaire est que leur coût est supérieur.

5.4 Combinateurs de fonctions de hachage

L'idée de combiner plusieurs fonctions de hachage en une seule meilleure fonction de hachage n'est pas nouvelle (elle est décrite dans la thèse de Preneel en 1993). L'intention sous-jacente est que même si l'une des fonctions de hachage est cassée, alors la combinaison pourrait continuer à être sûre. Le cas le plus simple consiste à combiner deux fonctions qui produisent des empreintes de n bits. Les deux combinateurs les plus simples sont le XOR et la concaténation :

$$H_{\oplus}(M) = H_1(M) \oplus H_2(M) \qquad \text{et} \qquad H_{\|}(M) = H_1(M) \| H_2(M)$$

TLS 1.0 (1999, maintenant déprécié) utilisait la concaténation de MD5 et SHA-1 pour hacher les données à signer. Pour authentifier les échanges, une fois qu'une clef de session a pu être établie, le même standard utilisait le XOR de HMAC-MD5 et HMAC-SHA-1. Tout ce bricolage a été remplacé par l'usage de SHA-2 dans TLS 1.2 (2008).

5.4.1 Attaques de Joux sur la concatenation [124]

En 2004, Joux a montré [124] que la concaténation d'une fonction de type Merkle-Damgård avec n'importe quelle autre, ce qui donne des empreintes de $2n$ bits, n'offre pas plus de résistance aux collisions qu'une fonction de hachage idéale sur n bits. La concaténation ne permet donc pas d'améliorer génériquement la résistance aux collisions.

L'attaque repose sur le fait qu'on peut trouver des 2^k-*multicollisions* sur les fonctions de hachage itérées. L'idée est illustrée par la figure 5.7 : il s'agit de trouver deux blocs de message (m_0, m'_0) qui envoient l'IV sur un nouvel état interne h_0, puis deux autres qui envoient h_0 sur h_1, etc. Au final, il y a 2^k chemins différents dans le graphe qui aboutissent tous au même état interne de la fonction. On a donc une représentation compacte d'un ensemble de 2^k messages qui collisionnent. Construire une 2^k-multicollision nécessite génériquement $k2^{n/2}$ évaluations de la fonction de compression, et peut se faire sans mémoire. Le coût est donc équivalent au nombre d'opérations.

Pour obtenir une collision sur la concaténation de H_1 et H_2, où H_1 est de Merkle-Damgård, on calcule une $2^{n/2}$ multicollision sur H_1, puis on évalue H_2 sur les $2^{n/2}$ messages qui fixent H_1. On s'attend à en trouver deux qui donnent aussi une collision sur H_2. Ce processus peut se faire sans mémoire lui aussi.

La complexité totale de l'attaque est de l'ordre de $n2^{n/2}$ évaluations de H_1 et H_2. Et, ce qui est remarquable, c'est que le coût est le même.

Joux décrit également une attaque simple en préimage : calculer une 2^n-multicollision sur H_1 qui donne un état interne (x_1, x_2) ; tester 2^n blocs de messages m' aléatoires jusqu'à ce que $h_1(x, m')$ donne la bonne sortie pour H_1. Ensuite, tester les 2^n chemins possibles dans la multicollision jusqu'à ce qu'en ajoutant m' à la fin, on obtienne la bonne sortie pour H_2 aussi. Cette attaque coûte 2^n évaluations des fonctions de compression, et peut elle aussi se réaliser sans mémoire, donc avec le même coût.

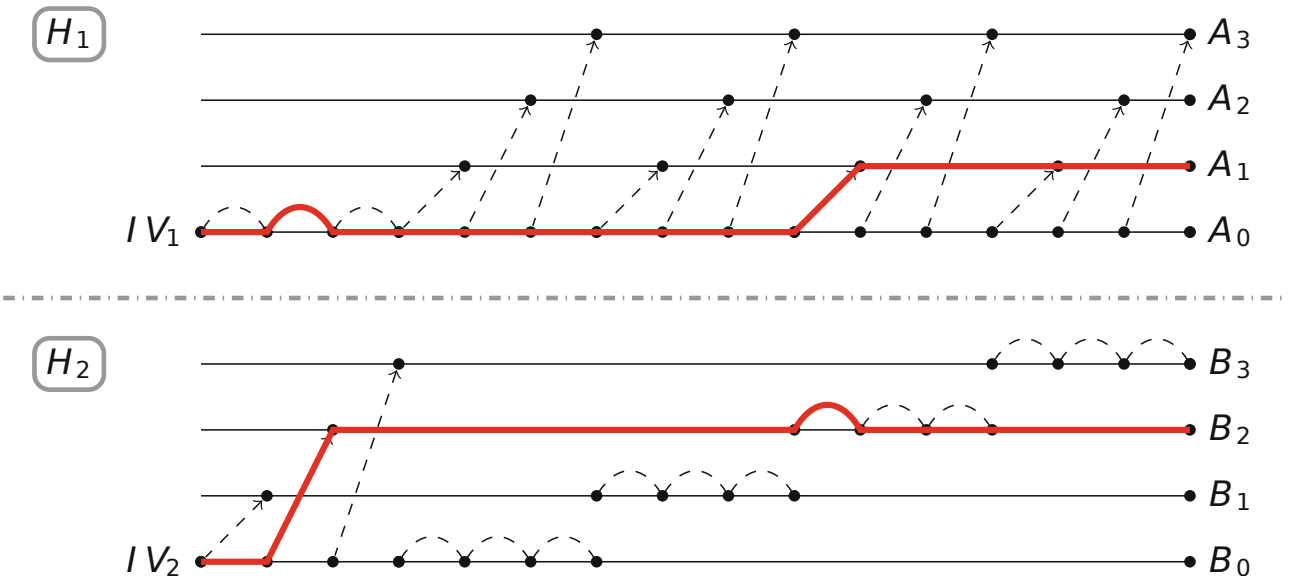


FIGURE 5.8 – « *Interchange Structure* » utilisée dans l'attaque de Leurent et Wang [150]. Avec N^2 collisions, on peut atteindre chacune des N^2 paires d'états internes (A_i, B_j) avec H_1 et H_2. Image : [150].

5.4.2 Attaque en préimage de Leurent et Wang sur le XOR [150]

Dans leur article au très bon titre « *The Sum Can Be Weaker Than Each Part* » [150], Leurent et Wang présentent une attaque en préimage sur le XOR de deux fonctions de hachage itérées de type Merkle-Damgård, qui nécessite $\tilde{\mathcal{O}}\left(2^{5n/6}\right)$ opérations et $2^{n/3}$ unités de mémoire.

L'attaque repose sur l'utilisation de l'« *interchange structure* » représentée par la figure 5.8. Après avoir trouvé 2^{2t} collisions, on peut produire une structure de données de taille 2^{2t} ainsi que deux séquences (A_i) et (B_j) de 2^t états internes de H_1 et H_2 respectivement. Ensuite, étant donné i et j, on peut produire (en temps linéaire) un message M_{ij} de taille 2^{2t} qui envoie les états initiaux des deux fonctions de hachage sur A_i et B_j simultanément.

La phase finale de l'attaque reçoit une empreinte cible $\overline{h}$ et répète 2^{n-2t} fois la procédure suivante : choisir un bloc de message aléatoire m ; chercher (i, j) qui réalisent une collision $h_1(A_i, m) = \overline{h} \oplus h_2(B_j, m)$; le cas échéant, le message $M_{ij} \| m$, qui occupe $\mathcal{O}\left(2^{2t}\right)$ blocs, est bien une préimage de $\overline{h}$.

Le précalcul et la phase finale nécessitent respectivement $2^{2t+n/2}$ et 2^{n-t} évaluations des fonctions de compression. Pour la phase finale, Leurent et Wang proposent de trouver la collision en construisant deux listes de taille 2^t (avec $h_1(A_i, m)$ et $\overline{h} \oplus h_2(B_j, m)$ respectivement) puis en testant si leur intersection est non-vide. Ils ont choisi $t = n/6$, ce qui équilibre les nombres d'opérations du précalcul et de la phase finale à $2^{5n/6}$.

L'espace nécessaire au stockage de la structure de données initiale est 2^{2t}. Réfléchir au coût de cette attaque pose plusieurs problèmes intéressants, car 1) le précalcul a une dimension intrinsèquement séquentielle, et 2) le gros du calcul n'accède qu'à une petite partie de la mémoire nécessaire. On est donc loin du domaine où on peut facilement se convaincre que le théorème 1 épuise la question.

Si on suppose qu'on peut émettre la préimage en « *streaming* », sans avoir besoin de la stocker [2], alors on peut réduire la consommation mémoire en faisant le précalcul deux fois :

— Une première fois pour produire les états internes accessibles (A_i) et (B_j).

2. Dans la théorie de la complexité, il est admis qu'une machine qui fonctionne en espace M peut produire une sortie de taille supérieure à M. Dans notre cas, on pourrait imaginer qu'un FPGA doté d'une toute petite mémoire fasse un calcul et produise « en *streaming* » un résultat plus gros que sa mémoire.

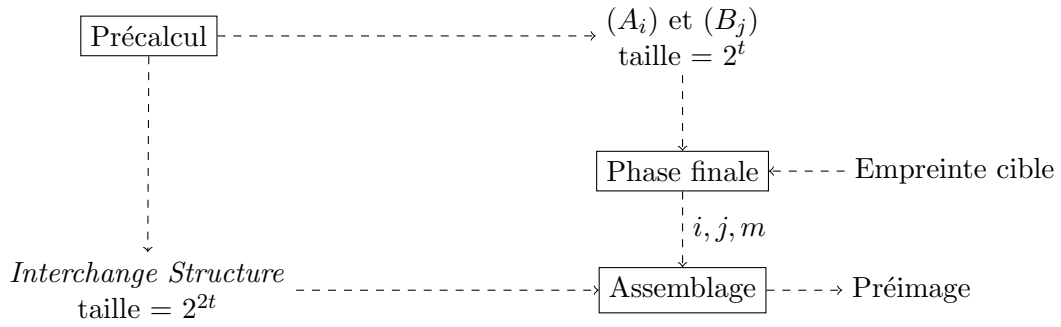


FIGURE 5.9 – Représentation schématique et flot de données dans l'attaque de Leurent et Wang [150].

— Une deuxième fois pour produire l'« *interchange structure* », qui n'a pas besoin d'être stockée mais peut être consommée au fur-et-à-mesure par la production de la préimage.

Ceci permet de réduire la consommation mémoire de 2^{2t} à 2^t. En effet, le précalcul peut s'effectuer avec une mémoire de taille 2^t : il suffit de stocker les 2×2^t états internes qui apparaissent figure 5.8, puis de chercher les collisions sans mémoire.

Cependant, le précalcul a une dimension intrinsèquement séquentielle, donc il faut faire attention avant d'appliquer directement les résultats de la section 4.2 pour estimer son coût, car celui-ci suppose un grand degré de parallélisation. Il s'agit de trouver 2^{2t} collisions, or on ne peut commencer à chercher la $(i+1)$-ème qu'après avoir trouvé la i-ème, mais la taille de la mémoire est imposée, c'est 2^t. Le raisonnement de la section 4.2.2 montre qu'on peut s'en tirer avec un coût optimal, en utilisant des processeurs qui ont une mémoire constante, tant que $t \leq 3/8$. Sous cette condition, le coût du précalcul est donc bien $2^{2t+n/2}$, sur une machine de taille 2^t.

La phase de « meet-in-the-middle » décrite par les auteurs effectue 2^{n-t} accès (« intensifs ») à une mémoire de taille 2^t, donc elle coûte $2^{n-2t/3}$.

Avec le choix de $t = 1/6$, le coût de l'attaque est $2^{8n/9}$. Elle est donc déjà, sous sa forme « originale », moins coûteuse que la recherche exhaustive. Cependant, ceci est améliorable car le coût de la phase finale domine celui du précalcul. Le choix $t = 3n/16$ fait baisser le coût total à $2^{7n/8}$. Ce nouveau choix de t déséquilibre les nombres d'opérations des deux phases et augmente le nombre d'opérations total à $2^{7n/8}$.

Cependant, la phase finale est, dans le fond, également une recherche de collision, donc on pourrait elle aussi la faire avec des algorithmes moins naïfs pour réduire son coût. Pour chaque choix du bloc de message aléatoire m, on cherche une collision entre deux fonctions $f_1, f_2 : \{0,1\}^t \to \{0,1\}^n$, avec $t < n/2$. On a vu section 4.2.4 que ceci peut se faire avec un coût de $2^{6t/5}$. Alors, poser $t = 5n/28$ aboutit à un coût total de $2^{6n/7}$.

Pour résumer, supposons qu'on dispose d'une machine possédant $p = 2^{5n/28}$ processeurs, chacun possédant une mémoire de taille constante. On pose $t = 5/28$. Le précalcul effectue $2^{5n/14}$ recherches de collisions qui nécessitent chacune $2^{9n/28}$ unités de temps sur la machine parallèle. Le précalcul nécessite donc $2^{19n/28}$ unités de temps en tout.

La phase finale, elle, effectue des lots de $2^{n/14}$ recherches de collision en parallèle, en affectant $2^{3n/28}$ processeurs à chacune. Chaque lot nécessite $2^{3n/28}$ unités de temps, et il y a $2^{4n/7}$ lots. Le temps total est donc également $2^{19n/28}$.

Au passage, tout ceci montre que la complexité en mémoire de l'attaque peut être réduite, à condition qu'on s'autorise à *streamer* la préimage à la fin.

5.4.3 Attaque en seconde préimage de Dinur sur la concaténation [86]

Dinur a proposé peu après une amélioration de ces résultats, avec une attaque en préimage sur le XOR qui nécessite $2^{2n/3}$ opérations, et surtout une attaque en seconde préimage sur la concaténation en $2^{3n/4}$ opérations. C'est de cette dernière dont il est question ici.

Le hachage d'un message M donne lieu à une séquence de paires d'états internes $(a_0, b_0), (a_1, b_1), ...$ L'attaque se décompose en trois phases, elles-mêmes constituées de plusieurs étapes :

A. Construire un message expansif [127] qui permet d'atteindre un état unique $(\hat{x}, \hat{y})$.

B. Construire une paire d'états spéciaux $(\overline{x}, \overline{y})$ et la « connecter » à un couple (a_i, b_i) de valeurs de chaînage obtenues en hachant M.

C. Connecter l'état interne de sortie du message expansif $(\hat{x}, \hat{y})$ à l'état spécial $(\overline{x}, \overline{y})$.

La seconde préimage est obtenue en remplaçant le début de M (jusqu'à l'indice i) par le message expansif, le bout de message qui le « connecte » à l'état spécial, puis le bout de message qui connecte ce dernier au message M. Seules les deux dernières phases sont critiques, donc on ignore ici la première. La complexité dépend du nombre de blocs du message M dont on cherche une seconde préimage, qui est noté 2^ℓ.

L'état spécial $(\overline{x}, \overline{y})$ est caractérisé par le fait que les deux chaînes de bits doivent pouvoir être obtenues en itérant les fonctions de compression un nombre élevé de fois. Si on fixe un bloc de message m, alors on peut examiner le *graphe fonctionnel* de $f_1(x) = h_1(x, m)$ (cf. section 4.2.1). L'état spécial est donc de *profondeur élevée*. Cette attaque utilise la structure de données décrite section 4.2.7 pour représenter des morceaux du graphe fonctionnel.

La phase B de l'attaque en seconde préimage, qui produit un état spécial et le connecte au message de départ, dépend d'un paramètre g_1 (à choisir) et fonctionne de la manière suivante :

1. Construire deux structures de données $\mathcal{D}_i \leftarrow \textsc{Build}(f_i, g_1)$, pour $i = 1, 2$.
2. Répéter $2^{2n-2g_1-\ell}$ fois : initialiser un dictionnaire vide T, générer un bloc aléatoire m', puis
   (a) Pour tout $x \in \mathcal{D}_1$, calculer $h_1(x, m')$ et vérifier si cela donne l'un des états internes « originaux » a_i. Si oui, stocker $i \mapsto x$ dans T.
   (b) Pour tout $y \in \mathcal{D}_2$, calculer $h_2(y, m')$ et vérifier si cela donne l'un des b_j. Si oui, et si la clef j est présente dans $T[j]$, terminer l'algorithme. L'état spécial est $(T[j], y)$ et m' l'envoie sur (a_j, b_j), donc le « connecte » au message de départ.

Le nombre d'opérations nécessaire à l'exécution de la phase B est donc $2^{g_1} + 2^{2n-g_1-\ell}$.

La phase C de l'attaque en seconde préimage, qui connecte le message expansif à l'état spécial, dépend d'un autre paramètre $g_2 > g_1$ (à choisir lui aussi). Tous les détails ne sont pas décrits ici, mais en gros, il faut :

1. Construire deux structures de données $\mathcal{D}_i \leftarrow \textsc{Build}(f_i, g_2)$, pour $i = 1, 2$.
2. Répéter 2^{3g_1-n} fois : générer un bloc aléatoire m' puis itérer $h_1(\cdot, m')$ et $h_2(\cdot, m')$ à partir de $(\hat{x}, \hat{y})$ jusqu'à ce que les deux chaînes obtenues aient une intersection non-vide avec $\mathcal{D}_1$ et $\mathcal{D}_2$, respectivement. (on fixe une borne supérieure de 2^{n-g_1} itérations pour éviter une boucle infinie).

La première étape nécessite 2^{g_2} opérations. La taille moyenne des chaînes dans la 2ème étape est de 2^{n-g_2}, donc elle nécessite $2^{3g_1-g_2}$ opérations. Choisir $g_2 = 3g_1/2$ minimise donc la durée totale de la phase C.

Équilibrer les phases B et C revient à choisir $g_1 = \frac{2}{5}(2n - \ell)$, ce qui donne une complexité en temps de $2^{3(2n-\ell)/5}$, tant que $\ell \leq 3n/4$. La complexité optimale, qui est $2^{3n/4}$ opérations, est alors obtenue avec les messages les plus longs possibles. L'attaque nécessite moins de 2^n opérations (donc améliore l'attaque en préimage de Joux déjà décrite) lorsque $\ell > n/3$.

Étude du coût

L'étude de coût de cette attaque soulève bien des problèmes intéressants, notamment car plusieurs portions sont intrinsèquement séquentielles et que la complexité en espace est élevée. Le fait que sa complexité temporelle et spatiale soit $2^{3n/4}$ la rend « suspecte » au regard du théorème 1.

Un problème qui a été ignoré dans la discussion ci-dessus est qu'il faut bien hacher le message de départ pour produire la séquence des états internes (a_i, b_i). Or, comme on l'a déjà discuté section 5.1, ceci est lent et nécessite une grosse machine, donc le coût de l'opération est $2^{2\ell}$. Rien que ceci contraint la taille maximale des messages visés à $\ell \leq 2^{n/2}$, donc le nombre d'opérations grimpe automatiquement à $2^{9n/10}$ ou plus. De plus, ceci impose une borne inférieure sur la taille de la machine.

On a déjà discuté section 4.2.7 que les coûts des étapes B.1 et C.1 sont $2^{2g_1-n/2}$ et $2^{2g_2-n/2}$, respectivement.

Examinons maintenant les autres étapes. L'étape B.2 teste de manière répétée l'appartenance à un gros ensemble statique, donc vu ce qu'on a déjà dit dans la section 4.1, elle coûte $2^{2n-g_1-2\ell/3}$. Le coût total de la phase B est alors :

$$2^{2g_1-n/2} + 2^{2n-g_1-2\ell/3}$$

À l'opposé, l'étape C.2 effectue des accès parcimonieux à la mémoire, grâce à l'utilisation des points distingués : il suffit de tester si les itérés appartiennent à $\mathcal{D}_i$ lorsqu'ils sont distingués, or une fraction 2^{n-g_2} des points est distinguée, donc un seul accès mémoire suffit en moyenne. Ceci nécessite $2^{3g_1-g_2}$ opérations et la fréquence des accès est 2^{g_2-n}. Toujours avec 2^p processeurs, le temps d'exécution est $2^{3g_1-g_2-p}$ sur une machine de taille $2^p + 2^{n-g_2} + (2^p \cdot 2^{g_2-n})^{3/2}$ (d'après le théorème 1). Le coût de cette étape est donc :

$$2^{3g_1-g_2} + 2^{3g_1-2g_2-p+n} + 2^{3g_1+p/2+g_2/2-3n/2}$$

La valeur optimale du (log du) nombre de processeurs est $p = \frac{5}{3}(n - g_2)$. Le coût ainsi minimisé est alors :

$$2^{3g_1-g_2} + 2^{3g_1-g_2/3-2n/3}$$

En fait, le premier terme domine toujours le deuxième lorsque $g_2 < n$, et c'est donc toujours le cas. On aboutit à la conclusion (non-triviale) que le coût de l'étape C.2 est égal à son nombre d'opérations. Le coût total de la phase C de l'attaque est donc :

$$2^{2g_2-n/2} + 2^{3g_1-g_2}$$

Minimiser le coût total de l'attaque revient à minimiser le max de tous les exposants rencontrés, c'est-à-dire résoudre le problème d'optimisation linéaire suivant :

$$\begin{array}{rll}
\text{minimiser} & x & \\
\text{sous les contraintes} & 2\ell \leq x, & \text{hachage de } M \\
& \frac{n}{2} \leq g_1, & \text{contrainte sur } g_1 \\
& n - \ell \leq g_1, & \text{idem} \\
& g_1 \leq g_2, & \text{contrainte sur } g_2 \\
& 2g_1 - \frac{n}{2} \leq x, & \text{étape B.1} \\
& 2n - g_1 - \frac{2}{3}\ell \leq x, & \text{étape B.2} \\
& 2g_2 - \frac{n}{2} \leq x, & \text{étape C.1} \\
& 3g_1 - g_2 \leq x. & \text{étape C.2}
\end{array}$$

Sa solution optimale est $x = 23n/22$ (ce résultat est facile à obtenir en utilisant SageMath comme interface à la librairie PPL qui résout ce genre de problèmes sur les rationnels). Le coût de l'attaque

ne peut donc pas descendre sous celui de la recherche exhaustive. En tout cas, il faudrait trouver de nouveaux raffinements algorithmiques pour espérer y arriver.

Aussi bête que ça puisse paraître, le point bloquant est le *hachage du message initial*. Si on le négligeait, alors on obtiendrait un coût total de $2^{23n/26}$ avec des messages de cette taille-là.

5.5 Codes d'authentification de messages

On s'intéresse ici aux attaques en contrefaçon universelle, donc les plus puissantes ; il s'agit pour l'adversaire de forger un tag valide pour un message imposé. Pour cela, l'adversaire à accès à un oracle qui lui fournit le *tag* de n'importe quel message.

5.5.1 Attaque de Peyrin et Wang sur HMAC [169]

En 2014, Peyrin et Wang [169] ont présenté la première attaque sur HMAC capable de forger un *tag* valide pour n'importe quel message de longueur $2^{n/6}$ blocs en temps $2^{5n/6}$, un résultat remarquable. Pour simplifier, on suppose que l'état interne de la fonction de hachage fait n bits (la même taille que les tags).

L'attaque fonctionne même sur NMAC, c'est-à-dire avec deux clefs complètement indépendantes. Rappelons que NMAC peut se voir comme :

$$\mathsf{NMAC}(K_1, K_2, M) = H(K_1, H(K_2, M)),$$

où $H(K, M)$ désigne une fonction de hachage de Merkle-Damgård dans laquelle la clef K est utilisée comme vecteur d'initialisation. HMAC est la construction où les deux clefs sont dérivées d'une même clef « maîtresse » K en XORant deux constantes ipad et opad différentes dessus.

Pour produire un tag valide pour un message M imposé, l'attaque fonctionne en calculant une seconde préimage sur $H(K_2, \cdot)$, et ceci sans connaître K_2. Alors, il suffit d'obtenir le *tag* de la seconde préimage, et c'est automatiquement un *tag* valide pour M aussi.

L'obstacle majeur qu'il s'agit de contourner est que l'adversaire ne connaît pas la séquence des états internes $x_0, x_1, \ldots$, obtenus lorsque $H(K_2, M)$ est calculé, or les attaques en seconde préimage habituelles (qui sont des variantes de la *long-message attack*) en ont besoin. Le problème se ramène donc à déterminer *un* des états internes x_i, car alors les suivants peuvent être calculés facilement (la clef K_2 n'intervient qu'au début).

Pour s'en sortir, la première idée consiste à calculer, pour tout i, une paire de blocs $m_i \neq m'_i$ tels que $f(x_i, m_i) = f(x_i, m'_i)$. Ceci permet de tester, avec faible probabilité d'erreur, si une chaîne de bits $\hat{x}$ est égale à x_i, alors même que x_i est inconnu : il suffit de tester si $f(\hat{x}, m_i) = f(\hat{x}, m'_i)$. Par conséquent, pour tout $i \leq 2^\ell$, on cherche une paire de blocs $m_i \neq m'_i$ qui réalisent une collision entre les *tags* de $M_0 \| \ldots \| M_i \| m$ et $M_0 \| \ldots \| M_i \| m'$. Ceci nécessite en tout $2^{\ell+n/2}$ requêtes à l'oracle avec des messages de longueur 2^ℓ (ou moins). Ceci nécessite donc un temps $2^{2\ell+n/2}$.

La deuxième idée astucieuse de Peyrin et Wang consiste à exploiter le graphe de la fonction de compression avec un bloc m fixé. Une procédure astucieuse, qui n'est pas décrite ici, permet de déterminer la *hauteur* dans ce graphe de chacun des x_i. Rappelons que la hauteur est la distance qui sépare un nœud du cycle qu'il y a nécessairement dans la composante connexe qui le contient. Ces hauteurs sont en principe inférieures à $2^{n/2}$; elles sont donc (heuristiquement) toutes distinctes si M est de taille négligeable devant $2^{n/4}$; comme l'attaque utilise un message M de taille $2^\ell = 2^{n/6}$, tout va bien. On suppose que le message M est découpé en blocs, avec $M = M_0 \| M_1 \| \ldots \| M_{2^\ell}$.

Pour identifier l'un des x_i, on maintient un ensemble Y contenant $2^{n-\ell}$ chaînes de n bits, partitionné par hauteur. On s'attend à ce que l'un des x_i appartienne à Y, et c'est celui-là qu'on va retrouver. L'ensemble Y est exactement la structure de données représentant une partie du graphe fonctionnel qu'on a déjà discuté dans la section 4.2.7, c'est-à-dire le résultat de l'exécution de $\textsc{Build}(f, n - \ell)$.

Pour chaque x_i, on examine le sous-ensemble de Y composé des nœuds de la même hauteur que x_i, et on les teste tous avec la procédure décrite ci-dessus. Ceci nécessite en tout $2^{n-\ell}$ évaluations de la fonction de compression, car chaque x_i a une hauteur distincte ; les sous-ensembles de Y qui correspondent à chaque x_i sont donc disjoints. Construire Y nécessite aussi $2^{n-\ell}$ opérations. Enfin, une fois que l'un des x_i est récupéré, calculer une seconde préimage « normale » (à la Kelsey-Schneier [127]) nécessite encore $2^{n-\ell}$ opérations.

Étude du coût

Du point de vue du coût, plusieurs problèmes se posent, et le premier est la « facturation » des invocations répétés de l'oracle, avec des messages de taille exponentielle. On suppose ici qu'un nombre non-borné d'instances de l'oracle peuvent être invoquées en parallèle, mais que le temps de transmission à l'oracle d'un message de k blocs est proportionnel à k.

Ensuite, un autre problème est que la nécessité de devoir accéder aux nœuds dans Y « de manière aléatoire » et de pouvoir accéder instantanément à la hauteur de chaque nœud de Y semble interdire les techniques permettant une représentation compacte de Y. Du coup, tester si $X \cap Y = \emptyset$, autrement dit effectuer : « *pour chaque x_i, examiner le sous-ensemble de Y composé des nœuds de la bonne hauteur* », signifie en réalité calculer une jointure entre X et Y où les hauteurs servent de clef, et ceci ceci coûte $2^{\frac{4}{3}(n-\ell)}$ (cf. section 4.3). Or, comme ℓ est nécessairement inférieur à $n/4$, pour garantir le caractère distinct des hauteurs des x_i, il semble que l'attaque, du moins sous sa forme originale, ne puisse pas coûter moins cher que la recherche exhaustive.

Pour tenter d'améliorer les choses, on peut remarquer que si Y est stocké de manière compacte avec des points distingués (comme discuté dans la section 4.2.7), alors il suffit de stocker la hauteur de chaque point distingué pour être capable d'énumérer tous les points de Y avec leur hauteur. On peut donc, au lieu d'énumérer X et tester les morceaux de Y qui correspondent, faire l'inverse : énumérer Y et tester *le* nœud (éventuel) de X qui correspond à chaque nœud de Y. On a vu que le coût de la construction de Y est alors $2^{3n/2-2\ell}$.

La phase de recherche de collision entre X et Y se ramène alors exactement au test d'appartenance à un gros ensemble discuté section 4.1, et coûte donc $2^{n-\frac{2}{3}\ell}$.

Le coût de l'attaque s'obtient alors en résolvant le problème d'optimisation linéaire suivant :

$$\begin{array}{rll} \text{minimiser} & x & \\ \text{sous les contraintes} & \ell \leq \dfrac{n}{4}, & \text{contrainte sur } \ell \\ & \dfrac{n}{2} + 2\ell \leq x, & \text{création des « filtres »} \\ & \dfrac{3}{2}n - 2\ell \leq x, & \text{création de } Y \\ & n - \dfrac{2}{3}n \leq x. & \text{intersection } X \cap Y \end{array}$$

sommer la 2ème et la 3ème inéquation révèle que $n \leq x$, donc que le coût ne peut pas descendre en dessous de 2^n (atteint pour $\ell = n/4$). Cette attaque ne peut donc pas coûter moins que la recherche exhaustive.

5.5.2 Amélioration de Guo, Peyrin, Sasaki et Wang [118]

L'attaque précédente a ensuite été améliorée par les mêmes auteurs, accompagnés de Guo et Sasaki [118] : la nouvelle version cible des messages de taille $2^{n/4}$ en temps $2^{3n/4}$.

Il y a deux améliorations. La première consiste à noter qu'il suffit de calculer *une* paire de blocs $m \neq m'$ qui réalisent une collision entre les *tags* de $M\|m$ et $M\|m'$. En effet, pour tester si $\hat{x} = x_i$, il suffit alors de tester si

$$f(f(\dots f(f(x_i, M_i), M_{i+1}) \dots, M_{2^\ell}), m) = f(f(\dots f(f(\hat{x}, M_i), M_{i+1}) \dots, M_{2^\ell}), m').$$

Cette observation réduit le temps de production du « filtre » de $2^{n/2+2\ell}$ à $2^{n/2+\ell}$. Par contre, elle augmente la complexité du test de $\hat{x}$ de $\mathcal{O}(1)$ à $\mathcal{O}(2^\ell)$.

La deuxième amélioration part de l'observation (empirique) que si on échantillonne 2^x sommets dans le graphe fonctionnel en utilisant la procédure BUILD de la section 4.2.7, alors pour toute hauteur $\lambda \leq 2^{n/2}/n$, on va tirer environ $2^{x-n/2}$ sommets de hauteur λ. Autrement dit, les sommets qui ont une hauteur inférieure à $2^{n/2}$ ont une hauteur à peu près uniformément répartie. Du coup, à peu de choses près, l'ensemble Y contient $2^{n/2-\ell}$ sommets de chaque hauteur. Du coup, l'attaque devient : pour chaque x_i, extraire les $2^{n/2-\ell}$ sommets de Y qui ont la même hauteur, et pour chacun d'entre eux, tester si la valeur suggérée de x_i est la bonne. Il y a 2^ℓ valeurs de x_i à examiner, pour chacun d'entre eux il y a $2^{n/2-\ell}$ valeurs candidates, et tester chaque candidat nécessite 2^ℓ opérations. La durée totale est donc $2^{n/2+\ell}$.

Enfin, calculer la seconde préimage coûte toujours $2^{n-\ell}$, donc choisir $\ell = 2^{n/4}$ optimise la complexité en temps.

Étude du coût

On considère toujours la reformulation de l'attaque suivante : stocker le graphe fonctionnel sous forme de points distingués ; itérer sur les nœuds de Y ; pour chaque $y \in Y$, interroger un (gros) dictionnaire pour déterminer s'il y a un nœud x_i de la bonne hauteur (la réponse est oui avec probabilité $2^{\ell-n/2}$) ; le cas échéant, tester la valeur candidate pour x_i. Il y a au final $2^{n/2}$ tests à effectuer.

Le test d'une valeur de x_i nécessite un temps 2^ℓ et c'est une opération intrinsèquement séquentielle (qui ne nécessite pas de mémoire). Minimiser le coût revient alors à résoudre le problème :

$$\begin{aligned}
\text{minimiser} \quad & x \\
\text{sous les contraintes} \quad & \ell \leq \frac{n}{4}, && \text{contrainte sur } \ell \\
& \frac{n}{2} + \ell \leq x, && \text{création DU « filtre »} \\
& \frac{3}{2}n - 2\ell \leq x, && \text{création de } Y \\
& n - \frac{2}{3}\ell \leq x. && \text{intersection } X \cap Y
\end{aligned}$$

Las ! La solution optimale a encore un coût de 2^n. Pour s'en sortir, il faudrait relâcher la borne supérieure imposée sur la taille de la contrefaçon, mais il faut du coup changer un peu l'attaque, car l'unicité des hauteurs des sommets de X ne sera plus garantie. Cette unicité simplifie la présentation, mais elle n'est sûrement pas indispensable. Si on suppose que $\ell < n/2$ et que les hauteurs des sommets de X sont uniformément réparties dans $[0; 2^{n/2}]$, alors le nombre de nœuds qui ont la même hauteur est majoré par $\mathcal{O}(n)$ avec forte probabilité, ce qui n'est pas de nature à changer l'exposant de l'attaque. Avec $\ell \leq n/2$, alors un coût de $2^{5n/6}$ est atteint pour $\ell = n/3$.

L'attaque originale n'était donc pas viable, mais ses raffinements successifs le sont.

5.5.3 Attaque de Leurent, Nandi et Sibleyras sur SUM-ECBC [147]

En 2018, Leurent, Nandi et Sibleyras [147] ont proposé une collection d'attaques contre des MACs offrant en principe une sécurité « au-delà du paradoxe des anniversaires ». Ces attaques sont principalement évaluées dans un modèle où le nombre d'opérations effectuées par l'adversaire est illimité, mais où le nombre de requêtes adressées à l'oracle de MAC est, lui, comptabilisé. Ces attaques viennent alors en complément des preuves effectuées dans le même modèle.

L'abstract de l'article en question dit néanmoins :

> Moreover, we give a variant of the attack against SUM-ECBC and GCM-SIV2 with time and data complexity $\tilde{\mathcal{O}}\left(2^{6n/7}\right)$.

Cette fois ci, il s'agit de vraies attaques en contrefaçon universelle dans le modèle habituel. La première (sur SUM-ECBC) se ramène à résoudre une instance du problème 4XOR sur $12n/7$ bits, avec quatre listes de taille $2^{3n/7}$ — il n'y a donc qu'une seule solution.

Nommons A, B, C et D les quatre listes. Une adaptation directe de l'algorithme de Schroeppel-Shamir [177] permet de résoudre le problème 4XOR efficacement. Pour chaque chaîne de $3n/7$ bits x, calculer les jointures $U \leftarrow (x + A) \bowtie_{3n/7} B$ et $U \leftarrow (x + C) \bowtie_{3n/7} D$, puis calculer l'intersection entre U et V. Ceci nécessite $2^{6n/7}$ opérations et $2^{3n/7}$ unités de mémoire. Vu ce qu'on a dit section 4.3 sur le coût du calcul des jointures, on conclut que l'attaque coûte exactement 2^n. Elle n'est donc pas meilleure que la recherche exhaustive.

Chapitre 6

Conclusion

Ce mémoire discute du coût de certaines attaques cryptographiques et de certains algorithmes qui peuvent jouer un rôle dans la cryptanalyse. La première conclusion qu'on peut en tirer, c'est que cela ne remet pas en cause la cryptanalyse « théorique » : le chapitre 5 démontre que certaines attaques très sophistiquées (par exemple celle de la section 5.5.2) ont un coût asymptotiquement plus faible que la force brute. On peut donc faire des choses assez théoriques tout en s'intéressant à leur coût.

A contrario, une attaque qui nécessite moins d'opérations que la force brute mais qui coûte plus cher n'a que peu de chances de se transformer en menace concrète.

Dans presque tous les cas, le coût des calculs est une donnée intéressante pour comparer des algorithmes. Si les données nécessaires à un calcul tiennent dans la mémoire d'un seul ordinateur, raisonner sur la base du seul nombre d'opérations n'est pas déraisonnable.

Mais si ce n'est pas le cas, alors le coût est un meilleur prédicteur de la difficulté des calculs, à mon avis. Les raisonnements du chapitre 4, et la discussion des implantations existantes, quand il y en a, vont dans ce sens. Dès que la complexité en espace dépasse la taille de la mémoire d'un nœud de calcul et nécessite de passer par un réseau, tout change complètement.

Les algorithmes qui résolvent les problèmes classiques sur les réseaux euclidiens d'une part, et le problème du sac à dos d'autre part, sont les grands absents de ce chapitre. Pourtant, les problèmes de coût n'y sont pas négligeables, car certains algorithmes ont une complexité en mémoire exponentielle : pour les réseaux, il serait intéressant de comparer les techniques d'énumération et de crible (qui nécessitent moins d'opérations mais beaucoup de mémoire) du point de vue du coût.

Le modèle du coût discuté au chapitre 3 est de fait un modèle théorique abstrait, et les petites expériences réalisées pour ce manuscrit n'apportent pas de réponse ferme et définitive à la question de son adéquation à la réalité. Il postule l'existence d'un « prix de la mémoire » : dans la plupart des cas, un algorithme qui effectue T opérations et nécessite M cases mémoires aura souvent un coût de $TM^{1/3}$, sous une parallélisation optimale. Mais on peut se demander si ce facteur $1/3$ est pertinent, ou bien s'il faut le remplacer par une autre constante (non-nulle).

Au passage, décrire des implantations « de coût optimal » (égal au nombre d'opérations) de certains algorithmes est loin d'être trivial, et cela peut soulever des problèmes théoriques intéressants. Par exemple, si un problème est P-complet (par exemple, le flot maximal ou la programmation linéaire), alors on n'en connaît pas, à ce jour, d'implantation parallèle de profondeur poly-logarithmique. Il y a donc une limite au parallélisme qu'on sait en extraire. Est-ce que cela implique que le coût de n'importe quelle implantation sera strictement supérieur au nombre d'opérations ?

Ce sont mes tentatives d'implanter des attaques cryptographiques ou des algorithmes pour la cryptanalyse qui m'ont amenées à vouloir réfléchir au coût des calculs. En faisant des « travaux pratiques », on est vite confronté au fait que raisonner sur la base du seul nombre d'opérations est problématique. Pourtant, c'est la manière de raisonner majoritaire dans notre thème de recherche, qui a un

positionnement souvent tourné vers la théorie.

D'autres communautés de chercheurs (par exemple celle du calcul scientifique), dont le travail consiste davantage à produire des programmes concrets, accordent plus d'importance aux réalisations pratiques. Sans surprise, leur conception de la complexité des calculs est très différente.

À l'avenir, je compte essayer de mettre à profit le petit capital accumulé lors de la rédaction de ce manuscrit. Je suis convaincu que raisonner sur la base du coût des calculs ne peut qu'amener à produire des attaques et des programmes plus efficaces sur du vrai matériel. C'est ce terrain que je compte continuer à explorer. Beaucoup d'algorithmes autour et alentour de la cryptographie ne sont pas implantés, et leur intérêt pratique n'est même pas discuté. Ce manuscrit apporte un certain nombre de réponses, notamment dans le cas de la résolution de systèmes polynomiaux ou du problème 3XOR. Mais dans le cas du décodage des codes correcteurs d'erreur, il me semble que la situation est encore assez ouverte.

Curriculum Vitae

Emplois

— Sep. 2007 → Nov. 2011 : soutenance de ma thèse préparée dans l'équipe crypto de l'École normale supérieure (« Etudes d'hypothèses algorithmiques et attaques de primitives cryptographiques »).
— Nov. 2011 → Sep. 2012 : post-doc dans l'équipe de crypto de l'université de Versailles / Saint-Quentin en Yvelines.
— Sep. 2012 → Nov. 2020 : maître de conférence dans l'équipe Calcul Formel et Haute-Performance de l'université de Lille.
— Nov. 2020 → : maître de conférence dans l'équipe ALMASTY (algorithmes pour la sécurité des communications) à Sorbonne Université

Recherche ~~de financement~~ partenariale

— ANR BRUTUS (14–18) : partenaire (« Chiffrements authentifiés et résistants aux attaques par canaux auxiliaires »)
— ANR JCJC GORILLA (20–24) : porteur (« Cryptanalyse algorithmique avec de vraies implantations »)
— ANR ASTRID POSTCRYPTUM (20–24) : partenaire (« Cryptanalyse algébrique pour la cryptographie post-quantum »)
— ANR KLEPTOMANIAC (21–25) : partenaire (« Key Length Estimates : Practical and Theoretical Optimizations and Modern Approaches on NFS Instances for Accurate Costs »)

Encadrement de doctorants

— 2015–2018 : Claire Delaplace (à 50%). « Algorithmes d'algèbre linéaire pour la cryptographie ».
— 2020– : Ambroise Fleury (à 50%). « Vers des algorithmes post-quantiques efficaces. Exploration de la cyber-sécurité embarquée ».

Encadrement de stagiaires de M2

— 2013 : Laurent Grémy (école des mines d'Alès, ensuite doctorant au LORIA), « autour du problème LPN »
— 2015 : Claire Delaplace (université de Versailles), « algèbre linéaire creuse directe modulo p »
— 2019 : Ryan Lefebvre (université de Lille, ensuite SS2i), « Amélioration d'heuristiques de partitionnement de graphes, avec application à la résolution de systèmes linéaires »
— 2019 : Mellila Bouam (École Supérieure d'Informatique, Alger, ensuite autre parcours de M2), « Implantation haute-performance de méthodes de résolution du problème 3XOR »
— 2021 : Julia Sauvage (Sorbonne Université), « Cryptanalyse du générateur pseudo-aléatoire PCG »
— 2021 : Yasmine Ikhelif (Sorbonne Université), « Algorithmes de décodage des codes linéaires dans le modèle du coût complet »
— 2021 : Quentin Hammerer (Sorbonne Université), « Vers une implantation de référence de l'algorithme de Joux-Vitse »

— 2022 : Mickaël Hamdad (Sorbonne Université), « Étude d'algorithmes pour le problème des paires les plus proches »

Encadrement de stagiaires de M1

— 2019 : Julia Sauvage (Sorbonne Université), « Cryptanalyse de PRNGs survendus »
— 2020 : Jérôme Bonacchi (EPU de Sorbonne Université), « amélioration d'un algorithme de transposition de matrice creuse »
— 2020 : Mickaël Hamdad (Sorbonne Université), « signatures basées sur le problème MQ »
— 2021 : Paul-Antoine Levesque (Sorbonne Université), « Factorisation PLUQ de matrices creuses, multi-thread, modulo p »

Publications

Je ne distingue pas journaux et conférences. Les articles marqués ($\star$) ont été soumis avant ma soutenance de thèse.

2021 *A Simple Deterministic Algorithm for Systems of Quadratic Polynomials over* $\mathbb{F}_2$. CB, Claire Delaplace et Monika Trimoska. SOSA 2022.

2021 *Computational Records with Aging Hardware : Controlling Half the Output of SHA-256.* Mellila Bouam, CB, Claire Delaplace et Camille Noûs. Parallel Computing [39]

2021 *Cryptanalysis of Modular Exponentiation Outsourcing Protocols.* CB, Florette Martinez et Damien Vergnaud. The Computer Journal [60].

2020 *Parallel Structured Gaussian Elimination for the Number Field Sieve.* CB et Paul Zimmerman. Mathematical Cryptology [62].

2020 *Predicting the PCG Pseudo-Random Number Generator In Practice.* CB, Florette Martinez et Julia Sauvage. TOSC [59].

2018 *Revisiting and Improving Algorithms for the 3XOR Problem.* CB, Claire Delaplace et Pierre-Alain Fouque. TOSC [44].

2017 *Parallel Sparse PLUQ Factorization modulo p.* CB, Claire Delaplace et Marie-Émilie Voge. PASCO 2017 [46].

2017 *Fast Lattice-Based Encryption : Stretching Spring.* CB, Claire Delaplace, Pierre-Alain Fouque et Paul Kirchner. PQCrypto 2017 [45].

2016 *Sparse Gaussian Elimination modulo p : an Update.* CB et Claire Delaplace. CASC 2016 [43].

2016 *New Second-Preimage Attacks on Hash Functions.* Elena Andreeva, CB, Orr Dunkelman, Pierre-Alain Fouque, Jonathan Hoch, John Kelsey, Adi Shamir et Sébastien Zimmer. Journal of Cryptology [6].

2014 *Cryptographic Schemes Based on the ASASA Structure : Black-box, White-box, and Public-key.* Alex Biryukov, CB et Dmitry Khovratovich. Asiacrypt 2014 [32].

2013 *Provable Second Preimage Resistance Revisited.* CB et Bastien Vayssière. SAC 2013 [61].

2013 *Fast Exhaustive Search for Quadratic Systems in* $\mathbb{F}_2$ *on FPGAs.* CB, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen et Bo-Yin Yang. SAC 2013 [42].

2013 *Graph-Theoretic Algorithms for the Isomorphism of Polynomials Problem.* CB, Pierre-Alain Fouque et Amandine Véber. Eurocrypt 2013 [57].

2011 ($\star$) *Practical Key-recovery For All Possible Parameters of SFLASH.* CB, Pierre-Alain Fouque et Gilles Macario-Rat. Asiacrypt 2011 [56].

2011 ($\star$) *Automatic Search of Attacks on round-reduced AES and Applications.* CB, Patrick Derbez et Pierre-Alain Fouque. Crypto 2011 [48].

2011 ($\star$) *New Insights on Impossible Differential Cryptanalysis.* CB, Orr Dunkelman, Pierre-Alain Fouque et Gaëtan Leurent. SAC 2011 [49].

2011 ($\star$) *Practical Cryptanalysis of the Identification Scheme Based on the Isomorphism of Polynomial With One Secret Problem.* CB, Jean-Charles Faugère, Pierre-Alain Fouque et Ludovic Perret. PKC 2011 [52].

2010 ($\star$) *Low Data Complexity Attacks on AES.* CB, Patrick Derbez, Orr Dunkelman, Nathan Keller and Pierre-Alain Fouque. IEEE Tr. on Information Theory [47].

2010 ($\star$) *Fast Exhaustive Search for Polynomial Systems in* $\mathbb{F}_2$. CB, Hsieh-Chung Chen, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, Adi Shamir and Bo-Yin Yang. CHES 2011 [41].

2010 ($\star$) *Security Analysis of SIMD.* CB, Pierre-Alain Fouque et Gaëtan Leurent. SAC 2010 [55].

2010 ($\star$) *Attacks on Hash Functions based on Generalized Feistel, Application to Reduced-Round Lesamnta and SHAvite-3/512.* CB, Orr Dunkelman, Pierre-Alain Fouque et Gaëtan Leurent. SAC 2010 [51].

2010 ($\star$) *Another Look at Complementation Properties.* CB, Orr Dunkelman, Pierre-Alain Fouque et Gaëtan Leurent. FSE 2010 [50].

2009 ($\star$) *A Family of Weak Keys in HFE and the Corresponding Practical Key-Recovery.* CB, Pierre-Alain Fouque, Antoine Joux et Joana Treger. Journal of Mathematical Cryptology [54].

2009 ($\star$) *Herding, Second Preimage and Trojan Message Attacks Beyond Merkle-Damgård.* Elena Andreeva, CB, Orr Dunkelman et John Kelsey. SAC 2009 [7].

2008 ($\star$) *Analysis of the Collision Resistance of RadioGatún using Algebraic Techniques.* CB et Pierre-Alain Fouque. SAC 2008 [53].

2008 ($\star$) *Second Preimage Attacks on Dithered Hash Functions.* Elena Andreeva, CB, Pierre-Alain Fouque, Jonathan Hoch, John Kelsey, Adi Shamir et Sébastien Zimmer. Eurocrypt 2008 [8].

2007 ($\star$) *Using First-Order Theorem Provers in the Jahob Data Structure Verification System.* Viktor Kuncak, Thomas Wies, Karen Zee et Martin Rinard. VMCAI 2007 [58].

Bibliographie

[1] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella Béguelin, and Paul Zimmermann. Imperfect forward secrecy : how diffie-hellman fails in practice. *Commun. ACM*, 62(1) :106–114, 2019.

[2] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9) :1116–1127, 1988.

[3] Martin R. Albrecht, Léo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn W. Postlethwaite, and Marc Stevens. The General Sieve Kernel and New Records in Lattice Reduction. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 717–746. Springer, 2019.

[4] Josh Alman. An illuminating algorithm for the light bulb problem. In Jeremy T. Fineman and Michael Mitzenmacher, editors, *2nd Symposium on Simplicity in Algorithms, SOSA 2019, January 8-9, 2019, San Diego, CA, USA*, volume 69 of *OASICS*, pages 2 :1–2 :11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[5] Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 522–539. SIAM, 2021.

[6] Elena Andreeva, Charles Bouillaguet, Orr Dunkelman, Pierre-Alain Fouque, Jonathan J. Hoch, John Kelsey, Adi Shamir, and Sébastien Zimmer. New second-preimage attacks on hash functions. *J. Cryptol.*, 29(4) :657–696, 2016.

[7] Elena Andreeva, Charles Bouillaguet, Orr Dunkelman, and John Kelsey. Herding, second preimage and trojan message attacks beyond merkle-damgård. In Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography, 16th Annual International Workshop, SAC 2009, Calgary, Alberta, Canada, August 13-14, 2009, Revised Selected Papers*, volume 5867 of *Lecture Notes in Computer Science*, pages 393–414. Springer, 2009.

[8] Elena Andreeva, Charles Bouillaguet, Pierre-Alain Fouque, Jonathan J. Hoch, John Kelsey, Adi Shamir, and Sébastien Zimmer. Second preimage attacks on dithered hash functions. In Nigel P. Smart, editor, *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, volume 4965 of *Lecture Notes in Computer Science*, pages 270–288. Springer, 2008.

[9] Kazumaro Aoki and Yu Sasaki. Preimage attacks on one-block md4, 63-step MD5 and more. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *Selected Areas in Cryptography, 15th International Workshop, SAC 2008, Sackville, New Brunswick, Canada, August 14-15, Revised Selected Papers*, volume 5381 of *Lecture Notes in Computer Science*, pages 103–119. Springer, 2008.

[10] Nicolas Aragon, Julien Lavauzelle, and Matthieu Lequesne. decodingchallenge.org, 2019.

[11] Gwénolé Ars, Jean-Charles Faugère, Hideki Imai, Mitsuru Kawazoe, and Makoto Sugita. Comparison between XL and gröbner basis algorithms. In Pil Joong Lee, editor, *Advances in Cryptology - ASIACRYPT 2004, 10th International Conference on the Theory and Application of Cryptology and Information Security, Jeju Island, Korea, December 5-9, 2004, Proceedings*, volume 3329 of *Lecture Notes in Computer Science*, pages 338–353. Springer, 2004.

[12] Jean-Philippe Aumasson. Too much crypto. *IACR Cryptol. ePrint Arch.*, 2019 :1492, 2019.

[13] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins : Sort vs. hash revisited. *PVLDB*, 7(1) :85–96, 2013.

[14] Achiya Bar-On, Orr Dunkelman, Nathan Keller, Eyal Ronen, and Adi Shamir. Improved key recovery attacks on reduced-round AES with practical data and memory complexities. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 185–212. Springer, 2018.

[15] Magali Bardet, Jean-Charles Faugère, and Bruno Salvy. On the complexity of the F5 gröbner basis algorithm. *J. Symb. Comput.*, 70 :49–70, 2015.

[16] Magali Bardet, Jean-Charles Faugère, Bruno Salvy, and Pierre-Jean Spaenlehauer. On the complexity of solving quadratic boolean systems. *J. Complexity*, 29(1) :53–75, 2013.

[17] Anja Becker, Antoine Joux, Alexander May, and Alexander Meurer. Decoding random binary linear codes in $2^{n/20}$: How 1+1=0 improves information set decoding. *IACR Cryptol. ePrint Arch.*, 2012 :26, 2012.

[18] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4) :275–292, 2005.

[19] Daniel J. Bernstein. Circuits for integer factorization : a proposal, 2001. URL : `http://cr.yp.to/papers.html`.

[20] Daniel J. Bernstein. Understanding brute force, 2005. Available online : `http://cr.yp.to/papers.html#bruteforce`.

[21] Daniel J. Bernstein. Better price-performance ratios for generalized birthday attacks. In *SHARCS'07 : Special-purpose Hardware for Attacking Cryptographic Systems*, 2007.

[22] Daniel J. Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, and Wen Wang. Classic mceliece, 2017.

[23] Daniel J. Bernstein and Tanja Lange. Batch NFS. In Antoine Joux and Amr M. Youssef, editors, *Selected Areas in Cryptography - SAC 2014 - 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers*, volume 8781 of *Lecture Notes in Computer Science*, pages 38–58. Springer, 2014.

[24] Daniel J Bernstein, Tanja Lange, Ruben Niederhagen, Christiane Peters, and Peter Schwabe. FSBday : Implementing Wagner's Generalized Birthday Attack. In *INDOCRYPT*, pages 18–38, 2009.

[25] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Attacking and defending the mceliece cryptosystem. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography, Second International Workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17-19, 2008, Proceedings*, volume 5299 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2008.

[26] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Smaller decoding exponents : Ball-collision decoding. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 743–760. Springer, 2011.

[27] Jérémy Berthomieu, Christian Eder, and Mohab Safey El Din. msolve : A library for solving polynomial systems. In Frédéric Chyzak and George Labahn, editors, *ISSAC '21 : International Symposium on Symbolic and Algebraic Computation, Virtual Event, Russia, July 18-23, 2021*, pages 51–58. ACM, 2021.

[28] Luk Bettale, Jean-Charles Faugère, and Ludovic Perret. Hybrid approach for solving multivariate systems over finite fields. *J. Math. Cryptol.*, 3(3) :177–197, 2009.

[29] Luk Bettale, Jean-Charles Faugère, and Ludovic Perret. Solving polynomial systems over finite fields : improved analysis of the hybrid approach. In Joris van der Hoeven and Mark van Hoeij, editors, *International Symposium on Symbolic and Algebraic Computation, ISSAC'12, Grenoble, France - July 22 - 25, 2012*, pages 67–74. ACM, 2012.

[30] Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in-)security of 64-bit block ciphers : Collision attacks on HTTP over TLS and openvpn. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 456–467. ACM, 2016.

[31] Gianfranco Bilardi and Franco P. Preparata. Area-time lower-bound techniques with applications to sorting. *Algorithmica*, 1(1) :65–91, 1986.

[32] Alex Biryukov, Charles Bouillaguet, and Dmitry Khovratovich. Cryptographic schemes based on the ASASA structure : Black-box, white-box, and public-key (extended abstract). In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 63–84. Springer, 2014.

[33] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2 : New generation of memory-hard functions for password hashing and other applications. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 292–302. IEEE, 2016.

[34] Andreas Björklund, Petteri Kaski, and Ryan Williams. Solving systems of polynomial equations over GF(2) by a parity-counting self-reduction. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 26 :1–26 :13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[35] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, François-Xavier Standaert, John P. Steinberger, and Elmar Tischhauser. Key-alternating ciphers in a provable setting : Encryption using a small number of public permutations - (extended abstract). In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, volume 7237 of *Lecture Notes in Computer Science*, pages 45–62. Springer, 2012.

[36] Dan Boneh, editor. *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*. Springer, 2003.

[37] Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, Arjen K. Lenstra, and Peter L. Montgomery. Solving a 112-bit prime elliptic curve discrete logarithm problem on game consoles using sloppy reduction. *Int. J. Appl. Cryptogr.*, 2(3) :212–228, 2012.

[38] Wieb Bosma, John J. Cannon, and Catherine Playoust. The Magma Algebra System I : The User Language. *J. Symb. Comput.*, 24(3/4) :235–265, 1997.

[39] Mellila Bouam, Charles Bouillaguet, Claire Delaplace, and Camille Noûs. Computational records with aging hardware : Controlling half the output of sha-256. *Parallel Computing*, page 102804, 2021.

[40] Fabrice Boudot, Pierrick Gaudry, Aurore Guillevic, Nadia Heninger, Emmanuel Thomé, and Paul Zimmermann. Comparing the difficulty of factorization and discrete logarithm : A 240-digit experiment. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 62–91. Springer, 2020.

[41] Charles Bouillaguet, Hsieh-Chung Chen, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, Adi Shamir, and Bo-Yin Yang. Fast exhaustive search for polynomial systems in $\mathbb{F}_2$. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 203–218. Springer, 2010.

[42] Charles Bouillaguet, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, and Bo-Yin Yang. Fast Exhaustive Search for Quadratic Systems in $\mathbb{F}_2$ on FPGAs. In *Selected Areas in Cryptography*, volume 8282 of *Lecture Notes in Computer Science*, pages 205–222. Springer, 2013. `https://eprint.iacr.org/2013/436.pdf`.

[43] Charles Bouillaguet and Claire Delaplace. Sparse gaussian elimination modulo p : An update. In Vladimir P. Gerdt, Wolfram Koepf, Werner M. Seiler, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing - 18th International Workshop, CASC 2016, Bucharest, Romania, September 19-23, 2016, Proceedings*, volume 9890 of *Lecture Notes in Computer Science*, pages 101–116. Springer, 2016.

[44] Charles Bouillaguet, Claire Delaplace, and Pierre-Alain Fouque. Revisiting and improving algorithms for the 3xor problem. *IACR Trans. Symmetric Cryptol.*, 2018(1) :254–276, 2018.

[45] Charles Bouillaguet, Claire Delaplace, Pierre-Alain Fouque, and Paul Kirchner. Fast lattice-based encryption : Stretching spring. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017, Utrecht, The Netherlands, June 26-28, 2017, Proceedings*, volume 10346 of *Lecture Notes in Computer Science*, pages 125–142. Springer, 2017.

[46] Charles Bouillaguet, Claire Delaplace, and Marie-Emilie Voge. Parallel sparse PLUQ factorization modulo p. In Jean-Charles Faugère, Michael B. Monagan, and Hans-Wolfgang Loidl, editors, *Proceedings of the International Workshop on Parallel Symbolic Computation, PASCO@ISSAC 2017, Kaiserslautern, Germany, July 23-24, 2017*, pages 8 :1–8 :10. ACM, 2017.

[47] Charles Bouillaguet, Patrick Derbez, Orr Dunkelman, Pierre-Alain Fouque, Nathan Keller, and Vincent Rijmen. Low-data complexity attacks on AES. *IEEE Trans. Inf. Theory*, 58(11) :7002–7017, 2012.

[48] Charles Bouillaguet, Patrick Derbez, and Pierre-Alain Fouque. Automatic search of attacks on round-reduced AES and applications. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 169–187. Springer, 2011.

[49] Charles Bouillaguet, Orr Dunkelman, Pierre-Alain Fouque, and Gaëtan Leurent. New insights on impossible differential cryptanalysis. 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 243–259. Springer, 2011.

[50] Charles Bouillaguet, Orr Dunkelman, Gaëtan Leurent, and Pierre-Alain Fouque. Another look at complementation properties. In Seokhie Hong and Tetsu Iwata, editors, *Fast Software Encryption, 17th International Workshop, FSE 2010, Seoul, Korea, February 7-10, 2010, Revised Selected Papers*, volume 6147 of *Lecture Notes in Computer Science*, pages 347–364. Springer, 2010.

[51] Charles Bouillaguet, Orr Dunkelman, Gaëtan Leurent, and Pierre-Alain Fouque. Attacks on hash functions based on generalized feistel : Application to reduced-round *Lesamnta* and *SHAvite-3*$_{512}$. In Alex Biryukov, Guang Gong, and Douglas R. Stinson, editors, *Selected Areas in Cryptography - 17th International Workshop, SAC 2010, Waterloo, Ontario, Canada, August 12-13, 2010, Revised Selected Papers*, volume 6544 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2010.

[52] Charles Bouillaguet, Jean-Charles Faugère, Pierre-Alain Fouque, and Ludovic Perret. Practical cryptanalysis of the identification scheme based on the isomorphism of polynomial with one

secret problem. In Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi, editors, *Public Key Cryptography - PKC 2011 - 14th International Conference on Practice and Theory in Public Key Cryptography, Taormina, Italy, March 6-9, 2011. Proceedings*, volume 6571 of *Lecture Notes in Computer Science*, pages 473–493. Springer, 2011.

[53] Charles Bouillaguet and Pierre-Alain Fouque. Analysis of the collision resistance of radiogatúnusing algebraic techniques. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *Selected Areas in Cryptography, 15th International Workshop, SAC 2008, Sackville, New Brunswick, Canada, August 14-15, Revised Selected Papers*, volume 5381 of *Lecture Notes in Computer Science*, pages 245–261. Springer, 2008.

[54] Charles Bouillaguet, Pierre-Alain Fouque, Antoine Joux, and Joana Treger. A family of weak keys in HFE and the corresponding practical key-recovery. *J. Math. Cryptol.*, 5(3-4) :247–275, 2012.

[55] Charles Bouillaguet, Pierre-Alain Fouque, and Gaëtan Leurent. Security analysis of SIMD. In Alex Biryukov, Guang Gong, and Douglas R. Stinson, editors, *Selected Areas in Cryptography - 17th International Workshop, SAC 2010, Waterloo, Ontario, Canada, August 12-13, 2010, Revised Selected Papers*, volume 6544 of *Lecture Notes in Computer Science*, pages 351–368. Springer, 2010.

[56] Charles Bouillaguet, Pierre-Alain Fouque, and Gilles Macario-Rat. Practical key-recovery for all possible parameters of SFLASH. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 667–685. Springer, 2011.

[57] Charles Bouillaguet, Pierre-Alain Fouque, and Amandine Véber. Graph-theoretic algorithms for the "isomorphism of polynomials" problem. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 211–227. Springer, 2013.

[58] Charles Bouillaguet, Viktor Kuncak, Thomas Wies, Karen Zee, and Martin C. Rinard. Using first-order theorem provers in the jahob data structure verification system. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, volume 4349 of *Lecture Notes in Computer Science*, pages 74–88. Springer, 2007.

[59] Charles Bouillaguet, Florette Martinez, and Julia Sauvage. Practical seed-recovery for the PCG pseudo-random number generator. *IACR Trans. Symmetric Cryptol.*, 2020(3) :175–196, 2020.

[60] Charles Bouillaguet, Florette Martinez, and Damien Vergnaud. Cryptanalysis of Modular Exponentiation Outsourcing Protocols. *The Computer Journal*, 05 2021. bxab066.

[61] Charles Bouillaguet and Bastien Vayssière. Provable second preimage resistance revisited. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, volume 8282 of *Lecture Notes in Computer Science*, pages 513–532. Springer, 2013.

[62] Charles Bouillaguet and Paul Zimmermann. Parallel structured gaussian elimination for the number field sieve. *Mathematical Cryptology*, 0(1) :22–39, Jan. 2021.

[63] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[64] R. P. Brent and H. T. Kung. The chip complexity of binary arithmetic. In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, STOC '80, pages 190–200, New York, NY, USA, 1980. ACM.

[65] Richard P. Brent. An improved monte carlo factorization algorithm. *BIT Numerical Mathematics*, 20 :176–184, 1980.

[66] Johannes A. Buchmann, Richard Lindner, and Markus Rückert. Explicit hard instances of the shortest vector problem. In Johannes A. Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography, Second International Workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17-19, 2008, Proceedings*, volume 5299 of *Lecture Notes in Computer Science*, pages 79–94. Springer, 2008.

[67] Richard H. Byrd, Mary E. Hribar, and Jorge Nocedal. An interior point algorithm for large-scale nonlinear programming. *SIAM Journal on Optimization*, 9(4) :877–900, 1999.

[68] Anne Canteaut and Florent Chabaud. A new algorithm for finding minimum-weight words in a linear code : Application to mceliece's cryptosystem and to narrow-sense BCH codes of length 511. *IEEE Trans. Inf. Theory*, 44(1) :367–378, 1998.

[69] Anne Canteaut and Nicolas Sendrier. Cryptanalysis of the original mceliece cryptosystem. In Kazuo Ohta and Dingyi Pei, editors, *Advances in Cryptology - ASIACRYPT '98, International Conference on the Theory and Applications of Cryptology and Information Security, Beijing, China, October 18-22, 1998, Proceedings*, volume 1514 of *Lecture Notes in Computer Science*, pages 187–199. Springer, 1998.

[70] Antoine Casanova, Jean-Charles Faugère, Gilles Macario-Rat, Jacques Patarin, Ludovic Perret, and Jocelyn Ryckeghem. GeMSS : A Great Multivariate Short Signature. Research report, UPMC - Paris 6 Sorbonne Universités ; INRIA Paris Research Centre, MAMBA Team, F-75012, Paris, France ; LIP6 - Laboratoire d'Informatique de Paris 6, December 2017.

[71] Certicom Research. Certicom ecc challenges. `https://www.certicom.com/content/dam/certicom/images/pdfs/challenge-2009.pdf`, 1997. [Online ; accessed 15-May-2020].

[72] Anando G. Chatterjee, Mahendra K. Verma, Abhishek Kumar, Ravi Samtaney, Bilel Hadri, and Rooh Khurram. Scaling of a fast fourier transform and a pseudo-spectral fluid solver up to 196608 cores. *J. Parallel Distributed Comput.*, 113 :77–91, 2018.

[73] Charles Clos. A study of non-blocking switching networks. *The Bell System Technical Journal*, 32(2) :406–424, 1953.

[74] Andrew R. Conn, Katya Scheinberg, and Luis N. Vicente. *Introduction to Derivative-Free Optimization*. Society for Industrial and Applied Mathematics, USA, 2009.

[75] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *J. Comput. Syst. Sci.*, 7(4) :354–375, 1973.

[76] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[77] Paul W. Coteus, Shawn Hall, Todd Takken, Rick A. Rand, S. Tian, Gerard V. Kopcsay, Randy Bickford, Francis P. Giordano, Christopher Marroquin, and Mark J. Jeanson. Packaging the IBM blue gene/q supercomputer. *IBM J. Res. Dev.*, 57(1/2) :2, 2013.

[78] Nicolas T. Courtois, Louis Goubin, Willi Meier, and Jean-Daniel Tacier. Solving underdefined systems of multivariate quadratic equations. In David Naccache and Pascal Paillier, editors, *Public Key Cryptography, 5th International Workshop on Practice and Theory in Public Key Cryptosystems, PKC 2002, Paris, France, February 12-14, 2002, Proceedings*, volume 2274 of *Lecture Notes in Computer Science*, pages 211–227. Springer, 2002.

[79] Nicolas T. Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In Bart Preneel, editor, *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Springer, 2000.

[80] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice E. Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp : Towards a realistic model of parallel computation. In Marina C. Chen and Robert Halstead, editors, *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), San Diego, California, USA, May 19-22, 1993*, pages 1–12. ACM, 1993.

[81] M. Delcourt, T. Kleinjung, A.K. Lenstra, S. Nath, D. Page, and N. Smart. Using the cloud to determine key strengths – triennial update. Cryptology ePrint Archive, Report 2018/1221, 2018. `https://eprint.iacr.org/2018/1221`.

[82] The FPLLL development team. fplll, a lattice reduction library. Available at `https://github.com/fplll/fplll`, 2016.

[83] Whitfield Diffie and Martin E. Hellman. Exhaustive cryptanalysis of the NBS data encryption standard. *Computer*, 10(6) :74–84, 1977.

[84] Jintai Ding, Joshua Deaton, Vishakha, and Bo-Yin Yang. The nested subset differential attack : A practical direct attack against luov which forges a signature within 210 minutes. Cryptology ePrint Archive, Report 2020/967, 2020. `https://eprint.iacr.org/2020/967`.

[85] Jintai Ding, Joshua Deaton, Vishakha, and Bo-Yin Yang. The nested subset differential attack - A practical direct attack against LUOV which forges a signature within 210 minutes. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 329–347. Springer, 2021.

[86] Itai Dinur. New attacks on the concatenation and XOR hash combiners. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 484–508. Springer, 2016.

[87] Itai Dinur. Tight time-space lower bounds for finding multiple collision pairs and their applications. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part I*, volume 12105 of *Lecture Notes in Computer Science*, pages 405–434. Springer, 2020.

[88] Itai Dinur. Cryptanalytic applications of the polynomial method for solving multivariate equation systems over GF(2). In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 374–403. Springer, 2021.

[89] Itai Dinur. Improved algorithms for solving polynomial systems over GF(2) by multiple parity-counting. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2550–2564. SIAM, 2021.

[90] Itai Dinur, Orr Dunkelman, Nathan Keller, and Adi Shamir. New attacks on feistel structures with improved memory complexities. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 433–454. Springer, 2015.

[91] Itai Dinur, Orr Dunkelman, Nathan Keller, and Adi Shamir. Key recovery attacks on iterated even-mansour encryption schemes. *J. Cryptology*, 29(4) :697–728, 2016.

[92] Itai Dinur, Orr Dunkelman, and Adi Shamir. Improved attacks on full GOST. In Anne Canteaut, editor, *Fast Software Encryption - 19th International Workshop, FSE 2012, Washington, DC, USA, March 19-21, 2012. Revised Selected Papers*, volume 7549 of *Lecture Notes in Computer Science*, pages 9–28. Springer, 2012.

[93] Vivien Dubois, Pierre-Alain Fouque, Adi Shamir, and Jacques Stern. Practical cryptanalysis of SFLASH. In Alfred Menezes, editor, *Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings*, volume 4622 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2007.

[94] Ilya Dumer. On minimum distance decoding of linear codes. In *Proc. 5th Joint Soviet-Swedish Int. Workshop Inform. Theory*, pages 50–52, 1991.

[95] Dave Dunning. Fabrics — why we love them and why we hate them.

[96] Cynthia Dwork, Andrew V. Goldberg, and Moni Naor. On memory-bound functions for fighting spam. In Boneh [36], pages 426–444.

[97] Amr Elmasry and Jyrki Katajainen. Lean programs, branch mispredictions, and sorting. In Evangelos Kranakis, Danny Krizanc, and Flaminia L. Luccio, editors, *Fun with Algorithms - 6th International Conference, FUN 2012, Venice, Italy, June 4-6, 2012. Proceedings*, volume 7288 of *Lecture Notes in Computer Science*, pages 119–130. Springer, 2012.

[98] Andre Esser, Robert Kübler, and Floyd Zweydinger. A faster algorithm for finding closest pairs in hamming metric. *CoRR*, abs/2102.02597, 2021.

[99] Andre Esser, Alexander May, and Floyd Zweydinger. Mceliece needs a break – solving mceliece-1284 and quasi-cyclic-2918 with modern isd. Cryptology ePrint Archive, Report 2021/1634, 2021. `https://ia.cr/2021/1634`.

[100] Jean-Charles Faugère. A new efficient algorithm for computing grobner bases (f4). *Journal of Pure and Applied Algebra*, 139(1-3) :61–68, 1999.

[101] Jean Charles Faugère. A new efficient algorithm for computing gröbner bases without reduction to zero (f5). In *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation*, ISSAC '02, page 75–83, New York, NY, USA, 2002. Association for Computing Machinery.

[102] Jean-Charles Faugère. Fgb : A library for computing gröbner bases. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings*, volume 6327 of *Lecture Notes in Computer Science*, pages 84–87. Springer, 2010.

[103] Jean-Charles Faugère and Antoine Joux. Algebraic cryptanalysis of hidden field equation (HFE) cryptosystems using gröbner bases. In Boneh [36], pages 44–60.

[104] Matthieu Finiasz and Nicolas Sendrier. Security bounds for the design of code-based cryptosystems. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 88–105. Springer, 2009.

[105] Philippe Flajolet and Andrew M. Odlyzko. Random mapping statistics. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *Advances in Cryptology - EUROCRYPT '89, Workshop on the Theory and Application of of Cryptographic Techniques, Houthalen, Belgium, April 10-13, 1989, Proceedings*, volume 434 of *Lecture Notes in Computer Science*, pages 329–354. Springer, 1989.

[106] Robert W. Floyd. Permuting information in idealized two-level storage. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 105–109. Plenum Press, New York, 1972.

[107] Electronic Frontier Foundation. *Cracking DES : Secrets of Encryption Research, Wiretap Politics & Chip Design*. O'Reilly Series. Electronic Frontier Foundation, 1998.

[108] Franz Franchetti, Yevgen Voronenko, and Gheorghe Almasi. Automatic generation of the hpc challenge's global fft benchmark for bluegene/p. In Michel Daydé, Osni Marques, and Kengo Nakajima, editors, *High Performance Computing for Computational Science - VECPAR 2012*, pages 187–200, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[109] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $\mathcal{O}(1)$ worst case access time. *J. ACM*, 31(3) :538–544, June 1984.

[110] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3) :424–436, 1993.

[111] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3) :533–551, 1994.

[112] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proc. IEEE*, 93(2) :216–231, 2005.

[113] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1) :4 :1–4 :22, 2012.

[114] Hiroki Furue, Shuhei Nakamura, and Tsuyoshi Takagi. Improving thomae-wolf algorithm for solving underdetermined multivariate quadratic polynomial problem. In Jung Hee Cheon and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 12th International Workshop, PQ-Crypto 2021, Daejeon, South Korea, July 20-22, 2021, Proceedings*, volume 12841 of *Lecture Notes in Computer Science*, pages 65–78. Springer, 2021.

[115] Bharan Giridhar, Michael Cieslak, Deepankar Duggal, Ronald G. Dreslinski, Hsing Min Chen, Robert Patti, Betina Hold, Chaitali Chakrabarti, Trevor N. Mudge, and David T. Blaauw. Exploring DRAM organizations for energy-efficient and resilient exascale memories. In William Gropp and Satoshi Matsuoka, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, pages 23 :1–23 :12. ACM, 2013.

[116] Jian Guo, Jérémy Jean, Ivica Nikolic, and Yu Sasaki. Meet-in-the-middle attacks on generic feistel constructions. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 458–477. Springer, 2014.

[117] Jian Guo, Jérémy Jean, Ivica Nikolic, and Yu Sasaki. Meet-in-the-middle attacks on classes of contracting and expanding feistel constructions. *IACR Trans. Symmetric Cryptol.*, 2016(2) :307–337, 2016.

[118] Jian Guo, Thomas Peyrin, Yu Sasaki, and Lei Wang. Updates on generic attacks against HMAC and NMAC. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 131–148. Springer, 2014.

[119] Frank K. Gürkaynak, Kris Gaj, Beat Muheim, Ekawat Homsirikamol, Christoph Keller, Marcin Rogawski, Hubert Kaeslin, and Jens peter Kaps. Lessons learned from designing a 65nm asic for evaluating third round sha-3 candidates," the third sha-3 candidate conference. In *Proc. Cryptographic Hardware and Embedded Systems workshop, CHES 2010*, pages 248–263, 2012.

[120] Takanori Isobe. A single-key attack on the full GOST block cipher. In Antoine Joux, editor, *Fast Software Encryption - 18th International Workshop, FSE 2011, Lyngby, Denmark, February 13-16, 2011, Revised Selected Papers*, volume 6733 of *Lecture Notes in Computer Science*, pages 290–305. Springer, 2011.

[121] Takanori Isobe and Kyoji Shibutani. Generic key recovery attack on feistel scheme. In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part I*, volume 8269 of *Lecture Notes in Computer Science*, pages 464–485. Springer, 2013.

[122] Takanori Isobe and Kyoji Shibutani. New key recovery attacks on minimal two-round even-mansour ciphers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 244–263. Springer, 2017.

[123] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[124] Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In Matthew K. Franklin, editor, *Advances in Cryptology - CRYPTO 2004, 24th Annual International CryptologyConference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.

[125] Antoine Joux. *Algorithmic cryptanalysis*. CRC Press, 2009.

[126] Antoine Joux and Vanessa Vitse. A Crossbred Algorithm for Solving Boolean Polynomial Systems. In *NuTMiC*, volume 10737 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2017. `https://eprint.iacr.org/2017/372.pdf`.

[127] John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In Ronald Cramer, editor, *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005.

[128] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman J. J. te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. In Tal Rabin, editor, *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, volume 6223 of *Lecture Notes in Computer Science*, pages 333–350. Springer, 2010.

[129] Thorsten Kleinjung, Claus Diem, Arjen K. Lenstra, Christine Priplata, and Colin Stahlke. Computation of a 768-bit prime field discrete logarithm. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 185–201, 2017.

[130] Thorsten Kleinjung, Arjen K. Lenstra, Dan Page, and Nigel P. Smart. Using the cloud to determine key strengths. In Steven Galbraith and Mridul Nandi, editors, *Progress in Cryptology - INDOCRYPT 2012*, pages 17–39, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[131] Lars R. Knudsen. The security of feistel ciphers with six rounds or less. *J. Cryptol.*, 15(3) :207–222, 2002.

[132] Donald E. Knuth. Computer programming as an art. *Commun. ACM*, 17(12) :667–673, December 1974.

[133] Donald E. Knuth. Theory and practice. *Theor. Comput. Sci.*, 90(1) :1–15, 1991.

[134] Donald E. Knuth. *Searching and sorting*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 10 January 1998. This is a full BOOK entry.

[135] Donald Ervin Knuth. *The art of computer programming, Volume I : Fundamental Algorithms, 3rd Edition*. Addison-Wesley, 1997.

[136] D. Kraft. *A software package for sequential quadratic programming*. Deutsche Forschungs- und Versuchsanstalt für Luft- und Raumfahrt Köln : Forschungsbericht. Wiss. Berichtswesen d. DFVLR, 1988.

[137] Sandeep S. Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. Breaking ciphers with COPACOBANA - A cost-optimized parallel code breaker. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 101–118. Springer, 2006.

[138] Po-Chun Kuo, Michael Schneider, Özgür Dagdelen, Jan Reichelt, Johannes A. Buchmann, Chen-Mou Cheng, and Bo-Yin Yang. Extreme enumeration on GPU and in clouds - - how many dollars you need to break SVP challenges -. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan,*

September 28 - October 1, 2011. Proceedings, volume 6917 of *Lecture Notes in Computer Science*, pages 176–191. Springer, 2011.

[139] Mario Lamberger, Florian Mendel, Vincent Rijmen, and Koen Simoens. Memoryless near-collisions via coding theory. *Des. Codes Cryptogr.*, 62(1) :1–18, 2012.

[140] Grégory Landais. *Mise en œuvre de cryptosystèmes basés sur les codes correcteurs d'erreurs et de leurs cryptanalyses. (Code-based cryptosystems and cryptanalysis implementation)*. PhD thesis, Pierre and Marie Curie University, Paris, France, 2014.

[141] Daniel Lazard. Gröbner-bases, gaussian elimination and resolution of systems of algebraic equations. In J. A. van Hulzen, editor, *Computer Algebra, EUROCAL '83, European Computer Algebra Conference, London, England, March 28-30, 1983, Proceedings*, volume 162 of *Lecture Notes in Computer Science*, pages 146–156. Springer, 1983.

[142] Pil Joong Lee and Ernest F. Brickell. An observation on the security of mceliece's public-key cryptosystem. In Christoph G. Günther, editor, *Advances in Cryptology - EUROCRYPT '88, Workshop on the Theory and Application of of Cryptographic Techniques, Davos, Switzerland, May 25-27, 1988, Proceedings*, volume 330 of *Lecture Notes in Computer Science*, pages 275–280. Springer, 1988.

[143] Charles E. Leiserson. Fat-trees : Universal networks for hardware-efficient supercomputing. *IEEE Trans. Computers*, 34(10) :892–901, 1985.

[144] Arjen K. Lenstra, Adi Shamir, Jim Tomlinson, and Eran Tromer. Analysis of bernstein's factorization circuit. In Yuliang Zheng, editor, *Advances in Cryptology - ASIACRYPT 2002, 8th International Conference on the Theory and Application of Cryptology and Information Security, Queenstown, New Zealand, December 1-5, 2002, Proceedings*, volume 2501 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2002.

[145] Jeffrey S. Leon. A probabilistic algorithm for computing minimum weights of large error-correcting codes. *IEEE Trans. Inf. Theory*, 34(5) :1354–1359, 1988.

[146] Gaëtan Leurent. Time-memory trade-offs for near-collisions. 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 205–218. Springer, 2013.

[147] Gaëtan Leurent, Mridul Nandi, and Ferdinand Sibleyras. Generic attacks against beyond-birthday-bound macs. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 306–336. Springer, 2018.

[148] Gaëtan Leurent and Thomas Peyrin. SHA-1 is a shambles : First chosen-prefix collision on SHA-1 and application to the PGP web of trust. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1839–1856. USENIX Association, 2020.

[149] Gaëtan Leurent and Ferdinand Sibleyras. Low-memory attacks against two-round even-mansour using the 3-XOR problem. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part II*, volume 11693 of *Lecture Notes in Computer Science*, pages 210–235. Springer, 2019.

[150] Gaëtan Leurent and Lei Wang. The sum can be weaker than each part. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 345–367. Springer, 2015.

[151] Richard J. Lipton and Jonathan S. Sandberg. PRAM : a scalable shared memory. Technical report, Princeton University, 1988. Available online : `https://www.cs.princeton.edu/research/techreps/TR-180-88`.

[152] John D. C. Little. A proof for the queuing formula : $l = \lambda w$. *Operations Research*, 9(3) :383–387, 1961.

[153] Daniel Lokshtanov, Ramamohan Paturi, Suguru Tamaki, R. Ryan Williams, and Huacheng Yu. Beating brute force for systems of polynomial equations over finite fields. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 2190–2202. SIAM, 2017.

[154] Piotr Luszczek, David H. Bailey, Jack J. Dongarra, Jeremy Kepner, Robert F. Lucas, Rolf Rabenseifner, and Daisuke Takahashi. S12 - the HPC challenge (HPCC) benchmark suite. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA*, page 213. ACM Press, 2006.

[155] M. Donald MacLaren. Internal sorting by radix plus sifting. *J. ACM*, 13(3) :404–411, 1966.

[156] Alexander May, Alexander Meurer, and Enrico Thomae. Decoding random linear codes in $\tilde{\mathcal{O}}(2^{0.054n})$. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 107–124. Springer, 2011.

[157] Alexander May and Ilya Ozerov. On computing nearest neighbors with applications to decoding of binary linear codes. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 203–228. Springer, 2015.

[158] R. J. McEliece. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *Deep Space Network Progress Report*, 44 :114–116, January 1978.

[159] Darius Mercadier and Pierre-Évariste Dagand. Usuba : high-throughput and constant-time ciphers, by construction. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 157–173. ACM, 2019.

[160] Eric Miles and Emanuele Viola. Substitution-permutation networks, pseudorandom functions, and natural proofs. *J. ACM*, 62(6) :46 :1–46 :29, 2015.

[161] Mridul Nandi. Revisiting Security Claims of XLS and COPA. *IACR Cryptology ePrint Archive*, 2015 :444, 2015.

[162] Ruben Niederhagen, Kai-Chun Ning, and Bo-Yin Yang. Implementing joux-vitse's crossbred algorithm for solving MQ systems over GF(2) on gpus. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings*, volume 10786 of *Lecture Notes in Computer Science*, pages 121–141. Springer, 2018.

[163] Ruben Niederhagen and Bo-Yin Yang *et al.* Breaking ECC2K-130. *IACR Cryptology ePrint Archive*, 2009 :541, 2009. `https://eprint.iacr.org/2009/541.pdf`.

[164] Ivica Nikolić and Yu Sasaki. Refinements of the k-tree Algorithm for the Generalized Birthday Problem. In *ASIACRYPT*, pages 683–703. Springer, 2014.

[165] Ivica Nikolic, Lei Wang, and Shuang Wu. Cryptanalysis of round-reduced `led`. 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 112–129. Springer, 2013.

[166] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2) :122–144, 2004.

[167] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

[168] Colin Percival and Simon Josefsson. The scrypt password-based key derivation function. *RFC*, 7914 :1–16, 2016.

[169] Thomas Peyrin and Lei Wang. Generic universal forgery attack on iterative hash-based macs. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 147–164. Springer, 2014.

[170] John M. Pollard. A Monte Carlo method for factorization. *BIT*, 15 :331–334, 1975.

[171] E. Prange. The use of information sets in decoding cyclic codes. *IRE Transactions on Information Theory*, 8(5) :5–9, 1962.

[172] Jean-Jacques Quisquater and Jean-Paul Delescaille. Other cycling tests for des. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 255–256, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[173] Jean-Jacques Quisquater and Jean-Paul Delescaille. How easy is collision search ? application to des. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *Advances in Cryptology — EUROCRYPT '89*, pages 429–434, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[174] Daniel Richardson. Some undecidable problems involving elementary functions of a real variable. *The Journal of Symbolic Logic*, 33(4) :514–520, 1968.

[175] Yogish Sabharwal, Saurabh K. Garg, Rahul Garg, John A. Gunnels, and Ramendra K. Sahoo. Optimization of fast fourier transforms on the blue gene/l supercomputer. In *Proceedings of the 15th International Conference on High Performance Computing*, HiPC'08, page 309–322, Berlin, Heidelberg, 2008. Springer-Verlag.

[176] John E. Savage. Area—time tradeoffs for matrix multiplication and related problems in vlsi models. *Journal of Computer and System Sciences*, 22(2) :230 – 242, 1981.

[177] Richard Schroeppel and Adi Shamir. A $T = \mathcal{O}\left(2^{n/2}\right)$, $S = \mathcal{O}\left(2^{n/4}\right)$ Algorithm for Certain NP-Complete Problems. *SIAM J. Comput.*, 10(3) :456–464, 1981.

[178] Adi Shamir. Factoring Numbers in $\mathcal{O}(\log n)$ Arithmetic Steps. *Inf. Process. Lett.*, 8(1) :28–31, 1979.

[179] Jacques Stern. A method for finding codewords of small weight. In Gérard D. Cohen and Jacques Wolfmann, editors, *Coding Theory and Applications, 3rd International Colloquium, Toulon, France, November 2-4, 1988, Proceedings*, volume 388 of *Lecture Notes in Computer Science*, pages 106–113. Springer, 1988.

[180] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 570–596, Cham, 2017. Springer International Publishing.

[181] Marc Stevens, Arjen K. Lenstra, and Benne de Weger. Chosen-prefix collisions for MD5 and colliding X.509 certificates for different identities. In Moni Naor, editor, *Advances in Cryptology - EUROCRYPT 2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007, Proceedings*, volume 4515 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2007.

[182] Ulrich Tamm. Deterministic communication complexity of set intersection. *Discret. Appl. Math.*, 61(3) :271–283, 1995.

[183] The CADO-NFS Development Team. CADO-NFS, an implementation of the number field sieve algorithm, 2017. Release 2.3.0.

[184] The Rainbow team. Response to recent paper by ward beullens, 2021.

[185] Enrico Thomae and Christopher Wolf. Solving underdetermined systems of multivariate quadratic equations revisited. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *Public Key Cryptography - PKC 2012 - 15th International Conference on Practice and Theory in Public Key Cryptography, Darmstadt, Germany, May 21-23, 2012. Proceedings*, volume 7293 of *Lecture Notes in Computer Science*, pages 156–171. Springer, 2012.

[186] C. D. Thompson. Area-time complexity for vlsi. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, STOC '79, pages 81–88, New York, NY, USA, 1979. ACM.

[187] C. D. Thompson and H. T. Kung. Sorting on a mesh-connected parallel computer. *Commun. ACM*, 20(4) :263–271, April 1977.

[188] Yosuke Todo. Upper bounds for the security of several feistel networks. In Colin Boyd and Leonie Simpson, editors, *Information Security and Privacy - 18th Australasian Conference, ACISP 2013, Brisbane, Australia, July 1-3, 2013. Proceedings*, volume 7959 of *Lecture Notes in Computer Science*, pages 302–317. Springer, 2013.

[189] Monika Trimoska. *Combinatorics in Algebraic and Logical Cryptanalysis*. Theses, Université de Picardie - Jules Verne, January 2021.

[190] Monika Trimoska, Sorina Ionica, and Gilles Dequen. Time-memory analysis of parallel collision search algorithms. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2) :254–274, 2021.

[191] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, pages 263–277, New York, NY, USA, 1981. ACM.

[192] Peter van Emde Boas. Machine models and simulation. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A : Algorithms and Complexity*, pages 1–66. Elsevier and MIT Press, 1990.

[193] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptology*, 12(1) :1–28, 1999.

[194] Evangelos Vasilakis. An instruction level energy characterization of ARM processors. Technical Report MSU-CSE-06-2, Computer Architecture and VLSI Systems (CARV) Laboratory, Institute of Computer Science (ICS), Foundation of Research and Technology Hellas (FORTH), March 2015. Technical Report FORTH-ICS/TR-450.

[195] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0 : Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17 :261–272, 2020.

[196] P. Vitányi. Locality, communication, and interconnect length in multicomputers. *SIAM Journal on Computing*, 17(4) :659–672, 1988.

[197] J. Vuillemin. A combinatorial limit to the computing power of vlsi circuits. *IEEE Trans. Comput.*, 32(3) :294–300, March 1983.

[198] David Wagner. A generalized birthday problem. In *CRYPTO*, pages 288–304, 2002.

[199] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In Victor Shoup, editor, *Advances in Cryptology - CRYPTO 2005 : 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

[200] Michael J. Wiener. The Full Cost of Cryptanalytic Attacks. *J. Cryptology*, 17(2) :105–124, 2004. `https://doi.org/10.1007/s00145-003-0213-5`.

[201] Wikipedia contributors. Triple des — Wikipedia, the free encyclopedia, 2019. [Online ; accessed 11-September-2019].

[202] Wikipedia contributors. Rsa factoring challenge — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=RSA_Factoring_Challenge&oldid=944725244`, 2020. [Online ; accessed 15-May-2020].

[203] William A. Wulf and Sally A. McKee. Hitting the memory wall : implications of the obvious. *SIGARCH Computer Architecture News*, 23(1) :20–24, 1995.

[204] Bo-Yin Yang and Jiun-Ming Chen. Theoretical analysis of XL over small fields. In Huaxiong Wang, Josef Pieprzyk, and Vijay Varadharajan, editors, *Information Security and Privacy : 9th Australasian Conference, ACISP 2004, Sydney, Australia, July 13-15, 2004. Proceedings*, volume 3108 of *Lecture Notes in Computer Science*, pages 277–288. Springer, 2004.

[205] Takanori Yasuda, Xavier Dahan, Yun-Ju Huang, Tsuyoshi Takagi, and Kouichi Sakurai. A multivariate quadratic challenge toward post-quantum generation cryptography. *ACM Commun. Comput. Algebra*, 49(3) :105–107, 2015.