Linear algebra algorithms for cryptography

Claire Delaplace

UNIVERSITE BRETAGNE LOIRE MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
COMUE **U**NIVERSITE **B**RETAGNE **L**OIRE

Ecole Doctorale N°601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Claire Delaplace

Algorithmes d'algèbre linéaire pour la cryptographie

Thèse présentée et soutenue à RENNES , le 21 Novembre 2018
Unité de recherche : Institut de Recherche en Informatique et Système Aléatoire (IRISA), UMR 6074

Rapporteurs avant soutenance :
Guénaël Renault : Expert Cryptographie.
Damien Vergnaud : Professeur Université de Paris 6.

Composition du jury :

Examinateurs : Aurélie Bauer : Expert Cryptographie.
Jean-Guillaume Dumas : Professeur Université Grenoble Alpes.
María Naya-Plasencia : Directrice de Recherche Inria.
Marion Videau : Maître de Conférences Université de Lorraine.
Rapporteurs : Guénaël Renault : Expert Cryptographie.
Damien Vergnaud : Professeur Université de Paris 6.

Dir. de thèse : Pierre-Alain Fouque : Professeur Université de Rennes 1.
Co-dir. de thèse : Charles Bouillaguet : Maître de Conférence Université de Lille.

Remerciements

Je tiens à remercier tous ceux qui m'ont aidée à réaliser cette thèse, et tout d'abord l'ANR Brutus pour l'avoir financée.

Un grand merci à mes deux directeurs, Pierre-Alain Fouque et Charles Bouillaguet, qui m'ont soutenue et encadrée tout au long de ces trois années. Plus particulièrement je remercie Pierre-Alain pour sa patience et son calme à toute épreuve, et pour avoir toujours pris le temps de suivre mes travaux, malgré son emploi du temps chargé. Je remercie également Charles, pour son enthousiasme, sa bonne humeur, et pour m'avoir conseillée et guidée dans mes recherches, et cela malgré la distance entre Lille et Rennes.

Je remercie ensuite les membres du jury : Aurélie Bauer, Jean-Guillaume Dumas, Maria Naya-Plasencia, Guénaël Renault, Damien Vergnaud et Marion Videau. Je remercie d'autant plus Guénaël Renault et Damien Vergnaud, pour avoir accepté d'être les rapporteurs, et pour leur remarques pertinentes qui ont amélioré la qualité de ce manuscrit.

Durant ces trois ans j'ai travaillé dans deux équipes où j'ai pu bénéficier d'un excellent environnement. Tout d'abord à Lille, où je remercie les membres (et anciens membres) de l'équipe CFPH, pour leur accueil chaleureux: François Boulier (pour ses conseils et ses remarques), Yacine Bouzidi, François Lemaire (pour les discussions que nous avons eu), Adrien Poteaux, Alban Quadrat, Alexandre Sedoglavic, Alexandre Temperville (qui entamait sa dernière année de thèse quand je débutais et qui a ainsi joué un rôle de « grand frère » tout au long de cette première année) et Marie-Emilie Voge (avec laquelle j'ai également eu le plaisir de co-écrire un article). Une petite pensée également pour Eric Wegrzynowski, Léopold Weinberg et tous ceux avec lesquels j'avais l'habitude de manger le midi.

Pendant les deux années suivantes, j'ai eu le plaisir de cotoyer les membres de l'équipe EMSEC de l'IRISA de Rennes. Merci tout d'abord aux permanents : Patrick Derbez et Adeline Roux-Langlois pour leurs remarques et conseils, à Benoît Gérard, Clémentine Maurice et Mohamed Sabt pour leur enthousiasme et leur bonne humeur, merci aussi à Gildas Avoine, Stéphanie Delaune, et Barbara Kordy. Ces deux années n'auraient pas été les mêmes sans les thésards/thésardes et postdocs EMSEC, qui ont grandement contribué à la bonne ambiance de l'équipe. Merci à vous pour tous les bons moments passés ensemble, que ce soit lors des pauses cafés ou des soirées bowling. Plus particulièrement merci à Florent Tardif pour son humour sarcastique, Alban Siffer parce qu'il ne « boit pas l'eau des pâtes », Wojciech Widel for making the effort of learning a bit of french, Alexandre Debant pour son accent toulousain, Angèle Bossuat pour avoir enrichi mon vocabulaire d'onomatopées variées et d'expressions peu communes, Merci également aux « exilés » de l'étage vert avec lesquels j'ai passé le plus de temps dans nos bureaux mitoyens : Pauline Bert pour dire régulièrement des phrases étranges et amusantes sorties de leur contexte, Chen Qian parce qu'il est bizarre et qu'il voit des girafes là où le reste du monde ne voit qu'un trait, Guillaume Kaim pour l'ajout de « petit pain » à l'éternel débat « pain au chocolat » ou « chocolatine », et merci à Baptiste Lambin pour m'avoir supportée comme co-bureau pendant deux ans, merci pour les discussions improbables sur la grammaire française, anglaise et elfique, pour les débats finalement peu constructifs sur l'enseignement, pour tes « là haut » utilisés à tord et à travers... Je pense que ces deux ans auraient été plus longs, si je n'avais pas eu l'occasion de t'embêter régulièrement. Merci aussi à Adina Nedelcu et Céline Duguey, qui sont venues un jour par semaine égayer le bureau de leurs sourires respectifs. J'ai également une pensée pour les anciens thésards/postdocs: Raphael Bost, Pierre Karpman, Pierre Lestringant, Vincent Migliore, Brice Minaud, Cristina Onete (pour les conversations sur la musique ou la littérature), Benjamin Richard (le chauffeur de taxi le moins ponctuel de Rennes mais qui a le mérite d'être gratuit), et Cyrille Wiedling (pour ses gâteaux).

Enfin, j'en profite pour saluer les nouveaux: Adeline Bailly, Katharina Boudgoust, Xavier Bultel, Victor Mollimard, Solène Moreau, Sam Thomas et Weiqiang Wen.

Je remercie aussi Sylvain Duquesne pour avoir fait partie de mon CSID, avec Patrick Derbez.

Je tiens également à saluer mes autres co-auteurs: merci à Jonathan Bootle, Thomas Espitau, Mehdi Tibouchi, et Paul Kirchner pour leurs contributions à la réussite de cette thèse ; ainsi que Bonan Cuan, Aliénor Damien et Mathieu Valois, avec lesquels j'ai de plus passé une semaine à la fois fort sympathique et productive lors de l'évènement REDOCS'17. Ce fut un véritable plaisir de travailler avec vous. J'en profite pour remercier Pascal Lafourcade, pour ses encouragements et ses conseils valables en toutes circonstances.

Merci également Jean-Guillaume Dumas, Damien Stehlé, et Morgan Barbier pour m'avoir permis de présenter mes travaux respectivement sur l'Elimination Gaussienne creuse, SPRING, et le problème 3XOR dans leurs séminaires respectifs. Je remercie aussi la LinBox Team pour m'avoir permis de faire partie de ce projet. Je n'ai malheureusement pas eu (pris) le temps de m'y investir autant que je l'aurais souhaité. Merci également à Gaëtan Leurent pour avoir mis à notre disposition les codes source de SPRING-BCH et SPRING-CRT, qui m'ont été d'une grande aide.

J'en profite pour saluer le corps enseignant de l'Université de Versailles-Saint-Quentin en Yvelines, où j'ai effectué mes études de la L1 jusqu'au Master. Plus particulièrement merci à Luca Defeo pour avoir soutenu ma candidature de stage de M2 à l'Université de Lille, m'aidant ainsi à mettre un pied dans le monde de la recherche.

J'ai aussi une pensée (sans vous nommer car vous êtes nombreux) pour toutes les personnes avec lesquelles j'ai sympathisé lors de conférences ou d'autres déplacements. Merci pour les bons moments passés ensemble.

Pour finir, je voudrais remercier ma famille. En particulier, je remercie ma tante Agnès, pour m'avoir laissée utiliser sa chambre d'amis pendant le temps que j'ai passé à Lille. Enfin, je remercie mes parents, pour m'avoir toujours encouragée, et en particulier pendant ces derniers mois de thèse qui ont été assez éprouvants. Merci à mon père pour m'avoir donné le goût des mathématiques. Merci à ma mère, qui a du supporter régulièrement des discussions mathématiques pendant les repas, et pour avoir pris le temps de relire ces deux-cents pages de blabla technique afin de repérer un maximum de typos. Je ne serais jamais arrivée jusque là sans votre soutien.

Résumé (en français)

Contexte et organisation de cette thèse

Un algorithme est une suite d'instructions qui doivent être suivies afin de retrouver une solution à un problème donné. La complexité en temps d'un algorithme est donnée par le nombre d'étapes nécessaires pour retrouver une solution. La complexité en mémoire est la quantité d'espace nécessaire au déroulement de l'algorithme. Pour un problème donné, il existe autant d'algorithmes qu'il y a de façons de résoudre ce problème, ce qui fait beaucoup, en général. Tous ces algorithmes ne sont pas équivalents en ce sens qu'ils n'ont pas tous la même complexité en temps et en mémoire. Généralement, nous considérons que le « meilleur » algorithme est celui avec la plus petite complexité en temps. Ce n'est pourtant pas toujours le cas, selon ce que nous recherchons. En effet, il existe des algorithmes qui ont la meilleure complexité en temps en théorie, mais qui sont cependant totalement inutilisables en pratique (par exemple [LG14]).

Cette thèse traite principalement de plusieurs problèmes rencontrés en cryptologie et en calcul formel. En m'intéressant aux aspects algorithmiques de ces problèmes, j'ai souvent été confrontée à cet écart entre la théorie et la réalité. Les travaux que j'ai effectués portent surtout sur l'algèbre linéaire, avec des applications en cryptologie et principalement en cryptanalyse.

J'ai tout d'abord travaillé sur des problèmes d'algèbre linéaire creuse exacte. Ce genre de problèmes apparait en particulier comme une sous-routine des algorithmes de factorisation et de logarithmes discrets [CAD15].

En parallèle, avec l'essor de la cryptographie post-quantique, je me suis intéressée à la cryptographie à base de réseaux euclidiens. Plus particulièrement, j'ai participé à la mise en place d'un générateur pseudo-aléatoire basé sur le problème *(Ring) Learning With Roundings* [BPR12]. Plus récemment, je me suis penchée sur une variante du problème *Learning With Errors* sans réduction modulaire.

Enfin, j'ai travaillé sur le problème des anniversaires généralisé. Je me suis intéressée à un cas particulier de ce problème, appelé le problème 3XOR. Ce problème est notamment connu pour avoir des applications dans la cryptanalyse du mode de chiffrement authentifié COPA (voir [Nan15]). En réfléchissant à des moyens d'améliorer les techniques pour résoudre ce problème, j'ai été amenée à regarder des problèmes d'algèbre linéaire sur $\mathbb{F}_2$, certains se ramenant à des problèmes de théorie des codes.

En résumé, on pourrait situer mes travaux de thèse dans trois domaines différents, mais reliés entre eux: (1) l'algèbre linéaire creuse exacte modulo un nombre premier, (2) l'algèbre linéaire exacte sur $\mathbb{F}_2$, (3) l'algèbre linéaire dense avec du bruit. Nous avons la progression suivante:

$$(1) \xrightarrow{\text{matrices creuses} \to \text{aléatoires}} (2) \xrightarrow{\text{exact} \to \text{bruit}} (3)$$

C'est ainsi que j'ai choisi d'organiser cette thèse: du creux vers le dense, de l'exact vers le bruité. Il est à noter cependant que ce choix est arbitraire, et dans les faits, les trois parties de cette thèse peuvent se lire indépendamment les unes des autres.

Contributions de cette thèse

Algèbre linéaire creuse exacte

Pour résoudre un système linéaire dense, calculer le rang ou encore le déterminant d'une matrice dense dans un corps, on utilise généralement la méthode du pivot de Gauss (ou de la factorisation

LU). Etant donné une matrice A de taille $n \times m$, on décompose A de la manière suivante: $A = LU$, où L est trapézoïdale inférieure avec une diagonale non nulle, de taille $n \times r$, U est trapézoïdale supérieure de taille $r \times m$, dont les coefficients sur la diagonale sont tous égaux à 1. r représente ici le rang de la matrice A. Il est également possible de calculer une factorisation PLUQ. Dans ce cas, la matrice A est décomposée en $A = PLUQ$, où L et U sont comme énoncé au dessus, et P et Q sont des matrices de permutations, respectivement des lignes et des colonnes de A.

Pour résoudre le même genre de problème, dans le cas de l'algèbre linéaire creuse, il existe principalement deux familles d'algorithmes : Les méthodes directes et les méthodes itératives. Les méthodes directes sont similaires au cas dense en ce sens où elles visent à construire une version échelonnée de la matrice de départ. Ce processus produit généralement à la sortie une matrice plus dense que celle de départ. Ce phénomène s'appelle le remplissage. Lorsque ce remplissage devient trop important, le processus ralentit et peut même échouer si la mémoire nécessaire n'est pas disponible. Les méthodes itératives, d'un autre côté, sont sûres. La matrice de départ n'est jamais modifiée et les opérations effectuées sont généralement des produits de matrice-vecteur qui ne nécessitent pas beaucoup de mémoire supplémentaire. Leur complexité en temps est également prévisible, de l'ordre de $\mathcal{O}(r\mathrm{nz}(A))$, où r est le rang de la matrice A et $\mathrm{nz}(A)$ le nombre de coefficients non nuls dans A. Cependant, lorsque A est très creuse et/ou structurée elles sont généralement plus lentes que les méthodes directes. Toutefois, ces méthodes représentent souvent la seule possibilité dans le cas des matrices où le remplissage rend les algorithmes directs inutilisables en pratique.

Dans une étude de 2002, Dumas et Villard [DV02] ont testé et comparé différents algorithmes dédiés au calcul du rang de matrices creuses modulo un premier p assez petit (typiquement, les entiers modulo p tiennent sur un mot machine). Plus particulièrement, ils comparent un algorithme de pivot de Gauss creux (méthode directe) avec la méthode de Wiedemann (méthode itérative) sur une collection de matrices que d'autres chercheurs leur ont procurées et qui est disponible en ligne [Dum12]. Ils ont observé que, bien que les méthodes itératives n'échouent jamais et peuvent être assez efficaces, les méthodes directes peuvent être parfois bien plus rapides. C'est particulièrement le cas lorsque la matrice de départ est « presque triangulaire », et que la méthode du pivot de Gauss n'a alors presque rien à faire.

Il s'en suit que les deux méthodes sont valables. En pratique, il est possible de faire la chose suivante: « Essayer une méthode directe, si il y a trop de remplissage, arrêter et recommencer avec une méthode itérative ». Notons également que ces deux méthodes peuvent être combinées. Effectuer k étapes du pivot de Gauss, diminue la taille de la « matrice résiduelle » d'autant, tout en augmentant le nombre de coefficients non-nuls. La stratégie suivante peut alors être considérée: « Tant que le produit (nombre de lignes restantes) $\times$ (nombre de coefficients non-nuls) décroit, effectuer une étape du pivot de Gauss. Lorsque ce produit ne décroit plus, passer à une méthode itérative ». Par exemple, ce genre de méthode a été utilisé par [KAF+10] et [KDL+17] en tant que sous-routine respectivement pour la factorisation d'un entier de 768-bit et le calcul d'un logarithme discret modulo un entier de 768-bit. L'algorithme que nous avons mis en place dans [BD16] se prête également très bien à ce genre d'hybridation.

Mise en place d'un nouvel algorithme de pivot de Gauss creux [BD16]. Le but initial de ce premier article, écrit conjointement avec Charles Bouillaguet, était de vérifier si les conclusions de l'étude [DV02] pouvaient être affinées en utilisant un algorithme de pivot de Gauss creux plus sophistiqué. Pour cela nous nous sommes inspirés des méthodes d'éliminations utilisées dans le monde numérique. Nous nous sommes intéressés en particulier à un algorithme de factorisation PLUQ, dû à Gilbert et Peierls et appelé GPLU [GP88].

Dans un premier temps, nous avons effectué des tests afin de comparer l'algorithme GPLU à l'algorithme du pivot de Gauss existant dans la libraire LinBox [Lin08]. Nos résultats montrent que, bien que l'algorithme GPLU semble être plus souvent plus rapide que l'algorithme de LinBox, il est difficile de départager ces deux algorithmes qui ont chacun leurs points forts et leurs faiblesses. Plus de détails à ce propos sont donnés dans le chapitre 3.

Dans un deuxième temps, nous avons mis au point un nouvel algorithme de factorisation PLUQ s'appuyant sur les avantages des deux algorithmes cités précédemment. Notre algorithme fonctionne

plus ou moins de la manière suivante: Nous permutons au préalable les lignes et les colonnes de la matrice de manière à obtenir une matrice dont la sous-matrice principale sera triangulaire supérieure. Les coefficients sur sa diagonale seront alors choisis comme pivots. Nous nous servons ensuite de ces pivots pour éliminer les coefficients en dessous et nous réitérons la méthode.

Nous avons implémenté cette méthode dans une librairie appelée SpaSM, qui nous a servi à tester nos idées en pratique. Cette librairie est disponible publiquement à l'adresse suivante:

`https://github.com/cbouilla/spasm`

Nous avons comparé cet algorithme à la fois avec l'algorithme du pivot de Gauss implémenté dans LinBox et l'implémentation de l'algorithme GPLU de SpaSM. Il en ressort que notre nouvel algorithme est toujours plus rapide que l'algorithme de LinBox et presque toujours plus rapide que l'algorithme GPLU. De plus, nous l'avons également comparé à la méthode de Wiedemann pour certaines matrices assez larges qui ne pouvaient pas être traitées par une méthode directe avant. Nous obtenons une accélération considérable dans certains cas, notre algorithme pouvant être plus de 100 fois plus rapide en utilisant une méthode de sélection de pivots assez naïve. Ce nouvel algorithme ainsi que ces expériences sont détaillés dans le chapitre 4.

Amélioration des heuristiques de sélection de pivots [BDV17]. Lors du calcul d'une factorisation PLUQ, les permutations P et Q des lignes et des colonnes de A doivent être choisies de manière à ce que les matrices L et U soient les plus creuses possible. Nous prétendons qu'une bonne stratégie de sélection de pivots est essentielle pour réduire le remplissage lors des étapes d'éliminations. Malheureusement, il a été prouvé que trouver la séquence de pivots qui minimise ce remplissage revient à résoudre un problème d'optimisation, dont le problème décisionnel associé est NP-complet [Yan81]. Nous devons alors utiliser des heuristiques qui nous permettent de trouver une « bonne » séquence de pivots.

Dans ce deuxième article, écrit avec Charles Bouillaguet et Marie-Emilie Voge, nous nous sommes focalisés sur les méthodes de sélections de pivots dits « structurels », c'est à dire de pivots qui apparaissent naturellement dans la matrice après permutation des lignes et des colonnes. Il s'agit exactement du genre de pivots que nous recherchons lors de la première phase de l'algorithme décrit au dessus. Dans [BD16] nous nous reposions sur une heuristique simple proposée par Faugère et Lachartre en 2010 [FL10]. Les travaux de [BDV17] consistent à présenter des alternatives à cette méthode. Nous présentons notamment une heuristique gloutonne, facilement parallelisable, qui nous permet de retrouver jusqu'à 99.9 % du nombre total des pivots dans certains cas. Une version parallèle de cet algorithme nous permet de calculer le rang de certaines très grosses matrices de [Dum12] en quelques secondes (quelques minutes pour les plus grosses matrices), alors que l'implémentation (séquentielle) de l'algorithme de Wiedemann disponible dans LinBox met plusieurs heures, voire plusieurs jours à produire le même résultat. Dans le cas le plus extrême, notre algorithme était capable de calculer le rang de la matrice en moins de quatre minutes en utilisant 24 cœurs d'un cluster. Nous avons essayé de calculer le rang de cette même matrice avec l'algorithme (séquentiel) de Wiedemann, mais notre programme a été coupé au bout de 16 jours par les gestionnaires. Enfin, il nous faut mentionner que cette méthode ne fonctionne pas tout le temps. Il y a en effet des matrices de [Dum12] pour lesquelles nous sommes toujours incapables de calculer le rang en utilisant des méthodes directes, et pour lesquelles, l'algorithme de Wiedemann reste la seule solution. Plus de détails sur ces résultats sont donnés dans le chapitre 5.

Ainsi, dans cette partie de ma thèse, je me suis focalisée sur la mise en place d'algorithmes de factorisation LU creuse efficace en pratique. Une grande partie de mon travail a été de réfléchir à des façons d'adapter des méthodes utilisées en analyse numérique au contexte de l'algèbre linéaire exacte. À ce titre, j'ai réfléchi à plusieurs méthodes, que j'ai implémentées dans SpaSM. Une seule s'est finalement avérée suffisamment efficace. J'ai également contribué à la mise au point de nouvelles heuristiques de sélection de pivots, et j'en ai implémenté une décrite en section 5.2.

Problème 3XOR

Le problème des anniversaires est un outil très utilisé en cryptanalyse. Étant donné deux listes $\mathcal{L}_1$ et $\mathcal{L}_2$ de chaines de bits tirées uniformément aléatoirement dans $\{0,1\}^n$, trouver $\mathbf{x}_1 \in \mathcal{L}_1$ et $\mathbf{x}_2 \in \mathcal{L}_2$ tel que $\mathbf{x}_1 \oplus \mathbf{x}_2 = 0$ (le symbole $\oplus$ représente ici l'opération exclusive-OR entre deux chaînes de bits). On peut facilement vérifier qu'une solution existe avec forte probabilité, dès que $|\mathcal{L}_1| \times |\mathcal{L}_2| \gg 2^n$. Cette solution peut être trouvée en temps $\mathcal{O}\left(2^{n/2}\right)$ par des algorithmes assez simples (trier les listes et les parcourir est une possibilité). En 2002, Wagner [Wag02] a proposé une généralisation de ce problème à k listes. Pour un k fixé nous appelons ce problème le problème kXOR. Wagner a montré que le problème 4XOR peut se résoudre en temps et en espace $\mathcal{O}\left(2^{n/3}\right)$ si on prend en entrée des listes de tailles $2^{n/3}$, au lieu de $2^{n/4}$, la taille minimale que les listes doivent avoir afin qu'il y ait une solution avec forte probabilité.

Améliorer les algorithmes pour le problème 3XOR [BDF18]. Dans cet article écrit avec Charles Bouillaguet et Pierre-Alain Fouque, nous nous sommes intéressés au cas du problème 3XOR. Nous supposons qu'il existe trois listes $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3$ qui peuvent contenir un nombre arbitrairement grand d'entrées et chacune de ces entrées est une chaîne de bits tirée de manière uniformément aléatoire et indépendamment des autres dans $\{0,1\}^n$. Le but est alors de trouver $(\mathbf{x}_1, \mathbf{x}_2\mathbf{x}_3) \in \mathcal{L}_1 \times \mathcal{L}_2 \times \mathcal{L}_3$ tel que $\mathbf{x}_1 \oplus \mathbf{x}_2 \oplus \mathbf{x}_3 = 0$. Dans ce cas, l'algorithme de Wagner n'apporte rien de plus que la méthode quadratique « classique » qui consiste à créer toutes les paires d'éléments $(\mathbf{x}_1, \mathbf{x}_2)$ de $\mathcal{L}_1 \times \mathcal{L}_2$ et de vérifier si $\mathbf{x}_1 \oplus \mathbf{x}_2$ est dans $\mathcal{L}_3$.

En 2014 Nikolić et Sasaki [NS15] ont proposé une amélioration de l'algorithme de Wagner, basée sur des résultats de probabilités dus à Mitzenmacher [Mit96]. Leur algorithme est en théorie de l'ordre de $\sqrt{n/\ln(n)}$ fois plus rapide que l'algorithme de Wagner, pour des listes de taille $2^{n/2}\ln(n/2)/\sqrt{n/2}$. Quelques années auparavant cependant, Joux avait déjà proposé un meilleur algorithme [Jou09]. Son idée était de trouver une matrice carrée M de taille n, inversible à coefficient dans $\mathbb{F}_2$ et de résoudre le problème en considérant les listes $\mathcal{L}'_1, \mathcal{L}'_2$ et $\mathcal{L}'_3$, où $\mathcal{L}'_i = \{\mathbf{x}M, \text{ où } \mathbf{x} \in \mathcal{L}_i\}$. En effet, on peut vérifier que si $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$ est solution au problème avec les listes $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3$, alors $(\mathbf{x}_1M, \mathbf{x}_2M, \mathbf{x}_3M)$ est solution au problème avec les listes $\mathcal{L}'_1, \mathcal{L}'_2, \mathcal{L}'_3$. De plus, si M est inversible, il y a équivalence. Joux propose de choisir la matrice M de telle sorte que $n/2$ éléments de $\mathcal{L}_3$ commencent par $n/2$ zéros. Ce genre de matrices se trouve facilement. Il cherche ensuite tous les couples $(\mathbf{x}_1, \mathbf{x}_2)$ de $\mathcal{L}'_1 \times \mathcal{L}'_2$ qui coïncident sur les $n/2$ premiers bits et vérifie si $\mathbf{x}_1 \oplus \mathbf{x}_2$ est dans $\mathcal{L}'_3$. Cette méthode lui permet de gagner un facteur de l'ordre de $\sqrt{n}$ par rapport à la méthode de Wagner, et est donc en ce sens meilleure que la méthode de Nikolić et Sasaki. Combiner ces deux méthodes dans un nouvel algorithme semble être assez difficile.

Indépendamment, Bernstein [Ber07] a proposé un compromis données/mémoire assez simple qu'il a appelé le « Clamping », qui permet de réduire la taille des listes d'entrée et également la longueur n des chaines de caractères considérées. En effet, si les listes de départ sont de taille 2^ℓ où $\ell > n$, il est toujours possible de se ramener à un problème en dimension n' avec des listes de taille $2^{n'/3}$, en supprimant toutes les entrées qui ne commencent pas par k zéros où k est tel que $n' = n - k = (\ell - k)/3$. Plus de détails à propos de ces diverses méthodes sont donnés dans le chapitre 7

Nous pouvons remarquer que la mémoire est souvent le facteur limitant de tous ces algorithmes. À ce titre, nous prétendons que garder les listes les plus petites possibles est la meilleure stratégie à aborder. Le « Clamping » de Bernstein va d'ailleurs dans ce sens. Malheureusement dans le cas où les listes de départ font exactement $2^{n/3}$, le meilleur algorithme semble être l'algorithme quadratique décrit un peu plus haut.

La contribution majeure de notre travail consiste alors à la mise en place d'un nouvel algorithme qui généralise l'idée de Joux à n'importe quelle taille de listes. Cette généralisation consiste à réitérer sa méthode plusieurs fois, en parcourant ainsi toutes les entrées de la troisième liste. De manière succincte, nous sélectionnons une sous-liste arbitraire de taille bien choisie d'éléments $\mathbf{z}$ dans la troisième liste et nous trouvons une matrice M telle que $\mathbf{z}M$ commence par un nombre fixé k de zéros. Nous cherchons ensuite tous les couples $(\mathbf{x}, \mathbf{y})$ tel que $(\mathbf{x} \oplus \mathbf{y})M$ commence par k zéros, nous vérifions ensuite si il existe un $\mathbf{z}$ dans la sous-liste choisie telle que $\mathbf{z} = \mathbf{x} \oplus \mathbf{y}$. Nous

réitérons la procédure avec une nouvelle sous-liste, jusqu'à ce que toutes les entrées de $\mathcal{L}_3$ aient été balayées une fois et une seule. Cette méthode nous permet d'être de l'ordre de n fois plus rapide que l'algorithme quadratique. Nous proposons également des idées afin de réduire le nombre d'itérations (d'un facteur constant) de la procédure. De plus, en utilisant le « Clamping », notre méthode nous permet en théorie de trouver une solution au problème plus rapidement que celle de Nikolić et Sasaki dans les mêmes conditions. Tout ceci est détaillé dans le chapitre 8.

Enfin, il nous faut rappeler qu'en 2005, Baran, Demain et Pătraşcu [BDP05] ont proposé un algorithme générique sous-quadratique pour le problème 3SUM qui étant donné trois ensembles d'entiers A, B, C consiste à trouver un triplet $(a, b, c) \in A \times B \times C$ tel que $a + b = c$. Nous montrons que cet algorithme peut être facilement transposé dans le cas du problème 3XOR. En théorie, il est environ $n^2/\log^2 n$ fois plus rapide que l'algorithme quadratique dans les mêmes conditions. Cependant cet algorithme nous semble difficilement pratique. Nous développons cela dans le chapitre 9.

En résumé, dans cette deuxième partie, j'ai cherché à répondre à la question suivante : « Si quelqu'un veut résoudre le problème 3XOR en pratique, comment devrait-il s'y prendre ? » Afin de répondre à cette question, j'ai commencé par implémenter les différents algorithmes existants (Wagner, Nikolić et Sasaki, et Joux), pour de petites valeurs de n, allant jusqu'à 64. Ceci nous a permis de constater que ce qui réclame plus de mémoire que strictement nécessaire devient très vite difficile à utiliser en pratique. Partant de là, j'ai réfléchi à des méthodes capables de résoudre ce problème dans le cas où la taille des listes doit être minimale. L'algorithme présenté dans le chapitre 8 est l'aboutissement de ce travail de recherche.

Problèmes liés à *Learning With Errors*

Le problème *Learning With Errors* (LWE) proposé par Regev en 2005 [Reg05] est un problème fondamental de la cryptographie à base de réseaux euclidiens. Ce problème peut-être vu dans un contexte d'algèbre linéaire de la manière suivante : étant donnés une matrice A de taille $m \times n$, $m > n$ et un vecteur $\mathbf{c}$, à coefficient dans $\mathbb{Z}/q\mathbb{Z}$, le but est de trouver $\mathbf{s}$ tel que $A\mathbf{s} + \mathbf{e} = \mathbf{c} \pmod q$, où $\mathbf{e}$ est un « petit » vecteur d'erreur. Ce problème est prouvé aussi difficile que certains problèmes difficiles de réseaux euclidiens, pour des paramètres bien choisis. Il existe plusieurs variantes de ce problème (par exemple Ring-LWE, Module-LWE), où la matrice A est plus ou moins structurée. D'autres variantes consistent à imposer certaines conditions sur le secret $\mathbf{s}$ (par exemple binaire, creux), ou sur l'erreur.

Nous nous intéressons ici à deux autres variantes un tant soit peu différentes. La première variante que nous considérons est appelée *Learning With Rounding* (LWR). Le problème LWR a été proposé par Banerjee, Peikert et Rosen en 2012 [BPR12] comme une version dé-randomisée de LWE. Le système considéré est alors légèrement différent : étant donnés une matrice A de taille $m \times n$, $m > n$ et un vecteur $\mathbf{c}$ à coefficients dans $\mathbb{Z}/q\mathbb{Z}$ le but est de trouver $\mathbf{s}$ tel que : $\lfloor A\mathbf{s} \rceil_p = \mathbf{c} \pmod p$, où $\lfloor \cdot \rceil_p$ peut être vue comme la fonction qui retourne les $\log p$ premiers bits des coefficients du vecteur en argument.

L'autre est une variante de LWE sans réduction modulaire, que nous appelons ILWE pour Integer-LWE. Le système considéré sur $\mathbb{Z}$ est alors $A\mathbf{s} + \mathbf{e} = \mathbf{c}$, où l'erreur $\mathbf{e}$ peut être assez grande. Ce problème apparait naturellement dans certains domaines tels que l'apprentissage statistique ou l'analyse numérique et semble avoir moins d'applications en cryptographie. Nous pouvons cependant en citer une, et pas des moins intéressantes. En effet, comme cela a déjà été évoqué dans [EFGT17], il est possible, après avoir récupéré des informations complémentaires par canaux auxiliaires, de retrouver entièrement la clé secrète du schéma de signature BLISS [DDLL13] en résolvant un tel système. Rappelons que si BLISS ne fait pas partie des candidats à la compétition du NIST, il reste malgré tout un des schémas de signature post-quantique assez répandu: en effet, il existe plusieurs implémentations, sur différentes plateformes tel que des FPGA et microcontrôleurs. BLISS a également été implémenté dans la suite VPN strongSwan.

Construction d'un nouveau PRG basé sur LWR. [BDFK17] Les auteurs de [BPR12] ont prouvé que résoudre LWR pour des paramètres bien choisis est au moins aussi difficile que résoudre

LWE, et se ramène donc à des problèmes difficiles de réseaux euclidiens. Partant de cette hypothèse, ils ont créé une famille de fonctions pseudo-aléatoires, connue sous le nom de BPR, dont la sécurité est prouvée, en admettant la difficulté de LWE. Cette famille est cependant non pratique en raison de la taille de ses paramètres. En 2014, les auteurs de [BBL+14] ont proposé de réduire la taille des paramètres afin d'avoir une famille de fonctions pseudo-aléatoires efficaces (dont le temps d'exécution est comparable à celui d'AES). Cette famille s'appelle SPRING pour « Subset Product with Rounding over a rING ». Les réductions de sécurité de BPR ne s'appliquent plus dans le cas de SPRING, les paramètres étant trop petits. Dans ce travail en commun avec Charles Bouillaguet, Pierre-Alain Fouque et Paul Kirchner, nous proposons un générateur pseudo-aléatoire dans la lignée de SPRING, plus rapide que les instances proposées par les auteurs de [BBL+14]. Nous présentons également une analyse de sécurité de notre primitive et proposons des idées pour réduire la taille des clés de SPRING, qui reste l'un des principaux inconvénients de ce type de construction, enfin, nous présentons une implémentation optimisée de notre schéma, sur Intel en utilisant les instructions SSE2 et AVX2, ainsi que sur ARM en utilisant les instructions NEON. Le code de ces implémentations est disponible publiquement à cette adresse :

`https://github.com/cbouilla/spriiiiiiiing`

Cette construction est détaillée dans le chapitre 12.

LWE sur les entiers [BDE+18]. Dans cet article en collaboration avec Jonathan Bootle, Thomas Espitau, Pierre-Alain Fouque et Mehdi Tibouchi, nous présentons le problème ILWE, pour Integer-LWE, et nous discutons de sa difficulté. C'est sans grande surprise que nous le trouvons beaucoup plus facile à résoudre que LWE, et ce, même lorsque l'erreur $\mathbf{e}$ est grande (mais pas super-polynomiale en la variance de A). Nous montrons que d'un point de vue théorie de l'information, le nombre de lignes minimal m que A doit avoir pour que le problème soit faisable est de l'ordre de $\Omega\left(\sigma_e^2/\sigma_a^2\right)$, où σ_a^2 représente la variance de A et σ_e^2 est la variance de l'erreur.

Le meilleur algorithme que nous proposons pour résoudre ce problème est la méthode des moindres-carrés. Dans ce cas précis, la valeur minimale de m est de l'ordre de $\Omega\left(\max((\sigma_e^2/\sigma_a^2)\cdot\log n, n)\right)$. Nous proposons également une méthode qui, d'après les travaux de Candes et Tao [CT07], permet de retrouver $\mathbf{s}$ dans le cas où $m < n$, lorsque $\mathbf{s}$ est très creux et de densité connue.

Notre principale motivation concernant ce problème était d'améliorer l'attaque proposée par les auteurs de [EFGT17]. Dans ce papier, les auteurs remarquent qu'il est possible de récupérer deux fonctions du secret $\mathbf{s}$ de BLISS, par canaux auxiliaires. L'une d'entre elles est quadratique, l'autre linéaire mais bruitée. Les auteurs de [EFGT17] ont alors rejeté l'idée d'exploiter cette dernière information, car cela reviendrait à résoudre un problème analogue à LWE en grande dimension, et se sont focalisés sur la fonction quadratique. Cependant, le système linéaire bruité considéré est en réalité une instance ILWE, et est donc facile à résoudre en utilisant les techniques énoncées ci-dessus. De plus, il s'applique à toutes les clés de BLISS (L'attaque de [EFGT17] ne s'applique qu'à 7% des clés).

Nous proposons également des tests afin de corroborer les résultas énoncés plus haut. En particulier, nous proposons une simulation de l'attaque par canaux auxiliaires contre BLISS. En utilisant la méthode des moindres-carrés, nous sommes capable de retrouver les secrets de BLISS–0, BLISS–I et BLISS–2 en quelques minutes. En comparaison l'attaque de [EFGT17] peut prendre plusieurs jours CPU. Le seul point noir est qu'il faut récupérer un nombre considérable de traces par canaux auxiliaires. Ces travaux sont détaillés dans le chapitre 14.

En résumé, dans cette troisième partie j'ai regardé deux aspects de la cryptologie basée sur le problème LWE. J'ai commencé par contribuer à la construction d'une nouvelle primitive de cryptographie symétrique, que j'ai implémentée sur Intel avec instructions SSE2 et sur ARM avec instructions NEON, en m'inspirant du code des auteurs de [BBL+14]. Dans un second temps, j'ai réfléchi sur des méthodes pour résoudre le problème LWE sur les entiers, afin d'améliorer des attaques par canaux auxiliaires contre une signature post-quantique. Plus précisément, j'ai regardé comment modéliser ce problème en programmation linéaire et en programmation linéaire entière.

Pour ce faire, j'ai créé plusieurs modèles sous Gurobi [GO18]. La méthode des moindres carrés semble cependant s'avérer être la plus efficace pour résoudre ce problème.

Autres travaux

En parallèle des travaux liés à ma thèse, j'ai eu l'occasion de m'intéresser à la détection de logiciels malveillants dans des fichiers PDF, en utilisant des méthodes d'apprentissage automatisé (machine-learning en anglais). Il s'agit d'un travail commun avec Bonan Cuan, Aliénor Damien et Mathieu Valois, accepté en tant que « position paper » à SECRYPT2018. Il a été réalisé dans le cadre de l'évènement REDOCS organisé par le pré-GDR sécurité.

Détection de logiciel malveillant dans des PDF [CDDV18]. Des milliers de fichiers PDF sont disponibles sur le web. Ils ne sont pas tous aussi inoffensifs qu'ils paraissent. En réalité, les PDF peuvent contenir divers objets tel que du JavaScript et/ou du code binaire. Il arrive que ces objets soient malicieux. Un PDF peut alors essayer d'exploiter une faille dans le lecteur afin d'infecter la machine sur laquelle il est ouvert. En 2017, soixante-huit failles ont été découvertes dans Adobe Acrobat Reader [CVE17]. Plus de cinquante d'entre elles peuvent être exploitées pour faire tourner un code arbitraire. Il est a noter que chaque lecteur de PDF a ses propres vulnérabilités, et un PDF malicieux peut trouver un moyen d'en tirer parti. Dans ce contexte, plusieurs travaux ont proposé d'utiliser des techniques dite d'*apprentissage automatisé* (machine learning en anglais) afin de détecter les PDF malicieux ([Kit11, MGC12, Bor13] par exemple). Ces travaux reposent plus ou moins sur la même idée: repérer des caractéristiques dites « discriminantes » (c-à-d qui apparaissent le plus souvent dans des PDF malicieux) et trier les PDF en utilisant un algorithme de classification. Pour un PDF donné, cet algorithme de classification prendra en entrée les caractéristiques sélectionnées et, en considérant le nombre d'occurences de chacune d'entre elles, déterminera si le PDF est sain ou si il peut être malveillant. Dans ce travail, nous nous sommes focalisés sur des méthodes *d'apprentissage supervisé* : l'algorithme de classification est au préalable entrainé avec des PDF dont l'état (sain ou malveillant) est connu. Une phase de test est ensuite réalisée, afin de vérifier la précision des prédictions effectuées par l'algorithme. Plusieurs approches ont été considérées, à la fois dans le choix des caractéristiques discriminantes et dans le choix de l'algorithme de classification lui-même. En réalité beaucoup d'algorithmes de classification peuvent être utilisés: NaiveBayes, DecisionTree, RandomForest et SVM en sont des exemples. Les auteurs de [MGC12] commencent par décrire leur propre manière de sélectionner les caractéristiques discriminantes et utilisent ensuite un algorithme RandomForest pour la classification, alors que Borg [Bor13] se repose sur le choix de caractéristiques discriminantes proposé par Stevens [Ste06] et utilise un algorithme SVM (Support Vector Machine) pour la classification. Les deux approches semblent donner des résultats précis.

Cependant, il est toujours possible de contourner ce genre d'algorithmes de détection. Plusieurs attaques ont été proposées à cet effet (par exemple [AFM+13, BCM+13]). Les auteurs de [BCM+13] proposent une attaque par descente de gradient dans le but d'échapper à la vigilance des SVM et des algorithmes de classification basés sur des réseaux de neurones. D'un autre côté, les auteurs de [AFM+13] expliquent comment il est possible d'obtenir des informations sur l'ensemble d'entrainement d'un algorithme de classification cible. Pour ce faire, ils utilisent un autre algorithme de classification, qui agit sur un ensemble d'algorithmes de classification, entrainés avec différents ensembles de données. Leur but est de détecter ainsi des propriétés intéressantes dans l'ensemble d'entrainement utilisé par l'algorithme de classification cible et de tirer parti de cette connaissance pour attaquer l'algorithme de classification en question.

Dans [CDDV18], nous présentons trois aspects de la détection de logiciels malveillants dans les PDF. La première partie de notre travail a consisté à implémenter notre propre algorithme de classification. Nous avons utilisé un SVM, car cet algorithme produit de bons résultats tout en restant assez simple. Nous avons exploré différentes possibilités concernant le choix des caractéristiques discriminantes. Notre premier choix était basé sur la sélection proposée par Stevens [Ste06]. Nous avons par la suite affiné ce choix en sélectionnant également les caractéristiques qui, relativement à notre ensemble de données, nous paraissaient les plus discriminantes. Nous avons entraîné et testé

notre SVM avec un ensemble de 10 000 PDF sains et 10 000 PDF malicieux provenant de la base de donnée Contagio [Con13]. Nous avons ainsi obtenu un algorithme de classification dont le taux de succès était supérieur à 99%.

Dans un second temps, nous avons cherché à contourner notre SVM en forgeant des PDF malicieux de différentes manières, la plus prometteuse étant une attaque de type descente de gradient très similaire à celle proposée par les auteurs de [AFM+13].

Enfin, nous avons réfléchi à la façon d'éviter ce type d'attaques. Nous proposons trois méthodes. La première consiste à instaurer un seuil pour chaque caractéristique, empêchant ainsi à un adversaire d'augmenter considérablement le nombre d'objets. Nous avons également proposé un choix de caractéristiques plus approprié, afin de rendre notre SVM plus résistant aux attaques de type descente de gradient. Enfin, nous avons proposé de ré-entraîner notre SVM avec des PDF forgés. Ces diverses contremesures nous ont permis de bloquer jusqu'à 99.99% des attaques par descente de gradient.

Liste de mes publications

[BD16] **Sparse Gaussian Elimination Modulo p: An Update**
co-écrit avec Charles Bouillaguet (CASC 2016)

[BDFK17] **Fast Lattice-Based Encryption: Stretching SPRING**
co-écrit avec Charles Bouillaguet, Pierre-Alain Fouque et Paul Kirchner (PQCrypto 2017)

[BDV17] **Parallel Sparse PLUQ Factorisation Modulo p**
co-écrit avec Charles Bouillaguet et Marie-Emilie Voge (PASCO 2017)

[BDF18] **Revisting and Improving Algorithm for the 3XOR problem**
co-écrit avec Charles Bouillaguet et Pierre-Alain Fouque (TosC/FSE 2018)

[CDDV18] **Malware Detection in PDF files using Machine Learning**
co-écrit avec Bonan Cuan, Aliénor Damien et Mathieu Valois (position paper accepté à SECrypt 2018)

[BDE+18] **LWE without Modular Reduction and Improved Side-Channel Attacks against BLISS**
co-écrit avec Jonathan Bootle, Thomas Espitau, Pierre-Alain Fouque et Mehdi Tibouchi (accepté à ASIACRYPT2018)

"All computer science is algorithm."

– Donald Knuth –

Contents

Chapter 1

Foreword

In this chapter, we present some important definitions and notations that will be used throughout this thesis.

1.1 Mathematical Notations and Basic Definitions.

Useful notations. We denote by $\oplus$ the bitwise exclusive-OR (XOR) operation. The base-2 logarithm of some value x is denoted by $\log(x)$ or $\log x$. The natural logarithm of x is denoted by $\ln(x)$ or $\ln x$. If we denote by A a list or a table, the i-th element of this list/table is denoted by $A[i]$. If A is empty we note $A = \perp$, let S be a finite set. We denote by $|S|$ its cardinality.

Set and algebraic structures. We denote by $\mathbb{Z}$ the set of integers, $\mathbb{N}$ the set of positive integers, by $\mathbb{R}$ the field of real numbers and by $\mathbb{C}$ the field of complex numbers. Given a positive integer a, we denote by $\mathbb{Z}_a$ the ring $\mathbb{Z}/a\mathbb{Z}$ of integers modulo a. Let p be a prime number. We denote by $\mathbb{F}_p = \mathbb{Z}_p$, the finite field with p elements. Let n be a positive integer, we denote by $\mathbb{F}_p^n$ the n-dimensional vector space over $\mathbb{F}_p$. Given a ring R, we denote by R^* the group of invertible elements of R.

Vectors and vector spaces. Let $\mathbb{F}_p^n$ be a vector space over $\mathbb{F}_p$, and let k be a positive integer such that $1 \leq k \leq n$. A vector $\mathbf{v}$ of $\mathbb{F}_p^n$ is written in bold. The first coefficient of a vector is usually indexed by 0. Let $\mathbf{v}$ be a vector, we denote either by $\mathbf{v}[i]$ or by v_i the coefficient of $\mathbf{v}$ indexed by i. We denote by $\mathbf{v}_i[j]$ the coefficient indexed by j of a vector $\mathbf{v}_i$. Unless stated otherwise, vectors are row vectors. Given two vectors $\mathbf{a}$ and $\mathbf{b}$ of $\mathbb{F}_p^n$, we denote by $\langle \mathbf{a}, \mathbf{b} \rangle = \sum_i a_i b_i$ the *inner product* or *dot product* of $\mathbf{a}$ and $\mathbf{b}$. Given two vectors $\mathbf{a}$ and $\mathbf{b}$, both with coefficients in $\mathbb{F}_p$, possibly of different length, we denote by $\mathbf{a}|\mathbf{b}$ the concatenation of $\mathbf{a}$ and $\mathbf{b}$.

The vector subspace $\mathscr{V}$ of $\mathbb{F}_p^n$ spanned by $\{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ is denoted by $\mathscr{V} = \mathsf{Span}\{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$. Let $\mathbb{F}_p^n$ be a vector space. For all $1 \leq i \leq n$, $\mathbf{e}_i$ is the vector of $\mathbb{F}_p^n$ such that the i-th coefficient of $\mathbf{e}_i$ is 1, and all the other ones are 0. $\mathscr{B} = (\mathbf{e}_1, \ldots, \mathbf{e}_n)$ is the canonical basis of $\mathbb{F}_p^n$. We recall that a vector subspace of $\mathbb{F}_p^n$ with the hamming metric is called a *linear code*. In this thesis, we consider only *binary linear code* (i.e. $p = 2$). A binary linear code is usually referred only as a "code" for simplicity. A code $\mathscr{C}$ of length n and dimension k is called a $[n, k]$-code. If its minimum distance d (i.e. the Hamming weight of the non-zero vector which has the smallest Hamming weight) is known, $\mathscr{C}$ can also be referred as a $[n, k, d]$-code.

Matrices. Let A be an n-by-m matrix over $\mathbb{F}_p$. The first row (resp. column) of A is usually indexed by 0. We denote a_{ij} the coefficient on row i and column j, and by A_{ij} some sub-matrix of A indexed by i and j. The *rank* of A is the number of rows (resp. columns) of A that are linearly

independent. An n-by-n matrix is said to be *square*. Unless stated otherwise, we denote by $\mathbf{a}_i$ the row of A indexed by i, and by ${}^t\mathbf{a}_j$ the column of A indexed by j.

A square matrix A, such that $a_{ij} = a_{ji}$ for all (i, j) is said to be *symmetric*. We denote by ${}^tA$ the *transpose matrix* of A. We call *lower trapezoidal* any n-by-m matrix L whose coefficients l_{ij} with $j > i$ are zeroes. Equivalently, an n-by-m matrix U is called *upper trapezoidal* if all its coefficients u_{ij} with $j < i$ are zeroes.

A matrix which is both lower and upper trapezoidal is called a *diagonal matrix*. For instance the *identity matrix* of size n, is the square diagonal matrix, with n rows and columns, whose coefficients on the diagonal are 1.

Let A be an n-by-m matrix over $\mathbb{F}_p$. The *minimum polynomial* of A is the monic polynomial (i.e. single-variable polynomial whose leading coefficient is 1) μ over $\mathbb{F}_p$ of least degree such that $\mu(A) = 0$. Let $\mathbf{v}$ be a vector of $\mathbb{F}_p^n$ and λ be a an element of the algebraic closure of $\mathbb{F}_p$, such that $\lambda {}^t\mathbf{v} = A {}^t\mathbf{v}$. We call $\mathbf{v}$ *eigenvector* of A and λ *eigenvalue* of A associated to $\mathbf{v}$. The *determinant* $\det(A)$ of A is equal to the product of the eigenvalues.

Bit Strings. A bit is an element of $\{0, 1\}$. Let n be a positive integer. A bit-string $\mathbf{s}$ of length n is a vector of n bits. The set of all n-bit vectors can be seen as the vector space $\mathbb{F}_2^n$, with the XOR ($\oplus$) being the sum. As we represent bit-strings as vectors, we choose to index by 0 the least significant bit (LSB). For instance the 4-bit string associated to 13 is: 1011.

Probability. Let A be an event. The probability that A happens is denoted by $\mathbb{P}[A]$. Let X be a discrete random variable that can take values in a discrete set S. When defined, the *expected value* of X, denoted $\mathbb{E}[X]$, is given by the sum $\mathbb{E}[X] = \sum_{k \in S} k\mathbb{P}[X = k]$; when defined, the *variance* of X, denoted $\mathrm{Var}(X)$, is given by $\mathbb{E}[X^2 - \mathbb{E}[X]^2]$; the *standard deviation* of X, denoted by $\sigma(X)$ is given by $\sqrt{\mathrm{Var}(X)}$. Given a parameter $a > 0$, we recall that by *Markov's inequality* we have:

$$\mathbb{P}[X \geq a] \leq \frac{\mathbb{E}[X]}{a}.$$

It follows *Chebychev's inequality*:

$$\mathbb{P}[|X - \mathbb{E}[X]| \geq a] \leq \frac{\mathrm{Var}(X)}{a^2},$$

and *Chernoff generic bounds*:

$$\mathbb{P}[X \geq a] \leq \frac{\mathbb{E}[e^{tX}]}{e^{ta}}, \quad \forall t > 0.$$

We have a special case when X is the sum of n independent random variables from $\{0, 1\}^n$. If we denote by μ the mean of X, we have:

$$\mathbb{P}[X \geq (1 + \delta)\mu] \leq e^{-\frac{-\delta^2}{2+\delta}\mu}, \quad \forall \delta > 0$$
$$\mathbb{P}[X \leq (1 - \delta)\mu] \leq e^{-\frac{\mu\delta^2}{2}}, \quad \forall 0 < \delta < 1$$

Given a probability distribution χ, we call σ its *standard deviation* and μ its *mean*. If $\mu = 0$, χ is said to be centred.

Asymptotic notations. We use the standard Landau notations: $\mathcal{O}, \Omega, \Theta, o, \omega$. We use the $\tilde{\mathcal{O}}$ notation to hide poly-logarithmic factors. In other words:

$$f(n) = \tilde{\mathcal{O}}\left(g(n)\right) \iff \exists c \in \mathbb{N}, \text{ such that } f(n) = \mathcal{O}\left(g(n) \log^c n\right).$$

We also use the following notations:

$$f(n) = \mathsf{poly}\left(n\right) \iff \exists c \in \mathbb{N}, \text{ such that } f(n) = \mathcal{O}\left(n^c\right),$$

and

$$f(n) = \mathsf{negl}\left(n\right) \iff \exists c \in \mathbb{N}, \text{ such that } f(n) = o(n^{-c}).$$

1.2 Cryptologic Background.

Basics. Strictly speaking *cryptography* aims to construct schemes that provide secure communication in presence of a third party called adversary. *Cryptanalysis* aims to attack these schemes, and point out their weakness. *Cryptology* is the study that regroups both cryptography and cryptanalysis. In this thesis, however, we use the word "cryptography" as a synonym of "cryptology", which is quite common. Cryptography has many goals, the first one being assuring *confidentiality* of a communication (i.e. a third cannot have read the messages), but also *integrity* (i.e. a message cannot by modified by a third party) and *authenticity* (i.e. the origin of the message is certified).

We can divide modern cryptography in two families: *symmetric cryptography* and *asymmetric cryptography*. In symmetric cryptography, two parties – let us call them Alice and Bob – that aims to communicate share a common secret information, called *secret key* that they use to encrypt message and/or authenticate them. In asymmetric cryptography, Bob has a pair of keys $(\mathsf{sk}, \mathsf{pk})$, where sk is his private key, and is known only by him, and pk is his public key, which can be known by anyone. Now, if Alice wants to send a message to Bob, she uses pk to encrypt the message, and only Bob, who knows sk, is able to decrypt it. Bob can also use his secret key to sign a message, and Alice (or anyone who has access to the message) can check that Bob is indeed the sender of the message using pk.

Asymmetric cryptography is based on hard problems, and is slow, but its security can be proven. It is in practice not used to communicate but more to exchange keys in a secure way, or prove one's identity. Symmetric cryptography on the other hand is fast, but its security usually relies on the fact that no attack is known. It is used to encrypt longer messages.

In this thesis, we are more interested by underlying problems we may encounter in cryptography, than these practical considerations, so we will not develop any further.

Cryptographic problems. As mentioned above, symmetric cryptography is usually not based on hard problems. It is still hard to break because no public information related to the secret is known. The attacks consist mostly in finding correlations between different ciphers and/or messages and try to guess informations on the secret key from that. As such one of the most important problem in symmetric cryptography is the *birthday problem*: given a function f that produces random outputs find two different inputs x and y, such that $f(x) = f(y)$. It is well known that this problem can be solved in $\mathcal{O}(2^n/2)$, where n is the bit-length of any output of f. The second part of this thesis is dedicated to a variant of this problem.

Asymmetric cryptography on the other hand relies on somehow hard mathematical problems. We can cite for instance the famous RSA scheme [RSA78], which relies on the hardness of factoring an integer $n = pq$, where p and q are large primes, or the Diffie-Helman protocol [DH76], which relies on the hardness of solving the discrete logarithm problem in a finite group. However, with the threat of the upcoming quantum computer, these two problems may one day become easy, and all cryptography based on them insecure.

To remedy this situation, post-quantum cryptography has been developed in the last few years. It regroups all the schemes that are, for now, assumed to be secured against quantum algorithms. We can distinguish five main families of post-quantum schemes: Lattice-based cryptography, code-based cryptography, multivariate cryptography, isogeny-based cryptography, and hash-based cryptography. In the third part of this thesis we will talk more about hard problems we may encounter in Lattice-based cryptography: in particular, the LWE problem.

1.3 Algorithms and Complexity

Decision problems and complexity classes. We call *decision problem* a problem whose expected answer is either yes or no, depending on input values. A decision problem is said to be *decidable* if there exists an algorithm that always terminates and returns a correct answer, "yes" or "no", to said problem.

A decision problem $\mathcal{P}$ belongs to the *NP class* (Non deterministic Polynomial time), if there exists a polynomial time *verifier* to the problem when the answer is "yes". In other words, given an

instance X of $\mathcal{P}$, such that the answer to $\mathcal{P}$ is "yes", there exists a certificate c such that given X and c, V returns "yes" in polynomial time. Furthermore, if c' is not a valid certificate, V will return "no" when given as input X and c', and this even if the actual answer to $\mathcal{P}$ is "yes". Similarly a problem is said to belong to the *co-NP class* if $\mathcal{P}$ is decidable, and if there exists a polynomial time verifier to the problem when the answer is "no". A problem $\mathcal{P}$ belongs to the *P class*, if it can be solved in polynomial time by a deterministic algorithm.

A problem $\mathcal{P}$ is said to be *NP-Complete*, if $\mathcal{P}$ is in NP, and any other problem in NP is *reducible* to $\mathcal{P}$ in polynomial time. Basically, this means that any instance of any problem $\mathcal{Q}$ in NP can be transformed into an instance of $\mathcal{P}$ in polynomial time. We can define *co-NP-Completeness* in a similar way.

A problem $\mathcal{P}$ is said to be *NP-Hard*, if $\mathcal{P}$ is a decision problem, and is at least as hard to solve as any problem of the NP class. Note that $\mathcal{P}$ is not necessarily in NP. In fact, it is not even required that $\mathcal{P}$ is decidable.

The list of complexity classes given here is far from being an exhaustive one. However, this is enough for the rest of this thesis.

Search and optimisation problems. A *search problem* consists in finding a solution that satisfies some conditions on the input value, if this solution exists. The *associated decision problem* would be to determine whether the solution exists. Given a search problem which has many solutions, an *optimisation problem* consists in finding the "best" one among all of them. More precisely, given an instance X of a search problem $\mathcal{P}$, and given S the set of all possible solutions to $\mathcal{P}$, the goal is to find an element $s \in S$ that either minimise or maximise a given measure of s. The *associated decision problem* to an optimisation problem would be, to determine if there exists a solution $s \in S$ such that the measure is smaller (resp. greater) than a given bound k.

We can extend the notion of NP-Completeness and NP-Hardness to optimisation problems as follows: we say a problem $\mathcal{P}$ is an *NP-Complete* (resp. NP-Hard) *optimisation problem* if the associated decision problem is NP-Complete (resp. NP-Hard).

Remark 1. This is abusing the definitions of NP-Completeness and NP-Hardness. Indeed, only decision problems can be called either NP-Complete or NP-hard. However, these definitions, although formally incorrect, are pretty convenient and appear to be widely utilised in the literature (e.g. [RT75, Yan81, GHL01]).

NP-Complete/NP-Hard optimisation problems and approximations. When an optimisation problem is considered hard, it is usual not to search for *the best* solution, but to settle for *a good* solution instead. Sometimes it is possible to find an approximate solution to one of these problems with a proof that the solution is "close" to the optimal one, meaning that the distance between the solution returned in the worst case and the optimal solution is proved to be smaller than some specific bound. Problems which can be approximated this way are said to be *approximable*. An algorithm which can find such a solution is called *approximation algorithm*.

Some problems are easy to approximate within a constant multiplicative factor, other are proved to be impossible to approximate within constant or even polynomial multiplicative factors unless $P = NP$. In this case, we have to search for other means to find "good" solutions to a problem. We use *heuristics*. A heuristic is also a way to find an approximate solution to a given optimisation problem, but unlike approximation algorithm, we have no guarantee on the quality of the solution thus found. It may be very close to the optimal if we are lucky, but it can also be very far.

PART

1

Sparse Gaussian Elimination Modulo p

Introduction

"The enchanting charms of this sublime science reveal only to those who have the courage to go deeply in it."

– Carl Friedrich Gauss –

Sparse Linear Algebra

A sparse matrix is essentially a matrix in which a large proportion of the coefficients are zeroes. An example of a sparse matrix is given in Figure 1.1: The black pixels here represent non-zero coefficients and the white ones are zeroes.

Given a matrix A, one may want to compute its rank, or solve a linear system $\mathbf{x} \cdot A = \mathbf{b}$, or compute its determinant. In dense linear algebra, we would usually solve these problems using *gaussian elimination*.

Our concern is to solve these problems, when the input matrix is sparse (i.e. if most of its coefficients are zero). In this case, there are essentially two families of algorithms to solve these problems: iterative methods and direct methods.

Iterative Methods and Direct Methods

Iterative methods. Iterative methods are widely used in exact linear algebra, as they allow to handle the largest instances of the problems mentioned above. They work mostly by computing vector-matrix products, and the matrix A is never changed. Their main advantage is that they always terminate and return the desired solution, and that their space complexity is rather small, as they require only to store the matrix plus two vectors. Their execution time is easy to predict, and they are parallelisable up to a certain point. Their main draw back is that they require to perform on average $\mathcal{O}(n)$ matrix vector products, whose time complexity is proportional to the number of non-zero coefficients of A. For very sparse or/and structured matrices, this is usually too much. The most famous of these methods are the Wiedemann Algorithm [Wie86] and its parallel version Block-Wiedemann [Cop94].

Direct methods. Direct methods (e.g. Gaussian Elimination), usually work by computing an echelonised version of the input matrix A, in order to obtain a triangular matrix in its stead. The main issue of these methods, is that the triangular matrix thus obtained is often much denser than the original one. In fact, during the echelonisation process, some entries that were originally zeroes become non-zero ones. This phenomenon is called fill-in. Because of this, it is quite hard to evaluate the time complexity of these methods, and the space complexity is usually a limiting factor. Because of the fill-in, these methods can become very slow, and sometimes, we may run out of memory, which causes the procedure to fail. This is why they are not very populated in the world of exact linear algebra.

In a paper from 2002, Dumas and Villard [DV02] surveyed and benchmarked algorithms dedicated to rank computations for sparse matrices modulo a small prime number p. In particular,

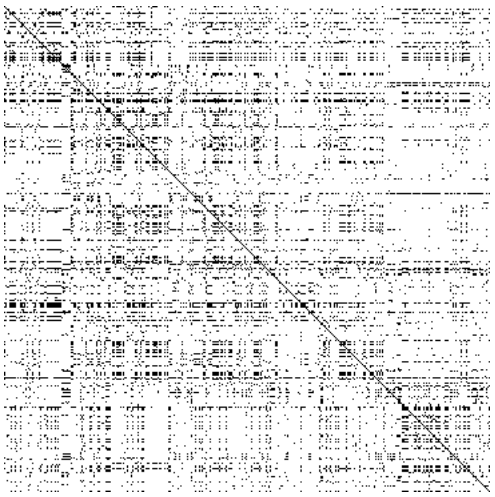


Figure 1.1 – An example of sparse matrix: `GAG/mat364` from [Dum12].

they compared the efficiency of sparse gaussian elimination and the Wiedemann algorithm on a collection of benchmark matrices that they collected from other researchers and made available on the web [Dum12]. They observed that while iterative methods are fail-safe and can be practical, direct methods can sometimes be much faster. This is in particular the case when matrices are almost triangular, so that gaussian elimination barely has anything to do.

Our concern is with direct methods. We aim to develop new algorithms to reduce this fill-in, and thus increase their chance of success. We focussed on the topic of sparse gaussian elimination as these algorithms can be applied to any matrix, square or not, and possibly rank-deficient.

Sparse Gaussian Elimination

There are essentially two ways to compute a sparse gaussian elimination. The first is very similar to the dense algorithm: at each step one pivot is chosen, and the coefficients below are eliminated. This requires to update the matrix every time. The second one works by solving a sequence of triangular systems. The choice of the pivot is made afterward "in the dark". We found out that both of these methods have their own strengths and weaknesses, and none of them is clearly better than the other.

Pivot selection heuristics. A good choice of pivot is primordial to maintain sparsity during the echelonisation process. Unfortunately, finding the pivots that will minimise the fill-in is known to be an NP-Complete optimisation problem. Sparse gaussian elimination methods then use heuristics to select hopefully good pivots. These heuristics are called pivoting strategies.

Our Contributions

In [BD16], our original intention was to check whether the conclusions of the survey of Dumas and Villard [DV02] could be refined by using more sophisticated sparse elimination techniques from the numerical world. To do so, we developed the SpaSM software library (SPArse Solver Modulo p). Its code is publicly available in a repository hosted at:

`https://github.com/cbouilla/spasm`

We mostly focused on the computation of the rank because it raises the same challenges as solving linear systems while being slightly simpler. Our algorithm can nonetheless be easily adapted to other linear algebra problems (e.g. linear system solving, determinant computation).

We implemented the algorithms described in [BD16] and [BDV17] as well as some other – less successful – ones, in this library. This was necessary to test their efficiency in practice. We benchmarked them using matrices from Dumas's collection [Dum12], and compared them with

the algorithms used in [DV02], which are publicly available inside the LinBox [Lin08] library. This includes a sparse gaussian elimination, and the Wiedemann algorithm. We obtained large speedups on some cases, in particular the pivot selection heuristic we developed in [BDV17] allowed us to compute the rank of some of the largest matrices from [Dum12], in a few minutes. Before that, these matrices could only be handled by iterative methods, and it required many days.

The algorithms we present here can possibly work over any field. However, we recall that in the numerical world, preserving sparsity is not the only concern. One also has to cope with numerical stability. As our work focusses on exact linear algebra, this is not a problem for us.

Organisation of the Part

In Chapter 2, we recall basic definitions for sparse linear algebra and graph theory, that will be utilised in the rest of the part. We also describe the data structures used to store sparse matrices and sparse vectors, as well as useful algorithms such as the ones to solve sparse triangular systems.

In Chapter 3, we start by giving a high-level description of the most famous iterative method: the Wiedemann Algorithm, as well as the parallel Block-Wiedemann Algorithm. Afterward, we recall how the two types of Gaussian elimination mentioned above work. We also present some pivoting strategies. Finally, we give a brief overview of the existing implementation of sparse gaussian elimination in exact linear algebra, as well as in the numerical world. We take this opportunity to introduce our own library, SpaSM.

In Chapter 4, we present an algorithm which was the main contribution of [BD16]. This Algorithm combines methods from both the classical gaussian elimination and the GPLU algorithm. This new algorithm basically consists in picking a set of pivots that can be chosen "a priori" without performing any arithmetical operation, but only doing permutations of the rows and columns of the initial matrix. Once these pivots are selected, all non-zero coefficients bellow can be eliminated in one fell swoop. Using a simple pivot-selection heuristic designed in the context of Gröbner basis computation [FL10], we were able to achieve large speedup on some cases (100× and more) compared to previous algorithms. In particular, this method always outperforms the sparse gaussian elimination implemented in LinBox.

Finally, in Chapter 5, we recall that the problem of finding the largest set of such a priori pivots is a difficult task. It is indeed equivalent to solving a graph NP-complete optimisation problem. We propose two new pivots selection heuristics that we introduced first in [BDV17]. One of them in particular has proved to be very efficient, allowing us to find up to 99.9% of all the pivots in some cases. We also present a parallel version of our algorithm, which is quite scalable.

Chapter 2

Of Sparse Matrices, Graphs and Sparse Triangular Systems

In this chapter, we recall important definitions related to (sparse) linear algebra and present the usual data structure that we actually utilise to store sparse matrices and sparse vectors. We also recall some important definitions from graph theory. Armed with these tools, we explain how we can actually solve sparse triangular systems. The algorithm described here will be utilised as a subroutine in the following chapters.

2.1 Sparse Matrices

It is not really easy to define formally what a sparse matrix is, as the word "sparse" refers to a vague notion. Usually, we say that a matrix is sparse if most of its coefficients are zeroes. On the other side, if many coefficients are non-zeroes, we say that the matrix is dense. It is, however, not clear what the words "many" and "most of" mean. Let us try to have a better understanding of this notion. Let A be an n-by-m matrix. We denote by $\mathrm{nz}(A)$ the number of non-zero coefficients of A. The *density* of matrix A is defined by the following formula:

$$d(A) := \frac{\mathrm{nz}(A)}{n \cdot m}. \tag{2.1}$$

In other words, the density of A is the ratio: number of non-zero coefficients divided by total number of coefficients.

Let $\epsilon \leq 1$ be a positive parameter. We can define the ϵ-*sparsity* of a matrix as follows:

Definition 2.1. A matrix A of density $d(A)$. A is ϵ-*sparse* if $d(A) \leq \epsilon$

Another way to define the words "sparse" and "dense" would be to refer to the data structure that is used to store the matrix. For instance in [Dav06], Davis calls "dense" every n-by-m matrix A whose entries are stored in a bi-dimensional table, with n rows and m columns. On the contrary, A is called "sparse" if it is stored using a special data structure, that keeps only the non-zeroes coefficients in memory.

2.1.1 Notations and Definitions

In the rest of the part, we assume that p is a prime number, and $\mathbb{F}_p$ is the finite field with p elements.

Let $\mathbf{v}$ be a vector of $\mathbb{F}_p^n$. We denote by v_i the $(i+1)$-th coefficient of $\mathbf{v}$ (indexation starts at 0), and by $\mathbf{v}_i$ a vector indexed by some parameter i.

Let A be an n-by-m matrix over $\mathbb{F}_p$. We denote by r its (possibly unknown) rank. If $r = \min(n, m)$, A is said to be *full rank*. If $r < \min(n, m)$, A is said to be *rank-deficient*.

A *non zero entry* of A is an entry (i, j) such that a_{ij} is non-zero. The collection of entries (i, i) of a matrix is called *main diagonal* of the matrix.

We recall that a *lower trapezoidal* matrix is an n-by-m matrix L whose entries above the main diagonal are zeroes. Equivalently, an n-by-m matrix U is *upper trapezoidal* if all its coefficients below the main diagonal are zeroes. In the following example, L is lower trapezoidal and U is upper trapezoidal.

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 1 & -2 & 0 \\ 0 & 2 & 0 & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} 2 & 0 & 0 & 1 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 76 \\ 0 & 0 & 0 & 0 & 5 \end{pmatrix} \tag{2.2}$$

A *lower triangular* (resp. *upper triangular*) matrix is a square lower trapezoidal (resp. upper trapezoidal) matrix. In the example above, L is lower triangular. U, on the other hand, is not upper triangular, as it contains more columns than rows.

Remark 1. The coefficients on the main diagonal of a lower or upper trapezoidal matrix are not necessary non-zeroes.

When writing down a sparse matrix, it is usual to forget the zero coefficients. Therefore, the following 4-by-4 matrix over $\mathbb{F}_5$:

$$A = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 4 & 0 & 0 \\ 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}, \tag{2.3}$$

becomes:

$$A = \begin{pmatrix} 1 & & & 2 \\ & 4 & & \\ 3 & & 1 & \\ & & & 4 \end{pmatrix} \tag{2.4}$$

2.1.2 Data Structures

Let A be an n-by-m matrix over $\mathbb{F}_p$. We present two data structures that can be utilised to store A. These data structures aim to store only the non-zero coefficients of A. However, it may happen that a coefficient a_{ij}, whose value is zero, is actually stored in these structures, if some numerical cancellation has occurred for instance. For now on, we will call "non-zero" entry of A every entry (i, j) that is actually stored, while using one of this structure, even if the coefficient a_{ij} is null. Let us call `nz` the number of non-zero entries according to this definition. In the following, we denote by `nzmax` an upper bound on `nz`.

Remark 2. `nzmax` is in fact the allocated size to store the non-zero coefficients of the matrix. When it is possible, we try to have `nzmax` = `nz` = $\mathrm{nz}(A)$.

Triplet format. This is perhaps the simplest and most intuitive way to represent a sparse matrix. We consider here a list of all the non-zeroes coefficients of the matrix given in an arbitrary order. Hence, we require two tables `Ai` and `Aj` of indexes, respectively representing the rows and columns indexes i and j, such that (i, j) is a non-zero entry. We also need a table `Ax` of elements of $\mathbb{F}_p$, to store the value of a_{ij}. These three tables are of size `nzmax`.

To have a better understanding of this structure, let us have a look at a small example. Let us consider the matrix defined in Equations 2.3 and 2.4. We can write A in triplet format as follows:

Ai	Aj	Ax
0	3	2
2	2	1
1	1	4
0	0	1
3	3	4
2	0	3

Remark 3. There are many ways to write a matrix in triplet format (in fact, there are `nzmax`! ways). The dense representation, on the other hand, is unique.

It is quite easy to understand this structure, which is quite similar to the way sparse matrices are actually stored in data basis (e.g. [Dum12]). It is also quite easy to transpose a matrix stored that way, all we have to do is to exchange tables `Ai` and `Aj`. However, it is not the most appropriate one for most of the algorithms we will describe in the following sections. Thus, we will now describe a more suitable structure in the paragraph bellow.

Compressed Sparse Row format. To store A in *Compressed Sparse Row* (CSR) format, we require two tables `Ap` and `Aj` of indexes, with `Ap` of size $n+1$ and `Aj` of size `nzmax` $\geq \mathrm{nz}(A)$. We also require a table `Ax` of size `nzmax`, of elements of $\mathbb{F}_p$, to store the value of the non-zero coefficients of A.

Let us abuse notations by calling $\mathrm{nz}(\mathbf{a}_i)$ the number of non-zero entries on the $(i+1)$-th row of A. We define `Ap` in the following way:

1. `Ap[0]` $= 0$,
2. $\forall$ `i` $\in \{0, \ldots, n\}$, `Ap[i+1]` $=$ `Ap[i]` $+ \mathrm{nz}(\mathbf{a}_{\mathtt{i}})$.

It is easy to check that `Ap[n]` is equal to `nzmax`.

The indexes j of the columns such that the entries (`i`, `j`) are non-zero, are stored from `Aj[Ap[i]]` to `Aj[Ap [i+1] - 1]`, in arbitrary order. The corresponding numerical values are stored accordingly in `Ax`.

Let us go back to our example given by Equations 2.3 and 2.4. We give a way to store this matrix in CSR format bellow:

Ap	Aj	Ax
0	0	1
2	3	2
3	1	4
5	2	1
6	0	3
	3	4

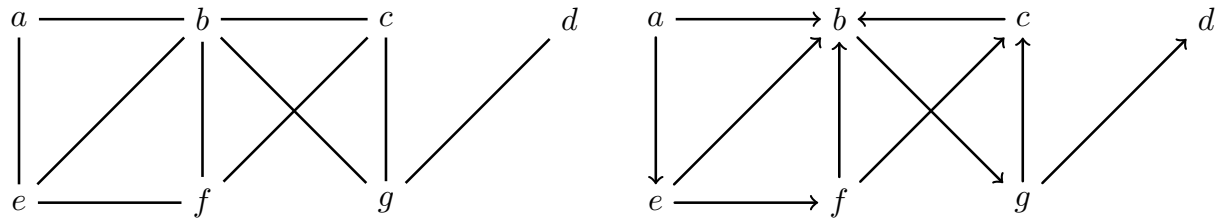
Remark 4. Once again, there are many ways to store a given matrix A using this structure.

Remark 5. Accessing to a given column of a matrix stored in CSR format requires basically to go through all non-zero entries of the matrix. On the other hand, it is quite easy to access a given row of the matrix.

Usually, these kinds of data structures require less storage than a dense representation of the matrix. This gain of space has a price. It is harder to access the data. None of these structures enables us to obtain the value of a_{ij} in constant time.

Storing a sparse vector. We are now concerned about the storage of sparse vectors. We give a simple structure to store them.

Definition 2.2. Let $\mathbf{b}$ be a sparse vector. The set of indices i such that $b_i \neq 0$ is called the *non-zero pattern*, or just the *pattern* of $\mathbf{b}$. This set is denoted by $\mathcal{B}$.



(a) Example of undirected graph.

(b) Example of directed graph.

Figure 2.1 – Examples of graphs.

In practice, we store the pattern of $\mathbf{b}$ in a table `bi`, and we store the corresponding numerical values in a table `bx`. For instance, the pattern of the following vector:

$$\mathbf{b} = \begin{pmatrix} 0 & 0 & 2 & 0 & 1 & 0 & 1 \end{pmatrix},$$

is $\mathcal{B} = \{2, 4, 6\}$. We could store it the following way:

bi	bx
4	1
2	2
6	1

Sometimes, we may temporarily need to store a sparse vector in a dense structure. We say that we "scatter" the vector. The pattern table enables us to directly find where the non-zero entries are, without having to scan through the whole vector. We proceed as follows:

for all `i` $<$ nz($\mathbf{b}$) **do**
  Set `j` $\leftarrow$ `bi[i]`
  Get $\mathbf{b}_{\texttt{j}}$

Remark 6. This "scatter" technique is a good compromise between sparse and dense data structure. Unfortunately, it cannot be applied to sparse matrices. Indeed, "scattering" a sparse matrix into a dense one would require to much memory.

2.2 A Bit of Graph Theory

There is a strong link between sparse linear algebra and graph theory. Indeed, it is common knowledge that a (sparse) graph can be represented by sparse matrices. As such, many of the problems that we will encounter in the rest of the part can also be expressed in terms of graphs.

Before going any further, we recall some important notions of graph theory, that will later be useful.

2.2.1 Important Definitions

A graph is a pair $\mathcal{G} = (V, E)$ where V is the set of *vertices* or *nodes*, and $E \subseteq V \times V$ is the set of *edges* that connect together two vertices. We must distinguish between to types of graphs: *directed* and *undirected* graphs.

In a *undirected graph*, each edge $(x, y) \in E$ is identical to the edge (y, x). In other word, the edge goes from x to y and goes back from y to x.

In a *directed graph*, each edge $(x, y) \in E$ is an *arrow*. x is called the *tail* of the arrow, and y is called the *head*. In this case, the edge goes from x to y, but does not go back from y to x.

We present an example of undirected graph in Figure 2.1a. as well as an example of directed graph in Figure 2.1b

Remark 1. Every undirected graph is equivalent to a directed graph: we simply have to replace each edge by two arrows with opposite directions.

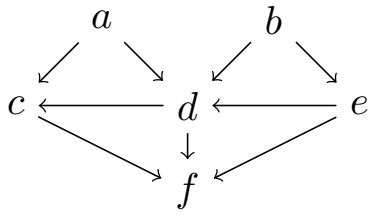


Figure 2.2 – Example of a Direct Acyclic Graph (DAG).

Let $\mathcal{G} = (V, E)$ be a directed or undirected graph. Let (x, y) be an edge, we say that y is a *direct successor* of x and x is a *direct predecessor* of y. We call *degree* of x the number of direct successors of x. In an undirected graph, two vertices x and y are said to be *adjacent* if there is an edge between them.

We say that an edge e *connects* or *joins* two nodes x and y if $e = (x, y)$. An edge e is said to be *incident* to a node x if there is a node y such that $e = (x, y)$ or $e = (y, x)$. Similarly a node x is said to be *incident* to an edge e, if there is a node y such that $e = (x, y)$ or $e = (y, x)$. We will abusively say that x is a node (or a vertex) of e. Two edges e_1 and e_2 that are incident to a same node are said to be *co-incident*.

A graph $\mathcal{G}' = (V', E')$ is a *subgraph* of $\mathcal{G} = (V, E)$ if $V' \subset V$ and $E' \subset E \cap (V' \times V')$. Let V' be a subset of V. We say that $\mathcal{G}' = (V', E')$ is the subgraph of $\mathcal{G}$ *induced* or *vertex-induced* by V', if $E' = E \cap (V' \times V')$. Let E' be a subset of $\mathcal{G}$. We say that $\mathcal{G}' = (V', E')$ is *edge-induced* by E' if V' is the set of endpoints of the edges from E'.

Let's take a walk. Let $\mathcal{G} = (V, E)$ be an undirected or directed graph. A *walk* is defined as a sequence of alternating vertices and edges $v_1, e_1, v_2, e_1, ..., e_{k-1}, v_k$, that are not necessarily distinct, such that all v_i are in V and all e_i are in E, and where each edge $e_i = (v_i, v_{i+1})$. The *length* of a walk is the number of edges that compose it. In this case, the length of the walk is $k-1$. A *trail* is a walk such that all edges $e_1, \ldots, e_k$ are distinct. A *path* is a trail such that all vertices $v_1, \ldots, v_k$ are distinct, except possibly the first and the last one. A *closed walk* is a walk such that the first vertex v_1 and the last one v_k are equal. A *circuit* is a closed trail. A *cycle* is a closed path.

Example 2.1. In the example given in Figure 2.1a, the following sequence $a, (a, b), b, (b, f), f, (f, e), e, (e, b), b, (b, f), f$ is a walk of length 5. This is not a trail as (b, f) comes twice in the sequence. On the other hand, the sequence $a, (a, b), b, (b, f), f, (f, e), e, (e, b), b, (b, g), g$ is a trail of length 5, as no edge is repeated. This is however not a path, as the vertex b is repeated twice, and is not at the extremities. The sequence $b, (b, c), c, (c, g), g, (g, b), b, (b, e), e, (e, a), a, (a, b), b$ is also a trail. Furthermore it is a circuit as it is closed (it starts and ends with the same vertex b). The sequence $a, (a, b), b, (b, g), g, (g, d), d$, is a path of length 3. The sequence $e, (e, b), b, (b, c), c, (c, f), f, (f, c), e$ is also a path. The only repeated vertex is e, which is at the extremities. This means that this sequence is also a cycle.

We now assume that $\mathcal{G}$ is a directed graph. A *semi-walk* is defined as a sequence of alternating vertices and edges $v_1, e_1, v_2, e_1, ..., e_{k-1}, v_k$, that are not necessarily distinct, such that all v_i are in V and all e_i are in E, and where each edge $e_i = (v_i, v_{i+1})$ or $e_i = (v_{i+1}, v_i)$. We can define accordingly the terms *semi-trail*, *semi-path*, *semi-circuit* and *semi-cycle*. A directed graph $\mathcal{G}$ that contains no cycle is called a *directed acyclic graph* (DAG). We give an example of a DAG in Figure 2.2. No cycle appears in this graph, however there are a lot of semi-cycles (e.g. $a, (a, c), c, (d, c), d, (a, d), a$).

As a walk (or a semi-walk) is uniquely defined by the edges of which it consists, for now on, any walk $v_1, e_1, v_2, e_1, ..., e_{k-1}, v_k$ will simply be represented by the sequence of its edges $e_1, e_1, ..., e_{k-1}$.

We say that a node x is *reachable* by a node y if there exists a path from y to x. For instance x is reachable by x via a a path of length 0. Given a node y, we call *successor* of y any node x that is reachable by y. In particular, a *direct successor* of y is a successor that can be reached from y via a path of length 1. In a similar way, we call *predecessor* of x, any node y such that x is reachable by y. If x is a direct successor of y, y is a *direct predecessor* of x.

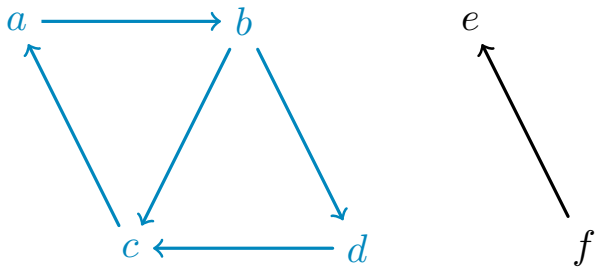


Figure 2.3 – An example of a graph $\mathcal{G}$ with two connected components. The blue one is even a strongly connected one.

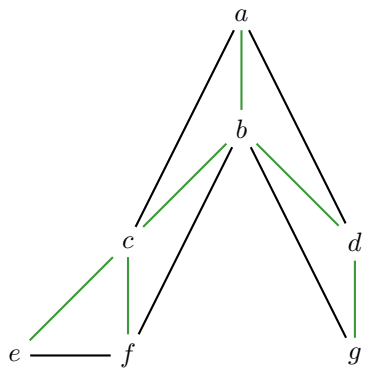


Figure 2.4 – Example of a spanning tree in green.

Connected graphs and strongly connected graphs. An undirected graph $\mathcal{G}$ is said to be *connected* if for all pairs of vertices $(x, y) \in V \times V$, there is a path between x and y. A directed graph $\mathcal{G}$ is said to be *connected* if for all pairs of vertices $(x, y) \in V \times V$, there is a semi-path between x and y. An undirected graph $\mathcal{G}$ is said to be *strongly connected* if for all pairs of vertices $(x, y) \in V \times V$, there is a path between x and y.

A subgraph $\mathcal{G}'$ of $\mathcal{G}$ is a *connected component* of G if it is a connected graph, which is not strictly included in another connected subgraph $\mathcal{G}''$ of G.

A subgraph $\mathcal{G}'$ of $\mathcal{G}$ is a *strongly connected component* of G if it is a strongly connected graph which is not strictly included in another strongly connected subgraph $\mathcal{G}''$ of G.

In Figure 2.3, we give an example of a graph $\mathcal{G}$ with two connected components. The blue one is a strongly connected component. It is easy to check that for all pairs of nodes $(x, y) \in \{a, b, c, d\} \times \{a, b, c, d\}$ there is a path that connects x to y. The sub-graph induced by $\{a, b, c\}$ is also a strongly connected graph. It is not however a strongly connected component of $\mathcal{G}$, as it is included in the subgraph induced by $\{a, b, c, d\}$.

Trees and forests. An undirected acyclic graph with only one connected component is called a *tree*. An undirected acyclic graph with more than one connected component is called a *forest*.

Let $\mathcal{G} = (V, E)$ be an undirected graph. We call *spanning tree* of $\mathcal{G}$, a sub-graph $\mathcal{T} = (V', E')$, of $\mathcal{G}$, such that $V' = V$, and $\mathcal{T}$ is a tree. An example of a spanning tree is given in Figure 2.4.

2.2.2 Graph Search

Let $\mathcal{G} = (V, E)$ a graph and $s \in V$ a vertex of $\mathcal{G}$. Graph search algorithms allow to find all the successors of s in $\mathcal{G}$. There are essentially two ways to proceed: using a Breadth-first search (BFS) or using a Depth-first search (DFS). These two algorithms are well-known in graph theory, as such, we will not fully describe them, but only recall their idea. A full description of these algorithms can for instance be found in [CLRS01].

During these procedures, the nodes can take three colours: white (unmarked), red (visited), and black (treatment over).

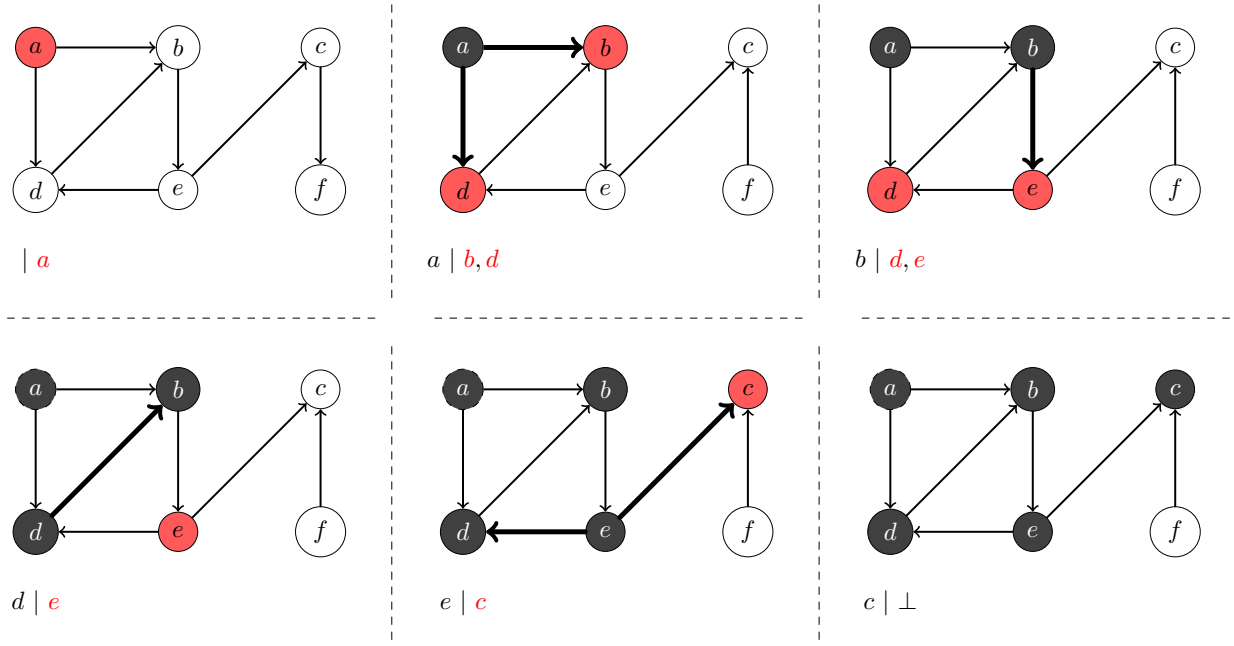


Figure 2.5 – Example of breadth-first search.

Remark 2. We actually need only two additional colours, (here red and black) in these two procedures. The white colour represents a default colour all nodes have at the beginning of the procedures.

Breadth-first search (BFS). At the beginning of the procedure, all nodes save s are unmarked, and s is marked in red. We also dispose of a queue that contains only s. At each step, the first element x of the queue is dequeued, each of its uncoloured direct successors is marked in red and enqueued. Once this is done, we mark x in black (its treatment is over) and add it in the list R of the successors of s. This procedure lasts until there is no element left in the queue. At the end, we return R. We give an example to illustrate this procedure in figure 2.5. As each edge is visited at most once and each node only a constant number of times, the time complexity of the procedure is $\mathcal{O}(|V| + |E|)$.

Depth-first search. At the beginning of the procedure, all nodes save s are white, and s is red. We also dispose of a stack which only consists of s. At each step, while the node on the top of the stack still has unmarked direct successors we arbitrarily choose one of them, mark it in red an push it on the stack. Whenever the node on the top of the stack has no more unmarked direct successors, we mark it in black, pop it out of the stack, and add it to the list R of the successors of s. This procedure lasts until there is no element left in the stack. At the end, we return R. We give an example to illustrate this procedure in Figure 2.6. Once again, each edge is at most visited once, and each node only a constant number of times. Then the worst case time complexity of this procedure is $\mathcal{O}(|V| + |E|)$.

Sorting a DAG in topological order. Recall that a DAG $\mathcal{G}$ does not contain any cycle. In particular, this means that there exists at least one node s such that s does not have any predecessor. We say that s is a *source*. We also have that for two nodes x and y, if y is a successor of x, then x is not a successor of y. It may be useful to have the nodes of $\mathcal{G}$ sorted in a way such that for any node x, all the predecessors come before x, and all its successors come after. Such an order is called a *topological order*.

Proposition 2.1. *A DFS ran on a DAG and a node s returns the successors of s, sorted in topological order.*

Remark 3. In fact, the last element of R will be our source s, and for all indexes i and j, such that

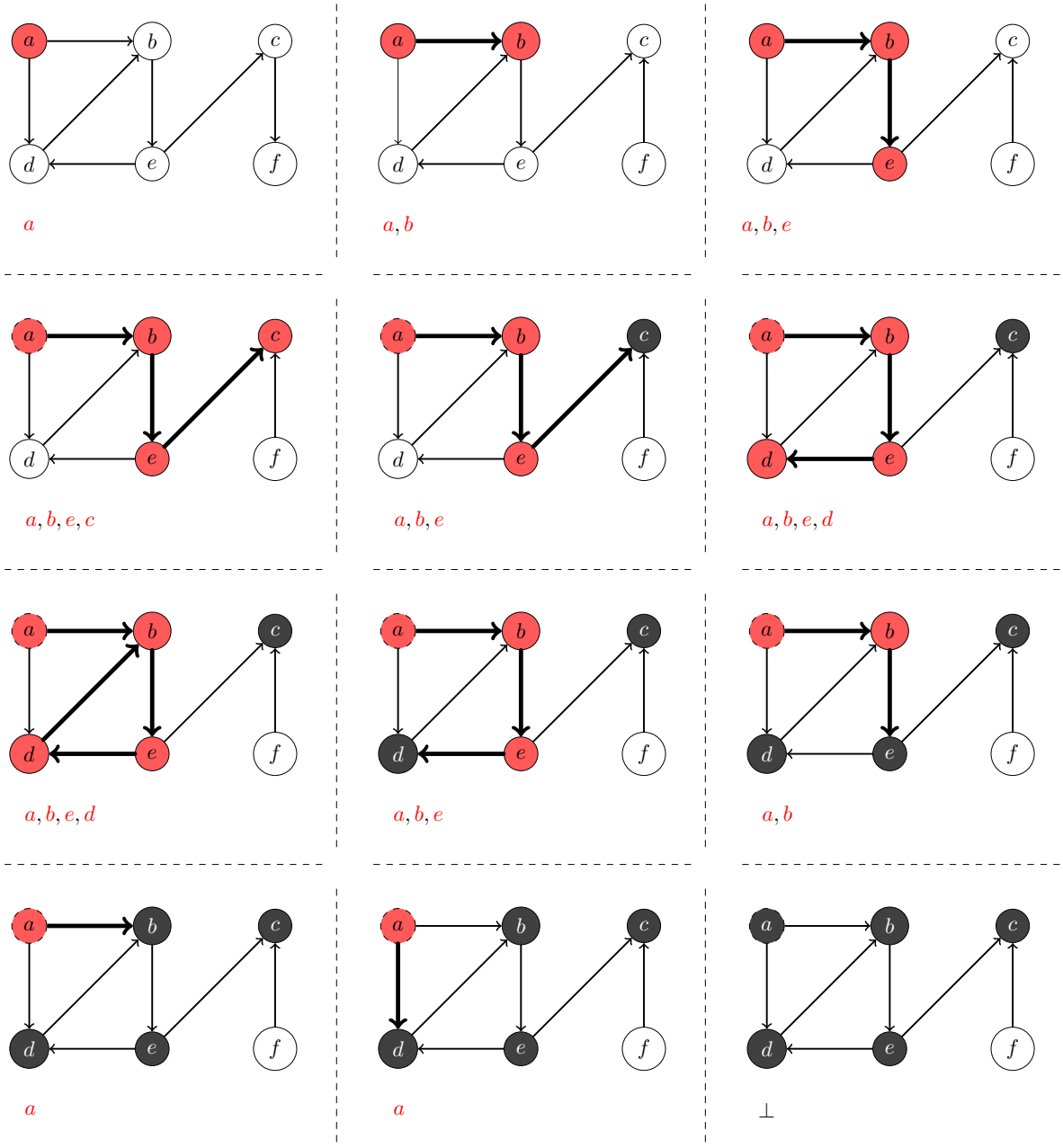


Figure 2.6 – Example of a depth-first search.

$R[i]$ is a predecessor $R[j]$, we will have $i > j$. This looks more like a "reverse" topological order. To actually get the entries in topological order, as defined above, you will have to reverse R.

This is a well known result in graph theory. We recall the proof as it was given in [CLRS01].

Proof. We need to show that for any two distinct vertices x, y, if there is a edge $x \to y$, then y comes before x in R. When the edge $x \to y$ is explored during the procedure, y is not red. Otherwise y would have been an ancestor of x, contradicting the fact that $\mathcal{G}$ is a DAG. Hence, y is either black (already treated) or white (not yet encountered). If y is white, then y is added to the stack, and marked as a descendant of x, it will then be popped from the stack and added to R before x. If y is black, then its treatment is finished, meaning its is already in R, so x will eventually come after. □

2.2.3 Bipartite Graph and Matching

A bipartite graph is a graph $\mathcal{G} = (V, E)$ where V can be split in two disjoint sets X and Y, such that every edge of E connects a vertex from X with a vertex from Y. We can denote $\mathcal{G}$ by the triplet (X, Y, E). An example of a bipartite graph is given by Figure 2.7.

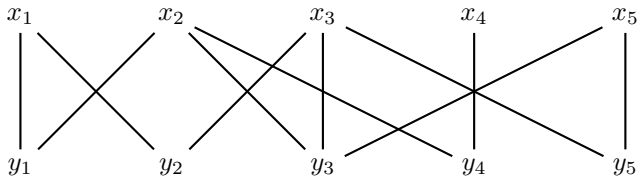


Figure 2.7 – Example of a bipartite graph.

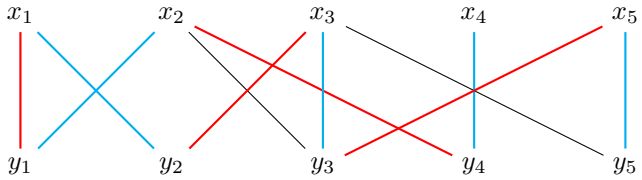


Figure 2.8 – Example of two maximal matchings. The blue one is even a perfect matching.

Let $\mathcal{G} = (X, Y, E)$ be an undirected bipartite graph. A *matching* in $\mathcal{G}$, is a subset $\mathcal{M}$ of E such that if $(x_1, y_1) \in \mathcal{M}$ and $(x_2, y_2) \in \mathcal{M}$, then $x_1 \neq x_2$ and $y_1 \neq y_2$. In other words, a matching can be seen as a set of edges without common vertices. This notion can be extended to non-bipartite graphs.

A matching $\mathcal{M}$ is said to be *maximal* if there is no matching $\mathcal{M}'$ such that $\mathcal{M} \subsetneq \mathcal{M}' \subset E$. A matching $\mathcal{M}$ is said to be *maximum* if there is no matching $\mathcal{M}'$ such that $|\mathcal{M}'| \geq |\mathcal{M}|$. A matching $\mathcal{M}$ is said to be *perfect* if it matches all vertices of the graph. This assumes that $|X| = |Y|$.

Remark 4. A maximum matching is also maximal. In fact, if there was another matching $\mathcal{M}'$ such that $\mathcal{M} \subsetneq \mathcal{M}' \subset E$, then $|\mathcal{M}'| > |\mathcal{M}|$.

Remark 5. It is well known that it is possible to find a maximum matching in a bipartite graph in time $\mathcal{O}((|X| + |Y|)|E|)$, using the Ford-Fulkerson algorithm [FF56]. It is possible to make this drop to $\mathcal{O}\left(\sqrt{|X| + |Y|}|E|\right)$, using the Hopcroft-Karp algorithm [HK73]. A generalisation of the later algorithm by Blum [Blu99] allows to find a maximum in a general graph $\mathcal{G} = (V, E)$ in time $\mathcal{O}\left(\sqrt{|V|}|E|\right)$. This is however not the topic of this thesis so we do not detail any further.

Let $\mathcal{G} = (X, Y, E)$ be an undirected bipartite graph, and let $\mathcal{M}$ be a matching in $\mathcal{G}$. We call *alternating path* in $\mathcal{G}$ any path in which the edges alternatively belong and do not belong to the matching. An *augmenting path* is an alternating path such that the first and the last edges are not in the matching.

We illustrate these notions in Figure 2.8. We give two examples of maximal matching in a bipartite graph. The red one is not maximum, as there is an other matching (the blue one) that consists of more edges. The blue one, on the other hand is maximum, and even perfect. Considering the blue matching, the path

$$(x_4, y_4), (y_4, x_2), (x_2, y_1), (y_1, x_1), (x_1, y_2), (y_2, x_3), (x_3, y_3), (y_3, x_5), (x_5, y_5)$$

is an alternating one. If we consider the red matching, the same path is an augmenting one.

2.2.4 Independent Sets.

Let $\mathcal{G} = (V, E)$ be an undirected graph. We call *independent set* of $\mathcal{G}$ any subset $\mathcal{I}$ of V such that for every two vertices in $\mathcal{I}$ there is no edge connecting the two. The *size* of an independent set $\mathcal{I}$ is the number of vertices it contains. We denote it by $|\mathcal{I}|$. A set $V \setminus \mathcal{I}$, where $\mathcal{I}$ is an independent set, is called a *vertex cover*.

An independent set $\mathcal{I}$ of $\mathcal{G}$ is said to be *maximal* if there is no independent set $\mathcal{I}'$ of $\mathcal{G}$ such that $\mathcal{I} \subsetneq \mathcal{I}'$. An independent set $\mathcal{I}$ of $\mathcal{G}$ is said to be *maximum* if there is no independent set $\mathcal{I}'$, such that $|\mathcal{I}'| > |\mathcal{I}|$. Examples of independent sets are given in Figure 2.9. The two independent sets shown on this figure are maximal.

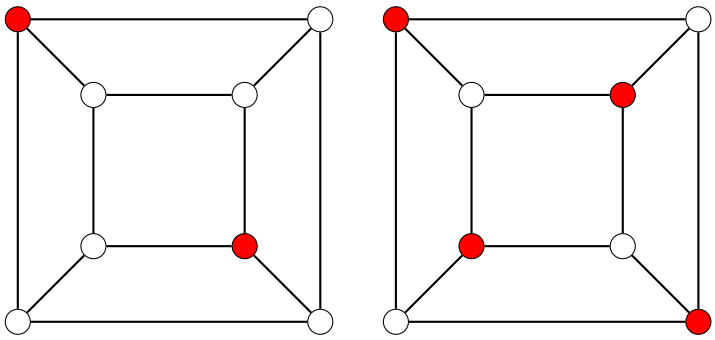


Figure 2.9 – Example of two maximal independent sets. The right one is even maximum.

Remark 6. We can easily check that a maximum set is also a maximal one. Indeed, if there was an independent $\mathcal{I}'$ such that $\mathcal{I} \subsetneq \mathcal{I}'$, then $|\mathcal{I}'| > |\mathcal{I}|$.

Maximum independent set problem. Finding a maximum independent set is an NP-Hard optimisation problem. Approximating the problem is also a difficult task [Hås99].

In this thesis we are particularly interested in finding a large independent set in a bipartite graph $\mathcal{G} = (R, C, E)$. In this case the problem is simpler. Indeed, in 1931 Köning proved the following result:

Theorem 2.2 (Köning's theorem)**.** *In any bipartite graph, the number of edges in a maximal matching is equal to the number of vertices in a minimal vertex cover.*

Furthermore, it is possible to find a minimal vertex set in a bipartite graph using maximum matching algorithms such as the Hopcroft-Karp one, and thus to find a minimum vertex cover S in a bipartite graph in $\mathcal{O}\left(\sqrt{|R \cup C|}|E|\right)$. The set $\mathcal{I} = (R \cup C) \setminus S$ is a maximum independent set.

We do not detail this method here, but choose to present a greedy algorithm that allows us to find a hopefully large independent set in quasi-linear time $\mathcal{O}(|R \cup C| \log |E|)$, that will be utilised as a subroutine in an algorithm presented in Section 5.2.

The idea is quite naive. If there remain more row vertices than column vertices, add the row vertex of smallest degree to $\mathcal{I}$ and remove it, and all the columns vertices to which it is connected from the graph. Otherwise, do the same with the column vertex of smallest degree. We summarise this procedure in Algorithm 1.

Algorithm 1 Finding a (large) independent set in bipartite graph.

Require: $\mathcal{G} = (R, C, E)$.
Ensure: $\mathcal{I}$: an independent set of $\mathcal{G}$.

```
Set I ← ⊥
while G ≠ ⊥ do
    if |R| ≥ |C| then
        Search for a vertex v of smallest degree in R
    else
        Search for a vertex v of smallest degree in C
    Set I ← I ∪ {v}
    for all x such that (v, x) ∈ E do
        remove x from G
    remove v from G
```

We illustrate this procedure in Figure 2.10: as we choose R_0 which is a row of smallest degree and add it to our empty set $\mathcal{I}$. This forces us to remove the column C_1 to which R_0 is connected from the graph. The node R_1 is not connected anymore and we can add it to $\mathcal{I}$ for free. Now, we have more columns than rows, we add C_0 (which is of smallest degree) to $\mathcal{I}$ and remove R_2, to which C_0 is connected, from the graph. Once again, this allows us to add a node, C_2, to $\mathcal{I}$

for free. We have now the same number of rows than columns. We decide to add R_3 to $\mathcal{I}$, hence removing both C_3 and C_4, which gives us R_4 for free. We finally end up with an independent set $\mathcal{I} = \{R_0, R_1, C_0, C_2, R_3, R_4\}$.

This can be implemented in time $\mathcal{O}((|R| + |C|) \log |E|)$ using two binary heaps to store remaining vertices of each kind.

2.2.5 Sparse Matrices and Graphs

Matrix representation of a graph. Let $\mathcal{G} = (V, E)$. We call *adjacency matrix* of $\mathcal{G}$, the $|V|$-by-$|V|$ matrix A, indexed by the nodes of $\mathcal{G}$, such that $a_{ij} = 1$ if there is $(v_i, v_j) \in E$ and 0 otherwise. We give an example of a directed graph and its adjacency matrix in Figure 2.11.

We call *incidence matrix* of an undirected graph $\mathcal{G}$, the $|V|$-by-$|E|$ matrix A, where the rows represent the nodes of $\mathcal{G}$ and the columns the edges, such that $a_{ij} = 1$ if v_i is a node of e_j, and 0 otherwise. If $\mathcal{G}$ is a directed graph, it is still possible to define its incidence matrix the following way: we say that $a_{ij} = 1$ if v_i is the head of e_j, -1 if it is its tail, and 0 otherwise. If $\mathcal{G} = (X, Y, E)$ is an undirected bipartite graph, we call *occurrence matrix* of $\mathcal{G}$ the $|X|$-by-$|Y|$ matrix A, where the rows represent the nodes in X and the columns represent the nodes in Y. We give an example of a bipartite graph and its occurrence matrix in Figure 2.12.

These representations are unique modulo rows and columns permutations.

Sparse matrices seen as graph. If we consider only its pattern (i.e. the positions of non-zero coefficients and not their actual values), any sparse matrix can be associated to a sparse graph.

Symmetric matrices can be seen as adjacency matrices of undirected graphs. When a matrix is square but non symmetric, it can be seen as the adjacency matrix of a directed graph. A non-square matrix can be seen as the occurrence matrix of an undirected bipartite graph.

Triangular matrices and DAG. We first need to introduce the notion of (α, β, γ)-*adjacency matrix*, for any graph $\mathcal{G}$ with no loop (i.e. an edge that connect a vertex to itself). Given three fixed parameters α, β, γ it is possible to define the (α, β, γ)-*adjacency matrix* of $\mathcal{G}$, where for all $i \neq j$, $a_{ij} = \alpha$ if there is $(v_i, v_j) \in E$, and β otherwise, and all diagonal coefficients are γ.

The adjacency matrix defined above is in fact a $(1, 0, 0)$-adjacency matrix.

Let U be an upper-triangular matrix of size n with non-zero diagonal. We claim that the pattern of U can be seen as the $(1, 0, 1)$-adjacency matrix of a DAG $\mathcal{G}$.

Indeed, if $\mathcal{G} = (V, E)$, with $V = \{v_0, \ldots, v_{n-1}\}$, for all i, j such that $i \neq j$, $(v_i, v_j) \in E$ if and only if $u_{ij} \neq 0$. Yet, $u_{ij} \neq 0$ implies that $j > i$, as the matrix is upper triangular, and $i \neq j$. Then $(v_i, v_j) \in E$ only if $j > i$. Then, if there is a path between two nodes v_i and v_j in $\mathcal{G}$ then $j > i$, hence there is no cycle in $\mathcal{G}$.

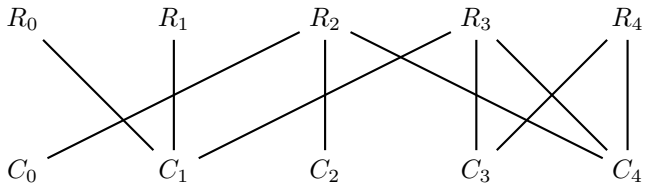
Furthermore, we also claim that if $\mathcal{G}$ is a DAG with n nodes, then there is an upper-triangular matrix U of size n with non-zero diagonal, such that U is the $(1, 0, 1)$-adjacency matrix of $\mathcal{G}$.

The argument is quite similar. We consider here that the nodes in V are sorted in topological order. Then for all couples of nodes (v_i, v_j), $(v_i, v_j) \in E$ implies that $i < j$. In other words, for all i, j such that $i \neq j$, $u_{i,j} \neq 0$ implies that $i < j$. We give an example of a DAG and its $(1, 0, 1)$-adjacency matrix in Figure 2.13. In what follows, the $(1, 0, 1)$-adjacency matrix of a DAG $\mathcal{G}$ will abusively be called "adjacency matrix" of $\mathcal{G}$, for simplicity.

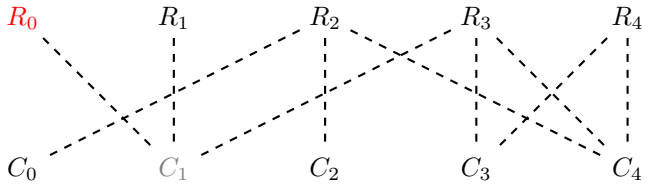
2.3 Sparse Triangular Solving

We are now going to present algorithms to solve a linear system $\mathbf{x} \cdot U = \mathbf{b}$, where U is a sparse upper triangular matrix of size n stored using a CSR data structure. We assume that all diagonal coefficients of U are non-zero.

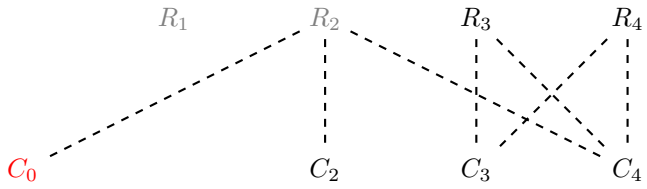
We need to distinguish two cases, depending if the right-hand side (RHS) $\mathbf{b}$ of the system is dense or sparse.



$\mathcal{I} = \bot$



$\mathcal{I} = \{R_0\}$



$\mathcal{I} = \{R_0, R_1, C_0\}$

$\mathcal{I} = \{R_0, R_1, C_0, C_2, R_3\}$

$\mathcal{I} = \{R_0, R_1, C_0, C_2, R_3, R_4\}$

Figure 2.10 – Illustration of the independent set search method.

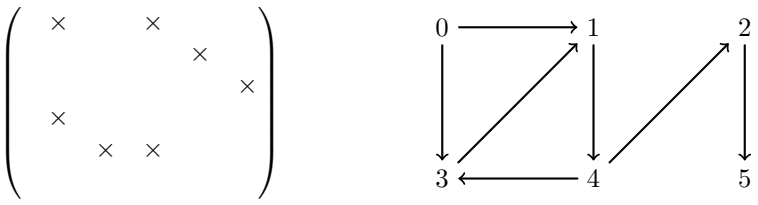


Figure 2.11 – Example of a directed graph and its adjacency matrix.

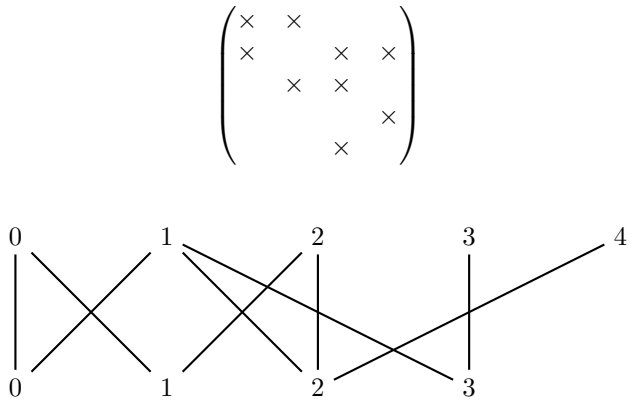


Figure 2.12 – Example of a bipartite graph and its occurrence matrix.

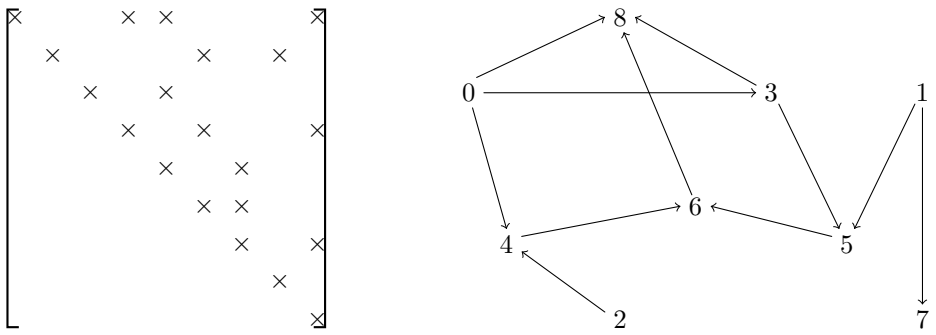


Figure 2.13 – Example of a DAG and its $(1, 0, 1)$-adjacency matrix.

Algorithm 2 Solving upper triangular system: dense algorithm.

Require: $\mathbf{b}$, U.
Ensure: $\mathbf{x}$ such that $\mathbf{x} \cdot U = \mathbf{b}$
1: $\mathbf{x} \leftarrow \mathbf{b}$
2: **for** $j = 0$ to $n - 1$ **do**
3: **for** $i = 0$ to $j - 1$ **do**
4: $x_j \leftarrow (x_j - x_i u_{ij})/u_{jj}$
5: **return** $\mathbf{x}$

2.3.1 With Dense RHS.

For each index j, $0 \leq j \leq n$:

$$b_j = \sum_{i=0}^{j} x_i u_{ij}, \tag{2.5}$$

Assuming that u_{jj} is non-zero, we can reverse this equation, and thus:

$$x_j = \frac{1}{u_{jj}} \left(b_j - \sum_{i=0}^{j-1} x_i \right). \tag{2.6}$$

If U were stored using a dense data structure, we would use the classical method described in Algorithm 2. However, we recall that our matrix U is stored using a CSR data structure, and this column-by-column algorithm is not fitted to this kind of structure. We need to swap the two for loop. Furthermore, as U is sparse, it is quite useless to go through all entries; looking only at the non-zero ones is enough. All in all, we come up with Algorithm 3.

Algorithm 3 Solving upper triangular system: sparse matrix, dense RHS.

Require: $\mathbf{b}$, U.
Ensure: $\mathbf{x}$ such that $\mathbf{x} \cdot U = \mathbf{b}$
1: $\mathbf{x} \leftarrow \mathbf{b}$
2: **for** $i = 0$ to $n - 1$ **do**
3: $\quad x_i \leftarrow x_i / u_{ii}$
4: $\quad$ **for all** $j > i$ such that $u_{ij} \neq 0$ **do**
5: $\quad\quad x_j \leftarrow x_i - x_i u_{ij}$
6: **return** $\mathbf{x}$

Following the same idea, Algorithm 4 can be utilised to solve $\mathbf{x} \cdot L = \mathbf{b}$, where L is a lower triangular matrix of size n, stored using a CSR structure. In this case, we assume that all l_{ii} are non-zero coefficients.

Algorithm 4 Solving lower triangular system: sparse matrix, dense RHS.

Require: $\mathbf{b}$, L.
Ensure: $\mathbf{x}$ such that $\mathbf{x} \cdot U = \mathbf{b}$
1: $\mathbf{x} \leftarrow \mathbf{b}$
2: **for** $i = 0$ to $n - 1$ **do**
3: $\quad x_i \leftarrow x_i / u_{ii}$
4: $\quad$ **for all** $j > i$ such that $u_{ij} \neq 0$ **do**
5: $\quad\quad x_j \leftarrow x_i - x_i u_{ij}$
6: **return** $\mathbf{x}$

2.3.2 With Sparse RHS.

We consider now a system $\mathbf{x}A = \mathbf{b}$ where both A and $\mathbf{b}$ are sparse. A first naive idea to solve this kind of system would be to scatter $\mathbf{b}$ into a dense vector $\mathbf{x}$, and to solve the system in similar fashion to dense RHS case, using Algorithm 5.

We denote by f the total number of field operations (additions and multiplications) over $\mathbb{F}_p$ that are to be performed during the procedure (i.e. during steps 4 and 6). Considering that the time complexity of the scatter procedure is proportional to the number of non-zero coefficients of the input vector, step 1 requires $\mathcal{O}(\mathrm{nz}(\mathbf{b}))$ operations. Furthermore, as n tests are to be performed in step 3, the time complexity of this procedure would be $\mathcal{O}(n + \mathrm{nz}(\mathbf{b}) + f)$. However, if U and $\mathbf{b}$ are both sparse, we can hope that $\mathbf{x}$ will also be sparse. Then, assuming that we know the non-zero pattern of $\mathbf{x}$, $\mathcal{X}$, in advance, and that $\mathcal{X}$ is sorted, we do not have to check whether x_i is non-zero anymore, and using Algorithm 6, we can find $\mathbf{x}$ in $\mathcal{O}(\mathrm{nz}(\mathbf{b}) + f)$.

Algorithm 5 Solving upper triangular system: sparse matrix, sparse RHS: first idea.

Require: $\mathbf{b}$, U.
Ensure: $\mathbf{x}$ such that $\mathbf{x} \cdot U = \mathbf{b}$
1: $\mathbf{x} \leftarrow \mathbf{b}$
2: **for** $i = 0$ to $n - 1$ **do**
3: **if** $x_i \neq 0$ **then**
4: $x_i \leftarrow x_i / u_{ii}$
5: **for all** $j > i$ such that $u_{ij} \neq 0$ **do**
6: $x_j \leftarrow x_i - x_i u_{ij}$
7: **return** $\mathbf{x}$

Algorithm 6 Solving upper triangular system: sparse matrix, sparse RHS.

Require: $\mathbf{b}$, U.
Ensure: $\mathbf{x}$ such that $\mathbf{x} \cdot U = \mathbf{b}$
1: $\mathbf{x} \leftarrow \mathbf{b}$
2: **for all** $i \in \mathcal{X}$ **do**
3: $x_i \leftarrow x_i / u_{ii}$
4: **for all** $j > i$ such that $u_{ij} \neq 0$ **do**
5: $x_j \leftarrow x_i - x_i u_{ij}$
6: **return** $\mathbf{x}$

Finding $\mathcal{X}$. All we have to do now is to find and sort $\mathcal{X}$. According to Algorithm 6, a coefficient x_j of $\mathbf{x}$ is non-zero, either if b_j is non-zero or if there is an index i such that $x_j u_{ij} \neq 0$. If we assume for a while that there has been no numerical cancelation, then we have the following implications:

1. $b_j \neq 0 \implies x_j \neq 0$,
2. $x_i \neq 0$ and $\exists j,\ u_{ij} \neq 0 \implies x_j \neq 0$.

These implications are illustrated by Figure 2.14. This is in fact a graph problem, as it was noticed by Gilbert and Peierls in [GP88].

Theorem 2.3 (Gilbert and Peierls [GP88])**.** *Define the directed graph $\mathcal{G}_U = (V, E)$ with nodes $V = \{1 \dots m\}$ and edges $E = \{(i, j) | u_{ij} \neq 0\}$. Let $\mathrm{Reach}_U(i)$ denote the set of nodes reachable from i via paths in $\mathcal{G}_U$ and for a set $\mathcal{B}$, let $\mathrm{Reach}_U(\mathcal{B})$ be the set of all nodes reachable from any nodes in $\mathcal{B}$. Assuming there is no numerical cancellation, the non-zero pattern $\mathcal{X} = \{j | x_j \neq 0\}$ of the solution $\mathbf{x}$ to the sparse linear system $\mathbf{x} \cdot U = \mathbf{b}$ is given by $\mathcal{X} = \mathrm{Reach}_U(\mathcal{B})$, where $B = \{i | b_i \neq 0\}$.*

Proof. The proof is quite trivial. We only have to rewrite the two conditions in terms of graph: $i \in \mathcal{X}$ if either one of the two following conditions is satisfied:

1. $j \in \mathcal{B} \implies j \in \mathcal{X}$,
2. $i \in \mathcal{X}$ and $\exists j,\ (i, j) \in E \implies j \in \mathcal{X}$.

From here we can easily conclude that $\mathcal{X}$ and $\mathrm{Reach}_U(\mathcal{B})$ are the same set. □

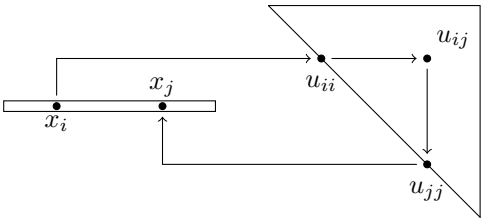


Figure 2.14 – x_i and u_{ij} implies $x_j \neq 0$.

From here, we can find the elements of $\mathcal{X}$ by performing a graph search. This leaves us with sorting them. If x_j and x_i both belong to $\mathcal{X}$, and if we want to replace x_j by $x_j - x_i u_{ij}$ (step 5 of Algorithm 6), we need to have computed x_i before we compute x_j. In other words, if there is an edge $(i, j) \in E$, then i must come before j in $\mathcal{X}$. Thus, $\mathcal{X}$ must be sorted in topological order. We perform a DFS to obtain $\mathcal{X}$ in this order. This can be performed in time $\mathcal{O}(\mathrm{nz}(\mathbf{x}) + \mathrm{nz}(U))$.

Chapter 3

Of Sparse Linear Algebra: Iterative vs Direct Methods, Left vs Right

In this chapter, we present some aspects of the state of the art in sparse linear algebra. We briefly recall the idea of the most famous of the iterative methods: the Wiedemann Algorithm, before focusing on direct methods. We present two famous sparse gaussian elimination algorithms, as well as some pivoting strategies that aim to reduce the fill-in. Finally, we give a brief overview of various software that implement sparse gaussian eliminations. We also present our own library SpaSM.

3.1 Iterative Methods

Iterative methods can be used in exact linear algebra to perform the usual operations on sparse matrices (e.g. computation of the rank, solving linear systems). They basically work by performing a series of matrix-vector products, and only need to store one or two vectors in addition to the matrix. The matrix itself is never modified.

This thesis focuses on direct method however, as such, we do not get into the details of how iterative methods works. We just briefly recall the idea of the Wiedemann and Block-Wiedemann algorithm, and refer the interested reader to [Wie86, Cop94, Kal95] for more details about these methods.

3.1.1 Wiedemann Algorithm

High level idea. We consider first the easy case where A is square. Let A be an n-by-n matrix, and $\mathbf{b}$ an arbitrary vector in $\mathbb{F}_p^n$. We consider the following sequence:

$$\mathbf{b}, \mathbf{b}A, \ldots, \mathbf{b}A^n, \ldots \tag{3.1}$$

Let $P_\mathbf{b}$ the monic polynomial with scalar coefficients of minimum degree such that $\mathbf{b}P_\mathbf{b}(A) = 0$. If we denote by μ_A the *minimal polynomial* of A (i.e. monic polynomial with scalar coefficients of minimum degree such that $\mu_A(A) = 0$), then $P_\mathbf{b}$ divides μ_A and thus $\deg(P_\mathbf{b}) \leq \deg(\mu_A)$.

Indeed, we can easily check that $\mathcal{P} = \{P \mid bP(A) = 0\}$ is an ideal of $\mathbb{F}_p[X]$. Every ideal of $\mathbb{F}_p[X]$ is principal, meaning generated by a single element of minimum degree. Here $P_\mathbf{b}$ is a generator of $\mathcal{P}$. Furthermore, we have $\mathbf{b}\mu_A(A) = 0$, and thus μ_A is in $\mathcal{P}$, as such, μ_A is a multiple of $P_\mathbf{b}$.

If you know how to compute $P_\mathbf{b}$, you can easily:

1. Solve $\mathbf{x}A = \mathbf{b}$, if A is full rank,
2. Find a vector in the kernel of A, if A is rank-deficient,

3. Compute the μ_A.

First notice that $P_{\mathbf{b}}(X) = c + Q(X)X$, for some $c \in \mathbb{F}_p$ and some $Q \in \mathbb{F}_p[X]$. We can check that $c = 0$ only if A is rank deficient. We first have the equivalence:

$$c = 0 \iff 0 \quad \text{is a root of } P_{\mathbf{b}}.$$

Then, if $c = 0$, 0 is root of μ_A, and thus 0 is an eigenvalue of A, and thus, A is rank deficient. As such, if A is full rank then $P_{\mathbf{b}}(X) = c + Q(X)X$, for some $c \in \mathbb{F}_p^*$. Then, as long as $\mathbf{b}P_{\mathbf{b}}(A) = 0$ we have $c\mathbf{b} + \mathbf{b}Q(A)A = 0$, and thus

$$(-c^{-1}\mathbf{b}Q(A)) \cdot A = \mathbf{b},$$

Thus $-c^{-1}\mathbf{b}Q(A)$ is solution to $\mathbf{x}A = \mathbf{b}$.

Now, $P_{\mathbf{b}}(X) = Q(X)X$ implies that A is rank-deficient. Furthermore, as $\mathbf{b}P_{\mathbf{b}}(A) = 0$ we have:

$$(\mathbf{b}Q(A)) \cdot A = 0$$

Thus $\mathbf{b}Q(A)$ is a non trivial element of the kernel of A.

Finally, if you know how to compute $P_{\mathbf{b}}$ for any arbitrary $\mathbf{b}$ of $\mathbb{F}_p^n$, then you can compute many of such polynomials for different $\mathbf{b}$ vectors. The minimal polynomial μ_A of ${}^tA$ will be the least common multiple of all polynomials found this way.

If A is a non square n-by-m matrix over $\mathbb{F}_p$, then, it is possible to build a m-by-n sparse matrix B, such that if A is full rank, the m-by-m matrix BA is also full rank with high probability. It is possible to use the Wiedemann algorithm on this matrix BA. However, the results obtained will hold for A only with a certain probability, and thus the procedure will have to be repeated more times.

Computing the polynomial $P_{\mathbf{b}}$. It can be done using the Berlekamp-Massey algorithm. Choose a random vector $\mathbf{u} \in \mathbb{F}_p^n$, and compute the sequence:

$$\langle \mathbf{u}, \mathbf{b} \rangle, \langle \mathbf{u}, \mathbf{b}A \rangle, \ldots, \langle \mathbf{u}, \mathbf{b}A^{2n} \rangle. \tag{3.2}$$

We can compute its minimal polynomial $P_{\mathbf{u},\mathbf{b}}$ using the Berlekamp-Massey algorithm. $P_{\mathbf{u},\mathbf{b}}$ divides $P_{\mathbf{b}}$ and is actually equal to $P_{\mathbf{b}}$ with probability greater than $1/(6 \log n)$.

Computing the rank of a matrix. The computation of the rank r of a matrix A using Wiedemann Algorithm is a little more tricky. The point is to find some square matrix $\tilde{A}$ whose minimum polynomial $\mu_{\tilde{A}}$ is of degree d, such that with high probability $r = d + c$, where c is a known constant. Then, to compute the rank r of A, you find the minimum polynomial of this matrix $\tilde{A}$ using Wiedemann Algorithm.

There are results from [KS91] and [EK97] which explain how to find such a matrix $\tilde{A}$. In fact, Kaltofen and Saunders [KS91] showed that if A is square of size n and if U_1 and U_2 are two upper-triangular Toeplitz matrices, with a unit diagonal, such that all coefficients above the diagonal are randomly chosen in $\mathbb{F}_p^*$, and if D is a diagonal matrix, whose diagonal coefficients are also randomly chosen in $\mathbb{F}_p^*$, then the minimum polynomial of the matrix $\tilde{A} = U_1 A {}^tU_2 D$ will have degree $r + 1$. with probability $1 - (3n(n+1))/2(p-1)$. Another result from Eberly and Kaltofen [EK97] states that if A is a non-necessarily square n-by-m matrix, and if D_1 and D_2 are two random diagonal matrices whose diagonal coefficients are in $\mathbb{F}_p^*$, then the minimum polynomial of $\tilde{A} = D_1 {}^tA D_2 A D_1$ will have degree r with high probability. We refer the interested reader to [KS91, EK97] for more details about these methods.

Block-Wiedemann Algorithm In [Cop94], Coppersmith proposed a generalisation of the method above. In fact, to compute the minimum polynomial of $\tilde{A}$, one has to compute several sequences like the one given in Equation 3.2, for different random vectors $\mathbf{u}$ and $\mathbf{v}$. Instead of computing them one after the other, Coppersmith proposed to compute them all in the same time. In other words, he does not compute elements of the type:

$$x = \langle \mathbf{u}, \mathbf{b}A^i \rangle = \mathbf{u}^t A^{it} \mathbf{b},$$

anymore, but:

$$X = \langle U, BA^i \rangle = U^t A^{it} B,$$

where U, B and X are now matrices over $\mathbb{F}_q$. Computing the entries of X can then be done in parallel.

3.1.2 Discussion

Iterative methods can be slow but they are sure. Their time complexity is essentially $\mathcal{O}(r\text{nz}(A))$ where r is the rank of A, and their running time is quite easy to predict. However, they can sometimes be very slow. The matrix A can be seen as some kind of "black box" that is only used to compute the sequences. As such, it is never modified and the extra space required is then small, and thus they cannot fail by lack of memory. For this reason, they are the only option for matrices on which direct methods are impractical.

Opposed to the iterative methods, the direct methods work by computing an echelonised version of the input matrix. In this case the matrix is inevitably modified, and eventually become denser. This makes the direct methods quite unpredictable, as their running time will depend of this fill-in. They can become very slow, or even fail by lack of memory. On the other hand, when they work they are usually faster than the iterative methods.

In [DV02] survey, Dumas and Villard actually benchmarked both methods on matrices from the [Dum12] collection. They showed that when direct methods terminate, they can be much faster than Wiedemann Algorithm. It follows that both methods are worth trying. In practical situations, a possible workflow could be: "try a direct method; if there is too much fill-in, abort and restart with an iterative method".

Lastly, it is well-known that both methods can be combined. Performing one step of elimination reduces the size of the "remaining" trailing sub-matrix (called the Schur complement) by one, while increasing the number of non-zeros. This may decrease the time complexity of running an iterative method on the Schur complement. This strategy can be implemented as follows: "While the product of the number of remaining rows and remaining non-zeros decreases, perform an elimination step; then switch to an iterative method". For instance, one phase of the record-setting factorisation of a 768-bit number [KAF+10] was to find a few vectors on the kernel of a $2\,458\,248\,361 \times 1\,697\,618\,199$ very sparse matrix over $\mathbb{F}_2$ (with about 30 non-zero per row). A first pass of elimination steps (and discarding "bad" rows) reduced this to a roughly square matrix of size 192 796 550 with about 144 non-zero per row. A parallel implementation of the block-Wiedemann [Cop94] algorithm then finished the job.

3.2 Sparse Gaussian Elimination

We may essentially distinguish two ways of computing a gaussian elimination of a sparse matrix: one very similar to the dense gaussian elimination in which at each step, a pivot is chosen, and the coefficients below are then eliminated, by computing an appropriate linear combination of the rows. The other one, due to Gilbert and Peierls [GP88], processes the matrix row-by-row (or column-by-column). At each step, it starts with the elimination step on a given row i, and the pivot on this row is arbitrary chosen afterward. In this section, we present both of these methods and we compare them.

3.2.1 Gaussian Elimination and PLUQ Factorisation

We start this section by recalling what a PLUQ factorisation and a gaussian elimination are.

PLUQ factorisation. Given an n-by-m matrix A, of rank $r \leq \min(n, m)$, there are matrices P, L, U and Q such that $PA = LUQ$, with L lower trapezoidal n-by-r matrix with non-zero coefficients on its diagonal, U upper trapezoidal r-by-m matrix with unit coefficients on its diagonal, P is a permutation matrix acting over the rows of A, and Q a permutation matrix acting over the columns of U.

This decomposition is not unique.

Remark 1. Usually, in the literature, L has unit coefficients on its main diagonal, while U has non-zero coefficients on its main diagonal. In the paragraphs above, we choose to do it the other way round, mainly for implementation reasons.

To compute such a factorisation of A, we utilise a procedure that is called *gaussian elimination*.

Gaussian elimination. The *gaussian elimination* is an algorithm that given an n-by-m input matrix A of (unknown) rank r computes and returns a *row echelon form* U of A, that is an r-by-m upper trapezoidal matrix U whose rows are linear combinations of the rows of A. Those linear combinations can be stored, if required, in a matrix L.

Algorithm 7 Dense Gaussian Elimination.

Require: A
Ensure: L, U, such that $LU = A$
1: $L \leftarrow 0$, $U \leftarrow A$
2: $r \leftarrow \min(i, j)$
3: **for** $j = 0$ to $r - 1$ **do**
4: $\quad \mathbf{u}_j \leftarrow u_{jj}^{-1} \cdot \mathbf{u}_j$
5: $\quad l_{jj} \leftarrow u_{jj}$
6: $\quad$ **for** $i = j + 1$ to $n - 1$ **do**
7: $\quad\quad \mathbf{u}_i \leftarrow \mathbf{u}_i - u_{ij} \cdot \mathbf{u}_j$
8: $\quad\quad l_{ij} \leftarrow -u_{ij}$

We recall the usual (dense) algorithm: at each step i, choose a coefficient $a_{ij} \neq 0$ called the *pivot* and "eliminate" every coefficient under it by adding suitable multiples of row i to the rows below. A description of this procedure is given in Algorithm 7. Here, we denote by $\mathbf{u}_i$ the row of U indexed by i. For simplicity, we assume here that the matrix A is full rank, and that there is no permutation of the rows and the columns.

Remark 2. In some case, for instance if we want to compute the rank or the determinant of the matrix, keeping L in memory is not required. We only need to know U.

This method is said to be *right-looking*, because at each step it accesses the data stored at the bottom-right of A. This contrasts with *left-looking* algorithms (or the *up-looking* row-by-row equivalent described in Section 3.2.3) that accesses the data stored in the left of L. We illustrate the notions of right-looking, left-looking and up-looking in Figures 3.1a, 3.1b and 3.1c. The dark area is the echelonisation front. While the algorithms process, they only access data in the shaded area.

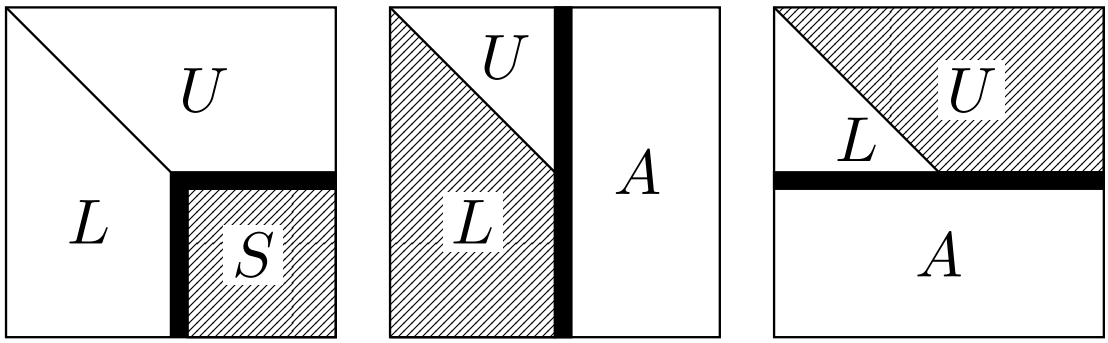
3.2.2 The Classical Right-Looking Algorithm

If we assume that we have performed k steps of the PLUQ decomposition, with $k \leq r$, then we have

$$PAQ^{-1} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}, \tag{3.3}$$

such that A_{00} is square, nonsingular and can be factored as $A_{00} = L_{00} \cdot U_{00}$. It leads to the following factorisation of A :

$$PAQ^{-1} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & \\ L_{10} & I \end{pmatrix} \cdot \begin{pmatrix} U_{00} & U_{01} \\ & S \end{pmatrix}, \tag{3.4}$$



(a) right-looking. (b) left-looking. (c) up-looking.

Figure 3.1 – Data access pattern of several PLUQ factorisation algorithms.

According to Equations 3.3 and 3.4 we have the following relations:

1. $L_{00}U_{01} = A_{01} \implies U_{01} = L_{00}^{-1}A_{01}$,
2. $L_{10}U_{00} = A_{10} \implies L_{10} = A_{10}U_{00}^{-1}$,
3. $L_{10}U_{01} + S = A_{11} \implies S = A_{11} - L_{10}U_{01}$.

From here, we first have:

$$S = A_{11} - A_{10}U_{00}^{-1}U_{01}, \tag{3.5}$$

and replacing U_{01} by the expression given above, we find that the Equation 3.5 is equivalent to:

$$S = A_{11} - A_{10}A_{00}^{-1}A_{01}. \tag{3.6}$$

S thus defined is the *Schur Complement* of A_{11} in A. This is a hopefully sparse $(n-k)$-by-$(m-k)$ matrix. We denote by $\mathbf{s}_i$ (resp. $\mathbf{u}_i$) the i-th row of S (resp. U).

The way we proceed to deal with step $k+1$ is akin to the dense case: we choose a *pivot* s_{ij} in S, and use it to cancel the coefficients $s_{\ell j}$ for all $\ell \neq i$ such that $s_{\ell j} \neq 0$. In order to do so, we replace $\mathbf{s}_\ell$ by $\mathbf{s}_\ell - (s_{\ell j}/s_{ij}) \cdot \mathbf{s}_i$, and update the permutation, so that $\mathbf{s}_{ij}$ will be on the top-left corner of S.

However, during this elimination process, some coefficients $s_{\ell t}$ that were initially zero, may become non-zero. See for instance, the example bellow:

$$\begin{pmatrix} 1 & 2 & 2 & & & -1 & \\ -1 & & & 1 & 3 & & 2 \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 2 & 2 & & & -1 & \\ & 2 & 2 & 1 & 3 & -1 & 2 \end{pmatrix}.$$

After eliminating entry $(1, 0)$ using entry $(1, 1)$ as pivot, the second row of the matrix become dense. This is what we call fill-in. It can be dealt with, up to some point, using pivoting strategies.

3.2.3 The Left-looking GPLU Algorithm

The GPLU Algorithm was introduced by Gilbert and Peierls in 1988 [GP88]. It is also abundantly described in [Dav06], up to implementation details. During the k-th step, the k-th column of L and U are computed from the k-th column of A and the previous columns of L. The algorithm thus only accesses data on the left of the echelonisation front, and the algorithm is said to be "left-looking". It does not compute the Schur complement at each stage.

We are now going to present a row-by-row (as opposed to column-by-column) variant of this method which is then "up-looking".

The main idea behind the algorithm is that the next row of L and U can both be computed by solving a triangular system. If we ignore row and column permutations, this can be derived from the following 3-by-3 block matrix expression :

$$\begin{pmatrix} L_{00} & & \\ \mathbf{l}_{10} & 1 & \\ L_{20} & \mathbf{l}_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{00} & \mathbf{u}_{01} & U_{02} \\ & u_{11} & \mathbf{u}_{12} \\ & & U_{22} \end{pmatrix} = \begin{pmatrix} A_{00} & \mathbf{a}_{01} & A_{02} \\ \mathbf{a}_{10} & a_{11} & \mathbf{a}_{12} \\ A_{20} & \mathbf{a}_{21} & A_{22} \end{pmatrix}.$$

Here $(\mathbf{a}_{10}\ a_{11}\ \mathbf{a}_{12})$ is the k-th row of A. L is assumed to have unit diagonal. Assume we have already computed $(U_{00}\ \mathbf{u}_{01}\ U_{02})$, the first k rows of U. Then we have:

$$\begin{pmatrix}\mathbf{l}_{10} & u_{11} & \mathbf{u}_{12}\end{pmatrix} \cdot \begin{pmatrix} U_{00} & \mathbf{u}_{01} & U_{02} \\ & 1 & \\ & & Id \end{pmatrix} = \begin{pmatrix}\mathbf{a}_{01} & a_{11} & \mathbf{a}_{21}\end{pmatrix}. \tag{3.7}$$

Thus, the whole PLUQ factorisation can be performed by solving a sequence of n sparse triangular systems.

Selecting the pivots. To solve the triangular system, we need to make sure that u_{11} is non-zero, only because it is how the sparse triangular solver algorithms described in section 2.3.2 work. However, it may happen that, during the process, u_{11} become zero. There are two possible cases:

1. If $u_{11} = 0$ and $\mathbf{u}_{12} \neq \mathbf{0}$, then we can permute the column j_0 where u_{11} is with another (arbitrary) one j_1, such that $j_0 < j_1$. This means that we will have to deal with a permutation Q over the columns of A all the time.

2. If $u_{11} = 0$ and $\mathbf{u}_{12} = 0$, then there is no way we can permute the columns of U to bring a non-zero coefficient on the diagonal. This means that the current row of A is a linear combination of the previous rows.

When the factorisation is over, the number of non-empty rows of U is the rank r of the matrix A.

As detailed below, the choice of the pivots is important while performing an LU factorisation. Choosing a "bad" pivot at a given step can produce too much fill-in and cause the process to abort. In the case of the GPLU algorithm the pivots selection has to be done in the dark, with the short-term objective to keep U sparse (since this keeps the triangular solver fast). It is however possible to pre-select a set of good pivots before the beginning of the procedure, so that, when the time comes, a good pivot will be chosen. This kind of pre-computation step is called *static pivoting*.

As long as U stay sparse, the GPLU algorithm performs the elimination steps very efficiently. However, aside from possibly pre-selected pivots, the GPLU algorithm does not use any strategy to choose a "good pivot" at a given step. In this case, it becomes harder to prevent fill-in to appear.

3.2.4 Pivoting Strategies

The fill-in which occurs during sparse gaussian elimination can be dealt with up to a certain point by what is called *pivoting strategies*. These strategies aim to find "a good" set of pivots, meaning a set of pivots that will reduce the fill-in during the elimination step. In the following example, choosing the top-left entry as a pivot and performing an elimination step results in a fully dense submatrix, while choosing the bottom-right entry leads to no fill-in at all.

$$\begin{pmatrix} \otimes & \times & \times & \times & \times \\ \times & \times & & & \\ \times & & \times & & \\ \times & & & \times & \\ \times & & & & \otimes \end{pmatrix}$$

However finding this set is quite hard. In fact, it has been conjectured by Rose and Tarjan [RT75], and proved later by Yannakakis [Yan81] that this is an NP-Complete optimisation problem.

Instead of finding the "best pivots", it is usual to deal with "good pivots" that can be found using heuristics.

Dynamic Pivoting.

The right-looking algorithm enjoys a clear advantage in terms of pivot selection, because it has the possibility to explore the Schur complement. The short-term objective of pivot selection is to keep

the Schur complement sparse. At each step, before performing the elimination, the "best possible" pivot is chosen in the Schur Complement and the permutations P and Q are updated.

These strategies, that consist in choosing each pivot "on-line", during the numerical factorisation are called *dynamic pivoting*. Markowitz pivoting [Mar57] chooses the next pivot in a greedy way, to minimise fill-in caused by the next elimination step while excluding those that would lead to numerical instability. At each step, the Markowitz pivoting strategy chooses a pivot a_{ij} that minimise the product $(r_i - 1)(c_j - 1)$, where r_i (resp. c_j) is the number of non zero elements on the row i (resp. the column j) of the Schur Complement. This product is called the *Markowitz cost*. Furthermore, when utilised to solve a sparse numerical system, the pivot must also be chosen to preserve numerical stability. Indeed, a_{ij} can be chosen as a pivot only if $|a_{ij}| \geq u \max_k |a_{kj}|$, for some parameter $0 < u \leq 1$.

Dynamic pivoting strategies are more likely to reduce fill-in, however, searching for the "best possible" pivot at each step takes a lot of time.

Static Pivoting.

In this case, the choice of the pivots is made in advance, before any elimination step is performed, and does not take much time.

For instance, in the numerical world, when the input matrix is square and invertible but not symmetric, pivots are chosen to maintain both sparsity and numerical accuracy. As such, the actual choice of pivots will ultimately depend on numerical values that are not known in advance, and choosing an a priori sequence of pivots is quite difficult. One option is to look at the structure of $A + {}^tA$ or $A{}^tA$. These two matrices are symmetric. When a matrix $\tilde{A}$ is symmetric positive definite, a sparse Cholesky factorisation $\tilde{A} = {}^tLL$ can be computed. In that case, the pivots are on the diagonal, and many algorithms have been designed to select an a priori order on the pivots, meaning a permutation such that ${}^tP\tilde{A}P$ is easier to factorise than $\tilde{A}$. Approximate Minimum Degree [ADD96] or Nested Dissection [Geo73] are such algorithms. They only exploit the structure of A (the locations of non-zero entries) and disregard the numerical values. The idea is to use these symmetric pivot-selection algorithms to define an order in which the rows of A will be processed. Then, during the numerical factorisation, choose inside each row the pivot that maximises accuracy (this is called "partial pivoting").

Structural pivots. When the coefficients of the input matrix live in an exact field such as $\mathbb{Q}$ or $\mathbb{Z}_p$, numerical accuracy is no longer a problem. However, numerical cancellation becomes much more likely (and *does* occur if A is rectangular or rank-deficient). We must then be careful not to choose as pivot a coefficient that may be eliminated during the procedure.

Entries that can be used as pivots can sometimes be identified based solely on the pattern of non-zero entries. In the following matrix, circled entries can be chosen as pivots regardless of the actual values. We call these *structural pivots*.

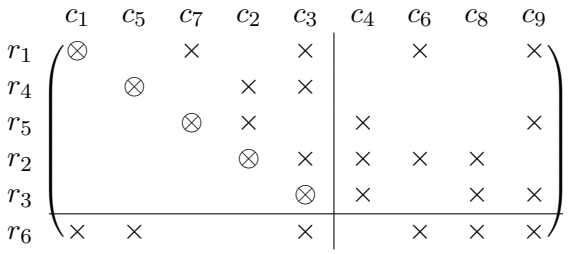
Example 3.1.

$$\begin{array}{c} \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \end{array} \begin{array}{c} \begin{array}{ccccccccc} c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 & c_9 \end{array} \\ \begin{pmatrix} \otimes & & \times & & & \times & \times & & \times \\ & \otimes & \times & \times & & \times & & \times & \\ & & \otimes & \times & & & & \times & \times \\ & \times & \times & & \otimes & & & & \\ & \times & & \times & & & \otimes & & \times \\ \times & & \times & & \times & \times & & \times & \times \end{pmatrix} \end{array}$$

Table 3.1 – Running time (in seconds) of the two algorithms on extreme cases.

Matrix	rows	columns	non-zero	rank	Right-Looking	GPLU
A	100 000	1 000	995 076	1000	3.0	0.02
B	100 000	1 000	4 776 477	200	1.7	9.5

The rows and columns can be permuted as follows:



A set of k structural pivots has been identified if the rows and columns of the matrix can be permuted such that the top-left $k \times k$ submatrix is upper-triangular, as shown in this example.

3.2.5 Left or Right?

We discuss the efficiencies of the right-looking gaussian elimination algorithm implemented in LinBox and our implementation of the GPLU algorithm in SpaSM.

These two algorithms are incomparable. We illustrate this by exhibiting two situations where each one consistently outperforms the other. We generated two random matrices with different properties (situations A and B in Table 3.1). The entries of matrix A are identically and independently distributed. There are zero with probability 99%, and chosen uniformly at random modulo p otherwise. GPLU terminates very quickly in this case because it only process the first ≈ 1000 rows before stopping, having detected that the matrix has full column-rank.

In matrix B, one row over 1000 is random (as in matrix A), and the 999 remaining ones are sparse linear combinations of a fixed set of 100 other random sparse rows (as in matrix A). The right-looking algorithm discovers these linear combinations early and quickly skips over empty rows in the Schur complement. On the other hand, GPLU has to work its way throughout the whole matrix.

We now give further "real" examples of the behaviour of these two algorithms, using matrices from [Dum12]. We quickly observed that no algorithm is always consistently faster than the other; one may terminate instantly while the other may run for a long time and vice-versa. In order to perform a systematic comparison, we decided to set an arbitrary threshold of 60 seconds, and count the number of matrices that could be processed in this amount of time. LinBox could dispatch 579 matrices, while SpaSM processed 606. Amongst these, 568 matrices could be dealt with by both algorithms in less than 60s. This took 1100s to LinBox and 463s to SpaSM. LinBox was faster 112 times, and SpaSM 456 times. These matrices are "easy" for both algorithms, and thus we will not consider them anymore.

Table 3.2 shows the cases where one algorithm took less than 60s while the other took more. There are cases where each of the two algorithms is catastrophically slower than the other. We conclude there is no clear winner (even if GPLU is usually a bit faster). The hybrid algorithm described in the following chapter outperforms both.

3.3 Implementations of Sparse Gaussian Elimination

3.3.1 In the Numerical World

A large body of work has been dedicated to sparse direct methods by the numerical computation community. Direct sparse numerical solvers have been developed during the 1970's, so that several

Table 3.2 – Comparison of sparse elimination techniques. Times are in seconds.

Matrix	Right-looking	GPLU
`Franz/47104x30144bis`	39	488
`G5/IG5-14`	0.5	70
`G5/IG5-15`	1.6	288
`G5/IG5-18`	29	109
`GL7d/GL7d13`	11	806
`GL7d/GL7d24`	34	276
`Margulies/cat_ears_4_4`	3	184
`Margulies/flower_7_4`	7.5	667
`Margulies/flower_8_4`	37	9355
`Mgn/M0,6.data/M0,6-D6`	45	8755
`Homology/ch7-8.b4`	173	0.2
`Homology/ch7-8.b5`	611	45
`Homology/ch7-9.b4`	762	0.4
`Homology/ch7-9.b5`	3084	8.2
`Homology/ch8-8.b4`	1022	0.4
`Homology/ch8-8.b5`	5160	6
`Homology/n4c6.b7`	223	0.1
`Homology/n4c6.b8`	441	0.2
`Homology/n4c6.b9`	490	0.3
`Homology/n4c6.b10`	252	0.3
`Homology/mk12.b4`	72	9.2
`Homology/shar_te2.b2`	94	1
`Kocay/Trec14`	80	31
`Margulies/wheel_601`	7040	4
`Mgn/M0,6.data/M0,6-D11`	722	0.4
`Smooshed/olivermatrix.2`	75	0.6

software packages were ready-to-use in the early 1980's (MA28, SPARSPAK, YSMP, ...). For instance, MA28 is an early "right-looking" sparse LU factorisation code described in [DER89, DR79]. It uses Markowitz pivoting [Mar57] to choose pivots in a way that maintains sparsity.

Most of these direct solvers start with a symbolic analysis phase that ignores the numerical values and just examines the pattern of the matrix. Its purpose is to predict the amount of fill-in that is likely to occur, and to pre-allocate data structures to hold the result of the numerical computation. The complexity of this step often dominated the running time of early sparse codes. In addition, an important step was to choose *a priori* an order in which the rows (or columns) were to be processed in order to maintain sparsity.

Sparse linear algebra often suffers from poor computational efficiency, because of irregular memory accesses and cache misses. More sophisticated direct solvers try to counter this by using *dense* linear algebra kernels (the BLAS and LAPACK) which have a much higher FLOPS (Floating-point Operation Per Second) rate.

The supernodal method [AGL+87] works by clustering together rows (or columns) with similar sparsity pattern, yielding the so-called *supernodes*, and processing them all at once using dense techniques. Modern supernodal codes include CHOLDMOD [CDHR08] (for Cholesky factorisation) and SuperLU [DEG+99]. The former is used in Matlab on symmetric positive definite matrices.

In the same vein, the multi-frontal method [DR83] turns the computation of a sparse LU factorisation into several, hopefully small, *dense* LU factorisations. The starting point of this method is the observation that the elimination of a pivot creates a completely dense sub-matrix in the Schur complement. Contemporary implementations of this method are UMFPACK [Dav04] and MUMPS [ADKL01]. The former is also used in Matlab in non-symmetric matrices.

Finally some implementation of left-looking LU algorithms are also available, for instance, in the SPARSPAK library cited earlier. The GPLU algorithm is implemented in Matlab, and is used for very sparse non symmetric matrices. It is also the heart of the specialised library called KLU [DN10], dedicated to circuit simulation. Another implementation is also available in the CSPARSE library.

3.3.2 In Exact Linear Algebra

The world of exact sparse direct methods is much less populated. Besides LinBox, we are not aware of many implementations of sparse gaussian elimination capable of computing the rank of a matrix modulo p. According to its handbook, the MAGMA [BCP97] computer algebra system computes the rank of a sparse matrix by first performing sparse gaussian elimination with Markowitz pivoting, then switching to a dense factorisation when the matrix becomes dense enough. The Sage [Sag13] system uses LinBox.

Some specific applications rely on exact sparse linear algebra. All competitive factoring and discrete logarithms algorithms work by finding a few vectors in the kernel of a large sparse matrix. Some controlled elimination steps are usually performed, which make the matrix smaller and denser. This has been called "structured gaussian elimination" [LO90]. The idea is to declare some columns that have a "large number" of non-zero coefficients as "heavy", and to work so that the sparsity will be preserved only on the remaining "light" columns. It basically works as follows:

1. Remove all columns that only have one non-zero coefficient, and the rows in which these non-zero coefficients are.
2. Among the light columns, select the heaviest and declare them as "heavy".
3. Delete some rows that have a large number of non-zero coefficients in the "heavy" columns.
4. For each row, which consists only of one coefficient that is either 1 or -1 on a "light" column, subtract appropriate multiples of this row from all other rows that have a non-zero coefficient on said light column, in order to make these coefficients zero.

The process can be continued until the matrix is fully dense (after which it is handled by a dense solver), or stopped earlier, when the resulting matrix is handled by an iterative algorithm. The current state-of-the-art factoring codes, such as CADO-NFS [CAD15], seem to use the sparse elimination techniques described in [Cav00].

Modern Gröbner basis algorithms [Fau99, Fau02] work by computing the reduced row-echelon form of particular sparse matrices. An ad hoc algorithm has been designed, exploiting the mostly triangular structure of these matrices to find pivots without performing any computation [FL10] (more detail about this method is given in Section 4.1.2). An open-source implementation, GBLA [BEF+16], is available.

SpaSM. While working on this topic, we developed the SpaSM software library (SPArse Solver Modulo p). Its code is publicly available in a repository hosted at:

`https://github.com/cbouilla/spasm`

This software is implemented in C. The matrices are stored in the CSR format. Our code is heavily inspired by CSPARSE ("A Concise Sparse Matrix Package in C"), written by Davis and abundantly described in his book [Dav06]. It has been modified to work row-wise (as opposed to column-wise), and more importantly to deal with non-square or singular matrices. It also provides an implementation of the GPLU algorithm. In its default setting, it proceeds the following way: transpose the matrix if it has more columns than rows, use a greedy structural pivots selection heuristic we proposed in [BDV17], before actually starting the factorisation.

All the algorithms that we will describe in the two following chapters have been implemented in this library. We essentially focussed on the computation of the rank of sparse matrices, with minor adaptations, it can also solve other related problems (e.g. solving sparse systems, computation of the determinant, finding a basis of the kernel).

Chapter 4

Of a New Hybrid Algorithm: Getting the Best of Two Worlds

In this chapter we present a new sparse gaussian elimination algorithm, that combines strategies from both the classical right-looking Algorithm and the GPLU Algorithm. Using a simple pivots selection heuristic due to Faugère and Lachartre, we are able to outperform both of the two methods afore-mentioned. We also compare ourself with the Wiedemann Algorithm from LinBox on large matrices, and figure out that in some case our new algorithm is up to 100× faster. We focus on the computation of the rank, as it raises the same challenge as solving systems while outputting only a single integer. This is basically the work presented in a publication at CASC 2016, with Charles Bouillaguet [BD16].

4.1 A New Hybrid Algorithm

The right-looking method chooses the pivot in the Schur Complement step by step. It can greedily choose the "best" pivot possible (i.e. the one that produces minimum fill-in in next step Schur Complement). However, using such heuristic at each step takes time. In the GPLU algorithm, the choice is done once and for all and so does not take much time, but the choice of pivots at a given step is not the best possible. In this section, we introduce an algorithm which is a good compromise between right-looking and left-looking methods.

4.1.1 High Level Idea of the Procedure

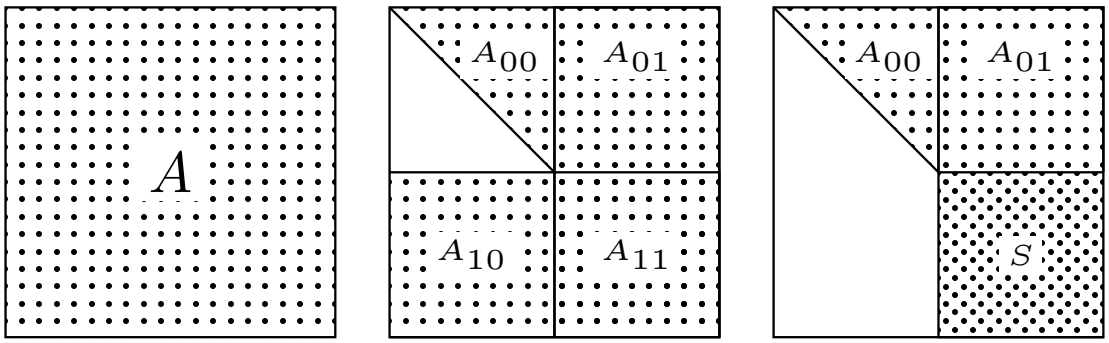
The main idea behind it is the following:

1. Find a set of "a priori" pivots in A, without performing any arithmetical operations.
2. If possible, compute the Schur complement S of A with respect to these pivots.
3. Compute the rank of S by any means (including recursively).

In the first step, we search for permutations P and Q of the rows and the columns of A, such that a triangular submatrix, with non-zero coefficients on the diagonal appears on the top left of A. The coefficients on the diagonal are pivots of A, and they can be used to eliminated all coefficients bellow.

The first two steps of this procedure are represented in Figure 4.1. The rows and the columns of the input matrix A (Figure 4.1a) are permuted so that a triangular structure appears on the top of the matrix (Figure 4.1b). Then the elimination step is performed to compute S in one fell swoop (Figure 4.1c).

It is easy to estimate the size of S. If A is an n-by-m matrix, and if k pivots have been found during the first step of the procedure, the S will be an $(n-k)$-by-$(m-k)$ matrix. If it is small



(a) Original Matrix. (b) Structural Pivots. (c) Schur Complement.

Figure 4.1 – Structural pivots and Schur Complement computation.

.

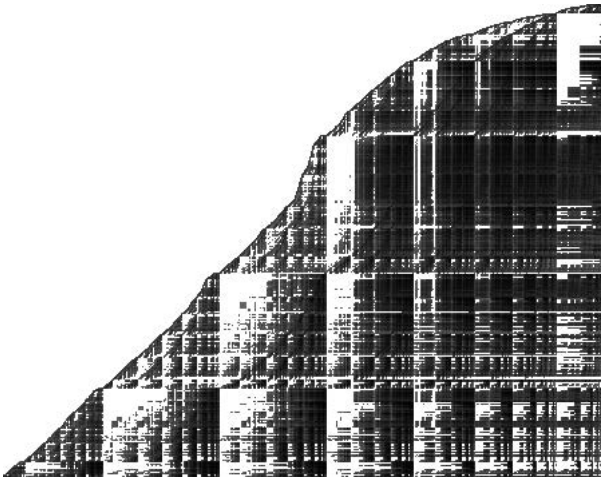


Figure 4.2 – Example of a Gröbner Basis matrix: `Grobner/c8_mat11` from [Dum12].

enough, it can be stored as a dense matrix, and we can compute its rank using dense gaussian elimination.

If S is large, there are two possible cases: S is either sparse enough to be computed and stored on the CSR format, or S is dense. As we will discuss in the next section, it is possible to estimate the density of S beforehand, and adapt our algorithm accordingly.

In the first case, several options are possible. We can either use the same technique recursively, or switch to GPLU, or switch to the Wiedemann algorithm. Some guesswork is required to find the best strategy. By default we found that allowing only a limited number of recursive calls (usually less than 10, often 3) and the switching to GPLU yields good results.

If S is big and dense, in most of the cases there is nothing we can do. In these cases, we have to abort and report failure. However, there are some particular cases (when the rank of S is very small), where we can still compute the rank of S without actually forming the matrix [SY09].

4.1.2 Initial Choice of Structural Pivots: The Faugère-Lachartre Heuristic

In [BD16], we propose to utilise the Faugère-Lachartre (FL) heuristic [FL10], as it is simple, easy to implement, and already leads to good results, while remaining fast.

At the aim of computing the row-echelon form of sparse matrices arising from Gröbner basis computations, Faugère and Lachartre [FL10] have observed that the leftmost entry of each row can be chosen as a pivot if no other pivot has previously been chosen on the same column. Because the Gröbner-basis matrices are nearly triangular (see Figure 4.2), this strategy often finds 99.9% of the maximum number of pivots of these particular matrices.

The idea of the procedure is the following: Each row is mapped to the column of its leftmost coefficient. All these coefficients are then “candidate pivots”. When several rows have the same leftmost coefficient, we select the candidate on the sparsest row. Finally we move the selected rows

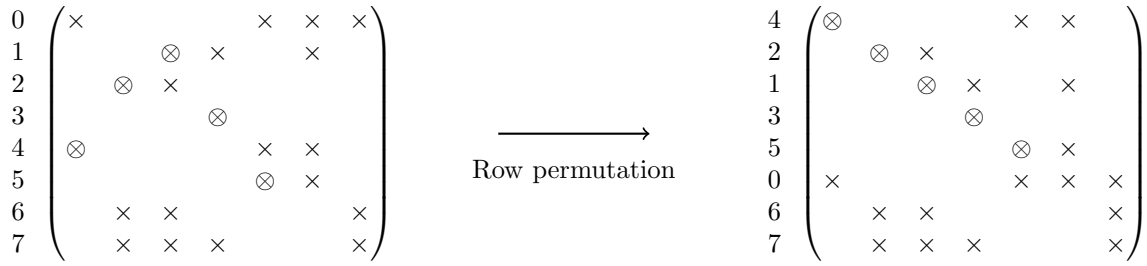


Figure 4.3 – Illustration of the Faugère-Lachartre heuristic.

before the others and sort them by increasing position of the leftmost coefficient.

We illustrate this method in Figure 4.3. Rows 0 and 4 have their leftmost coefficient on the same column. They are both "candidates". As row 4 contains more zero-coefficient, we choose $(4, 0)$ as a pivot, to keep the matrix U as sparse as possible. Same goes for rows 2, 6 and 7. Permuting the rows in ascending order gives a upper-triangular leading principal sub-matrix.

Finding more structural pivots. We claim that being able to find many structural pivots is desirable. In the extreme case, if *all* pivots can be found based on the structure of the matrix, then the echelonisation process can be carried over without any arithmetic operation, just by permuting the rows and columns — and without any fill-in.

Unfortunately, finding the largest set of structural pivots is an NP-Complete optimisation problem [GHL01]. This is why we have to be content with a hopefully large set of pivots, that can be found using heuristics. In the next chapter, we will discuss more about this problem, and present some new strategies we designed in order to find larger sets, and thus improve the whole procedure.

4.2 Dealing with the Schur Complement

4.2.1 Computation

If k structural pivots are found in the first step of the procedure, we are given permutation P and Q such that the k-by-k upper-left block of PAQ^{-1} is upper-triangular with non-zero diagonal and A can be decomposed as follows:

$$PAQ^{-1} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & \\ & I \end{pmatrix} \cdot \begin{pmatrix} U_{00} & U_{01} \\ A_{10} & A_{11} \end{pmatrix},$$

where L_{00} is a diagonal matrix with non-zero coefficients on its diagonal, and U_{00} is upper triangular with unit coefficients on its diagonal. This is in fact the part of U provided by the Faugère-Lachartre heuristic. What we aim to compute is $S = A_{11} - A_{10}U_{00}^{-1}U_{01}$, the Schur Complement of A with respect to U_{00}. We can compute S row-by-row. To obtain the ith row, $\mathbf{s}_i$ of S, we follow the same idea than the one of the GPLU algorithm. Let us denote by $(\mathbf{a}_{i0}\ \mathbf{a}_{i1})$ the ith row of $(A_{10}\ A_{11})$ and consider the following system:

$$(\mathbf{x}_0\ \mathbf{x}_1) \cdot \begin{pmatrix} U_{00} & U_{01} \\ & Id \end{pmatrix} = (\mathbf{a}_{i0}\ \mathbf{a}_{i1}). \tag{4.1}$$

We get $\mathbf{x}_1 = \mathbf{a}_{i1} - \mathbf{x}_0 U_{01} = \mathbf{a}_{i1} - \mathbf{a}_{i0}U_{00}^{-1}U_{01}$. From the definition of S given by Equation 3.5, we have $\mathbf{x}_1 = \mathbf{s}_i$. Thus S can be computed solving a succession of sparse triangular systems. Because we chose U to be as sparse as possible, this process is fast. Furthermore, as the rows of S can be computed independently (unlike the GPLU algorithm), it is possible to parallelise the computation of S. It is also possible to estimate the density of S beforehand. This is how we proceed:

1. Pick t random rows of the matrix $(A_{10}\ A_{11})$, for some parameter $t < n - k$.
2. For each of these rows $(\mathbf{a}_{i0}\ \mathbf{a}_{i1})$, solve the system 4.1 and compute the density of $\mathbf{s}_i$.

3. Compute the average density.

If the t rows are randomly chosen, we can hope that this will give us a relatively good estimation of the density of S. If this estimated density is lower than this threshold, then we can obtain it entirely following the idea described above. If it is greater than some threshold, for instance 0.1, we decide that S would be to dense to be computed. In this case the situation is more complicated, and most of the time, our algorithm is likely to fail.

4.2.2 Special Cases

Useful results. We assume that we have found k pivots already, and denote by M a t-by-$n-k$ random matrix. We have:

$$M \cdot \begin{pmatrix} A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} A'_0 & A'_1 \end{pmatrix}.$$

Let $X = (X_0\ S')$ be a matrix satisfying the following equation.

$$\begin{pmatrix} X_0 & S' \end{pmatrix} \cdot \begin{pmatrix} U_{00} & U_{01} \\ & I \end{pmatrix} = \begin{pmatrix} A'_0 & A'_1 \end{pmatrix}.$$

We have $S' = A'_1 - A'_0 U_{00}^{-1} U_{01}$. According to the definition of A'_0 and A_1, we obtain:

$$S' = M(A_{11} - A_{10} U_{00}^{-1} U_{01}),$$

and thus we have:

$$S' = MS \tag{4.2}$$

Remark 1. We can easily see that the rank $r_{S'}$ of S' is at most equal to the rank r_S of S. Recall that the rank of a matrix is equal to the dimension of the vector space spanned by its rows. Let $\mathscr{C}_S$ be the vector subspace of $\mathbb{F}_p$ spanned by the rows of S, and $\mathscr{C}_{S'}$ be the vector subspace of $\mathbb{F}_p$ spanned by the rows of S'. We can quickly check that $\mathscr{C}_{S'} \subseteq \mathscr{C}_S$. In fact For all $\mathbf{x}$ in $\mathscr{C}_{S'}$, there is a $\mathbf{y}$ such that $\mathbf{x} = \mathbf{y}S'$, and then $\mathbf{x} = (\mathbf{y}M)S$ and thus $x \in \mathscr{C}_S$. It is enough to prove $r_{S'} \leq r_S$.

We claim that if M is an n-by-m full rank matrix, with $n \geq m$, then this is indeed an equality. It is quite easy to see that when M is square and thus invertible. In this case, $S = M^{-1}S'$. Then, from Remark 1, we have $r_S \leq r_{S'}$, and thus $r_S = r_{S'}$.

The case $n > m$ is slightly more difficult to prove. We claim that in this case, there still is an m-by-n matrix $\bar{M}$ such that $S = \bar{M}S'$, and thus we are still able to use the result from Remark 1 to prove the equality. Indeed, if M is full rank, then there is an m-by-m sub-matrix of M_0 such that M_0 is invertible. We denote by P the permutation that pushes this sub-matrix on the top of M. We denote by M_1 the sub-matrix of M such that:

$$PM = \begin{pmatrix} M_0 \\ M_1 \end{pmatrix}.$$

Let $\bar{M}$ be the matrix defined as follows:

$$\bar{M} = \begin{pmatrix} M_0^{-1} & 0 \end{pmatrix} \cdot P.$$

We have:

$$\begin{aligned} \bar{M} \cdot M &= \begin{pmatrix} M_0^{-1} & 0 \end{pmatrix} \cdot (PM) \\ &= \begin{pmatrix} M_0^{-1} & 0 \end{pmatrix} \cdot \begin{pmatrix} M_0 \\ M_1 \end{pmatrix} \\ &= I_m. \end{aligned} \tag{4.3}$$

It follows that $\bar{M}S' = \bar{M}MS = S$.

Unfortunately, this result cannot be extended to the case where M is an n-by-m full rank matrix with $n \leq m$. Indeed, it is a fact that in this case, there is no m-by-n matrix $\bar{M}$ such that $\bar{M}M = I_m$. Unluckily, this is exactly the case that matters to us in the current situation.

We can still prove the following heuristic result:

Lemma 4.1. *If M is an n-by-m random matrix with $n < m$, such that $S' = MS$. If we denote r_S and $r_{S'}$, respectively the rank of S and S', then, with high probability, $r_S \geq r_{S'}$.*

Proof. Let us denote by S_0 an arbitrary r_S-by-m sub-matrix of S, such that S_0 spans the vectorial subspace $\mathscr{C}_S$. Let $\mathcal{I}$ be the subset of $\{0 \dots n-1\}$ such that S_0 is induced by the rows indexed by $\mathcal{I}$. We denote S_1 the sub-matrix of S induced by the rows from $\{0 \dots n-1\} \setminus \mathcal{I}$. Let M_0 be the m-by-r_S sub-matrix induced by the columns from $\mathcal{I}$, and let M_1 be the sub-matrix induced by the columns from $\{0 \dots n-1\} \setminus \mathcal{I}$. We have:

$$S' = M_0 \cdot S_0 + M_1 \cdot S_1. \tag{4.4}$$

As $\mathsf{Span}\{S_0\} = \mathscr{C}_S$, in particular any rows of S_1 is actually a vector from $\mathsf{Span}\{S_0\}$. Thus, there is an $(n-r)$-by-r_S matrix R, such that:

$$S_1 = RS_0.$$

It follows that:

$$S' = (M_0 + M_1 R) \cdot S_0. \tag{4.5}$$

We denote M' the matrix $M_0 + M_1 R$. This is an m-by-r_S matrix, with $m \geq r_S$. Furthermore, if M is a random matrix, then M' will also be a random matrix, as the linear combination of two random matrices, M_0 and M_1. Thus with hight probability M' will be full rank. Indeed, the probability that a random m-by-k matrix, with $m \leq k$ is full is given by:

$$\begin{aligned}
\mathbb{P}[\text{matrix is full rank}] &= \frac{\prod_{i=0}^{m}(q^k - q^i)}{q^{km}} \\
&\geq \frac{(q^k - q^{m-1})^m}{q^{km}} \\
&\geq \frac{q^{km}(1 - q^{m-1-k})^m}{q^{km}} \\
&\geq (1 - q^{m-1-k})^m
\end{aligned}$$

In our case, M' is then full rank with probability greater than $(1 - q^{m-1-r_s})$. From the discussion above, it follows that the rank of S' is equal to the rank of S_0 with high probability. As the rank of S_0 is indeed r_S, this concludes the proof. □

Tall and skinny Schur complement. It is beneficial to consider a special case, when S has much more rows than columns (we transpose S if it has much more columns than rows).

This situation can be detected as soon as the pivots have been selected, because we know that S has size $(n-k) \times (m-k)$. In this case where S is very tall and very narrow, it is wasteful to compute S entirely, even if S is sparse. For instance, a naive solution consists in choosing a small constant ϵ, building a dense matrix S' of size $(m-k+\epsilon)$-by-$(m-k)$ with random (dense) linear combinations of the rows of S, and computing its rank using dense linear algebra. S' can be formed, using the trick described above by taking random linear combinations of the remaining rows of A, and then solving triangular system, just like before.

In fact, denoting by M the $(m-k+\epsilon)$-by-$(n-k)$ matrix which represents these linear combinations, it follows from Equation 4.2 that $S' = MS$. Furthermore, as discussed above, if we denote r'_S the rank of S' and r_S the rank of S, we have $r_{S'} = r_S$, with high probability.

Large and dense Schur complement with small rank. As mentioned before, when S is large and dense, our procedure will most likely fail. There is however, a favourable case that arises when the number of structural pivots found during the first step of the algorithm is close the rank of A. In this case, the rank of S will be small and the L and U matrices obtained during the PLUQ factorisation will be very thin, and we can hope to store them in memory. In order to compute the rank of S without actually computing S, we implemented the following technique:

1. Guess an upper-bound r_{max} on the rank of S,
2. Choose a random r_{max}-by-$(n-k)$ matrix M,
3. Compute $S' = M \times S$ (dense but small),
4. Compute a dense gaussian elimination of S',
5. If S' has full rank, set r_{max} to $2r_{max}$ and restart the procedure.

4.3 Implementation and Results

All the experiments we carried on an Intel core i7-3770 with 8GB of RAM. Only one core was ever used. We used J.-G. Dumas's Sparse Integer Matrix Collection [Dum12] as benchmark matrices. We restricted our attention to the 660 matrices with integer coefficients. Most of these matrices are small and their rank is easy to compute. Some others are pretty large. In all cases, their rank is known, as it could always be computed using the Wiedemann algorithm. We fixed $p = 42013$ in all tests.

We give here the results we presented in [BD16]. Using the refined pivots selection algorithm from [BDV17], described in the next chapter, we were able to outperformed these results, especially for the largest matrices from [Dum12] as discussed in Section 5.4.

Comparison with other direct methods. In section 3.2.5, we claimed that our algorithm outperforms both the right-looking algorithm of LinBox and our own implementation of GPLU. Indeed, in Table 4.1, we present the timing of our algorithm along with the timing of the fastest of the two afore-mentioned methods, for the same set of matrices than in section 3.2.5. We can see that our algorithm is almost always the fastest, and always at least as fast as the best previous algorithm (which in this case appears to be GPLU), save for the case of the matrix `Mgn/M0,6.data/M0,6-D11`, where the GPLU algorithm is 0.2 second faster than our algorithm in our benchmark.

Remark 1. We can also wonder how our algorithm compares to GBLA [BEF+16] and to the SGE implemented in CADO-NFS [CAD15]. This is difficult, because they are both tailored for very specific matrices. GBLA has been designed for matrices arising in Gröbner basis computations, and exploits their specific shape. For instance, they have (hopefully small) *dense* areas, which GBLA rightly store in dense data structures. This phenomenon does not occur in our benchmark collection, which is ill-suited to GBLA. GBLA is nevertheless undoubtedly more efficient on Gröbner basis matrices. Something similar can be said about the SGE, which also exploits the fact that some columns are dense, and do not care about the fill-in in these areas.

Direct methods vs. iterative methods. We now turn our attention to the remaining matrices of the collection, the "hard" ones. Some of these are yet unnameable to any form of elimination, because they cause too much fill-in. However, some matrices can be processed much faster by our hybrid algorithm than by any other existing method.

For instance, `relat9` is the third largest matrix of the collection; computing its rank takes a little more than two days using the Wiedemann algorithm. Using the "Tall and Narrow Schur Complement" technique described in Section 4.2.2, the hybrid algorithm computes its rank in 34 minutes. Most of this time is spent in forming a size-8937 dense matrix by computing random dense linear combinations of the 9 million remaining sparse rows. The rank of the dense matrix is quickly computed using the `Rank` function of FFLAS-FFPACK [FFL14]. A straightforward parallelisation

Table 4.1 – Comparison of sparse elimination techniques. Times are in seconds.

Matrix	Best(LinBox, GPLU)	Hybrid
Franz/47104x30144bis	39	1.7
G5/IG5-14	0.5	0.4
G5/IG5-15	1.6	1.1
G5/IG5-18	29	8
GL7d/GL7d13	11	0.3
GL7d/GL7d24	34	11.6
Margulies/cat_ears_4_4	3	0.1
Margulies/flower_7_4	7.5	2.5
Margulies/flower_8_4	37	3.7
Mgn/M0,6.data/M0,6-D6	45	0.1
Homology/ch7-8.b4	0.2	0.2
Homology/ch7-8.b5	45	10.7
Homology/ch7-9.b4	0.4	0.4
Homology/ch7-9.b5	8.2	3.4
Homology/ch8-8.b4	0.4	0.5
Homology/ch8-8.b5	6	2.9
Homology/n4c6.b7	0.1	0.1
Homology/n4c6.b8	0.2	0.2
Homology/n4c6.b9	0.3	0.2
Homology/n4c6.b10	0.3	0.2
Homology/mk12.b4	9.2	1.5
Homology/shar_te2.b2	1	0.2
Kocay/Trec14	31	4
Margulies/wheel_601	4	0.3
Mgn/M0,6.data/M0,6-D11	0.4	0.6
Smooshed/olivermatrix.2	0.6	0.1

Table 4.2 – Some harder matrices. Times are in seconds. M.T. stands for "Memory Thrashing".

Matrix	n	m	$\|A\|$	Right-looking	Wiedemann	Hybrid
kneser_10_4	349651	330751	992252	923	9449	0.1
mk13.b5	135135	270270	810810	M.T.	3304	41
M0,6-D6	49800	291960	1066320	42	979	0.1
M0,6-D7	294480	861930	4325040	2257	20397	0.8
M0,6-D8	862290	1395840	8789040	20274	133681	7.7
M0,6-D9	1395480	1274688	9568080	M.T.	154314	27.6
M0,6-D10	1270368	616320	5342400	22138	67336	42.7
M0,6-D11	587520	122880	1203840	722	4864	0.5
relat8	345688	12347	1334038	M.T.	244	2
rel9	5921786	274667	23667185	M.T.	127675	1204
relat9	9746232	274667	38955420	M.T.	176694	2024

using OpenMP brings this down to 10 minutes using the 4 cores of our workstation. The same goes for the `rel9` matrix, which is similar.

The rank of the `M0,6-D9` matrix, which is the 9-th largest of the collection, could not be computed by the right-looking algorithm within the limit of 8GB of memory. It takes 42 hours to the Wiedemann algorithm to find its rank. The standard version of the hybrid algorithm finds it in less than 30 seconds.

The `shar_te.b3` matrix is an interesting case. It is a very sparse matrix of size 200200 with only 4 non-zero entries per row. Its rank is 168310. The right-looking algorithm fails, and the Wiedemann algorithm takes 3650s. Both GPLU and the hybrid algorithm terminate in more than 5 hours. However, performing *one* iteration of the hybrid algorithm computes a Schur complement of size 134645 with 7.4 non-zero entries per row on average. We see that the quantity $n|A|$, a rough indicator of the complexity of iterative methods, decreases a little. Indeed, computing the rank of the first Schur complement takes 2422s using the Wiedemann algorithm. This results in a 1.5× speed-up.

All-in-all, the hybrid algorithm is capable of quickly computing the rank of the 3rd, 6th, 9th, 11th and 13th largest matrices of the collection, whereas previous elimination techniques could not. Previously, the only possible option was the Wiedemann algorithm. The hybrid algorithm allows large speedup of 100×, 1000× and 10000× in these cases.

Chapter 5

Of Better Pivots Selections Heuristics

In this chapter, we recall that finding a set of structural pivots is equivalent to find a matching without alternating cycles in a bipartite graph. Unfortunately this problem has been proved to be an NP-Complete optimisation problem by Golumbic, Hirst and Lewenstein. We then present two heuristics that allow us to find an hopefully large set of structural pivots. The second one, in particular, has proved to be very efficient allowing us to recover up to 99.9% of the total number of pivots in some cases. This work has led to a publication at PASCO 2017 [BDV17] in a joined work with Charles Bouillaguet and Marie-Emilie Voge.

5.1 Searching for Structural Pivots

Given a matrix A, and a set of pivots $\mathcal{P}$ of A, We call *pivotal row* any row i such that there exists a j such that $a_{ij} \in \mathcal{P}$. We define a *pivotal column* accordingly.

5.1.1 Bi-Partite graph, Matching, and Alternate Path

Because two pivots cannot be on the same row, nor on the same column, we claim that $\mathcal{P}$ induces a matching in $\mathcal{G}$. This holds true, for any set of pivots, be they chosen a priori or during the factorisation. Indeed, $\mathcal{P} = \{a_{i_1,j_1} \dots a_{i_k,j_k}\}$ for some $k \geq 1$, such that all the i indexes on one hand and all the j indexes on the other hand are distinct. Rewriting this in terms of graph, this gives $\mathcal{P} = \{(r_{i_1}, c_{j_1}), \dots (r_{i_k}, c_{j_k})\}$, such the same node r_i (resp. c_j) does not appear several times.

As such, the size of a maximal matching gives an upper-bound on the rank of a sparse matrix. This bound is however not tight. For instance, in the following matrix,

$$\begin{pmatrix} \textcircled{1} & 1 & 1 \\ 1 & \textcircled{1} & 1 \end{pmatrix},$$

the two circled entries form a maximal matching between rows and columns, yet the matrix has rank 1. The elimination of the first entry makes the second candidate pivot disappear.

In terms of matching, the definition of structural pivots implies the absence of *alternating cycles* on the graph, with respect to the matching defined by the pivots. Given a matching, $\mathcal{M}$, an alternating cycle is a cycle such that one edge over two is in $\mathcal{M}$ and one edge over two is not. A matching that does not induce alternating cycles is said to be *uniquely restricted* [GHL01], because it is the single matching between these vertices in the graph.

Structural pivots and uniquely restricted matching. We now recall an important theorem mentioned in [HS93, GHL01], which formalise the link between uniquely restricted matching (from now on denoted by URM) and structural pivots.

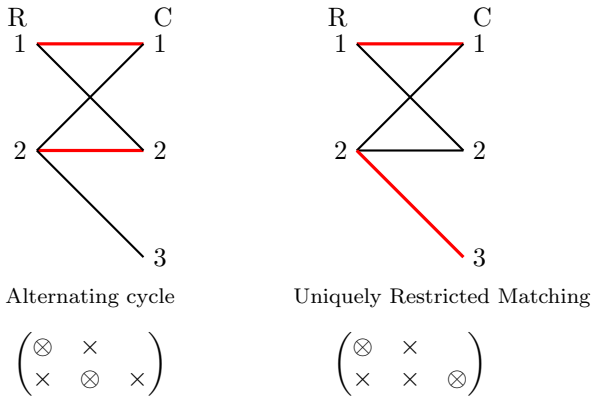


Figure 5.1 – Matching in bi-partite graph, and matrix representation.

Theorem 5.1. *There exist row and column permutations P and Q such that the $k \times k$ principal submatrix of PAQ is upper-triangular if and only if there exists a uniquely restricted matching of size k in the bipartite graph $\mathcal{G}$ associated to A.*

This theorem is not original. Nevertheless, we are going to prove it again, for completeness.

Here is an idea of the proof of this theorem. Given a matching $\mathcal{M} = \{(r_1, c_1), \dots, (r_k, c_k)\}$ over a bipartite graph $\mathcal{G}$, we form the directed graph $\mathcal{G}_\mathcal{M}$ as follows: its vertices are $\{1, \dots, k\}$ and there is an arrow from i to j in $\mathcal{G}_\mathcal{M}$ if and only if there is an edge between r_i and c_i in $\mathcal{G}$. We show that $\mathcal{M}$ is an URM is equivalent to saying that $\mathcal{G}_\mathcal{M}$ is a DAG. As the $(1, 0, 1)$-adjacency matrix of $\mathcal{G}_\mathcal{M}$ is formed by extracting rows $r_1, \dots, r_k$ and columns $c_1, \dots, c_k$ of the occurrence matrix of $\mathcal{G}$, it is enough to prove the theorem.

Figure 5.2 illustrates this idea. We have a bipartite graph $\mathcal{G}$ and a matching $\mathcal{M}$ in red, next to it we represent the associated graph $\mathcal{G}_\mathcal{M}$, and the occurrence matrix A of $\mathcal{G}$ which also appears to be the $(1, 0, 1)$-adjacency matrix of $\mathcal{G}$.

First, we prove the following lemma:

Lemma 5.2. *There is a one-to-one correspondence between alternating paths of $\mathcal{G}$ with respect to $\mathcal{M}$ starting by a row vertex r_i, ending with a column vertex c_j, and in which all vertices are matched on the one hand, and (directed) paths of $\mathcal{G}_\mathcal{M}$ on the other hand.*

Proof of Lemma 5.2. Let $(v_1, v_2) \dots (v_{k-1}, v_k)$ be a directed path in $\mathcal{G}_\mathcal{M}$. Then $(r_{v_1}, c_{v_2})(c_{v_2}, r_{v_2}) \dots (r_{v_{k-1}}, c_{v_k})$ is an alternating path in $\mathcal{G}$: the edges (r_{v_i}, c_{v_i}) belong to the matching, and the edges $(r_{v_i}, c_{v_{i+1}})$ exist in $\mathcal{G}$ by definition of $\mathcal{G}_\mathcal{M}$.

Let $(r_{v_1}, c_{v_2}), (c_{v_2}, r_{v_2}) \dots (r_{v_{k-1}}, c_{v_k})$ be an alternating path in $\mathcal{G}$. We assume, without loss of generality, that $\{r_{v_i}, c_{v_i}\} \in \mathcal{M}$ for all i. The edges $(r_{v_i}, c_{v_{i+1}})$ connect two matched vertices in $\mathcal{G}$, but they do not belong to the matching. This implies that there are edges (v_i, v_{i+1}) in $\mathcal{G}_\mathcal{M}$. Thus $(v_1, v_2), \dots, (v_{k-1}, v_k)$ is a directed path in $\mathcal{G}_\mathcal{M}$. □

Example 5.1. In Figure 5.2, the directed path $(0, 1), (1, 3), (3, 5)$ in $\mathcal{G}_\mathcal{M}$ corresponds to the alternating path $(r_0, c_1)(c_1, r_1), (r_1, c_3), (c_3, r_3)(r_3, c_5)$ in $\mathcal{G}$.

This correspondence makes it easy to prove Theorem 5.1.

Proof of Theorem 5.1. If PAQ has an upper-triangular $k \times k$ principal submatrix, then we form the matching

$$\mathcal{M} = \{(P(i), Q(i)) \mid 1 \leq i \leq k\}.$$

The associated directed graph $G_\mathcal{M}$ is acyclic: its adjacency matrix is the $k \times k$ upper-triangular principal submatrix of PAQ. There is no alternating cycle in A: by the correspondence given in Lemma 5.2, this would imply a cycle in $G_\mathcal{M}$.

Conversely, let $\mathcal{M} = \{(r_1, c_1), \dots, (r_k, c_k)\}$ be a uniquely restricted matching on A. By the correspondence given in Lemma 5.2, this implies that the associated directed graph $G_\mathcal{M}$ is acyclic. Let M denote its adjacency matrix (it is a submatrix of A, as argued above). $G_\mathcal{M}$ can be topologically sorted: with a simple DFS, we find a permutation P of its vertices such that PMP is

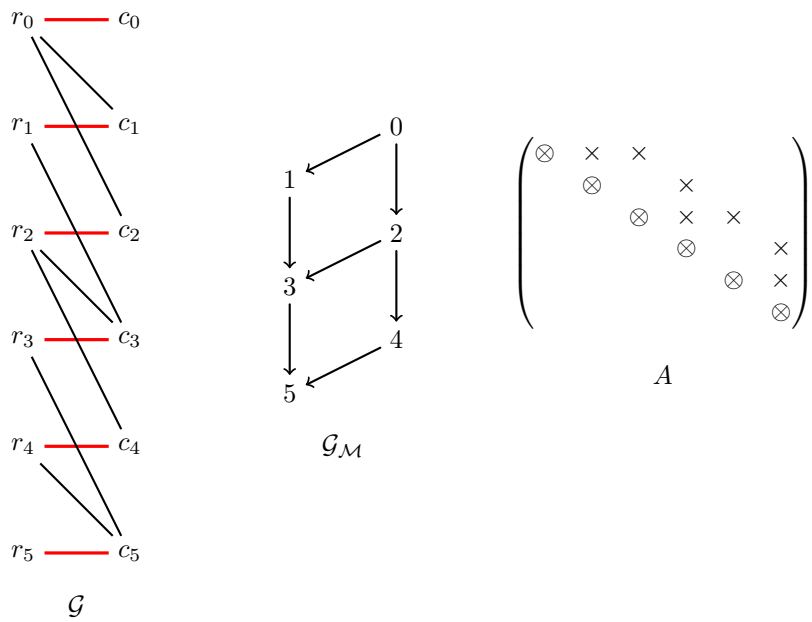


Figure 5.2 – A graph G with a matching M, the associated graph $\mathcal{G}_{\mathcal{M}}$ and the matrix A.

upper-triangular. This is enough for our purpose: we build row and column permutations P' and Q' such that the first k rows (resp. columns) are $r_{P(1)}, \ldots, r_{P(k)}$ (resp. $c_{P(1)}, \ldots, c_{P(k)}$). It follows that the principal $k \times k$ submatrix of $P'AQ'$ is PMP, which is upper-triangular. □

The maximum URM problem. To our knowledge the interest in maximum matching without alternating cycle first appears in [CC79]. However the notion of matching without (or with) alternating cycle had already been used before. Matchings with alternating cycle appear in [TIR78] to enumerate perfect and maximum matchings in bipartite graphs.

The problem of finding a maximum URM in a given graph was then formally introduced in [GHL01] under the name Maximum Uniquely Restricted Matching problem, following the work of [HS93]. The authors of [GHL01] showed that this is an NP-Complete optimisation problem for general and bipartite graphs. Some other results for special classes of graphs were given.

5.2 A Quasi-Linear Heuristic Algorithm for the Maximum URM Problem

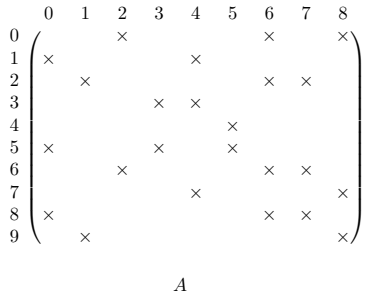
This algorithm is based on the works of [Bra02] and [Sch13] which are related to connectivity and ear decomposition of graphs.

The algorithm. Let $\mathcal{G} = (R, C, E)$ be a bipartite undirected graph. If $\mathcal{G}$ is an acyclic graph (i.e. $\mathcal{G}$ is either a tree or a forest as $\mathcal{G}$ is an undirected graph), then by definition, any matching of $\mathcal{G}$ will be an URM.

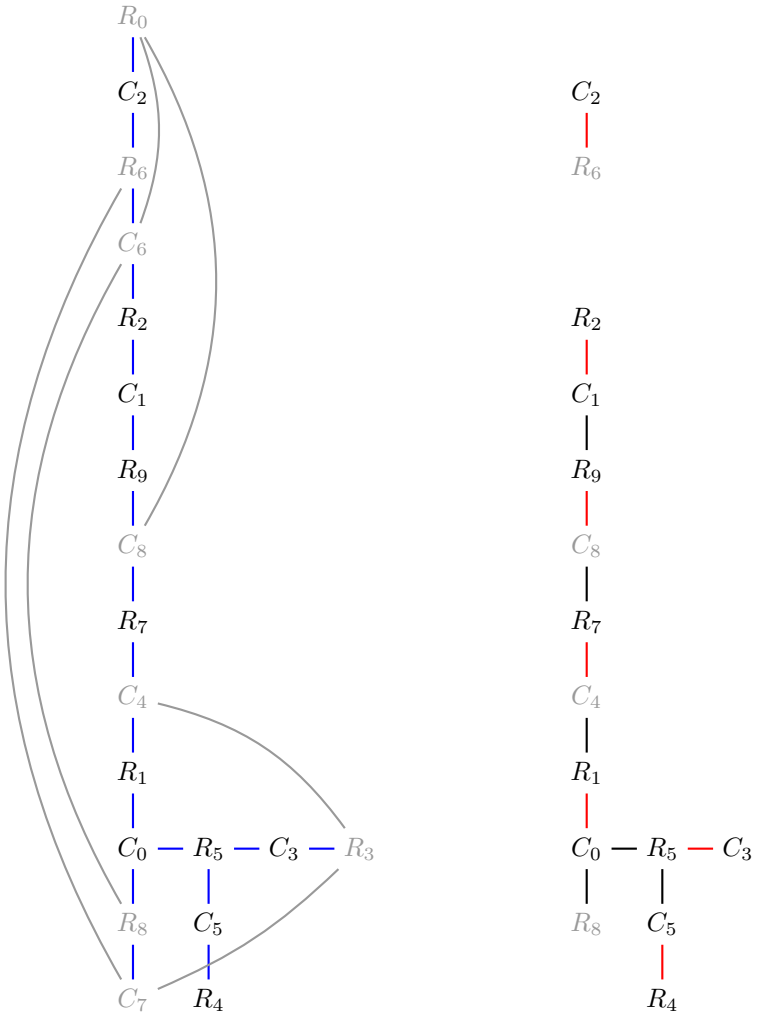
The idea is thus to compute a spanning tree $\mathcal{T}$ of $\mathcal{G}$, then remove all vertices from $\mathcal{T}$ that induce cycles in $\mathcal{G}$. We consider then the subgraph $\mathcal{T}'$ induced by the remaining vertices. Any matching of $\mathcal{T}'$ will be a URM of $\mathcal{G}$.

The difficult part is to detect what are the vertices that may cause cycles. Let us denote by V the set of vertices x such that for all y such that $(x, y) \in E$, (x, y) is an edge of $\mathcal{T}$. We could for instance consider the subgraph $\mathcal{T}'$ induced by the set V, but this would require to remove too many vertices, possibly all of them. Indeed V may even be empty.

We choose instead the following approach: Consider the graph $\mathcal{G}'$ formed by the edges not in $\mathcal{T}$, and let $\mathcal{I}$ be an independent set of $\mathcal{G}'$, and we consider the subgraph $\mathcal{T}'$ of $\mathcal{T}$ induced by $\mathcal{I} \cup V$. We claim that a matching of $\mathcal{T}'$ will be a URM of $\mathcal{G}$. Indeed, consider a cycle in $\mathcal{G}$, it cannot be contained inside $\mathcal{T}$, because $\mathcal{T}$ is a tree. Therefore, there is an edge $x \leftrightarrow y$ of the cycle in $\mathcal{G}'$.



(a) Input matrix.



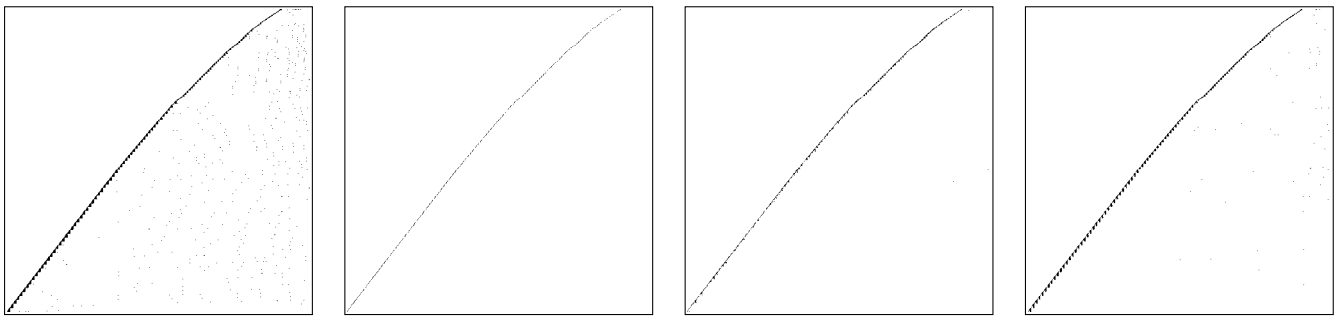
(b) Spanning tree and URM.

Figure 5.3 – Illustration of Algorithm 8.

Because $\mathcal{I}$ is an independent set of A', it cannot contain both x and y. This implies that x and y cannot both be matched in $\mathcal{M}$. This shows that the cycle cannot be alternating.

The whole procedure is summarised in Algorithm 8. We also show an example in Figure 5.3. Figure 5.3a represents the input matrix. In Figure 5.3b, we represent the associated bi-partite graph $\mathcal{G}$ on the left. The part in blue is a spanning tree of $\mathcal{G}$. The grey edges are the edges of the graph $\mathcal{G}'$. The grey vertices are the vertices of $\mathcal{G}'$. On the right, we represent the subtree $\mathcal{T}'$. The grey nodes are the nodes that belong the independent set $\mathcal{I}$ of $\mathcal{G}'$. The red edges are the edges of the matching.

Complexity analysis. Computing the spanning tree $\mathcal{T}$ of $\mathcal{G}$ and the graph $\mathcal{G}'$ and V can be done in time $\mathcal{O}(|R| + |C| + |E|)$, using a DFS. Once $\mathcal{I}$ is determined, computing $\mathcal{T}'$ is also an easy task. Computing a maximum matching on a tree can also be done in time $\mathcal{O}(|E|)$ recursively.



Original matrix
rank 168310

Faugère-Lachartre
65536 pivots

Tree-based
73492 pivots

Greedy
160740 pivots

Figure 5.4 – The `Homology/shar_te.b3` matrix, and the structural pivots found by each algorithm.

The difficulty lies in the computation of the independent set $\mathcal{I}$. We used the method described in Section 2.2.4. This gives us a total running time of $\mathcal{O}(n \log \mathrm{nz}(A) + \mathrm{nz}(A))$.

Remark 1. We could instead have used the Hopcroft-Karp algorithm [HK73] for minimum vertex cover algorithm, to find a *maximum* independent set, instead of a "large" one. However, the time complexity required would have been $\mathcal{O}(n + \mathrm{nz}(A)(\sqrt{n+m}))$. We chose to favour speed rather than optimised results.

Algorithm 8 URM computation using spanning trees.

Require: $\mathcal{G} = (R, C, E)$.
Ensure: $\mathcal{M}$: a URM matching of $\mathcal{G}$.
1: Compute $\mathcal{T} = (R, C, E_{\mathcal{T}})$: a spanning tree of $\mathcal{G}$.
2: Let $E' \leftarrow E \setminus E_{\mathcal{T}}$
3: Let $\mathcal{G}' = (R', C', E')$ be the subset of $\mathcal{G}$ edge-induced by E'.
4: Let $V \leftarrow (R \setminus R') \cup (C \setminus C')$.
5: Compute $\mathcal{I}$: an independent set of $\mathcal{G}'$.
6: Compute $\mathcal{T}'$: the subgraph of $\mathcal{T}$ induced by $V \cup \mathcal{I}$.
7: Compute $\mathcal{M}$: a maximum matching on $\mathcal{T}'$.

Discussion. There are cases where this strategy outperforms the Faugère-Lachartre heuristic (see Figure 5.4). However, in most cases it yields disappointingly bad results. We believe that it might only work well on very sparse matrices where $\mathcal{T}$ contains a non-negligible fraction of all non-zero entries.

5.3 A Simple Greedy Algorithm

The algorithm we are now going to present is the major contribution of [BDV17].

5.3.1 Description of the Procedure

Let us start by giving an intuitive idea of this algorithm. We start with an (empty) UR matching $\mathcal{P}$ and make it grow in a greedy way: we examine each candidate edge (r, c), where r and c are unmatched rows/columns; we check whether adding it to the matching would create an alternating cycle; if not, we add it to the matching.

Adding (r, c) to the matching creates an alternating cycle if and only if there is already an alternating path $(r, c_1), (c_1, r_1) \dots (r_k, c)$ in A with respect to $\mathcal{P}$. This means that c is reachable from r, via an alternating path. This makes it possible to check all candidate non-zero entries on the row r by performing a graph search.

If c is reached during the process, then (r, c) is a bad candidate and cannot be selected as pivot. On the other hand, if at the end of the search there still is an unvisited column c, then it means

that (r, c) is a valid candidate and can be selected as pivot. The detail of this procedure is given in Algorithm 9.

Algorithm 9 URM computation by detecting cycles.

Require: $\mathcal{G} = (R, C, E)$, an (empty) URM matching $\mathcal{M}$.
Ensure: $\mathcal{M}$: a larger URM matching of $\mathcal{G}$.

```
1:  for all unmatched i do
2:      Set Q_i ← ⊥                                              ▷ Initialisation
3:      for all j such that a_ij ≠ 0 do
4:          if j is matched then
5:              Enqueue j in Q_i
6:          else
7:              Mark j as "candidate"
8:      while Q_i ≠ ⊥ do                                         ▷ BFS
9:          Dequeue the first element j of Q_i
10:         if j not matched then
11:             continue
12:         Let k be the row matched to j
13:         for all a_kℓ such that ℓ is not marked "visited" do
14:             if ℓ is marked as "candidate" then
15:                 Remove the mark on ℓ
16:             mark ℓ as visited
17:             Enqueue ℓ in Q_i
18:     for all j such that a_ij ≠ 0 do                          ▷ Detection
19:         if j is marked as "candidate" then
20:             Set M ← M ∪ {(i, j)}
21:             break
22:     for all j such that a_ij ≠ 0 and j marked as "candidate" do   ▷ Clean up
23:         Remove the mark on j
```

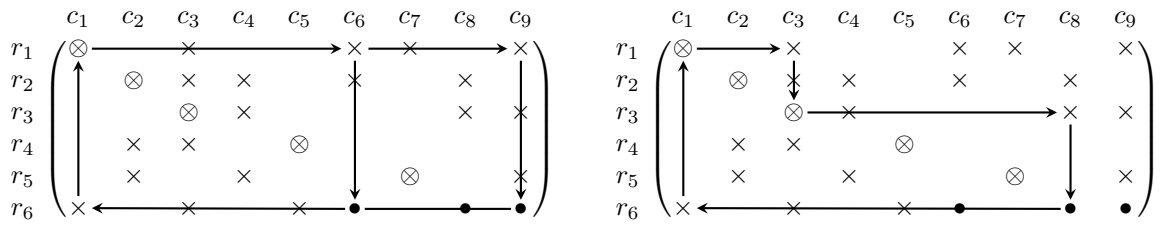
Remark 1. Notice that, if at the end of the search, there are several valid candidates, our algorithm chooses the first to come. There may be a more clever way to proceed, in order to preserve sparsity. We could have for instance chosen the sparsest column instead, or the column such that the number of non-zero elements on non-pivotal rows is the smallest. However, we have not tested if these heuristics will indeed lead to significantly better results.

Complexity analysis. The worst case complexity of the method is $\mathcal{O}(n \cdot \mathrm{nz}(A))$, which is the same complexity as the Wiedemann algorithm. However, we observed that, in practice, this is much faster. Our sequential implementation terminates in a few hours on our largest benchmark, versus many expected months for Wiedemann.

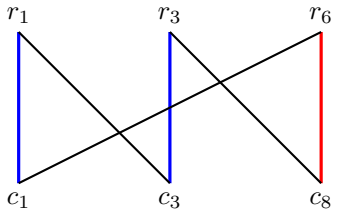
Remark 2. Most of the time is spent in the BFS step. We use the following early-abort strategy to speed it up. During the initialisation step, we count the number of candidates. During the BFS, each time we unmark a candidate, we decrement this counter. If it reaches zero, we jump to the clean-up step. On our collection of benchmarks, this gives a noticeable improvement.

Mixing the approaches. Note that this procedure can be started with a non-empty matching. In fact, one can first find a set of structural pivots using another technique (*e.g.* the F.-L. algorithm), and then enlarge the matching using this greedy algorithm. As a matter of fact, we figured out that the following strategy yields good results in many cases:

1. Find the F.-L. pivots, and add them to the matching.
2. For each non-pivotal column, if the upmost non-zero coefficient is on a non-pivotal row, add it to the matching.



(a) All three candidates are "bad".



(b) Alternating cycle with (r_6, c_8).

Figure 5.5 – Example of "bad" candidates.

3. Complete the matching using the greedy procedure above.

Case (2) is a particular case of the general algorithm given above. In the matrix from Example 3.1, the first three pivots are found performing step 1 of the previous strategy. The fourth is selected performing step 2. The last one is selected performing step 3. We represent the candidates on the last row by black bullets. As it is shown in Figure 5.5a, none of them can be selected, as in all the cases, an alternative cycle would appear. For instance, Figure 5.5b, gives an illustration of the alternating cycle obtained choosing candidate (r_6, c_8).

This greedy algorithm is capable of finding 99.9% of all the pivots in some matrices, when the other strategies we tried were not as good.

Remark 3. In certain cases, it can be interesting to permute the columns of the matrix first, to increase the number of structural pivots found by the FL algorithm. For instance, in the case of the GL7d matrices from Dumas's collection, we figured out that flipping them horizontally (i.e. permuting the first column with the last column, the second first with the second last and so on) enables us to find up to 50% more structural pivots with the FL algorithm. This is the case on the GL7d12 matrix shown in Figure 5.6a

5.3.2 A Parallel Algorithm

Because structural pivots search dominates the running time of our rank computations, we chose to parallelise the greedy algorithm described above. This raised interesting problems.

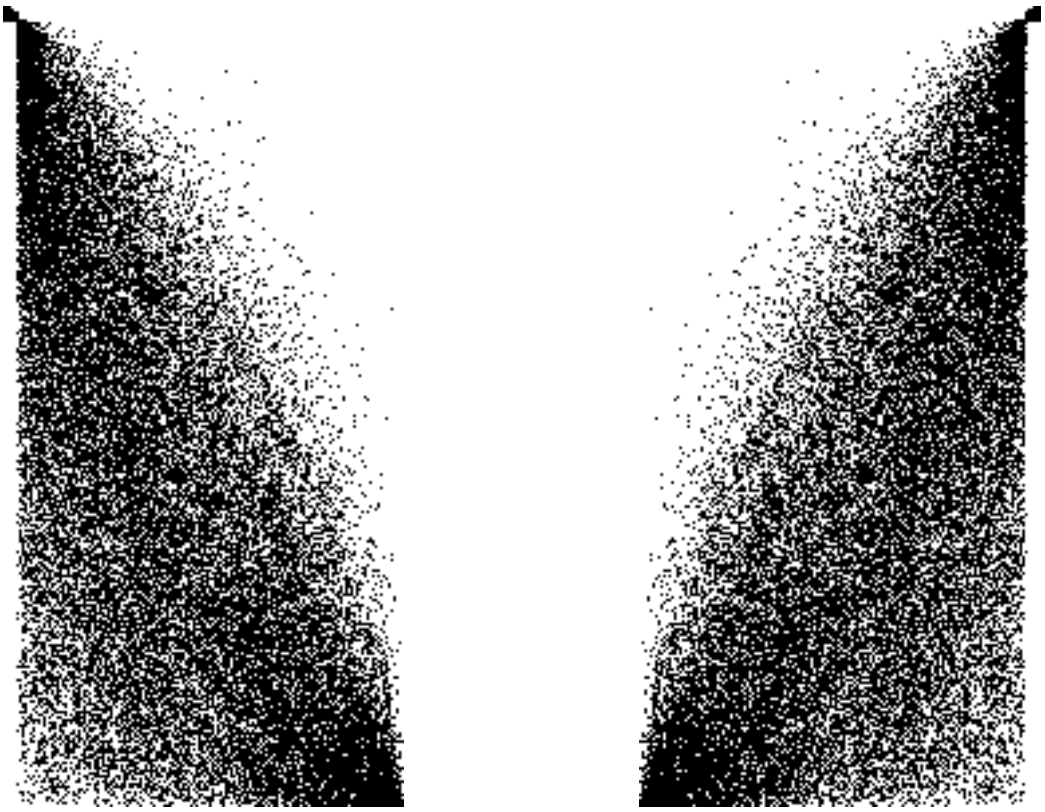
We denote by `qinv` the array of m integers, such that:

$$\texttt{qinv}[j] = \begin{cases} i & \text{if } (i,j) \text{ is in } \mathcal{M} \\ -1 & \text{if } j \text{ is not matched} \end{cases}$$

We also consider the array `mark` of m bytes, which for each rows i, contains the state of j while processing row i. For instance, we can have:

Parallelisation Strategy. The basic approach is to process k rows on k threads simultaneously. The description of the matching `qinv` is shared between threads, as well as `npiv`, an integer counting the size of the matching. In an ideal world, each thread would do its own job and update the matching independently.

The problem is that every time a thread adds an edge to the matching, new alternating paths are created in the graphs. Depending on the timing of operations, these paths may or may not be explored by concurrent BFS searches. More precisely, what may happen, is that two threads



(a) The original `GL7d12` matrix. (b) `GL7d12` after vertical swap.

Figure 5.6 – Swapping the `GL7d12` matrix to find more FL pivots.

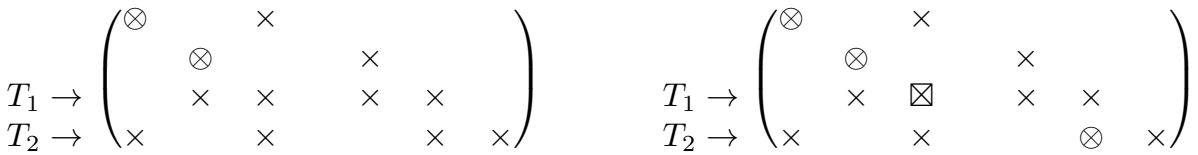


Figure 5.7 – Illustration of what can go wrong during parallelisation.

terminate their treatment more or less at the same time and choose pivots that together will create an alternating cycle. If you look at the example illustrated by Figure 5.7, two threads T_1 and T_2, respectively process rows 3 and 4. We assume that T_1 marks column 3 as candidate and starts its BFS. Shortly after T_2 terminates its treatment and adds $(4, 6)$ to $\mathcal{M}$. Unaware of that, T_1 chooses $(3, 3)$ as pivot. This produces an alternating cycle $(3, 3), (3, 6), (4, 6), (6, 3)$. In the rest of this section we are going to explain how we can bypass this problem.

Simple transactional approach. To avoid expensive synchronisations between threads, we used an optimistic approach based on transactions, as found in concurrent databases.

In its simplest form, it works by processing each row, using the algorithm of Section 5.3, in a transaction. Either the matching is not modified during the lifespan of the transaction, and it succeeds. If the matching has been modified, then the transaction rollbacks and has to be restarted. In most of the cases, examining a row does not reveal any new pivot, so that rollbacks are relatively rare. The redundant work that may occur is more than compensated by the nearly complete elimination of synchronisations. The procedure executed by each thread to process a row i is given by Algorithm 10.

The goal of this strategy is to minimise time spent in the critical section. In practice, we observed that time spent waiting to enter the critical section is negligible.

Algorithm 10 Simple transactional approach.

Require: A matrix A, a row i, the array `qinv` and `npiv`.
Ensure: An update of `qinv` and `npiv`.

1: Copy `npiv` in a local variable `npiv'`. ▷ Start transaction
2: Process row i ▷ Sequential process
3: **if** $\exists$ j such that a_{ij} is still a valid candidate **then** ▷ Commit
4: Set `bad` $\leftarrow$ `False`
5: Enter critical section
6: **if** `npiv` $>$ `npiv'` **then**
7: Set `bad` $\leftarrow$ `True`
8: **else**
9: Set `npiv` $\leftarrow$ `npiv` $+ 1$
10: Set `qinv`$[j] \leftarrow i$
11: Exit critical section
12: **if** `bad` $=$ `True` **then** ▷ Rollback
13: **go to** step 1.

Refined transactional approach. It is possible to refine this approach, to eliminate potential redundant work created by rollback. To this end, we add another shared array of m integers, `journal`, such that `journal`$[k]$ is the column of the k-th edge added to the matching.

When a thread tries to commit a new pivot a_{ij} and fails, then it has "missed" pivots on columns:

`journal[npiv']`, ..., `journal[npiv-1]`.

The corresponding rows are given by the array `qinv`. Then, during the rollback, you don't have to restart all the BFS, but only to check if these new pivots have not created cycles. The point is that if one of this new pivots is on a column j_{piv} such that $a_{ij_{piv}}$ is zero and j_{piv} has not been visited during the BFS, it can safely be ignored. Otherwise, the process has to be re-started.

It can be done using the procedure given by Algorithm 11. This eliminates almost all the extra work caused by rollbacks, while keeping the advantages.

Lock-free implemenation. Pushing things to the extreme, it is possible to implement both transactional strategies without lock nor critical section, if a compare-and-swap (CAS) hardware instruction is available (the CMPXCHG instruction is available on x86 processors since the 1990's). The C11 standard specifies that this functionnality is offered by the function:

`atomic_compare_exchange_strong`

of `<stdatomic.h>`. The `CAS(A,B,C)` instruction performs the following operation atomically: if $A = B$, then set $A \leftarrow C$ and return `True`, else return `False`.

If the journal is a linked list, then it can be updated atomically using a CAS. Each cell contains the column index of a pivot and a pointer to the next cell, `journal` is then a pointer to the first cell (or `NULL` initially). The simple strategy is then implemented as described in Algorithm 12.

When a transaction rollbacks, the "missed" pivots are those found by walking the `journal` list until the next link points to `journal'`. This allows to implement the "refined" strategy easily. It is necessary to perform the (potentially redundant) update of `qinv` in step 4: another thread could have been interrupted in step 3 after updating `journal` but before setting `qinv[j]`.

We did not implement this lock-free strategy, because the critical section did not appear to be a bottleneck. However, we have shown that it is possible to get rid of it.

Algorithm 11 Refined transactional approach.

Require: A matrix A, a row i, the arrays `qinv` and `journal` and `npiv`.
Ensure: An update of `qinv` and `npiv`.

1: Initialise Q_i with pivotal columns.
2: Copy `npiv` in a local variable `npiv'`. ▷ Start transaction
3: Process row i with the BFS ▷ Sequential process
4: **if** $\exists\ j$ such that a_{ij} is still a valid candidate **then** ▷ Commit
5: Set `bad` $\leftarrow$ `False`
6: Enter critical section
7: **if** `npiv` $>$ `npiv'` **then**
8: Set `bad` $\leftarrow$ `True`
9: **else**
10: Set `journal[npiv]` $\leftarrow j$
11: Set `npiv` $\leftarrow$ `npiv` $+ 1$
12: Set `qinv`$[j] \leftarrow i$
13: Exit critical section
14: **if** `bad` $=$ `True` **then** ▷ Rollback
15: **for** `npiv'` $\leq$ `k` $<$ `npiv` $- 1$ **do**
16: $j \leftarrow$ `journal[k]`
17: **if** $a_{ij} \neq 0$ **then**
18: Enqueue j
19: **go to** step 2
20: Remove the mark on indexed still marked as candidate ▷ Clean-up

Algorithm 12 Lock-free approach.

Require: A matrix A, a row i, the arrays `qinv` and `journal` and `npiv`.
Ensure: An update of `qinv` and `npiv`.

1: Initialise Q_i with pivotal columns.
2: Copy `journal` in a local variable `journal'`.
3: Allocate empty linked-list `new_journal` ▷ Start
4: Process row i with the BFS ▷ Sequential process
5: **if** $\exists\ j$ such that a_{ij} is still a valid candidate **then** ▷ Commit
6: Allocate new entry $(j,$ `journal'`$)$ to `new_journal`.
7: Set `good` $\leftarrow$ `CAS(journal, journal', new_journal)`.
8: **if** `good` $=$ `True` **then** ▷ `journal` $=$ `journal'`: no new pivots
9: Set `qinv`$[j] \leftarrow i$
10: **else** ▷ Rollback
11: Free `new_journal`
12: **go to** step 2
13: Remove the mark on indexed still marked as candidate ▷ Clean-up

Table 5.1 – Benchmark matrices.

GL7/GL7dk	$\|A\|$	n	m	rank
15	6,080,381	460,261	171,375	132,043
16	14,488,881	955,128	460,261	328,218
17	25,978,098	1,548,650	955,128	626,910
18	35,590,540	1,955,309	1,548,650	921,740
19	37,322,725	1,911,130	1,955,309	1,033,568
20	29,893,084	1,437,547	1,911,130	877,562
21	18,174,775	822,922	1,437,547	559,985
22	8,251,000	349,443	822,922	262,937

Table 5.2 – Benchmark Machines Specifications.

Name	CPU Type	# CPU	cores/CPU	L3 Cache/CPU
A	Xeon E5-2670 v3	2×	12	30MB
B	Xeon E5-4620	4×	8	16MB
C	Xeon E5-2695 v4	2×	18	45MB

5.4 Experiments, Results and Discussion

The algorithms described here have been implemented in C (using OpenMP) and assembled in SpaSM. Our code computing the rank modulo a word-size prime p (in our tests $p = 42013$) in parallel can be reduced to a single C file of 1200 lines of code (including IO).

We used the matrices from algebraic K-theory [DEGU07] as benchmarks (GL7/GL7dk, where k ranges between 10 and 26). They are amongst the largest in Dumas's sparse matrix collection [Dum12], and computing their rank is challenging. It was only feasible using iterative methods to this point — and it required many days. Table 5.1 gives standard information about them.

We benchmarked our implementation on three parallel machines, denoted by A, B, and C with different characteristics, summarised in table 5.2. A belongs to the university of Lille-1's cluster; B was kindly made available to us by the ANR HPAC project and is located in the university Grenoble-Alpes; C is used for teaching HPC at the Paris-6 university. They all contain Intel Xeon E5-xxxx CPUs with varying number of cores and amount of L3 cache. We focused performance measurement on the structural pivot selection phase, because its running time is dominating in practice, and because the way we compute the rank of the dense Schur Complement is rather naive.

Table 5.3 shows the number of structural pivots found using the FL heuristic and the greedy algorithm of section 5.3, as well as the proportion of the rank that this represent (*i.e.* $100 \times$ #pivots/rank). In order to maximise the number of pivots found by the FL heuristic, we first flipped the matrices horizontally (we swap the first and the last column, the second first and the second last,...). For the last matrices (GL7dk, with $k \geq 20$), we also transposed them as we figured out that this yields better results.

Table 5.4 gives the performances of the greedy algorithm. For each machine A, B, and C, the first column gives the running time (in seconds) of the algorithm using only one core, the second column gives the running time of the algorithm using all available cores. Finally the last column gives the speedup obtained using parallelisation (running time using one core / running time using all cores).

We discuss these results for two matrices: the easiest case (GL7d15) and the hardest case (GL7d19). We compare their timing with Wiedemann Algorithm.

5.4.1 Comparison with Wiedemann Algorithm

According to [DEGU07], for the considered matrices, the sequential running time of the sequential Wiedemann algorithm is estimated between 67 hours (GL7d15) and 322 days (GL7d19, the hardest

Table 5.3 – Number of structural pivots found using the FL heuristic and the greedy algorithm.

GL7dk	# FL pivots	% of the rank	# greedy pivots	% of the rank
15	128,452	97.28	132,002	99.97
16	317,348	96.69	328,079	99.96
17	602,436	96.10	626,555	99.94
18	879,241	95.39	920,958	99.92
19	980,174	94.83	1,032,419	99.89
20	817,944	93.21	877,317	99.97
21	522,534	93.31	559,784	99.96
22	246250	93.65	262817	99.95

Table 5.4 – Parallel efficiency of the greedy algorithm.

	Machine A			Machine B			Machine C		
GL7dk	1 core (s)	24 cores (s)	speedup	1 core (s)	32 cores (s)	speedup	1 core (s)	36 cores (s)	speedup
15	18	1	15.2	26	1	26.0	22	1	26.9
16	148	7	20.0	301	13	23.9	180	5	33.4
17	687	36	18.9	1502	105	14.4	897	29	30.9
18	2206	123	17.9	5649	388	14.6	2594	94	27.7
19	3774	222	17.0	9284	692	13.4	4212	197	21.4
20	1272	72	17.8	3657	197	18.5	1483	56	26.3
21	359	18	19.5	886	36	24.5	446	14	31.7
22	49	3	18.8	68	3	26.2	57	2	30.1

case). A parallel implementation of the Block-Wiedemann algorithm on a SGI Altix 370 with 64 Itanium2 processors allowed the authors of [DEGU07] to compute the rank of `GL7d15` in 2.25 hours using 30 processors and to compute the rank of `GL7d19` in 35 days using 50 processors in parallel.

We tried to check how these results held the test of time by running the sequential computation using LinBox (sequential Wiedemann algorithm) on a single core of machine A. The rank of `GL7d15` was found in 3 hours. We could not compute the rank of `GL7d19`: unfortunately, the cluster manager killed our job after only 16 days.

In strong contrast to these numbers, on a single core of A our code computed the rank of `GL7d15` in 18 seconds, and the rank of `GL7d19` in 1 hour (wall-clock time).

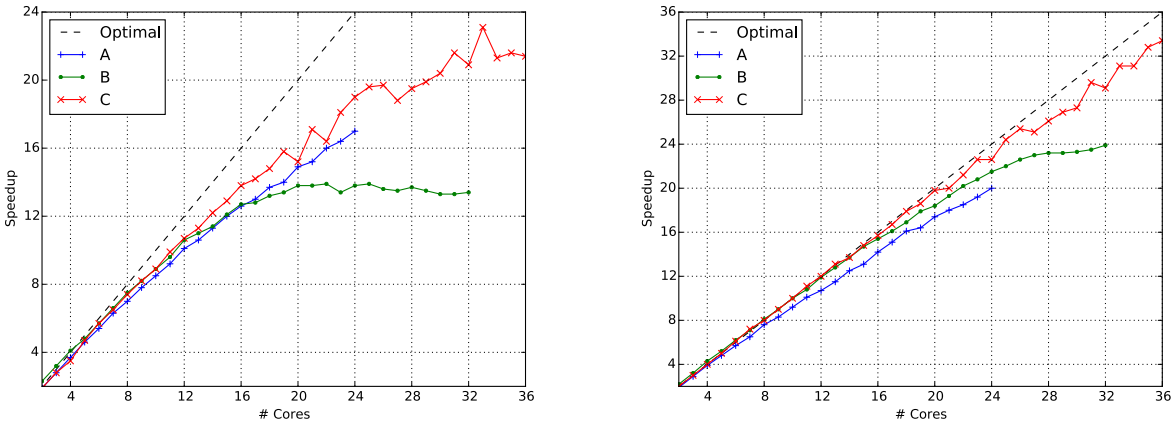
Furthermore, the parallel implementation of our algorithm computed the rank of these matrices in 0.8 seconds for `GL7d15` and `GL7d19` in 4 minutes (wall-clock time), using 36 cores of machine C.

Remark 1. To be totally fair, we should compare the parallel implementation of our algorithm with a parallel implementation of the Block-Wiedemann Algorithm on the same machine, using the same number core. However, proper public implementation of the Block-Wiedemann Algorithm on a field that is not $\mathbb{F}_2$ are hard to come by. The only one we know of is the one implemented in CADO-NFS, which is tuned for matrices arising from factorisation and discrete logarithm. The optimisation made in that way makes this code hardly customisable to our needs. However, considering that our sequential implementation is much faster that the sequential Wiedemann algorithm on these matrices, and that our parallel implementation does not scale too bad depending on the matrix and the machine (see discussion below), we have every reason to hope that for these matrices, our algorithm is also faster than the Block-Wiedemann Algorithm.

5.4.2 Scalability

The most interesting case is the largest matrix, `GL7d19`, whose rank computation is the toughest. It is the matrix where our parallel implementation scales worst (Figure 5.8a). With the smaller `GL7d16`, on the other hand, we observe near-perfect scalability (Figure 5.8b).

Figures 5.8a and 5.8b show that, given one matrix, our parallel implementation does not scale the same way on every benchmark machine. We noticed that our implementation does not scale very well on B, reaching a parallel efficiency (speedup/ #cores) of 63.01% in average using 32 cores. In fact, for the worst case (`GL7d19`) the parallel efficiency is 41.91%. On the other hand, the



(a) Structural pivot search, `GL7d19`.

(b) Structural pivot search, `GL7d16`.

Figure 5.8 – Scalability of the Greedy Algorithm.

implementation scales better on C, where we reach a parallel efficiency of 79.28% in average using 36 cores, and which can be up to 92.81% in the best case (`GL7d16`).

This phenomenon may be due to the size of the L3 Cache. The `qinv` array describing the matching is accessed frequently, in an unpredictable manner, by all the threads. Each thread also accesses frequently and unpredictably its own `mark` array defined as in equation 5.1 (the `queue` of each thread, on the other hand, is accessed with good spatial and temporal locality).

$$\texttt{mark}[j] = \begin{cases} 1 & \text{if } j \text{ is already matched to another row} \\ 0 & \text{if } (i,j) \text{ is a valid candidate} \\ -1 & \text{if } (i,j) \text{ is a bad candidate} \end{cases} \tag{5.1}$$

We thus expect the L3 cache of each CPU to contain a copy of `qinv` plus one `mark` per thread. When k threads run on each CPU, this amount of data requires $(4+k)m$ bytes.

This suggests that a strong performance degradation is likely to occur when this quantity goes beyond the size of the L3 cache. On machine B, for the `GL7d19` matrix, this should happen when more than 4 threads run on each CPU. And indeed, we observe a severe drop in parallel efficiency when using more than 16 threads on the 4 CPUs.

This might also explain why transposing the last matrices helps: they have much more columns than rows.

Lastly, this bottleneck could potentially be alleviated a little by using more efficient `marks`, using only 2 bits per column instead of 8. However, doing so would require more bit-twiddling, and its effect on the overall performance has yet to be evaluated.

Conclusion

We have designed an efficient algorithm for sparse gaussian elimination over finite fields. We mostly focus on the computation of the rank, because it raises the same issues as solving linear system, (i.e. computing a PLUQ factorisation of a matrix) while returning only a single integer. This algorithm basically consists of three steps: selecting a set of structural pivots before performing any arithmetical operations, computing the Schur Complement of the matrix with respect to these pivots in one fell swoop, and computing the rank of this Schur Complement by any means, for instance recursively.

One issue we may encounter is when the Schur Complement is big and totally dense. Generally in this case, there is nothing we can do, and we will have to abort and report memory failure. Some favorable cases happen when the number of structural pivots found beforehand is close to the rank. Then we can utilise tricks that will return the rank of the Schur Complement, without having to compute it.

We implemented it and benchmarked it using matrices from the Sparse Integer Matrix Collection of J.-G. Dumas. Using a simple pivot selection heuristic due to Faugère and Lachartre, our algorithm already outperformed both the classical right-looking algorithm, and the GPLU algorithm. This algorithm could also process more matrices than these two previous algorithms, without suffering from memory issue, and was also more efficient than the iterative methods for these large matrices when rank computation takes days with the Wiedemann Algorithm.

We eventually discuss of the importance of finding a large set of structural pivots before starting any elimination step. However, finding the largest set of such pivots is an NP-complete optimisation problem. We have to be content with an hopefully large set of them. At this aim, we proposed two new pivots selection heuristics, and among them, one which appears to be very efficient.

This new method allows us to compute the rank of bigger matrices than before, in a tiny fraction of the time that was needed before, using iterative methods. Furthermore, this method is parallelisable, and scales rather well for most of the matrices, depending on the RAM of the machine.

We have to highlight the limitations of our algorithm. It will not always work efficiently and will even fail badly on matrices where the fill-in is unavoidable. Our experience suggests that a large number of structural pivots can often be found in matrices that are somewhat triangular, such as the GL7dk matrices. In this case, the algorithm we present is more likely to be able to exploit this structure to allow much faster operations than the "slow-but-sure" iterative methods.

Finally, we can wonder if our method can be applied as a sub-routine during the linear algebra step of factorisation and discrete logarithms problems.

PART

2

3XOR Problem

Introduction

"There are 364 days when you can get unbirthday presents, and only one for birthday presents, you know."

– Lewis Carroll –

The 3SUM Problem is a well-known problem in computer science and many geometric problems have been reduced to it. We study the 3XOR variant which is more cryptologically relevant. In this problem, the attacker is given black-box access to three random functions F, G and H and she has to find tree inputs x, y and z such that $F(x) \oplus G(y) \oplus H(z) = 0$. In the case where only $2^{n/3}$ queries are made to each of the functions, the folklore quadratic algorithm, which consists in creating all $F(x) \oplus G(y)$ and checking if there is a z such that $H(z) = F(x) \oplus G(y)$, solves this problem in $\mathcal{O}\left(2^{2n/3}\right)$ in time and $\mathcal{O}\left(2^{n/3}\right)$ in space.

Generalised Birthday Problem and Cryptography

The 3XOR problem is a difficult case of the more general kXOR (or k-list) problem, also known as the generalised birthday problem. The birthday problem is a widely used cryptanalytical tool: given two lists $\mathcal{L}_1$ and $\mathcal{L}_2$ of bit-strings drawn uniformly at random from $\{0,1\}^n$, find $\mathbf{x}_1 \in \mathcal{L}_1$ and $\mathbf{x}_2 \in \mathcal{L}_2$ such that $\mathbf{x}_1 \oplus \mathbf{x}_2 = 0$. It is widely known that a solution exists with high probability as soon as $|\mathcal{L}_1| \oplus |\mathcal{L}_2| \gg 2^n$ holds. This solution can be found in $\mathcal{O}\left(2^{n/2}\right)$ time and space by simple algorithms: if the two lists are sorted, scanning one of them is a possibility. The generalised birthday problem (GBP) is defined as follows: given k lists of uniformly random n-bit vectors, find an element in each one of the lists such that the XOR of all these elements is equal to a target value (often assumed to be zero). This problem, in this general definition, has been introduced by Wagner [Wag02]. In the same paper, Wagner also proposed an algorithm which solves the GBP for all values of k, but works best when k is a power of two, but is rather disappointing in the case $k = 3$, in the sense that it could not be solved faster than the classical 2-list birthday problem. Wagner kXOR algorithm works by querying the functions more than strictly required from an information-theoretic point of view. This gives some leeway to target a solution of a specific form, at the expense of processing a huge amount of data. This method was later refined by Minder and Sinclair [MS12] in order to find a solution even when the lists are smaller than required by Wagner setting.

Before Wagner, the GBP problem had been studied only for a specific number of lists, usually $k = 4$. Schroeppel and Shamir [SS81] proposed an algorithm which finds all solutions to the 4XOR problem, with low memory consumption. Blum, Kalai and Wasserman [BKW03] used a similar idea while they designed their notorious BKW algorithm to solve the learning parity with noise (LPN) problem.

The 3XOR Problem in Cryptography

The 3XOR problem has received less attention in comparison. In [Nan15], Nandi exhibited a forgery attack against the COPA mode of operation for authenticated encryption requiring only

$2^{n/3}$ encryption queries and $\mathcal{O}\left(2^{2n/3}\right)$ operations. This attack works by reducing the problem of forging a valid ciphertext to the one of solving an instance of the 3XOR problem. This attack was later improved by Nikolić and Sasaki [NS15] to $\mathcal{O}\left(2^{n/2-\epsilon}\right)$ operations at the cost of making a little less than $2^{n/2}$ queries to the COPA oracle. At this end, Nikolić and Sasaki designed an improved 3XOR algorithm, and they obtained a speedup of about $\sqrt{n/\ln(n)}$ compared to the classical birthday algorithm. However, five years before Nikolić and Sasaki, Joux [Jou09, § 8.3.3.1] had already proposed a better algorithm with which he obtained a speedup of about $\sqrt{n}$ compared to Wagner.

Time and Memory Tradeoffs

The bottleneck of the kXOR (and more precisely the 3XOR) algorithms is that they usually require the manipulation of a huge amount of memory which quickly makes them impractical. Time-memory tradeoffs were proposed by Bernstein [Ber07] and improved later by Bernstein et al. [BLN+09] for the general kXOR problem. We found one of this tradeoff, called the "clamping trick", particularly interesting in the context of 3XOR. The idea is the following: when more queries than actually needed are performed, one may choose to keep only the entries that are zeroes on the first ℓ bits, with ℓ chosen so that the product of the sizes of the three lists would be exactly $2^{n-\ell}$. This reduces the problem to the one of finding a 3XOR over $n-\ell$ bits, in the case where there will be only one solution to the problem with high probability.

Other time-memory tradeoffs have been proposed by Nikolić and Sasaki [NS15], in the case of the kXOR when k is a power two.

Our Contributions

In [BDF18], we have presented a new 3XOR algorithm that generalises Joux's method. Our algorithm can be applied to any size of input lists, and may be utilised to find all solutions to the 3XOR problem (and not only one). When the three input lists have the same size $2^{n/3}$, it has a time complexity of $\mathcal{O}\left(2^{2n/3}/n\right)$ and $\mathcal{O}\left(2^{n/3}\right)$ in space, and thus gains a $\times n$ speedup compared to the quadratic algorithm in this case. This algorithm is practical: it is up to $3\times$ faster than the quadratic algorithm. Furthermore, using Bernstein's "clamping trick", we show that it is possible to adapt this algorithm to any number of queries, so that it will always be better than the Nikolić-Sasaki Algorithm in the same settings. To gain a deeper understanding of these problems, we propose to solve actual 3XOR instances for the SHA256 hash function. We present practical remarks, along with a 96-bit 3XOR of SHA256. We also revisit a 3SUM algorithm by Baran-Demaine-Pătraşcu [BDP05] which is asymptotically $n^2/\log^2 n$ times faster than the quadratic algorithm when adapted to our 3XOR problem, but is otherwise completely impractical.

Organisation of the Part

In Chapter 6 we give a formal definition of the problem, recall its context along with a few applications. We also present the computational model, as well as algorithmic and probabilistic tools that we will utilise throughout the part.

In Chapter 7 we revisit previous work such as Wagner 4XOR algorithm. We explain why this method does not work well for the case of the 3XOR. We present later improvements by Joux and Nikolic-Sasaki, and we detail Bernstein's clamping trick.

In Chapter 8 we present the main contribution of [BDF18]: our new algorithm for the 3XOR problem. We also present time-memory tradeoff using the clamping trick, and explain how large values of n should be handled in practice. We also present some possible improvements of our algorithm, that may allow us to gain constant factors, on the running time. Finally we present our experimentations, give some implementation details, and present a 3XOR over 96-bit of SHA256.

In Chapter 9 we present an adaptation of a 3SUM algorithm to our 3XOR case. This algorithm due to Baran Demain and Pătraşcu [BDP05], and usually called BDP, is asymptotically faster than any other. However, in our particular case, we claim that it is impractical, and we will explain why.

Chapter 6

Of the 3XOR Problem: Context and Preliminaries

In this chapter, we give some generalities about the 3XOR problem. We namely recall some important probability results related to this problem, and we also present the computational model that we will utilise. We recall some applications and related problems, and finally we present the algorithmic tools that we will need in the following chapters.

6.1 Generalities

We can define the 3XOR problem as follows:

Problem 6.1. Let $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ be three arbitrary large lists whose entries are uniformly random bit-strings of $\{0,1\}^n$. Find $(\mathbf{a},\mathbf{b},\mathbf{c}) \in \mathcal{A}\times\mathcal{B}\times\mathcal{C}$ such that $\mathbf{a}\oplus\mathbf{b}\oplus\mathbf{c}=0$.

6.1.1 Notations and Definitions

Usually, n will represent the *size of our problem*, meaning the length of the bit-strings we consider in our problem. From now on we will see an n-bit string as a vector of $\mathbb{F}_2^n$.

Given a list $\mathcal{L}$ we denote by $\mathcal{L}[i]$ the element of the list indexed by i, meaning the $(i+1)$-th element of $\mathcal{L}$ as the indexation starts at 0. Given a parameter i, We denote by $\mathcal{L}_i$, a list indexed by i. We denote by $\mathcal{L}[i..j]$ the sublist containing the elements $\mathcal{L}[i],\mathcal{L}[i+1],\dots\mathcal{L}[j-1]$, by $\mathcal{L}[i..]$ the sublist containing all elements of $\mathcal{L}$ starting from $\mathcal{L}[i]$, and by $\mathcal{L}[..i]$, the sublist containing all elements of $\mathcal{L}$ up to $\mathcal{L}[i-1]$ included. This is similar to the "slice" notation in Python.

For a given vector $\mathbf{x}$, $\mathbf{x}[i..j]$ denotes the sub-vector formed by taking the coordinates from i to $j-1$. We define $\mathbf{x}[i..]$ and $\mathbf{x}[..i]$ accordingly. For instance, $\mathcal{L}[i][..\ell]$ denotes the sub-vector formed by taking the first ℓ coordinates of the $(i+1)$-th element of $\mathcal{L}$. Given an n-bit vector $\mathbf{x}$ and a parameter $0<\ell<n$, we call ℓ-bit *prefix* of $\mathbf{x}$ the sub-vector $\mathbf{x}[..\ell]$. We denote by $\mathcal{L}^{[\mathbf{p}]}$ the sublist of $\mathcal{L}$ composed by all vectors whose first bits are $\mathbf{p}$ (the size of this prefix is usually clear, given the context). If $\mathbf{x}$ and $\mathbf{y}$ are two bit-strings, we denote by $(\mathbf{x}|\mathbf{y})$ the bit-string obtained, when concatenating $\mathbf{x}$ and $\mathbf{y}$.

Given a list $\mathcal{L}$ containing n-bit vectors and an n-bit vector $\mathbf{y}$ over $\mathbb{F}_2$, we denote by $\mathcal{L}\oplus\mathbf{y}$ the list $\mathcal{L}\oplus\mathbf{y}=\{\mathbf{x}\oplus\mathbf{y}|\mathbf{x}\in\mathcal{L}\}$.

Given a list $\mathcal{L}$ containing n-bit vectors and an n-by-n matrix M over $\mathbb{F}_2$, we denote by $\mathcal{L}M$ the list $\mathcal{L}M=\{\mathbf{x}M|\mathbf{x}\in\mathcal{L}\}$

Given two lists $\mathcal{A}$ and $\mathcal{B}$, and an integer $0<\ell\leq n$, we call *join* of $\mathcal{A}$ and $\mathcal{B}$ on the first ℓ bit, the sublist $\mathcal{A}\bowtie_\ell\mathcal{B}$ of $\mathcal{A}\times\mathcal{B}$ such that for all $(\mathbf{a},\mathbf{b})\in\mathcal{A}\bowtie_\ell\mathcal{B}$, $\mathbf{a}[..\ell]=\mathbf{b}[..\ell]$. More formally, we have:

$$\mathcal{A}\bowtie_\ell\mathcal{B}=\{(\mathbf{a},\mathbf{b})\in\mathcal{A}\times\mathcal{B}|(\mathbf{a}\oplus\mathbf{b})[..\ell]=0\}. \tag{6.1}$$

Following the same idea, we denote by $\mathcal{A} \bowtie_{\ell,\mathbf{p}} \mathcal{B}$ the sublist of $\mathcal{A} \times \mathcal{B}$ such that for all $(\mathbf{a}, \mathbf{b}) \in \mathcal{A} \bowtie_{\ell,p} \mathcal{B}$, $\mathbf{a}[..\ell] \oplus \mathbf{p} = \mathbf{b}[..\ell]$. Then again, we have:

$$\mathcal{A} \bowtie_{\ell,p} \mathcal{B} = \{(\mathbf{a}, \mathbf{b}) \in \mathcal{A} \times \mathcal{B} | (\mathbf{a} \oplus \mathbf{b})[..\ell] = \mathbf{p}\}. \tag{6.2}$$

Given a list $\mathcal{L}$, we call *size* of $\mathcal{L}$ the number of elements in $\mathcal{L}$. We denote this size by $N_{\mathcal{L}}$. Accordingly, $N_{\mathcal{A}}, N_{\mathcal{B}}$ and $N_{\mathcal{C}}$ respectively represent the size of the lists $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$. Let $\mathbf{p}$ be a given prefix, we denote by $N_{\mathcal{L}}^{\mathbf{p}}$ the size of the sublist $\mathcal{L}^{[\mathbf{p}]}$ of the elements of $\mathcal{L}$ that share the same prefix $\mathbf{p}$.

6.1.2 Probability Results

Balls into bins. We consider the following model: we have N balls that we dispatch into m bins. Each ball is put into a bin chosen uniformly at random and independently from the other balls, so that the probability that a given ball falls into a given bin is $1/m$. Here, the balls represent vectors, and the bins all possible values they can take.

In this setting, the probability that any two balls collide (i.e. are in the same bins) is:

$$\begin{aligned}
\mathbb{P}[\text{2 given balls in 1 bin}] &= \sum_{i=1}^{m} \mathbb{P}[(\text{ball 1 in bin } i) \cap (\text{ball 2 in bin } i)] \\
&= \sum_{i=1}^{m} \mathbb{P}[\text{ball 1 in bin } i]\mathbb{P}[\text{ball 2 in bin } i] \\
&= \sum_{i=1}^{m} \frac{1}{m^2} \;=\; \frac{1}{m}.
\end{aligned} \tag{6.3}$$

In terms of vectors, this could be translated as the probability that two $(\log m)$-bit vectors have the same value.

Now, if we throw N balls uniformly at random in m bins, the expected number N_b of balls in any given bin b is:

$$\begin{aligned}
\mathbb{E}[N_b] &= \sum_{i=1}^{N} \mathbb{P}[\text{Ball } i \text{ is in bin } b] \\
&= \sum_{i=1}^{N} \frac{1}{m} \\
&= \frac{N}{m}.
\end{aligned} \tag{6.4}$$

We have the following result:

Theorem 6.1 (Stated as in Shoup [Sho09])**.** *Suppose that N balls are thrown uniformly at random and independently into m bins. The probability that at least one bin contains more than one is bounded as follows:*

$$1 - e^{-\frac{N(N-1)}{2m}} \leq \mathbb{P}[\text{There is a bin with more than 1 ball}] \leq \frac{N(N-1)}{2m}$$

For instance, if $N = 2^{n/2}$ is $m = 2^n$, then the probability that there is a collision will be close to $1/2$.

Now, given a pack of m balls thrown uniformly at random into m bins, we may wonder how many balls are there in the bin that contains the most balls? In his thesis, Mitzenmacher [Mit96] actually answers this question.

Theorem 6.2 (Mitzenmacher [Mit96, Lemma 2.14])**.** *The maximum load is $\Theta(\ln(m)/\ln(\ln(m)))$, with high probability.*

In terms of vectors, this means that given a list of m uniformly random vectors of $\mathbb{F}_2^{\log m}$, we can expect to find $\Theta(\ln(m)/\ln(\ln(m)))$ that are identical.

Remark 1. Before Mitzenmacher, it was already a known result that the maximum load is $\mathcal{O}(\ln(m)/\ln(\ln(m)))$ [Gon81]. In [Mit96], Mitzenmacher proved that this bound is actually tight.

Birthday Paradox and generalisations. We assume now that we have two packs of balls. The first pack contains N_1 blue balls and the second pack contains N_2 red balls. The question we may ask is: what is the expected number of collisions between blue balls and red balls? In other words, we are asking the number of pairs of balls with one red ball and one blue ball, such that the two balls share the same bin.

To answer this question, we consider the variable X_{ij} that is 1 if the blue ball i and the red ball j collide and 0 otherwise. The number of balls that collide can be given by $X = \sum_i^{N_1} \sum_j^{N_2} X_{i,j}$. We have:

$$\begin{aligned}\mathbb{E}[X] &= \sum_i \sum_j \mathbb{E}[X_{ij}] \\ &= \sum_i \sum_j \mathbb{P}[\text{2 balls in 1 bin}] \;=\; \frac{N_1 N_2}{m}.\end{aligned} \tag{6.5}$$

From here, we can see that we dare to expect to have a collision between a blue and a red ball, we need to have $\mathbb{E}[X] = 1$, and then:

$$N_1 N_2 = m. \tag{6.6}$$

In terms of vectors, this means that if we have two lists $\mathcal{L}_1$ and $\mathcal{L}_2$ of $(\log m)$-bit vectors, and we hope to have a couple $(\mathbf{x}_1, \mathbf{x}_2) \in \mathcal{L}_1 \oplus \mathcal{L}_2$ such that $\mathbf{x}_1 = \mathbf{x}_2$, then the product of the size of $\mathcal{L}_1$ and $\mathcal{L}_2$ must be m. We also have the following result:

Theorem 6.3 (Stated as in Vaudenay [Vau05])**.** *If $N_1 = \alpha\sqrt{m}$ red balls and $N_2 = \beta\sqrt{m}$ blue balls are thrown independently and uniformly at random into m bins, then the probability that there is no bin that simultaneously contains red and blue balls, satisfies*

$$\mathbb{P}[\text{No bin contains a red ball and a blue ball}] \xrightarrow[m\to+\infty]{} e^{-\alpha\beta}$$

Wagner's Generalised Birthday Problem, is slightly harder to represent with balls and bins. Still, we give a similar result for the case of the 3XOR. We assume that we have two packs of balls the first pack contains red balls that are each indexed by one integer $0 \leq i < N_{\mathcal{A}}$, different for each ball, the second pack contains blue balls that are indexed by two integers $0 \leq j < N_{\mathcal{B}}$ and $0 \leq k < N_{\mathcal{C}}$ such that a couple (j, k) will uniquely represent a ball. We throw uniformly at random the balls of these two packs into m bins. The pack of red balls represents one of the input list, say $\mathcal{A}$ that contains $N_{\mathcal{A}}$ elements, and the pack of blue balls represents the list $\mathcal{B} \oplus \mathcal{C}$ of the xor of the elements of the two other lists, such that $\mathcal{B}$ contains $N_{\mathcal{B}}$ elements and $\mathcal{C}$ contains $N_{\mathcal{C}}$ elements. For instance, the ball indexed by (j, k) stands for the element $\mathcal{B}[j] \oplus \mathcal{C}[k]$. It is clear that there are overall $N_{\mathcal{B}} N_{\mathcal{C}}$ blue balls. Then according to Equation 6.5, the expected number of collisions between a blue and a red ball will be:

$$\mathbb{E}[N_{col}] = \frac{N_{\mathcal{A}} N_{\mathcal{B}} N_{\mathcal{C}}}{m}, \tag{6.7}$$

and then if we have:

$$N_{\mathcal{A}} N_{\mathcal{B}} N_{\mathcal{C}} = m, \tag{6.8}$$

we can expect to have one collision, and one only, between the blue and the red balls. In terms of lists, this means that there we expect to find one couple $(\mathbf{a}, \mathbf{u}) \in \mathcal{A} \times (\mathcal{B} \oplus \mathcal{C})$ such that $\mathbf{a} = \mathbf{u}$. Then if Equation 6.8 is satisfied, we expect to find a 3XOR between $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$.

Following the same idea, it is possible to prove that the expected number of solutions of the kXOR problem is:

$$\mathbb{E}[N_{col}] = \frac{\prod_{i=1}^{k} N_i}{m}, \tag{6.9}$$

where N_i is the size of $\mathcal{L}_i$, and then, if:

$$\prod_{i=1}^{k} N_i = m,$$

we can expect to have one collision, and one only, with high probability.

6.1.3 Computational Model

We are considering a word-RAM (*Random Access Machine*) model. In other words, we consider that we have a machine in which each "memory cell" contains a w-bit word. We assume that the usual arithmetic (addition, subtraction, multiplication, integer division, modulo) and bitwise operations (AND, OR, XOR, NOT) on w-bit words, as well as comparison of two w-bit integers and memory access with w-bit addresses, are constant time. Furthermore we assume that $w = \theta(n)$. This model is said to be *transdichotomous* (i.e. the machine words fit the size of the problem).

We also assume that we have black-box access to three oracles $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$. When queried with an n-bit integer i, the oracle $\mathbf{A}$ returns the n-bit value $\mathbf{A}(i) = \mathcal{A}[i]$ (the $(i+1)$-th element of the list $\mathcal{A}$). The same goes for $\mathbf{B}$ and $\mathbf{C}$. It is understood that, in most cryptographic applications of the 3XOR problem, querying the oracles actually corresponds to evaluating a cryptographic primitive. As such, we assume that the oracles implement random functions. We can now reformulate Problem 6.1 in a way that suits better our model:

Problem 6.2. Given three random oracles $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$, that produce n-bit (apparently) uniformly random values, find integers i, j and k, such that $\mathbf{A}(i) \oplus \mathbf{B}(j) \oplus \mathbf{C}(k) = 0$.

The machine on which the actual algorithms run is allowed to query the oracles, to store anything in its own memory and to perform elementary operations on w-bit words. An algorithm running on this machine solves the 3XOR problem if it produces a triplet (i, j, k) such that $\mathcal{A}[i] \oplus \mathcal{B}[j] \oplus \mathcal{C}[k] = 0$.

The relevant performance metrics of these algorithms are the amount of memory they need (M words), the number of elementary operations they perform (T) and the number of queries they make to each oracle (Q).

Most algorithms for the 3XOR problem begin by querying the oracle $\mathbf{A}$ on consecutive integers $0, 1, \ldots, N_{\mathcal{A}} - 1$ (the same goes for $\mathbf{B}$ and $\mathbf{C}$ with respective upper-bounds of $N_{\mathcal{B}}$ and $N_{\mathcal{C}}$) and storing the results in memory. We therefore assume that algorithms start their execution with a "local copy" of the lists, obtained by querying the oracles, except when explicitly stated otherwise.

To avoid degenerate cases, we assume that the number of queries to the oracles are exponentially smaller than 2^n. More precisely, we assume that there is a constant $\varepsilon > 0$ such that $\max\{N_{\mathcal{A}}, N_{\mathcal{B}}, N_{\mathcal{C}}\} < 2^{(1-\varepsilon)n}$.

Recovering colliding inputs. It is usually simpler in practice to implement algorithms that produce the colliding values $(\mathcal{A}[i], \mathcal{B}[j], \mathcal{C}[k])$ instead of the colliding inputs (i, j, k). The reason for that is that most algorithms sort at least one of the lists, so that $\mathcal{L}[i]$ is not longer at index i in the array holding $\mathcal{L}$. It would be possible to store pairs $(\mathcal{A}[i], i)$ in memory, sorting on $\mathcal{A}[i]$ and retaining the association with i at the expense of an increased memory consumption. In practice it is much simpler to find the colliding values, then re-query the oracles again to find the corresponding inputs, at the expense of doubling the total number of queries.

6.2 Applications and Related Problems

6.2.1 Applications of the 3XOR Problem

There are only a few applications in cryptography for the 3XOR problem, the best-known one being a forgery attack against the COPA mode of authenticated encryption. We recall the idea of this attack in the following subsection.

However, a sensible improvement for the 3XOR problem may help to improve Wagner's method for the kXOR problem when k is not a power of two.

Attack against COPA Based Authenticated Encryption Schemes

This attack was initially proposed by Nandi [Nan15], and shortly after Nikolić and Sasaki proposed a variant using their 3XOR algorithm as subroutine. We briefly recall the idea of the attack here.

COPA mode of authenticated encryption. COPA is a block authenticated encryption mode, that basically works as follows: given an input message M which actually consists of $d-1$ n-bit vectors denoted $\mathbf{m}_0 \dots \mathbf{m}_{d-2}$ and (possibly) one last vector $\mathbf{m}_{d-1}$ of length s with $0 \leq s < n$, it returns:

$$F_{\mathsf{COPA}}(\mathbf{m}_0 \dots \mathbf{m}_{d-1}) = \begin{cases} \mathbf{c}_0|\dots|\mathbf{c}_{d-2}|\mathbf{t} & \text{if } s = 0 \text{ (there is no } \mathbf{m}_{d-1}\text{).} \\ \mathbf{c}_0|\dots|\mathbf{c}_{d-2}|\mathsf{XLS}(\mathbf{m}_{d-1}|\mathbf{t}) & \text{if } s > 0 \text{ (the last block is incomplete),} \end{cases}$$

where $\mathbf{t}$ is the authentication tag, and XLS is a pseudo-random permutation (PRP). $\mathbf{t}$ depends only on the first $d-1$ message blocks and on the secret key. In other words, given two different $\mathbf{m}_{d-1}$ and $\mathbf{m}'_{d-1}$, the vectors $\mathbf{t}$ obtained while computing $F_{\mathsf{COPA}}(\mathbf{m}_0 \dots \mathbf{m}_{d-1})$ and $F_{\mathsf{COPA}}(\mathbf{m}_0 \dots \mathbf{m}'_{d-1})$ will be identical. The output of $\mathsf{XLS}(\mathbf{m}_{d-1}|\mathbf{t})$ and $\mathsf{XLS}(\mathbf{m}'_{d-1}|\mathbf{t})$ on the other hand will be totally different.

A ciphertext $(\mathbf{c}|\mathbf{t}')$ is valid if and only if the following conditions are all satisfied:

- $\mathbf{c}$ is the ciphertext corresponding to first $d-1$ message blocks.
- $\mathbf{t}'$ is either the authentication tag $\mathbf{t}$ (if the last block is complete) or $\mathsf{XLS}^{-1}(\mathbf{t}') = (\mathbf{m}_{d-1}|\mathbf{t})$, where $\mathbf{t}$ is the authentication tag (if the last block is incomplete).

A verifier who knows the secret key can check these two conditions.

Remark 1. The actual function F_{Enc} used to encrypt $\mathbf{m}_0, \dots, \mathbf{m}_{d-2}$ is not really important here, as the attack is fully based on a flaw in the design of the XLS PRP. We can even assume that F_{Enc} is a random injective function.

Let us just say that, in a simplified version, XLS is an invertible function that given as input a bit-string $(\mathbf{p}|\mathbf{q})$ where $\mathbf{p}$ is of length n and $\mathbf{q}$ is of length $n-1$, returns as output a bit-string $(\mathbf{c}|\mathbf{d})$, where $\mathbf{c}$ is of length n and $\mathbf{d}$ is of length $n-1$. In [Nan14], Nandi proposes a distinguisher against XLS, which he revisits in [Nan15]. Basically, Nandi notices that if two queries, $\mathsf{XLS}(\mathbf{p}_1|\mathbf{q}_1) = (\mathbf{r}_1|\mathbf{s}_1)$ and $\mathsf{XLS}(\mathbf{p}_1|\mathbf{q}_2) = (\mathbf{r}_2|\mathbf{s}_2)$, are made to the XLS oracle, then with probability $1/2$, $\mathsf{XLS}^{-1}(\mathbf{r}_2|\mathbf{s}_3) = (\mathbf{p}_3|\mathbf{q}_3)$, with:

1. $\mathbf{s}_3 = \mathbf{s}_1 \oplus \mathbf{q}_1 \oplus \mathbf{q}_2 \oplus (\mathbf{s}_1 \oplus \mathbf{q}_1 \oplus \mathbf{s}_2 \oplus \mathbf{q}_2)^{\lll 2}$;
2. $\mathbf{q}_3 = (\mathbf{s}_1 \oplus \mathbf{q}_1)^{\lll 2} \oplus \mathbf{q}_2 \oplus (\mathbf{q}_2 \oplus \mathbf{s}_2)^{\lll 2}$,

where the $\lll 2$ operator denotes a 2-bit left rotation. If XLS was indistinguishable from a truly random permutation, this probability would have been 2^{-n+1}. This flaw in the design of XLS leads to a forgery attack against the COPA authenticated encryption family.

Attack against COPA. This is a chosen-plaintext attack. The attacker is assumed to have access to a COPA oracle, and can query it with whatever message suits her. The general idea of the attack is the following: For any fixed $\mathbf{m}_i \in \{0,1\}^n$, and for $(\mathbf{p}_1|\mathbf{q}_1)$ as above, we have:

$$F_{\mathsf{COPA}}(\mathbf{m}_i|\mathbf{p}_1|\mathbf{q}_1) = (\mathbf{c}_i|\mathbf{r}_1|\mathbf{s}_1),$$

where $\mathbf{c}_i$ is the encryption of $\mathbf{m}_i$, and $(\mathbf{r}_1|\mathbf{s}_1) = \mathsf{XLS}(\mathbf{p}_1|\mathbf{s}_1)$. We also have:

$$F_{\mathsf{COPA}}(\mathbf{m}_i|\mathbf{p}_1|\mathbf{q}_2) = (\mathbf{c}_i|\mathbf{r}_2|\mathbf{s}_2),$$

and with probability $1/2$, there is a $\mathbf{p}_3$ such that:

$$F_{\mathsf{COPA}}(\mathbf{m}_i|\mathbf{p}_3|\mathbf{q}_3) = (\mathbf{c}_i|\mathbf{r}_2|\mathbf{s}_3),$$

with $\mathbf{q}_3$ and $\mathbf{s}_3$ defined as above. The point is that $\mathbf{p}_3$ is unknown, and it is not possible to make decryption queries to find it. The decryption query is then simulated by solving an instance of the 3XOR problem. We summarise this attack in Algorithm 13.

Algorithm 13 Nandi's attack against COPA.

Require: A COPA oracle F_{COPA}, a parameter $N \geq 2^{n/3}$.
Ensure: A forged cipher (C, T), where T is a valid tag, or $\perp$ if the attack fails.

1: **for** $0 \leq i < N-1$ **do**
2: Choose $\mathbf{m}_i \in \{0,1\}^n$ and set $(\mathbf{c}_i|\mathbf{t}_i) \leftarrow F_{\mathsf{COPA}}(\mathbf{m}_i)$
3: Set $\mathbf{q}_i \leftarrow \mathbf{t}_i[1..]$
4: Set b such that $|I| = |\{i \text{ s.t. } t_i[0] = b\}| \geq N/2$.
5: Partition I into I_1 and I_2 such that $|I_1| = |I_2|$.
6: Choose an arbitrary $\mathbf{m} \in \{0,1\}^{n-1}$
7: **for all** $i \in I$ **do**
8: Set $(\mathbf{c}_i|\mathbf{r}_i|\mathbf{s}_i) \leftarrow F_{\mathsf{COPA}}(\mathbf{m}_i|\mathbf{m})$
9: Build the lists $\mathcal{A} \leftarrow \{\mathbf{q}_i \text{ for } i \in I\}$, $\mathcal{B} \leftarrow \{(\mathbf{r}_j \oplus \mathbf{q}_j)^{\lll 2} \text{ for } j \in I_2\}$ and $\mathcal{C} \leftarrow \{\mathbf{q}_k \oplus (\mathbf{r}_k \oplus \mathbf{q}_k)^{\lll 2}\}$
10: Find $(\mathbf{a}_i, \mathbf{b}_j, \mathbf{c}_k) \in \mathcal{A} \times \mathcal{B} \times \mathcal{C}$ such that $\mathbf{a}_i = \mathbf{b}_j \oplus \mathbf{c}_k$
11: **if** $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ not found **then**
12: **return** $\perp$
13: Set $\mathbf{s}^* \leftarrow (\mathbf{s}_j \oplus \mathbf{q}_j \oplus \mathbf{q}_k) \oplus (\mathbf{s}_j \oplus \mathbf{q}_j \oplus \mathbf{s}_k \oplus \mathbf{q}_k)$
14: **return** $(\mathbf{c}_i|\mathbf{r}_j|\mathbf{s}^*)$.

Nandi claims that the output $(\mathbf{c}_i|\mathbf{r}_i|\mathbf{s}^*)$ is a valid ciphertext with probability $1/4$. Indeed, to be valid $(\mathbf{c}_i|\mathbf{r}_j|\mathbf{s}^*)$ must satisfy $(\mathbf{c}_i|\mathbf{r}_i|\mathbf{s}^*) = F_{\mathsf{COPA}}(\mathbf{m}_i|\mathbf{m}^*)$, for some $\mathbf{m}^* \in \{0,1\}^n - 1$. According to the definition of F_{COPA} this amounts to saying that $\mathsf{XLS}((\mathbf{m}^*|\mathbf{t}_i[0])|\mathbf{t}_i[1..]) = (\mathbf{r}_i|\mathbf{s}^*)$. If we denote by $\mathbf{p}^*$ the vector $(\mathbf{m}^*|\mathbf{t}_i[0])$ and by $\mathbf{q}_i$ the vector $\mathbf{t}_i[1..]$ we have the following equivalence: $(\mathbf{c}_i|\mathbf{r}_i|\mathbf{s}^*)$ is valid, if and only if there is $\mathbf{p}^* \in \{0,1\}^n$ which satisfies the two following conditions:

[*Condition 1*] $\mathbf{p}^*[n-1] = b$

[*Condition 2*] $\mathsf{XLS}(\mathbf{p}^*|\mathbf{q}_i) = (\mathbf{r}_i|\mathbf{s}^*)$

Now, recall that $(\mathbf{r}_k|\mathbf{s}_k) = \mathsf{XLS}(\mathbf{p}|\mathbf{q}_k)$ and $(\mathbf{r}_j|\mathbf{s}_j) = \mathsf{XLS}(\mathbf{p}|\mathbf{q}_j)$, where $\mathbf{p} = (\mathbf{m}|b)$. Then with probability $1/2$, $\mathsf{XLS}^{-1}(\mathbf{r}_k, \mathbf{s}^*) = (\mathbf{p}^*|\mathbf{q}^*)$, with:

1. $\mathbf{s}^* = (\mathbf{s}_j \oplus \mathbf{q}_j \oplus \mathbf{q}_k) \oplus (\mathbf{s}_j \oplus \mathbf{q}_j \oplus \mathbf{s}_k \oplus \mathbf{q}_k)$
2. $\mathbf{q}^* = (\mathbf{s}_j \oplus \mathbf{q}_j)^{\lll 2} \oplus \mathbf{q}_k \oplus (\mathbf{q}_k \oplus \mathbf{s}_k)^{\lll 2}$

$\mathbf{s}^*$ is actually chosen to satisfy this in step 13. And the 3XOR step ensures that $\mathbf{q}_j = \mathbf{q}^*$. So, with probability $1/2$, *Condition 2* is satisfied. *Condition 1* is also satisfied with probability $1/2$, as $\mathbf{p}^*$ is uniformly distributed. The two conditions are independent. It follows that the attack will succeed with probability $1/4$.

Other Applications

An improved 3XOR algorithm is also interesting for searching parity check relations in fast correlation attacks, with $k = 3$. The authors of [CJM02], studied the problem only with $k \geq 4$.

A recent work concerning Time/Memory tradeoff for the BKW algorithm [EHK+18] accepted at CRYPTO 2018, propose to consider finding cXORs for a constant $c > 2$, instead of 2XORs, to cancel some blocks of bits. They claim then that the LPN problem can then be solved in time $\mathsf{T} = 2^{\log c \frac{k}{\log k} + o(1)}$ and memory $\mathsf{M} = 2^{\frac{\log(c) k}{(c-1)k} + o(1)}$. This allows them to reduce the memory consumption almost by a factor $1/c$ of the algorithm, at the cost of increasing the by a $(\log c)$ factor the running time exponent. The parameter c should not be too high however, otherwise the running time will be too high. In a personal conversation with the authors of [EHK+18], they stated that an algorithm which solves the 3XOR problem significantly faster than the birthday problem could possibly lead to an interesting time-memory tradeoff in BKW. Unfortunately, all existing algorithms for the 3XOR problem (including ours) only gain a polynomial factor compared to birthday search, which disappears in the $o(1)$ component of the exponent. Finally, we recall that the idea of computing cXORs instead of 2XORs has been studied before, first by Levieil and Fouque [LF06] and more recently by Zang et al. [ZJW16].

6.2.2 Related Problems

Generalised Birthday Problem and Variants

Generalised Birthday Problem. In 2002, Wagner introduced the Generalised Birthday Problem (GBP) [Wag02]. He considers an arbitrary number of lists k, with $k > 2$. His goal was to find an element $\mathbf{x}_i$ in each one the $\mathcal{L}_i$ lists such that: $\mathbf{x}_1 \oplus \mathbf{x}_2 \oplus \ldots \oplus \mathbf{x}_k = 0$. This problem is also known as the k-list or the kXOR problem. Wagner studied the difficulty of this problem in different settings. In particular, he proposed an efficient algorithm for the case where k is a power of two (see Section 7.2.1 for the case $k = 4$). The authors of [EHK+18] also improved Wagner's method for some values of k, using dissection techniques [DDKS12]. However, the particular case of the 3XOR problem remains a difficult case of the Generalised Birthday Problem.

Approximate-3XOR Quite recently, Both and May [BM17] have studied the following variant of the 3XOR problem: Given three lists $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, of n-bit vectors, find $(\mathbf{a}, \mathbf{b}, \mathbf{c}) \in \mathcal{A} \oplus \mathcal{B} \oplus \mathcal{C}$ such that $wt(\mathbf{a} \oplus \mathbf{b} \oplus \mathbf{c}) = \delta n$, for some approximation parameter $\delta \in [0, 1/2]$. In this case, they were able to design an algorithm which solves this problem quite efficiently, even for somewhat small values of δ. For instance, when $\delta = 0.1$, they achieved a running time of about $2^{0.2818n}$. However, when $\delta = 0$ (i.e. the case of the actual 3XOR problem), their method is similar to the one in [NS15], and brings no improvement compared to previous work.

Decisional 3SUM/3XOR and Complexity Assumption

The (decisional) 3SUM problem with three lists of size N has been widely studied, in the world of complexity theory. Indeed, many geometric problems can be reduced to it in sub-quadratic time, and thus are said to be 3SUM-hard [GO95]. Testing whether a set of points in the plane contains three collinear points is a notable example thereof. It has been conjectured that solving the decisional 3SUM problem requires $\Omega\left(N^2\right)$ operations [Eri95]. More recent works [Pat10, KPP16] tend to thinks that it can be solved in time $\Omega\left(N^{2-o(1)}\right)$.

The 3XOR variant of this problem has been less studied than the 3SUM. Nonetheless, it has been shown that a $\mathcal{O}\left(N^{2-\varepsilon}\right)$ algorithm for the (decisional) 3XOR with lists of size N would imply a faster-than-expected algorithm for listing triangles in a graph [Vio12, JV13]. A recent result from [DSW18], accepted at MFCS'18, shows that if the 3XOR problem can be solved in time $\Omega\left(n^{2-o(1)}\right)$, it would also reduce the time complexity of the offline SetDisjointness and offline SetIntersection problems. The interesting point is that a similar result for these two problems had been proved before in [KPP16], with the 3SUM problem. This makes the authors of [DSW18] wonder if there might be some relation between the optimal expected time for the 3XOR problem

and the 3SUM problem. For the record, given a finite set C, finite families A and B of subsets of C, and q pairs of subset $(S, S') \in A \times B$, the offline SetDisjointness problem consists in finding all of the q pairs (S, S') with $S \cap S' \neq \emptyset$, and the offline SetIntersection problem consists in listing all elements of the intersections $S \cap S'$ of the q pairs (S, S').

Remark 2. The 3XOR problem of which we talk in this section is slightly different than the one we are concern with. Indeed, the input lists are supposed to be of size at most N, that are not necessarily uniformly random, and there may be no solution at all, while in our case, the lists consist of uniformly random elements, and may be as big as we want, so that there will be at least one solution. The two problems are however very similar. In our case, randomness allows us to use some algorithmic tricks, which may not apply directly to a more generic case (e.g. radix sort, dispatch procedure).

6.3 Algorithmic Tools

Algorithms for the 3XOR problem process exponentially-long lists of random n-bit vectors. In this section, we review the algorithmic techniques needed to perform three reoccurring operations on such lists: sorting, testing membership and right-multiplication of each vector by an $n \times n$ matrix. We recall that these operations can be performed in time and memory that is proportional to the size of the input lists.

6.3.1 Hashing

The simplest algorithm for the 3XOR problem works by considering all pairs $(\mathbf{a}, \mathbf{b}) \in \mathcal{A} \times \mathcal{B}$ and checking if $\mathbf{a} \oplus \mathbf{b} \in \mathcal{C}$. This is guaranteed to find all solutions. The problem is to quickly test whether an element belongs in a list $\mathcal{C}$.

It has long been known that hashing allows for dictionaries with $\mathcal{O}(1)$ access-time and constant multiplicative space overhead [FKS84]. This is enough to make the algorithm above run in time $\mathcal{O}(N_{\mathcal{A}} N_{\mathcal{B}} + N_{\mathcal{C}})$. In this case, it is thus beneficial to let $\mathcal{C}$ be the largest of the three lists. All we need to do now, is to find a hash table that will fit our problem.

Linear probing. A simple hash table of size $2N_{\mathcal{C}}$ with *linear probing* allows to test membership with 2.5 probes on average [Knu98, §6.4]. We recall that linear probing is a scheme used to resolve collisions in hash tables. Let us consider a hash table T, with hash function h. To insert an element $\mathbf{x}$ in T, check if $T[h(\mathbf{x})]$ is empty. If so, $\mathbf{x}$ is inserted at index $h(\mathbf{x})$. If not, probe index $h(\mathbf{x}) + 1$, then $h(\mathbf{x}) + 2$, and so on, until an empty cell is found. We give an example of a hash table that utilise linear probing in Figure 6.1. In this example, the hash function utilised is $h : x \to x \mod 8$, the insertion order is given on the right. To search for a given $\mathbf{x}$ in T, first probe $h(\mathbf{x})$. If $T[h(\mathbf{x})]$ contains something else than $\mathbf{x}$, then probe $T[h(\mathbf{x}) + 1]$, and so on, until finding either an empty cell ($\mathbf{x}$ does not belong to the table), or a cell whose stored key is $\mathbf{x}$.

Because the hashed elements are uniformly random, taking some lowest-significant bits yields a very good hash.

Cuckoo hashing. Linear probing is already good, but *Cuckoo Hashing* improves the situation to 2 probes in the worst case [PR01]. In this case two arrays of size $N_{\mathcal{C}}$, T_1 and T_2, with two hash functions h_1 and h_2 are considered. If $\mathbf{x}$ belongs to the hash table, it is either in $T_1[h_1(\mathbf{x})]$ or in $T_2[h_2(\mathbf{x})]$. To insert an element $\mathbf{x}$ in the table, set $T_1[h_1(\mathbf{x})]$ to $\mathbf{x}$. If $T_1[h(\mathbf{x})]$ was not empty, then there was a key $\mathbf{y}$ stored in this cell before. The insertion removes $\mathbf{y}$ from table T_1, so we have to insert it in $T_2[h_2(\mathbf{y})]$. If there was already a key $\mathbf{z}$ stored in $T_2[h_2(\mathbf{y})]$, then $\mathbf{z}$ is removed, so we insert it in $T_1[h_1(\mathbf{z})]$, and so on. We illustrate this in Figure 6.2. This process may fail because of cycles, in which case the two arrays have to be rebuilt with new hash functions. Insertion requires an expected constant number of operations. In principle, Cuckoo hashing needs a family of universal hash functions, which is a drawback in our case, as their evaluation is in general computationally costly. We may use the usual universal hash functions $h_{a,b} : \mathbf{x} \mapsto (a\mathbf{x} + \mathbf{b}) \bmod p$, where p is a prime.

$T[0]$	16	(2)
$T[1]$	1	(1)
$T[2]$	32	(5)
$T[3]$	67	(3)
$T[4]$	36	(6)
$T[5]$	44	(7)
$T[6]$	80	(8)
$T[7]$	15	(4)

Figure 6.1 – Example of a hash table that utilises linear probing.

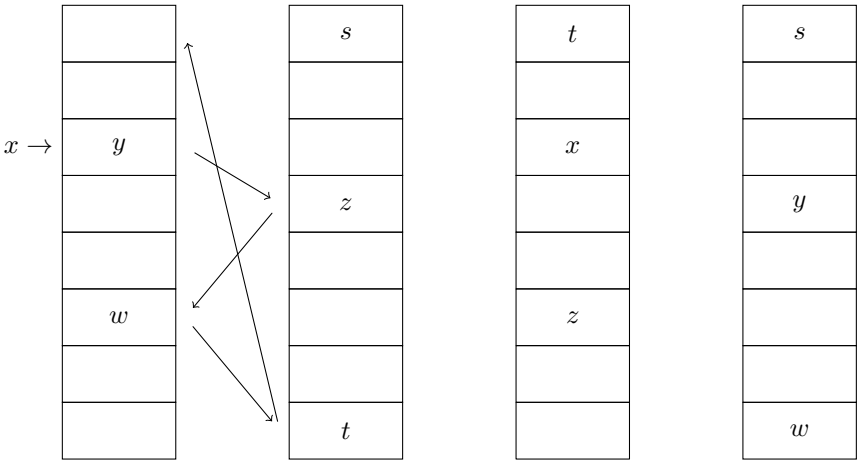


Figure 6.2 – Example of a hash table that utilise cuckoo hashing.

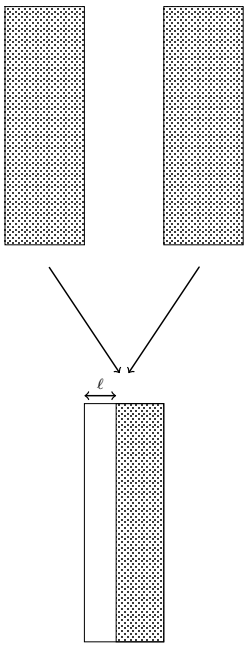


Figure 6.3 – Illustration on the join over ℓ bits.

We found that Cuckoo hashing was about 2× faster than linear probing. It allows to make the quadratic algorithm quite efficient, with only a small constant hidden in the $\mathcal{O}$ notation: around 10 CPU cycles are enough to process a pair.

Remark 1. Note that the size of the hash tables above are given as an example. The same methods would also work with tables (slightly) smaller.

6.3.2 Sorting and Joining

The evaluation of the "join" $\mathcal{A} \bowtie_\ell \mathcal{B}$ of two lists $\mathcal{A}$ and $\mathcal{B}$ of size $2^{\alpha n}$ is ubiquitous in generalised birthday algorithms following Wagner's ideas. All these algorithms rely on a common idea: they target a solution $\mathbf{a} \oplus \mathbf{b} \oplus \mathbf{c} = 0$ such that $\mathbf{c}[0..\ell] = 0$ (for some a priori fixed value of ℓ), which implies that $\mathbf{a}[..\ell] = \mathbf{b}[..\ell]$. These special solutions can be found efficiently by performing a linear-time join operation between $\mathcal{A}$ and $\mathcal{B}$. Algorithm 14 is thus the workhorse of these techniques. An illustration of the join is given in Figure 6.3.

It is also possible to check if a pair $(\mathbf{a}, \mathbf{b})$ of $\mathcal{A} \bowtie_\ell \mathcal{B}$ XORs to an element of $\mathcal{C}$ as the algorithm goes. For that, we only have to slightly modify Algorithm 14 into Algorithm 15

Radix sort. It is well-known that an array of N random ℓ-bit vectors can be sorted in *linear* time [Knu98, §5.2.5]: the randomness of the input allows to beat the $\Omega(N \log N)$ lower-bound of comparison-based sorting. Here is a way to do it using $\mathcal{O}(\sqrt{N})$ extra storage: perform two passes of *radix sort* (see Figure 6.4 for an illustration of the radix sort) on $0.5 \log N$ bits, then finish sorting with insertion sort. Morally, for each pass of radix sort, we initialise $\sqrt{N}$ counters corresponding to the possible prefixes, then we scan through the list, pick up the $0.5 \log N$-bit prefix of each vector, increment the corresponding counter. Then the entries can directly be dispatched in the right output bucket.

Each pass of radix sort requires $\sqrt{N}$ words to store the counters. Besides that, the input array can in principle be permuted in-place [Knu98, §5.2.1]. The two passes of radix sort guarantee that the array is sorted according to its first $\log N$ bits. This reduces the expected number of inversions from $\approx N^2/4$ to $\approx N/4$. Thus, the expected running time of the insertion sort will be linear.

Complexity analysis of Algorithm 15. Step 1 of Algorithm 15 runs in time linear in $N_\mathcal{A} + N_\mathcal{B}$. Steps 11 to 18 are repeated at most $N_\mathcal{A} + N_\mathcal{B}$ times and require a constant number of operations on w-bit words. The test of step 15 is executed once for each pair $(\mathbf{a}, \mathbf{b}) \in \mathcal{A} \times \mathcal{B}$ with $\mathbf{a}[..\ell] = \mathbf{b}[..\ell]$. The expected number of such pairs is $N_\mathcal{A} N_\mathcal{B} / 2^\ell$. Considering that we also have to initialise

Algorithm 14 Join.

Require: Lists $\mathcal{A}, \mathcal{B}$ and a parameter $0 < \ell < n$.
Ensure: All $\mathcal{A} \bowtie_\ell \mathcal{B}$.

Sort $\mathcal{A}$ and $\mathcal{B}$ according to their first ℓ bits
Set $\mathcal{A} \bowtie_\ell \mathcal{B} \leftarrow \perp$
Set $i \leftarrow 0$, $j \leftarrow 0$
while $i < N_\mathcal{A}$ and $j < N_\mathcal{B}$ **do**
    if $\mathcal{A}[i][..\ell] < \mathcal{B}[j][..\ell]$ **then**
        $i \leftarrow i + 1$
    else if $\mathcal{A}[i][..\ell] > \mathcal{B}[j][\ell]$ **then**
        $j \leftarrow j + 1$
    else
        Set $\mathbf{p} \leftarrow \mathcal{A}[i][..\ell]$, $j_0 \leftarrow j$
        while $i < N_\mathcal{A}$ and $\mathcal{A}[i][..\ell] = \mathbf{p}$ **do**
            $j \leftarrow j_0$
            while $j < N_\mathcal{B}$ and $\mathcal{B}[j][..\ell] = \mathbf{p}$ **do**
                $\mathcal{A} \bowtie_\ell \mathcal{B} \leftarrow \mathcal{A} \bowtie_\ell \mathcal{B} \cup \{(\mathcal{A}[i], \mathcal{B}[j])\}$
                $j \leftarrow j + 1$
            $i \leftarrow i + 1$

Algorithm 15 Join and test membership.

Require: Lists $\mathcal{A}, \mathcal{B}$ and a hash table T which contains n-bit vectors $\mathbf{c}$ such that $\mathbf{c}[..\ell]$
Ensure: All $(a, b) \in \mathcal{A} \times \mathcal{B}$ such that $a \oplus b \in \mathcal{C}$.

Sort $\mathcal{A}$ and $\mathcal{B}$ according to their first ℓ bits
Initialise a hash table with entries of $\mathcal{C}$
Set $Res \leftarrow \perp$
Set $i \leftarrow 0$, $j \leftarrow 0$
while $i < N_\mathcal{A}$ and $j < N_\mathcal{B}$ **do**
    if $\mathcal{A}[i][..\ell] < \mathcal{B}[j][..\ell]$ **then**
        $i \leftarrow i + 1$
    else if $\mathcal{A}[i][..\ell] > \mathcal{B}[j][\ell]$ **then**
        $j \leftarrow j + 1$
    else
        Set $\mathbf{p} \leftarrow \mathcal{A}[i][..\ell]$, $j_0 \leftarrow j$
        while $i < N_\mathcal{A}$ and $\mathcal{A}[i][..\ell] = \mathbf{p}$ **do**
            $j \leftarrow j_0$
            while $j < N_\mathcal{B}$ and $\mathcal{B}[j][..\ell] = \mathbf{p}$ **do**
                if TestMembership$(\mathcal{A}[i] \oplus \mathcal{B}[j], T) =$ `True` **then**
                $Res \leftarrow Res \cup \{(\mathcal{A}[i], \mathcal{B}[j])\}$
                $j \leftarrow j + 1$
            $i \leftarrow i + 1$

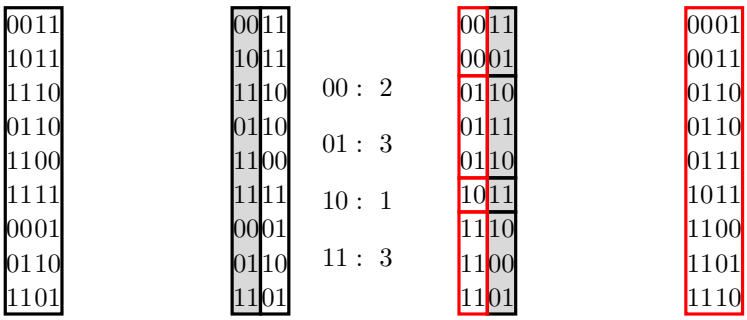


Figure 6.4 – Radix sort on 2 bits.

a hash table with the elements of $\mathcal{C}$, the total expected running time of the algorithm is thus $\mathcal{O}\left(N_{\mathcal{A}} + N_{\mathcal{B}} + N_{\mathcal{C}} + N_{\mathcal{A}}N_{\mathcal{B}}/2^{\ell}\right)$.

6.3.3 Matrix Multiplication

We consider a $n \times n$ matrix M over $\mathbb{F}_2$. Computing the list $\mathcal{A}M$ naively would require $\mathcal{O}(n \times N_{\mathcal{A}})$ operations. Fortunately, if $N_{\mathcal{A}} = 2^{\alpha n}$, this can improved to $\mathcal{O}(2^{\alpha n})$ operations, using a trick similar to the "method of Four Russians" [KADF70].

The method of the Four Russians. We briefly recall how the original Four Russian method works, before explaining our adaptation to our case. The authors of [KADF70] consider two square matrices A and B of size N with $N = \mathcal{O}(\log w)$, where w is the size of a machine word and they aim to compute the product $C = AB$. To do so, they first split B into blocks B_{ij} of size $t \times t$ where $t = \varepsilon \log N$ and ε is an arbitrary parameter greater than 0.

For each i, j they create a lookup table T_{ij} corresponding to B_{ij} such that for all $\mathbf{v} \in \mathbb{F}_2^t$, $T_{ij}[\mathbf{v}] = \mathbf{v}B_{ij}$. Then, for all rows $\mathbf{a}_i$ of A, they split $\mathbf{a}$ into slices of t entries. Denoting by $\mathbf{a}_i^k$ the k-th slice of $\mathbf{a}_i$, for each row $\mathbf{c}_i$ of C, a slice $\mathbf{c}_i^j$ of $\mathbf{c}_i$ satisfies $\mathbf{c}_i^j = \mathbf{a}_i^1 A_{1j} \oplus \cdots \oplus \mathbf{a}_i^{N/t} A_{(N/t)j}$.

The computation of each $\mathbf{a}_i^k A_{kj}$ can be done in constant time by looking at the table. Computing a slice $\mathbf{c}_i^j$ of c_i, can then be done by simply computing N/t XORs. All in all, this method requires to compute $N(N/t)$ such slice, and thus this computation has a time complexity of:

$$\mathsf{T} = \mathcal{O}\left(\frac{N^3}{t}\right) = \mathcal{O}\left(\frac{N^3}{\log N}\right),$$

and the computation of the tables requires:

$$\mathsf{T} = \mathcal{O}\left(\left(\frac{N}{\log N}\right)^2 N^{\varepsilon} \log N\right) = \mathcal{O}\left(\frac{N^{\varepsilon+2}}{\log N}\right).$$

Indeed, there are $(N/t)^2$ tables to compute, each one containing 2^t vectors, and the computation of an entry of each table requires to compute a vector-matrix product of size where the matrix is square of $\log N$. This product can be computed in $\mathcal{O}(\log N)$ as long as $\log N = \mathcal{O}(w)$. The extra memory required is:

$$\mathsf{M} = \mathcal{O}\left(\frac{N^{2+\varepsilon}}{\log N^2}\right).$$

This method has been widely studied and has known some improvements, decreasing the asymptotic complexity in the last few years [BW09, Cha15, Hua15].

Remark 2. This method was called the 4 Russians' Method "after the cardinality and the nationality of its inventors" Arlazarov, Dinic, Kronrod and Faradzev. It is however not clear if the four authors were actually Russians at the time they designed it (in 1970).

A different tradeoff. We are interested in a different case where we want to compute the product of a $2^{\alpha n}$-by-n matrix A and an n-by-n matrix M, where $n = \Theta(w)$.

Following the same idea as [KADF70], we divide M into slices of $t = n\frac{\alpha}{1+\varepsilon}$ rows (for an arbitrary $\varepsilon > 0$), and for each slice i, we precompute all the $2^{\frac{\alpha}{1+\varepsilon}n}$ linear combinations of the rows and store these in a table T_i. This takes $2^{\frac{\alpha}{1+\varepsilon}n}$ word operations.

Then, for each row $\mathbf{a}_i$ of A, the vector-matrix product $\mathbf{a}_i M$ can then be evaluated by dividing $\mathbf{a}_i$ into slices of size $n\frac{\alpha}{1+\varepsilon}$, and by looking at the corresponding entries $T_k[\mathbf{a}_i^k]$ for all k and XORing together the $\frac{1+\varepsilon}{\alpha}$ resulting vectors.

We claim that computing $\mathbf{a}_i M$ takes only a constant number of operations. Indeed, we actually only need to compute n/t XORs, with our definition of t, this consists in a constant number of XORs that can each be computed in constant time as long as $n = \Theta(w)$. We will need to repeat this step $2^{\alpha \cdot n}$ times. Hence, the whole complexity of the procedure is $\mathcal{O}(2^{\alpha n})$ word operations, but it requires to use $\mathcal{O}\left(2^{\frac{\alpha}{1+\varepsilon}n}\right)$ extra memory.

Chapter 7

Of Previous 3XOR Algorithms: Revisiting and Discussing Them

In this chapter, we revisit and discuss previous algorithms such as the naive quadratic algorithm, Wagner kXOR algorithms, and the later improvements in the 3XOR case by Nikolić and Sasaki, and by Joux. We also present a time-memory tradeoff introduced by Bernstein called the clamping trick.

7.1 The Quadratic Algorithm

A naive algorithm. We assume that we have made $N_{\mathcal{A}}$ queries to $\mathbf{A}$, $N_{\mathcal{B}}$ queries to $\mathbf{B}$ and $N_{\mathcal{C}}$ queries to $\mathbf{C}$, such that the product $N_{\mathcal{A}}N_{\mathcal{B}}N_{\mathcal{C}} = 2^n$. We expect to have exactly one triplet $(\mathbf{a}, \mathbf{b}, \mathbf{c}) \in \mathcal{A} \times \mathcal{B} \times \mathcal{C}$ such that $\mathbf{a} \oplus \mathbf{b} \oplus \mathbf{c} = 0$.

In this setting, we can find the solution to the problem using the folklore quadratic algorithm, which basically does the following:

1. Take all pairs $(\mathbf{a}, \mathbf{b})$ from $\mathcal{A} \times \mathcal{B}$,
2. If $(\mathbf{a}, \mathbf{b}) \in \mathcal{C}$, return $(\mathbf{a}, \mathbf{b}, \mathbf{a} \oplus \mathbf{b})$ and terminate the procedure,
3. Else, discard the pair and continue with an other one.

Armed with the algorithmic tools described in Section 6.3, the expected time of the procedure will be $T = \mathcal{O}\left(N_{\mathcal{A}}N_{\mathcal{B}} + N_{\mathcal{C}}\right)$ and the memory required will be $M = \mathcal{O}\left(N_{\mathcal{A}} + N_{\mathcal{B}} + N_{\mathcal{C}}\right)$.

For instance, if we decide to make $2^{n/3}$ queries to each oracles, which is the minimum number of queries required to have a solution with high probability, we will have $\mathsf{T} = \mathcal{O}\left(2^{2n/3}\right)$ and $\mathsf{M} = \mathcal{O}\left(2^{n/3}\right)$. As the algorithm is proportional to the product $N_{\mathcal{A}}N_{\mathcal{B}}$, we can decide to make less queries to oracles $\mathbf{A}$ and $\mathbf{B}$: only $2^{n/4}$, and then increase the number of queries to $\mathbf{C}$ up to $2^{n/2}$ so that there will still be a solution with high probability. In this particular setting, we have reduced the time complexity of the quadratic algorithm described above to $\mathsf{T} = \mathcal{O}\left(2^{n/2}\right)$, at the cost of increasing both the total number of queries and the space complexity to $\mathcal{O}\left(2^{n/2}\right)$.

This naive example shows that this problem is not as simple as it seems, and that time-memory tradeoffs are to be taken into account.

A worst-case quadratic algorithm. Recently Dietzfelbinger, Schlag and Walzer [DSW18] proposed a variant of this quadratic algorithm that allowed them to solve the problem in time $\mathcal{O}\left(N_{\mathcal{A}}(N_{\mathcal{C}} + N_{\mathcal{B}}) + N_{\mathcal{C}} \log N_{\mathcal{C}}\right)$, even in the worst-case, without hash tables. They actually precompute a binary tree associated to the one of the list (say $\mathcal{C}$), such that:

- The leafs of the tree are the element of the lists $\mathcal{C}$

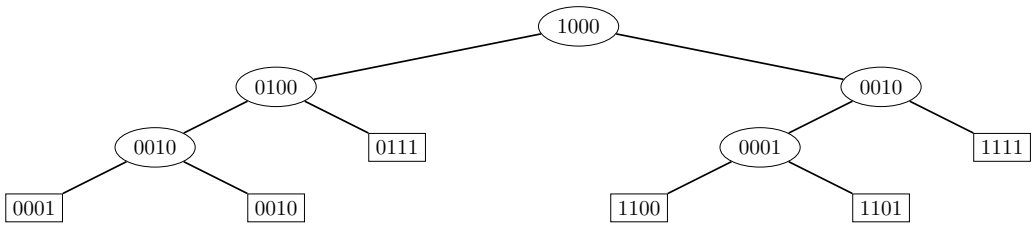


Figure 7.1 – Tree associated to the list $\{(0001), (0010), (0111), (1100), (1101), (1111)\}$.

- The inner nodes are bit strings $\mathbf{x}$, such that the first non-zero bit of $\mathbf{x}$, indicates the position of the first bit which differs in the leafs of the sub-tree whose root is $\mathbf{x}$.

This kind of structure is sometimes referred as Patricia Tree (e.g. [Knu98, § 6.3]). For a given inner node $\mathbf{x}$ whose first non zero bit is $\mathbf{x}[i]$. We denote by $\textsc{LeftSubtree}(T, \mathbf{x})$ the sub-tree of T, whose root is the left child of $\mathbf{x}$. Equivalently $\textsc{RightSubtree}(T, \mathbf{x})$ is the sub-tree of T, whose root is the right child of $\mathbf{x}$. All the leafs $\mathbf{c}_\ell$ of $\textsc{LeftSubtree}(T, \mathbf{x})$ are such that $\mathbf{c}_\ell[i] = 0$, and all the leafs $\mathbf{c}_r$ of $\textsc{RightSubtree}(T, \mathbf{x})$ are such that $\mathbf{c}_r[i] = 1$. An example of such a tree is given in Figure 7.1.

The authors of [DSW18] use as a subroutine a Traverse procedure, which takes as input a vector $\mathbf{a}$ and a tree T associated to the list $\mathcal{C}$, and returns the sorted list $\mathbf{a} \oplus \mathcal{C}$ in time $\mathcal{O}(N_\mathcal{C})$. This procedure is given in Algorithm 16. Using this trick, they are able to search a 3XOR in time $\mathcal{O}(N_\mathcal{A}(N_\mathcal{C} + N_\mathcal{B}) + N_\mathcal{C} \log N_\mathcal{C} + N_\mathcal{B} \log N_\mathcal{B})$ using the following procedure:

1. Pre-compute the tree T associated to the elements of $\mathcal{C}$ and sort $\mathcal{B}$.
2. For all $\mathbf{a} \in \mathcal{A}$, compute the sorted list $\mathbf{a} \oplus \mathcal{C}$.
3. Scan through the lists $\mathbf{a} \oplus \mathcal{C}$ and $\mathcal{B}$ to search for collisions.

Step 1 can be performed in time $\mathcal{O}(N_\mathcal{C} \log N_\mathcal{C} + N_\mathcal{B} \log N_\mathcal{B})$ (this step consists mostly in sorting the lists). Step 2 consists in visiting the nodes of a binary tree which contains at most $2N_\mathcal{C} - 1$ nodes, and this step has to be performed $\mathcal{O}(N_\mathcal{A})$ times so the total complexity of step 2 is $\mathcal{O}(N_\mathcal{A} N_\mathcal{C})$. Finally step 3 requires about $\mathcal{O}(N_\mathcal{C} + N_\mathcal{B})$ operations, and has to be performed $\mathcal{O}(N_\mathcal{A})$ times.

Algorithm 16 Dietzfelbinger, Schlag and Walzer Traverse algorithm.

Require: A tree T associated to the list $\mathcal{C}$, a vector $\mathbf{a}$
Ensure: The list $\mathcal{L} = \mathbf{a} \oplus \mathcal{C}$ in sorted order.

```
1:  if T = {c} then                                   ▷ T consists only of one node
2:      Emit a ⊕ x
3:  else
4:      Let r be the root of T                        ▷ T has at least three nodes
5:      if a ⊕ r > a then
6:          Traverse(LeftSubtree(T, r), a)
7:          Traverse(RightSubtree(T, r), a)
8:      else
9:          Traverse(RightSubtree(T, r), a)
10:         Traverse(leftSubtree(T, r), a)
```

7.2 Wagner's Algorithm for the kXOR Problem

In this section we recall the idea of Wagner' s algorithm for the kXOR problem. We detail the case $k = 4$, where the method works well and explain why it cannot easily be transposed to the case $k = 3$.

7.2.1 Wagner's 4-Tree Algorithm

We first explain Wagner's Algorithm in the case of the 4XOR problem. We consider four lists $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}_3$ and $\mathcal{L}_4$, containing $2^{n/3}$ n-bit entries each. In this setting, from Equation 6.9, the expected number of solutions to the problem is:

$$\mathbb{E}[N_{col}] = \frac{2^{4n/3}}{2^n} = 2^{n/3}.$$

The idea of the 4-tree algorithm is to choose some arbitrary $n/3$-bit prefix $\mathbf{p}$ such that the targeted solution $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4) \in \mathcal{L}_1 \times \mathcal{L}_2 \times \mathcal{L}_3 \times \mathcal{L}_4$ satisfies: $\mathbf{x}_{12} = \mathbf{x}_1 \oplus \mathbf{x}_2$, $\mathbf{x}_{34} = \mathbf{x}_3 \oplus \mathbf{x}_4$, and $\mathbf{x}_{12}[..n/3] = \mathbf{x}_{34}[..n/3] = \mathbf{p}$. For simplicity, we assume that $\mathbf{p} = 0$.

According to Wagner, such a solution $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4)$ exists with high probability. Indeed, if you consider the joined list $\mathcal{L}_1 \bowtie_{n/3} \mathcal{L}_2$ and if you denote by N_{12} the number of couples $(\mathbf{x}_1, \mathbf{x}_2) \in \mathcal{L}_1 \bowtie_{n/3} \mathcal{L}_2$ (i.e. couples $(\mathbf{x}_1, \mathbf{x}_2) \in \mathcal{L}1 \times \mathcal{L}_2$ such that $\mathbf{x}_1[..n/3] = \mathbf{x}_2[..n/3]$), Equation 6.5 implies:

$$\mathbb{E}[N_{12}] = \frac{2^{2n/3}}{2^{n/3}} = 2^{n/3}.$$

Similarly, $\mathbb{E}[N_{34}] = 2^{n/3}$, where N_{34} is the number of elements in $\mathcal{L}_3 \bowtie_{n/3} \mathcal{L}_4$.

A solution, to the problem is then given by two couples: $(\mathbf{x}_1, \mathbf{x}_2)$ in $\mathcal{L}_1 \bowtie_{n/3} \mathcal{L}_2$ and $(\mathbf{x}_3, \mathbf{x}_4)$ in $\mathcal{L}_3 \bowtie_{n/3} \mathcal{L}_4$ such that if we denote by $\mathbf{x}_{12}$ the XOR $\mathbf{x}_{12} = \mathbf{x}_1 \oplus \mathbf{x}_2$ and $\mathbf{x}_{34}$ the XOR $\mathbf{x}_{34} = \mathbf{x}_3 \oplus \mathbf{x}_4$, $\mathbf{x}_{12}$ and $\mathbf{x}_{34}$ agree on the remaining $2n/3$ bits. In other words,

$$\mathbf{x}_{12}[n/3..] = \mathbf{x}_{34}[n/3..].$$

The expected number of solutions is then:

$$\mathbb{E}[N_{sol}] = \frac{N_{12}N_{34}}{2^{2n/3}} \simeq \frac{2^{2n/3}}{2^{2n/3}} = 1.$$

This method is summarised in Algorithm 17.

Remark 1. This method is called the 4-tree Algorithm because it can be illustrated by a tree with four leaves representing the input lists (see Figure 7.2).

Algorithm 17 Wagner's 4-Tree Algorithm.

Require: Lists $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3$ and $\mathcal{L}_4$ of size $2^{n/3}$
Ensure: A triplet $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) \in \mathcal{L}_1 \times \mathcal{L}_2 \times \mathcal{L}_3$ such that $\mathbf{x}_1 \oplus \mathbf{x}_2 \oplus \mathbf{x}_3 \in \mathcal{L}_4$
1: Compute the list $\mathcal{L}_1 \bowtie_{n/3} \mathcal{L}_2$.
2: Initialise a hash table T containing the $\mathbf{x}_1 \oplus \mathbf{x}_2$ for all $(\mathbf{x}_1, \mathbf{x}_2) \in \mathcal{L}_1 \bowtie_{n/3} \mathcal{L}_2$
3: Find a couple $(\mathbf{x}_3, \mathbf{x}_4) \in \mathcal{L}_3 \bowtie_{n/3} \mathcal{L}_4$ such that $\mathbf{x}_3 \oplus \mathbf{x}_4 \in T$.
4: Set $\mathbf{y} \leftarrow \mathbf{x}_3 \oplus \mathbf{x}_4$, and set $i \leftarrow 0$
5: **do**
6: Set $(\mathbf{x}_1, \mathbf{x}_2) \leftarrow (\mathcal{L}_1 \bowtie_{n/3} \mathcal{L}_2)[i]$
7: $i \leftarrow i + 1$
8: **while** $\mathbf{x}_1 \oplus \mathbf{x}_2 \neq \mathbf{y}$
9: **return** $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$

Complexity analysis. The time and space complexity of Algorithm 17 is $\mathcal{O}\left(2^{n/3}\right)$. Indeed, as discussed in Section 6.3, creating the join list $\mathcal{L}_1 \bowtie_{n/3} \mathcal{L}_2$ can be done in time $\mathsf{T} = \mathcal{O}(N_1 + N_2 + N_{12})$, and in space $\mathsf{M} = \mathcal{O}\left(\sqrt{N_1} + \sqrt{N_2}\right)$ using Algorithm 14. Initialising a hash table with the elements of $\mathcal{L}_1 \bowtie_{n/3} \mathcal{L}_2$ can then be done in time and space $\mathcal{O}(N_{12})$. Step 3 can be done in time $\mathsf{T} = \mathcal{O}(N_3 + N_4 + N_{34} + N_{12})$ and space $\mathsf{M} = \mathcal{O}\left(\sqrt{N_3} + \sqrt{N_4}\right)$ using Algorithm 15. Recovering the couple $(\mathbf{x}_1, \mathbf{x}_2)$ of $\mathcal{L}_1 \bowtie_{n/3} \mathcal{L}_2$ such that $\mathbf{x}_1 \oplus \mathbf{x}_2 = \mathbf{x}_3 \oplus \mathbf{x}_4$ will require at most to scan through the entire list $\mathcal{L}_1 \bowtie_{n/3} \mathcal{L}_2$.

All in all, this results in a time complexity of

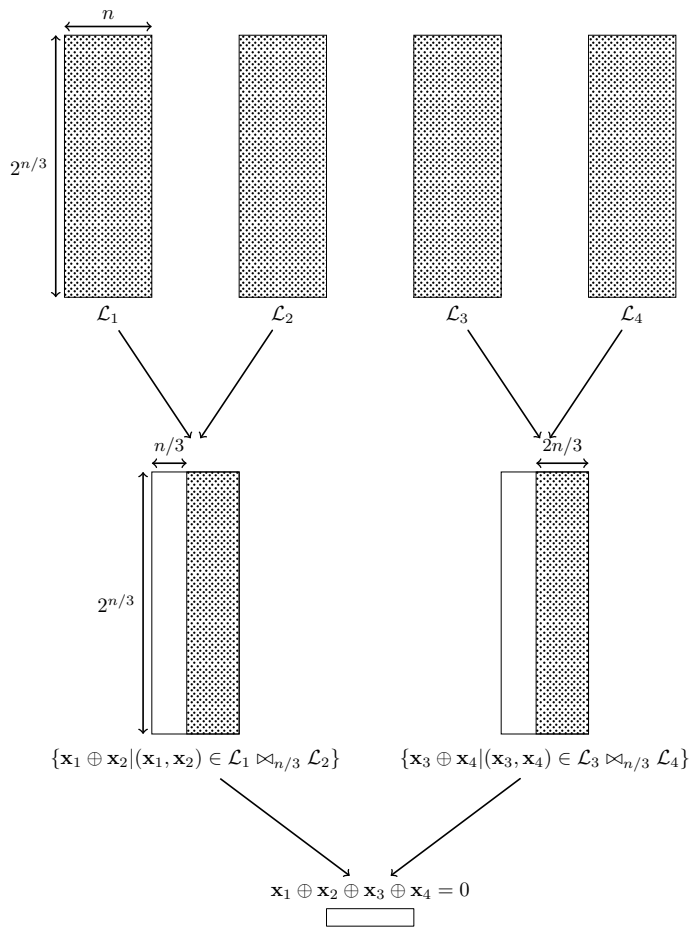


Figure 7.2 – Illustration of Wagner 4-tree Algorithm.

$$\mathsf{T} = \mathcal{O}\left(N_1 + N_2 + N_3 + N_4 + N_{12} + N_{34}\right) = \mathcal{O}\left(2^{n/3}\right),$$

and a space complexity of

$$\mathsf{M} = \mathcal{O}\left(N_{12} + \sqrt{N_1} + \sqrt{N_2} + \sqrt{N_3} + \sqrt{N_4}\right) = \mathcal{O}\left(2^{n/3}\right).$$

Remark 2. This algorithm can easily be generalised to solve the kXOR problem when k is a power of two. Indeed, the procedure has just to be reiterated a certain number of times. The algorithm would then work with a time and space complexity of $\mathcal{O}\left(k2^{n/(\log k+1)}\right)$, and require input lists of size $2n/(\log k + 1)$.

7.2.2 The 3XOR Case

We are more concerned with the 3XOR problem. In this case, the idea behind Wagner's algorithm does not pay off, and does not yield anything better than the quadratic algorithm. Indeed, let us consider three lists $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$ each containing N strings of length n. We choose an arbitrary ℓ-bit prefix $\mathbf{p}$, and we target a special solution $(\mathbf{a}, \mathbf{b}, \mathbf{c}) \in \mathcal{A} \times \mathcal{B} \times \mathcal{C}$ such that $\mathbf{c}[..\ell] = \mathbf{p}$.

Following the same idea as previously, it is possible to build the join list $\mathcal{A} \bowtie_\ell \mathcal{B}$. If we denote by N_{join} the number of elements of this list, we will have:

$$\mathbb{E}[N_{join}] = \frac{N^2}{2^\ell}.$$

If we actually want N_{join} to be equal to N, so that the computation of the join list will require exactly $\mathcal{O}(N)$ operations, we will have:

$$N = 2^\ell.$$

Furthermore, the expected number $N^{\mathbf{p}}$ of vectors $\mathbf{c}$ of $\mathcal{C}$ such that $\mathbf{c}[..\ell] = \mathbf{p}$ is:

$$\mathbb{E}[N^{\mathbf{p}}] = \frac{N}{2^\ell} = \frac{2^\ell}{2^\ell} = 1.$$

Let us call $\mathbf{c}$ this single element. From here $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ – with $(\mathbf{a}, \mathbf{b}) \in \mathcal{A} \bowtie_\ell \mathcal{B}$ – is a solution to the 3XOR problem, if $\mathbf{a} \oplus \mathbf{b}$ and $\mathbf{c}$ agree on their $(n-\ell)$ last bits. The expected number of couples of $\mathcal{A} \bowtie_\ell \mathcal{B}$ that actually satisfies this condition is:

$$\mathbb{E}[N_{sol}] = \frac{2^\ell}{2^{n-\ell}},$$

and thus, there will be a solution with high probability, as long as $\ell = n - \ell$. It follows that $\ell = 1/2$.

In a nutshell, Wagner's algorithm applied to the 3XOR problem requires as input three lists $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$ of size $2^{n/2}$, and search for a pair of elements $(\mathbf{a}, \mathbf{b}) \in \mathcal{A} \oplus \mathcal{B}$ such that $\mathbf{a} \oplus \mathbf{b}$ matches a single specific target in $\mathcal{C}$. This is exactly the same as the naive birthday search with two lists.

Remark 3. We do not require $N_{\mathcal{C}}$ to be $2^{n/2}$. In fact, we can consider a list $\mathcal{C}$ reduced to only one element $\mathbf{c}$ from the beginning of the procedure. In this case the arbitrary prefix $\mathbf{p}$ is $\mathbf{c}[..n/2]$.

Improving Wagner's idea. Although Wagner's method does not seem to work as it is for the case of the 3XOR, it has known improvement in the following years. Indeed, let us denote by $N_{\mathcal{C}}^{\mathbf{p}}$ the size of $\mathcal{C}^{[\mathbf{p}]}$, and assume that $N_{\mathcal{A}} = N_{\mathcal{B}} = 2^\ell$ so that the expected size of $\mathcal{A} \bowtie_{\ell,\mathbf{p}} \mathcal{B}$ will also be 2^ℓ, then the expected number of couples $(\mathbf{a}, \mathbf{b})$ from $\mathcal{A} \bowtie_{\ell,\mathbf{p}} \mathcal{B}$ that satisfy $\mathbf{a} \oplus \mathbf{b} = \mathbf{c}$ for some $\mathbf{c}$ in $\mathcal{C}^{[\mathbf{p}]}$ is:

$$\mathbb{E}[N_{sol}] = \frac{2^\ell N_{\mathcal{C}}^{\mathbf{p}}}{2^{n-\ell}}.$$

If we actually want to have only one solution with high probability, ℓ must satisfy:

$$\ell + \log(N_{\mathcal{C}}^{\mathbf{p}}) = n - \ell. \tag{7.1}$$

Then, choosing $N_{\mathcal{A}} = N_{\mathcal{B}} = 2^{n/2}/\sqrt{N_{\mathcal{C}}^{\mathbf{p}}}$, one can find a solution to the 3XOR problem with input lists $\mathcal{A}, \mathcal{B}, \mathcal{C}^{[\mathbf{p}]}$ in time and space:

$$\mathsf{T} = \mathsf{M} = \mathcal{O}\left(\frac{2^{n/2}}{\sqrt{N_{\mathcal{C}}^{\mathbf{p}}}}\right), \tag{7.2}$$

using the following, simple procedure:

1. Set $\mathbf{y}$ to be a vector of $\mathbb{F}_2^n$ such that $\mathbf{y}[..\ell] = \mathbf{p}$ and set $\mathcal{C}^{[\mathbf{p}]} \leftarrow \mathcal{C}^{[\mathbf{p}]} \oplus \mathbf{y}$ and $\mathcal{A} \leftarrow \mathcal{A} \oplus \mathbf{y}$
2. Initialise a hash table T with the entries of $\mathcal{C}^{[\mathbf{p}]}$
3. Search a couple $(\mathbf{a}, \mathbf{b}) \in \mathcal{A} \bowtie_\ell \mathcal{B}$ such that $\mathbf{a} \oplus \mathbf{b} \in T$ using Algorithm 15.

Remark 4. We can assume without loss of generality that the prefix $\mathbf{p}$ is 0. Indeed, for all $\mathbf{y}$ of $\mathbb{F}_2^n$, if $\mathbf{a} \oplus \mathbf{b} \oplus \mathbf{c} = 0$ then $(\mathbf{a} \oplus \mathbf{y}) \oplus \mathbf{b} \oplus (\mathbf{c} \oplus \mathbf{y}) = 0$. From here, if $\mathbf{p}$ is not 0, we can find a vector $\mathbf{y}$ such that $\mathbf{y}[..\ell] = \mathbf{p}$ and replace the elements $\mathbf{c}$ of $\mathcal{C}^{[\mathbf{p}]}$ by $\mathbf{c} \oplus \mathbf{y}$, and the elements $\mathbf{a}$ of $\mathcal{A}$ by $\mathbf{a} \oplus \mathbf{y}$ without changing the set of solutions. That is exactly what is done in step 1 of the method described above. This can be done in time $\mathcal{O}\left(N_{\mathcal{A}} + N_{\mathcal{C}}^{\mathbf{p}}\right)$ without extra space.

The difficult part remains to compute the list $\mathcal{C}^{[\mathbf{p}]}$ in time and space that is proportional to $2^{n/2}/\sqrt{N_{\mathcal{C}}^{\mathbf{p}}}$, so that the time and space complexity claimed in Equation 7.2 will still hold. Two methods that actually do this job were independently proposed by Joux [Jou09] on one hand, and Nikolić and Sasaki [NS15] on the other hand. We will recall these two methods in the two following sections.

7.3 Nikolić and Sasaki's Algorithm

7.3.1 Description of the Procedure

The high-level idea of Nikolić and Sasaki's improvement of Wagner's algorithm is the following. They start with three lists $\mathcal{A}, \mathcal{B}, \mathcal{C}$ such that $N_{\mathcal{A}} = N_{\mathcal{B}} = N_{\mathcal{C}} = 2^{\ell}$, with ℓ being a little less than $n/2$. Then they search for the most frequent ℓ-bit prefix $\mathbf{p}$ in $\mathcal{C}$, and they consider the sublist $\mathcal{C}^{[\mathbf{p}]}$ of $\mathcal{C}$, of the elements $\mathbf{c}$ such that $\mathbf{c}[..\ell] = \mathbf{p}$ (the other elements of $\mathcal{C}$ are discarded). Then, for every pair $(\mathbf{a}, \mathbf{b})$ of $\mathcal{A} \bowtie_{\ell,\mathbf{p}} \mathcal{B}$ they test whether $\mathbf{a} \oplus \mathbf{b}$ is in $\mathcal{C}^{[\mathbf{p}]}$ using the method described previously. A description of this method is given by Algorithm 18. We also summarise the whole idea in Figure 7.3.

Algorithm 18 Nikolić and Sasaki's Algorithm.

Require: Lists $\mathcal{A}, \mathcal{B}, \mathcal{C}$ of size 2^{ℓ}
Ensure: A couple $(\mathbf{a}, \mathbf{b}) \in \mathcal{A} \times \mathcal{B}$ such that $\mathbf{a} \oplus \mathbf{b} \in \mathcal{C}$
1: Initialise a table T_C of size 2^{ℓ} with zeroes
2: Set $\mathbf{p}_{max} \leftarrow 0$
3: **for** $0 \leq i < 2^{\ell}$ **do**
4: $\quad \mathbf{c} \leftarrow \mathcal{C}[i]$
5: $\quad T_C[\mathbf{c}[..\ell]] \leftarrow T_C[\mathbf{c}[..\ell]] + 1$
6: $\quad$ **if** $T_C[\mathbf{c}[..\ell]] > T_C[\mathbf{p}_{max}]$ **then**
7: $\quad\quad$ Set $\mathbf{p}_{max} \leftarrow \mathbf{c}[..\ell]$
8: Set $\mathbf{y} \leftarrow (\mathbf{p}_{max}|0)$ such that $\mathbf{y}$ is of length n
9: Init a hash table T with $\mathbf{c} \oplus \mathbf{y}$ for all $\mathbf{c}$ of $\mathcal{C}^{[\mathbf{p}_{max}]}$
10: **for** $0 \leq i < 2^{\ell}$ **do**
11: $\quad \mathcal{A}[i] \leftarrow \mathcal{A}[i] \oplus \mathbf{y}$
12: Find a couple $(\mathbf{a}, \mathbf{b}) \in \mathcal{A} \bowtie_{n/2} \mathcal{B}$ such that $\mathbf{a} \oplus \mathbf{b}$ is in T
13: **return** $(\mathbf{a} \oplus \mathbf{y}, \mathbf{b})$

Remark 1. There is no need to actually store $\mathcal{A} \bowtie_{\ell,\mathbf{p}} \mathcal{B}$. We can check if a couple $(\mathbf{a}, \mathbf{b})$ of $\mathcal{A} \bowtie_{\ell,\mathbf{p}} \mathcal{B}$ is in $\mathcal{C}$ as the process goes using Algorithm 15.

7.3.2 Choice of the parameters and complexity.

According to Mitzenmacher's thesis [Mit96], if we consider lists of 2^{ℓ} uniformly random elements, the size of the sublist $\mathcal{C}^{[\mathbf{p}]}$, where $\mathbf{p}$ is the most frequent ℓ-bit prefix in $\mathcal{C}$, will be $N_{\mathcal{C}}^{\mathbf{p}} = \Theta(\ell/\ln(\ell))$. Indeed, if we see our vectors as "balls" and all the possible prefixes they can take as "bins", we are searching for the number of balls in the bin that contains the maximum load (see section 6.1.2). From Theorem 6.2, we obtain $N_{\mathcal{C}}^{\mathbf{p}} = \Theta(\ell/\ln(\ell))$.

According to Equation 7.1, a solution will be found with high probability as long as:

$$\ell = \frac{1}{2}(n - \log(N_{\mathcal{C}}^{\mathbf{p}})). \tag{7.3}$$

Considering the actual expected value of $N_{\mathcal{C}}^{\mathbf{p}}$, we have:

$$\ell = \frac{n}{2} - \varepsilon(\ell),$$

where $\varepsilon(\ell) \simeq \log(\ell) - \log(\ln(\ell))$. Thus

$$\ell \simeq \frac{n}{2} - \log(n/2) + \log(\ln(n/2)),$$

and it follows that the expected time space and queries complexity of the procedure is

$$\mathsf{T} = \mathsf{M} = \mathsf{Q} = \mathcal{O}\left(\frac{2^{n/2}}{\sqrt{n/\ln(n)}}\right). \tag{7.4}$$

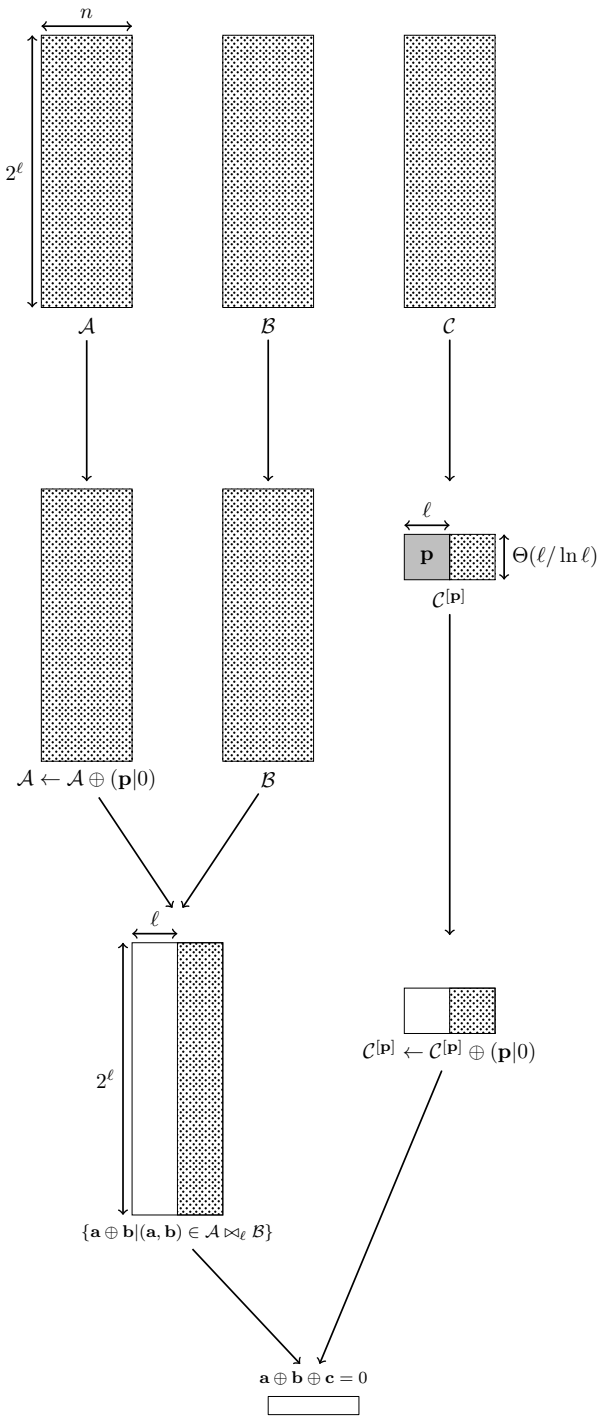


Figure 7.3 – Illustration of Nikolić and Sasaki's Algorithm.

Nikolić and Sasaki thus obtain a $\sqrt{n/\ln(n)}$ speedup compared to Wagner and the Quadratic Algorithm.

7.4 Joux's Algorithm

Before Nikolić and Sasaki, Joux had already proposed a better improvement of Wagner's algorithm [Jou09]. He considers three lists, $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$ such that $N_{\mathcal{A}} = N_{\mathcal{B}} = 2^{n/2}/\sqrt{n/2}$, and $N_{\mathcal{C}} = n/2$. The list $\mathcal{C}$ is then much smaller than the two other lists. Joux then proposes to find an invertible matrix M, and to compute the lists $\mathcal{A}' = \mathcal{A}M$, $\mathcal{B}' = \mathcal{B}M$, and $\mathcal{C}' = \mathcal{C}M$, such that all vectors $\mathbf{c}$ from $\mathcal{C}'$ have the same $n/2$-bit prefix $\mathbf{c}[..n/2] = 0$.

If M is an invertible matrix, then finding a 3XOR with input lists $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$, is equivalent to finding a 3XOR with input lists $\mathcal{A}'$, $\mathcal{B}'$ and $\mathcal{C}'$. Indeed, if $\mathbf{a} \oplus \mathbf{b} \oplus \mathbf{c} = 0$, for some $(\mathbf{a}, \mathbf{b}, \mathbf{c}) \in \mathcal{A} \times \mathcal{B} \times \mathcal{C}$, then $\mathbf{a}M \oplus \mathbf{b}M \oplus \mathbf{c}M = 0$, and reciprocally, if $\mathbf{a}' \oplus \mathbf{b}' \oplus \mathbf{c}' = 0$, for some $(\mathbf{a}', \mathbf{b}', \mathbf{c}')$ then $\mathbf{a}'M^{-1} \oplus \mathbf{b}'M^{-1} \oplus \mathbf{c}'M^{-1} = 0$.

Algorithm 19 Joux's Algorithm.

Require: Lists $\mathcal{A}, \mathcal{B}$ of size $2^{n/2}/\sqrt{n/2}$ and $\mathcal{C}$ of size $n/2$.
Ensure: A couple $(\mathbf{a}, \mathbf{b}) \in \mathcal{A} \times \mathcal{B}$ such that $\mathbf{a} \oplus \mathbf{b} \in \mathcal{C}$
1: Compute an n-by-n matrix M, such that $\forall \mathbf{c}' = \mathbf{c}M$ with $\mathbf{c} \in \mathcal{C}$, $\mathbf{c}'[..\ell] = 0$.
2: Set $\mathcal{A} \leftarrow \mathcal{A}M$, $\mathcal{B} \leftarrow \mathcal{B}M$ and $\mathcal{C} \leftarrow \mathcal{C}M$
3: Init a hash table T with the elements of $\mathcal{C}$
4: Find a couple $(\mathbf{a}, \mathbf{b}) \in \mathcal{A} \bowtie_{n/2} \mathcal{B}$ such that $\mathbf{a} \oplus \mathbf{b}$ is in T
5: **return** $(\mathbf{a}M^{-1}, \mathbf{b}M^{-1})$

Joux's method is detailed in Algorithm 19. We also illustrate it in Figure 7.4.

7.4.1 Finding the matrix M

We give the following more general result, which will also be useful in Chapter 8:

Let ℓ be an integer such that $1 \leq \ell \leq n/2$. Let C be any $(n-\ell)$-by-n matrix. Then, C can be decomposed using the PLUQ factorisation into: $C = PLUQ$, where P and Q are permutation matrices, respectively of size $(n-\ell)$-by-$(n-\ell)$ and n-by-n, L is lower-trapezoidal (i.e. a non necessary square matrix whose coefficients above the main diagonal are zeroes) with non-zero diagonal and U is upper-trapezoidal with non-zero diagonal. L is a $(n-\ell)$-by-r matrix and U is a r-by-n matrix, where r is the rank of C. We denote U_0 the r-by-r triangular sub-matrix U, and U_1 the sub-matrix of U such that $U = (U_0|U_1)$.

Proposition 7.1. *Let M be the n-by-n matrix such that:*

$$M^{-1} = \begin{pmatrix} 0 & I_{n-r} \\ U_0 & U_1 \end{pmatrix} \cdot Q.$$

M is, indeed, invertible, and if we denote by X the $(n-\ell)$-by-r matrix $X = PL$, we have:

$$CM = (0|X).$$

The proof of this statement is trivial, but we still give it for completeness.

Proof. It is easy to see that M is indeed invertible. In fact, if we denote by $\bar{M}$ the matrix:

$$\bar{M} = \begin{pmatrix} 0 & I_{n-r} \\ U_0 & U_1 \end{pmatrix},$$

as defined in Proposition 7.1, if we permute the first $n-r$ rows of $\bar{M}$ and the last r ones, we have a triangular matrix, with a non-zero diagonal. Q is a permutation matrix so, it is invertible. Then the product $\bar{M}Q$ is invertible.

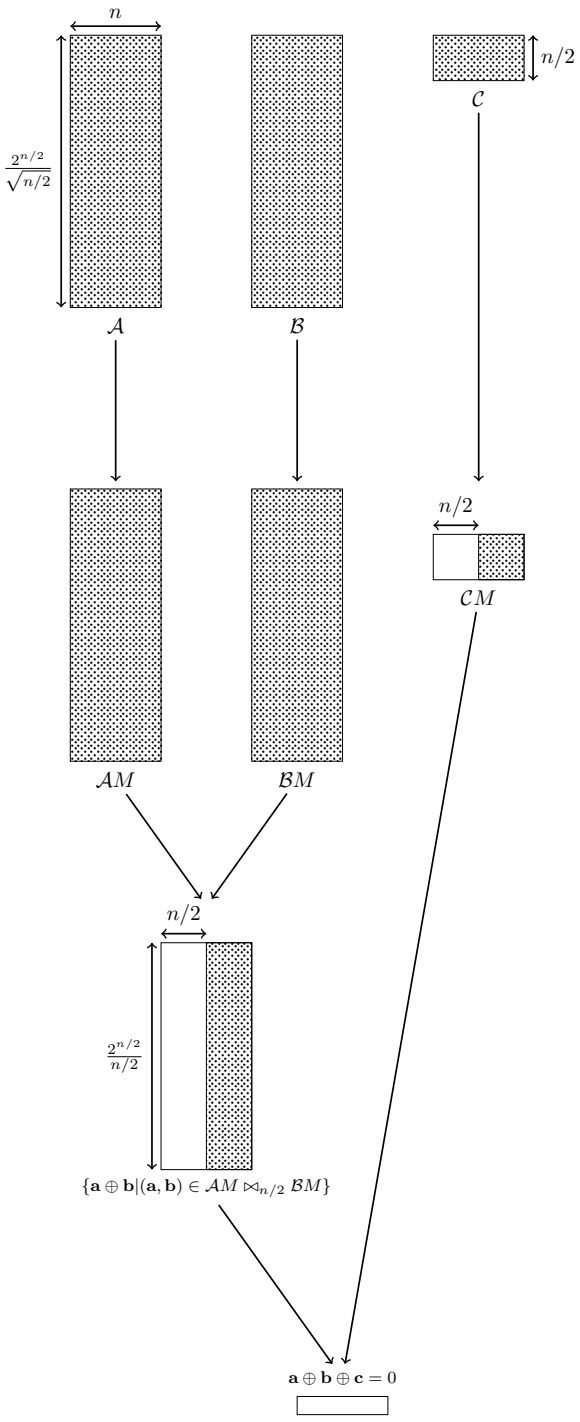


Figure 7.4 – Illustration of Joux's Algorithm.

Moreover, let X be a $(n-\ell)$-by-r matrix. We have the following relation:

$$\begin{pmatrix} 0 & X \end{pmatrix} \cdot \begin{pmatrix} 0 & I_{n-r} \\ U_0 & U_1 \end{pmatrix} = \begin{pmatrix} XU_0 & XU_1 \end{pmatrix} = XU.$$

We can set X to be equal to PL. Indeed, P is an $(n-\ell)$-by-$(n-\ell)$ matrix, and L is an $(n-\ell)$-by-r matrix, so their product will be an $(n-\ell)$-by-r matrix. In this case

$$\begin{pmatrix} 0 & X \end{pmatrix} \cdot \bar{M} = PLU = CQ^{-1},$$

and then

$$\begin{pmatrix} 0 & X \end{pmatrix} \cdot \bar{M} = C.$$

Choosing $M = Q^{-1}\bar{M}^{-1}$ and reversing the equation above concludes the proof.

□

Here, we are concerned with the case $\ell = n/2$. This result tells us that we can always find a matrix M, such that for each vector $\mathbf{c}$ from $\mathcal{C}$, $(\mathbf{c}M)[..n-r] = 0$, where r is the size of any linearly independent subset of $\mathcal{C}$ that spans $\mathsf{Span}\{\mathcal{C}\}$). As the vectors from $\mathcal{C}$ are uniformly random, it is more likely that $r = N_{\mathcal{C}} = n/2$. However if by accident $r < n/2$, then $\mathbf{c}' = \mathbf{c}M$ will start with more than $n/2$ zeroes, and thus $\mathbf{c}'[..n/2] = 0$.

Remark 1. In [Jou09], Joux stated that, as the vectors of $\mathcal{C}$ are likely to be linearly independent, there exists a basis $\mathscr{B}$ of $\mathbb{F}_2^n$, such that the last $n/2$ vectors of $\mathscr{B}$ are the vectors of $\mathcal{C}$. M can then be chosen as the matrix that transforms each vector of $\mathbb{F}_2^n$ to the basis $\mathscr{B}$. Then, for all $\mathbf{c} \in \mathcal{C}$, $\mathbf{c}M$ will start with $n/2$ zeroes.

7.4.2 Complexity analysis.

Finding the matrix M can be done in time $\mathcal{O}\left(n^3\right)$ by computing a naive PLUQ factorisation over $\mathbb{F}_2$. Then, using the trick described in section 6.3.3, it is possible to perform the basis change in time and space $\mathcal{O}\left(2^{n/2}/\sqrt{n/2}\right)$. Solving the 3XOR instance after that can be done in time and space $\mathcal{O}\left(2^{n/2}/\sqrt{n}\right)$ as discussed in Section 7.2.2.

In this case, the expected size of the join list is:

$$\mathbb{E}[N_{join}] = \frac{2^n}{(n/2)\cdot 2^{n/2}} = \frac{2^{n/2}}{n/2},$$

which is a bit smaller than $N_{\mathcal{A}} = N_{\mathcal{B}} = 2^{n/2}/\sqrt{n/2}$.

Remark 2. One can think of improving Joux's method choosing more balanced parameters. Indeed, we could decide to set $N_{\mathcal{A}} = N_{\mathcal{B}} = 2^\ell$, and $N_{\mathcal{C}} = n-\ell$, and search for a matrix M such that the first ℓ bits of each vectors of $\mathcal{C}$ are zeroes. The expected size of the joined list $\mathcal{A} \bowtie_\ell \mathcal{B}$ would then be 2^ℓ, and following Equation 7.1, if:

$$\ell = \frac{n}{2} - \frac{1}{2}\log(n-\ell),$$

there will be a solution with high probability, and the time and space complexity of the procedure would be $\mathcal{O}\left(2^\ell\right)$. The speedup obtained compared to Joux's algorithm, however, does not seem to be significant. Indeed, The first order approximation of ℓ gives us:

$$\ell \simeq \frac{n}{2} - \frac{\log(n/2)}{2},$$

and thus, $2^\ell \simeq 2^{n/2}/\sqrt{n/2}$ which is exactly the size of $\mathcal{A}$ and $\mathcal{B}$.

7.5 Time and Memory Tradeoffs

7.5.1 Discussion

Wagner's Algorithm and the ones inspired by it used the same kind of tradeoff: increasing both queries and memory complexity in order to reduce the time complexity. We are tempted to say that Joux's Algorithm is the "best" of the algorithms described, as it has the best time complexity. However, the huge amount of data that is required quickly becomes a bottleneck, as explained in the concrete example below.

Example 7.1 (Computation of a 96-bit 3XOR). We are given three oracles **A**, **B** and **C** that, upon queries, return 96-bit uniformly random outputs each. We aim to find a 3XOR between these outputs.

On a standard 64-bit computer, Joux's algorithm requires around $2^{45.2}$ operations, but just storing the lists will already require 1.3 petabyte of memory. This makes this procedure hardly practical.

On the other hand, if we choose to reduce the size of the lists to $2^{96/3} = 2^{32}$ elements each, and to utilise the quadratic algorithm to solve the problem, then the number of operations required will be about 2^{64}, but storing the list will only require about 206 gigabyte of memory, which is already easier to handle.

In short, if we want to solve a 3XOR in practice, it is important to keep the lists small, as the memory will quickly become the limiting factor.

To deal with this issue, Bernstein proposed in [Ber07] a tradeoff which was afterward called "clamping" [BLN+09].

Remark 1. In [NS15], Nikolić and Sasaki also proposed a tradeoff to reduce the memory of the kXOR problem, where k is a power of two. As their method does not immediately apply to the case of the 3XOR, we do not recall it here.

7.5.2 Bernstein's Clamping Trick

The idea, which may at first seem counterintuitive, is that the amount of data can be reduced if more queries than strictly required are allowed. Assume that $2^{\alpha n}$ (resp. $2^{\beta n}$, $2^{\gamma n}$) queries to oracles **A** (resp. **B**, **C**), and that $\alpha + \beta + \gamma \geq 1$. Let us denote by κ the parameter:

$$\kappa = (\alpha + \beta + \gamma - 1)/2. \tag{7.5}$$

When querying the oracles, it is possible to immediately reject the outputs that are not zero on the first κn bits. Naturally, this only works when $\kappa \geq \min(\alpha, \beta, \gamma)$. Doing this "clamping", one comes up with three lists $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$ of $(1-\kappa)n$-bit entries.

Furthermore, from Equation 6.4, the expected size $N_{\mathcal{A}}$ of the list $\mathcal{A}$ satisfies:

$$\mathbb{E}[N_{\mathcal{A}}] = \frac{2^{\alpha n}}{2^{\kappa n}} = 2^{(\alpha-\kappa)n}.$$

Th same goes for the expected size $\mathbb{E}[N_{\mathcal{B}}] = 2^{(\beta-\kappa)n}$ of $\mathcal{B}$ and $\mathbb{E}[N_{\mathcal{C}}] = 2^{(\gamma-\kappa)n}$ of $\mathcal{C}$. Then, we have:

$$\log(N_{\mathcal{A}} N_{\mathcal{B}} N_{\mathcal{C}}) = n(\alpha + \beta + \gamma - 3 \cdot \kappa).$$

With the definition of κ from above, this leads to:

$$\log(N_{\mathcal{A}} N_{\mathcal{B}} N_{\mathcal{C}}) = n(3/2 - 1/2 \cdot (\alpha + \beta + \gamma)) = n(1 - (\alpha + \beta + \gamma - 1)/2),$$

Then, the product $N_{\mathcal{A}} N_{\mathcal{B}} N_{\mathcal{C}}$ is equal to $2^{(1-\kappa)n}$. Hence, this trick allows to reduce the size of the instance to $(1-\kappa)n$ instead of n.

Same number of queries to each oracle. In the particular case where each oracle is queried the same number of times, say $2^{\lambda n}$, then κ satisfies:

$$\frac{(1-\kappa)}{3}n = (\lambda - \kappa)n. \tag{7.6}$$

Now, denoting n' the quantity $(1-\kappa)n$, it is clear that performing the clamping over κn bits leads to three lists $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$ of N vectors of length n', with:

$$\mathbb{E}[N] = 2^{(\lambda-\kappa)n} = 2^{n'/3}.$$

For instance, assume that $2^{n/2}$ queries to each of the oracles are allowed. We have $\kappa = 1/4$, and then, solving the 3XOR problem, can be done in $\mathcal{O}\left(2^{n/2}\right)$ operations using only $\mathcal{O}\left(2^{n/4}\right)$ words of memory. In fact, the three lists we obtain contain roughly $2^{n/4}$ entries of $n' = 3 \cdot n/4$ bits. A solution to the problem can then be found by any algorithm that is able to recover a solution to the problem over n' bits when the size of the three input list is $2^{n'/3}$. Using the quadratic algorithm described above, this would require $\mathcal{O}\left(2^{2 \cdot n'/3}\right) = \mathcal{O}\left(2^{n/2}\right)$ operations.

Minimising the product TM. When it comes to time and memory tradeoff, one way to quantify the actual cost of an algorithm is the product TC where C is the cost (for instance in Euros) of a machine. For simplicity, we reduce the price of a machine to its amount of available memory. If someone can afford to pay for M' in memory, and spend T' running the algorithm, he can run $\mathsf{MT}/\mathsf{M}'\mathsf{T}'$ separate computation, assuming that M' divide M and T divide T'. Indeed, he can get access to M/M' processors to run T/T' sequential computations of the algorithm. It is clear now, that it is worthwhile to keep the product TM low.

We claim that if querying the oracles can be done in constant time, and if we denote by $\mathsf{T}_{\mathsf{tot}}$ the time complexity required to create the lists and to process them, then, in this setting, making $2^{n/2}$ queries to the oracles, and performing the clamping on $\kappa n = n/4$ bits actually minimises the product $\mathsf{T}_{\mathsf{tot}}\mathsf{M}$.

Indeed, if we assume that the oracles can be queried in constant time, then the time required to create the lists and to process them using the quadratic algorithm will be:

$$\mathsf{T}_{\mathsf{tot}} = \mathcal{O}\left(2^{\lambda n} + 2^{2(\lambda-\kappa)n}\right). \tag{7.7}$$

With κ defined as in Equation 7.5, we actually have:

$$\lambda - \kappa = \lambda - \frac{3\lambda - 1}{2} = \frac{1-\lambda}{2}.$$

Then, Equation 7.7 becomes:

$$\mathsf{T}_{\mathsf{tot}} = \mathcal{O}\left(2^{\lambda n} + 2^{(1-\lambda)n}\right). \tag{7.8}$$

If $\lambda < 1/2$ then it is the processing part that dominates procedure. On the over hand, when $\lambda > 1/2$, the creation of the lists dominates the whole procedure. We reach the equilibrium when $\lambda = 1/2$. The memory required is

$$\mathsf{M} = \mathcal{O}\left(2^{(1-\lambda)n/2}\right). \tag{7.9}$$

Let $\lambda \geq 1/2$. In this case the creation of the lists dominates the procedure and we have:

$$\mathsf{T}_{\mathsf{tot}}\mathsf{M} = \mathcal{O}\left(2^{\lambda n} 2^{(1-\lambda)n/2}\right)$$

Taking the log and forgetting the constants hidden in the $\mathcal{O}$ notation, this gives:

$$\begin{aligned}\log(\mathsf{T_{tot}M}) &= \lambda n + (1-\lambda)n/2 \\ &= \frac{(2\lambda+1-\lambda)}{2}n \\ &= \frac{\lambda+1}{2}n \end{aligned} \tag{7.10}$$

And then:

$$\mathsf{T_{tot}M} = \mathcal{O}\left(2^{(\lambda+1)n/2}\right). \tag{7.11}$$

This reaches its minimum $2^{3n/4}$ when $\lambda = 1/2$.

Consider now the case $\lambda \leq 1/2$. Processing the lists dominates the procedure and we have:

$$\begin{aligned}\mathsf{T_{tot}M} &= \mathcal{O}\left(2^{(1-\lambda)n}2^{(1-\lambda)n/2}\right) \\ &= \mathcal{O}\left(2^{3(1-\lambda)n/2}\right) \end{aligned} \tag{7.12}$$

This reaches its minimum $2^{3n/4}$ when $\lambda = 1/2$, which actually proves our claim.

For completeness, we give the Time-Memory product of Joux and Nikolić-Sasaki's methods:

$$(\mathsf{T_{tot}M})_{Joux} = \mathcal{O}\left(\frac{2^n}{n}\right), \qquad (\mathsf{T_{tot}M})_{NS} = \mathcal{O}\left(\frac{2^n \ln n}{n}\right).$$

Remark 2. This is assuming that querying the oracles can be done in constant time. However, as we will discuss in section 8.4.1, sometimes querying the oracles can take a lot of time. In this case it can be interesting to perform the clamping on less bits, even if it means processing larger lists.

Chapter 8

Of a New 3XOR by Linear Change of Variable Algorithm

In this chapter, we propose a new algorithm which is asymptotically n times faster than the quadratic algorithm. It is a generalisation of Joux's Algorithm. This new algorithm is faster in practice than the quadratic algorithm for relevant parameter sizes. We also present some constant time improvements, and give some implementation details along with a 3XOR of SHA256. This work is the main contribution of a paper [BDF18] accepted at ToSC/FSE 2018. This is joint work with Charles Bouillaguet and Pierre-Alain Fouque.

8.1 A New Algorithm

Similar to Joux's algorithm, we exploit the fact that when M is an $n \times n$ invertible matrix over $\mathbb{F}_2$, then $\mathbf{a} \oplus \mathbf{b} \oplus \mathbf{c} = 0$ if and only if $\mathbf{a}M \oplus \mathbf{b}M \oplus \mathbf{c}M = 0$. The sets of solutions in $\mathcal{A} \times \mathcal{B} \times \mathcal{C}$ and $\mathcal{A}M \times \mathcal{B}M \times \mathcal{C}M$ are the same.

8.1.1 Algorithm Description

The idea of our algorithm is to select an arbitrary slice of $n - k$ vectors from $\mathcal{C}$ and to choose a matrix M such that the first k entries of the slice become zero. The Algorithm then finds all the triplets $\mathbf{a} \oplus \mathbf{b} \oplus \mathbf{c} = 0$ where $\mathbf{c}$ belongs to the slice. Repeating this procedure for all slices yields all the possible solutions. This procedure is fully described in Algorithm 20. We illustrate it in Figure 8.1.

Algorithm 20 3XOR by linear change of variables.

Require: Lists $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$ of respective size $N_{\mathcal{A}}$, $N_{\mathcal{B}}$ and $N_{\mathcal{C}}$.
Ensure: All $(\mathbf{a}, \mathbf{b}) \in \mathcal{A} \times \mathcal{B}$ such that $\mathbf{a} \oplus \mathbf{b} \in \mathcal{C}$

1: Set $\ell \leftarrow \lceil \log \min(N_{\mathcal{A}}, N_{\mathcal{B}}) \rceil$.
2: $Res \leftarrow \perp$
3: **for** $0 \leq i < N_{\mathcal{C}}/(n - \ell)$ **do**
4: Set $j \leftarrow i(n - \ell)$ and $k \leftarrow \min((i + 1)(n - \ell), N_{\mathcal{C}})$
5: Compute an n-by-n matrix M, such that $\forall \mathbf{c}' = \mathbf{c}M$ with $\mathbf{c} \in \mathcal{C}[j..k]$, $\mathbf{c}'[..\ell] = 0$.
6: Set $\mathcal{A}' \leftarrow \mathcal{A}M$, $\mathcal{B}' \leftarrow \mathcal{B}M$ and $\mathcal{C}' \leftarrow \mathcal{C}[j..k]M$
7: Init a hash table T with the elements of $\mathcal{C}'$
8: Search couples $(\mathbf{a}', \mathbf{b}') \in \mathcal{A}' \bowtie_{\ell} \mathcal{B}'$ such that $\mathbf{a}' \oplus \mathbf{b}'$ is in T
9: **if** Such a pair $(\mathbf{a}', \mathbf{b}')$ is found **then**
10: $Res \leftarrow Res \cup \{(\mathbf{a}M^{-1}, \mathbf{b}M^{-1}\}$

Remark 1. It is possible to keep only one copy of each list and to perform the basis change in place. Instead of Step 6 we should do:

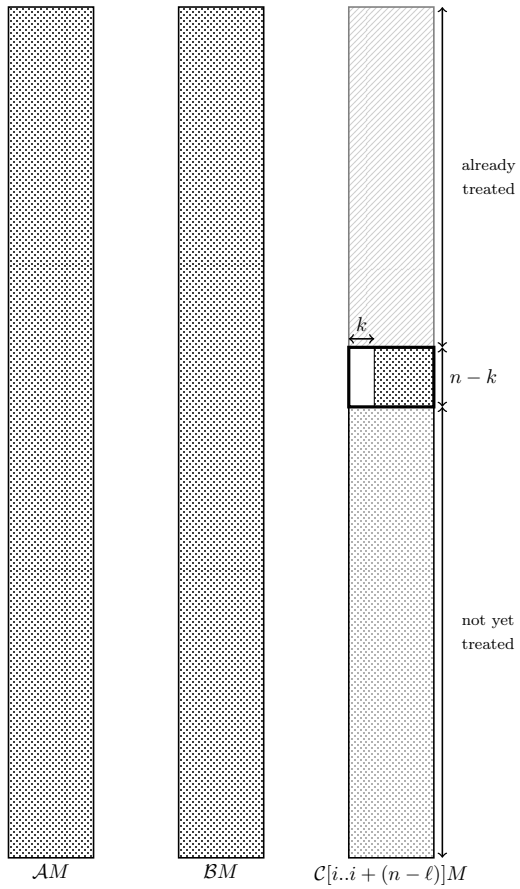


Figure 8.1 – Illustration of Algorithm 20.

Set $\mathcal{A} \leftarrow \mathcal{A}M_{prev}^{-1}M$ (and same for $\mathcal{B}$, $\mathcal{C}$)

Set $M_{prev}^{-1} \leftarrow M^{-1}$.

At the first iteration, M_{prev}^{-1} must be initialised to the identity matrix.

Remark 2. Note that this procedure returns all solutions to the 3XOR problem. The Nikolić-Sasaki algorithm, for instance returns only one among many.

8.1.2 Complexity Analysis

Theorem 8.1. *Algorithm 20 finds all the solutions to 3XOR problem with $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ in expected time:*

$$\mathsf{T} = \mathcal{O}\left((N_{\mathcal{A}} + N_{\mathcal{B}})\frac{N_{\mathcal{C}}}{n}\right), \tag{8.1}$$

and space M *linear in* $N_{\mathcal{A}} + N_{\mathcal{B}} + N_{\mathcal{C}}$.

Proof. We first observe that the algorithm only needs to store two copies of the input lists (This can be brought down to a single copy, updated in-place). This establishes the linear space claim.

We can assume without loss of generality that $N_{\mathcal{A}} \leq N_{\mathcal{B}}$. Following the same arguments as the ones detailed in Section 7.4, one iteration of the main loop takes on average $\mathcal{O}(N_{\mathcal{A}} + N_{\mathcal{B}})$ operations, and the extra memory required is still on average $\mathcal{O}(N_{\mathcal{A}} + N_{\mathcal{B}})$. In fact finding the basis change, and applying it requires $\mathcal{O}(|A| + |B|)$ operations, and additional memory that is lower than $c2^{\ell}$ for some constant c. The definition of ℓ ensures that the space complexity of this procedure is lower that $cN_{\mathcal{A}}$. Step 8 requires $\mathcal{O}\left(N_{\mathcal{A}} + N_{\mathcal{B}} + N_{\mathcal{A}}N_{\mathcal{B}}/2^{\ell}\right)$ operations. The choice of ℓ, once again, ensures that this is in fact $\mathcal{O}(N_{\mathcal{A}} + N_{\mathcal{B}})$.

These steps are repeated $N_{\mathcal{C}}/(n - \log(N_{\mathcal{A}}))$. The hypothesis that $\log(A) < (1-\varepsilon)n$ guarantees that the denominator is $\Omega(n)$, and yields the claimed complexity. □

Using this algorithm, it is beneficial to choose $\mathcal{C}$ as the smallest of the three lists.

8.2 Useful Tricks

8.2.1 Mixing our Algorithm and the Clamping Trick

As discussed in section 7.5.2, it is possible to reduce both the space (M) and time (T) complexity, if more queries (Q) are allowed.

We assume that $2^{\lambda n}$ queries to each oracles are allowed, then using the clamping trick, we can reduce the problem to the one finding a 3XOR over $(1-\kappa)n$ bits, with $\kappa = (3\lambda - 1)/2$. The size of the input lists will then be about $2^{(1-\lambda)n/2} = 2^{\lambda n/3}$. Processing them with Algorithm 20 will have a time complexity of

$$\mathsf{T} = \mathcal{O}\left(\frac{2^{(1-\lambda)n}}{n}\right), \tag{8.2}$$

and space complexity of

$$\mathsf{M} = \mathcal{O}\left(2^{(1-\lambda)n/2}\right). \tag{8.3}$$

Comparison with [NS15]. Let us assume for instance that $2^{n/2-\varepsilon}$ queries to each of the oracles are allowed. Then, following Equations 8.2 and 8.3, combining the clamping trick and Algorithm 20 leads to a 3XOR in time $\mathcal{O}\left(2^{n/2+\varepsilon}/n\right)$ and space $\mathcal{O}\left(2^{(n/2+\varepsilon)/2}\right)$.

Nikolić and Sasaki [NS15] allow themselves to perform $2^{n/2-\varepsilon}$ queries to each of the oracles, where

$$\varepsilon \simeq \frac{1}{2}\log\left(\frac{n/2}{\ln(n/2)}\right).$$

They come up with an algorithm that has a time and space complexity about $\mathcal{O}\left(2^{n/2}/\sqrt{n/\ln n}\right)$ (see section 7.3). With the same number of queries, we are able to solve the 3XOR problem, using only M machine words, where:

$$\begin{aligned}\mathsf{M} &= \mathcal{O}\left(2^{n/4}\cdot 2^{\varepsilon/2}\right)\\ &= \mathcal{O}\left(2^{n/4}\cdot 2^{\log(n/\ln(n))/4}\right)\\ &= \mathcal{O}\left(2^{n/4}\cdot \sqrt[4]{n/\ln(n)}\right),\end{aligned}$$

and in time T, such that:

$$\begin{aligned}\mathsf{T} &= \mathcal{O}\left(\frac{2^{n/2+\varepsilon}}{n}\right)\\ &= \mathcal{O}\left(\frac{2^{n/2}\cdot\sqrt{n/\ln(n)}}{n}\right)\\ &= \mathcal{O}\left(\frac{2^{n/2}}{\sqrt{n\ln(n)}}\right).\end{aligned}$$

This gives us a $\ln(n)$ speedup compared to Nikolić and Sasaki's Algorithm with the same number of queries, and our method requires much less space.

Minimising the product TM. Let us assume that querying the oracles can be done in constant time. We denote by $\mathsf{T}_{\mathsf{tot}}$ the time complexity required to create the lists and to process them.

Proposition 8.2. *Making $2^{n/2-\log n}$ queries to the oracles and performing the clamping on $\kappa n = n/4 + \log(n)/4$ bits minimises the product $\mathsf{T}_{tot}\mathsf{M}$.*

Proof. In this setting, we have:

$$\mathsf{T}_{\mathsf{tot}} = \mathcal{O}\left(2^{\lambda n} + \frac{2^{2(\lambda-\kappa)n}}{n}\right) = \mathcal{O}\left(2^{\lambda n} + 2^{(1-\lambda)n-\log n}\right).$$

This reaches its equilibrium when:

$$\begin{aligned}\lambda_{eq} n &= (1-\lambda_{eq})n - \log n \\ \lambda_{eq} &= \frac{1}{2}\left(1 - \frac{\log n}{n}\right).\end{aligned}$$

When $\lambda > \lambda_{eq}$, the generation of the lists dominates the procedure, When $\lambda < \lambda_{eq}$ processing the lists dominates the procedure. According to Equation 7.11, when $\lambda \geq \lambda_{eq}$, we have:

$$\mathsf{T}_{\mathsf{tot}}\mathsf{M} = \mathcal{O}\left(2^{(\lambda+1)n/2}\right).$$

This reaches its minimum for $\lambda = \lambda_{eq}$. When $\lambda \leq \lambda_{eq}$, we have:

$$\begin{aligned}\mathsf{T}_{\mathsf{tot}}\mathsf{M} &= \mathcal{O}\left(2^{(1-\lambda)n-\log n} 2^{(1-\lambda)n/2}\right) \\ &= \mathcal{O}\left(2^{3(1-\lambda)n/2-\log(n)}\right) \\ &= \mathcal{O}\left(\frac{2^{3(1-\lambda)n/2}}{n}\right).\end{aligned}$$

This reaches its minimum for $\lambda = \lambda_{eq}$. Furthermore, this minimum is:

$$\mathsf{T}_{\mathsf{tot}}\mathsf{M} = \mathcal{O}\left(\frac{2^{3n/2}}{\sqrt[4]{n}}\right),$$

□

which is a very small gain (only $\sqrt[4]{n}$) compared to the Time-Memory product obtained in Section 7.5.2.

Remark 1. Once again, in practice, performing the clamping on less bits can be advantageous, depending on the time required to query the oracles.

8.2.2 Computing a 3XOR for Large n

The word RAM [FW93] model is helpful to think about the complexities of algorithms, however, it hits limitations when one tries to actually implement things. In fact, we may have to cope with values of n that are actually larger than the size of a word w. The clamping trick is already a good way to reduce the size of our problem, but it may not be enough. We used the following two-step strategy to deal with this problem:

1. Find and store all (i, j, k) such that $\mathcal{A}[i][..w] \oplus \mathcal{B}[j][..w] \oplus \mathcal{C}[k][..w] = 0$
2. For each triplet (i, j, k) thus found, check if $\mathcal{A}[i][w..] \oplus \mathcal{B}[j][w..] \oplus \mathcal{C}[k][w..] = 0$

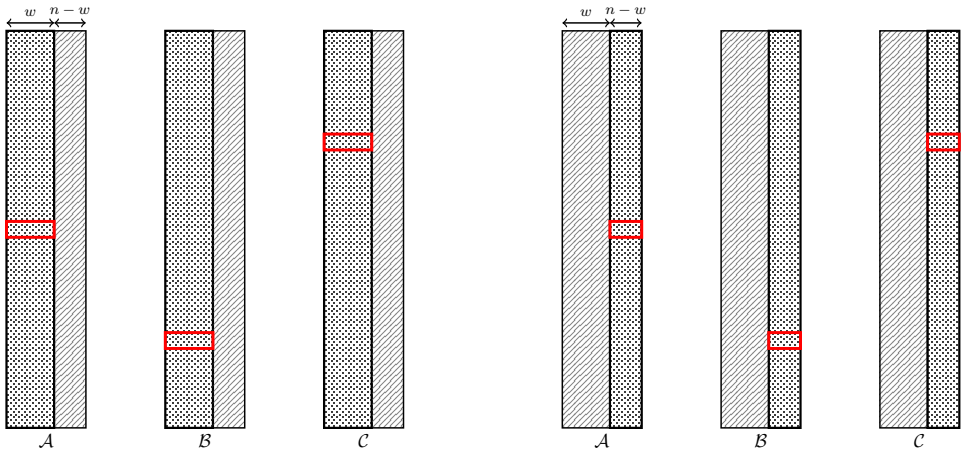


Figure 8.2 – Dealing with n larger than a machine word.

This idea is illustrated by Figure 8.2. The first step can easily be handled by Algorithm 20. We run our Algorithm on lists $\bar{\mathcal{A}}, \bar{\mathcal{B}}$ and $\bar{\mathcal{C}}$ of size $2^{n/3}$, that contains w-bit entries.

Then, from results given in Section 6.1.2 Equation 6.7, we expect to find N_{sol} triplet (i, j, k) in the first step of the procedure, with

$$\mathbb{E}[N_{sol}] = 2^{n-w}.$$

Checking these solutions in the second step should require about $\mathcal{O}(N_{sol}) = \mathcal{O}(2^{n-w})$ operations, and no extra storage.

Remark 2. This trick can also be applied to the Quadratic Algorithm. In fact, it can be applied to any 3XOR algorithm that, given as input $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, can return all triplets that are solutions.

8.3 Constant-Factor Improvements

In this section, we discuss possible improvements of Algorithm 20. Our goal is to reduce the number of iterations of the main loop. This would mean dealing with larger sublists of $\mathcal{C}$ at each iteration. In Algorithm 20, at each iteration, an invertible n-by-n matrix M is found, that sends the first ℓ bits of $n - \ell$ arbitrary vectors to zero. In this chapter we discuss means to find matrices that have the same effect on more than $n - \ell$ vectors, in order to process larger slices in each iteration. As such, less iterations would be needed.

Let w denote the size of a machine word, and let $m = \min(w, n)$ (i.e. m is the size of the instance that will actually be processed using Algorithm 20). We recall our goal is to find many (one per iteration of Algorithm 20) $(m-\ell)$-dimensional subspace $\mathscr{V}$ of $\mathbb{F}_2^m$ such that $\mathscr{V} \cap \mathcal{C}$ contains more than $m - \ell$ elements. We could for instance proceed as follows:

1. Find a sublist $\bar{\mathcal{C}}$ of $\mathcal{C}$ of size $N_{\bar{\mathcal{C}}} > (n - \ell)$ and an invertible matrix M such that all elements of $\bar{\mathcal{C}}M$ starts with ℓ zeroes.
2. Perform one iteration of Algorithm 20
3. Set $\mathcal{C}$ to $\mathcal{C} \setminus \bar{\mathcal{C}}$ and go to step 1.

This is a "one-at-a-time" approach. At each step, we search for the sublist $\bar{\mathcal{C}}$, and thus, if we do not want the complexity of our procedure to increase, we have to set up a "budget" of $\mathcal{O}(N_{\mathcal{C}})$ in time and space.

Another way to approach this problem would be to precompute all sublists $\bar{\mathcal{C}}_i$ in advance, and execute Algorithm 20 afterward. This would give the following "all-at-once" type of approach:

1. Permute the entries of $\mathcal{C}$ so that, for known parameters $N_0, N_1, \ldots$, with $N_i > n - \ell$, the first N_0 entries of $\mathcal{C}$ span a subspace $\mathscr{V}_0$ of dimension $n - \ell$, the following N_1 span an other subspace $\mathscr{V}_1$ of dimension $n - \ell$, and so on.

2. Run Algorithm 20, such that at each iteration i of the main loop, N_i vectors of $\mathcal{C}$ are transformed to vectors which start with $n-\ell$ zeroes.

As the pre-computation is done in advance, we can slightly increase our "budget" to $\mathcal{O}\left(N_{\mathcal{C}}^2/n\right)$. In this case, the time complexity should not increase, as $\mathcal{C}$ is the smallest of all the lists (if they are of different size).

8.3.1 Description of the Problem

We are given a list $\mathcal{C}$ which contains $N_{\mathcal{C}}$ uniformly random entries of n bits each. We aim to find an invertible matrix M that maximises (or at least increases) the number of $\mathbf{x} \in \mathcal{C}$ such that $(\mathbf{x}M)[..\ell] = 0$. Here M can be seen as a basis change matrix. Our problem is thus to find a basis $\mathscr{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ of $\mathbb{F}_2$ such that, if we call $\mathcal{C}'$ the list $\mathcal{C}$ written in this new basis, the subset of the elements of $\mathcal{C}'$ that starts with ℓ zeroes is the largest possible.

We choose $\mathscr{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ to be a basis of $\mathbb{F}_2^n$, such that there is a "large" sublist $\mathcal{C}_0$ of $\mathcal{C}$ such that for all $\mathbf{c}$ in $\mathcal{C}_0$, $\mathbf{c}$ is in $\mathsf{Span}\{\mathbf{b}_{\ell+1}, \ldots, \mathbf{b}_n\}$ (i.e. the subspace of $\mathbb{F}_2^n$ spanned by $\mathbf{b}_{\ell+1}, \ldots, \mathbf{b}_n$). Let M be the n-by-n invertible matrix that transforms all vectors of $\mathbb{F}_2^n$ to basis $\mathscr{B}$.

Remark 1. We can easily check that for all $\mathbf{c} \in \mathcal{C}_0$, the vector $\mathbf{c}' = \mathbf{c}M$ satisfies $\mathbf{c}'[..\ell] = 0$. Indeed, $\mathbf{c}$ being a linear combination of $\mathbf{b}_{\ell+1}, \ldots \mathbf{b}_n$ implies that there are α_i for $\ell < i \leq n$ in $\mathbb{F}_2$, such that

$$\mathbf{c} = \bigoplus_{i=\ell+1}^{n} \alpha_i \mathbf{b}_i.$$

Then, by linearity:

$$\begin{aligned} \mathbf{c}' = \mathbf{c}M &= \bigoplus_{i=\ell+1}^{n} \alpha_i M \mathbf{b}_i \\ &= \bigoplus_{i=\ell+1}^{n} \alpha_i \mathbf{e}_i. \end{aligned}$$

Reciprocally, if $\mathbf{c}' = \alpha_{\ell+1}\mathbf{e}_{\ell+1} \oplus \ldots \oplus \mathbf{e}_n$, we have:

$$\begin{aligned} \mathbf{c} = \mathbf{c}'M^{-1} &= \bigoplus_{i=\ell+1}^{n} \alpha_i M^{-1} \mathbf{e}_i \\ &= \bigoplus_{i=\ell+1}^{n} \alpha_i \mathbf{b}_i. \end{aligned}$$

The problem is then to find an invertible n-by-n matrix M over $\mathbb{F}_2$, such that for all $\mathbf{c}$ in $\mathcal{C}_0$, $(\mathbf{c}M)[..\ell] = 0$. In other words, if C is a $N_{\mathcal{C}_0}$-by-n matrix, whose rows are vectors from $\mathcal{C}_0$, and X an arbitrary $(n-\ell)$-by-n matrix, we want to find an invertible n-by-n matrix M such that

$$CM = (0|X). \tag{8.4}$$

From here, it is clear that our problem can be reformulated in more algebraic terms.

If we denote by $\mathscr{V}_0$, the $(n-\ell)$-dimensional subspace of $\mathbb{F}_2^n$ containing all vectors whose first ℓ coordinates are zeroes. The basis change matrix M should be chosen such that it sends the largest number of input vectors from $\mathcal{C}$ to $\mathscr{V}_0$. Alternatively let $\mathscr{V}$ be the pre-image of $\mathscr{V}_0$ through M: $\mathbf{x}M$ belongs to $\mathscr{V}_0$ if and only if $\mathbf{x}$ belongs to $\mathscr{V}$.

Finding a subspace $\mathscr{V}$ having a large intersection with the input list is the actual difficult task. Indeed, once a basis $(\mathbf{b}_{\ell+1}, \ldots, \mathbf{b}_n)$ is found, building an invertible matrix M that sends $\mathscr{V}$ to $\mathscr{V}_0$ is easy. We therefore are facing the following computational problem:

Problem 8.1. Let N, n, ℓ be three parameters. Given a list $\mathcal{L}$ of N uniformly random vectors of $\mathbb{F}_2^n$, the goal is to find a $(n-\ell)$-dimensional subspace $\mathscr{V}$ of $\mathbb{F}_2^n$, such that $\mathscr{V} \cap \mathcal{L}$ is as large as possible.

In our case, the list $\mathcal{L}$ is $\mathcal{C}$, $N = N_{\mathcal{C}}$ and we do not have to find only one, but many of such subspaces (one per iteration of Algorithm 20). Furthermore, we do not want this pre-computation step to dominate the whole procedure. As such, we set a "budget" of: $\mathcal{O}(N)$ operations in both time and space if this research is done at each iteration, or $\mathcal{O}(N^2/n)$ in time and $\mathcal{O}(N)$ in space if all subspaces are to be pre-computed in advance. As such, we give up on finding optimal solutions and instead look for heuristics that produce quick results.

Let $\bar{\mathcal{L}}$ be the (hopefully large) sublist of $\mathcal{L}$ such that $\forall \mathbf{x} \in \bar{\mathcal{L}}$, $\mathbf{x} \in \mathscr{V}$. We propose two approaches to compute $\bar{\mathcal{L}}$ from $\mathcal{L}$. One of them starts with an empty list $\bar{\mathcal{L}}$ and adds new elements in it, as the algorithm goes. The other one starts with $\bar{\mathcal{L}} = \mathcal{L}$ and at each step, removes from $\bar{\mathcal{L}}$ elements that do not belong to $\mathscr{V}$. This latter approach consists in searching for a vector space of dimension $(n-\ell)$ over $\mathbb{F}_2$, defined by a system of equations that are more often satisfied by vector from $\mathcal{L}$ than not. We claim that this problem is related to decoding problems. As such, we start by recalling some coding theory background in the following subsection. We will then present this decoding-based method in Section 8.3.3. Finally, we will present the other method, which consists in searching for linear dependencies in $\mathcal{L}$, in Section 8.3.4. This method allows us to find several $(n-\ell)$-dimensional vector spaces, and thus, to pre-compute all sub-sets at once, before actually running Algorithm 20.

Unfortunately, both methods described here only enable us to gain a constant factor on the number of iterations and nothing more significant.

8.3.2 Coding Theory and Information Set Decoding

Basic Facts About Random Linear Binary Code

A *linear binary (error-correcting) code* $\mathscr{C}$ of *length* n and *dimension* k, or as it is sometimes called a $[n,k]$ binary code is a subspace of $\mathbb{F}_2^n$ of dimension k. It contains 2^k codewords of n bits. We call the ratio $R := k/n$ the *rate* of $\mathcal{C}$. A linear code $\mathscr{C}$ can be seen as the linear span of the rows of a k-by-n matrix G called the *generator matrix* of the code. Formally:

$$\mathscr{C} = \{\mathbf{x}G | \mathbf{x} \in \mathbb{F}_2^k\}. \tag{8.5}$$

A linear code $\mathscr{C}$ can also be seen as the kernel of an $(n-k)$-by-n matrix H called the *parity check matrix* of the code. Formally:

$$\mathscr{C} = \{\mathbf{c} \in \mathbb{F}_2^n | \mathbf{c}^t H = 0\} \tag{8.6}$$

We call *(Hamming) weight* of a codeword $\mathbf{c}$ and we denote by $wt(\mathbf{c})$ the number of non-zero coefficients of $\mathbf{c}$.

Remark 2. This definition can be extended to any finite field $\mathbb{F}_q$. In fact, in general, a $[n,k]$ linear code is defined as being a k-dimensional subspace of $\mathbb{F}_q^n$, with a metric (usually the Hamming metric, sometimes the rank metric).

Small Weight Codewords

Minimum distance and Relative distance. The *minimum-distance* d of the code is the minimum weight of non-zero codewords. A $[n,k]$ code of minimum distance d is sometimes denoted as a $[n,k,d]$ code. We denote by $\delta := d/n$ the *relative distance* of the code.

Determining the minimum-distance of a linear code is a difficult problem. In fact, given a (small) positive integer $0 < w < n$, determining whether d is smaller than w is NP-complete. However, if $\mathscr{C}$ is a random code, more can be said about it. In fact, if we denote by $\mathcal{H}$ the binary entropy function:

$$\begin{aligned} \mathcal{H} :& [0,1] \to [0,1] \\ & x \mapsto -x\log x - (1-x)\log(1-x), \end{aligned} \tag{8.7}$$

We can approximate the relative distance δ of $\mathscr{C}$ using the following equation:

$$\mathcal{H}(\delta) = 1 - R. \tag{8.8}$$

This is due to the fact that a random linear code "almost reaches" the Gilbert-Vashamov bound [Gil52, Var57].

Theorem 8.3 (Asymptotic version of the Gilbert-Varshamov bound for binary codes)**.** *Let n and k be positive integers, let $R = n/k$, and let δ be a real in $(0, 1/2]$ that satisfies:*

$$R \leq 1 - \mathcal{H}(\delta), \tag{8.9}$$

then there exists a linear $[n, k, d]$ code over $\mathbb{F}_2$, with $d \geq \delta n$. Moreover, with high probability, any random linear $[n, k]$ code over $\mathbb{F}_2$ should satisfy:

$$R = 1 - \mathcal{H}(\delta), \tag{8.10}$$

where $d = \delta n$ is the minimum distance of the code.

This is a well known result of coding theory, that we are not going to prove here. A complete demonstration can be found in [Rot06], for the case of a linear code over any finite field $\mathbb{F}_q$.

Syndrome decoding problem. We are interested in finding a *minimum-weight codeword* in a random linear binary code $\mathscr{C}$ (i.e. a word $\mathbf{c} \in \mathscr{C}$ such that $wt(\mathbf{c}) = d$). This is related to the *Syndrom Decoding Problem* (SDP).

Problem 8.2 (SDP). Given a matrix $(n-k)$-by-n matrix H (the parity check matrix of our code $\mathscr{C}$), a target vector $\mathbf{s}$ (called syndrome), the goal is to find a vector $\mathbf{x}$ such that: $wt(\mathbf{x}) = t$, for some small known $t < n - k$.

Remark 3. In the particular setting where the target is $\mathbf{s} = 0$, and $t = d$ is the minimum-distance of $\mathscr{C}$, an algorithm which solve the SDP returns a minimum weight codeword in $\mathscr{C}$.

Unfortunately, the associated decisional problem is known to be NP-Complete [BMVT78]. The first ideas to solve the SDP problem were introduced by McEliece [McE78] in the security analysis of his cryptosystem, and the problem has been widely studied since. The most famous algorithm is the Las Vegas randomised *Information Set Decoding* procedure, which was introduced by Prange [Pra62] and has been improved several time since (e.g. [LB88, Ste88, MMT11, BJMM12, MO15]).

Information Set Decoding

Prange's original idea. The first ISD algorithm is due to Prange [Pra62]. His idea is to find a permutation P of the columns of H. Then denoting $\bar{H}$ the matrix HP, he aims to find a vector $\bar{\mathbf{x}}$ in $\mathbb{F}_2^n$, such that:

1. $\bar{\mathbf{x}}^t \bar{H} = \mathbf{s}$,
2. $wt(\bar{\mathbf{x}}) = t$,
3. $\bar{\mathbf{x}}[n-k..] = 0$.

Then, $\mathbf{x} = \bar{\mathbf{x}} P$ will be solution to the syndrome decoding problem. Indeed:

$$\begin{aligned} \bar{\mathbf{x}}^t \bar{H} = \mathbf{c} &\implies \mathbf{x} P^t P^t H = \mathbf{c} \\ &\implies \mathbf{x}^t H = \mathbf{c}. \end{aligned} \tag{8.11}$$

Remark 4. If the procedure described in Algorithm 21 ends, then it actually solves the Syndrome decoding problem. In fact, we assume that at some point, a permutation P that satisfies the three conditions stated above is picked. Then, we can obtain a *row-reduced echelon* form of $\bar{H}$, computing the Gauss-Jordan elimination of $\bar{H}$. In other words, as $\bar{H}$ is full-rank, we can find an invertible matrix V of size $(n-k)$, such that $V\bar{H}$ looks as follows:

Algorithm 21 Prange's ISD Algorithm.

Require: A $(n-k)$-by-n matrix H, a syndrome $\mathbf{s}$, an integer $0 < t < n-k$.
Ensure: A vector $\mathbf{x}$ such that $wt(\mathbf{x}) = t$ and $\mathbf{x}^t H = \mathbf{s}$.

```
Set x_1 ← 0
while wt(x_1) ≠ t do
    Pick a random permutation P of the columns of H
    Set H̄ ← HP
    Find an invertible matrix V such that VH̄ is the row-reduced echelon form of H̄
    Set x_1 ← s^t V
Set x ← (x_1|0)
return xP
```

$$V\bar{H} = (I_{n-k}|X),$$

where, X is an arbitrary matrix. Then, if we denote by $\mathbf{x}_1$ the vector of $\mathbb{F}_2^{n-k}$ such that $\mathbf{x}_1 := \mathbf{s}^t V$, and by $\mathbf{x}$ the vector of $\mathbb{F}_2^n$ such that $\mathbf{x} = (\mathbf{x}_1|0)$, we have:

$$\begin{aligned}
\mathbf{x}^t\bar{H} &= \mathbf{x}^t(VH)^t V^{-1} \\
&= (\mathbf{x}_1|0)\begin{pmatrix} I_{n-k} \\ {}^tX \end{pmatrix} {}^tV^{-1} \\
&= \mathbf{x}_1 {}^tV^{-1} \\
&= \mathbf{s}^t V {}^tV^{-1} \\
&= \mathbf{s}
\end{aligned} \tag{8.12}$$

The expected time complexity of this algorithm in the general case has been analysed in [CG90], and is about $\mathcal{O}\left(2^n \cdot F(k/n)\right)$, with:

$$F(R) = (1-R)(1-\mathcal{H}(\delta/(1-R))). \tag{8.13}$$

This simple procedure has known many improvements, first by Lee and Brickell [LB88], which gain a polynomial time factor compared to Prange Algorithm. Stern [Ste88] and Dumer [Dum91], improved this idea to reduce the worst case complexity. May, Meurer and Thomae [MMT11] reduced again this worst case complexity using representation techniques, and this was later improved in Becker, Joux, May and Meurer [BJMM12]. This last algorithm (BJMM) is currently the best known asymptotic algorithm for decoding random linear in the worst case. In [MO15], May and Ozerov proposed a way to speed-up the BJMM algorithm, by improving the time complexity of a sub-routine called the "Nearest-Neighbour problem". This idea was recently improved by Both and May [BM18], leading to an asymptotic speed-up of the BJMM algorithm in the worst case. Independently, Canteau and Chabaud proposed in [CC98] a variant of the ISD designed specifically to find a small weight codeword.

However, since Stern's algorithm, all these improvements actually mean to reduce the worst case complexity (i.e. when the rate of the code is around 0.46), and are not significant in other cases. Furthermore, the more recent variants are harder to implement, and require more extra storage. We claim that Lee and Brickell's algorithm should be efficient enough for our application, while remaining simple.

Lee and Brickell's Algorithm Lee and Brickell [LB88] propose a way to relax Prange's algorithm, in order to amortise the cost of the Gauss-Jordan elimination. Like before, they search for a specific permutation P of the column of H and, if we denote $\bar{H}$ the matrix HP, they search for a vector $\bar{\mathbf{x}}$ in $\mathbb{F}_2^n$, such that for a specific integer $0 \leq p \leq t$:

1. $\bar{\mathbf{x}}^t\bar{H} = \mathbf{s}$,

2. $wt(\bar{\mathbf{x}}[..n-k]) = t - p$,

3. $wt(\bar{\mathbf{x}}[n-k..]) = p$.

Remark 5. Prange's algorithm can be seen as the special case where $p = 0$.

Algorithm 22 Lee and Brickell's ISD Algorithm.

Require: A $(n-k)$-by-n matrix H, a syndrome $\mathbf{s}$, an integer $0 < t < n-k$, an integer $0 \leq p < t$.
Ensure: A vector $\mathbf{x}$ such that $wt(\mathbf{x}) = t$ and $\mathbf{x}^t H = \mathbf{s}$.

```
Set x_1 ← 0, x_2 ← 0
while True do
    Pick a random permutation P of the columns of H
    Set H̄ ← HP
    Find an invertible matrix V such that VH̄ is the row-reduced echelon form of H̄
    Set X to the sub-matrix build from the k last column of VH̄
    for all x_2 s.t. wt(x_2) = p do
        Set x_1 ← s^t V + x_2^t X
        if wt(x_1) = w − p then
            Set x ← (x_1|x_2)
            return xP
```

We detail the Lee-Brickell method in Algorithm 22. Once again, we can see that if the algorithm ends, it returns a solution to the syndrome decoding problem. Indeed, we assume that a permutation P that satisfies the three conditions above has been picked. We have $V\bar{H} = (I_{n-k}|X)$. Then, if we consider the couple of vectors $(\mathbf{x}_1, \mathbf{x}_2) \in \mathbb{F}_2^{n-k} \times \mathbb{F}_2^k$, such that $\mathbf{x}_1 := \mathbf{s}^t V + \mathbf{x}_2{}^t \bar{X}$, and if we denote $\mathbf{x}$ the vector of $\mathbb{F}_2^n$ such that $\mathbf{x} = (\mathbf{x}_1|\mathbf{x}_2)$, we have:

$$\begin{aligned}
\mathbf{x}^t \bar{H} &= \mathbf{x}^t (VH)^t V^{-1} \\
&= (\mathbf{x}_1|\mathbf{x}_2) \begin{pmatrix} I_{n-k} \\ {}^t X \end{pmatrix} {}^t V^{-1} \\
&= (\mathbf{x}_1{}^t V + \mathbf{x}_2{}^t X)^t V^{-1} \\
&= (\mathbf{s}^t V + \mathbf{x}_2{}^t X + \mathbf{x}_2{}^t X)^t V^{-1} \qquad (8.14)
\end{aligned}$$

and as we are working over $\mathbb{F}_2$, it follows:

$$\mathbf{x}^t \bar{H} = \mathbf{s}^t V {}^t V^{-1} = \mathbf{s} \qquad (8.15)$$

Remark 6. Usually taking $p = 2$ is optimal.

This procedure does not allow to reduce significantly the time complexity of Prange's algorithm. In fact, it can only gain a polynomial factor bounded by $n(n-k)$.

8.3.3 Finding Biased Equations over $\mathcal{L}$

Recall that an $(n-\ell)$-dimensional subspace of $\mathbb{F}_2^n$ is defined by a system of ℓ linear equations in n variables. Finding a vector space $\mathcal{V}$ having a large intersection with $\mathcal{L}$ amounts to finding ℓ linear equations that are simultaneously biased over $\mathcal{L}$ (i.e. more often simultaneously true than the contrary).

We propose a greedy approach to find these equations: first initialise a list $\bar{\mathcal{L}}$ to $\mathcal{L}$, find a biased equation E_1 over $\bar{\mathcal{L}}$, then remove from $\bar{\mathcal{L}}$ all vectors that do not satisfy E_1, and re-iterate the method $\ell - 1$ times. This reduces the problem to that of finding a single biased equation over $\mathcal{L}$.

Finding *the* most biased equation over $\bar{\mathcal{L}}$ is tempting but too expensive: an exhaustive search would require $\mathcal{O}\left(2^{n+k}\right)$ operations and FFT-like methods such as the Walsh transform still have a workload of order $\mathcal{O}\left(k \cdot 2^n\right)$. We have to settle for a hopefully "good" if not optimal bias.

Intuition of the method. Finding a biased linear equation over $\mathcal{L}$ is a *decoding problem*. Consider the binary linear code $\mathscr{C}$ spanned by the columns of $\mathcal{L}$, and denote G the generator matrix of $\mathscr{C}$. If the columns of $\mathcal{L}$ are linearly independent (which is most likely the case as, they are uniformly random), then G is the matrix whose columns are the entries of $\mathcal{L}$. Let $\mathbf{y}$ be a low-weight codeword: this means that there is a vector $\mathbf{x}$ such that $\mathbf{y} = \mathbf{x}G$ and $\mathbf{y}$ has a low Hamming weight (amongst all possible vectors y). We claim that $\mathbf{x}$ describes the coefficients of one of the most biased linear equations over $\mathcal{L}$.

Indeed, for some $0 \leq i < N$, $\mathbf{y}[i] = 0$ if and only if $\langle \mathbf{x}, {}^t\mathbf{g}_i \rangle = 0$, where ${}^t\mathbf{g}_i$ is the column of G indexed by i. By definition of G, this is the same as saying that

$$\mathbf{y}[i] = 0 \iff \langle \mathbf{x}, \mathcal{L}[i] \rangle = 0.$$

If we call $\mathcal{E}$ the equation defined by the coefficients of $\mathbf{x}$, then

$$\mathbf{y}[i] = 0 \iff \mathcal{E}(\mathcal{L}[i]) = 0.$$

Now, if $\mathbf{y}$ has low Hamming weight, it means that $\mathbf{y}[i]$ is more often 0 than not. This is enough to prove that $\mathbf{x}$ actually describes the coefficients of one of the most biased linear equations over $\mathcal{L}$.

As discussed in Section 8.3.2, ISD algorithms can find codewords $\mathbf{c}$ of Hamming weight $wt(c) = d$ where d is the minimum distance of the code. Their complexity is exponential in the length of the code (here, the length of the code is equal to the number of vector in the input list), and thus we has to set an upper-bound N_{max}, on the length of the code we can expect to process given a time budget of N operations. We will actually determine N_{max} later.

Thus, before using decoding algorithm, we have to reduce the input list below N_{max}. To this end, we initialise $\bar{\mathcal{L}}$ to $\mathcal{L}$ and while the size of $\bar{N}$ of $\bar{\mathcal{L}}$ is greater than N_{max}, we choose an arbitrary equation $\mathcal{E}$, and remove from $\bar{\mathcal{L}}$ all entries that do not satisfy $\mathcal{E}$. Each iteration of this process should halve the size of $\bar{\mathcal{L}}$.

Procedure description. Let N_{max} and ε be fixed parameters such that N_{max} represents an upper bound on the length of the initial code $\mathscr{C}$, and a parameter ε. Our method basically works as follows:

1. Initialise $\bar{\mathcal{L}}$ to $\mathcal{L}$, $\bar{N}$ to N and t (a counter to the number of equations) to 0.
2. While $N > N_{max}$, select an arbitrary equation and remove from $\bar{\mathcal{L}}$ all the elements that do not satisfy this equation. At each iteration increment t.
3. While $t < \ell$:
   (a) Find a small weight codeword $\mathbf{c}$ in the code spanned by the columns of $\mathcal{L}$. Indeed we search for a $\mathbf{c}$ such that $wt(\mathbf{c}) \leq d + \varepsilon$, where d is the minimum distance of the code and ε a small approximation parameter.
   (b) Remove from $\bar{\mathcal{L}}$ all entries $\bar{\mathcal{L}}[i]$ such that $\mathbf{c} = 1$.

This procedure is detailed in Algorithm 23.

Theorem 8.4. *Algorithm 23 returns a subset of $\mathcal{L}$ that spans a $(n-\ell)$-dimensional subspace of $\mathbb{F}_2^n$.*

Proof. We claim that at each iteration of the *Search biased equations* loop, $\bar{\mathcal{L}}$ consists of vectors which belong to a vector subspace of $\mathbb{F}_2^n$ of dimension $n - t$.

Indeed, at the end of the *Filter* step, $\bar{\mathcal{L}} \subset \mathscr{V}_k$, where $\mathscr{V}_k$ is the vector subspace of $\mathbb{F}_2^n$ consisting of vectors whose first k coordinates are zeroes.

Now, recall that $\mathbf{c}$ is a codeword which belongs to the code spanned by the columns of $\bar{\mathcal{L}}$. This means that there exists a vector $\mathbf{x} \in \mathbb{F}_2^{\bar{N}}$ such that $\mathbf{c} = \mathbf{x}^t G$. The for loop at the end of Algorithm 23 removes from $\bar{\mathcal{L}}$ all vectors that are not orthogonal to $\mathbf{x}$. It follows that after this step, $\bar{\mathcal{L}}$ consists of vectors that live in $\mathscr{V}_{t+1} = \mathscr{V}_t \cap \{0, \mathbf{x}\}^\perp$. $\mathscr{V}_{t+1}$ is indeed a vector space, as the intersection of two vector spaces. Furthermore, because $\mathbf{c}$ is non-zero, at least one vector will be removed from the list. This vector belongs to $\mathscr{V}_t$ but is not in $\mathscr{V}_{t+1}$. It follows that the dimension of $\mathscr{V}_{t+1}$ is at least one less that the dimension of $\mathscr{V}_t$. □

Algorithm 23 ISD-based Approach.

Require: A list $\mathcal{L}$ (Sorted in ascending order) of size N, two parameters: N_{max} and ε.
Ensure: A (hopefully large) sublist $\bar{\mathcal{L}}$, such that all entries of $\bar{\mathcal{L}}$ satisfy ℓ linear equations.

```
Set t ← ⌈log N_C / N_max⌉                                   ▷ Initialisation
j ← 0
while L[j][..t] = 0 do                                      ▷ Filter
    j ← j + 1
Set L̄ ← L[..j]
while t < ℓ do                                              ▷ Search biased equations
    Create G the matrix whose rows are the entries of L̄
    Estimate the minimum distance d of the code C spanned by ᵗG
    Find a non-zero codeword c in C s.t. wt(c) ≤ d + ε.
    Set t ← t + 1
    for all i s.t. c[i] = 1 do
        Remove L̄[i] from L̄
```

Estimation of N_{max}. Knowing N the size of the initial list $\mathcal{L}$ it is possible to estimate the upper bound N_{max} over the length of the code that the ISD algorithm can process within our time budget.

Let us denote by f the function $f(x, y) := x \cdot F(y/x)$ (for $0 < y < x$), where F is the function given by Equation 8.13. As discussed in Section 8.3.2, a minimum-weight codeword in a code $\mathscr{C}$ of length N_{max} and dimension $n - \ell$ can be found using the Lee-Brickell Algorithm in time less than $\mathcal{O}\left(2^{f(N_{max}, n-\ell)}\right)$.

If we denote by T_{ISD}, the total time spent in the $k-\ell$ iterations of the Biased equation research, we claim that

$$T_{ISD} \leq (k - \ell) \cdot 2^{f(N_{max}, n-\ell)}. \tag{8.16}$$

The function $f(x, y)$ grows with x, and also grows with y when $y < 0.1207x$. This is enough to prove Inequality (8.16), assuming that N_{max} is much larger than $n - \ell$.

To be sure that the complexity of Algorithm 23 does not exceed $\mathcal{O}(N)$, we want to find the maximum value of N_{max} so that $T_{ISD} < N$. We decide to choose N_{max} so that

$$(k - \ell) \cdot 2^{f(N_{max}, n-\ell)} = N. \tag{8.17}$$

Interesting considerations. Let us assume that we dispose of three lists of size 2^ℓ, for some parameter $\ell < n/2$. An interesting question would be: is it possible to reduce the time and space complexity of Joux's algorithm using the following simple variant?

1. Find a sublist $\bar{\mathcal{C}}$ of $\mathcal{C}$ such that $\bar{\mathcal{C}}$ span a $(n-\ell)$-dimensional subspace of $\mathbb{F}_2^n$ using the procedure above
2. Compute an invertible matrix M such that the elements of $\bar{\mathcal{C}}M$ start with ℓ zeroes
3. Solve the 3XOR problem with input lists $\mathcal{A}M$, $\mathcal{B}M$ and $\bar{\mathcal{C}}M$.

Indeed, if $\mathcal{C}$ is large enough, according to Equation 7.2, choosing ℓ so that $2^\ell = 2^{n/2}/\sqrt{\bar{N_\mathcal{C}}}$, would allow to reduce the time and space complexity of the procedure to:

$$\mathsf{T} = \mathsf{M} = \mathcal{O}\left(2^{n/2}/\sqrt{\bar{N_\mathcal{C}}}\right).$$

Estimating the value of ℓ and of $N_\mathcal{C}$ can be done numerically (for instance using the bisection algorithm).Table 8.1 gives an estimation of the parameters for some values of n and k. Unfortunately, it allows to gain only a constant speedup about $\sqrt{2}$ compared to Joux's Algorithm, and requires a lot of space. However, in practice, for small values of n, this method is faster than both Joux and Nikolić-Sasaki algorithms.

Table 8.1 – Parameter estimation of the ISD-based algorithm for some value of n.

n	ℓ	N_{max}	expected size of $\bar{\mathcal{L}}$
64	28	382	81
128	60	1996	154
256	123	9894	285
512	252	60271	545

Table 8.2 – Benchmarks.

(a) $n = 52$

method	avg. time (s)	$\lvert L_1\rvert + \lvert L_2\rvert + \lvert L_3\rvert$
Joux	39.0	$52'644'552$
NS	65.74	$100'663'296$
ISD	24.18	$41'943'040$

(b) $n = 64$

method	time (s)	$\lvert L_1\rvert + \lvert L_2\rvert + \lvert L_3\rvert$
Joux	1387	$3'037'000'532$
NS	1933	$6'442'450'944$
ISD	900	$5'368'709'120$

Indeed, we implemented this algorithm, along with the ones of Joux and Nikolić-Sasaki in C using M4RI [AB12] to deal with binary matrices. Our objective was to compare these different algorithms in practice on small values of n (up to 64) where the computation is feasible.

As the probability of success of the classic birthday algorithm on n bits, with input lists of size $2^{n/2}$ is close to $1/2$, we adjusted the size of the lists in order to get a 99% success probability in each case.

We carried out benchmarks using a small parameter ($n = 52$) on a workstation (Intel Core i7, 16 Gb RAM) and a larger parameter ($n = 64$) on a server (Intel Xeon E5-2686 v4, 488Gb RAM). The results are given by Table 8.2. These benchmarks confirm that the algorithm described above is actually faster than the previous state of the art by a factor 1.5, as expected.

8.3.4 Finding Linear Dependencies Using Wagner's Algorithm

Let $2 \leq k < \ell$. The main idea is to search for distinct elements $\mathbf{x}_0, \dots \mathbf{x}_k$ of $\mathcal{L}$ such that:

$$\mathbf{x}_0 = \mathbf{x}_1 \oplus \cdots \oplus \mathbf{x}_k.$$

Then, assuming that $\mathbf{x}_1, \dots, \mathbf{x}_k$ are linearly independent, they can be chosen as vectors of a basis $\mathscr{B}$ of $\mathscr{V}$. Let us assume that $\bar{\mathcal{L}} \supseteq \{\mathbf{x}_0, \mathbf{x}_1, \dots \mathbf{x}_k\}$. If we are able to find about $(n-\ell)/k$ such $(k+1)$-tuples, then the size $\bar{N}$ of $\bar{L}$ will be at least $(n-\ell)(k+1)/k$. As we need to create $\mathcal{O}(N/n)$ of such sublists, we would like to find $\mathcal{O}(N)$ tuples.

To find these tuples, we actually need to solve the $(k+1)$XOR problem, where all the input lists are $\mathcal{L}$ (a slight variation is required to enforce a "pairwise distinct" condition). Indeed, if $(\mathbf{x}_0, \mathbf{x}_1, \dots \mathbf{x}_k) \in \mathcal{L}^{k+1}$ is a solution to the $(k+1)$XOR problem, then $\mathbf{x}_0 \oplus \cdots \oplus \mathbf{x}_k = 0$ and, in particular, $\mathbf{x}_0 = \mathbf{x}_1 \oplus \cdots \oplus \mathbf{x}_k$.

Remark 7. The smaller k is, the more substantial the gain is. On the other hand, the problem is also harder to solve. Indeed, if $k = 2$, finding only one triplet $(\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2) \in \mathcal{L}^3$ such that $\mathbf{x}_0 = \mathbf{x}_1 \oplus \mathbf{x}_2$ would require to solve a 3XOR instance. This is already too hard.

We claim that setting k equal to 3 is a good choice in our case. This would allow us to find sublists of size at least $4(n-\ell)/3$.

Intermediate result. Let us assume that $N = \alpha 2^{n/3}$ for some parameter α. Let t be a well chosen parameter. Following Wagner 4-tree algorithm, we can, for instance do the following:

1. Compute the join list $\mathcal{L}_{join} = \mathcal{L}[..N/2] \bowtie_t \mathcal{L}[N/2..]$. The parameter t is chosen so that the expected size of $\mathcal{L}_{join}$ is N.
2. Compute the set $\mathcal{S}$ of all $((\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2)) \in \mathcal{L}_{join}[..N/2] \times \mathcal{L}_{join}[N/2..]$, such that $\mathbf{x}_1 \oplus \mathbf{y}_1 \oplus \mathbf{x}_2 \oplus \mathbf{x}_3$.

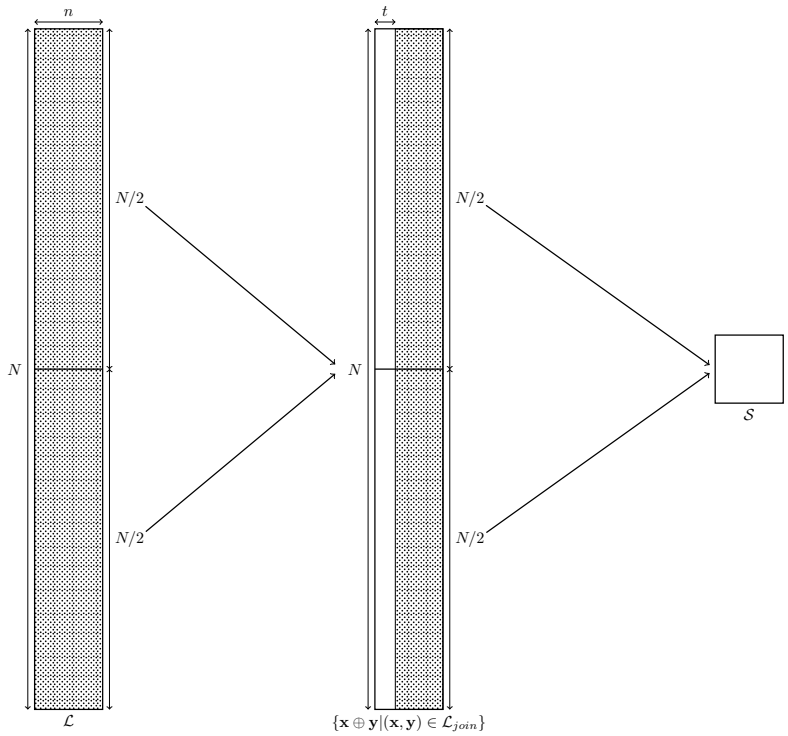


Figure 8.3 – Variant of Wagner 4tree algorithm.

This is actually exactly Wagner 4-tree algorithm, with a different choice of parameters that suits our problem better. We summarise this idea in Figure 8.3.

Proposition 8.5. *The procedure above returns a set $\mathcal{S}$ of $\alpha^3/16$ quadruplets $(\mathbf{x}_1, \mathbf{y}_1, \mathbf{x}_2, \mathbf{y}_2) \in \mathcal{L}^4$ such that $\mathbf{x}_1 \oplus \mathbf{y}_1 \oplus \mathbf{x}_2 \oplus \mathbf{y}_2 = 0$, in time and space $\mathcal{O}\left(\alpha 2^{n/3}\right)$.*

Proof. We divide our proof in two parts. First we are going to prove that this procedure actually solves the problem, then we will show that the time and space complexity given are indeed exact.

1. The procedure above returns $\alpha^3/16$ solutions to the 4XOR problem. The size of both of the lists $\mathcal{L}[..N/2]$ and $\mathcal{L}[N/2..]$ is $N/2 = (\alpha/2)2^{n/3}$. Thus the expected size of the list $\mathcal{L}_{join}$ is:

$$\mathbb{E}[N_{join}] = \frac{\alpha^2 2^{2n/3}}{2^{2+t}} = \alpha^2 2^{2n/3-t-2}.$$

If we want $\mathbb{E}[N_{join}] = N$, t must satisfy:

$$\begin{aligned} \alpha^2 2^{2n/3-t-2} &= \alpha 2^{n/3} \\ \alpha 2^{n/3-t-2} &= 1. \end{aligned}$$

Taking the logarithm, this leads to:

$$\log(\alpha) + n/3 - t - 2 = 0,$$

and thus

$$t = \log(\alpha) + n/3 - 2. \tag{8.18}$$

Then, the expected number of elements in $\mathcal{S}$ will be:

$$\mathbb{E}[N_{sol}] = \frac{\alpha^2 2^{2n/3}}{2^{2+n-t}}.$$

Replacing t by the value given in Equation 8.18, we obtain:

$$\mathbb{E}[N_{sol}] = \alpha^2 2^{3n/3-2+\log(\alpha)-n-2} = \frac{\alpha^3}{16}.$$

2. The procedure above has a time and space complexity of $\mathcal{O}\left(\alpha 2^{n/3}\right)$. As discussed in Section 6.3, the first step can efficiently be computed in time and space $\mathcal{O}(N)$, using Algorithm 14. The second step requires first to store half of the entries of $\mathcal{L}_{join}$ in a hash table, and then for the remaining entries, to compute one XOR and to search for some matching element in said hash table. This can also be done in time and space $\mathcal{O}(N)$. □

Instead of doing the join on the first t bits, we can choose to take a random subset $\mathcal{J}$ of $\{0, \ldots, n-1\}$, and compute the lists $\mathcal{L}_{join}$ of couples (x, y) that agree on the columns indexed by $\mathcal{J}$. If we reiterate the procedure above $\mathcal{O}(N/n)$ times, with different subsets $\mathcal{J}$ we can find about $\alpha^3 \cdot N/(16n)$ quadruplets $(\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$ such that $\mathbf{x}_0 = \mathbf{x}_1 \oplus \mathbf{x}_2 \oplus \mathbf{x}_3$.

We claim that we should get enough quadruplets as long as $N \geq \sqrt[3]{16n} \cdot 2^{n/3}$. Indeed, we already mentioned that we need about N quadruplets. And we have just shown that if $N = \alpha 2^{n/3}$, we should get about $N_{sol} = (N/n) \cdot (\alpha^3/16)$ solutions with N/n iteration of the previous method. If $\alpha \geq \sqrt[3]{16n}$, this gives:

$$N_{sol} \geq \frac{N}{n} \cdot \frac{16n}{16},$$

and thus $N_{sol} \geq N$.

Remark 8. It is also possible to apply a variant of the 4-tree algorithm, which enforces the condition pairwise distinct, with $\mathcal{L}^4$ as input. This should allow us to get rid of the constant factor 1/16.

Computing All Sublists at Once Using the 4XOR-Based Algorithm

Recall that the problem we want to solve is the following: find several sublists of $\mathcal{C}$ such that all these sublists span (disjoint) $(m-\ell)$-dimensional subspaces of $\mathbb{F}_2^m$ (where $m = \min(n, w)$ is the size of the instance to be processed). We can find them all at once, before actually running Algorithm 20.

In this setting, if we do not want the time complexity of the pre-computation step, to dominate the whole algorithm, we have to set a "budget" of $\mathcal{O}(N_{\mathcal{C}}^2/m)$ for this pre-computation step. Using the method we have just described, if $N_{\mathcal{C}} = 2^{m/2}$, we are in the interesting case where we will find about $2^{m/2}$ such quadruplets with only a constant number of iteration of the procedure above, and thus in time and space $\mathcal{O}(N_{\mathcal{C}})$.

We then need to isolate a (large) subset of pairwise disjoint quadruplets. Finding the biggest possible subset is the 4D MATCHING problem. Given four finite disjoint sets X, Y, Z, T. Let S be a subset of $X \times Y \times Z \times T$, and let $M \subseteq S$ such that if $(x_1, y_1, z_1, t_1) \in M$ and $(x_2, y_2, z_2, t_2) \in M$ then $x_1 \neq x_2$, $y_1 \neq y_2$, $z_1 \neq z_2$, $t_1 \neq t_2$.

$\mathcal{M}$ is said to be *maximal* if there is no matching $\mathcal{M}'$ such that $\mathcal{M} \subsetneq \mathcal{M}' \subseteq S$.

$\mathcal{M}$ is said to be *maximum* if there is no matching $\mathcal{M}' \subseteq S$ such that $|\mathcal{M}'| > |\mathcal{M}|$. These definitions are illustrated by Figure 8.4.

Finding a *maximum* 4D matching is NP-Complete optimisation problem. However, any *maximal* 4D matching is a 4-approximation to a maximum 4D matching, and it can be found efficiently by a greedy algorithm. Let $\mathcal{M}$ denote a 4D matching, and r denote its cardinality. We claim that performing the method above a constant number of time, we should be able to create a matching $\mathcal{M}$ such that $r \simeq 2^{n/2}$.

We build a permuted list $\mathcal{D}$ as follows: start from an empty list; for each $\{x, y, z, t\} \in \mathcal{M}$, append $\mathcal{C}[x], \mathcal{C}[y], \mathcal{C}[z]$ and $\mathcal{C}[t]$ to $\mathcal{D}$. Finally, append $\mathcal{C} - \mathcal{D}$ to $\mathcal{D}$, in any order. We obtain sublists 1/4 larger that with the original Algorithm 20. This algorithm can be updated to exploit $\mathcal{D}$, with a running time reduced by 25%.

Remark 9. This method can possibly be improved. Indeed, assume that:

$$\mathbf{x}_0 = \mathbf{x}_1 \oplus \mathbf{x}_2 \oplus \mathbf{x}_3,$$

(a) Maximal 4D matching that is not maximum.

(b) Maximum 4D matching.

Figure 8.4 – A maximal matching is shown on Figure 8.4a. This matching is not maximum. A larger one is shown on Figure 8.4b.

and

$$\mathbf{y}_0 = \mathbf{x}_0 \oplus \mathbf{y}_1 \oplus \mathbf{y}_2.$$

Assuming that $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{y}_1$ and $\mathbf{y}_2$ are linearly independent, they can be chosen as basis vectors. Then five basis vectors give two linear dependencies. This is however a very small gain, and we have not given much thought of how it can be exploited.

8.4 Experimentation and Results

In order to have a better understanding of the tradeoffs we presented in this section as well as the previous one, we chose to tackle an academic "practical" problem: Computing a 3XOR on n bits of the SHA256 hash function, for the largest possible value of n.

Most of the literature devoted to the generalised birthday problem is mostly theoretical, in particular because the exponential space requirement of theses algorithms makes them quickly impractical. One notable exception is this paper from Bernstein et al. [BLN+09], which provides the source code of a high quality implementation of Wagner's k-list algorithm. Studying this code was enlightening to us.

We performed our tests on a "Haswell" Core i5 CPU. Our implementation of the quadratic algorithm (see Section 7.1) takes about 340 CPU hours, while our implementation of Algorithm 20 (see Section 8.1) takes 105 CPU hours.

8.4.1 Computing a 96-bit 3XOR

General method. Note that 2^{32} queries to each oracles should be enough to find a 3XOR with high probability. In fact, if we create lists $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$ of size 2^{32}, the product of the size of the lists is 2^{96}, and then there should be a solution to the problem with high probability.

We allow ourself to make 2^{48} queries, which is then more than strictly required from an information theoretic point of view. This enable us to perform a clamping on $24 = (3 \cdot 48 - 96)/2$ bits. We come up with three lists $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ of about N entries, with

$$N = 2^{48-24} = 2^{24}.$$

As the entries all started with 24 zeroes, we only had to consider $n' = 96 - 24 = 72$ bits for each of them to solve the 3XOR problem in $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$. In this setting, only one solution is expected.

The size of a word on a standard machine is $w = 64$ which is less than 72. To find our solution, we use the trick described in Section 8.4.2, which enables us to solve the 3XOR problem, on bit-strings that do not fit in a machine word. We actually proceeded as follows:

We can create the lists $\bar{\mathcal{A}}$, $\bar{\mathcal{B}}$ and $\bar{\mathcal{C}}$, such that for all i $\bar{\mathcal{A}}[i] = \mathcal{A}[i][..64]$ (resp. $\bar{\mathcal{B}}[i] = \mathcal{B}[i][..64]$ and $\bar{\mathcal{C}}[i] = \mathcal{C}[i][..64]$) These three lists were of size about 2^{24}, and contained 64-bit entries.

Processing them with either the Quadratic Algorithm or Algorithm 20 should allow us to find about $2^{72-64} = 2^8 = 256$ 64-bit solutions. For each of them, we can recover the corresponding triplets (i, j, k) by creating $\bar{\mathcal{A}}$, $\bar{\mathcal{B}}$ and $\bar{\mathcal{C}}$ over again.

Once this is done, for each triplet (i, j, k) we only have to get $(\mathcal{A}[i][64..], \mathcal{B}[j][64..], \mathcal{C}[k][64..])$ and check if $\mathcal{A}[i][64..] \oplus \mathcal{B}[j][64..] \oplus \mathcal{C}[k][64..]$. This means about 256 tests to perform.

Creating the lists cannot be neglected. A single core of a modern CPU is capable of evaluating SHA256 on single-block inputs about 20 million times per second, so that creating the lists takes 11,851 CPU hours sequentially. Put another way, generating the input lists is 100 times slower than actually processing them.

It would have been smarter to do the clamping on 22 bits instead of 24. This would have reduced the total running time by a 2.5 factor at the expense of making the lists 4 times bigger. This is ultimately dependent on the speed at which oracles can be evaluated.

8.4.2 Clusters, Cache, Register and Other Gory Details

The Quadratic algorithm : A more practical design. The summary description given above suggests to allocate a hash table that holds the whole $\mathcal{C}$ list. Most accesses to this hash table are likely to incur the penalty of a cache miss, and this can be avoided at no cost. We dispatch the entries of $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$ into buckets of small expected size using their most-significant bits. An eventual 3XOR must lie in $A^{[\mathbf{u}]} \times B^{[\mathbf{v}]} \times C^{[\mathbf{u}\oplus\mathbf{v}]}$ for some $\mathbf{u}, \mathbf{v}$. The idea is to process buckets of $\mathcal{C}$ one-by-one, storing them in a small hash table. Then, we consider all the pairs from the corresponding bins of A and B. This yields Algorithm 24. The parameters k and ℓ should be chosen so that the hash table created at step 5 and the corresponding portions of $\mathcal{A}$ and $\mathcal{B}$ fit in the L3 cache.

Algorithm 24 Cache-Friendly Quadratic Algorithm.

Require: Lists $\mathcal{A}, \mathcal{B}, \mathcal{C}$ of size $N_{\mathcal{A}}$, $N_{\mathcal{B}}$ and $N_{\mathcal{C}}$, and two parameters k and ℓ
Ensure: A couple $(\mathbf{x}, \mathbf{y}) \in \mathcal{A} \times \mathcal{B}$ such that $\mathbf{x} \oplus \mathbf{y} \in \mathcal{C}$
1: Sort $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ on their first k bits.
2: **for all** $0 \le i, j < 2^\ell$ **do**
3: **for** $i2^{k-\ell} \le u < (i+1)2^{k-\ell}$ **do**
4: $\mathbf{u} \leftarrow$ BinaryRepresentation(u)
5: Initialise a hash table T with the entries of $\mathcal{C}^{[\mathbf{u}]}$
6: **for** $j2^{k-\ell} \le v < (j+1)2^{k-\ell}$ **do**
7: $\mathbf{v} \leftarrow$ BinaryRepresentation(v)
8: **for all** $\mathbf{x} \in \mathcal{A}^{[\mathbf{v}]}$ and all $\mathbf{y} \in \mathcal{B}^{[\mathbf{u}\oplus\mathbf{v}]}$ **do**
9: **if** TestMembership$(\mathbf{x} \oplus \mathbf{y}, T) =$ `True` **then**
10: **return** $(\mathbf{x}, \mathbf{y})$

Parallelisation is easy: all iterations of the outer loops on i, j can be done concurrently on several machines. The three-level loop structure guarantees that one iteration of Steps 3 to 10 only needs to read $2^{n-\ell}$ entries of each list. ℓ is ideally chosen so that the corresponding portions of both $\mathcal{A}$ and $\mathcal{B}$ fit in L3 cache. 2^{18} entries of $\mathcal{A}$ and $\mathcal{B}$ fit in 2 Megabytes of L3 cache, and a few minutes will be necessary to process the corresponding 2^{36} pairs. Memory bandwidth is not a problem, and the quadratic algorithm scales well on multi-core machines. The algorithm can also be run on machines with limited memory. Algorithm 24 runs at 10–11 cycles per pair processed on a "Haswell" Core i5 CPU.

Implementation of Algorithm 20. The efficient implementation of joins is a non-trivial problem, which has been studied for a while by the database community. We used a reasonably efficient sort-join, but it turns out that hash-joins have the favours of the experts (for now).

The longest operation of each iteration is the sorting step of Algorithm 20, which accounts for 50% of the total running-time. We use three passes of radix-256 sort for the case of $n = 96$ (where the value of the k parameter is 24). The out-of-place version is 2–3× faster than the in-place version (but requires twice more memory). We use the M4RI library [AB12] to compute the PLUQ factorisation. To solve our $n = 96$ problem, a single iteration of Algorithm 20 runs at 75 CPU cycles per list item processed on the same "Haswell" Core i5 CPU.

It is easy to parallelise the loop on i (each iteration takes slightly less than 1s for $n = 96$). The problem is that both the full $\mathcal{A}$ and $\mathcal{B}$ must fit in RAM, as they are entirely read in each iteration. When the lists only have 2^{24} entries (as it is the case for $n = 96$), they only require 256 Megabytes. On a multi-core machine, one iteration can be run concurrently per core. One potential problem is that this may saturate the memory bandwidth: each iteration reads 1.25 Gigabytes from memory in about 1s, so on a large chip with 18 cores/36 threads, up to 22.5 Gigabytes/s of memory bandwidth would be required.

A further problem is that, for larger values of n, it becomes impossible to keep many independent copies of the lists in memory. In that case, the sorting and merging operations themselves have to be parallelised, and this is a bit less obvious. If a single copy of the lists does not fit in memory, a distributed sort/merge will be required, and it is likely that the communication overhead will make this less efficient than the quadratic algorithm.

Distributed computation. It is easy to run these algorithms on clusters of loosely-connected machines. We used a simple master-slave approach, in which the iterations to be parallelised are numbered. When a slave node becomes ready, it requests an iteration number to the master node. When it has finished processing it, it sends the partial solutions found in the iteration back to the master. The master stores a log of all accomplished iterations. Communications were handled by the ∅MQ library [Hin13]. We actually ran the algorithms on several hundred cores concurrently.

8.4.3 Result and Future Work

Consider the three ASCII strings:

$$x = \texttt{FOO-0x0000B70947F064A1}$$
$$y = \texttt{BAR-0x000013F9e450DF0b}$$
$$z = \texttt{FOOBAR-0x0000e9B2CF21D70a}$$

We can check that:

```
  SHA256(x) = 000000a9 4fc67b35 beed47fc addb8253 911bb4fa ecaee2d9 f46f7f10 5c7ba78c
^ SHA256(y) = 00000017 d29b29eb a0ef2522 db22d0cc 5d48d2f9 36149197 6430685b 1266ee76
^ SHA256(z) = 000000be 9d5d52de 1e0262de e51c1119 edff081d 868fe419 879932ab bbcfe66e
================================================================================
            = 00000000 00000000 00000000 93e54386 21ac6e1e 5c359757 17c625e0 f5d2af94
```

After completing this 96-bit 3XOR, we embarked on a project to compute a 128-bit 3XOR on SHA256. To this end, we found a way to use off-the-shelf bitcoin miners, which evaluate SHA256 about one million times faster than a CPU core. Bitcoin miners naturally produce bit-strings whose SHA256 is zero on (at least) the 32 most significant bits. We plan to accumulate three lists of 2^{32} entries, and then to use Algorithm 20 to find 2^{32} partial solutions on 64 bits, amongst which one should lead to a 128-bit 3XOR (thanks to the extra 32 bits of clamping).

Chapter 9

Of an Adaptation of the Baran-Demaine-Pătraşcu Algorithm to the 3XOR Problem

In this chapter, we revisit an algorithm that was introduced by Baran, Demaine and Pătraşcu. After briefly recalling how this works in the 3SUM case, we present an adaptation to the 3XOR problem. We show that this algorithm has an asymptotic complexity of $\mathcal{O}\left(2^{2n/3}\log^2 n/n^2\right)$. However, as we discuss, this algorithm is not practical. This consists in the second contribution of [BDF18].

9.1 BDP Algorithm for the 3SUM Problem

Initially this algorithm was designed for the 3SUM problem over $(\mathbb{Z},+)$, where the size of the three lists is bounded by some parameter N. The goal was to determine whether a 3SUM exists in the input lists or not and to return it, when appropriate. Formally the problem considered by the authors of [BDP05] can be stated as follows:

Problem 9.1. Given three lists $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ of N integers each, search for a triplet $(a,b,c)\in\mathcal{A}\times\mathcal{B}\times\mathcal{C}$, such that $a+b=c$ if it exists.

The idea of BDP is to first dispatch the elements into buckets, that represent sub-instances, and solve only the sub-instance that may contain a solution. For each of these sub-instances a constant time preliminary test is performed beforehand. If the test passes, then a full search is required. Else, it is certain that no solution will be found in this sub-instance, and there is no need for further investigation. Our 3XOR adaptation is based on the same idea.

For the record, we give a more detailed version of BDP 3SUM algorithm below:

1. Fix a parameter $m=\mathcal{O}\left(\log N/\log\log N\right)$, hash the three lists separately and dispatch them into n/m buckets, using a hash function h_1 which returns a k-bit value and satisfies the following linearity property: if $x+y=z$, then $h_1(z)=h_1(x+y)\in h_1(x)+h_1(y)+\{0,1\}$ mod 2^k.

2. The i-th bucket of each list is denoted by $\mathcal{A}_i^b,\mathcal{B}_i^b$, and $\mathcal{C}_i^b$. Considering a pair of buckets $(\mathcal{A}_i^b,\mathcal{B}_j^b)$, there exist exactly two buckets $\mathcal{C}_{i_1}^b,\mathcal{C}_{i_2}^b$ such that for all $(x,y)\in\mathcal{A}_i^b\times\mathcal{B}_i^b$, if $x+y\in\mathcal{C}$, then $x+y\in\mathcal{C}_{i_1}^b$ or $x+y\in\mathcal{C}_{i_2}^b$. This is due to the linearity property of h_1.

3. For all $(\mathcal{A}_i^b,\mathcal{B}_j^b,\mathcal{C}_{i_1}^b\cup\mathcal{C}_{i_2}^b)$ that may contain a solution, solve the sub-instance using the following procedure:

   (a) If the buckets consist of more than $3m$ elements, treat the exceeding elements independently, so that all buckets have, after that, at most $3m$ elements. This is basically doing $\mathcal{O}\left(N/m\right)$ additional search.

(b) The problem consists now in solving sub-instances $(\mathcal{A}_i^b, \mathcal{B}_j^b, \mathcal{C}_{i_1}^b \cup \mathcal{C}_{i_2}^b)$ so that the number of elements in the lists is bounded by $\mathcal{O}(m)$. The authors of [BDP05] use a second level of hashing, to pack all the elements of a bucket on a single word. To do so, they use a linear hash function h_2 which returns an s-bit value with $s = \Theta(\log w)$. For the $\mathcal{C}$ buckets they pack two elements per values, $h_2(z)$ and $h_2(z) - 1$. They check if there is a triplet $(h_2(x), h_2(y), h_2(z))$ such that $h_2(x) + h_2(y) \in h_2(z) + \{0, 1\} \mod 2^s$, in constant time, by looking-up at a pre-computed table. If such a triplet does not exist, they know there is no solution for sure. However, if such a triplet exists, they have to solve the full instance to check if (x, y, z) is actually a solution.

If $\log(N) = \mathcal{O}(w)$, BDP Algorithm has a time complexity of $\mathcal{O}\left(N^2/\frac{\log^2(N)}{(\log\log N)^2}\right)$, assuming the lists are sorted. This result was actually proved in [BDP05]. We do not demonstrate it over again, as the proof is very similar to the one that will be given in the Section 9.3, concerning our own algorithm.

9.2 Our Adaptation of BDP

We consider three lists $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ of $2^{n/3}$ n-bit elements (this can be generalised to different sizes of input lists). The adaptation described here is asymptotically $n/\log^2 n$ times faster than the algorithm we describe in Chapter 8, but is hardly practical.

9.2.1 High Level Description of the Procedure

Let $\mathbf{u}$ be a bit-string of length smaller than n. We recall that $\mathcal{A}^{[\mathbf{u}]}$ represents the sublist of $\mathcal{A}$ that starts with the prefix $\mathbf{u}$. The high-level idea of the procedure is the following:

1. Dispatch $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ in buckets according to their first k bits.
2. For each triplets of buckets $(\mathcal{A}^{[\mathbf{u}]}, \mathcal{B}^{[\mathbf{v}]}, \mathcal{C}^{[\mathbf{u}\oplus\mathbf{v}]})$, perform a preliminary *constant time* test. The test returns `False` when it is certain that no solution will be found in this triplet.
3. For each triplet that has not been rejected, use the quadratic algorithm on this reduced instance.
4. A bucket may contain more elements than desired for the preliminary test. In this case, these exceeding elements are to be treated separately.

Basically for each triplet of buckets, this test consists in checking if there is a partial collision on s bit, with s a well-chosen parameter. To do that, we have previously pre-computed a table T. Each triplet is associated to a specific bit-string $\mathbf{t}$ of T so that the $T[\mathbf{t}] = 0$ implies there is no solution in this sub-instance for sure. We describe this test in more details below.

9.2.2 Preliminary Test

Let $w = \Theta(n)$ denote the size of a word. Let s be such that $s = \kappa_1 \cdot \log(w)$, with κ_1 a constant to be determined. Let $k < n/3$ be the parameter defined above, and $P \geq 1$ a parameter to be determined, such that $3Ps \leq w$.

Let T be a table of $2^{3 \cdot P \cdot s}$ elements. For each index $\mathbf{t}$, where t can be represented as in Figure 9.1:

$$T[\mathbf{t}] = \begin{cases} 1 & \text{if } \exists\ i, j, \ell \text{ s.t. } \mathbf{a}[i] \oplus \mathbf{b}[j] \oplus \mathbf{c}[\ell] = 0 \\ 0 & \text{otherwise.} \end{cases} \tag{9.1}$$

Then, for all three sets of s-bit vectors, that contain each at most P elements, one can check if a solution to the 3XOR problem exists by a constant time look-up in T.

By construction, $|T| = 2^{3 \cdot P \cdot s}$. We do not want the additional space to exceed the space required to store the lists. We set:

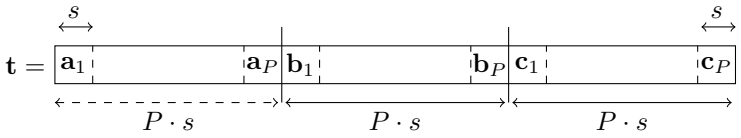


Figure 9.1 – Representation of an index $\mathbf{t}$ of the table T.

$$3 \cdot P \cdot s = \min(1/3 \cdot n, w). \tag{9.2}$$

For all $\mathbf{x} \in \mathbb{F}_2^n$, let h be the function that returns s consecutive bits of $\mathbf{x}$ starting from bit k, or formally $h : \mathbf{x} \to \mathbf{x}[k..k+s]$. For all $0 \leq u < 2^k$, we denote $\mathbf{u}$ the binary representation of u such that: $\mathbf{u}$ is a bit-string of length 2^k, $\mathbf{u}[0]$ is the least significant bit of u (i.e. $\mathbf{u}[0] = u \mod 2$). We denote by $h(\mathcal{A}^{[\mathbf{u}]})$ the list:

$$h(\mathcal{A}^{[\mathbf{u}]}) = \{\mathbf{a} = h(\mathbf{x}), \text{ s.t. } \mathbf{x} \in \mathcal{A}^{[\mathbf{u}]}\}.$$

We define $h(\mathcal{B}^{[\mathbf{u}]})$ and $h(\mathcal{C}^{[\mathbf{u}]})$ accordingly for all values of u.

For all $0 \leq v, w < 2^k$, we accordingly denote $\mathbf{v}$ and $\mathbf{w}$, the binary representations of v and w. A triplet of buckets $(\mathcal{A}^{[\mathbf{u}]}, \mathcal{B}^{[\mathbf{v}]}, \mathcal{C}^{[\mathbf{w}]})$ may contain a solution of the 3XOR problem, *only if* $\mathbf{w} = \mathbf{u} \oplus \mathbf{v}$. Now, for each of such triplets $(\mathcal{A}^{[\mathbf{u}]}, \mathcal{B}^{[\mathbf{b}]}, \mathcal{C}^{[\mathbf{u} \oplus \mathbf{v}]})$, the idea of the algorithm is to check if a solution may exist or not by first checking if there is a solution to the instance $(h(\mathcal{A}^{[\mathbf{u}]}), h(\mathcal{B}^{[\mathbf{v}]}), h(\mathcal{C}^{[\mathbf{u} \oplus \mathbf{v}]}))$. In other words, we check whether there is a partial solution over s bits.

If this test fails, we know for sure that there is no solution and we can move on to the next triplet of buckets, without having to perform any other operation. On the other hand, if the test passes, we only know that there is a solution to the 3XOR problem with probability $(1/2)^{n-k-s}$.

Solving each of the 2^{2k} small instances $(h(\mathcal{A}^{[\mathbf{u}]}), h(\mathcal{B}^{[\mathbf{v}]}), h(\mathcal{C}^{[\mathbf{u} \oplus \mathbf{v}]}))$ can be done in constant time, using table T.

Remark 1. One can notice that if a bucket contains more than P vectors then these additional vectors are not considered during the preliminary test. Thus, we have to treat them separately afterward. However, if we choose P to be equal to $\kappa_2 \cdot m$, with a well chosen κ_2, we can ensure that the number of buckets that will contain more than P elements is very small.

Using Chernoff bounds, we figured out that, if we choose κ_2 to be around 4.162, the probability that a bucket will contain more than P elements will be $2^{-4 \cdot m}$.

Remark 2. Taking this into account, we have:

$$m = 2^{n/3-k} = \Theta(n/\log n). \tag{9.3}$$

This is a direct consequence of the definition of P and s and of Equation 9.2.

9.2.3 Full Description of the Procedure

Let k be a parameter such that $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are dispatched into 2^k buckets according to their first k bits. For each small set $\mathcal{A}^{[\mathbf{u}]}$ (resp. $\mathcal{B}^{[\mathbf{v}]}$ and $\mathcal{C}^{[\mathbf{u} \oplus \mathbf{v}]}$), we keep a Ps-bit vector $\mathbf{t}_A[\mathbf{u}]$ (resp. $\mathbf{t}_B[\mathbf{v}]$ and $\mathbf{t}_C[\mathbf{u} \oplus \mathbf{v}]$), in which the elements of $h(\mathcal{A}^{[\mathbf{u}]})$ (resp. $h(\mathcal{B}^{[\mathbf{v}]})$ and $h(\mathcal{C}^{[\mathbf{u} \oplus \mathbf{v}]})$) are stored. With this construction, assuming that the number of elements of each bucket does not exceed P, a solution to the instance $(h(\mathcal{A}^{[\mathbf{u}]}), h(\mathcal{B}^{[\mathbf{v}]}), h(\mathcal{C}^{\mathbf{u} \oplus \mathbf{v}}))$ exists if and only if $T[\mathbf{t}_A[\mathbf{u}] \mid \mathbf{t}_B[\mathbf{v}] \mid \mathbf{t}_C[\mathbf{u} \oplus \mathbf{v}]] = 1$. The procedure described in Algorithm 25 is used to initialise the tables $\mathbf{t}_A$, $\mathbf{t}_B$ and $\mathbf{t}_C$.

If there are exceeding elements in the buckets they are to be treated independently. All in all, this leads to Algorithm 26:

Remark 3. After us, the authors of [DSW18] have proposed another adaptation of the BDP algorithm for the 3XOR problem. Their algorithm is designed to solve the problem for the case where the lists are sets of elements that are non necessarily random. As such, they had to use linear hash functions for the dispatch step, and for the preliminary test. Their version is then closer to the original BDP algorithm than ours.

Algorithm 25 Initialise vectors.

Require: Lists $\mathcal{A}$ dispatched into 2^k buckets.
Ensure: The table $\mathbf{t}_A$
1: **for** $0 \leq u < 2^k$ **do**
2: Set $\mathbf{t}_A[\mathbf{u}] \leftarrow \perp$.
3: **for** $0 \leq i < \max(P, N_{\mathcal{A}}^{\mathbf{u}})$ **do**
4: $\mathbf{t}_A[\mathbf{u}] \leftarrow v_A[\mathbf{u}] \mid h\left(A^{[\mathbf{u}]}[i]\right)$

Algorithm 26 Adaptation of BDP algorithm to our problem.

Require: Three lists $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ and the precomputed table T
Ensure: All couples $(\mathcal{A}[i], \mathcal{B}[j])$ such that $\mathcal{A}[i] \oplus \mathcal{B}[j]$ is an element of $\mathcal{C}$
1: Dispatch $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$, according to their first k bits
2: Use Algorithm 25 to create the tables $\mathbf{t}_A$, $\mathbf{t}_B$ and $\mathbf{t}_C$
3: **for** $0 \leq u, v < 2^k$ **do**
4: $\mathbf{t} \leftarrow \mathbf{t}_A[\mathbf{v}] | \mathbf{t}_B[\mathbf{u} \oplus \mathbf{v}] | \mathbf{t}_C[\mathbf{u}]$
5: **if** $T[\mathbf{t}] = 1$ **then** ▷ There may be a solution in this sub-instance
6: **for all** couples $(\mathcal{A}[i], \mathcal{B}[j])$ in $\mathcal{A}^{[\mathbf{u}]} \times \mathcal{B}^{[\mathbf{u} \oplus \mathbf{v}]}$ **do**
7: **if** $\mathcal{A}[i] \oplus \mathcal{B}[j]$ is in $\mathcal{C}^{[\mathbf{u}]}$ **then**
8: **return** $(\mathcal{A}[i], \mathcal{B}[j])$
9: **else if** $A^{[\mathbf{v}]} > P$ **then** ▷ There is more than P elements of $\mathcal{A}$ that start by $\mathbf{v}$
10: **for all** couples $(\mathcal{A}[i], \mathcal{B}[j])$ in $\mathcal{A}^{[\mathbf{v}]}[P..] \times \mathcal{B}^{[\mathbf{u} \oplus \mathbf{v}]}$ **do**
11: **if** $\mathcal{A}[i] \oplus \mathcal{B}[j]$ is in $\mathcal{C}^{[\mathbf{u}]}$ **then**
12: **return** $(\mathcal{A}[i], \mathcal{B}[j])$
13: **else if** $B^{[\mathbf{u} \oplus \mathbf{v}]} > P$ **then** ▷ There is more than P elements of B that start by $\mathbf{u} \oplus \mathbf{v}$
14: **for all** couples $(\mathcal{A}[i], \mathcal{B}[j])$ in $\mathcal{A}^{[\mathbf{v}]} \times \mathcal{B}^{[\mathbf{u} \oplus \mathbf{v}]}[P..]$ **do**
15: **if** $\mathcal{A}[i] \oplus \mathcal{B}[j]$ is in $\mathcal{C}^{\mathbf{u}}$ **then**
16: **return** $(\mathcal{A}[i], \mathcal{B}[j])$
17: **else if** $C^{[\mathbf{u}]} > P$ **then** ▷ There is more than P elements of C that start by $\mathbf{u}$
18: **for all** couples $(\mathcal{A}[i], \mathcal{B}[j])$ in $\mathcal{A}^{[\mathbf{v}]} \times \mathcal{B}^{[\mathbf{u} \oplus \mathbf{v}]}$ **do**
19: **if** $\mathcal{A}[i] \oplus \mathcal{B}[j]$ is in $\mathcal{C}^{\mathbf{u}}$ **then**
20: **return** $(\mathcal{A}[i], \mathcal{B}[j])$

9.3 Complexity Analysis and Discussion

9.3.1 Complexity Analysis

Theorem 9.1. *With input lists of size $2^{n/3}$, the time complexity of the BDP algorithm in our model is*

$$\mathcal{O}\left(\frac{2^{2n/3}\cdot\log^2(n)}{n^2}+N_{test}\cdot\frac{n^2}{\log^2 n}\right),$$

where N_{test} is the number of triplets that pass the test. The expected value of N_{test} is:

$$\mathbb{E}[N_{test}]=\left(\frac{2^{2\cdot n/3}}{m}-1\right)\cdot\left(1-\left(1-\frac{1}{w^{\kappa_1}}\cdot\left(1-\frac{1}{w^{\kappa_1}}\right)\right)^{\Theta(n^3/\log^3(w))}\right)+1.$$

When n grows to infinity, N_{test} is equivalent to 1. Thus, the asymptotic complexity of this algorithm is

$$\mathcal{O}\left(\frac{2^{2n/3}\cdot\log^2 n}{n^2}\right).$$

Before we prove this theorem, we need first to introduce some intermediate results.

Lemma 9.2. *Let us denote by N_{test} the number of triplets $(\mathcal{A}^{[\mathbf{u}]},\mathcal{B}^{[\mathbf{v}]},\mathcal{C}^{[\mathbf{u}\oplus\mathbf{v}]})$. The time complexity of our adaptation of the BDP Algorithm is:*

$$T_{BDP}=\mathcal{O}\left(\frac{2^{2n/3}\cdot\log^2 n}{n^2}+N_{test}\cdot\frac{n^2}{\log^2 n}\right).$$

Proof. The time complexity of the full procedure is:

$$T_{BDP}=T_{dispatch}+2^{2\cdot k}\cdot T_{test}+N_{test}\cdot T_{solve}+T_{additional},$$

where: $T_{dispatch}$ is the time it takes to dispatch the elements of the three lists according to their k first bits. As discussed in Section 6.3,

$$T_{dispatch}=\mathcal{O}\left(N_{\mathcal{A}}+N_{\mathcal{B}}+N_{\mathcal{C}}\right)=\mathcal{O}\left(2^{n/3}\right).$$

T_{test} is the time complexity of testing one sub-instance $(\mathcal{A}^{[\mathbf{u}]},\mathcal{B}^{[\mathbf{v}]},\mathcal{C}^{[\mathbf{u}\oplus\mathbf{v}]})$. This is constant time, by a lookup in a precomputed table. We will have to perform this test for all triplets $(A^{[\mathbf{u}]},B^{[\mathbf{v}]},C^{[\mathbf{u}\oplus\mathbf{v}]})$, that means $2^{2\cdot k}$ times.

T_{solve} is the time required to solve one small instance $(\mathcal{A}^{[\mathbf{u}]},\mathcal{B}^{[\mathbf{v}]},\mathcal{C}^{[\mathbf{u}\oplus\mathbf{v}]})$. If we denote by $N_{\mathcal{A}}^{[\mathbf{u}]}$ the number of elements in $\mathcal{A}^{[\mathbf{u}]}$, and accordingly $N_{\mathcal{B}}^{[\mathbf{v}]}$ and $N_{\mathcal{C}}^{[\mathbf{u}\oplus\mathbf{v}]}$, the number of elements in $\mathcal{B}^{[\mathbf{v}]}$ and $\mathcal{C}^{[\mathbf{u}\oplus\mathbf{v}]}$.

$$T_{solve}=\mathcal{O}\left((N_{\mathcal{C}}^{[\mathbf{u}\oplus\mathbf{v}]}+N_{\mathcal{A}}^{[\mathbf{u}]}\cdot N_{\mathcal{B}}^{[\mathbf{v}]})\right).$$

Furthermore, as $N_{\mathcal{C}}^{[\mathbf{u}\oplus\mathbf{v}]}$, $N_{\mathcal{A}}^{[\mathbf{u}]}$ and $N_{\mathcal{B}}^{[\mathbf{v}]}$ are all $\mathcal{O}(m)$,

$$T_{solve}=\mathcal{O}\left(m^2\right).$$

$T_{additional}$ is the time required for additional searches, when a bucket contains more than P elements. As we have chosen P so that this event is very unlikely, this can be neglected.

All in all we obtain:

$$T_{BDP}=\mathcal{O}\left(2^{n/3}+2^{2\cdot k}+N_{test}\cdot m^2\right).$$

In addition, we have $2^k=2^{n/3}/m$ and $m=\Theta(n/\log(n))$. This is enough to conclude the proof of this lemma. □

Lemma 9.3. *The expected number of triplets that pass the test is:*

$$\mathbb{E}[N_{test}] = \left(2^{2\cdot k} - 1\right) \cdot \left(1 - \left(1 - \frac{1}{w^{\kappa_1}} \cdot \left(1 - \frac{1}{w^{n-\kappa_1}}\right)\right)^{\Theta(n^3/\log^3(w))}\right) + 1.$$

Proof. Let us consider a sub-instance $(\mathcal{A}^{[\mathbf{u}]}, \mathcal{B}^{[\mathbf{v}]}, \mathcal{C}^{[\mathbf{u}\oplus\mathbf{v}]})$. Let $(\mathbf{a}, \mathbf{b}, \mathbf{c}) \in \mathcal{A}^{[\mathbf{u}]} \times \mathcal{B}^{[\mathbf{v}]} \times \mathcal{C}^{[\mathbf{u}\oplus\mathbf{v}]}$. Recalling that $h(\mathbf{a})$ (resp. $h(\mathbf{b})$, $h(\mathbf{c})$) represents s arbitrarily chosen bits of $\mathbf{a}$ (resp. $\mathbf{b}$, $\mathbf{c}$), that are uniformly random, the probability π_{FP} that $h(\mathbf{a}) \oplus h(\mathbf{b}) = h(\mathbf{c})$, knowing that $\mathbf{a} \oplus \mathbf{b} \neq \mathbf{c}$ with $(\mathbf{a}, \mathbf{b}, \mathbf{c}) \in \mathcal{A}^{[u]} \times \mathcal{B}^{[\mathbf{v}]} \times \mathcal{C}^{[\mathbf{u}\oplus\mathbf{v}]}$ is:

$$\pi_{FP} = \frac{1}{2^s} \cdot \left(1 - \frac{1}{2^{n-k-s}}\right).$$

Using Chernoff bounds, we know that the size of the small lists $h(\mathcal{A}^{[\mathbf{u}]})$, $h(\mathcal{B}^{[\mathbf{v}]})$ and $h(\mathcal{C}^{[\mathbf{u}\oplus\mathbf{v}]})$ are $\Theta(m)$ with high probability. Then, we can estimate the probability that a given instance is a false positive to the preliminary test is:

$$\mathbb{P}[\text{False positive}] = \left(1 - (1 - \pi_{FP})^{\Theta(m^3)}\right),$$

with $m = \Theta(n/\log(w))$.

Furthermore, given the size of the list, we expect only to find one solution. Thus, we expect only one true positive in the preliminary test. Then, there should be only *one* triplet among 2^{2k} that contains a solution. The $2^{2k} - 1$ others are expected to contain none.

From here, we can estimate that N_{test} is:

$$\mathbb{E}[N_{test}] = (2^{2k} - 1) \cdot \left(1 - (1 - \pi_{FP})^{\Theta(n^3/\log^3(w))}\right) + 1.$$

□

Lemma 9.4. *If $w(n) \in \Theta(n)$, and $\beta > 0$ is a constant, then choosing $\kappa_1 = 3$, ensures that $N_{test} \underset{0^+}{\sim} 1$.*

Proof. We set $k(n) = n/3 - \Theta(\log(n/w(n)))$. In particular $n - k(n) = 2n/3 + \Theta(\log(n/w(n)))$. We also set:

$$F(n) = \left(1 - \frac{1}{w(n)^{\kappa_1}}\left(1 - \frac{1}{w(n)^{n-k(n)-\kappa_1}}\right)\right)^{\beta\cdot n^3/\log^3(w(n))}$$

and then

$$\ln(F(n)) = \beta \cdot \frac{n^3}{\log^3(w(n))} \cdot \ln\left(1 - \frac{1}{w(n)^{\kappa_1}}\left(1 - \frac{1}{w(n)^{n-k(n)-\kappa_1}}\right)\right).$$

We first prove that the ln factor is asymptotically equal to $-1/w(n)^{\kappa_1} + \mathcal{O}\left(1/n^{2\kappa_1}\right)$. At this end, we need to consider the function:

$$g(n) = w(n)^{2n/3+\gamma\log(n/\log(w(n)))},$$

for some constant $\gamma > 0$.

$$\ln(g(n)) = \left(\frac{2n}{3} + \gamma\log\frac{n}{\log w(n)}\right)\log w(n).$$

As $w(n) = \Theta(n)$, we have $\lim_{n\to+\infty}\ln(g(n)) = +\infty$. It follows that:

$$\lim_{n\to+\infty}\frac{1}{w(n)^{n-k(n)}} = 0,$$

hence, as γ, κ are positive,

$$\lim_{n\to+\infty} \frac{1}{w(n)^{\kappa_1}} - \frac{1}{w(n)^n} = 0.$$

Thus,

$$\ln\left(1 - \left(\frac{1}{w(n)^{\kappa_1}} - \frac{1}{w(n)^{n-k(n)}}\right)\right) \underset{+\infty}{=} -\left(\frac{1}{w(n)^{\kappa_1}} - \frac{1}{w(n)^{n-k(n)}}\right) + \mathcal{O}\left(\frac{1}{w(n)^{2\kappa_1}} + \frac{1}{w(n)^{2(n-k(n)}}\right)$$
$$\underset{+\infty}{=} -\frac{1}{w(n)^{\kappa_1}} + \mathcal{O}\left(\frac{1}{n^{2\kappa_1}}\right).$$

It follows that:

$$\ln(F(n)) \underset{+\infty}{=} -\beta \cdot \frac{n^3}{\log^3(w(n))} \cdot \frac{1}{w(n)^{\kappa_1}} + \mathcal{O}\left(\frac{1}{n^3 \log^3(n)}\right).$$

As $w(n) \in \Theta(n)$, there are two constants β_1, β_2 strictly greater than 0, and a certain n_0 such that for all $n \geq n_0$,

$$-\beta_1 \cdot \frac{n^{3-\kappa_1}}{\log^3(w(n))} + \mathcal{O}\left(\frac{n^{3-2\kappa_1}}{\log^3(n)}\right) \leq \ln(F(n)) \leq -\beta_2 \cdot \frac{1}{\log^3(w(n))} + \mathcal{O}\left(\frac{n^{3-2\kappa_1}}{\log^3(n)}\right).$$

Choosing $\kappa_1 \geq 3$ will ensure $\ln(F(n)) \to 0$ when n grows up to infinity. We choose then $\kappa_1 = 3$. From here we can deduce the following equation:

$$\lim_{n\to+\infty} \left(1 - \frac{1}{w(n)^3}\left(1 - \frac{1}{w(n)^{n-3}}\right)\right)^{\beta \cdot n^3/\log^3(w(n))} = 1, \tag{9.4}$$

or in a simpler way:

$$(1 - 1/\pi_{FP})^{\Theta(n^3/\log^3(w))} = 1$$

The proof Lemma 9.4, is trivial from here. □

Theorem 9.1 is a direct consequence of all these results.

9.3.2 Theory vs Practice

While it is asymptotically more efficient than any other algorithms introduced here, the BDP algorithm fails in practice for reasonable values of n (e.g. $n = 96$). In fact, for $n = 96$, and $w = 64$, from Equation 9.2, we have:

$$3 \cdot P \cdot s = 3 \cdot \kappa_1 \log(w) \cdot \kappa_2 m = 32.$$

Choosing $\kappa_1 = 3$ as in Lemma 9.4, and $\kappa_2 = 4.162$, as in Remark 1, we obtain:

$$m = \frac{32}{3 \cdot 3 \cdot 4.162 \cdot \log 64} = \frac{32}{224.748} \simeq 0.142.$$

The best we could hope, in that case, is to process "batches" composed of... a tenth of an entry!

Remark 1. We have been told that this comparison is not fair, as in practice one will not use the Chernoff bounds, and would rather choose a smaller value of P and decide to treat exceeding elements independently, hoping that there will be only a few of them. We claim that even in this case, this algorithm is still impractical.

Indeed, assume that we choose to take $\kappa_2 = 1$ so that P will be exactly m. Then, we obtain $m \simeq 0.593$, which is still less than one element.

It would be interesting to know for which value of n this method would be worth trying. We have not thought about the question much, but it seems quite hard to answer. Indeed, what appears to be a limiting factor here is the size of the machine word, which is a constant 64, and do not grow with n as in the RAM model. Increasing the value of n on a 64-bit machine, will then probably lead to even worst results.

Conclusion

Here, we discussed about the 3XOR problem. We revisited and discussed previous algorithms, from the most naive quadratic one, to improvements of Wagner's method, by Joux in one hand and Nikolić and Sasaki in the other. We also took into account several problems such as memory consumption, and query complexity. We presented Bernstein "clamping trick" to reduce the first one, at the cost of increasing the later. We claimed that, when querying the oracles costs a lot of time, it may be interesting to reduce slightly the number of queries, at the cost of increasing the space and the time complexity.

Once we had all these practical considerations in mind, we presented a new algorithm, which generalises the idea of Joux to any size of input lists. The algorithm we presented in Section 8 can be applied to recover all solutions to the 3XOR problem and not only one. Whenever only one solution is required, we can use the clamping trick to reduce both the running time and the memory consumption of our algorithm. We showed that, in this case, for the same number of queries, our algorithm is asymptotically faster than the Nikolić-Sasaki one, and requires to manipulate much less data. We also presented some possible improvements of our algorithm which aimed to reduce the number of iterations of the whole procedure. Unfortunately, we only managed to gain a constant improvement in theory, and we have not yet tested how these methods would behave in practice. We also explained our experiments and gave some details about the implementation. In practice, our algorithm ran about 3 times faster than our implementation of the quadratic algorithm for $n = 96$.

In the last chapter, we presented an older algorithm due to Baran, Demain and Pătraşcu that solves the 3SUM problem over the integers. After recalling the general idea of the method, we transposed it to the case of the 3XOR problem. This algorithm has the best asymptotic complexity, but is impractical.

At the aim of understanding the time-space-query tradeoffs and other practical considerations for larger values of n better, we have embarked on a project to compute a 128-bit 3XOR of SHA256. At this aim, we have acquired an Antminer S7 bitcoin-miner, to generate the lists and perform the clamping. With the default settings the clamping was done on 36 bits of the output. After tuning the bitcoin-miner, we reduced this clamping to 33 bits. This allows us to reduce the time required for generating the lists, at the cost of increasing their size and thus the time complexity of the procedure. For now on we are able to generate enough samples to solve a 3XOR over 120-bit of SHA256, and we keep-on mining.

There are still open questions. First of all: is there a setting where the 3XOR problem can be solved faster than $\mathcal{O}\left(2^{n/2-o(1)}\right)$? Although it seems very unlikely due to conjecture upper bounds for the 3SUM problem, nothing is clear yet. In fact, we do not even know for sure if the 3XOR problem is indeed as hard as the 3SUM problem.

Another interesting point would be to know whether our algorithm can be transposed into the 3SUM setting. In this case, one will encounter some issues due to carries, that will introduce errors. Hopefully these errors will be small and can be dealt with efficiently, enumerating the solutions to an approximate-3SUM instance.

Finally, it could also be interesting to see if our algorithm could be applied to other kXOR instance. when $k = 2^\alpha \cdot 3$, the instance would naturally reduce itself to a 3-XOR using Wagner k-tree algorithm. When $k = 3^\beta$, we may possibly generalise the notion of joined list, and compute lists of the type:

$$\mathcal{L}_i \bowtie_\ell \mathcal{L}_j \bowtie_\alpha \mathcal{L}_k = \{(\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k) \in \mathcal{L}_i \times \mathcal{L}_j \times \mathcal{L}_k | \mathbf{x}_i[..\ell] \oplus \mathbf{x}_j[..\ell] \oplus \mathbf{x}_k[..\ell] = 0\}.$$

With our algorithm it is possible to compute all triplets which satisfy this condition, but the required time will be almost quadratic in the size of the input lists. On the other hand, this would require to compute and store less lists. And so, it is not clear whether there is something that could be gained compared to Wagner's algorithm. It would also be interesting to know whether a similar method can be applied to other values of k (e.g. $5, 7$). We have not given much thought to the matter so far, but we do not see why this would not work. However, once again, we do not know if the gain compared to Wagner's algorithm would be worthwhile.

PART 3

LWE-Related Problems and Applications

Introduction

"From error to error, one discovers the entire truth."

– Sigmund Freud –

Learning With Errors

Regev introduced the *learning with error* (LWE) problem in 2005 [Reg05]. Given an m-by-n matrix over $\mathbb{Z}$, with $m > n$, and a vector $\mathbf{c}$ of $\mathbb{Z}^m$, this problem basically consists in solving the noisy system

$$A\mathbf{s} + \mathbf{e} = \mathbf{c} \mod q,$$

where $\mathbf{e}$ is a short error vector, whose coefficients are sampled according to a specific distribution over $\mathbb{Z}_q$ (typically a discrete Gaussian distribution). Regev showed that, for suitable parameters, this problem is as hard as worst-case lattice problems, in the quantum setting. A few years later, Brakerski et al. [BLP+13] showed a similar reduction in the classical setting. These results are a cornerstone of modern lattice-based cryptography, which is to a large extend based on LWE or related problems.

Variants of the Problem.

Adding structure and different distributions. Many variants of the LWE have been developed in the last few years, mostly with the goal of improving the efficiency of Lattice-based cryptography. An important part of these works has been to extend LWE to different kinds of structure. This started with the introduction in 2009 of Polynomial-LWE (PLWE) by Stehlé et al. [SSTX09]. One year later, the celebrated ring variant (RLWE) of LWE was proposed by Lyubashevsky, Peikert and Regev [LPR10, LPR13]. Langlois and Stehlé introduced the module variant of LWE (MLWE) in 2015 [LS15], and finally Roşca et al. [RSSS17] introduced the Middle-Product-LWE variant (MPLWE). A recent study from Roşca, Stehlé and Wallet presents new reductions results concerning these different problems [RSW18].

Another important line of work concerning the LWE problem is devoted to the analysis of the problem with different distributions. Typically, papers have studied the case where the error distribution is a non-Gaussian one and/or very small [AG11, MP13, DMQ13], other have studied the case where the secret $\mathbf{s}$ is sampled from a non-uniform distribution [ACPS09, BLP+13, AFFP14, BG14, BGPW16, Alb17], or even when the matrix A is not uniform [Gal11, HM17]. The case where auxiliary information about the secret is provided has also been widely studied [GKPV10, DGK+10, LPSS17].

Finally, in the context of the recent NIST post-quantum cryptography contest, further exotic variants have emerged. We can mention for instance Compact-LWE [Liu17, LLKN17], that has been broken [BTX18, XY18, LLPX18]; the learning with truncation variant that is considered in pqNTRUSign [HPWZ17]; and the Mersenne variants of RLWE, introduced for the ThreeBears [Ham17] and Mersenne-756839 [AJPS17].

Learning with rounding. Another important variant of the LWE problem is the *learning with rounding* (LWR) problem. This problem was first introduced in 2012 by Banerjee, Peikert and Rosen [BPR12], as a de-randomisation of LWE, and its security has been studied since (e.g. [AKPW13, BGM+16]). Here is the considered variant:

$$\lfloor A\mathbf{s} \rceil_p = \mathbf{c} \mod p,$$

where $\lfloor \cdot \rceil_p$ function basically consists in dropping the least significant bits of the input vector. It has been shown in the articles aforementioned that, for certain parameters, this problem is at least as hard as the LWE problem. Note that a ring variant RLWR has been introduced at the same time that the LWR problem.

This de-randomisation allows for instance to build deterministic symmetric primitives such as pseudorandom functions and pseudorandom generators [BPR12, BBL+14, BGL+14, BDFK17], or authenticated encryption schemes [BBSJ16] based on the LWE problem.

LWE over the integers. In a paper accepted in ASIACRYPT 2018, we introduced a simple variant of LWE in which the computations are carried out over $\mathbb{Z}$ rather than $\mathbb{Z}_q$. More precisely, we consider a problem which we called ILWE, for Integer-LWE, which consists in recovering the secret $\mathbf{s} \in \mathbb{Z}^n$ of the following noisy system:

$$A\mathbf{s} + \mathbf{e} = \mathbf{c},$$

where the coefficients of A and $\mathbf{e}$ follow some fixed distributions on $\mathbb{Z}$. This problem may occur more naturally in statistical learning theory or numerical analysis than it does in cryptography. Indeed, as opposed to LWE, it is usually not hard. It can even be easily solved when the error is larger (but not super-polynomially larger in σ_a) than the product $A\mathbf{s}$, under relatively mild conditions on the distributions involved.

Our Contributions

We present here two contributions related to the LWE problem. The first one is the design of a cryptographic primitive, more precisely a pseudorandom generator, based on the RLWR problem. This is the main contribution of [BDFK17]. The other contribution we present here is related to the ILWE problem aforementioned. We propose a rigorous treatment of the question of its hardness, in particular we analyse the number of samples required to solve this problem, and we explain how it can be utilised to recover BLISS secret keys, in a side channel attack.

Building Pseudorandom Generators Using RLWR

The SPRING pseudorandom function (PRF) has been described by Banerjee, Brenner, Leurent, Peikert and Rosen in 2014 [BBL+14]. It is quite fast, only 4.5 times slower than the AES (without hardware acceleration) when used in counter mode. SPRING is similar to a PRF introduced by Banerjee, Peikert and Rosen in 2012 [BPR12], whose security relies on the hardness of the LWR problem, which can itself be reduced to hard lattice problems. However, there is no such chain of reductions relating SPRING to lattice problems, because it uses small parameters for efficiency reasons.

Consequently, the heuristic security of SPRING is evaluated using known attacks and the complexity of the best known algorithms for breaking the underlying hard problem.

Design. In [BDFK17], we proposed a simpler, faster PRG derived from SPRING and revisit the security of all these schemes. On a desktop computer, our variant, called SPRING-RS for SPRING with rejection-sampling, is four times faster than SPRING-CRT, and using AVX2 instructions, it is twice more efficient than the AES-128 without AES-NI instructions and 5 times less efficient than the AES-128 with AES-NI instructions on recent CPUs.

As the main disadvantage of SPRING is its large key size, we also propose several ways to reduce it. The first one uses a 128-bit secret key and another PRG in order to forge the secret

polynomials of SPRING. The other ones use "smaller" instantiations of SPRING in a bootstrapping phase to generate the secret polynomials. We distinguish two cases. (1) In the first case, we use a SPRING instantiation with five secret polynomials to forge other secret polynomials that will be the secret key of another SPRING instantiation, and we reiterate the process until we have a $(k+1)$-polynomial SPRING instantiation. In this case, all secrets are reset at each step, and we never use the same polynomial in two different instantiations. (2) In the second case, we assume that our SPRING instantiation has circular security, and we use only the three first polynomials $\mathbf{a}, \mathbf{s}_1, \mathbf{s}_2$ to forge the next polynomial, using this partial SPRING instantiation, and we reiterate the process until all $\mathbf{s}_i$ are forged. This reduces the key to 3072 bits.

Security analysis. We propose an update of the security estimation of the underlying RLWR problem, using the public LWE-estimator from Albrecht et al. [APS15]. We also propose a new attack against SPRING, to distinguish the output of a SPRING PRG from uniform. This attack can be applied to all SPRING variants. We choose to describe it only for the case of SPRING-RS Finally, we propose an algebraic attack, with the assumption that an adversary may have learned exactly which coefficients have been rejected using side-channel timing attacks. Then, he will have to solve a polynomial system, using Gröbner bases algorithms, to recover the coefficients of the secret polynomials.

Improved Side-Channel attack against BLISS.

The second contribution is devoted to analysing the ILWE problem where the matrix A and the error e follow fixed distributions. As already mentioned, this problem is much easier than LWE. We however give a concrete analysis of the hardness of the problem. We also provide almost tight bounds on the number of samples needed to recover the secret $\mathbf{s}$.

Our main motivation for studying the ILWE problem is a side-channel attack against the BLISS lattice-based signature scheme described by Espitau et al. [EFGT17].

Side-Channel attack against BLISS. BLISS [DDLL13] is one of the most prominent, efficient and widely implemented lattice-based signature schemes, and it has received significant attention in terms of side-channel analysis. Several papers [BHLY16, PBY17, EFGT17] have pointed out that, in available implementations, certain parts of the signing algorithm can leak sensitive informations about the secret key via various side-channels like cache timing, electromagnetic emanations and secret-dependent branches. They have shown that this leakage can be exploited for key recovery.

We are in particular interested in the leakage that occurs in the rejection sampling step of BLISS signature generation. Rejection sampling is an essential element of the construction of BLISS and other lattice-based signatures following Lyubashevsky's "Fiat–Shamir with aborts" framework [Lyu09]. Implementing it efficiently in a scheme using Gaussian distributions, as is the case for BLISS, is not an easy task, however, and as observed by Espitau et al., the optimisation used in BLISS turns out to leak two functions of the secret key via side-channels: an *exact*, *quadratic* function, as well as a *noisy*, *linear* function.

The attack proposed by Espitau et al. relies only on the quadratic leakage, and as a result uses very complex and computationally costly techniques from algorithmic number theory (a generalisation of the Howgrave-Graham–Szydlo algorithm for solving norm equations). In particular, not only does the main part of their algorithm take over a CPU month for standard BLISS parameters, but also technical reasons related to the hardness of factoring make their attack only applicable to a small fraction of BLISS secret key (around 7%; these are keys satisfying a certain smoothness condition). They note that using the *linear* leakage instead would be much simpler if the linear function was exactly known, but cannot be done due to its noisy nature: recovering the key then becomes a high-dimensional noisy linear algebra problem analogous to LWE, which should therefore be hard.

However, the authors missed an important difference between that linear algebra problem and LWE: the absence of modular reduction. The problem can essentially be seen as an instance of ILWE instead, and our analysis thus shows that it is easy to solve. This results in a much more

computationally efficient attack taking advantage of the leakage in BLISS rejection sampling, which moreover applies to *all* secret keys.

Hardness of ILWE On the theoretical side, our first contribution is to prove that, in an information-theoretic sense, solving the ILWE problem requires at least $m = \Omega((\sigma_e/\sigma_a))^2$ samples from the ILWE distribution when the coefficients of the error $\mathbf{e}$ are drawn from a sub-Gaussian distribution of standard deviation σ_e, and the coefficients of the matrix A are drawn from a sub-Gaussian distribution of standard deviation σ_a. We show this by estimating the statistical distance between the distributions arising from two distinct secret vectors $\mathbf{s}$ and $\mathbf{s}'$. In particular, the ILWE problem is hard when σ_e is super-polynomially (in σ_a) larger than σ_a, but can be easy otherwise, including when σ_e exceeds σ_a by a large polynomial factor.

We then provide and analyse concrete algorithms for solving the problem in that case. Our main focus is least squares regression followed by rounding. Roughly speaking, we show that this approach solves the ILWE problem with m samples when $m \geq C \cdot (\sigma_e/\sigma_a)^2 \log n$ for some constant C (and is also a constant factor larger than n, to ensure that the noise-free version of the corresponding linear algebra problem has a unique solution, and that the covariance matrix of the rows $\mathbf{a}$ of A is well-controlled). Our result applies to a very large class of distributions for A and $\mathbf{e}$ including bounded distributions and discrete Gaussians. It relies on sub-Gaussian concentration inequalities.

Interestingly, ILWE can be interpreted as a bounded distance decoding problem in a certain lattice in $\mathbb{Z}^n$ (which is very far from random), and the least squares approach coincides with Babai's rounding algorithm for the approximate closest vector problem (CVP) when seen through that lens. As a side contribution, we also show that even with a much stronger CVP algorithm (including an exact CVP oracle), one cannot improve the number of samples necessary to recover $\mathbf{s}$ by more than a constant factor. And on another side note, we also consider alternate algorithms to least squares when very few samples are available (so that the underlying linear algebra system is not even full-rank), but the secret vector is known to be sparse. In this case, linear programming techniques from [CT07] can solve the problem efficiently.

We provide experimental results both for the plain ILWE problem and the BLISS attack which are consistent with the theoretical predictions (only with better constants). In particular, we obtain a much more efficient attack on BLISS than the one in [EFGT17], which moreover applies to 100% of possible secret keys. The only drawback is that our attack requires a larger number of traces (around 20000 compared to 512 in [EFGT17] for BLISS–I parameters).

Organisation of the Part

In Chapter 10, we present some important notions and definitions that will be useful throughout this part. We also introduce more formally the LWE problem, and some of its variants, or related problems. Finally, we present a quick overview of the most famous attacks against LWE.

In Chapter 11, after having recalled important definitions related to pseudorandom functions (PRFs) and pseudorandom generators (PRGs), we present the BPR family of PRFs. This family is actually proved to be a secure family of PRFs, assuming the hardness of LWE, but is unpractical. We also present the practical variants SPRING-BCH and SPRING-CRT from [BBL+14], that do not enjoy the security reduction.

In Chapter 12, we present our instantiation of SPRING: The SPRING-RS family PRGs. We also introduce a family of PRFs derived from SPRING-RS, as well as a security analysis of our scheme.

In Chapter 13, we describe the BLISS signature scheme and explain why it is vulnerable to side-channel attacks. We show how it is possible to obtain an ILWE instance that would allow us to recover the secret. We also quickly mention the previous attack from [EFGT17], without getting into the details.

Finally, in Chapter 14 we formally introduce the ILWE problem, and present an analysis of its hardness, first from an information theoretic point of view, and then we present concrete ways to solve this problem, the most promising one being the well-known least squares method. Finally, we give the details of the distributions of the ILWE instance arising from the BLISS side-channel attack, and we present experimental results.

Chapter 10

Of Learning With Errors and Related Problems

In this chapter, we recall some important notions, related to lattices, and probability distributions. We also present LWE and other related problems. Finally, we give a quick overview of some of the most famous attacks against LWE.

10.1 Generalities

10.1.1 Notations and Definitions

In this part, to comply with related work, vectors are usually column vectors. Furthermore, the indexation starts with 1 instead of 0. Let $\mathbf{x}$ be a vector.

For $r \in \mathbb{R}$, we denote by $\lfloor r \rceil$ the nearest integer to r (rounding down when r is half integer).

Norm, matrices, and eigenvalues. Let $\mathbf{x} = (x_1 \dots x_n) \in \mathbb{R}^n$, and let $p \in [1, +\infty)$ be an integer. The p-norm of $\mathbf{x}$ is given by $\|\mathbf{x}\|_p = (|x_1|^p + \cdots + |x_n|^p)^{1/p}$, and the infinity norm is given by $\|\mathbf{x}\|_p = \max(|x_1|, \dots, |x_n|)$. Let $r > 0$ and let $\mathbf{c} \in \mathbb{R}^n$. For $p \in [1, +\infty]$, we denote by $\mathcal{B}_m^{(p)}(\mathbf{c}, r)$ the n-dimensional closed ball of centre $\mathbf{c}$ and radius m with respect to the p-norm: namely the subset of $\mathbb{R}^n$ such that:

$$\mathcal{B}_m^{(p)}(\mathbf{c}, r) = \{\mathbf{x} \in \mathbb{R}^n \mid \|\mathbf{x} - \mathbf{c}\|_p \leq r\}.$$

Let A be an m-by-n matrix over $\mathbb{R}$. For $p \in [1, +\infty]$ the norm operator denoted by $\|A\|_p^{\text{op}}$, with respect to the p-norm is given by:

$$\|A\|_p^{\text{op}} = \sup_{\mathbf{x} \in \mathbb{R}^n \{0\}} \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|_p} = \sup_{\|\mathbf{x}\|_p = 1} \|A\mathbf{x}\|.$$

Recall that given a matrix A, an eigenvalue of A is an element λ of $\mathbb{C}$ such that there is a vector $\mathbf{v}$ and $A\mathbf{v} = \lambda\mathbf{v}$. We denote by $\lambda_{max}(A)$ the scalar:

$$\lambda_{max}(A) = \max\{|\lambda|, \text{ such that } \lambda \text{ is an eigenvalue of } A\},$$

and accordingly,

$$\lambda_{min}(A) = \min\{|\lambda|, \text{ such that } \lambda \text{ is a, eigenvalue of } A\}.$$

We also have the following useful equality:

$$\|A\|_2^{\text{op}} = \sqrt{\lambda_{max}({}^tAA)} \tag{10.1}$$

This is due to the fact that:

$$\|A\|_2^{\text{op}} = \sup_{\mathbf{x} \neq 0} \frac{\langle A\mathbf{x}, A\mathbf{x} \rangle}{\langle \mathbf{x}, \mathbf{x} \rangle} = \sup_{\mathbf{x} \neq 0} \frac{{}^t\mathbf{x}{}^tAA\mathbf{x}}{{}^t\mathbf{x}\mathbf{x}}.$$

The ratio $R_{{}^tAA}(\mathbf{x}) = {}^t\mathbf{x}{}^tAA\mathbf{x}/{}^t\mathbf{x}\mathbf{x}$ is called *Rayleigh quotient* of ${}^tAA$. Now, as ${}^tAA$ is symmetric, there is an orthonormal basis that diagonalises ${}^tAA$. In this basis:

$$R_{{}^tAA}(\mathbf{x}) = \frac{\sum_i}{\lambda_i({}^tAA)x_i^2}\sum_i x_i^2.$$

In particular, this is maximal when $\mathbf{x}$ is an eigenvector of ${}^tAA$ associated to $\lambda_{max}({}^tAA)$. In this case, $R_{{}^tAA}(\mathbf{x}) = \lambda_{max}({}^tAA)$, which proves our claim.

Rings. Let n be a power of 2, We denote by R the polynomial ring:

$$R := \mathbb{Z}[X]/(X^n + 1).$$

Concretely all elements $\mathbf{a} \in R$ are polynomials of degree at most $n-1$, and can be represented as follows:

$$\mathbf{a} = \sum_{i=0}^{n-1} a_{i+1}X^i$$

$\mathbf{a}$ is uniquely determined by its coefficients $(a_1, \ldots, a_n)$. We associate $\mathbf{a}$ to the vector over $\mathbb{Z}^n$ whose coefficients are $(a_1, \ldots, a_n)$, and we abuse the notation, by denoting this vector also by $\mathbf{a}$. We also associate to the polynomial $\mathbf{a}$, the skew-circulant matrix:

$$A = \begin{pmatrix} a_1 & -a_n & \ldots & -a_3 & -a_2 \\ a_2 & a_1 & \ddots & -a_4 & -a_3 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ a_{n-1} & a_{n-2} & \ldots & a_1 & -a_n \\ a_n & a_{n-1} & \ldots & a_2 & a_1 \end{pmatrix} \tag{10.2}$$

We have the following properties:

Proposition 10.1. *Let A and B be two skew-circulant matrices respectively associated to polynomials $\mathbf{a}$ and $\mathbf{b}$ of R. Let C be the matrix: $C = AB$. Then:*

1. *C is skew-circulant*
2. *$C = BA$*
3. *The first column of C defines the coefficients of the polynomial $\mathbf{c}$ such that $\mathbf{c} = \mathbf{ab} = \mathbf{ba}$.*
4. *The vector $\mathbf{c}$ satisfies $\mathbf{c} = A\mathbf{b} = B\mathbf{a}$.*

Let q be a positive integer, we denote by R_q the ring:

$$R_q := \mathbb{Z}_q[X]/(X^n + 1). \tag{10.3}$$

Concretely all elements $\mathbf{a} \in R_q$ are polynomials of degree at most $n-1$, whose coefficients a_i, for all $1 \leq i \leq n$, are elements of $\mathbb{Z}_q$. Similarly we can define the vector $\mathbf{a} \in \mathbb{Z}_q^n$ of the coefficients $(a_1 \ldots a_n)$ of the polynomial $\mathbf{a}$. We also define the skew-circulant matrix A associated to the polynomial $\mathbf{a}$ as in Equation 10.2. The properties given in Proposition 10.1 still hold in R_q. We denote by R^* (resp. R_q^*) the group of invertible elements of R (resp. R_q).

Lattices. Let $m \geq n$. A *Lattice* is a discrete additive subgroup of $\mathbb{R}^m$. In other words, let $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ be a family of linearly independent vectors of $\mathbb{R}^m$.

$$\mathcal{L} = \mathbf{b}_1\mathbb{Z} + \ldots \mathbf{b}_n\mathbb{Z} = \{\sum_{i=1}^{n} x_i\mathbf{b}_i \mid \mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{Z}^n\}.$$

The family $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ is called *basis* of $\mathcal{L}$. Given the m-by-n full rank matrix B whose columns are given by $\mathbf{b}_1, \ldots, \mathbf{b}_n$, B is said to be the *basis matrix* of $\mathcal{L}$. For now, we will indifferently refer

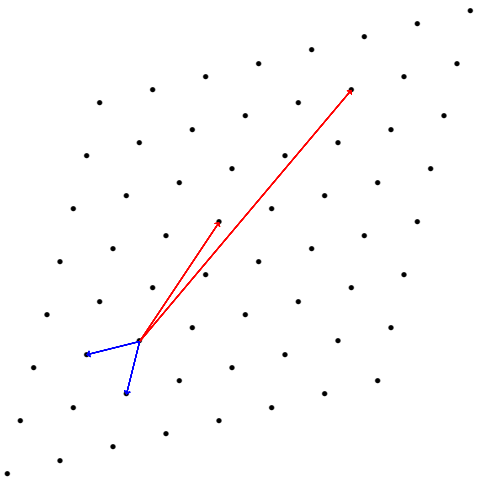


Figure 10.1 – Example of a "good" and a "bad" basis in a lattice.

to a basis or to the basis matrix whose columns are the basis vector. We denote by $\Lambda(B)$ the lattice whose basis is B. This basis is not unique. Furthermore, as we will discuss later, some bases are "better" than other ones. A "good basis" of a lattice $\mathcal{L}$ satisfies some norm and orthogonality properties. In Figure 10.1, the red basis is a "bad" one. On the other hand, the vector in the blue basis are short and as orthogonal as possible, it is a "good" basis.

The dimension of $\mathcal{L}$ is given by m, and its *rank* is given by n. If $m = n$, then $\mathcal{L}$ is said to be a *full-rank* lattice. In this thesis we only need to consider full rank lattices. Recall that given a subset S of $\mathbb{R}^n$, $\mathsf{Span}\{S\}$ is the vector subspace of $\mathbb{R}^n$ spanned by the elements of S. Let $\mathcal{L}$ be a lattice of dimension n. For $1 \leq i \leq n$, we can define the *i-th successive minimum* of $\mathcal{L}$, with respect to the p-norm as:

$$\lambda_i^{(p)} = \min\{r \in \mathbb{N} : \dim \mathsf{Span}\{L \cap \mathcal{B}_n^{(p)}(0, r)\} \geq i\}.$$

In particular the *first minimum* of $\mathcal{L}$ in the p-norm is defined by:

$$\lambda_1^{(p)}(\mathcal{L}) = \min_{\mathbf{x} \in \mathcal{L} \setminus \{0\}} \|\mathbf{x}\|_p = \begin{cases} \min\limits_{\mathbf{x} \in \mathcal{L} \setminus \{0\}} \left(\sum\limits_{i=1}^{m} x_i^p \right)^{1/p} & \text{if } p < +\infty \\ \min\limits_{\mathbf{x} \in \mathcal{L} \setminus \{0\}} \left(\max\limits_i |x_i| \right) & \text{if } p = +\infty \end{cases}$$

We call *dual* of a lattice $\mathcal{L} = \Lambda(B)$, the lattice:

$$\mathcal{L}^* = \{\mathbf{x} \in \mathbb{R}^m \mid {}^tB\mathbf{x} = 0\}.$$

The dual of a lattice is unique and does not depend on the choice of the basis. Let B be an m-by-n full-rank matrix whose coefficients are in $\mathbb{Z}_q$. The *q-ary lattice* of dimension m and rank n spanned by B is defined as follows:

$$\mathcal{L} = \Lambda_q(B) = \{\mathbf{y} = B\mathbf{x} \mod q \mid \mathbf{x} \in \mathbb{Z}^n\}$$

and its dual:

$$\mathcal{L}^* = \{\mathbf{x} \in \mathbb{Z}^m \mid {}^tB\mathbf{x} = 0 \mod q\}.$$

Remark 1. Let $\mathcal{L} = \Lambda_q(B)$ be a q-ary lattice of dimension m and rank n. There exists a full-rank lattice $\mathcal{L}'$ over $\mathbb{Z}^m$, such that $\forall \mathbf{y} \in \mathcal{L}$ any representant of $\mathbf{y}$ in $\mathbb{Z}^m$ is in $\mathcal{L}'$. Furthermore, if $\mathbf{z} \in \mathcal{L}'$, then there is a vector $\mathbf{y} \in \mathcal{L}$ such that $\mathbf{y} = \mathbf{z} \pmod q$

Indeed, let $\mathbf{y}$ be a vector of $\mathcal{L}'$, and $\bar{\mathbf{y}}$ be any representant of $\mathbf{y}$ in $\mathbb{Z}^m$. There is a vector $\mathbf{u} \in \mathbb{Z}^m$ such that: $\mathbf{v} = B\mathbf{x} + q\mathbf{u}$. Now, $\mathbf{v}$ belongs to the lattice $\mathcal{L}'$ over $\mathbb{Z}^m$ such that:

$$\mathcal{L}' = \left\{ \mathbf{v} = \begin{pmatrix} B & qI_m \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x} \\ \mathbf{u} \end{pmatrix} \mid \mathbf{x} \in \mathbb{Z}^n, \mathbf{u} \in \mathbb{Z}^m \right\}.$$

Now, let $\mathbf{v}$ be an element of $\mathcal{L}'$, in particular there is an $\mathbf{x} \in \mathbb{Z}^n$, such that $\mathbf{v} = B\mathbf{x} \pmod q$. Thus $\mathbf{v}$ is a representant in $\mathbb{Z}^n$ of some $\mathbf{y} \in \mathcal{L}$.

10.1.2 Probability Distributions.

Distributions. Let S be a discrete set. We denote by $\mathcal{U}(S)$ the uniform distribution over S. Let X be a random variable, that follows a distribution χ. We write $X \sim \chi$. We write $x \leftsquigarrow \chi$ to say that x is drawn in S according to χ. If χ is a discrete probability distribution over S. For all $s \in S$, we denote by $\chi(s)$ the probability $\mathbb{P}[X = s]$ that a given sample $X \sim \chi$ is equal to s. In particular, for any function $f : S \to \mathbb{R}$,

$$\mathbb{E}[f(s)] = \sum_{s \in S} f(s)\chi(s).$$

Let χ_1 and χ_2 be two probability distributions over S. We define the *statistical distance* $\Delta(\chi_1, \chi_2)$ of χ_1 and χ_2 as:

$$\Delta(\chi_1, \chi_2) = \frac{1}{2} \sum_{s \in S} |\chi_1(s) - \chi_2(s)|.$$

If $\Delta(\chi_1, \chi_2)$ is small enough, then χ_1 and χ_2 are the same distribution. We also define the *Kullback-Leibler (KL) divergence* as:

$$D_{KL}(\chi_1 || \chi_2) = \sum_{s \in S} \chi_1(s) \ln \frac{\chi_1(s)}{\chi_2(s)}. \tag{10.4}$$

The statistical distance and the KL divergence are related by *Pinsker's inequality*:

$$\Delta(\chi_1, \chi_2) \leq \sqrt{\frac{1}{2} D_{KL}(\chi_1 || \chi_2)}. \tag{10.5}$$

Indistinguishability. Let $\eta \in \mathbb{N} \setminus \{0\}$ be a parameter. Let χ_1 and χ_2 be two distributions over S. χ_1 and χ_2 are said to be *statistically indistinguishable* with respect to the (security) parameter η if:

$$\Delta(\chi_1, \chi_2) \leq \mathsf{negl}(\eta).$$

Given an adversary $\mathcal{A}$, who can run any probabilistic polynomial time (in η) algorithm A outputting a single bit. The *advantage* of $\mathcal{A}$ *distinguishing* χ_1 and χ_2, is defined by:

$$\mathsf{Adv}_{\chi_1,\chi_2}(\mathcal{A}) = |\mathbb{P}[A(\chi_1)] = 1| - |\mathbb{P}[A(\chi_2)] = 1|$$

We say that χ_1 and χ_2 are *computationally indistinguishable* if:

$$\mathsf{Adv}_{\chi_1,\chi_2}(\mathcal{A}) \leq \mathsf{negl}(\eta).$$

In other words, two distributions are computationally indistinguishable, if there is no polynomial time (in η) algorithm A whose behaviour significantly changes depending whether its input is drawn from χ_1 or from χ_2.

Furthermore, from [VV14, BCG16], given two distributions χ_1, χ_2, distinguishing whether $\chi_1 = \chi_2$ or $\Delta(\chi_1, \chi_2) \geq \varepsilon$ requires at least $\Omega(1/\varepsilon^2)$ samples.

Rejection sampling. Rejection sampling is a method that was introduced by von Neumann [vN51]. It is used to sample a random variable from a target probability distribution χ, given a source bound to a different probability distribution χ'. To do so, a sample x is drawn from χ' and is accepted with probability:

$$\mathbb{P}[\text{sample } x \text{ accepted}] = \frac{\chi(x)}{M\chi'(x)},$$

where $M > 0$ is a real. If x is not accepted, then the procedure is restarted. It is possible to prove that if for all x, $\chi(x) \leq M \cdot \chi'(x)$, then this rejection sampling procedure produces the distribution χ. The parameter M is chosen so that the number of times the procedure will have to be restarted is as small as possible.

Discrete Gaussian and sub-Gaussian distributions. Let ρ_r be the function that maps x from $\mathbb{R}$ to

$$\rho_r = \exp\left(-\frac{\pi x^2}{r^2}\right).$$

First notice that the sum $S = \sum_{x\in\mathbb{Z}} \rho(r)$ converges. Indeed:

$$S = 1 + 2\sum_{x\in\mathbb{N}\setminus\{0\}} \rho_r(x) = 1 + 2\sum_{x\in\mathbb{N}\setminus\{0\}} \exp\left(-\frac{\pi x^2}{r^2}\right).$$

And, as:

$$x^2 \exp\left(-\frac{\pi x^2}{r^2}\right) \xrightarrow[x\to\infty]{} 0,$$

There exists $k > 0$ such that for all $x \geq k$

$$\exp\left(-\frac{\pi x^2}{r^2}\right) \leq \frac{1}{x^2}.$$

As the $\sum_{x\in\mathbb{N}} 1/(x^2) < \infty$, this is enough to conclude that S is well-defined.

The *discrete Gaussian* distribution D_r^n over $\mathbb{Z}$ is defined as the distribution, in which $x \in \mathbb{Z}$ is drawn with probability:

$$D_r(x) = \frac{\rho_r(x)}{\sum_{y\in\mathbb{Z}} \rho_r(y)}.$$

Let ρ_r be the function that maps a vector $\mathbf{x}$ from $\mathbb{R}^n$ to

$$\rho_r(\mathbf{x}) = \exp\left(-\frac{\pi\|\mathbf{x}\|_2^2}{r^2}\right).$$

Once again, it is possible to show that $\sum_{\mathbf{x}\in\mathbb{Z}^n} \rho_r(\mathbf{x})$ is well-defined. We call *discrete Gaussian* distribution D_r^n over $\mathbb{Z}^n$, the distribution, in which $\mathbf{x} \in \mathbb{Z}^n$ is drawn with probability:

$$D_r^n(\mathbf{x}) = \frac{\rho_r(\mathbf{x})}{\sum_{\mathbf{y}\in\mathbb{Z}^n} \rho_r(\mathbf{y})}.$$

Given an n-dimensional lattice $\mathcal{L}$, for all positive real ϵ, the *smoothing parameter* $\eta_\epsilon(\mathcal{L})$ of $\mathcal{L}$ is the smallest r such that

$$\sum_{\mathbf{x}\in\mathcal{L}^*\setminus\{0\}} \rho_{1/r}(\mathbf{x}) \leq \epsilon.$$

The notion of a *sub-Gaussian distribution* was introduced by Kahane in [Kah60], and can be defined as follows:

Definition 10.1. A random variable X over $\mathbb{R}$ is said to be *τ-sub-Gaussian* for some $\tau > 0$ if the following bound holds for all $s \in \mathbb{R}$:

$$\mathbb{E}\big[\exp(sX)\big] \leq \exp\Big(\frac{\tau^2 s^2}{2}\Big). \tag{10.6}$$

A τ-sub-Gaussian probability distribution is defined in the same way.

Let X be a sub-Gaussian random variable, there is a minimal τ such that X is τ-sub-Gaussian. This τ is sometimes called the *sub-Gaussian moment* of the random variable (or of its distribution).

As expressed in the following lemma, sub-Gaussian distributions always have mean zero, and their variance is bounded by τ^2.

Lemma 10.2. *A τ-sub-Gaussian random variable X satisfies:*

$$\mathbb{E}[X] = 0 \quad \textit{and} \quad \mathbb{E}[X^2] \leq \tau^2.$$

Proof. For s around zero, we have:

$$\mathbb{E}[\exp(sX)] = 1 + s\mathbb{E}[X] + \frac{s^2}{2}\mathbb{E}[X^2] + o(s^2).$$

Since, on the other hand, $\exp(s^2\tau^2/2) = 1 + \frac{s^2}{2}\tau^2 + o(s^2)$, the result follows immediately from Equation 10.6. □

Many usual distributions over $\mathbb{Z}$ or $\mathbb{R}$ are sub-Gaussian. This is in particular the case for Gaussian and discrete Gaussian distributions, as well as all *bounded* probability distributions with mean zero.

Lemma 10.3. *The following distributions are sub-Gaussian.*

1. *The centred normal distribution $\mathcal{N}(0, \sigma^2)$ is σ-sub-Gaussian.*
2. *The centred discrete Gaussian distribution D_r of parameter r is $\frac{r}{\sqrt{2\pi}}$-sub-Gaussian for all $r \geq 0.283$.*
3. *The uniform distribution $\mathscr{U}_\alpha$ over the integer interval $[-\alpha, \alpha] \cap \mathbb{Z}$ is $\frac{\alpha}{\sqrt{2}}$-sub-Gaussian for $\alpha \geq 3$.*
4. *More generally, any distribution over $\mathbb{R}$ of mean zero and supported over a bounded interval $[a, b]$ is $(\frac{b-a}{2})$-sub-Gaussian.*

Moreover, in the cases (1)–(3) above, the quotient $\tau/\sigma \geq 1$ between the sub-Gaussian moment and the standard deviation satisfies:

1. *$\tau/\sigma = 1$;*
2. *$\tau/\sigma < \sqrt{2}$ assuming $r \geq 1.85$;*
3. *$\tau/\sigma \leq \sqrt{3/2}$*

respectively.

The most interesting property concerning sub-Gaussian distributions is that they satisfy a very strong tail bound.

Lemma 10.4. *Let X be a τ-sub-Gaussian distribution. For all $t > 0$, we have*

$$\mathbb{P}[X > t] \leq \exp\left(-\frac{t^2}{2\tau^2}\right). \tag{10.7}$$

Proof. Set $t > 0$. For all $s \in \mathbb{R}$ we have, by Markov's inequality:

$$\mathbb{P}[X > t] = \mathbb{P}[\exp(sX) > e^{st}] \leq \frac{\mathbb{E}[\exp(sX)]}{e^{st}}$$

since the exponential is positive. Using the fact that X is τ-sub-Gaussian, we get:

$$\mathbb{P}[X > t] \leq \frac{\exp\left(\frac{\tau^2 s^2}{2}\right)}{e^{st}} = \exp\left(\frac{s^2\tau^2}{2} - st\right)$$

and the right-hand side is minimal for $s = t/\tau^2$, which exactly gives Equation 10.7. □

It is possible to extend the notion of sub-Gaussian random variables to higher dimension. First we recall the following lemma:

Lemma 10.5. *Let $X_1, \ldots, X_n$ be independent random variables such that X_i is τ_i-sub-Gaussian. For all $\mu_1, \ldots, \mu_n \in \mathbb{R}$, the random variable $X = \mu_1 X_1 + \cdots + \mu_n X_n$ is τ-sub-Gaussian with:*

$$\tau^2 = \mu_1^2\tau_1^2 + \cdots + \mu_n^2\tau_n^2.$$

Proof. Since the X_i's are independent, we have, for all $s \in \mathbb{R}$:

$$\begin{aligned}\mathbb{E}[\exp(sX)] &= \mathbb{E}\big[\exp\left(s(\mu_1 X_1 + \cdots + \mu_n X_n)\right)\big] \\ &= \mathbb{E}\Big[\exp(\mu_1 s X_1)\ldots\exp(\mu_n s X_n)\Big] \\ &= \prod_{i=1}^{n} \mathbb{E}\Big[\exp(\mu_i s X_i)\Big].\end{aligned}$$

Now, since X_i is τ_i-sub-Gaussian, we have

$$\mathbb{E}\big[\exp(\mu_i s X_i)\big] \leq \exp\left(\frac{s^2(\mu_i\tau_i)^2}{2}\right)$$

for all i. Therefore:

$$\begin{aligned}\mathbb{E}[\exp(sX)] &\leq \prod_{i=1}^{n} \exp\left(\frac{s^2(\mu_i\tau_i)^2}{2}\right) \\ &\leq \exp\left(\frac{s^2((u_1\tau_1)^2 + \cdots + (u_n\tau_n)^2)}{2}\right) \\ &\leq \exp\left(\frac{s^2\tau^2}{2}\right)\end{aligned}$$

with $\tau^2 = \mu_1^2\tau_1^2 + \cdots + \mu_n^2\tau_n^2$ as required. □

We can now define what a *τ-sub-Gaussian random vector* is:

Definition 10.2. A random vector $\mathbf{x}$ in $\mathbb{R}^n$ is called a *τ-sub-Gaussian random vector* if for all vectors $\mathbf{u} \in \mathbb{R}^n$ with $\|\mathbf{u}\|_2 = 1$, the inner product $\langle \mathbf{u}, \mathbf{x}\rangle$ is a τ-sub-Gaussian random variable.

Clearly, from Lemma 10.5 if $X_1, \ldots, X_n$ are independent τ-sub-Gaussian random variables, then the random vector $\mathbf{x} = (X_1, \ldots, X_n)$ is τ-sub-Gaussian. Indeed, for any $\mathbf{u} \in \mathbb{R}^n$, of norm $\|\mathbf{u}\|_2 = 1$, we have: $\langle \mathbf{u}, \mathbf{x}\rangle$ is τ'-sub-Gaussian with

$$\tau' = \tau(u_1^2 + \ldots u_n^2) = \tau\|\mathbf{u}\|^2 = \tau.$$

The bound on the sub-Gaussian moment can be used to derive a bound with high probability on the infinity norm as follows:

Lemma 10.6. *Let $\mathbf{v}$ be a τ-sub-Gaussian random vector in $\mathbb{R}^n$. Then:*

$$\Pr\left[\|\mathbf{v}\|_\infty > t\right] \leq 2n \cdot \exp\Big(-\frac{t^2}{2\tau^2}\Big).$$

Proof. If we write $\mathbf{v} = (v_1, \ldots, v_n)$, we have $\|\mathbf{v}\|_\infty = \max(v_1, \ldots, v_n, -v_1, \ldots, -v_n)$. Therefore, the union bound shows that:

$$\Pr\left[\|\mathbf{v}\|_\infty > t\right] \leq \sum_{i=1}^{n} \Pr[v_i > t] + \Pr[-v_i > t]. \tag{10.8}$$

Now each of the random variables $v_1, \ldots, v_n, -v_1, \ldots, -v_n$ can be written as the scalar product of $\mathbf{v}$ with a unit vector of R^n. Therefore, they are all τ-sub-Gaussian. If X is one of them, the sub-Gaussian tail bound of Lemma 10.4 shows that $\Pr[X > t] \leq \exp\big(-\frac{t^2}{2\tau^2}\big)$. From Equation 10.8, we obtain the desired result. □

A nice feature of sub-Gaussian random vectors is that the image of such a random vector under any linear transformation is again sub-Gaussian.

Lemma 10.7. *Let $\mathbf{x}$ be a τ-sub-Gaussian random vector in $\mathbb{R}^n$, and $A \in \mathbb{R}^{m\times n}$. Then the random vector $\mathbf{y} = A\mathbf{x}$ is τ'-sub-Gaussian, with $\tau' = \|A^T\|_2^{\text{op}} \cdot \tau$.*

Proof. Fix a unit vector $\mathbf{u}_0 \in R^m$. We want to show that the random variable $\langle \mathbf{u}_0, \mathbf{y} \rangle$ is τ'-sub-Gaussian. To do so, first observe that:

$$\langle \mathbf{u}_0, \mathbf{y} \rangle = \langle {}^tA\mathbf{u}_0, \mathbf{x} \rangle = \mu \langle \mathbf{u}, \mathbf{x} \rangle$$

where $\mu = \|{}^tA\mathbf{u}_0\|_2$, and $\mathbf{u} = \frac{1}{\mu}{}^tA\mathbf{u}_0$ is a unit vector of $\mathbb{R}^n$. Since $\mathbf{x}$ is τ-sub-Gaussian, we know that the inner product $\langle \mathbf{u}, \mathbf{x} \rangle$ is a τ-sub-Gaussian random variable. As a result, by Lemma 10.5 in the trivial case of a single variable, we obtain that $\langle \mathbf{u}_0, \mathbf{y} \rangle = \mu \langle \mathbf{u}, \mathbf{x} \rangle$ is $(|\mu|\tau)$-sub-Gaussian. But by definition of the operator norm, $|\mu| \leq \|{}^tA\|_2^{\text{op}}$, and the result follows.

□

10.2 Algorithmic Problems

10.2.1 Some Hard Lattice Problems

We give here a brief overview of some of the most important problems on lattices. As these are not the topic of this thesis, we will not develop them.

We consider a lattice $\mathcal{L}$ defined by one of its basis B, and some approximation factor $\gamma \leq 1$. We define these problems in the 2-norm.

The *shortest vector problem* with approximation factor γ (SVP_γ) consists in finding a vector $\mathbf{x} \in \mathcal{L}$ such that $0 < \|x\|_2 \leq \gamma \lambda_1^{(2)}(L)$. In other words, the goal is to find a non-zero vector $\mathbf{x} \in \mathcal{L}$ such that its norm is smaller than γ times the norm of the smallest non-zero vector in $\mathcal{L}$.

Given a parameter $d > 0$, the *gap shortest vector problem* with approximation factor γ (GapSVP_γ) consists in deciding whether $\lambda_1^{(2)}(L) \leq d$ or $\lambda_1^{(2)} \geq \gamma d$.

The *shortest independent vector problem* with approximation factor γ (SIVP_γ) consists in finding n linearly independent vectors $\mathbf{x}_i \in \mathcal{L}$ such that $\max_i \|\mathbf{x}_i\|_2 \leq \gamma \lambda_n^{(2)}(L)$.

Given a target vector $\mathbf{t} \in \mathbb{R}^n$, the *closest vector problem* with approximation factor γ (CVP_γ) consists in finding a vector $\mathbf{x} \in \mathcal{L}$ such that $\|\mathbf{x} - \mathbf{t}\|_2 \leq \gamma \text{dist}(\mathbf{t}, \mathcal{L}) = \gamma \inf_{e \in \mathcal{L}} \|\mathbf{e} - \mathbf{t}\|_2$.

Given a target vector $\mathbf{t} \in \mathbb{R}^n$, such that $\text{dist}(\mathbf{t}, \mathcal{L}) \leq \gamma^{-1} \lambda_1^{(2)}(\mathcal{L})$, the *bounded distance decoding problem* with approximation factor γ (BDD_γ) consists in finding a vector $\mathbf{x} \in \mathcal{L}$ such that $\|\mathbf{x} - \mathbf{t}\| = \text{dist}(\mathbf{t}, \mathcal{L})$.

The decisional problems associated to the problems stated above are known to be NP-Hard for certain values of γ, and polynomial for some other ones.

10.2.2 Learning With Errors

The Learning With Errors (LWE) problem has been introduced by Regev [Reg05] in 2005. This problem is one of the two most fundamental problems in lattice-based cryptography, along with the Short Integer Solution (SIS) problem.

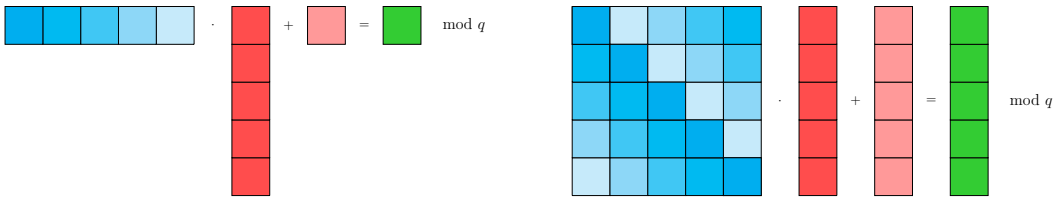
Let n and q be parameters and let χ be an error distribution over $\mathbb{Z}$. Let m be an arbitrarily big integer.

Problem 10.1 ($\mathsf{LWE}_{n,m,q,\chi}$). Given a uniformly random m-by-n matrix A over $\mathbb{Z}_q$, a vector $\mathbf{c} \in \mathbb{Z}_q^m$, a secret uniformly random vector $\mathbf{s} \in \mathbb{Z}_q^n$ and an unknown noise vector $\mathbf{e} \in \mathbb{Z}^m$, such that each coefficient of $\mathbf{e}$ is draw from χ, the goal is to solve the following system:

$$A\mathbf{s} + \mathbf{e} = \mathbf{c} \mod q \tag{10.9}$$

We can also define the decision variant of this problem. Let us consider a secret vector $\mathbf{s} \in \mathbb{Z}_q^n$, and an error distribution χ over $\mathbb{Z}$. We call *LWE distribution* with respect to $\mathbf{s}$ and χ, and denote by $\mathscr{A}_{\mathbf{s},\chi}$, the distribution over $\mathbb{Z}_q^n \times \mathbb{Z}_q$, obtained by choosing a vector $\mathbf{a} \in \mathbb{Z}_q^n$ uniformly at random, and returning $(\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e)$, where e is a noise variable drawn from χ. The decisional version of LWE, denoted by $\mathsf{DecLWE}_{n,m,q,\chi}$ is then defined as follows:

Problem 10.2 ($\mathsf{DecLWE}_{n,m,q,\chi}$). Let $\mathbf{s} \in \mathbb{Z}_q^n$ be a uniformly random secret vector. The goal is to distinguish between m independent samples from $\mathscr{A}_{\mathbf{s},\chi}$, and m samples drawn independently from the uniform distribution over $\mathbb{Z}_q^n \times \mathbb{Z}_q$.



An LWE sample. An RLWE sample.

Figure 10.2 – LWE and RLWE samples.

Hardness of the problem. Regev [Reg09] proved that solving $\mathsf{DecLWE}_{n,m,q,\chi}$ is as hard as solving $\mathsf{LWE}_{n,m,q,\chi}$. In the same paper, he also proved that there exists a polynomial quantum reduction from SIVP to LWE, when the modulus q is prime. Later, Brakerski et al. [BLP+13] proved that there also exists a classical polynomial reduction from GapSVP to LWE. These results can be summarised in the following theorem:

Theorem 10.8 (As stated in [Lep14], from [LLS14])**.** *Let $m, q \geq 2$ be integers, and let $\alpha \in (0,1)$ be such that $\alpha q \geq 2\sqrt{n}$. Let χ be a centred discrete Gaussian distribution over $\mathbb{Z}$ of standard deviation αq.*

1. *If $q = \mathsf{poly}(n)$ is prime, there exists a quantum polynomial reduction from SIVP_γ in dimension n to $\mathsf{LWE}_{q,n,m,\chi}$, with $\gamma = \tilde{O}(n/\alpha)$.*
2. *For any q, there exists a classical polynomial time reduction from GapSVP_γ, in dimension $\Theta(\sqrt{n})$ to $\mathsf{LWE}_{q,n,m,\chi}$, with $\gamma = \tilde{O}(n^2/\alpha)$.*

As these problems are known to be NP-Hard for certain values of γ, some instances of DecLWE are NP-Hard. Other ones are polynomial.

Ring variant. Lyubashevski, Peikert and Regev introduced in [LPR10], a ring variant of LWE. As we do not actually need the general definition which requires to introduce too many notions, we will only focus on the particular case of cyclotomic polynomial rings $R = \mathbb{Z}[X]/(X^n+1)$, where n is a power of 2. Let R_q be the ring $\mathbb{Z}_q[X]/(X^n+1)$. We can define the search Ring-LWE (or RLWE) problem as follows:

Problem 10.3 ($\mathsf{RLWE}_{n,m,q,\chi}$). Let $\mathbf{s}$ be a secret uniformly random polynomial of R_q. Given $2m$ polynomials, $\mathbf{a}^{(i)}, \mathbf{c}^{(i)} = \mathbf{a}^{(i)} \cdot \mathbf{s} + \mathbf{e}^{(i)}$ where all the $\mathbf{a}^{(i)}$ polynomials are drawn uniformly at random from R_q, and the coefficients of the $\mathbf{e}_i$ polynomials are drawn independently according to χ. The goal is to recover $\mathbf{s}$.

This problem can be reformulated in terms of matrices. Indeed, if we denote by $a_1^{(i)} \dots a_n^{(i)}$ the coefficients of $\mathbf{a}^{(i)}$. Then, $\mathbf{a}^{(i)}$ can be represented by the following skew-circulant matrix:

$$A^{(i)} = \begin{pmatrix} a_1^{(i)} & -a_n^{(i)} & \dots & -a_2^{(i)} \\ a_2^{(i)} & a_1^{(i)} & \ddots & -a_3^{(i)} \\ \vdots & \ddots & \ddots & \vdots \\ a_n^{(i)} & \dots & a_2^{(i)} & a_1^{(i)} \end{pmatrix} \tag{10.10}$$

Then, if we consider the $\mathbf{c}^{(i)}, \mathbf{s}$ and $\mathbf{e}^{(i)}$ polynomials as vectors of $\mathbb{Z}_q$, they satisfy the following relation:

$$A^{(i)}\mathbf{s} + \mathbf{e}^{(i)} = \mathbf{c}^{(i)} \mod q$$

The $\mathsf{RLWE}_{n,m,q,\chi}$ problem is then to solve the following noisy system:

$$\begin{pmatrix} A^{(1)} \\ A^{(2)} \\ \vdots \\ A^{(m)} \end{pmatrix} \cdot \mathbf{s} + \begin{pmatrix} \mathbf{e}^{(1)} \\ \mathbf{e}^{(2)} \\ \vdots \\ \mathbf{e}^{(m)} \end{pmatrix} = \begin{pmatrix} \mathbf{c}^{(1)} \\ \mathbf{c}^{(2)} \\ \vdots \\ \mathbf{c}^{(m)} \end{pmatrix} \mod q. \tag{10.11}$$

We illustrate an LWE sample and an RLWE instance in Figure 10.2.

Let us consider a secret vector $\mathbf{s} \in R_q$, and an error distribution χ over $\mathbb{Z}$. We call *RLWE distribution* with respect to $\mathbf{s}$ and χ, the distribution over $R_q \times R_q$, obtained by choosing a polynomial $\mathbf{a} \in R_q$ uniformly at random, and returning $(\mathbf{a}, \mathbf{as} + \mathbf{e})$, where each coefficient of $\mathbf{e}$ is drawn from χ. The decisional version of RLWE, denoted by $\mathsf{DecRLWE}_{n,m,q,\chi}$ is then defined as follows:

Problem 10.4 ($\mathsf{DecRLWE}_{n,m,q,\chi}$). Let $\mathbf{s} \in R_q^n$ be a uniformly random secret polynomial. The goal is to distinguish between m independent samples from a RLWE distribution with respect to $\mathbf{s}$ and χ, and m samples drawn independently from the uniform distribution over $R_q \times R_q$.

Remark 1. Before [LPR10], Stehlé et al. [SSTX09] introduced the *Polynomial Learning With Error* problem PLWE, which consider the following ring $\mathbb{Z}_q[X]/(f(X))$, where $f(X)$ is a monic irreducible polynomial of $\mathbb{Z}[X]$. The instances of RLWE we described above are also instances of PLWE, with $f(X) = X^n + 1$.

Other variants of the LWE problem have been introduced afterward, we can mention *module-LWE* (MLWE) from Langlois and Stehlé [LS15] and more recently *middle-product-LWE* (MPLWE) from Roşca et al. [RSSS17]. As these variants are not in the topic of this thesis, we will not describe them. We refer the interested reader to [LS15, Lan14] for MLWE and [RSSS17] for MPLWE. For reductions and links between all these problems, we refer the reader to [RSW18].

Remark 2. Usually the secret $\mathbf{s}$ is not chosen uniformly at random, but instead, is rather small. Sometimes $\mathbf{s}$ may follow the same distribution χ than the error $\mathbf{e}$. There are other variants where the coefficients of $\mathbf{s}$ are drawn from a small subset of $\mathbb{Z}$ (e.g. $\{0, 1\}$ or $\{0, 1, -1\}$), in other case the secret may be sparse. This remark holds for both LWE and RLWE.

10.2.3 Learning With Rounding

In [BPR12], Banerjee, Peikert and Rosen introduced the Learning With Rounding (LWR) problem, and its ring analog, as "derandomised" version of the (Ring) LWE problem. Indeed, their idea was to generate the error terms efficiently and deterministically, while preserving the hardness of the problem. Instead of adding a small random error vector to the product $A\mathbf{s}$, they proposed to apply a "rounding" that will map the product to the nearest element in $\mathbb{Z}_p$, where $p < q$. To do so, they introduced a rounding function $\lfloor \cdot \rceil_p$ that maps each element x of $\mathbb{Z}_q$ to $\lfloor x \rceil_p$ in $\mathbb{Z}_p$, such that:

$$\lfloor x \rceil_p = \left\lfloor \frac{p}{q} \bar{x} \right\rceil \mod p, \tag{10.12}$$

where $\bar{x} \in (-q/2, \dots q/2] \cap \mathbb{Z}$ is a specific representation of x modulo q.

Remark 3. This definition can be extended coefficient-wise to R_q. In other words, for all $\mathbf{x}$ of R_q, $\lfloor \mathbf{x} \rceil_p$ is the polynomial $\mathbf{y}$ of R_p such that for all $0 \leq i < n$, $y_i = \lfloor x_i \rceil_p$. The ceil function $\lceil \cdot \rceil_p$ and the floor function $\lfloor \cdot \rfloor_p$ are defined accordingly.

Formally, the (search) LWR problem can be defined as follows:

Problem 10.5 ($\mathsf{LWR}_{n,m,q,p}$). Given a uniformly random m-by-n matrix A over $\mathbb{Z}_q$, a vector $\mathbf{c} \in \mathbb{Z}_p^m$, and a secret uniformly random vector $\mathbf{s} \in \mathbb{Z}_q^n$, the goal is to solve the following system:

$$\lfloor A\mathbf{s} \rceil_p = \mathbf{c} \mod p, \tag{10.13}$$

Let us consider a secret vector $\mathbf{s} \in \mathbb{Z}_q^n$ and a parameter $p < q$. We call *LWR distribution* with respect to $\mathbf{s}$ and p and denote by $\mathscr{L}_{\mathbf{s},p}$, the distribution over $\mathbb{Z}_q^n \times \mathbb{Z}_p$, obtained by choosing a vector $\mathbf{a} \in \mathbb{Z}_q^n$ uniformly at random, and returning $(\mathbf{a}, \lfloor \langle \mathbf{a}, \mathbf{s} \rangle \rceil_p)$. The decisional version of LWR, denoted by $\mathsf{DecLWR}_{n,m,q,\chi}$ is then defined as follows:

Problem 10.6 ($\mathsf{DecLWR}_{n,m,q,p}$). Let $\mathbf{s} \in \mathbb{Z}_q^n$ be a uniformly random secret vector. The goal is to distinguish between m independent samples from $\mathscr{L}_{\mathbf{s},p}$, and m samples drawn independently from the uniform distribution over $\mathbb{Z}_q^n \times \mathbb{Z}_p$.

Ring variant. The authors of [BPR12] also proposed a ring variant RLWR of their problem. Below, we recall the definition of the (search) RLWR problem:

Problem 10.7 ($\mathsf{RLWE}_{n,m,q,p}$). Let $\mathbf{s}$ be a secret uniformly random polynomial of R_q. Given $2m$ polynomials, $\mathbf{a}^{(i)}, \mathbf{c}^{(i)} = \left\lfloor \mathbf{a}^{(i)} \cdot \mathbf{s} \right\rceil_p$, where all the $\mathbf{a}^{(i)}$ polynomials are drawn uniformly at random from R_q. The goal is to recover $\mathbf{s}$.

Once again, this can be reformulated in terms of matrices and vectors. Let $A(i)$ be the matrix associated to $\mathbf{a}^{(i)}$, where $A^{(i)}$ is defined as in Equation 10.10. Solving the RLWR problem amounts to solve the following rounded system:

$$\left\lfloor \begin{pmatrix} A^{(1)} \\ A^{(2)} \\ \vdots \\ A^{(m)} \end{pmatrix} \cdot \mathbf{s} \right\rceil_p = \begin{pmatrix} \mathbf{c}^{(1)} \\ \mathbf{c}^{(2)} \\ \vdots \\ \mathbf{c}^{(m)} \end{pmatrix} \mod p. \tag{10.14}$$

Let us consider a secret vector $\mathbf{s} \in R_q$. We call *RLWR distribution* with respect to $\mathbf{s}$ and p, the distribution over $R_q \times R_p$, obtained by choosing a polynomial $\mathbf{a} \in R_q$ uniformly at random, and returning $(\mathbf{a}, \lfloor \mathbf{as} \rceil_p)$. The decisional version of RLWR, denoted by $\mathsf{DecRLWR}_{n,m,q,p}$ is then defined as follows:

Problem 10.8 ($\mathsf{DecRLWE}_{n,m,q,p}$). Let $\mathbf{s} \in R_q^n$ be a uniformly random secret polynomial. The goal is to distinguish between m independent samples from a RLWR distribution with respect to $\mathbf{s}$ and p, and m samples drawn independently from the uniform distribution over $R_q \times R_p$.

Reduction to (Ring)LWE. The authors of [BPR12] also showed that for any distribution over the secret $\mathbf{s}$, the (Ring)LWR problem is at least as hard as the (Ring)LWE problem. Formally, they gave the following result:

Theorem 10.9 ([BPR12]). *Let χ be a B-bounded distribution over $\mathbb{Z}$, and let $q \geq pBn^{\omega(1)}$. Let $m = \mathsf{poly}(m)$.*

1. *For any distribution over $\mathbf{s}\mathbb{Z}_q$, $\mathsf{DecLWR}_{n,m,q,p}$ is at least as hard as $\mathsf{DecLWE}_{n,m,q,\chi}$, for the same distribution over $\mathbf{s}$.*

2. *For any distribution over $\mathbf{s}R_q$, $\mathsf{DecRLWR}_{n,m,q,p}$ is at least as hard as $\mathsf{DecRLWE}_{n,m,q,\chi}$, for the same distribution over $\mathbf{s}$.*

This theorem was proved by the authors of [BPR12]. Later proofs by [AKPW13, BGM+16, ASA16] aimed to reduce the modulus q to about $Bmpn$, considering a fixed number of samples m.

Error rates. One important notion is the one of error rates. It allows to quantify the possible values the error can take. Consider an $\mathsf{LWE}_{n,m,q,\chi}$ instance. We assume the error distribution χ to be centred of mean 0. Let $\alpha > 0$ be a parameter such that the magnitude of an error $e \sim \chi$ is roughly αq.The parameter α is called the *relative error rate* of the LWE error distribution. The quantity αq is called the *absolute error rate*.

It is possible to estimate the error rate of an LWR instance. Consider the following LWR system:

$$\lfloor A\mathbf{s} \rceil_p = \mathbf{c} \mod p,$$

where A is an m-by-n matrix over $\mathbb{Z}_q$, and $\mathbf{s}$ is a vector of $\mathbb{Z}_q^n$. Let us denote by $\mathbf{e}$ the vector of $\mathbb{Z}_q^n$ such that $\mathbf{e} = A\mathbf{s} - \lfloor A\mathbf{s} \rceil_p \mod q$. $\mathbf{e}$ is in a way the "error vector" of the instance. Let e_i be an arbitrary coefficient of $\mathbf{e}$. e_i can take about q/p values. Indeed, the bit-length of e_i is $\log_q - \log p$. If we represent $\mathbb{Z}_q$ as the set: $\{-(q-1)/2, \ldots, (q-1)/2\}$, then $|e_i|$ belongs to $[0, q/(2p)) \cap \mathbb{Z}$, and thus, we say that the *absolute error rate* αq is $q/(2p)$, and thus, the *relative error rate* α is thus given by $1/(2p)$.

10.2.4 Other Related Problems

For completeness, we recall the definition of the (Inhomogeneous) Shortest Integer problem ((I)SIS), as well as the Learning Parity with Noise (LPN), even though they are not essential for this thesis.

SIS and ISIS. The *shortest integer solution problem* (SIS) is with LWE the most important problem used in lattice cryptography. It has been introduced by Ajtai in [Ajt96], where it has been shown that there is connection between worst-case lattice problems and the average-case SIS.

The SIS problem in 2-norm can be defined as follows. Given a uniformly random n-by-m matrix where $m = \mathsf{poly}(n)$, a positive integer q, and a parameter $\beta > 0$ the goal is to find a non-zero vector $\mathbf{x} \in \mathbb{Z}^m$ such that $A\mathbf{x} = 0 \pmod q$ and $\|\mathbf{x}\|_2 \leq \beta$.

Micciancio and Regev [MR07] showed that for large enough q the SIS problem thus defined is, on average, at least as hard as SIVP_γ with $\gamma = \tilde{\mathcal{O}}(\sqrt{n}\beta)$.

The inhomogeneous ISIS variant of SIS, can be defined as follows. Given a uniformly random n-by-m matrix, where $m = \mathsf{poly}(n)$, a positive integer q, a parameter $\beta > 0$, and a target vector $\mathbf{c} \in \mathbb{Z}^n$, the goal is to find a non-zero vector $\mathbf{x} \in \mathbb{Z}^m$ such that $A\mathbf{x} = \mathbf{b} \pmod q$ and $\|\mathbf{x}\|_2 \leq \beta$.

Remark 4. It is possible to convert an LWE instance into a particular ISIS one. Indeed, consider the following LWE instance:

$$A\mathbf{s} + \mathbf{e} = \mathbf{c} \mod q,$$

where A is an m-by-n matrix, with $m > n$. We can rewrite it as:

$$\begin{pmatrix} A & I_m \end{pmatrix} \begin{pmatrix} \mathbf{s} \\ \mathbf{e} \end{pmatrix} = \mathbf{c} \mod q.$$

Now, denoting by A' the matrix $(A\ I_m)$ and by $\mathbf{x}$ the vector $\begin{pmatrix} \mathbf{s} \\ \mathbf{e} \end{pmatrix}$, we come up with the following ISIS instance:

$$A'\mathbf{x} = \mathbf{c} \mod q.$$

Peikert and Rosen [PR06] on one hand and Lyubashevsky and Micciancio [LM06] on the other hand both proposed a ring variant of the (I)SIS problem.

LPN. The *learning parity with noise problem* LPN can essentially be defined as follows: Given an m-by-n matrix A over $\mathbb{Z}_2$, a parameter $\tau \in (0, 0.5)$, and an error vector $\mathbf{e}$, whose coefficients are each drawn independently from a Bernouilli distribution of parameter τ, the goal is to solve the following noisy system:

$$A\mathbf{s} \oplus \mathbf{e} = \mathbf{c}.$$

Remark 5. It is possible to see LPN as a particular case of LWE with $q = 2$. In fact, in [Reg05], Regev introduced LWE as a generalisation of LPN to moduli other than 2.

10.3 Overview of the Most Usual Attacks Against LWE

We give a brief overview of some of the most usual algorithms used to solve the LWE problem. We present here some lattice-based algorithms, as well as the combinatorial BKW algorithm, and some algebraic methods due to Arora and Ge. As this is not the topic of this thesis, we just recall the idea of these different methods, without detailing their optimisation, nor their complexity analysis.

In this section, we consider the following noisy system from Equation 10.9, and shown in Figure 10.3 and we present some algorithms that aim to solve it.

10.3.1 Solving LWE Via Lattice Reduction

High level idea. We consider the following lattice:

$$\mathcal{L} = \Lambda_q(A) = \{\mathbf{v} \in \mathbb{Z}^m \mid \mathbf{v} = A\mathbf{s}, \mathbf{s} \in \mathbb{Z}^n\}. \tag{10.15}$$

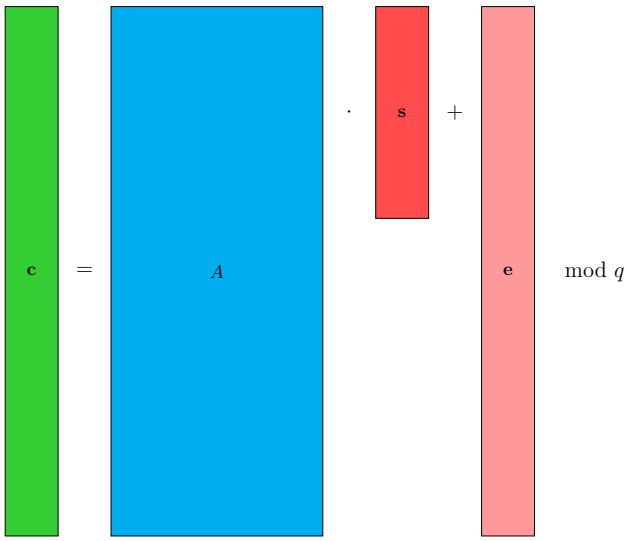


Figure 10.3 – LWE system, where A is an m-by-n matrix.

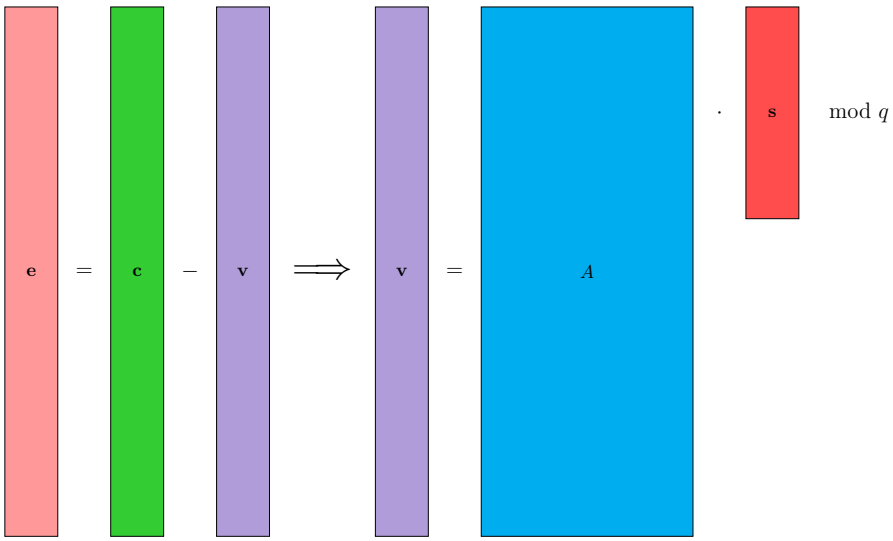


Figure 10.4 – Reducing an LWE instance to a CVP one.

We assume for now that we know how to solve the CVP problem over $\mathcal{L}$. Then we can find a vector $\mathbf{v} \in \mathcal{L}$, such that $\|\mathbf{v} - \mathbf{c}\|$ is small. Now, writing $\mathbf{e}$ as $\mathbf{c} - \mathbf{v}$ and setting $A\mathbf{s} = \mathbf{v} \mod q$, we can easily recover $\mathbf{s}$ using linear algebra, as A consists of more rows than columns. We illustrate this in Figure 10.4

To find this vector $\mathbf{v}$, there are several possibilities. One can for instance reduce the problem to an SVP instance in a lattice in dimension one larger. This technique due to Kannan is called embedding [Kan87]. Another method would be to solve CVP directly: for instance by enumerating all lattice points that are in a "small" ball centred in $\mathbf{v}$. We refer the reader to [AKS01, AKS02, GNR10, HPS11] for details about these methods. It is also possible to find the closest vector to $\mathbf{b}$ in $\mathcal{L}$ using projections techniques [Bab86]. We briefly recall these methods below.

Solving CVP using embedding techniques. We consider the lattice $\mathcal{L}$ given in Equation 10.15. We aim to find $\mathbf{v}$ in $\mathcal{L}$ such that the vector $\mathbf{e} = \mathbf{c} - \mathbf{v}$ is small.

The idea is to consider the matrix:

$$A' = \begin{pmatrix} A & \mathbf{c} \\ 0 & \kappa \end{pmatrix},$$

where κ is a well chosen constant (e.g. following [Gal11] $\kappa = 1$). A' spans a q-ary lattice $\mathcal{L}'$ of

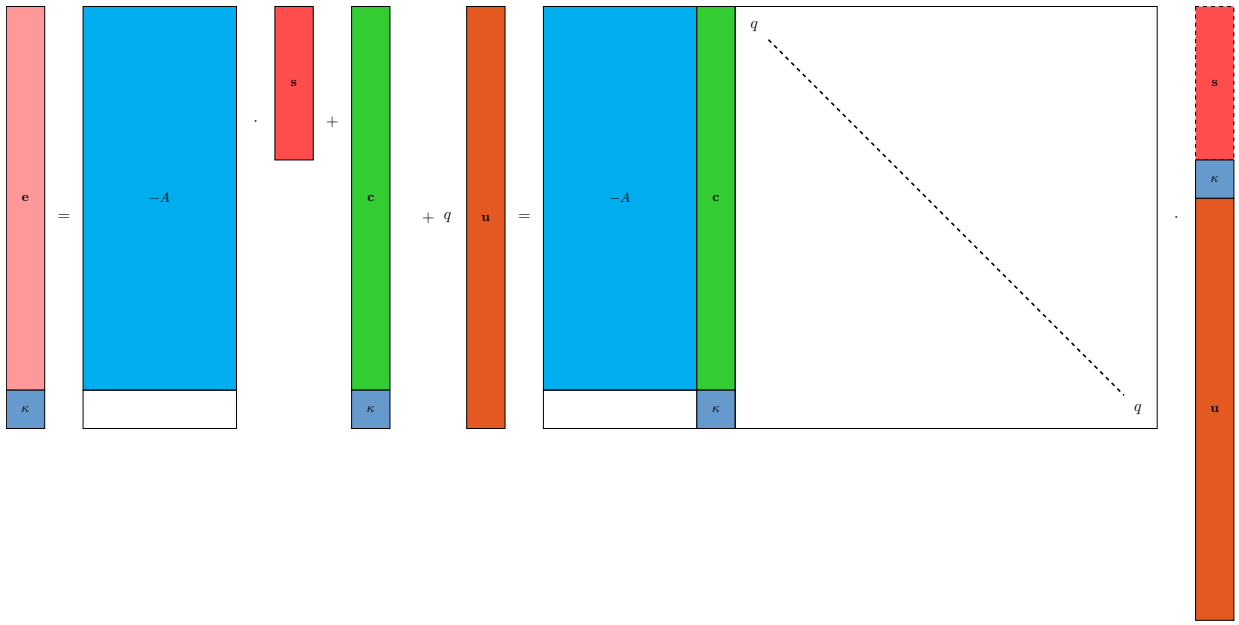


Figure 10.5 – Reducing LWE to an SVP one.

dimension $(m+1)$. In particular we have:

$$A'\begin{pmatrix}-\mathbf{s}\\1\end{pmatrix} = \begin{pmatrix}-A & \mathbf{c}\\ 0 & \kappa\end{pmatrix}\begin{pmatrix}-\mathbf{s}\\1\end{pmatrix} = \begin{pmatrix}-A\mathbf{s}+\mathbf{c}\\ \kappa\end{pmatrix} = \begin{pmatrix}\mathbf{e}\\ \kappa\end{pmatrix}.$$

Thus, $\begin{pmatrix}\mathbf{e}\\ \kappa\end{pmatrix}$ is a short vector in $\mathcal{L}'$. Now, if we find a short vector $\begin{pmatrix}\mathbf{e}\\ \kappa\end{pmatrix}$ in $\mathcal{L}$, and set $\mathbf{v} = \mathbf{c} - \mathbf{e}$, we are essentially done.

To find a short vector in $\mathcal{L}'$, we consider the lattice $\tilde{\mathcal{L}}'$ over $\mathbb{Z}^{m+1}$ consisting of the vectors $(A'\ qI_m)\mathbf{x}$ for $\mathbf{x} \in \mathbb{Z}^{n+1+m}$ and we find a "good" basis using lattice reduction algorithm (e.g. LLL, BKZ). A short vector $\tilde{\mathbf{y}}$ in $\tilde{\mathcal{L}}$ will give a short vector $\mathbf{y}$ in $\mathcal{L}$. We illustrate this in Figure 10.5

Solving CVP directly. It is also possible to solve CVP in $\mathcal{L}$ with target $\mathbf{c}$ directly. For that, we consider the lattice $\tilde{\mathcal{L}}$ over $\mathbb{Z}^m$ consisting of the vectors $(A\ qI_m)\mathbf{x}$, for $\mathbf{x} \in \mathbb{Z}^{n+m}$. The idea, once again, is to find a "good" basis B, then for a well chosen parameter r, to enumerate all vectors of $\tilde{\mathcal{L}}$ that are in a ball of centre $\mathbf{c}$ and radius r. Using this method, it is guaranteed to find the closest vector $\mathbf{v}$ to $\mathbf{c}$ in $\tilde{\mathcal{L}}$. There are several ways to chose the parameter r. For instance, r can be chosen as $\|\mathbf{b}_1\|$ where $\mathbf{b}_1$ is the shortest vector of the basis B, if B is reduced enough. One can also decide to choose a small value for r, and if the search fails, increase r and restart.

Another possibility, due to Babai [Bab86] is to project $\mathbf{c}$ on the lattice $\tilde{\mathcal{L}}$. The first intuitive way to do this is very easy to apprehend. Given a basis B of $\tilde{\mathcal{L}}$, where the basis vector are denoted by $\mathbf{b}_1, \ldots, \mathbf{b}_m$, the target vector $\mathbf{c}$ can be expressed as follows:

$$\mathbf{c} = \sum_{i=1}^{n} \alpha_i \mathbf{b}_i,$$

where $\alpha_i \in \mathbb{R}$, for all i. Then, it is possible to set $\mathbf{v}$ to the vector:

$$\mathbf{v} = \sum_{i=1}^{n} \lfloor \alpha_i \rceil\, \mathbf{b}_i.$$

Doing so, however, it is not guaranteed that $\mathbf{v}$ will be *the closest* vector to $\mathbf{c}$.

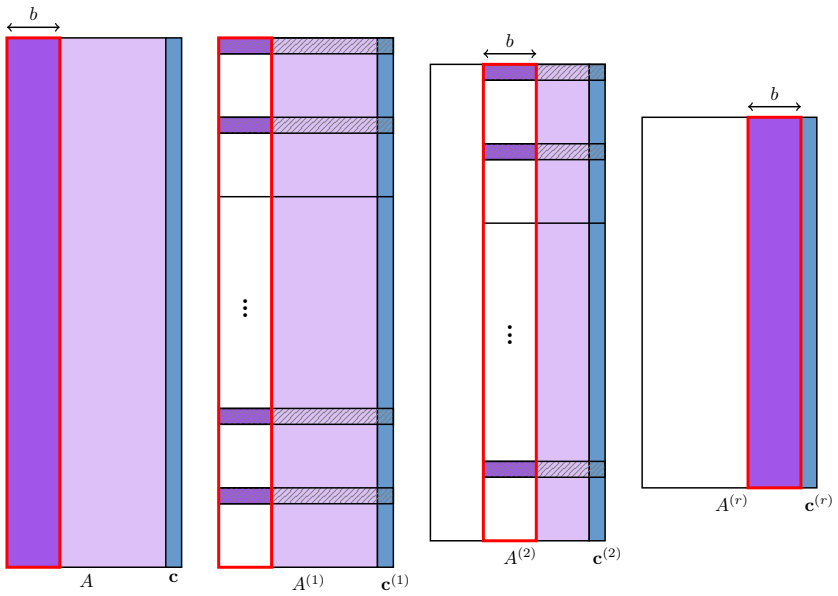


Figure 10.6 – Representation of the BKW algorithm.

The other method due to Babai, called Babai nearest plane consists in solving the problem inductively in smaller dimension. In brief, if $\mathbf{b}_1, \ldots, \mathbf{b}_n$ define a basis of $\tilde{\mathcal{L}}$, one can consider the subspace of $\mathbb{R}^n$, $V = \mathsf{Span}\{\mathbf{b}_1, \ldots \mathbf{b}_{n-1}\}$. Then the idea is to find a vector $\mathbf{y}_1 \in L$ such that the distance from $\mathbf{c}$ to the plane $V + \mathbf{y}_1$ is minimal. Then, denoting by $\mathbf{c}* \in V + \mathbf{y}_1$ the orthogonal projection of $\mathbf{w}$ onto $V + \mathbf{y}_1$, and by $\mathbf{c}_1$ the vector $\mathbf{c}_1 = \mathbf{c}^* - \mathbf{y}_1 \in V$, the problem is now to find the closest vector from $\mathbf{c}_1$ in the lattice $\tilde{\mathcal{L}}' = \tilde{\mathcal{L}} \cap U$. The solution $\mathbf{v}$ to the CVP instance will then be $\mathbf{v} = \mathbf{y}_1 + \mathbf{y}_2 + \cdots + \mathbf{y}_n$.

Reduced basis. As we have already mentioned, given a lattice $\mathcal{L}$ some bases of $\mathcal{L}$ are quite "good" and other are "bad". We say that a basis is "good" when it consists of short vectors that are more or less orthogonal to each other. It is quite easy to see that if we dispose of such a basis, solving the SVP problem, or the CVP problem becomes quite easy. In particular, if one of the vectors $\mathbf{b}_1$ of the basis B satisfies $\|\mathbf{b}_1\|_2 \leq \gamma\lambda_1^{(2)}$, then solving SVP_γ is trivial.

On the other hand, if the basis B consists of long vectors, then the problem is much harder. Moreover, finding a "good" basis of a lattice $\mathcal{L}$ when given a "bad" one is not an easy task. This is the goal of *lattice basis reduction algorithms*. The celebrated LLL Algorithm, due to Lenstra, Lenstra and Lovász [LLL82] already finds bases of quite good quality in polynomial time. However, this is usually not enough for cryptanalysis application.

Schnorr and Euchner's block-wise BKZ Algorithm [SE94] is usually preferred, as it provides bases of better qualities. Morally this algorithm takes as input a basis B of a lattice $\mathcal{L}$ and a parameter β and output a reduced basis B', where the vectors $\mathbf{b}'_1 \ldots \mathbf{b}'_n$ are ordered from the shortest to the largest, such that: for all $1 \leq i \leq n$, $\|\mathbf{b}'_i\| = \lambda_1^{(2)}(\Lambda(B'_{i..k}))$, where $B'_{i..k}$ is the sub-matrix of B' containing the columns $\mathbf{b}'_j$ to $\mathbf{b}'_k$, and where $k = \min(j + \beta - 1, n)$. From here, we can see that the quality of the basis increases with the parameter β (the block-size). For instance, if $\beta = m$ where m is the dimension of $\mathcal{L}$ then, in particular $\|\mathbf{b}_1\| = \lambda_1^{(2)}(\mathcal{L})$. However, the complexity of the BKZ algorithm also significantly increases as the parameter β grows.

Finally, we need to mention that the BKZ algorithm has known notable improvements in the last few years. Chen and Nguyen proposed an implementation called BKZ-2.0 [CN11] also fully described in [Che13]. In particular, their implementation utilises many heuristics such as preprocessing of local bases [Kan83], and efficient enumeration algorithms [GNR10, HPS11]. A public implementation of this algorithm has been made available by its authors. As for more public implementation of lattice reduction algorithms, we refer the interested reader to the FPLL library [dt16] also available as a Sage package. We can also mention the IALatRed library [EJ16].

10.3.2 Blum Kalai and Wasserman Algorithm

This algorithm was initially designed by Blum Kalai and Wasserman [BKW03] to solve the *Learning Parity with Noise* (LPN) problem, which can be seen as a particular case of the LWE problem over $\mathbb{F}_2$. It has later been adapted to solve LWE (e.g. [ACF+15, KF15, DTV15]). The idea of the BKW Algorithm is to perform a variant of the Gaussian elimination, that aims to cancel blocks of coefficients, instead of only one, at each iteration. Then, if the error is still small enough, and if the attacker disposes of sufficiently many samples, she can recover a part of the secret. By re-iterating the procedure, she can find back all coefficients of $\mathbf{s}$. We illustrate this method in Figure 10.6. Morally, we consider blocks of b coefficients, for a well chosen parameter b. We assume for simplicity that n is a multiple of b. The vectors are afterward dispatched into buckets according to their first b coefficients. In each bucket, one vector is chosen to be used to eliminate the first b coefficients of all other vectors of the bucket with small linear combinations (it is similar to the pivot in Gaussian elimination). The same changes are applied to the right-hand side $\mathbf{c}$. Only the vectors that start with b zeroes are kept. The process is re-iterated with the next block, and again, until we come up with a smaller matrix of the type:

$$\begin{pmatrix} 0 & A' \end{pmatrix}.$$

The system $A\mathbf{s} + e = \mathbf{c}$, can then be re-written as:

$$\begin{pmatrix} 0 & A' \end{pmatrix} \mathbf{s} + \mathbf{e}' = \begin{pmatrix} 0 & A' \end{pmatrix} \begin{pmatrix} \mathbf{s}_0 \\ \mathbf{s}_1 \end{pmatrix} + \mathbf{e}' = \mathbf{c}',$$

where $\mathbf{s}_1$ consists in the last b coefficients of the secret $\mathbf{s}$. It follows that:

$$A' \cdot \mathbf{s}_1 + \mathbf{e}' = \mathbf{c}'.$$

If we still dispose of enough samples, and if the error has not grown to much, (we can hope it will be the case, as only small linear combinations of the error were made), it is possible to recover $\mathbf{s}_1$. To recover the full secret, we have to re-iterate the method, considering the smaller LWE instance:

$$A_0\mathbf{s}_0 + \mathbf{e} = \mathbf{b} - A_1\mathbf{s}_1.$$

where A_0 represents the first columns of A, and A_1 the last b ones.

10.3.3 Algebraic Attacks

The algebraic attack against LWE were introduced by Arora and Ge [AG11]. This attack works by replacing the noisy LWE linear system by a polynomial system free of noise and then linearising the system. Albrecht et al. proposed a variant of this method using Gröbner basis instead of linearisation [ACF+15].

Let α be the relative error rate of the LWE instance. With high probability, each coefficient of $\mathbf{e}$ belongs to $[-\alpha q, \alpha q] \cap \mathbb{Z}$. Now consider the polynomial P such that:

$$P(X) = X \prod_{i=1}^{\alpha q} (X - i)(X + i).$$

The degree of P is $2\alpha q + 1$. For each coefficient e_i of $\mathbf{e}$, it is quite clear that with high probability, $P(e_i) = 0$. In particular, if the number m of LWE sample we dispose of is large enough, it is possible to build an m-by-n polynomial system in n unknown and of degree $2\alpha q + 1$. If $m = m_{AG} = n^{\mathcal{O}(2\alpha q+1)}$, it is possible to find a solution to this system by linearisation.

Albrecht et al. [ACF+15] were able to drop this value to $m = \sqrt[\theta]{m_{AG}}$, were $\theta = n^{2-\beta}$, for some constant β (e.g. $\beta = 1/5$).

Chapter 11

Of RLWR-based Pseudorandom Functions and Pseudorandom Generators

In this chapter, we recall generalities about the pseudorandom functions and pseudorandom generators. We also present the BPR family of PRFs as well as two previous SPRING instantiations, SPRING-BCH and SPRING-CRT.

11.1 Pseudorandom Functions and Pseudorandom Generators

11.1.1 Generalities

Definitions. *Pseudorandom functions* (PRFs) and *Pseudorandom Generators* (PRGs) are fundamental tools in symmetric cryptography.

Definition 11.1 (PRF)**.** Given two positive integers k and m (possibly distinct), a PRF F is a deterministic function mapping k-bit strings into m-bit strings, such that, upon the choice of a random and secret key, F is computationally indistinguishable from a truly random function, via adaptive "black-box" oracles calls.

In other words, if we consider two black-box oracles **F** and **G**, such that one of them implements F, and the other implements a truly random function, and an adversary who:

1. does not know which oracle implement which function,
2. cannot see what is actually implemented inside,
3. can make adaptive calls to each of the oracles,

the adversary cannot efficiently distinguish which one of the oracles implement F, with good probability.

Definition 11.2 (PRG)**.** Given two positive integers k and m, with $m \geq k$, a PRG G is a deterministic function mapping k-bit strings into m-bit strings, and whose output distribution is, upon the choice of a random input, computationally indistinguishable from the uniform distribution.

Proof of security vs efficiency. We can distinguish two classes of PRFs and PRGs: those who actually admit security proofs assuming the hardness of some problems (e.g. [GGM84, BBS86]), and those whose security is conjectured and is supported by the fact that there is no known attack, see for instance AES [DR99], the Keccak function [BDPVA09], among others. The first class is mainly theoretical, and the design of these PRFs/PRGs are impractical or too inefficient for real word uses. Design of the second classes are usually very fast, but may possibly become vulnerable to new types of attack.

In an intend to bridge this gap between efficiency and theoretical security, several works have been proposed in the last few years. Lyubashevsky et al. introduced in 2008 [LMPR08] a hash function, that enjoys an asymptotic proof of security, and is quite efficient for practical parameters. A few years later, along with the introduction of the LWR problem, the authors of [BPR12] proposed a family of PRFs, called BPR, that enjoys a proof of security, but remains impractical for the parameters proposed in [BPR12]. The authors of [BBL+14] chose to relax the conditions on the secret key and the size of the modulus to obtain an efficient variant of the BPR family of PRFs. Their design still enjoys a strong algebraic structure, even though there is no proof of security for their chosen set of parameters.

11.1.2 Building One From the Other

PRG from PRF. Once we have a PRF $f : \{0,1\}^k \to \{0,1\}^m$, it is quite easy to build a PRG $G : \{0,1\}^k \to \{0,1\}^{m'}$, with $m' \geq m$, using a counter mode. In fact, let $\mathbf{x} \in \{0,1\}^k$ be a random input of f, and let $(\mathbf{x}_n)$ be an arbitrary sequence of bit-strings such that:

1. $\mathbf{x}_0 = \mathbf{x}$
2. $\mathbf{x}_{i+1} = g(\mathbf{x}_i)$, for all i and for some function g that satisfies $\forall i < j \ g(\mathbf{x}_i) \neq g(\mathbf{x}_j)$.

For instance, if we denote by x the integer whose bits representation is $\mathbf{x}$, g can be the function $x \to x + 1$. Usually the *Gray-code counter mode* is preferred (see below).

Let α be a positive integer and set $m' = \alpha \cdot m$. G can be constructed such as in algorithm 27

Algorithm 27 PRG constructed from a PRF f.

Require: A PRF $f : \{0,1\}^k \to \{0,1\}^m$, a seed $\mathbf{x}_0$, and the function g to update the sequence $(\mathbf{x}_n)$.
Ensure: An apparently uniformly random bit-string $\mathbf{y}$ of length $m' = \alpha m$.

```
Set x ← x_0.
Set y ← empty string
Set m' ← 0
while m' < αm do
    Set y_0 ← f(x)
    Set y ← (y|y_0)
    Set x ← g(x)
    Set m' ← m' + m
return y
```

Gray-code counter mode. Given a positive integer k, A Gray code is a simple way to order bit-strings of length k, such that two successive values of the counter differ only by one bit. Assume the counter starts with 0. To transform the i-th value $\mathbf{x}$ of a counter to the $(i+1)$-th, the $(j+1)$-th bit of the counter has to be flipped, where j is the number of trailing zeroes in the binary expansion of $(i+1)$. Let us denote by $\mathbf{i}$ the binary expansion of i. We give an example below, for bit-strings of length 3:

i	x	$\mathbf{i}$
0	000	000
1	100	100
2	110	010
3	010	110
4	011	001
5	111	101
6	101	011
7	001	111

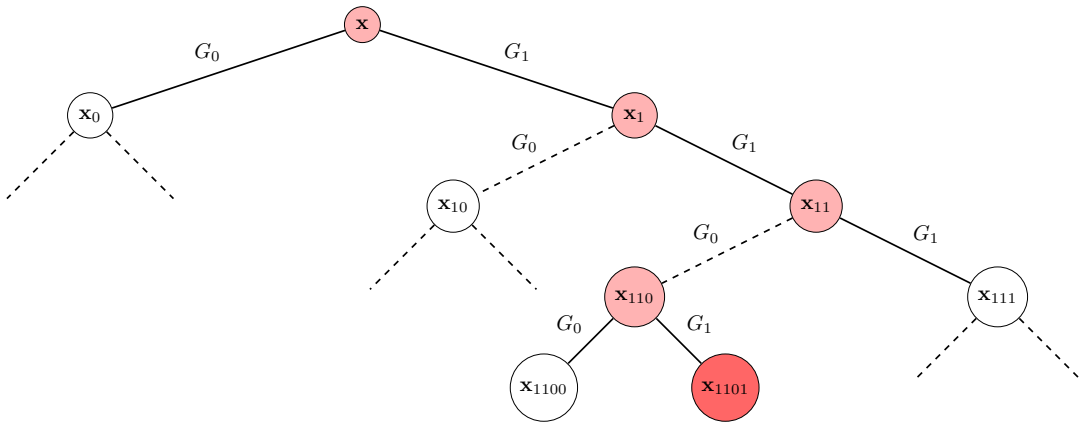


Figure 11.1 – Computation of $F_{\mathbf{x}}(1101)$ with the GGM construction.

PRF from PRG. It is also possible to construct a PRF from a PRG, although the transformation is not practical. This construction is due to Goldreich, Goldwasser and Micali [GGM84], and therefor known as the GGM construction. We briefly recall its idea.

Let us consider a PRG G, that maps k-bit strings to $2k$-bit strings, and let us denote by G_0 and G_1 the functions such that:

$$G(\mathbf{x}) = G_0(\mathbf{x})|G_1(\mathbf{x}),$$

for any $\mathbf{x} \in \{0,1\}^k$. Let $\mathbf{x}$ be a random bit string of length k, and consider the family of functions $F_{\mathbf{x}}$, that map ℓ-bit string into k-bit, for some ℓ such that for all $\mathbf{y} \in \{0,1\}^\ell$,

$$F_{\mathbf{x}}(\mathbf{y}) = \mathbf{x}_{\mathbf{y}},$$

where for all string $\mathbf{y}$, $\mathbf{x}_{\mathbf{y}}$ is defined as follows:

$$\mathbf{x}_{\mathbf{y}} = \begin{cases} \mathbf{x} & \text{if } \mathbf{y} \text{ is empty} \\ G_{\mathbf{y}[k-1]}(\mathbf{x}_{\mathbf{y}[..k-1]}) & \text{if } \mathbf{y} \text{ is of length at least 1} \end{cases}$$

For instance, if $\ell = 4$, $F_{\mathbf{x}}(1101) = G_1(G_0(G_1(G_1(\mathbf{x}))))$. This is illustrated by Figure 11.1.
The authors of [GGM84] showed that $F_{\mathbf{x}}$ is a secure PRF, as long as G is a secure PRG.

11.2 The BPR Family of PRF

In [BPR12] Banerjee Peikert and Rosen proposed along with the definition of (Ring-)LWR, a new family of PRF known as BPR. We explain how this family is build below.

11.2.1 Design

Let p, q and k be positive integers, let $\mathbf{a}, \mathbf{s}_1, \dots \mathbf{s}_k$ be secret polynomials of R_q^*. The function $F_{\mathbf{a},(\mathbf{s}_i)}$ is defined in the following way:

$$\begin{aligned} F_{\mathbf{a},\{\mathbf{s}_i\}} :\ & \{0,1\}^k \to R_p \\ \mathbf{x} = (x_1, \dots, x_k) \mapsto & \left\lfloor a \cdot \prod_{i=1}^{k} \mathbf{s}_i^{x_i} \right\rceil \end{aligned} \tag{11.1}$$

Furthermore, the authors of [BPR12] assumed that the $\mathbf{s}_i$ polynomials are drawn independently from some B-bounded distribution χ over R_q such that $B \geq n^{\Omega(k)}$, and that the modulus q is also larger than $n^{\Omega(k)}$. Then assuming the hardness of $\mathsf{RLWE}_{n,q,\chi}$ the function $F_{\mathbf{a},\{\mathbf{s}_i\}}$ is a secure PRF.

Theorem 11.1 ([BPR12]). *Let χ be the distribution over R where each coefficient is chosen independently from a discrete Gaussian distribution D_r, for some $r > 0$, and let $q \geq pk(\sqrt{n}\cdot\omega(\sqrt{\log n}))^k \cdot n^{\omega(1)}$. If each $\mathbf{s}^i$ is drawn independently from χ, the function $F_{\mathbf{a},\{\mathbf{s}_i\}}$, as defined in Equation 11.1 is pseudorandom, assuming the hardness of $\mathsf{Dec\text{-}RLWE}_{n,q,\chi}$.*

A full proof of this theorem, for the ringless variant of the function, is given in [BPR12]. We are not going to prove it over again in here, but only to recall the general idea of the proof.

The problem here is that such large parameters make the whole scheme totally impractical. However, as it was already noticed by the authors of [BPR12], choosing parameters that large seems to be an artefact of the proof, and the scheme may still be secure for appropriately distributed $\mathbf{s}_i$, small modulus q, and large k.

11.2.2 Idea of the Security Proof

In their proof, the authors of [BPR12] consider the function $G_{\mathbf{a},\{\mathbf{s}_i\}}$ that maps bit-strings of length k to a polynomial of R_q, such that:

$$G_{\mathbf{a},\{\mathbf{s}_i\}} = \mathbf{a}\prod_{i=1}^{k}\mathbf{s}_i^{x_i}.$$

In fact, $G_{\mathbf{a},\{\mathbf{s}_i\}}$ is the unrounded counterpart of $F_{\mathbf{a},\{\mathbf{s}_i\}}$. They also define the $\tilde{G}^{(i)}$ family of functions as follows:

- $\tilde{G}^{(0)}_{\mathbf{a}}$ is a constant equal to $\mathbf{a} \in R_q$.
- $\tilde{G}^{(i)} = \tilde{G}^{(i)}_{G^{(i-1)},\mathbf{s}_i,\mathbf{e}}$ for $0 < i \leq k$, is a function that maps an i-bit string to a polynomial of R_q, as follows:
$$\tilde{G}^{(i)}(\mathbf{x}|x_i) = \tilde{G}^{(i-1)}(\mathbf{x})\cdot\mathbf{s}_i^{x_i} + x_i\mathbf{e}_{\mathbf{x}},$$
where $\mathbf{e}_{\mathbf{x}} \in R_q$ is an error polynomial, such that for all $\mathbf{x} \in \{0,1\}^{i-1}$, $\mathbf{e}_{\mathbf{x}}$ is chosen independently from χ.

This leads to a function $\tilde{G} = \tilde{G}^{(k)}$, which can be seen has a randomised counterpart of G.

Remark 1. With the construction above, the gap between G and $\tilde{G}^{(i)}$ grows exponentially in k.

By induction over i, and assuming the hardness of LWE, It is possible to prove that $\tilde{G}^{(i)}$ is pseudorandom for all $1 \leq i \leq k$. Then, it is possible to show that $\left\lfloor\tilde{G}(\mathbf{x})\right\rceil_p = \left\lfloor G_{\mathbf{a},\{\mathbf{s}_i\}}(\mathbf{x})\right\rceil_p = F_{\mathbf{a},\{\mathbf{s}_i\}}(\mathbf{x})$, for all $\mathbf{x}$ (assuming the $\mathbf{s}_i$ polynomials are small). This proves that $F_{\mathbf{a},\{\mathbf{s}_i\}}$ is also a pseudorandom function.

11.3 The SPRING Family of PRF

In [BBL+14], Banerjee et al. proposed a new family of PRF called SPRING for **s**ubset **p**roduct with **r**ounding over a r**ing**. Their construction is very similar to BPR with smaller parameters so that the design will be practical. They actually proposed efficient software implementations on Intel and ARM processor. A FPGA (Field-Programmable Gate Array) implementation was also proposed [BGL+14].

11.3.1 Design

Let $k \in \{64, 128\}$ be the length of the input. Let $n = 128$, $q = 257$ and $p = 2$. Let $m \leq n$ be a positive integer which will be the length of the output of the PRF, and let g be a function (to be determined later) which takes as input a polynomial of R_p and returns a bit-string of length m. Let R be a function from R_q to $\{0,1\}^m$ such that for all $\mathbf{b}$ in R_q, $\mathsf{R}(\mathbf{b}) = g(\lfloor\mathbf{b}\rceil_p)$.

Let $\mathbf{a}, \mathbf{s}_1, \ldots, \mathbf{s}_k$, be polynomials of R_q^* chosen independently and uniformly at random. We define the function $F_{\mathbf{a},(\mathbf{s}_i)}$ such that:

$$F_{\mathbf{a},(\mathbf{s}_i)} : \{0,1\}^n \to \{0,1\}^m$$
$$\mathbf{x} = (x_1, \dots x_k) \to \mathsf{R}(a \cdot \prod_{i=1}^{k} \mathbf{s}_i^{x_i}) \tag{11.2}$$

Discussing the choice of parameters. The authors of [BBL+14] claim that choosing their parameters as above yields to a good compromise between efficiency and security. They also state that the choice of $q = 257$ is akin to the one made by Lyubashevsky et al. for the design of the SWIFFT hash function [LMPR08]. In fact, working modulo 257 allows efficient implementations of the Fast Fourier Transform (FFT), using vector instructions available on Intel and ARM CPUs.

However, no proof of security exists for a scheme with these kind of parameters yet. Furthermore, working with a small odd moduli q implies that the rounding function $\lfloor \cdot \rceil_p$ has a noticeable bias of $1/q$.

Definition 11.3. The bias of a distribution χ over $\mathbb{Z}_p$ is defined as follows:

$$\text{bias}(\chi) = \left| \mathbb{E}\left[e^{2i\pi X/p} \right] \right|,$$

where $X \sim \chi$.

We are now going to prove that SPRING is indeed biased for the case $p = 2$. The proof still holds for $p \in \{4, 8, 16\}$.

Proposition 11.2. *Whether q is odd, the output of SPRING has a bias of $1/q$.*

Proof. Let us denote by χ the distribution of the output of SPRING. From the definition of the bias, we have:

$$\text{bias}(\chi) = \left| e^0 \chi(0) + e^{i\pi} \chi(1) \right| = |\chi(0) - \chi(1)|$$

As q is odd, it follows:

$$\text{bias}(X) = \left| \frac{q+1}{2q} - \frac{q-1}{2q} \right| = \frac{1}{q}.$$

□

Remark 1. In [BPR12], the value of q was so huge that the bias of $1/q$ was not an actual issue.

To deal with this problem, the authors of [BBL+14] propose two solutions, that we are respectively going to detail in section 11.3.2 and 11.3.3

11.3.2 SPRING-BCH.

The first way proposed by the authors of [BBL+14] is to choose the function g to be a bias reducing function. They consider a $[128, 64, 22]$ BCH code, of generator matrix G, and they choose g to be defined as follows:

$$g : \{0,1\}^{128} \to \{0,1\}^{64}$$
$$\mathbf{x} \mapsto {}^t\mathbf{x} G$$

They claim that this reduces the bias of the output of SPRING-BCH to $\frac{1}{q^{22}}$. Indeed, Alberini and Rosen proved the following result:

Proposition 11.3 ([AR13])**.** *If G is the generator of a a binary linear $[n, m, d]$ code, and if $\mathcal{D}$ is a distribution over $\{0,1\}^m$, of independent bits with a bias $\text{bias}(\mathcal{D}) \leq \epsilon$, then $\text{bias}(G.\mathcal{D}) \leq \varepsilon^d$.*

Remark 2. This holds true for any family of binary linear code. The authors of [BBL+14] choose to use a (extended) BCH code mainly for implementation-related reasons.

To recover the original biased output, an adversary would have to solve the Syndrome Decoding Problem (SDP) with respect to the dual code. This problem is particularly hard when the rate k/n of the code is close to $1/2$, so the choice of the parameters $[128, 64, 22]$ makes SPRING-BCH quite safe against this kind of attacks. On the other hand, this requires to halve the length of the output. For now the most efficient attack consists in detecting this small bias of 2^{-176}, in the output.

11.3.3 SPRING-CRT

The second idea proposed by the authors of [BBL+14] is to work on the ring R_{2q} instead of R_q. This still allows efficient implementations, while removing the bias.

Indeed, by the Chinese Remainder Theorem, there is a simple ring isomorphism between R_{2q} and $R_2 \times R_q$. Indeed, for all $(\mathbf{b}_2, \mathbf{b}_q, \bar{\mathbf{b}}_2, \bar{\mathbf{b}}_q) \in R_2 \times R_q \times R_{2q} \times R_{2q}$, such that $\mathbf{b}_2 = \bar{\mathbf{b}}_2 \pmod 2$ and $\mathbf{b}_q = \bar{\mathbf{b}}_q \pmod q$, the pair $(\mathbf{b}_2, \mathbf{b}_q) \in R_2 \times R_q$ corresponds to a single $\mathbf{b} = q \cdot \bar{\mathbf{b}}_2 + (q+1)\bar{\mathbf{b}}_q \pmod p$

From here, the authors of [BBL+14] claim that it is possible to compute an unbiased rounding over R_{2q}. They do as follows: They consider the rounding function $\lfloor \cdot \rceil_2 : R_{2q} \to R_2$,

$$\lfloor \mathbf{b} \rceil_2 = \left\lfloor q\bar{\mathbf{b}}_2 + (q+1)\bar{\mathbf{b}}_q) \right\rceil .$$

From the definition of $\lfloor \cdot \rceil_2$ given in Equation 10.12, we get:

$$\lfloor \mathbf{b} \rceil_2 = \left\lfloor \bar{\mathbf{b}}_2 + \bar{\mathbf{b}}_q + (1/q) \cdot \bar{\mathbf{b}}_q \right\rceil \mod 2$$

Choosing the coefficients of $\bar{\mathbf{b}}_q$ from $[-q/2 \dots q/2) \cap \mathbb{Z}$, such that each coefficient of $(1/q) \cdot \bar{\mathbf{b}}_q$ is in the interval $[-1/2, 1/2)$, and then

$$\lfloor \mathbf{b} \rceil_2 = \bar{\mathbf{b}}_q + \bar{\mathbf{b}}_2 \mod 2. \tag{11.3}$$

However, the structure of the ring R_{2q} may be exploited by an attacker. We briefly recall the idea of the attack below.

Birthday Attack against SPRING-CRT. This attack was already introduced by Banerjee et al. [BBL+14] in the security analysis of their scheme. We choose to recall only the global idea of the method, to point out the flaw in the design of SPRING-CRT, without getting into the details.

The main idea is to cancel the R_2 component of the output of SPRING-CRT in order to recover the bias. To this aim, Banerjee et al. propose to split the input $\mathbf{x}$ of the function in two parts: $\mathbf{x} = (\mathbf{w}|\mathbf{z})$, where $\mathbf{w}$ is of length t.

Denoting by $\mathbf{b_w}$ the polynomial $\mathbf{b}_w = \prod_i s_i^{w_i}$ and by $\mathbf{b}_z$ the polynomial $\mathbf{b_z} = \prod_i \mathbf{s}_{i+t}^{z_i}$, the SPRING-CRT function can be re-written as follows:

$$F(\mathbf{w}|\mathbf{z}) = \lfloor \mathbf{a}\mathbf{b_w}\mathbf{b_z} \rceil_2 .$$

The goal in the attack is to find two bit-strings $\mathbf{w}$ and $\mathbf{w}'$, such that $\mathbf{w} \neq \mathbf{w}'$ and $\mathbf{b_w} = \mathbf{b_{w'}} \pmod 2$. Then, for all $\mathbf{P} \in R_q$,

$$\mathbf{b_w}\mathbf{P} = \mathbf{b_{w'}}\mathbf{P} \mod 2.$$

In particular, this holds when $\mathbf{P} = \mathbf{a}\mathbf{b_z}$ for any $\mathbf{z}$. Let us denote by $\mathbf{Q}$ the polynomial $\mathbf{Q} = \mathbf{b_w}\mathbf{P}$, and by $\mathbf{Q}'$ the polynomial $\mathbf{Q}' = \mathbf{b_{w'}}\mathbf{P}$. From Equation 11.3,

$$\lfloor \mathbf{Q} \rceil = \bar{\mathbf{Q}}_2 + \bar{\mathbf{Q}}_q \mod 2 = \mathbf{y}_2 \oplus \mathbf{y}_q,$$

where $\mathbf{y}_2, \mathbf{y}_q \in \{0,1\}^n$, such that $\bar{\mathbf{Q}}_2 = \mathbf{y}_2 \pmod 2$ and $\bar{\mathbf{Q}}_q = \mathbf{y}_q \pmod 2$. We define $\bar{\mathbf{Q}}'_2, \bar{\mathbf{Q}}'_q, \mathbf{y}'_2, \mathbf{y}'_q$ accordingly. We have:

$$\lfloor \mathbf{Q} \rceil \oplus \lfloor \mathbf{Q} \rceil' = \mathbf{y}_2 \oplus \mathbf{y}_q \oplus \mathbf{y}'_2 \oplus \mathbf{y}'_q.$$

Banerjee et al. claim that $\mathbf{y}_2 = \mathbf{y}'_2$. Indeed, recall that $\bar{\mathbf{Q}}_2 = \bar{\mathbf{b}}_{\mathbf{w},2} \cdot \bar{\mathbf{P}}_2$, and $\bar{\mathbf{Q}}'_2 = \bar{\mathbf{b}}_{\mathbf{w}',2} \cdot \bar{\mathbf{P}}_2$. Moreover, $\bar{\mathbf{b}}_{\mathbf{w}',2} = \bar{\mathbf{b}}_{\mathbf{w},2} \pmod 2$. This is enough to prove that $\mathbf{y}_2 = \mathbf{y}'_2$.

It follows that, for all $\mathbf{z}$, $F(\mathbf{w}|\mathbf{z}) \oplus F(\mathbf{w}'|\mathbf{z})$, the R_2 component is cancelled out, and there are some $\mathbf{y}_q, \mathbf{y}'_q$, such that $F(\mathbf{w}|\mathbf{z}) \oplus F(\mathbf{w}'|\mathbf{z}) = \mathbf{y}_q \oplus \mathbf{y}'_q$, where $\mathbf{y}_q$ and $\mathbf{y}'_q$ both have the same bias of $1/q$. It follows that $F(\mathbf{w}|\mathbf{z}) \oplus F(\mathbf{w}'|\mathbf{z})$ has a bias of $1/q^2$.

All in all, an adversary can then recover the bias in SPRING-CRT with $2^{2^{n/2}q^4/(4n)}$ queries and space complexity, and in time $2^n q^4/2$.

Remark 3. Banerjee et al. notice that the attack would still work if one searches for pairs $(\mathbf{w}, \mathbf{w}')$ such that $\mathbf{w} \neq \mathbf{w}'$ and $(\mathbf{b}_{\mathbf{w}} + \mathbf{b}_{\mathbf{w}'})^s = 0 \pmod 2$, for some s being a power 2. The probability of finding the couple is increased, at the cost of reducing the bias, as more XORs will have to be computed. This allows them to drop the time complexity of their attack to $2^{n/t}q^{4t}/(2t)$.

11.3.4 PRGs from SPRING

Using the counter mode, it is possible to turn every SPRING PRF into a PRG. The internal state is therefore composed of two k-bit integers i, j and a unit polynomial $\mathbf{P}$ in R_q^* (or R_{2q}^* in the case of SPRING-CRT). Each time the PRG is clocked:

1. i is incremented
2. The $(\ell+1)$-th bit of j is flipped, where ℓ is the number of trailing zeroes in i.
3. The polynomial $\mathbf{P}$ is multiplied by $\mathbf{s}_j$ (resp. $\mathbf{s}_j^{-1}$) when the ℓ-th bit of j is 1 (resp. 0).

The initial value of $\mathbf{P}$ is the secret element $\mathbf{a}$, which is part of the key. As an example, we show how this internal state evolves in the first 4 rounds:

Round	Counter	Update	$\mathbf{P}$
0	000	$\mathbf{P} \leftarrow \mathbf{a}$	$\mathbf{a}$
1	100	$\mathbf{P} \leftarrow \mathbf{P} \cdot \mathbf{s}_1$	$\mathbf{a}\mathbf{s}_1$
2	110	$\mathbf{P} \leftarrow \mathbf{P} \cdot \mathbf{s}_2$	$\mathbf{a}\mathbf{s}_1\mathbf{s}_2$
3	010	$\mathbf{P} \leftarrow \mathbf{P} \cdot \mathbf{s}_1^{-1}$	$\mathbf{a}\mathbf{s}_2$
4	011	$\mathbf{P} \leftarrow \mathbf{P} \cdot \mathbf{s}_3$	$\mathbf{a}\mathbf{s}_2\mathbf{s}_3$

The rounding is then applied to the internal state. The bias-reducing function is applied when required (in the case of SPRING-BCH) and the output string is then updated.

We utilised the same kind of construction for our own SPRING PRG, described in the following chapter.

Chapter 12

Of a New SPRING PRG: SPRING with Rejection-Sampling

In this chapter, we present a new SPRING PRG called SPRING-RS. We first describe the design of this new instantiation and propose some ways to reduce its large key size. We also present a PRF derived from SPRING-RS. Finally, we discuss the security of our scheme, and present some attacks. This work has led to a publication in PQCrypto2017 [BDFK17], with Charles Bouillaguet, Pierre-Alain Fouque and Paul Kirchner.

12.1 SPRING-RS

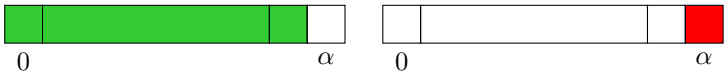
We introduce here a new PRG based on SPRING using rejection-sampling to eliminate the bias. Indeed, consider the case $q = 2^\alpha + 1$, for some integer α, a random element a of $\mathbb{Z}_q$ is either a random α-bit vector, or is equal to $2^\alpha = -1 \pmod q$ (see Figure 12.1).

Bit α is thus equal to one less often than the other bits. The probability that bit α of a is one is $1/q$, whereas, for another bit $i < \alpha$, the probability that bit i of a is one is $(q-1)/(2q)$. As such, we decided to consider the bit α as a bad value that has to be rejected, as explained below. We call this instantiation of SPRING, SPRING-RS for SPRING with rejection-sampling. It allows us to remove completely the bias. On the down side, SPRING-RS cannot be used as such as a PRF, as the rejection-sampling makes its output length variable from one call to another. Nonetheless we show that it can be used to make a reasonable PRG, when in counter-mode. Just like [BBL+14] we propose to use a counter-like mode using a Gray-code for efficiency. Then, when running SPRING in counter mode, we can compute the successive subset products with only one polynomial multiplication at each step. Finally, we show that it is still possible to build a SPRING-RS based PRF, if we are willing to reduce the length of the output.

12.1.1 Idea of the Design

Let $R = \mathbb{Z}_q[X]/(X^n + 1)$, where n is a power of two and q is a power of two plus one. Recall the SPRING function:

$$F_{\mathbf{a},\{\mathbf{s}_i\}}(x_1 \dots x_k) = \mathsf{R}\left(\mathbf{a}\prod_{i=1}^{k}\mathbf{s}_i^{x_i}\right)$$



Good values. Bad value.

Figure 12.1 – Illustration of the rejection sampling applied in SPRING-RS.

We build our PRG as described in Section 11.3.4. After the internal state has been updated, a variable number of pseudorandom bits are extracted from the new value of $\mathbf{P}$.

Extracting bits from the polynomial is done by the rounding operation. We apply the following rounding function to the n coefficients of $\mathbf{P}$ in parallel:

$$\begin{array}{rccl} & \mathbb{Z}_q & \longrightarrow & \mathbb{Z}_p \cup \{\perp\} \\ \mathsf{R}: & x & \mapsto & \begin{cases} \perp & \text{if } x = -1 \mod q \\ \lfloor px/q \rfloor & \text{otherwise} \end{cases} \end{array}$$

Algorithm 28 PRG based on SPRING using a rejection sampling instantiation.

Require: $\ell \geq 0$, d the width of available hardware vectors, the secrets parameters $\tilde{\mathbf{a}} \in R_q$ and $\tilde{\mathbf{s}} \in (R_q)^k$, Fast Fourier evaluation of the secret polynomial $\mathbf{a}$ and $(\mathbf{s}_i)_{i=0}^n$.
Ensure: An ℓ-bits long sequence of (pseudorandom) $\mathbb{Z}_p$ elements.

Initialization
$\tilde{P} \leftarrow \tilde{\mathbf{a}}$
$P \leftarrow \text{FFT}_{128}^{-1}(\tilde{P})$
$L \leftarrow$ empty string
$i \leftarrow 0$
$j \leftarrow 0$
$size \leftarrow 0$
while $size < \ell$ **do**
    # Extract output
    $(\mathbf{v_0}, \mathbf{v_1}, \ldots, \mathbf{v_{r-1}}) \leftarrow \text{Dispatch}(P)$
    for $i = 0$ to $r - 1$ **do**
        $\mathbf{v}' \leftarrow \text{Rounding}(\mathbf{v_i})$
        if $\perp \notin \mathbf{v}'$ **then**
            $L \leftarrow L \| \mathbf{v}'$
            $size \leftarrow size + d \cdot \log_2(p)$
    # Update internal state
    $i \leftarrow i + 1$
    $u \leftarrow \text{CountTrailingZeroes}(i)$
    $j \leftarrow j \oplus (1 \ll u)$
    if $j \;\&\; (1 \ll u) \neq 0$ **then**
        $\tilde{P} \leftarrow \tilde{P} \cdot \tilde{\mathbf{s}}_i$
    else
        $\tilde{P} \leftarrow \tilde{P} \cdot \tilde{\mathbf{s}}_i^{-1}$
    $P \leftarrow \text{FFT}_{128}^{-1}(\tilde{P})$
return L

This results in a sequence of n symbols. The $\perp$ are then "erased" from the output, yielding a variable number of elements of $\mathbb{Z}_p$, which are appended to the pseudorandom stream.

This produces uniformly distributed outputs in $\mathbb{Z}_p$ when the secrets $\mathbf{a}$ and the $\mathbf{s}_i$ polynomials are uniformly distributed in $\mathbb{Z}_q$. Rejecting one of the possible values (here, -1) effectively restricts the input set to $q - 1$ elements. As long as p divides $q - 1$, exactly $(q - 1)/p$ inputs yield each possible output.

Tuning for Vector Instructions. To obtain high performance implementations on modern hardware, it is necessary to be able to exploit vector instructions. In particular, it may be beneficial to tune some aspects of the function to the underlying hardware. Let d and r be integers such that $d \cdot r = n$, for any vector $\mathbf{b}$ in $\mathbb{Z}_q$, let $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{r-1}$ be d-wide vectors with coefficients in $\mathbb{Z}_q$ such

that:

$$\begin{pmatrix} \mathbf{v}_0 \\ \mathbf{v}_1 \\ \vdots \\ \mathbf{v}_{r-1} \end{pmatrix} = \begin{pmatrix} b_1 & b_2 & \dots & b_d \\ b_{d+1} & b_{d+2} & \dots & b_{2d} \\ & \vdots & & \\ b_{(r-1)d+1} & b_{(r-1)d+1} & \dots & b_n \end{pmatrix}. \tag{12.1}$$

Typically, d should be the width of available hardware vectors.

For each vector $\mathbf{v}_i$, such that $(\mathbf{v}_0, \dots \mathbf{v}_{r-1})$ represents the internal state P of the PRG, we apply the rounding function to all coefficients in parallel. If the resulting vector contains $\perp$, we reject the whole vector. Even though this also discards "good" coefficients, it allows a performance gain, because examining individual coefficients inside a hardware vector is often very inefficient. With $d = 1$, there is no wasted output. With $d = 8$ (SSE2 instructions on Intel CPUs, or NEON instructions on ARM CPUs), about 2.7% of the good coefficients are wasted. With $d = 16$ (AVX2 instructions), this goes up to 5.7%. This loss is a small price to pay compared to the twofold speedup that we get from using twice bigger hardware vectors.

12.1.2 The SPRING-RS PRG

All in all, we come up with the following construction. Let (k, n, q, p, d) be parameters such that:

$$k = 64,\ n = 128,\ q = 257,\ p \in \{2, 4, 8, 16\},\ d \in \{1, 8, 16\}.$$

[*Initialisation*] The internal state polynomial $\mathbf{P}$ is initialised to $\mathbf{a}$. The counter is initialised to 0.

[*Update*] At each step, $\mathbf{P}$ is multiplied by a secret polynomial $\mathbf{s}_i$ (or $\mathbf{s}_i^{-1}$) according to the Gray code.

[*Rounding*] The rounding operation is performed. We recover a polynomial $\mathbf{Q} \in R_p$, dispatched into buckets of d coefficients.

[*Rejection Sampling*] If there is a coefficient $Q_i = -1$ in a bucket, the whole bucket is rejected. Otherwise, the output stream is updated with the binary expansion of the coefficients of "good" buckets.

The whole procedure is detailed in Algorithm 28. We also illustrate how one iteration works in Figure 12.2.

Remark 1. In order to speedup computation, the secret polynomials are stored in their FFT evaluated form. Then, only one FFT per step has to be performed to recover the coefficients of the polynomial P. This was already the case in previous SPRING instantiation [BBL+14].

12.1.3 Reducing the Key's Size

One of the main disadvantage of SPRING is the large size of its key (8 kB). We present here several ways to reduce the size of the key for all instantiations of SPRING. The key is composed of $k + 1$ secret polynomials $\mathbf{a}, \mathbf{s}_1, \dots, \mathbf{s}_k$ over R_q^*. Each of such polynomials requires at least 1024 bits.

The most intuitive and most efficient way to reduce the key size of SPRING is to use a 128-bit master secret key denoted by K_m, and use it to generate pseudo-randomly the secret polynomials using... another PRG. This is a bit unsatisfying: why not use the other PRG in the first place? However, this would be beneficial if the other PRG is slow or even not cryptographically secure (consider the Mersenne Twister for instance).

In order to drop the need for another PRG, it would be natural to use SPRING to "bootstrap" itself and generate its own secret polynomials. We propose two ways to do so.

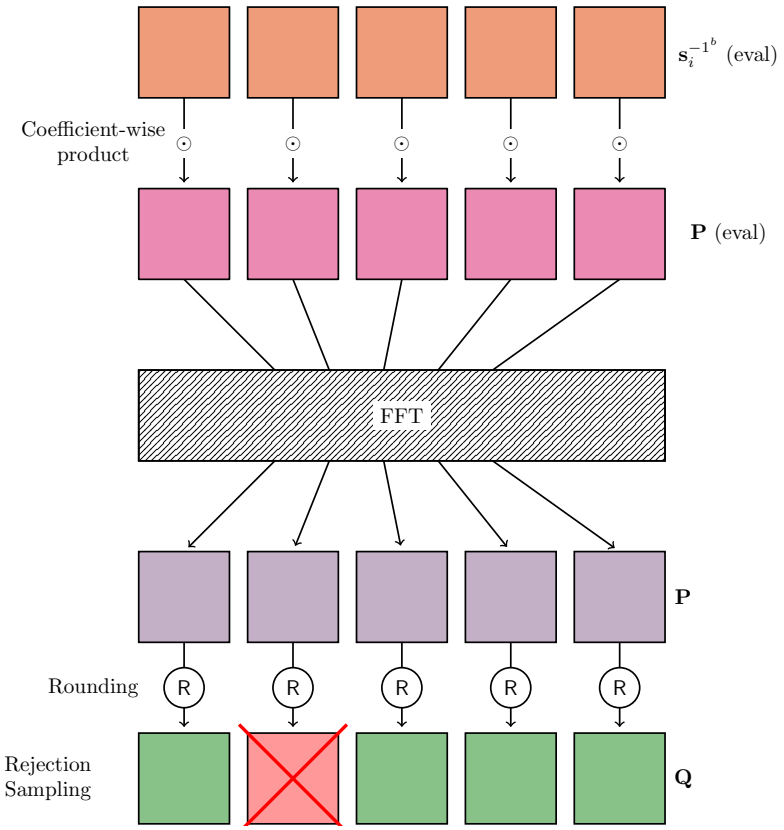


Figure 12.2 – Illustration of one iteration of SPRING-RS.

Using five secret polynomials. One possibility is to consider that the "master" key is composed of the 5 polynomials $\mathbf{a}_0, \mathbf{s}_{0,1}, \mathbf{s}_{0,2}, \mathbf{s}_{0,3}, \mathbf{s}_{0,4}$. Then, we may evaluate the "mini-SPRING" function $F^0_{\mathbf{a}_0,\mathbf{s}_0}$ such that, $F^0_{\mathbf{a}_0,\mathbf{s}_0}(\mathbf{x}) = S\left(\mathbf{a}_0 \cdot \prod_{i=1}^4 \mathbf{s}_{0,i}^{x_i}\right)$, for all $x \in \{0,1\}^4$. Using this small instantiation of SPRING-RS with $p = 16$, we may generate up to 8192 pseudo-random bits. This is large enough to forge seven polynomials with very high probability. We call $\mathbf{a}_1, \mathbf{s}_{1,1}, \ldots, \mathbf{s}_{1,6}$ these new polynomials. We reiterate the process with the mini-SPRING function $F^1_{\mathbf{a}_1,\mathbf{s}_1}$, and the output given by this PRG is large enough to forge between thirty and thirty-one polynomials. If we reiterate the process once more with those new polynomials, we will be able to forge the 65 polynomials $\mathbf{a}, \mathbf{s}_1, \ldots, \mathbf{s}_k$ of the full SPRING-RS. Using such a trick, we can substantially reduce the size of the key (from about 8 kB to about 700B).

Using three secret polynomials. It is possible to push this idea a bit further assuming the circular security of SPRING-RS. In that case, the "master" key is composed of the three secret polynomials $\mathbf{a}, \mathbf{s}_1, \mathbf{s}_2$, and we define the nano-SPRING function $F^0_{\mathbf{a},\mathbf{s}}$ such that, $F^0_{\mathbf{a},\mathbf{s}}(\mathbf{x}) = S\left(\mathbf{a} \cdot \mathbf{s}_1^{x_1} \cdot \mathbf{s}_2^{x_2}\right)$, for all $\mathbf{x} \in \{0,1\}^2$. Using this small instantiation of SPRING-RS, we can generate an output long enough to forge a new polynomial. This will be the next secret polynomial, $\mathbf{s}_3$. We reiterate the process with the micro-SPRING function $F^1_{\mathbf{a},\mathbf{s}}$, which is such that $F^1_{\mathbf{a},\mathbf{s}}(\mathbf{x}) = S(\mathbf{a} \cdot \prod_{i=1}^3 s_i^{x_i})$. We do not reset the Gray Counter, as long as the previous values will only give the output of $F^0_{\mathbf{a},\mathbf{s}}$. The new output thus generated is long enough to forge $\mathbf{s}_4$. If we reiterate the process once more, we will get an output long enough to generate two more polynomials. We reiterate it over again —two more times should be enough— until all the $\mathbf{s}_i$ are forged. We never reset the Gray Counter, otherwise an adversary may know all the $\mathbf{s}_i$ thus obtained.

12.1.4 A SPRING-RS PRF

It is clear that the rejection-sampling process often yields sequences of less than $n \log_2 p$ output bits. This makes it less-than-ideal to implement a PRF, which is always expected to return a specified amount of output bits. However, building a PRF is still possible if we accept a reduction in output size.

With the chosen parameters, we know that at least 96 $\mathbb{Z}_p$ elements survive the erasure process with probability greater than $1 - 2^{-156}$. Therefore, a possible yet inelegant workaround consists in making a PRF that returns the first 96 non-erased outputs. In the unlikely event that less than 96 truncated coefficients are available, the PRF output is padded with zeroes. The probability of this event is so low that it is undetectable by an adversary.

Implementing such a PRF efficiently is likely to be difficult, because of the amount of bit twiddling and juggling that is involved. Furthermore, unlike the CTR mode, we need to compute the product of many polynomials. To make this implementation efficient, we choose to store the discrete logarithms of the secret polynomials, as it was proposed in [BBL+14] and [BGL+14] so that the subset-product becomes a subset-sum. Each exponentiation by the final summed exponents is computed by a table look-up.

12.2 Security Analysis

Secure PRF and PRG over a polynomial ring R_q are described in [BPR12], assuming the hardness of Ring-LWE problem. However, for the BPR family to be secure, we need to make two assumptions:

1. The parameter q must be large (exponential in the input length k).
2. The $\mathbf{s}_i$ are drawn from the error distribution of the underlying RLWE instantiation.

In [BBL+14], Banerjee et al. show that relaxing those statements does not seem to introduce any concrete attack against the SPRING family, and its security seems to be still very high, even though SPRING is not provably secure. In our instantiation we slightly weaken SPRING by introducing two changes : (1) the rounding function S we use returns more bits, so more information about the internal state is returned, (2) some coefficients are rejected, so using side-channel timing attacks, an adversary may learn that some coefficients are equal to -1 (the rejected value). However, as we shall discuss in the following part, we do not think that this may undermine the security of SPRING. In fact, we believe that SPRING-RS is actually more secure than the previously proposed variants. Indeed, the best attack against SPRING-CRT requires about 2^{128} bit operations and 2^{119} space, the best attack against SPRING-BCH consists in detecting a small bias of 2^{-176}, and we claim that our best attack against SPRING-RS requires 2^{64} operations, but succeeds with advantage about 2^{-900} for the adversary.

Recall that the SPRING-RS function is given by:

$$F_{\mathbf{a},\{\mathbf{s}_i\}}(x_1, \ldots, x_k) = \mathsf{R}\left(\mathbf{a} \prod_i \mathbf{s}_i^{x_i}\right),$$

where $\mathbf{a}, \mathbf{s}_i \in R_q^*$. We have $q = 257$, $n = 128$ and $p \in \{2, 4, 8, 16\}$.

12.2.1 Security of the RLWR Instance.

We first aim to estimate the security of the underlying Ring-LWR instance, and not the actual SPRING scheme. The same kind of security analysis has been proposed by the authors of [BBL+14], for their sets of parameters. The authors analyse the complexity of various attacks: lattice reductions, BKW [BKW03], and algebraic attacks [AG11]. We propose an update of these results, for all of our sets of parameters.

Here, we assume that an attacker can generate 2^k $\mathbf{a} \in R_q$ polynomials, that she can control, then having access to a Ring-LWR oracle which, when queried with $\mathbf{a}$ returns $\mathbf{c} = \lfloor \mathbf{a} \cdot \mathbf{s} \rceil_p$, she aims to recover the secret polynomial $\mathbf{s}$. Denoting by $\mathbf{e}$ the error term $\mathbf{e} = \mathbf{a} \cdot \mathbf{s} - \lfloor \mathbf{a} \cdot \mathbf{s} \rceil_p \cdot p$. The polynomial $\mathbf{e}$ is in R_q, but each one of its coefficients belongs to $[-q/(2p), q/(2p))$. The considered equation can be re-written: $\mathbf{c} = \mathbf{a} \cdot \mathbf{s} + \mathbf{e}$, or, with matrices and vectors:

$$\mathbf{c} = A \cdot \mathbf{s} + \mathbf{e}.$$

Table 12.1 – Complexity of the (Ring-)LWR instance, using the BKW algorithm.

p	2	4	8	16
Time	$2^{141.6}$	$2^{108.6}$	$2^{93.5}$	$2^{77.7}$
Space	$2^{122.9}$	$2^{99.2}$	$2^{83.5}$	$2^{67.5}$

Considering this as an LWE instance in dimension n, with modulus q and error rate $\alpha = 1/(2p)$, it is possible to estimate the complexity of all generic attacks against LWE using public LWE-estimator such as [APS15].

We found out the best attack against this LWR instance is BKW, for all values of $p \in \{2, 4, 8, 16\}$. In Table 12.1, we present the time and space complexity of this attack. This is an update of the estimations proposed in [BDFK17], using Albrecht et al. LWE-estimator [APS15].

12.2.2 Birthday Type Attack on SPRING-RS

We assume that the adversary knows that she is facing a SPRING-RS PRG, and her goal is to predict a part of the output.

Intuition of the attack. Let $t < k$ be a positive integer. For all $\mathbf{x} \in \{0,1\}^n$, $\mathbf{x}$ can be decomposed the following way:

$$\mathbf{x} = (x_1, \ldots, x_t, x_{t+1}, \ldots, x_{k-1}, x_k).$$

If we denote by $\mathbf{z}$ the bit-string of length $(k-t-1)$, such that $z_1 = x_{t+1}$, $z_2 = x_{t+2}$, $\ldots$, $z_{k-t-1} = x_{k-1}$, and by $\mathbf{b_z}$ the polynomial of R_q such that:

$$\mathbf{b_z} := \prod_{i=1}^{k-t-1} \mathbf{s}_{i+t}^{z_i} = \prod_{i=t+1}^{k-1} \mathbf{s}_i^{x_i}.$$

The idea is to find two strings $\mathbf{z}_1$ and $\mathbf{z}_2$ such that:

1. $\mathbf{z}_1 \neq \mathbf{z}_2$
2. $\mathbf{b}_{\mathbf{z}_1} = \mathbf{b}_{\mathbf{z}_2}$

Then we will also have $\mathbf{b}_{\mathbf{z}_1}\mathbf{s}_k = \mathbf{b}_{\mathbf{z}_2}\mathbf{s}_k$. The problem is that given a vector $\mathbf{z}$ we do not know $\mathbf{b}_z$. We then have to find a $\mathbf{z}_1$ and a $\mathbf{z}_2$ that satisfy conditions 1 and 2, without being actually able to check the latter.

Let $\mathbf{z}$ be in $\{0,1\}^{k-t-1}$, let $\mathbf{w} \in \{0,1\}^t$ be an arbitrary string, and let $\mathbf{x} \in \{0,1\}^n$ be such that $\mathbf{x} = (\mathbf{w}|\mathbf{z}|0)$. Consider the string $\mathbf{y}_{\mathbf{w},\mathbf{z}}$ defined as the output of $F_{\mathbf{a},(\mathbf{s}_i)}(\mathbf{x})$. For simplicity, we ignore the rejection sampling for the moment. $\mathbf{y}_{\mathbf{w},\mathbf{z}}$ is then a bit-string of size $\log p \cdot n$ that satisfies:

$$\mathbf{y}_{\mathbf{w},\mathbf{z}} = \mathsf{R}(\mathbf{a} \prod_{i=1}^{t} \mathbf{s}_i^{w_i} \cdot \mathbf{b_z}).$$

Now, if $\mathbf{w}$ is a fixed bit-string of length t, for two bit-strings $\mathbf{z}_1$ and $\mathbf{z}_2$ such that $\mathbf{b}_{\mathbf{z}_1} = \mathbf{b}_{\mathbf{z}_2}$, we will have:

$$\mathbf{y}_{\mathbf{w},\mathbf{z}_1} = \mathbf{y}_{\mathbf{w},\mathbf{z}_2}. \tag{12.2}$$

Suppose now that $\mathbf{z}_1$ and $\mathbf{z}_2$ are two bit-strings such that $\mathbf{z}_1 \neq \mathbf{z}_2$ and the relation given by Equation 12.2 holds for sufficiently many values of $\mathbf{w}$, we claim that $\mathbf{b}_{\mathbf{z}_1} = \mathbf{b}_{\mathbf{z}_2}$ with high probability.

Taking $2^t = \log q / \log p$ different values of $\mathbf{w}$ should be enough. In this setting, if there are two distinct bit-strings $\mathbf{z}_1$ and $\mathbf{z}_2$ such that $\mathbf{y}_{\mathbf{w},\mathbf{z}_1} = \mathbf{y}_{\mathbf{w},\mathbf{z}_2}$ for all values of $\mathbf{w}$ then, $\mathbf{b}_{\mathbf{z}_1} = \mathbf{b}_{\mathbf{z}_2}$ with high probability. Actually the number false positive is at most $2^{2(k-t-1)}/p^{\log q/\log p \cdot n}$.

We consider for now on that $2^t \simeq \log q / \log p$. In particular, this means that $t = 1$, when $p = 16$, and in this case, $\mathbf{w}$ is only one bit. Indeed $\log q / \log p \simeq 8/4 = 2$.

Description of the attack. Formally, we assume that the attacker has access to a bit-string $\mathbf{y}$ of length $m > 2^{k-1} \cdot n \log p$ which is the output of a SPRING-RS PRG G, of parameter k, n, p, used in counter mode initialised at 0.

The length of the sequence generated by 2^{k-1} calls to $F_{\mathbf{a},(\mathbf{s}_i)}$ is at most $2^{k-1} \cdot n \log p$, and exactly this value if no rejection-sampling has occurred (which is very unlikely).

Let us denote by d the size of a vector with respect to the available hardware instructions. Recall that each time a "bad" coefficient appears, all d coefficients of the vector are rejected.

The attacker proceeds as follows:

1. She gets the subsequence $\mathbf{y}'$ of $\mathbf{y}$ of length $2^{k-1} \cdot n \log p$
2. She stores each possible subsequence of $\mathbf{y}'$ of length $\log q \cdot n$, with a step of $d \log p$, into a hash table: the first subsequence goes from index 1 to index $\log q \cdot n$, the next one goes from index $d \log p$ to $\log q \cdot n + d \log p$, and so on.
3. If a collision occurs, then there is a collision between the internal polynomials with high probability, and the attacker will be able to predict that another collision will occur further in the output string.

The probability that no rejection has been performed on a given $\log q \cdot n$-bit sequence is:

$$\mathbb{P}[\text{no rejection}] = \left(1 - \frac{1}{q}\right)^{2^t n}. \tag{12.3}$$

The probability that the attack succeeds is then the probability that a collision will occur, and that no rejection sampling has been performed on the sequence. It is given by the following formula:

$$\begin{aligned} \mathbb{P}[\text{Success}] &= \mathbb{P}[\text{collision } \cap \text{ no rejection}], \\ &= \mathbb{P}[\text{collision} \mid \text{no rejection}] \cdot \mathbb{P}[\text{no rejection}]. \end{aligned} \tag{12.4}$$

We need to estimate the probability of the event: "A collision occurs, knowing that the strings are of length $\log q \cdot n$." This can be represented in a *balls-into-bins* model (see Section 6.1.2). We are given $2^k/(d \log p)$ balls, that we want to dispatch into q^n bins. Assuming that we throw them uniformly at random (i.e. that each strings "looks like" a uniformly random one), the probability that two balls will be in the same bin satisfies:

$$\begin{aligned} \mathbb{P}[\text{Two balls collide}] &= \mathbb{P}[\bigcup_{i<j} \text{ ball } i \text{ and ball } j \text{ are in the same bin}], \\ &\leq \sum_{i<j} \mathbb{P}[\text{ball } i \text{ and ball } j \text{ are in the same bin}], \\ &\leq \binom{2^{k-1}/(d \log p)}{2} \frac{1}{q^n}, \\ &\leq \frac{2^{2(k-1)}}{2d^2 \log^2 p} \cdot \frac{1}{q^n}. \end{aligned} \tag{12.5}$$

Remark 1. This holds true only if the output behaves like a uniformly random string. Otherwise the probability of collision increases. In the case of SPRING-RS, the rejection sampling ensures that the output is uniformly distributed.

From Equations 12.3, 12.4 and 12.5, the probability that the attack will succeed will be:

$$\mathbb{P}[\text{Success}] \leq \frac{2^{2k-3}}{q^n d^2 \log^2 p} \cdot \left(1 - \frac{1}{q}\right)^{2^t n}.$$

We give the value of this probability in Table 12.2 for all possible parameters of SPRING-RS.

Table 12.2 – Probability that the attack will succeed for various SPRING-RS parameters.

	$p = 2$	$p = 4$	$p = 8$	$p = 16$
$d = 1$ (No vector instruction)	2^{-3046}	2^{-941}	2^{-913}	2^{-909}
$d = 8$ (Intel SSE2 & ARM NEON)	2^{-3052}	2^{-947}	2^{-919}	2^{-915}
$d = 16$ (Intel AVX2)	2^{-3054}	2^{-949}	2^{-921}	2^{-917}

12.2.3 Side-Channel Leakage

An obvious drawback of rejection sampling is the irregular rate at which output is produced. It is conceivable that a timing attack could reveal some information about the internal state of the PRG.

We consider a powerful adversary who is able to discover exactly which coefficients are rejected (she knows both their locations and the actual value of the counter $\mathbf{x}$).

Therefore, the attacker has access to equations of the form $(\mathbf{a}\prod_i \mathbf{s}_i^{x_i})_j = -1$ over $\mathbb{Z}_q$ where the x_i exponents are known and the unknowns are the coefficients of $\mathbf{a}$ and $\mathbf{s}_i$. Denote by $wt(\mathbf{x})$ the Hamming weight of $\mathbf{x}$, then each of these equations can be converted into a polynomial of degree $1 + wt(\mathbf{x})$ over the coefficients of $\mathbf{a}$ and all $\mathbf{s}_i$.

If the first $2^t n \log_2 p$ key-stream bits are observed, then we expect $n2^t/q$ coefficients to be rejected. This yields this many polynomial equations in $n(t + 1)$ variables, of which $n\binom{t}{\delta-1}/q$ are expected to be of degree δ. With the chosen parameters, $\delta \geq 12$ is needed to obtain more equations than unknowns. With $\delta = 12$, we obtain the smallest possible over-determined system, with 2032 polynomial equations in 1664 unknowns, of degree mostly larger than 2. Note that the ideal spanned by these polynomial is not guaranteed to be zero-dimensional. No known technique is capable of solving arbitrary systems of polynomial equations of this size. In particular, the complexity of computing a Gröbner basis of this many equations can be roughly estimated [Fau99, BFS04]: it is about 2^{2466}.

When t grows up to 64, the system becomes more over-determined: with $t = k = 64$, the largest possible value, we obtain 2^{63} equations in 8320 variables, with degrees up to 65. Storing this amount of data is completely unpractical. Neglecting this detail, we argue that a Gröbner basis computation will crunch polynomials of degree larger than 12: there are 2^{128} monomials of degree 12 in 8320 variables and only 2^{114} degree-12 multiples of the input equations (the computation generically stops when these two quantities match).

As such, we expect any Gröbner basis computation to perform, amongst others, the reduction to row echelon form of a sparse matrix of size $2^{114} \times 2^{128}$, a computationally unfeasible task.

12.3 Implementation and Results

12.3.1 Performances

Just like the designers of the original SPRING did, we implemented our variant using SIMD instructions available on most desktop CPUs. We borrow most implementation techniques from the designers of SPRING. This is in particular the case of an efficient implementation of polynomial multiplication in R^*_{257} thanks to a vectorized FFT-like algorithm using the SSE2 instructions set available in most desktop CPUs. These instructions operate on 128-bit wide "vector registers", allowing us to perform arithmetic operations on batches of eight $\mathbb{Z}_{257}$ coefficients in a single instruction.

We pushed the implementation boundary a little further by writing a new implementation of this operation using the AVX2 instructions set, providing 256-bit wide vector registers. These instructions are available on Intel CPUs based on the "Haswell" microarchitecture or later. Without surprise, this yields a twofold speedup and raises a few interesting programming problems.

Note that the AVX2-optimised FFT algorithm can be back-ported to all the cryptographic constructions relying on the same polynomial multiplication modulo 257, such as [BBL+14, LMPR08, LBF08], yielding the same 2× speedup.

Table 12.3 – Implementation results for SPRING variants, in Gray code counter mode (CTR). Speeds are presented in processor cycles per output byte. Starred numbers indicate the use of AES-NI instructions. Daggers indicate the use of AVX2 instructions.

	SPRING-BCH	SPRING-CRT	AES-CTR	SPRING-RS
ARM Cortex A7	445	[not implemented]	41	59
Core i7 "Ivy Bridge"	46	23.5	1.3*	6
Core i5 "Haswell"	19.5†	[not implemented]	0.68*	2.8†

We compare our implementation of SPRING-RS with previous SPRING implementations, and with AES in counter mode, in Table 12.3. We can see that our implementation of SPRING-RS is competitive with AES especially on ARM, where no NI instructions are available.

12.3.2 Implementation Details

Our only innovation compared to the implementation of [BBL+14] is the implementation of the rejection-sampling process, which is straightforward, as well as an implementation of fast polynomial multiplications using AVX2 instructions, that we describe next.

The problem comes down to computing a kind of FFT of size 128 modulo 257. We store $\mathbb{Z}_{257}$ in 16-bit words, in zero-centered representation. Using SSE2 instructions, and hardware vector registers holding 8 coefficients, a reasonable strategy is to perform one step of Cooley-Tukey recursive division, after which two size-64 FFTs have to be computed. This is done efficiently by viewing each input as an 8×8 matrix, performing 8 parallel size-8 FFTs on the rows, multiplying by the twiddle factors, transposing the matrix, and finally performing 8 parallel FFTs. The use of vector registers allows to perform the 8 parallel operations efficiently.

When AVX2 instructions are available, we have access to vector registers holding 16 coefficients. Several strategies are possible, and we describe the one we actually implemented. We view the input of a size-128 FFT as a 16×8 matrix. We perform 16 parallel size-8 FFTs on the rows, which is easy using the larger vector registers. Transposing yields a 8×16 matrix, and we need to perform 8 parallel size-16 FFTs on its rows.

This is the non-obvious part. To make full use of the large vector registers, we decided to store two rows in each vector register. Because the first pass of this size-16 FFT requires operations between adjacent rows, a bit of data juggling is necessary. We store rows 0 and 8 in the first register, rows 1 and 9 in the second, etc. This is done with the `VPERM2I128` instruction. Because the last pass requires operations between rows i and $i + 8$, which is again not possible if they are in the same register, we perform the same data-juggling operation again. This puts the rows back in the right order.

Performing the rejection sampling is easy. With AVX2 instructions, we use the `VPCMPEQW` to perform a coefficient-wise comparison with $(-1, \ldots, -1)$, and a `VPMOVMSKB` to extract the result of the comparison into a 16-bit integer. It is slightly different on an ARM processor with the NEON instructions set as there is nothing like `VPMOVMSKB`. However, we achieve the same result, using some tricks. We first convert the `int16x8_t` NEON vector into a `int8x16_t` NEON vector, and then we `ZIP` the low and the high part of this vector. This gives two `int8x8_t` NEON vector, `d0` and `d1`. We use the `VSRI` on `d0` and `d1` with constant 4, then we transfer the low and the high part of the obtained vector in ARM registers. Then, we obtain what we need using shift and xor. When only 128-bit vectors are available, the rejection sampling is easier to program with $d = 8$ (where d is the vector width), whereas $d = 16$ is slightly more programmer-friendly when 256-bit vectors are available. This is not a very hard constraint though, as both values of d can be dealt with efficiently using both instructions sets.

Chapter 13

Of the BLISS Signature Scheme and Side-Channel Leakage

In this chapter, we recall how the BLISS signature scheme works. We also explain why it is vulnerable to side-channel attacks, and we introduce the principal motivation of the work described in the next and last chapter.

13.1 The BLISS Signature Scheme

BLISS is a signature scheme proposed by Ducas, Durmus, Lepoint and Lyubashevsky [DDLL13], based on the RLWE problem. It is based on the "Fiat-Shamir with abort" paradigm of Lyubashevsky [Lyu09]. In particular, signature generation involves a *rejection sampling* step.

In this section, we first recall the idea of Lybashevsky "Fiat-Shamir with abort" scheme, and we present the BLISS signing algorithm.

13.1.1 Lattice-based Signature Using the Fiat-Shamir Heuristic

Let us first recall an useful trick, which is an important pillar of the "Fiat-Sharmir with abort" paradigm.

Basics about identification schemes. A standard three rounds identification scheme is a protocol between two entities: the prover $\mathcal{P}$ and the verifier $\mathcal{V}$, which consists in commit, challenge and response stages, plus a verification step. To prove her identity to $\mathcal{V}$, $\mathcal{P}$ has to prove that she knows some secret information sk (her secret key) corresponding to some public information pk (her public key), without revealing any information on sk. To understand how these kinds of protocols work, we consider, as an example, the Schnorr identification scheme [Sch89].

Let q be a large prime number, and let us assume that the discrete logarithm problem in $\mathbb{Z}_q$ cannot be solved efficiently by any adversary. Let p be a large prime divisor of $q-1$, and let $\mathsf{sk} \sim \mathcal{U}(\mathbb{Z}_p)$ be $\mathcal{P}$'s secret key. Let g be a generator of $\mathbb{Z}_q$ of order p, and $h = g^s$. $\mathcal{P}$ publishes the public key $\mathsf{pk} = (q, g, h)$.

The interaction between $\mathcal{P}$ and $\mathcal{V}$ is basically the following:

[*Commit.*] $\mathcal{P}$ generates $x \hookleftarrow \mathcal{U}(\mathbb{Z}_p)$ and sends $y = g^x$ to $\mathcal{V}$.

[*Challenge.*] $\mathcal{V}$ generates $c \hookleftarrow \mathcal{U}(\mathbb{Z}_p)$ and sends c to $\mathcal{P}$.

[*Response.*] $\mathcal{P}$ computes $z = y + c\mathsf{sk} \pmod p$ and sends z to $\mathcal{V}$.

[*Verification step.*] $\mathcal{V}$ computes g^z and checks whether $g^z = yh^c$.

Using the *Fiat-Shamir heuristic* [FS86], it is possible to convert this interactive proof of knowledge into a digital signature by replacing $\mathcal{V}$'s random challenge by a hash value, $c = H(y, \mu)$, where μ is the message to sign, and H some known hash function. Then $\mathcal{P}$ sends (z, c, μ) to $\mathcal{V}$. To check the signature, $\mathcal{V}$ can compute: $y' = h^{-c}g^z$, and check whether $\mathcal{H}(y', \mu)$ is equal to c.

Lattice-based signature and Fiat-Shamir with abort. In [Lyu08], Lyubashevsky presented a three rounds identification scheme using lattices. He encountered a small issue however. The response $\mathbf{z}$ to a given challenge $\mathbf{c}$ depends on the secret. To cope with this problem, Lyubashevsky proposed to use rejection sampling, to change the distribution of the response. This implies that with a certain probability the protocol has to be abort and restarted.

To understand better these notions, let us first consider this simple example heavily inspired by [Lyu12]. Let q be a positive integer, and D_r, be the centred discrete Gaussian distribution of parameter r. We first consider the three rounds identification protocol where $\mathcal{P}$ has to prove to $\mathcal{V}$ that she knows $\mathsf{sk} = S$ where S is an m-by-n matrix with small coefficients over $\mathbb{Z}_q$. Her public key is given by $\mathsf{pk} = (A, T)$, where A is an n-by-m matrix over $\mathbb{Z}_q$, and T is the n-by-n matrix which satisfies $T = AS \pmod q$. The interaction between $\mathcal{P}$ and $\mathcal{V}$ is basically the following:

[*Commit.*] $\mathcal{P}$ picks $\mathbf{x} \hookleftarrow D_r^m$ and sends $\mathbf{y} = A\mathbf{x} \mod q$ to $\mathcal{V}$.

[*Challenge.*] $\mathcal{V}$ generates $\mathbf{c} \hookleftarrow \mathscr{U}(\mathbb{Z}^n)$ and sends $\mathbf{c}$ to $\mathcal{P}$.

[*Response.*] $\mathcal{P}$ computes $\mathbf{z} = S\mathbf{c} + \mathbf{x}$, and then performs a rejection sampling step. If $\mathbf{z}$ fails the test, she aborts and restarts the protocol. Else she sends $\mathbf{z}$ to $\mathcal{V}$.

[*Verification step.*] $\mathcal{V}$ checks that $\|\mathbf{z}\|$ is small (i.e. $\mathbf{z}$ follows the distribution D_r^m) and that $A\mathbf{z} = T\mathbf{c} + \mathbf{y} \mod q$.

The rejection sampling step is crucial for the security of this protocol. Indeed, whenever $\mathbf{c} \sim \mathscr{U}(\mathbb{Z}^n)$, and $\mathbf{x} \sim D_r^m$, the distribution of $\mathbf{z} = S\mathbf{c} + \mathbf{x}$ will be D_r^m shifted by the vector $S\mathbf{c}$, and thus depends on the distribution of S. We call this shifted distribution $D_{r,S\mathbf{c}}^m$. Rejection sampling is then used to make $\mathbf{z}$ appears as if it was sampled from D_r^m.

Using the Fiat-Shamir heuristic, it is possible to convert this protocol into the following signature scheme. Let H be a known hash function which outputs elements of $\mathbb{Z}^n$ with small norm.

[*Initialisation.*] Pick $\mathbf{x} \hookleftarrow D_r^m$ and set $\mathbf{y} = A\mathbf{x} \mod q$.

[*Hash step.*] Set $\mathbf{c} = H(A\mathbf{y} \mod q, \mu)$, where μ is the message to sign.

[*Rejection Sampling.*] Set $\mathbf{z} = S\mathbf{c}+\mathbf{y}$. Return the signature $(\mathbf{z}, \mathbf{c})$ with probability $D_r^m(\mathbf{z})/D_{r,S\mathbf{c}}^m$, otherwise restart.

To check whether the signature is valid, the verifier checks if $\mathbf{c} = H(A\mathbf{z} - T\mathbf{c} \pmod q, \mu)$.

Remark 1. The signature scheme we describe above is basically the one from [Lyu12]. The BLISS signature scheme from [DDLL13] that we present below is an optimised ring variant of this scheme.

13.1.2 High-Level Description of BLISS

We consider the usual ring $R = \mathbb{Z}[X]/(X^n + 1)$, and R_q the ring of the elements of R modulo q.

Keys generation. BLISS keys generation is based on the NTRU assumption: if one picks two short secret polynomials $\mathbf{s}_1, \mathbf{s}_2 \in R_q$, such that the coefficients of $\mathbf{s}_1$ and $\mathbf{s}_2$ are small compared to q, and such that $\mathbf{s}_1$ is invertible, then the public key $\mathbf{v}$ defined as the quotient $\mathbf{s}_2/\mathbf{s}_1 \pmod q$ will appear to be pseudorandom.

In BLISS, the secret key consists in two polynomials $(\mathbf{s}_1, \mathbf{s}_2) \in R^2$ that are sparse, and of known density. Their non-zero coefficients are also quite small (in $[-2, 2]$ for $\mathbf{s}_1$ and in $[-3, 5]$ for $\mathbf{s}_2$), and the public key $\mathbf{v}_1$ is basically $\mathbf{v}_1 = 2\mathbf{v}_q \pmod{2q}$, where $\mathbf{v}_q = \mathbf{s}_2/\mathbf{s}_1 \pmod q$. We explain how the keys are generated (in a simplified version) in Algorithm 29

Algorithm 29 BLISS keys generation algorithm, simplified version.

function KeyGen( )
  do
    Sample $\mathbf{f}, \mathbf{g} \in R$ uniformly with $\lceil \delta_1 n \rceil$ coefficients in $\{-1, 1\}$, $\lceil \delta_2 n \rceil$ coefficients in $\{-2, 2\}$, and what remains is 0.
    Set $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow (\mathbf{f}, 2\mathbf{g})$.
    Set $\mathbf{s} \leftarrow (\mathbf{s}_1, \mathbf{s}_2)$.
  while $\mathbf{f}$ is not invertible mod q
  Set $\mathbf{v}_q \leftarrow (2\mathbf{g} + 1)/\mathbf{f} \pmod q$
  Set $\mathbf{v}_1 \leftarrow 2\mathbf{a}_q \pmod 2 q$
  return $(\mathsf{pk} = \mathbf{v}_1, \mathsf{sk} = \mathbf{s})$

Signing algorithm. To sign a message μ, one must first generate commitment values $\mathbf{y}_1, \mathbf{y}_2 \in R$, according to a discrete Gaussian distribution D_r^n. A hash $\mathbf{c}$ of the message, together with a compressed version of $\mathbf{u} = -\mathbf{v}_q\mathbf{y}_1 + \mathbf{y}_2 \pmod q$ is then computed. Then, the pair $(\mathbf{z}_1, \mathbf{z}_2)$ is generated, such that $\mathbf{z}_i = \mathbf{y}_i + (-1)^b \mathbf{s}_i \mathbf{c}$, where b is a random bit. To ensure that $(\mathbf{z}_1, \mathbf{z}_2, \mathbf{c})$ is independent of $\mathbf{s}$, the procedure is restarted with probability:

$$1 - \frac{1}{M \exp(-||\mathbf{sc}||^2/(2\sigma^2)) \cosh(\langle \mathbf{z}, \mathbf{s}c \rangle)/\sigma^2},$$

where $\mathbf{s} = (\mathbf{s}_1 | \mathbf{s}_2)$ and $\mathbf{z} = (\mathbf{z}_1 | \mathbf{z}_2)$. Finally the signature consists in the triplet $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$, where $\mathbf{z}_2^\dagger$ is a compressed version of $\mathbf{z}_2$.

Algorithm 30 BLISS signing algorithm.

function Sign$(\mu, pk = \mathbf{v}_1, sk = \mathbf{s} = (\mathbf{s}_1, \mathbf{s}_2))$
  $\mathbf{y}_1, \mathbf{y}_2 \leftarrow D_r^n$
  $\mathbf{u} = \zeta \cdot \mathbf{v}_1 \cdot \mathbf{y}_1 + \mathbf{y}_2 \mod 2q$
  $\mathbf{c} \leftarrow H(\lfloor \mathbf{u} \rceil_{2^d} \mod p, \mu)$
  choose a random bit b
  $\mathbf{z}_1 \leftarrow \mathbf{y}_1 + (-1)^b \mathbf{s}_1 \mathbf{c}$
  $\mathbf{z}_2 \leftarrow \mathbf{y}_2 + (-1)^b \mathbf{s}_2 \mathbf{c}$
  continue with probability $1/(M \exp(-\|\mathbf{sc}\|^2/(2\sigma^2)) \cosh(\langle \mathbf{z}, \mathbf{sc} \rangle/\sigma^2)$; otherwise **restart**
  $\mathbf{z}_2^\dagger \leftarrow (\lfloor \mathbf{u} \rceil_{2^d} - \lfloor \mathbf{u} - \mathbf{z}_2 \rceil_{2^d}) \mod p$
  return $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$

Note that:

$$\begin{aligned} -\mathbf{v}_q\mathbf{z}_1 + \mathbf{z}_2 &= -\mathbf{v}_q(\mathbf{y}_1 + (-1)^b\mathbf{s}_1) + \mathbf{y}_2 + (-1)^b\mathbf{s}_2 \mod q \\ &= -\mathbf{v}_q\mathbf{y}_1 + \mathbf{y}_2 + (-1)^b(-\mathbf{v}_q\mathbf{s}_1 + \mathbf{s}_2) \mod q \\ &= -\mathbf{v}_q\mathbf{y}_1 + \mathbf{y}_2 + (-1)^b(-\mathbf{s}_2 + \mathbf{s}_2) \mod q \\ &= -\mathbf{v}_q\mathbf{y}_1 + \mathbf{y}_2 = \mathbf{u} \mod q \end{aligned}$$

Similarly, the compressed version of $\mathbf{u}$ can be expressed only with $\mathbf{v}_q, \mathbf{z}_1$, and $\mathbf{z}_2$, so that the verification is possible. We explain how BLISS verification works in Algorithm 31. The first two conditions consist in verifying that $\mathbf{z}$ is distributed according to D_r^{2n}.

Compressing the signature. To reduce the size of the signature, the authors of [DDLL13] choose to drop a part of the bits of $\mathbf{z}_2$. This choice is similar to the one made in [GLP12]. Morally, the authors of [DDLL13] set a parameter d (which is the number of bits) they aim to drop. For all integer x in $[-q, q)$, according to Equation 10.12. we denote by $\lfloor x \rceil_{2^d}$ the d "high-order bits" of x such that:

$$x = \lfloor x \rceil_{2^d} \cdot 2^d + \left\lfloor x \mod 2^d \right\rceil,$$

Algorithm 31 BLISS verifying algorithm.

1: **function** VERIFY($\mu, pk = \mathbf{v}_1$)
2: **if** $\|(\mathbf{z}_1, 2^\dagger \mathbf{z}_2^\dagger)\|_2 > B_2$ **then reject**
3: **if** $\|(\mathbf{z}_1, 2^\dagger \mathbf{z}_2^\dagger)\|_\infty > B_\infty$ **then reject**
4: **if** $\mathbf{c} \neq H(\lfloor \zeta \mathbf{a}_1 \mathbf{z}_1 + \zeta q \mathbf{c} \rceil + \mathbf{z}_2^\dagger \pmod p, \mu)$ **then reject**
5: **accept**

and we extend this definition coefficient-wise to a polynomial of R_q. Then, $\mathbf{z}_2^\dagger$ is defined as:

$$\mathbf{z}_2^\dagger = \lfloor \mathbf{u} \rceil_{2^d} - \lfloor \mathbf{u} - \mathbf{z}_2 \rceil_{2^d} \mod p,$$

where $p = 2q/2^d$.

Rejection sampling. Once again, we can see that the distribution $D_{r,\mathbf{sc}}^{2n}$ of $\mathbf{z} = (\mathbf{z}_1|\mathbf{z}_2)$ depends on $\mathbf{s}$. To remove this dependency, the authors of [DDLL13] utilise rejection sampling.

In order to make sure that the distribution of $\mathbf{z}$ is independent of the secret key $\mathbf{s} = (\mathbf{s}_1|\mathbf{s}_2)$, a signature candidate $(\mathbf{z}, \mathbf{c})$ should be kept with probability:

$$\frac{D_r^{2n}(\mathbf{z})}{M D_{r,\mathbf{sc}}^{2n}(\mathbf{z})} = \frac{1}{M \exp(-||\mathbf{sc}||^2/(2\sigma^2)) \cosh(\langle \mathbf{z}, \mathbf{s}c \rangle)/\sigma^2}.$$

Computing this expression, involving transcendental functions, with sufficient precision is not an easy task. All existing implementations of BLISS [DL13, POG15, S+17] rely instead on the iterated Bernoulli trials techniques described in [DDLL13, §6] and fully detailed in [Lep14, §4.3]. We recall how these techniques work in Algorithm 32. The values $c_i = 2^i/f$ are precomputed, and the x_i's are the bits in the binary expansion of $x = \sum_{i=0}^{\ell-1} 2^i x_i$. BLISS uses $x = K - \|\mathbf{sc}\|^2$ for the input to the exponential sampler, and $x = 2\langle \mathbf{z}, \mathbf{sc} \rangle$ for the input to the cosh sampler.

Algorithm 32 Sampling algorithms for the distributions $\mathscr{B}_{\exp(-x/2\sigma^2)}$ and $\mathscr{B}_{1/\cosh(x/\sigma^2)}$.

1: **function** SAMPLEBERNEXP(x)
2: **for** $i = 0$ to $\ell - 1$ **do**
3: **if** $x_i = 1$ **then**
4: Sample $a \leftarrow \mathscr{B}_{c_i}$
5: **if** $a = 0$ **then return** 0
6: **return** 1

1: **function** SAMPLEBERNCOSH(x)
2: **if** $x < 0$ **then** $x \leftarrow -x$
3: Sample $a \leftarrow \mathscr{B}_{\exp(-x/f)}$
4: **if** $a = 1$ **then return** 1
5: Sample $b \leftarrow \mathscr{B}_{1/2}$
6: **if** $b = 1$ **then restart**
7: Sample $c \leftarrow \mathscr{B}_{\exp(-x/f)}$
8: **if** $c = 1$ **then restart**
9: **return** 0

A signature $(\mathbf{z}, \mathbf{c})$ is kept if and only if both functions SAMPLEBERNEXP($\log M - ||\mathbf{sc}||^2$) and SAMPLEBERNCOSH($2\langle \mathbf{z}, \mathbf{sc} \rangle$) return 1.

13.2 Side Channel Attack against BLISS

13.2.1 Leakage

Based on their description, it is clear that neither SAMPLEBERNEXP nor SAMPLEBERNCOSH runs in constant time. In fact, they iterate over the bits of their input and part of the code is executed when the bit is 1, and skipped when it is 0. Espitau et al. [EFGT17] observe that the inputs $x_{exp} = (\log M - ||\mathbf{sc}||_2^2$ of SAMPLEBERNEXP and $2\langle \mathbf{z}, \mathbf{sc} \rangle$ of SAMPLEBERNCOSH can then be read off directly on a trace of power consumption or electromagnetic emanations. This is very similar to the way naive square-and-multiply implementations of RSA leaks the secret exponent via simple power analysis [KJJR11]. It follows that a side-channel analysis allows to recover the squared norm $||\mathbf{sc}||^2$ and the scalar product $\langle \mathbf{z}, \mathbf{c} \rangle$.

13.2.2 Recovering the Secret From the Norm Leakage

In [EFGT17], Espitau et al. proposed an attack that exploits the norm leakage $||\mathbf{sc}||_2^2$. Their attack is based on a result due to Howgrave-Graham and Szydlo [HGS04], and describing it would require to introduce number theory notions that are beyond the topic of this thesis. We will instead try to give an idea of this attack, without getting into the details.

Let us denote by S the matrix:

$$S = \begin{pmatrix} S_1 & 0 \\ 0 & S_2 \end{pmatrix},$$

where S_1 (resp. S_2) is the skew-circulant matrix associated to the polynomial $\mathbf{s}_1$ (resp. $\mathbf{s}_2$). Then, considering $\mathbf{c}$ as a vector over $\mathbb{Z}^n$,

$$||\mathbf{sc}||_2^2 = \begin{pmatrix} {}^t\mathbf{c} & {}^t\mathbf{c} \end{pmatrix} \cdot \begin{pmatrix} {}^tS_1 & 0 \\ 0 & {}^tS_2 \end{pmatrix} \begin{pmatrix} S_1 & 0 \\ 0 & S_2 \end{pmatrix} \begin{pmatrix} \mathbf{c} \\ \mathbf{c} \end{pmatrix}$$

Denoting by X the matrix:

$$X = \begin{pmatrix} X_1 & 0 \\ 0 & X_2 \end{pmatrix} = \begin{pmatrix} {}^tS_1S_1 & 0 \\ 0 & {}^tS_2S_2 \end{pmatrix}$$

X_1 and X_2 are two skew-circulant matrices, whose first line is of the type:

$$(x_0\ x_1 \dots\ x_{n/2}\ 0\ -x_{n/2} \dots\ -x_2, -x_1),$$

as both X_1 and X_2 are the products of two conjugate skew-circulant matrices. Then, X consists of only n distinct unknowns, and thus only n linearly independent equations of the type:

$$||\mathbf{sc}||_2^2 = \begin{pmatrix} {}^t\mathbf{c} & {}^t\mathbf{c} \end{pmatrix} \cdot X \begin{pmatrix} \mathbf{c} \\ \mathbf{c} \end{pmatrix},$$

are required to fully recover X. Then, it is possible to recover the coefficients of the matrices X_1 and X_2, which appear to be the matrices associated respectively to $\mathbf{s}_1\bar{\mathbf{s}_1}$ and $\mathbf{s}_2\bar{\mathbf{s}_2}$.

The authors of [EFGT17] then utilise the Howgrave-Graham-Szydlo Algorithm, which basically allows to recover $\mathbf{s}_i \in R = \mathbb{Z}[X]/(X^n + 1)$ from the product $\mathbf{s}_i\bar{\mathbf{s}_i}$, as long as $\mathbf{s}_i$ satisfies some good properties. Indeed, at some point the Howgrave-Graham-Szydlo Algorithm requires to factorise the norm $\mathcal{N}(\mathbf{s}_i)$ of $\mathbf{s}_i$. As such, the authors of [EFGT17] restrict their attack to "weak" keys of BLISS, where this factorisation can be easily computed in the classical setting.

Remark 1. It may be interesting to notice that all instances of the BLISS post-quantum signature scheme are vulnerable to this attack, using quantum integer factorisation algorithms such as Shor's Algorithm [Sho99].

All in all, Espitau et al. show that it is possible to exploit the norm leakage in practice to recover the secret key from a little more than n signature traces ($n = 512$ in most of BLISS instances). However, their method is mathematically quite involved, and computationally costly (it actually takes over a month of CPU time for typical parameters). Furthermore, its major drawback is that it applies only to weak keys of BLISS, whose norm can easily be factorised (about 7% of BLISS secret keys).

13.2.3 Recovering the Secret Keys From the Scalar Product Leakage

We are more interested in the scalar product leakage, which is linear rather than quadratic in the secret key, and should thus be easier.

Easy case: BLISS without compression. As already noticed by the authors of [EFGT17], in a simplified version of BLISS where $(\mathbf{z} = (\mathbf{z}_1|\mathbf{z}_2), \mathbf{c})$ is returned as a signature, it is very easy to recover the secret key from about $2n$ side-channel traces. Indeed, the side-channel leakage gives access to traces:

$$\langle \mathbf{z}, \mathbf{sc} \rangle = \langle \mathbf{z}_1, \mathbf{s}_1\mathbf{c} \rangle + \langle \mathbf{z}_2, \mathbf{s}_2\mathbf{c} \rangle. \tag{13.1}$$

Denoting by $\bar{\mathbf{c}}$ the conjugate of $\mathbf{c}$, with respect to the inner product (i.e. the matrix associated to $\bar{\mathbf{c}}$ in the polynomial basis of R is the transpose of that of $\mathbf{c}$), it is possible to rewrite Equation 13.1 as follows:

$$\begin{aligned} \langle \mathbf{z}, \mathbf{sc} \rangle &= \langle \mathbf{z}_1\bar{\mathbf{c}}, \mathbf{s}_1 \rangle + \langle \mathbf{z}_2\bar{\mathbf{c}}, \mathbf{s}_2 \rangle \\ &= \langle \mathbf{a}, \mathbf{s} \rangle, \end{aligned}$$

where we let:

$$\mathbf{a} = (\mathbf{z}_1\bar{\mathbf{c}}, \mathbf{z}_2\bar{\mathbf{c}}) \in \mathbb{Z}^{2n}.$$

Now, if we collect leakage values from about $2n$ signatures, we can build the following linear system:

$$\mathbf{b} = A\mathbf{s},$$

where $b_i = \langle \mathbf{z}^{(i)}, \mathbf{sc}^{(i)} \rangle$ is known from side-channel leakage, and the i-th row $\mathbf{a}^{(i)}$ of A can be computed from the public signature $(\mathbf{z}, \mathbf{c}^{(i)})$. Then, $\mathbf{s}$ can be fully recovered, simply by solving a linear system.

Real BLISS signatures. In the actual BLISS scheme, the element $\mathbf{z}_2$ is returned in a compressed form $\mathbf{z}_2^\dagger$. The linear system arising from the scalar product leakage is then noisy. Indeed:

$$\begin{aligned} \langle \mathbf{z}, \mathbf{sc} \rangle &= \langle \mathbf{z}_1\bar{\mathbf{c}}, + \rangle \langle \mathbf{z}_2\bar{\mathbf{c}}, \mathbf{s}_2 \rangle \\ &= \langle \mathbf{z}_1\bar{\mathbf{c}}, \mathbf{s}_1 \rangle + \langle 2^d\mathbf{z}_2^\dagger\bar{\mathbf{c}}, \mathbf{s}_2 \rangle + \langle (\mathbf{z}_2 - 2^d\mathbf{z}_2^\dagger)\bar{\mathbf{c}}, \mathbf{s}_2 \rangle \\ &= \langle \mathbf{z}_1\bar{\mathbf{c}}, \mathbf{s}_1 \rangle + 2^d\langle \mathbf{z}_2^\dagger\bar{\mathbf{c}}, \mathbf{s}_2 \rangle + e \\ &= \langle \mathbf{a}, \mathbf{s} \rangle + e, \end{aligned}$$

where we let

$$\mathbf{a} = (\mathbf{z}_1\bar{\mathbf{c}} \mid 2^d\mathbf{z}_2^\dagger\bar{\mathbf{c}}) \in \mathbb{Z}^{2n} \quad \text{and} \quad e = \langle \mathbf{z}_2 - 2^d\mathbf{z}_2\dagger, \mathbf{s}_2\mathbf{c} \rangle.$$

$\mathbf{a}$ can be computed from the signature, and is therefor known to the attacker, whereas e remains unknown. Recovering $\mathbf{s}$ then amounts to solving a problem similar to LWE in dimension $2n$. This led the authors of [EFGT17] to conclude that this leakage cannot be exploited.

We claim that, because this system does not involve any modular reduction, solving it is significantly easier than solving an LWE instance. We will prove this claim in the next chapter.

Chapter 14

Of the LWE over the Integer Problem and How to Solve it

In this chapter, we present the Integer-LWE problem, we give a concrete analysis of hardness, and present ways to solve it. We detail the parameters of the instantiations arising from a BLISS side-channel, and propose experimental results. This basically consists of a paper that has been accepted in ASIACRYPT'18. This is joint work with Jonathan Bootle, Thomas Espitau, Pierre-Alain Fouque and Mehdi Tibouchi.

14.1 LWE over the Integer

14.1.1 Definition of the Problem

We define here a variant of the LWE problem "over the integers" (i.e. without modular reduction). We call this problem ILWE for "Integer-LWE".

Definition 14.1 (ILWE Distribution)**.** For any secret vector $\mathbf{s} \in \mathbb{Z}^n$ and any two probability distributions χ_a, χ_e over $\mathbb{Z}$, the ILWE distribution $\mathscr{D}_{\mathbf{s},\chi_a,\chi_e}$ associated with those parameters is a probability distribution over $\mathbb{Z}^n \times \mathbb{Z}$, where samples from $\mathscr{D}_{\mathbf{s},\chi_a,\chi_e}$ are of the form:

$$(\mathbf{a}, b) = (\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e), \text{ with } \mathbf{a} \sim \chi_a^n, \ e \sim \chi_e.$$

Remark 1. When the distributions χ_e and χ_a are clear given the context, we will simply denote the ILWE distribution $\mathscr{D}_{\mathbf{s}}$.

Problem 14.1 ((Search) $\mathsf{ILWE}_{n,m,\chi_a,\chi_e}$). We define the $\mathsf{ILWE}_{n,m,\chi_a,\chi_e}$ problem as the computational problem in which, given m samples $\{(\mathbf{a}_i, b_i)\}_{1 \leq i, \leq m}$ from a ILWE distribution $\mathscr{D}_{\mathbf{s},\chi_a,\chi_e}$ for some secret $\mathbf{s} \in \mathbb{Z}^n$, one is asked to recover the vector $\mathbf{s}$.

In terms of linear algebra, this problem consists in solving the noisy system:

$$\mathbf{b} = A\mathbf{s} + \mathbf{e}, \tag{14.1}$$

where A is m-by-n matrix, whose coefficients are drawn independently in $\mathbb{Z}$ according to the distribution χ_a, and $\mathbf{e}$ is a noise vector where each coefficient is drawn independently according to the distribution χ_e.

Remark 2. In the more general form, we impose no restriction on the distribution of the secret $\mathbf{s}$. In our tests, we however focused on somehow sparse secret $\mathbf{s}$, whose non-zero coefficients are small (typically in $\{\pm 1, \pm 2\}$), which is similar to the distribution of the secrets of $BLISS$.

14.1.2 Information Theoretic Analysis

One of the first question we can ask regarding the ILWE problem is how hard it is in an information-theoretic sense. In other words, given two vectors $\mathbf{s}, \mathbf{s}' \in \mathbb{Z}^n$, we may wonder how close the ILWE distributions $\mathscr{D}_{\mathbf{s},\chi_a,\chi_e}, \mathscr{D}_{\mathbf{s}',\chi_a,\chi_e}$ associated to $\mathbf{s}$ and $\mathbf{s}'$ are, or equivalently, how many samples we need to distinguish between them (even with unlimited computing power).

At least when the error distribution χ_e is either uniform or Gaussian, the statistical distance between $\mathscr{D}_{\mathbf{s}}$ and $\mathscr{D}_{\mathbf{s}'}$ admits a bound of the form $\mathcal{O}\left(\frac{\sigma_a}{\sigma_e}\|\mathbf{s}-\mathbf{s}'\|\right)$.
Furthermore the distributions are statistically indistinguishable when σ_e is superpolynomially larger than σ_a. In fact, we prove the following theorem:

Theorem 14.1. *Suppose that χ_e is either the uniform distribution $\mathscr{U}_\alpha$ in $[-\alpha,\alpha]\cap\mathbb{Z}$, for some positive integer r, or a centered discrete Gaussian distribution D_r, with $r \geq 1.60$. Then, for any two vectors $\mathbf{s}, \mathbf{s}' \in \mathbb{Z}^n$, the statistical distance between $\mathscr{D}_{\mathbf{s}}$ and $\mathscr{D}_{\mathbf{s}'}$ is bounded as:*

$$\Delta(\mathscr{D}_{\mathbf{s}}, \mathscr{D}_{\mathbf{s}'}) \leq C \cdot \frac{\sigma_a}{\sigma_e}\|\mathbf{s}-\mathbf{s}'\|_2,$$

where $C = 1/\sqrt{12}$ in the uniform case and $C = 1/\sqrt{2}$ in the discrete Gaussian case.

Before actually proving this theorem, we first need to prove some intermediate results.

Lemma 14.2. *The statistical distance between $\mathscr{D}_{\mathbf{s}}$ and $\mathscr{D}_{\mathbf{s}'}$ is given by:*

$$\Delta(\mathscr{D}_{\mathbf{s}}, \mathscr{D}_{\mathbf{s}'}) = \mathbb{E}[\Delta(\chi_e, \chi_e - \langle \mathbf{a}, \mathbf{s}-\mathbf{s}'\rangle)],$$

where $\chi_e + t$ denotes the translation of χ_e by the constant t, and the expectation is taken over $\mathbf{a} \leftarrow \chi_a^n$.

Proof. By definition of the statistical distance, we have:

$$\Delta(\mathscr{D}_{\mathbf{s}}, \mathscr{D}_{\mathbf{s}'}) = \frac{1}{2}\sum_{(\mathbf{a},b)\in\mathbb{Z}^{n+1}} \left|\mathscr{D}_{\mathbf{s}}(\mathbf{a},b) - \mathscr{D}_{\mathbf{s}'}(\mathbf{a},b)\right|.$$

Now to sample from $\mathscr{D}_{\mathbf{s}}$, one first samples $\mathbf{a}$ according to χ_a^n, independently samples e according to χ_e, and then returns $(\mathbf{a}, b)$ with $b = \langle \mathbf{a}, \mathbf{s}\rangle + e$. Therefore:

$$\mathscr{D}_{\mathbf{s}}(\mathbf{a},b) = \chi_a^n(\mathbf{a}) \cdot \chi_e(b - \langle \mathbf{a}, \mathbf{s}\rangle),$$

and similarly for $\mathscr{D}_{\mathbf{s}'}$. Thus, we have:

$$\begin{aligned}\Delta(\mathscr{D}_{\mathbf{s}}, \mathscr{D}_{\mathbf{s}'}) &= \frac{1}{2}\sum_{(\mathbf{a},b)\in\mathbb{Z}^{n+1}} \left|\chi_a^n(\mathbf{a}) \cdot (\chi_e(b - \langle \mathbf{a}, \mathbf{s}\rangle) - \chi_e(b - \langle \mathbf{a}, \mathbf{s}'\rangle))\right| \\ &= \left(\sum_{\mathbf{a}\in\mathbb{Z}^n} \chi_a^n(\mathbf{a})\right) \cdot \frac{1}{2}\left(\sum_{b\in\mathbb{Z}} \left|\chi_e(b - \langle \mathbf{a}, \mathbf{s}\rangle) - \chi_e(b - \langle \mathbf{a}, \mathbf{s}'\rangle)\right|\right).\end{aligned}$$

Setting $e = b - \langle \mathbf{a}, \mathbf{s}\rangle$, we have:

$$\left|\chi_e(b - \langle \mathbf{a}, \mathbf{s}\rangle) - \chi_e(b - \langle \mathbf{a}, \mathbf{s}'\rangle)\right| = \left|\chi_e(e) - \chi_e(\langle \mathbf{a}, \mathbf{s}\rangle + e - \langle \mathbf{a}, \mathbf{s}'\rangle)\right| = \left|\chi_e(e) - \chi_e(e + \langle \mathbf{a}, \mathbf{s}-\mathbf{s}'\rangle)\right|,$$

and thus,

$$\Delta(\mathscr{D}_{\mathbf{s}}, \mathscr{D}_{\mathbf{s}'}) = \left(\sum_{\mathbf{a}\in\mathbb{Z}^n} \chi_a^n(\mathbf{a})\right) \cdot \frac{1}{2}\left(\sum_{x\in\mathbb{Z}} \left|\chi_e(e) - \chi_e(e + \langle \mathbf{a}, \mathbf{s}-\mathbf{s}'\rangle)\right|\right).$$

We now observe that:

$$\frac{1}{2}\sum_{x\in\mathbb{Z}} \left|\chi_e(x) - \chi_e(x + \langle \mathbf{a}, \mathbf{s}-\mathbf{s}'\rangle)\right|$$

is exactly the statistical distance $\Delta(\chi_e, \chi_e - \langle \mathbf{a}, \mathbf{s}-\mathbf{s}'\rangle)$, and therefore we do obtain:

$$\Delta(\mathscr{D}_{\mathbf{s}}, \mathscr{D}_{\mathbf{s}'}) = \mathbb{E}[\Delta(\chi_e, \chi_e - \langle \mathbf{a}, \mathbf{s}-\mathbf{s}'\rangle)]$$

as required. □

In other words, this means that if we are able to bound the statistical distance between χ_e and a translated distribution χ_e+t, we can bound $\Delta(\mathscr{D}_{\mathbf{s}}, \mathscr{D}_{\mathbf{s}'})$. We provide such a bound over $\Delta(\chi_e, \chi_e+t)$ when χ_e is either uniform in a centred integer interval, or a discrete Gaussian distribution. First, we need to recall the following result from Micciancio and Regev [MR04].

Lemma 14.3 (From [MR04, Lemma 4.2])**.** *For any n-dimensional lattice $\mathcal{L}$, unit vector $\mathbf{u} \in \mathbb{R}^n$, and reals $0 < \varepsilon < 1$ and $r \leq 2\eta_\varepsilon(\mathcal{L})$:*

$$\left| \mathop{\mathbb{E}}_{\mathbf{x}\sim D_r}[\langle \mathbf{x}, \mathbf{u}\rangle^2 \leq \frac{r^2}{2\pi}] \right| \leq \frac{\varepsilon r^2}{1-\varepsilon}$$

Now, we can prove the second important lemma:

Lemma 14.4. *Suppose that χ_e is either the uniform distribution $\mathscr{U}_\alpha$ in $[-\alpha, \alpha]\cap\mathbb{Z}$ for some positive integer α, or the centred discrete Gaussian distribution D_r with parameter $r \geq 1.60$. In either case, let $\sigma_e = \sqrt{\mathbb{E}[\chi_e^2]}$ be the standard deviation of χ_e. We then have the following bound for all $t \in \mathbb{Z}$:*

$$\Delta(\chi_e, \chi_e + t) \leq C \cdot |t|/\sigma_e,$$

where C is the same constant as in Theorem 14.1.

Proof.

Case 1: χ_e is $\mathscr{U}[-\alpha, \alpha] \cap \mathbb{Z}$. We have:

$$\begin{aligned}
\Delta(\chi_e, \chi_e + t) &= \frac{1}{2}\sum_{x\in\mathbb{Z}} \|\chi_e(x) - \chi_e(x-t)\| \\
&= \frac{1}{2}\Big(\sum_{x\in[-\alpha,-\alpha+t)\cap\mathbb{Z}} |\chi_e(x)| + \sum_{x\in[-\alpha+t,\alpha]\cap\mathbb{Z}} |\chi_e(x) - \chi_e(x)| + \sum_{x\in(\alpha,\alpha+t]\cap\mathbb{Z}} |\chi_e(x)|\Big) \\
&= \frac{1}{2} \sum_{x\in([-\alpha,-\alpha+t)\cup(\alpha,\alpha+t])\cap\mathbb{Z}} |\chi_e(x)| \\
&= \frac{1}{2} \cdot \frac{1}{2\alpha+1} \cdot |S|,
\end{aligned}$$

where $S = ([-\alpha, -\alpha + t) \cup (\alpha, \alpha + t]) \cap \mathbb{Z}$. In particular:

$$|S| = \begin{cases} 2|t| & \text{if } t \leq 2\alpha \\ 2(2\alpha+1) & \text{otherwise} \end{cases}$$

Thus:

$$\Delta(\chi_e, \chi_e + t) = \min(1, |t|/(2\alpha+1)) \leq \frac{|t|}{2\alpha+1}.$$

Moreover, the variance of this uniform distribution is given by $\sigma_e^2 = \frac{2}{2\alpha+1}\sum_{x=1}^{\alpha} x^2 = \alpha(\alpha+1)/3$. It follows:

$$(2\alpha+1)^2 = 4\alpha(\alpha+1) + 1 = 12\sigma_e^2 + 1 \geq 12\sigma_e^2$$

Then,

$$\Delta(\chi_e, \chi_e + t) \leq \frac{|t|}{\sqrt{12}\sigma_e},$$

which concludes the proof for the case χ_e being $\mathscr{U}[-\alpha, \alpha] \cap \mathbb{Z}$.

Case 2: χ_e is a discrete Gaussian distribution D_r. Recall that:

$$\chi_e(x) = \frac{1}{\rho_r(\mathbb{Z})} \cdot \exp\Big(-\frac{\pi x^2}{r^2}\Big).$$

In order to bound the statistical distance between χ_e and $\chi_e + t$, we use first the Kullback-Leibler (KL) divergence (cf. Equation 10.4). We have:

$$D_{KL}(\chi_e \| \chi_e + t) = \sum_{x\in\mathbb{Z}} \chi_e(x) \cdot \ln \frac{\chi_e(x)}{\chi_e(x-t)}.$$

By definition of χ_e, the ln factor can be written as:

$$\begin{aligned}\ln \frac{\chi_e(x)}{\chi_e(x-t)} &= \ln \exp\Big(-\pi \frac{x^2-(x-t)^2}{r^2}\Big) \\ &= -\pi \frac{2tx - t^2}{r^2}.\end{aligned}$$

Therefore, we have:

$$\begin{aligned}D_{KL}(\chi_e \| \chi_e + t) &= \sum_{x\in\mathbb{Z}} \pi\chi_e(x) \cdot \frac{t^2 - 2tx}{r^2} \\ &= \mathbb{E}\Big[\pi \frac{t^2 - 2t\chi_e}{r^2}\Big] \\ &= \frac{\pi t^2}{r^2}\end{aligned}$$

since χ_e is centred.

From Pinsker's inequality given in Equation 10.5 we can bound the statistical distance as follows:

$$\Delta(\chi_e, \chi_e + t) \le \sqrt{\frac{1}{2} D_{KL}(\chi_e \| \chi_e + t)}.$$

It follows that:

$$\Delta(\chi_e, \chi_e + t) \le \sqrt{\frac{\pi}{2}} \cdot \frac{|t|}{r}$$

Moreover, if r satisfies $r \ge 2\eta_\varepsilon(\mathbb{Z})$ for some $\varepsilon \in (0,1)$, then by Lemma 14.3, we have:

$$\Big|\mathbb{E}[\chi_e^2] - \frac{r^2}{2\pi}\Big| \le \frac{\varepsilon r^2}{1-\varepsilon}.$$

Hence $\sigma_e = \sqrt{\mathbb{E}[\chi_e^2]} \le r\sqrt{\frac{1}{2\pi} + \frac{\varepsilon}{1-\varepsilon}}$, which gives:

$$\Delta(\chi_e, \chi_e + t) \le \sqrt{\frac{\pi}{2}\left(\frac{1}{2\pi} + \frac{\varepsilon}{1-\varepsilon}\right)} \cdot \frac{|t|}{\sigma_e} = C_\varepsilon \cdot \frac{|t|}{\sigma_e}$$

where $C_\varepsilon^2 = \frac{1}{4} + \frac{\pi}{2}\frac{\varepsilon}{1-\varepsilon}$. Taking for example $\varepsilon = 1/(1+2\pi)$, we get:

$$C_\varepsilon^2 = \frac{1}{4} + \frac{\pi}{2} \cdot \frac{1}{1+2\pi-1} = \frac{1}{4} + \frac{1}{4} = \frac{1}{2}.$$

Thus, we get

$$\Delta(\chi_e, \chi_e + t) \le C \cdot |t|/\sigma_e,$$

with $C = 1/\sqrt{2}$ as required, provided that $r \ge 2\eta_\varepsilon(\mathbb{Z})$, $\varepsilon = 1/(1+2\pi)$. We can find numerically that the smoothing parameter $\eta_\varepsilon(\mathbb{Z})$ of $\mathbb{Z}$, with respect to ε is 0.7955, which concludes the proof. □

The proof of Theorem 14.1 comes immediately by combining results from Lemma 14.2 and Lemma 14.4.

Proof of Theorem 14.1. Lemma 14.2 gives:

$$\Delta(\mathscr{D}_{\mathbf{s}}, \mathscr{D}_{\mathbf{s}'}) = \mathbb{E}[\Delta(\chi_e, \chi_e - \langle \mathbf{a}, \mathbf{s} - \mathbf{s}' \rangle)],$$

and according to Lemma 14.4, the statistical distance on the right-hand side is bounded as:

$$\Delta(\chi_e, \chi_e + \langle \mathbf{a}, \mathbf{s} - \mathbf{s}' \rangle) \leq \frac{C}{\sigma_e} \cdot |\langle \mathbf{a}, \mathbf{s} - \mathbf{s}' \rangle|.$$

It follows that:

$$\Delta(\mathscr{D}_{\mathbf{s}}, \mathscr{D}_{\mathbf{s}'}) \leq \frac{C}{\sigma_e} \cdot \mathbb{E}\Big[|\langle \mathbf{a}, \mathbf{s} - \mathbf{s}' \rangle|\Big] \leq \frac{C}{\sigma_e} \sqrt{\mathbb{E}\Big[\langle \mathbf{a}, \mathbf{s} - \mathbf{s}' \rangle^2\Big]},$$

where the second inequality is a consequence of the Cauchy-Schwarz inequality. Now, notice that for all $1, \leq i, j, \leq n$, $\mathbb{E}[a_i a_j] = \sigma_a^2 \delta_{ij}$. That is because each a_i is drawn independently from a centred distribution χ_a. It follows that, for any fixed $\mathbf{u} \in \mathbb{Z}^n$, we can write:

$$\begin{aligned}
\mathbb{E}\Big[\langle \mathbf{a}, \mathbf{u} \rangle^2\Big] &= \mathbb{E}\Big[\sum_{1 \leq i,j \leq n} u_i u_j a_i a_j\Big] \\
&= \sum_{1 \leq i,j \leq n} u_i u_j \mathbb{E}[a_i a_j] \\
&= \sigma_a^2 \|\mathbf{u}\|_2^2.
\end{aligned}$$

In particular:

$$\mathbb{E}[\langle \mathbf{a}, \mathbf{s} - \mathbf{s}' \rangle^2] = \sigma_a^2 \|\mathbf{s} - \mathbf{s}'\|_2^2,$$

and thus

$$\Delta(\mathscr{D}_{\mathbf{s}}, \mathscr{D}_{\mathbf{s}'}) \leq C \cdot \frac{\sigma_a}{\sigma_e} \|\mathbf{s} - \mathbf{s}'\|_2$$

□

This shows that distinguishing between $\mathscr{D}_{\mathbf{s}}$ and $\mathscr{D}_{\mathbf{s}'}$ requires about m samples, with:

$$m = \Omega\left(\frac{1}{\|\mathbf{s} - \mathbf{s}'\|^2} \left(\frac{\sigma_e}{\sigma_a}\right)^2\right). \tag{14.2}$$

In particular, recovering $\mathbf{s}$ (which implies distinguishing $\mathscr{D}_{\mathbf{s}}$ from all $\mathscr{D}_{\mathbf{s}'}$ with $\mathbf{s}' \neq \mathbf{s}$) requires m samples, with:

$$m = \Omega\left((\sigma_e/\sigma_a)^2\right). \tag{14.3}$$

In what follows, we will describe efficient algorithms that actually recover $\mathbf{s}$ from only slightly more samples than this lower bound.

Remark 3. Contrary to the results of the next section, which will apply to all sub-Gaussian distributions, we cannot establish an analogue of Lemma 14.4 using only a bound on the tail of the distribution χ_e. For example, if χ_e is supported over $2\mathbb{Z}$, then $\Delta(\chi_e, \chi_e + t) = 1$ for any odd t.

14.2 Solving ILWE

In this section, we present efficient algorithms to solve the ILWE problem. We are given m samples $(\mathbf{a}_i, b_i)$ from the ILWE distribution $\mathscr{D}_{\mathbf{s},\chi_a,\chi_e}$, and we aim to recover $\mathbf{s} \in \mathbb{Z}^n$. Since $\mathbf{s}$ can a priori be any vector, we, of course, need at least n samples to recover it. We are thus interested in the case when $m \geq n$. We recall here the system given in Equation 14.1:

$$\mathbf{b} = A\mathbf{s} + \mathbf{e},$$

where each coefficient of the m-by-n matrix A is drawn independently from χ_a, and each coefficient of $\mathbf{e}$ is drawn independently from χ_e. The couple $(A, \mathbf{b})$ is known, whereas $\mathbf{s}$ and $\mathbf{e}$ remain unknown.

To recover $\mathbf{s}$ we propose to find first an approximate solution $\tilde{\mathbf{s}} \in \mathbb{R}^n$ of the noisy linear system (14.1) and to simply round that solution coefficient by coefficient to get a candidate $\lfloor \tilde{\mathbf{s}} \rceil = (\lfloor \tilde{s}_1 \rceil, \ldots, \lfloor \tilde{s}_n \rceil)$ for $\mathbf{s}$. If we can establish the bound:

$$\|\mathbf{s} - \tilde{\mathbf{s}}\|_\infty < 1/2 \tag{14.4}$$

or, a fortiori, the stronger bound $\|\mathbf{s} - \tilde{\mathbf{s}}\|_2 < 1/2$, then it follows that $\lfloor \tilde{\mathbf{s}} \rceil = \mathbf{s}$ and the ILWE problem is solved.

The main technique we propose to use is least squares regression. Under the mild assumption that both χ_a and χ_e are sub-Gaussian distributions, we will show that the corresponding $\tilde{\mathbf{s}}$ satisfies the bound (14.4) with high probability when m is sufficiently large. Moreover, the number m of samples necessary to establish those bounds, and hence solve ILWE, is only a $\ln n$ factor larger than the information-theoretic minimum given in (14.3) (with the additional constraint that m should be a constant factor larger than n, to ensure that A is invertible and has well-controlled singular values).

We also briefly discuss lattice reduction as well as linear programming (LP) methods. We show that even an exact-CVP oracle cannot significantly improve upon the $\ln n$ factor of the least squares method. On the other hand, if the secret is known to be very sparse, there exist LP techniques which can recover the secret even in cases when $m < n$, where the least squares method is not applicable.

14.2.1 The least square Method

The first approach we consider to obtain an estimator $\tilde{\mathbf{s}}$ of $\mathbf{s}$ is the linear, unconstrained least squares method.

Description of the method. For all $1 \leq i \leq m$, we have:

$$b_i - \langle \mathbf{a}_i, \mathbf{s} \rangle = e_i.$$

$\tilde{\mathbf{s}}$ is chosen as a vector in $\mathbb{R}^n$ minimising the sum:

$$g(\mathbf{x}) = \sum_i^m e_i^2 = \sum_i^m (b_i - \langle \mathbf{a}_i, \mathbf{x} \rangle^2).$$

This minimum is found by setting the gradient of g to zero. Recalling that the gradient of g is given by the vector:

$$\nabla g(\mathbf{x}) = \begin{pmatrix} \frac{\partial g(\mathbf{x})}{\partial x_1} \\ \vdots \\ \frac{\partial g(\mathbf{x})}{\partial x_n} \end{pmatrix},$$

we come up with a system of n equations of the form:

$$-2 \sum_{i=1}^m e_i a_{ij} \tilde{s}_i = 0, \quad 1 \leq j \leq n.$$

It follows that $\tilde{\mathbf{s}}$ is simply a solution to the linear system:

$${}^tA A \tilde{\mathbf{s}} = {}^tA \mathbf{b}.$$

As a result, we can compute $\tilde{\mathbf{s}}$ in polynomial time (at most $O(mn^2)$) and it is uniquely defined if and only if ${}^tAA$ is invertible.

In the rest of the section, we will show that taking $m = \Omega\left(\max(n, (\frac{\sigma_e}{\sigma_a})^2 \ln n)\right)$, should lead to an invertible matrix ${}^tAA$. Furthermore, this bound appears to be quite tight in practice, as discussed in Section 14.4.

Analysing the least square Method

It is intuitively clear that ${}^tAA$ should be invertible when m is large. Indeed:

$$ {}^tAA = \sum_{i=1}^{m} \mathbf{a}_i {}^t\mathbf{a}_i $$

where the $\mathbf{a}_i$ vectors are the independent identically distributed rows of A, so by the law of large numbers, $\frac{1}{m}{}^tAA$ converges almost surely to $\mathbb{E}[\mathbf{a}{}^t\mathbf{a}]$ as m grows to infinity, where $\mathbf{a}$ is a random variable in $\mathbb{Z}^n$ sampled from χ_a^n. Furthermore, we have, for all $1 \leq i, j, \leq n$:

$$ \mathbb{E}\big[(\mathbf{a}{}^t\mathbf{a})_{ij}\big] = \mathbb{E}[a_i a_j] = \delta_{ij}\sigma_a^2, $$

and therefore we expect ${}^tAA$ to be close to $m\sigma_a^2 I_n$ for large m.

Intermediate results. Making the heuristic argument above rigorous is not an easy task. Assuming some tail bounds on the distribution χ_a, it is possible to prove that, with high probability, the smallest eigenvalue $\lambda_{\min}({}^tAA)$ is not much smaller than $m\sigma_a^2$ (and in particular ${}^tAA$ is invertible) for m sufficiently large, with a concrete bound on m. This type of bound on the smallest eigenvalue is exactly what we need for the rest of our analysis.

When χ_a is bounded, it is possible to apply a form of the so-called Matrix Chernoff inequality, such as [Tro12, Cor. 5.2]. However, we would prefer a result that applies to discrete Gaussian distributions as well, so we only assume a sub-Gaussian tail bound for χ_a. Such result can be derived from the following lemma adapted from Hsu et al. [HKZ12].

Lemma 14.5. *Let χ be a τ-sub-Gaussian distribution of variance 1 over $\mathbb{R}$, and consider m random vectors $\mathbf{x}_1, \ldots, \mathbf{x}_m$ in $\mathbb{R}^n$ sampled independently according to χ^m. For any $\delta \in (0,1)$, we have:*

$$ \Pr\left[\lambda_{\min}\left(\frac{1}{m}\sum_{i=1}^{m}\mathbf{x}_i{}^t\mathbf{x}_i\right) < 1 - \varepsilon(\delta, m)\right] < \delta $$

and

$$ \Pr\left[\lambda_{\max}\left(\frac{1}{m}\sum_{i=1}^{m}\mathbf{x}_i{}^t\mathbf{x}_i\right) > 1 + \varepsilon(\delta, m)\right] < \delta $$

where the error bound $\varepsilon(\delta, m)$ is given by:

$$ \varepsilon(\delta, m) = 4\tau^2\left(\sqrt{\frac{8\ln 9 \cdot n + 8\ln(2/\delta)}{m}} + \frac{\ln 9 \cdot n + \ln(2/\delta)}{m}\right). $$

Using this lemma, it is possible to show that if χ_a is a sub-Gaussian distribution, $\lambda_{\min}({}^tAA)$ is close to $m\sigma_a^2$, with probability $1 - 2^{-\eta}$ for $m = \Omega(n + \eta)$ (and similarly for $\lambda_{\max}$). Indeed, we can prove the following theorem:

Theorem 14.6. *Suppose that χ_a is τ_a-sub-Gaussian, and let $\tau = \tau_a/\sigma_a$. Let A be an $m \times n$ random matrix sampled from $\chi_a^{m\times n}$. There exist constants C_1, C_2 such that for all $\alpha \in (0,1)$ and $\eta \geq 1$, if $m \geq (C_1 n + C_2\eta)\cdot(\tau^4/\alpha^2)$ then*

$$ \Pr\left[\lambda_{\min}({}^tAA) < (1-\alpha)\cdot m\sigma_a^2\right] < 2^{-\eta}. \tag{14.5} $$

and similarly,

$$ \Pr\left[\lambda_{\max}({}^tAA) > (1+\alpha)\cdot m\sigma_a^2\right] < 2^{-\eta}. $$

Furthermore, one can choose $C_1 = 2^8 \ln 9$ and $C_2 = 2^9 \ln 2$.

Proof. Let $\mathbf{a}_i$ be the i-th row of A, and $\mathbf{x}_i = \frac{1}{\sigma_a}\mathbf{a}_i$. Then the coefficients of $\mathbf{x}_i$ follow a τ-sub-Gaussian distribution of variance 1, and every coefficient of each one of the $\mathbf{x}_i$ is independent from all the others, and so the $\mathbf{x}_i$ vectors satisfy the hypotheses of Lemma 14.5. It follows:

$$\frac{1}{m}\sum_{i=1}^{m}\mathbf{x}_i{}^t\mathbf{x}_i = \frac{1}{m\sigma_a^2}\sum_{i=1}^{m}\mathbf{a}_i{}^t\mathbf{a}_i = \frac{1}{m\sigma_a^2}{}^tAA.$$

Therefore, Lemma 14.5 shows that:

$$\mathbb{P}\Big[\lambda_{\min}({}^tAA) < (1-\varepsilon(2^{-\eta},m))\cdot m\sigma_a^2\Big] < 2^{-\eta}$$

and

$$\mathbb{P}\Big[\lambda_{\max}({}^tAA) > (1+\varepsilon(2^{-\eta},m))\cdot m\sigma_a^2\Big] < 2^{-\eta}$$

with $\varepsilon(\delta,m)$ defined as above. Thus, to obtain Equation 14.5, it is enough to take m such that $\varepsilon(2^{-\eta},m)\leq\alpha$.

The value $\varepsilon(\delta,m)$ can be re-written as $4\tau^2\cdot(\sqrt{8\rho}+\rho)$ where $\rho = \big(\ln 9\cdot n+\ln(2/\delta)\big)/m$. For the choice of m in the statement of the theorem, we necessarily have $\rho<1$ since $\sigma_a\leq\tau_a$, and hence $\tau^4\geq 1$. It follows that, $\varepsilon(\delta,m)\leq 16\tau^2\cdot\sqrt{\rho}$. Thus, to obtain the announced result, it is enough to choose:

$$m \geq \frac{2^8\tau^4}{\alpha^2}\Big(\ln 9\cdot n + \ln 2^{1+\eta}\Big),$$

which concludes the proof. □

From now on, let us suppose that the assumptions of Theorem 14.6 are satisfied for some $\alpha\in(0,1)$ and η. In particular, ${}^tAA$ is invertible with overwhelming probability, and we can thus write:

$$\tilde{\mathbf{s}} = ({}^tAA)^{-1}\cdot{}^tA\mathbf{b}.$$

As discussed in the beginning of this section, we would like to bound the distance between the estimator $\tilde{\mathbf{s}}$ and the actual solution $\mathbf{s}$ of the ILWE problem in the infinity norm, so as to obtain an inequality of the form of Equation 14.4. Since $\mathbf{b} = A\mathbf{s}+\mathbf{e}$, we have:

$$\begin{aligned}\tilde{\mathbf{s}}-\mathbf{s} &= ({}^tAA)^{-1}{}^tA\mathbf{b}-\mathbf{s}\\ &= ({}^tAA)^{-1}{}^tA(A\mathbf{s}+\mathbf{e})-\mathbf{s}\\ &= \mathbf{s}+({}^tAA)^{-1}{}^tA\mathbf{e}-\mathbf{s}\\ &= ({}^tAA)^{-1}{}^tA\mathbf{e}\end{aligned}$$

Let us denote M the matrix $({}^tAA)^{-1}{}^tA$. We assume that all the coefficients of $\mathbf{e}$ are τ_e-sub-Gaussian. Since they are also independent, following the definition of a sub-Gaussian vector, the vector $\mathbf{e}$ is a τ_e-sub-Gaussian random vector in the sense of Definition 10.2. Therefore, it follows from Lemma 10.7 that $\tilde{\mathbf{s}}-\mathbf{s} = M\mathbf{e}$ is $\tilde{\tau}$-sub-Gaussian, where:

$$\begin{aligned}\tilde{\tau} &= \|{}^tM\|_2^{\mathrm{op}}\cdot\tau_e\\ &= \tau_e\sqrt{\lambda_{\max}(M{}^tM)}\\ &= \tau_e\sqrt{\lambda_{\max}\big(({}^tAA)^{-1}{}^tA\cdot A({}^tAA)^{-1}\big)}\\ &= \tau_e\sqrt{\lambda_{\max}\big(({}^tAA)^{-1}\big)}\\ &= \frac{\tau_e}{\sqrt{\lambda_{\min}({}^tAA)}}.\end{aligned}$$

As a result, under the hypotheses of Theorem 14.6, $\tilde{\mathbf{s}}-\mathbf{s}$ is a $\frac{\tau_e}{\sigma_a\sqrt{(1-\alpha)m}}$-sub-Gaussian random vector, except with probability at most $2^{-\eta}$.

We can utilise Lemma 10.6, to bound $\|\tilde{\mathbf{s}}-\mathbf{s}\|_\infty$. This is all we need to establish a sufficient condition for the least squares approach to return the correct solution to the ILWE problem with good probability.

Determining the number of samples. We can now give a lower bound on the number of samples required. We prove the following theorem, for every sub-Gaussian distribution.

Theorem 14.7. *Suppose that χ_a is τ_a-sub-Gaussian and χ_e is τ_e-sub-Gaussian, and let $(A, \mathbf{b} = A\mathbf{s} + \mathbf{e})$ the data constructed from m samples of the ILWE distribution $\mathscr{D}_{\mathbf{s},\chi_a,\chi_e}$, for some $\mathbf{s} \in \mathbb{Z}^n$. There exist constants $C_1, C_2 > 0$ (the same as in the hypotheses of Theorem 14.6) such that for all $\eta \geq 1$, if:*

$$m \geq 4\frac{\tau_a^4}{\sigma_a^4}(C_1 n + C_2\eta) \quad \text{and} \quad m \geq 32\frac{\tau_e^2}{\sigma_a^2}\ln(2n)$$

then the least squares estimator $\tilde{\mathbf{s}} = ({}^tAA)^{-1}{}^tA\mathbf{b}$ satisfies $\|\mathbf{s} - \tilde{\mathbf{s}}\|_\infty < 1/2$, and hence $\lfloor\tilde{\mathbf{s}}\rceil = \mathbf{s}$, with probability at least $1 - \frac{1}{2n} - 2^{-\eta}$.

In the typical case when τ_a and τ_e are no more than a constant factor larger than σ_a and σ_e, Theorem 14.7 with $\eta = \ln(2n)$ says that there are constants C, C' such that whenever

$$m \geq Cn \qquad \text{and} \qquad m \geq C' \cdot \frac{\sigma_e^2}{\sigma_a^2}\ln n \tag{14.6}$$

one can solve the ILWE problem with m samples with probability at least $1 - 1/n$ by rounding the least squares estimator. The first condition ensures that ${}^tAA$ is invertible and controls its eigenvalues; a condition of that form is clearly unavoidable to have a well-defined least squares estimator. On the other hand, the second condition gives a lower bound of the form (14.3) on the required number of samples; this bound is only a factor $\ln n$ worse than the information-theoretic lower bound, which is quite satisfactory.

Proof of Theorem 14.7. Applying Theorem 14.6 with $\alpha = 1/2$ and the same constants C_1, C_2 as introduced in the statement of that theorem, we obtain that for $m \geq \frac{\tau_a^4}{\sigma_a^4}(4C_1 n + 4C_2\eta)$, we have

$$\mathbb{P}\Big[\lambda_{\min}(A^TA) < m\sigma_a^2/2\Big] < 2^{-\eta}. \tag{14.7}$$

Therefore, except with probability at most $2^{-\eta}$, we have $\lambda_{\min}(A^TA) \geq m\sigma_a^2/2$. We now assume that this condition is satisfied.

From above $\tilde{\mathbf{s}} - \mathbf{s}$ is a $\tilde{\tau}$-sub-Gaussian random vector with $\tilde{\tau} = \tau_e/\sqrt{\lambda_{\min}(A^TA)}$. Applying Lemma 10.6 with $t = 1/2$, we therefore have:

$$\begin{aligned}
\mathbb{P}[\|\tilde{\mathbf{s}} - \mathbf{s}\|_\infty > \frac{1}{2}] &\leq 2n \cdot \exp\Big(-\frac{1}{8\tilde{\tau}^2}\Big) \\
&\leq 2n \cdot \exp\Big(-\frac{\lambda_{\min}(A^TA)}{8\tau_e^2}\Big) \\
&\leq \exp\Big(\ln(2n) - \frac{m\sigma_a^2}{16\tau_e^2}\Big).
\end{aligned}$$

Thus, if we assume that $m \geq 32\frac{\tau_e^2}{\sigma_a^2}\ln(2n)$, it follows that:

$$\mathbb{P}[\|\tilde{\mathbf{s}} - \mathbf{s}\|_\infty > \frac{1}{2}] \leq \exp\big(\ln(2n) - 2\ln(2n)\big) = \frac{1}{2n}.$$

This concludes the proof. □

14.2.2 Using an Exact-CVP Oracle

Solving the ILWE problem by computing a least square estimator and simply rounding it can be seen as an application of Babai's rounding algorithm for CVP. Indeed, consider the following lattice:

$$\mathcal{L} := \{{}^tAA\mathbf{x} \mid \mathbf{x} \in \mathbb{Z}^n\}.$$

$\mathcal{L}$ is full rank when ${}^tAA$ is invertible (i.e. when m is large enough). The ILWE problem can be solved by recovering the vector $\mathbf{v} = {}^tAA\mathbf{s} \in \mathcal{L}$ given a close vector ${}^tA\mathbf{b}$, and then returning $({}^tAA)^{-1}\mathbf{v}$.

Recovering $\mathbf{v}$ from ${}^tA\mathbf{b} = \mathbf{v} + {}^tA\mathbf{e}$ is essentially an instance of the BDD problem in $\mathcal{L}$. Now, since for large m, the matrix ${}^tAA$ is almost scalar, and hence the corresponding lattice basis of $\mathcal{L}$ is somehow already reduced, one can try to solve this problem by applying a CVP algorithm like Babai rounding directly on this basis. This is indeed exactly what we did in the least square approach.

One could ask whether applying another CVP algorithm (e.g. Babai's *nearest plane* algorithm) could allow us to solve the problem with asymptotically fewer samples: for instance, get rid of the $\ln n$ factor in Equation 14.6. The answer is no. In fact, we have this much stronger result:

Theorem 14.8. *One cannot improve Condition 14.6 using the strategy above, even given access to an* exact-*CVP oracle for any p-norm, $p \in [2, \infty]$.*

Indeed, given such an oracle, the secret vector $\mathbf{v}$ can be recovered uniquely if and only if the vector of noise ${}^tA\mathbf{e}$ lies in a ball centred on $\mathbf{v}$ and of radius half the first minimum of $\mathcal{L}$ in the p-norm, $\lambda_1^{(p)}(\mathcal{L}) = \min_{\mathbf{x}\in\mathcal{L}\setminus\{0\}} \|\mathbf{x}\|_p$, that is:

$$\|A^T\mathbf{e}\|_p \leq \frac{\lambda_1^{(p)}(\mathcal{L})}{2}. \tag{14.8}$$

To take advantage of this condition, we need to get sufficiently precise estimates of both sides.

Estimation of the First Minimum.

As the matrix ${}^tAA$ is almost diagonal, one can estimate accurately the value of $\lambda_1^{(p)}(\mathcal{L})$. Indeed, ${}^tAA$ has a low orthogonality defect, so that it is in a sense already reduced. Hence, the shortest vector of this basis constitutes a very good approximation of the shortest vector of $\mathcal{L}$. We have in fact, the following lemma:

Lemma 14.9. *Suppose that χ_a is τ_a-sub-Gaussian, and let $\tau = \tau_a/\sigma_a$. Let A be an m-by-n random matrix sampled from $\chi_a^{m\times n}$. Let $\mathcal{L}$ be the lattice generated by the rows of the matrix ${}^tAA$. There exist constants C_1, C_2 (the same as in Theorem 14.6) such that for all $\alpha \in (0,1)$, $p \geq 2$ and $\eta \geq 1$, if $m \geq (C_1 n + C_2\eta)\cdot(\tau^4/\alpha^2)$ then*

$$\mathbb{P}\left[\lambda_1^{(p)}(\mathcal{L})({}^tAA) > m\sigma_a^2(1+\alpha)\right] \leq 2^{-\eta}. \tag{14.9}$$

Proof. By norm equivalence in finite dimension, for all $\mathbf{x} \in \mathbb{R}^n$ we have $\|\mathbf{x}\|_p \leq \|\mathbf{x}\|_2$ for all $p \in [2, +\infty]$. In particular, this implies that: $\lambda_1^{(p)}(\mathcal{L}) \leq \lambda_1^{(2)}(\mathcal{L})$. This bound is actually sharp. Without loss of generality it is then enough to prove the result in 2-norm.

We first notice that $\lambda_{max}({}^tAA) = \|{}^tAA\|_2^{\text{op}}$. Indeed, we have:

$$\|{}^tAA\|_2^{\text{op}} = \sqrt{\lambda_{max}({}^t({}^tAA){}^tAA)} = \sqrt{\lambda_{max}(({}^tAA)^2)} = \lambda_{max}({}^tAA)$$

Now, from Theorem 14.6, we can assert that except with probability at most $2^{-\eta}$, $\|{}^tAA\|_2^{\text{op}} \leq m\sigma_a^2(1+\alpha)$. Therefore, for any vector $\mathbf{x} \in \mathbb{Z}^n$, we have by definition of the operator norm:

$$\|{}^tAA\mathbf{x}\|_2 \leq m\sigma_a^2\|\mathbf{x}\|_2(1+\alpha),$$

with high probability. In particular, for any $\mathbf{x} \in \mathbb{Z}^n$ such that $\|\mathbf{x}\|_2 = 1$, we have

$$\lambda_1^{(2)}(\mathcal{L}) \leq \|{}^tAA\mathbf{x}\|_2 \leq (1+\alpha)m\sigma_a^2,$$

except with probability at most $2^{-\eta}$. □

Estimation of the p-Norm of ${}^tA\mathbf{e}$

Suppose that χ_e is a centred Gaussian distribution of standard deviation σ_e. The distribution of ${}^tA\mathbf{e}$ for $\mathbf{e} \sim \chi_e^n$ is then a Gaussian distribution of covariance matrix $\sigma_e^2{}^tAA \approx m\sigma_a^2\sigma_e^2 I_n$.

We have the following results:

Lemma 14.10. *For all $p \in [2, \infty)$,*

$$n^{1/p}\sigma_e\sigma_a\sqrt{\frac{m}{2\pi}} \leq \mathbb{E}[\|{}^tA\mathbf{e}\|_p], \tag{14.10}$$

and

$$C_\infty\sigma_e\sigma_a\sqrt{m\ln n} \leq \mathbb{E}[\|{}^tA\mathbf{e}\|_\infty], \tag{14.11}$$

with $C_\infty = \frac{3}{2}(1-1/e) - 1/\sqrt{2\pi} \simeq 0.23$.

Proving these results would require to introduce too many probability notions. As such we will simply admit them here. We simply state that the case $p < \infty$ is a consequence of [Sta84]. We also refer the interested reader to [vH14] for the case $p = \infty$.

Lemma 14.11. *There exist absolute constants $K, c > 0$, such that:*

$$\frac{1}{2}\mathbb{E}\Big[\|{}^tA\mathbf{e}\|_p\Big] \leq \|{}^tA\mathbf{e}\|_p \leq \frac{3}{2}\mathbb{E}\Big[\|{}^tA\mathbf{e}\|_p\Big], \tag{14.12}$$

except with probability $Ke^{-c\beta(n,p,1/2)}$, with:

$$\beta(m,p,\varepsilon) = \begin{cases} \varepsilon^2 n & \text{if } 1 < p \leq 2 \\ \max(\min(2^{-p}\varepsilon^2 n, (\varepsilon n)^{2/p}), \varepsilon p n^{2/p}) & \text{if } 2 < p \leq c_0 \ln n \\ \varepsilon \ln n & \text{if } p > c_0 \ln n \end{cases},$$

for any $0 < \varepsilon < 1$ and for $0 < c_0 < 1$ a fixed constant.

This is a consequence of a theorem from Dvoretzky [Dvo61] whose proof can be found in [PVZ17]. We refer the interested reader to [Dvo61, PVZ17] for more detail.

Summing up

For any fixed p, the probability that Equation 14.12 is false can be made as small as desired for large enough n. We can therefore assume that Equation 14.12 occurs with probability at least $1-\delta$ for some small $\delta > 0$.

In this case, Condition 14.8 asserts that if $\mathbb{E}\Big[\|{}^tA\mathbf{e}\|_p\Big] > \lambda_1^{(p)}(\mathcal{L})$ then $\mathbf{s}$ cannot be decoded uniquely in $\mathcal{L}$. Now using the result of Lemma 14.9 with $\alpha = 1/2$ and the previous estimates, we know that this is the case when:

$$n^{1/p}\sigma_e\sigma_a\sqrt{\frac{m}{2\pi}} > \frac{3}{2}m\sigma_a^2, \qquad \text{that is, } m < \left(\frac{\sigma_e}{\sigma_a}\right)^2 \frac{2n^{2/p}}{9\pi},$$

when $p < \infty$, and

$$0.23\sigma_e\sigma_a\sqrt{m\ln n} > \frac{3}{2}m\sigma_a^2, \qquad \text{that is, } m < 0.02\left(\frac{\sigma_e}{\sigma_a}\right)^2 \ln n,$$

otherwise. In both cases, it follows that we must have $m = \Omega\big((\sigma_e/\sigma_a)^2 \ln n\big)$ for the CVP algorithm to output the correct secret with probability greater than δ. Thus, this approach cannot improve upon the least squares bound 14.7 by more than a constant factor.

14.2.3 Linear Programming Approach

LP and MILP modelisation.

Linear Programming (LP) is a method for the optimisation of a linear objective function subject to linear inequality constraints. The variables considered in this model are real variables, represented using floating point arithmetic. *Integer Linear Programming* (ILP) is a variant of LP where the variables are integers. Finally in *Mixed Integer Linear Programming* (MILP) both real and integer values can appear in the same model. The state of the art in MILP solver is Gurobi [GO18], which is known to be one of the best software to solve problems that have been, beforehand, transformed into a MILP model.

Algorithms and complexity. LP problems can be solved in polynomial time in the number of equations. ILP problems and MILP problems, on the other hand, are NP-hard.

The most famous algorithm for LP problem is the simplex method, which is known to solve random problems efficiently (in a cubic number of steps), but whose worst case time complexity is exponential in the number of variables of the LP system [KM70]. The most efficient worst-case polynomial time (in number of variables of the system) algorithm is Kamarkar's algorithm, which is known to solve LP problems in $\mathcal{O}\left(n^{3.5}L\right)$ operations, where n is the number of variables of the system, and can be encoded using L bits. We are not going to recall how these two algorithms work here, as this is not the point of this section. We refer the interested reader to [Dan51] for details about the simplex method, and to [Kar84], for Kamarkar's algorithm.

To solve ILP and MILP problems, the usual method is to relax first the problem in a LP setting, where an approximate solution can be found efficiently using Kamarkar's algorithm for instance. Once this solution is found, if there is at least one coefficient which is not an integer, new constraints are added to the program, that are satisfied by all integer coefficients of the approximate solution, but violated by the non-integer ones. The process is repeated until an integer solution is found. The usual way to proceed is the following. The initial problem is split into two sub-problems. In the first one, a constraint is added to ensure that the non-integer coefficient is greater than or equal to the ceiling of the approximate value. In the second one, a constraint is added to ensure that the non-integer coefficient is lower than or equal to the floor of the approximate value. For more detail, we refer the reader to [Gom58, BCCN96, Mit98].

LWE and ILP. The LWE problem seems to be quite easy to rewrite in the (M)ILP setting. Indeed, given an $\mathsf{LWE}_{n,m,q,\chi_e}$ system:

$$\mathbf{b} = A\mathbf{s} + \mathbf{e} \mod q,$$

we can consider the following ILP program:

$$\begin{aligned} &\text{Find } \mathbf{s} \text{ subject to:} \\ &\begin{pmatrix} A & qI_n \end{pmatrix} \begin{pmatrix} \mathbf{s} \\ \mathbf{u} \end{pmatrix} \leq \mathbf{b} + B \\ &\begin{pmatrix} A & qI_n \end{pmatrix} \begin{pmatrix} \mathbf{s} \\ \mathbf{u} \end{pmatrix} \geq \mathbf{b} - B \end{aligned} \tag{14.13}$$

where B is a bound over the elements drawn from χ_e. Other constraints can be added, if we know that the secret is draw from a B'-bounded distribution, for instance.

Along these lines, Herold and May [HM17] find a way to attack a variant of an LWE-based crypto-system, due to Galbraith [Gal13] where the matrix A consists only of 0 and 1 coefficients. Using ILP, they were able to recover messages $\mathbf{u}$ from the cipher-text $\mathbf{c}$, where ${}^t\mathbf{u}A = {}^t\mathbf{c} \pmod q$, for the set of parameters given in [Gal13]. Their attack did not allow them to recover the secret $\mathbf{s}$ of the LWE system, however.

As in the ILWE setting, there is no modular reduction, we are tempted to use the following ILP program to solve our problem:

$$\begin{aligned} &\text{Find } \mathbf{s} \text{ subject to:} \\ &A\mathbf{s} \leq \mathbf{b} + B \\ &A\mathbf{s} \geq \mathbf{b} - B. \end{aligned} \tag{14.14}$$

Furthermore, as in BLISS case, we know exactly the L_1 norm N_1 of the secret, we can add the following constraints:

$$\begin{aligned} -u_i \leq s_i \leq u_i \\ \sum u_i = N_1 \end{aligned}$$

Remark 1. It is also possible to model exactly the absolute value a of a variable x using MILP constraints, using the following linear program:

$$\begin{aligned} &\text{Minimise } a: \\ &x \leq a \\ &x \geq -a \end{aligned}$$

This, however, proves rather disappointing. Indeed, MILP solvers tend to be very slow when the integers they have to handle are large. Unfortunately for us, the range of the coefficients of A seems to be a bit too large.

We then decided to utilise an LP relaxation of our model (with floating numbers), and then see if we can recover the solution with a simple rounding. Once again, all of our attempts have proved to be disappointing. We were never able to beat the number of samples required by the least squares method.

However, as discussed below, there is still a case where the LP methods can beat the least square method: when the secret is very sparse.

Sparse Secret and Compressed Sensing

Up until this point, we have only considered the case where the number m of samples is greater than the dimension n. Indeed, without additional information on the secret $\mathbf{s}$, this condition is necessary to get a well-defined solution to the ILWE problem even without noise.

Now suppose that the secret $\mathbf{s}$ is known to be sparse, more precisely that its number of non-zero coefficient is at most S, with $S \ll n$ known. We say that $\mathbf{s}$ is S-sparse. Even if the positions of these non-zero coefficients are not known, knowledge of the sparsity S may help in determining the secret, possibly even with fewer samples than the ambient dimension n with the sole additional knowledge of its sparsity. Of course more than S samples are, however, necessary.

Such a recovery is possible using *compressed sensing* techniques, following the results of Candès and Tao in [CT06, CT07].

Compressed sensing. Compressed sensing is a method due to Candès and Tao [CT06] to recover a sparse solution to an underdetermined linear system $A\mathbf{s} = \mathbf{b}$. In [CT07], the same authors show that this method can be extended to solve noisy system. The idea of compressed sensing can be summarised as follows: in many cases, finding the solution $\tilde{\mathbf{s}}$ to the system $A\mathbf{s} = \mathbf{b}$ with minimum Hamming weight amounts to finding the solution $\tilde{\mathbf{s}}$ to $A\mathbf{s} = \mathbf{b}$, such that $||\tilde{\mathbf{s}}||_1$ is minimised [CT06]. The latter can easily be re-written in a LP setting, and then solved quite efficiently with LP techniques.

Noisy systems and Dantzig selector. When the considered system is noisy, it is possible to find a sparse solution, using what is called the *Dantzig selector* [CT07]. The goal is to recover the solution $\tilde{\mathbf{s}}$ to a system $A\mathbf{s}+\mathbf{e} = \mathbf{b}$, that minimise $\|\tilde{\mathbf{s}}\|_1$ under the constraint $\|{}^tA\mathbf{e}\|_\infty \leq \sigma_e\sigma_e\sqrt{2\ln n}\cdot N$, where σ_e is the standard deviation of the noise, and N is an upper bound on the norm of the columns of A. This solution can be easily recovered with the following LP model:

$$\min \sum_{i=1}^{n} u_i \qquad \text{such that} \qquad -u_i \leq \tilde{s}_i \leq u_i \qquad \text{and}$$
$$-\sigma_e\sqrt{2\ln n}\cdot N \leq [{}^tA(\mathbf{b} - A\tilde{\mathbf{s}})]_i \leq \sigma_e\sqrt{2\ln n}\cdot N.$$

Remark 2. The estimator $\tilde{\mathbf{s}}$ is what is called the Dantzig selector. This name was chosen by Candès and Tao as a tribute to George Dantzig, the inventor of the simplex method, who died in 2005, while the two authors of [CT07] were working on their paper.

ILWE with sparse secret. When facing an ILWE instance $A\mathbf{s} + \mathbf{e} = \mathbf{b}$, such that $\mathbf{s}$ is known to be S-sparse, we can use the aforementioned technique to recover the secret. The idea is once again to find an approximate solution $\tilde{\mathbf{s}}$, such that $\|\tilde{\mathbf{s}} - \mathbf{s}\|_\infty$ is small enough to fully recover $\mathbf{s}$.

Table 14.1 – Maximum value of the ratio σ_e/σ_a to recover a S sparse secret in dimension n with the Dantzig selector .

n \ (S/n)	0.1	0.3	0.5	0.7	0.9
128	16.2	9.4	7.3	6.1	5.4
256	15.2	8.8	6.8	5.7	5.0
512	14.3	8.3	6.4	5.4	4.8
1024	13.6	7.8	6.0	5.1	4.5
2048	13.0	7.5	5.8	4.9	4.3

We search for a solution $\tilde{\mathbf{s}} = (\tilde{s}_1, \ldots, \tilde{s}_n)$ of the following linear program with $2n$ unknowns $\tilde{s}_i, \tilde{u}_i$, $1 \leq i \leq n$:

$$\begin{aligned} &\min \sum_{i=1}^{n} u_i \qquad \text{such that} \qquad -u_i \leq \tilde{s}_i \leq u_i \qquad \text{and} \\ &-\sigma_e\sigma_a\sqrt{2m\ln n} \leq \left[{}^tA(\mathbf{b} - A\tilde{\mathbf{s}})\right]_i \leq \sigma_e\sigma_a\sqrt{2m\ln n}. \end{aligned} \tag{14.15}$$

If the distributions χ_e and χ_a are Gaussian distributions of respective standard deviations σ_e and σ_a, the quality of the output of the program defined by Equation 14.16 is quantified as follows:

Theorem 14.12 (adapted from [CT07])**.** *Suppose* $\mathbf{s} \in \mathbb{Z}^n$ *is any* S*-sparse vector so that* $\ln(m\sigma_a^2/n)S \leq m$*. Then in many cases,* $\tilde{\mathbf{s}}$ *obeys the relation*

$$\|\tilde{\mathbf{s}} - \mathbf{s}\|_2^2 \leq 2C_1^2 S \ln n \left(\frac{\sigma_e}{\sqrt{m}\sigma_a}\right)^2 \tag{14.16}$$

for some constant $C_1 \approx 4$*.*

Hence as before, if $\|\tilde{\mathbf{s}} - \mathbf{s}\|_2^2 \leq 1/4$, we will have $\|\tilde{\mathbf{s}} - \mathbf{s}\|_\infty \leq 1/2$ and one can then decode the coefficients of $\mathbf{s}$ by rounding $\tilde{\mathbf{s}}$. From the theorem above, this condition is satisfied with high probability as soon as:

$$2C_1^2 \frac{S\ln n}{m}\left(\frac{\sigma_e}{\sigma_a}\right)^2 \leq \frac{1}{4}.$$

Here, we focus on solving the ILWE problem using only a small number of samples, typically, $m \leq n$. Thus, we deduce that the compressed sensing methodology can be successfully applied when

$$2C_1^2 S\ln n\left(\frac{\sigma_e}{\sigma_a}\right)^2 \leq \frac{m}{4} \leq \frac{n}{4},$$

and then,

$$S \leq \frac{n}{8C_1^2\ln n}\left(\frac{\sigma_a}{\sigma_e}\right)^2. \tag{14.17}$$

Let us discuss the practicality of this approach with regards to the parameters of the ILWE problem. First of all, note that in order to make Condition 14.17 non-vacuous, one needs σ_e and σ_a to satisfy:

$$2C_1\sqrt{\frac{2\ln n}{n}} \leq \frac{\sigma_a}{\sigma_e} \leq 2C_1\sqrt{2\ln n},$$

where the lower bound follows from the fact that S is a positive integer, and the upper bound from the observation that the right-hand side of 14.17 must be smaller than n to be of any interest compared to the trivial bound $S \leq n$. Practically speaking, this means that this approach is only interesting when the ratio σ_e/σ_a is relatively small; concrete bounds are provided in Table 14.1 for various sparsity levels and dimensions ranging from 128 to 2048.

Table 14.2 – Sets of BLISS parameters.

	BLISS–0	BLISS–I	BLISS–II	BLISS–III	BLISS–IV
$n' = 2n$	512	1024	1024	1024	1024
Modulus q	7681	12289	12289	12289	12289
Secret key densities (δ_1, δ_2)	(0.55, 0.15)	(0.3,0)	(0.3,0)	(0.42,0.3)	(0.45,0.06)
σ	100	215	107	250	271
κ	12	23	23	30	39
d	5	10	10	9	8

14.3 Application to the BLISS Case

We first recall the parameters of BLISS in Table 14.2.

14.3.1 Description of the Attack

Recall that by side channels, it is possible to have access to traces

$$b = \langle \mathbf{z}, \mathbf{sc} \rangle = \langle \mathbf{a}, \mathbf{s} \rangle + e,$$

where, e is an error term $e = \mathbf{z_2} - \mathbf{2^d z_2 s}_2$, $\mathbf{a} = (\mathbf{z}_1\bar{\mathbf{c}} \mid 2^d \mathbf{z}_2^\dagger \bar{\mathbf{c}}) \in \mathbb{Z}^{2n}$ is known from the signature, and $\mathbf{s}$ is the secret key we aim to recover. This is indeed an ILWE instance in dimension $n' = 2n$. Before actually solving it, let us determine the distributions χ_a and χ_e of coefficients of $\mathbf{a}$ and of the error e. It should therefore be feasible to recover the full secret key with the least squares method using about $m = \Omega\left((\sigma_e/\sigma_a)^2 \ln n\right)$.

Below, we detail the distributions χ_a and χ_e, and give an estimation of the value of σ_e and σ_a

Distribution of the error. Let us denote by $\mathbf{w}$ the vector $\mathbf{z}_2 - 2^d \mathbf{z}_2^\dagger$ and by $\mathbf{u}$ the vector $\mathbf{s}_2\mathbf{c}$, so that $e = \langle \mathbf{w}, \mathbf{u} \rangle$. Roughly speaking, $\mathbf{z}_2^\dagger$ is essentially obtained by keeping the $(\log q - d)$ most significant bit of $\mathbf{z}_2$, in a centred way. Therefore, we heuristically expect each coefficient of $\mathbf{w}$ to belong to $[-2^{d-1}, 2^{d-1}] \cap \mathbb{Z}$. In other words, we claim that $\mathbf{w} \sim \mathscr{U}_\alpha^n$, with $\alpha = 2^{d-1}$. In particular, the variance of these coefficients is $\alpha(\alpha+1)/3 \simeq 2^{2d}/12$.

As for u, its coefficients are obtained by the sum of κ coefficients of $\mathbf{s}_2$. We recall that $\mathbf{s}_2 = 2\mathbf{g}+1$, where each coefficient of $\mathbf{g}$ is a random polynomial with $\delta_1 \cdot n$ coefficients in $\{-1, 1\}$ and $\delta_2 \cdot n$ coefficients in $\{-2, 2\}$. Then, ignoring the constant coefficient which is shifted by 1, $\mathbf{s}_2$ can be seen as a vector with $\delta_1 \cdot n$ coefficients equal to ± 2 and $\delta_2 \cdot n$ coefficients equal to ± 4. This is a somewhat complicated distribution to describe, but we do not make a large approximation by pretending that all coefficients are sampled independently from $\{-4, -2, 0, 2, 4\}$ with respective probability $\delta_2/2, \delta_1/2, (1 - \delta_1 - \delta_2), \delta_1/2, \delta_2/2$. With this approximation, the distribution of each coefficient of $\mathbf{u}$ is clearly centred, and thus the variance of each coefficient of $\mathbf{u}$ is given by:

$$\kappa \mathbb{E}[\mathbf{s}_2[i]^2] = \kappa(16\delta_2 + 4\delta_1).$$

Now, if $\mathbf{u} = (u_1, \ldots, u_n)$, and $\mathbf{w} = (w_1, \ldots w_n)$, under the heuristic approximation above, the error e follows a certain bound distribution χ_e of variance σ_e^2, given by:

$$\begin{aligned}
\sigma_e^2 &= \mathbb{E}[e^2] \\
&= \mathbb{E}\left[\left(\sum_{i=1}^{n} w_i u_i \right)^2 \right] \\
&= \mathbb{E}\left[\left(\sum_{i,j} w_i w_j u_i u_j \right) \right] \\
&= \sum_{i,j} \mathbb{E}[w_i w_j u_i u_j].
\end{aligned}$$

Table 14.3 – Parameter estimation for ILWE instances arising from the side channel attack.

	BLISS–0	BLISS–I	BLISS–II	BLISS–III	BLISS–IV
$n' = 2n$	512	1024	1024	1024	1024
σ_a (theory)	346	1031	513	1369	1692
σ_e (theory)	1553	49695	49695	38073	24535
$\sigma_{\mathbf{a}_1}$ (exp.)	347	1031	513	1370	1691
$\sigma_{\mathbf{a}_2}$ (exp.)	349	2009	1418	1782	1814
σ_e (exp.)	1532	42170	32319	38627	23926

Since $\mathbf{u}$ and $\mathbf{w}$ are independent and the distribution of their coefficients is centred, this expression can be simplified as follows:

$$\begin{aligned}\sigma_e^2 &= \sum_i \mathbb{E}[w_i^2 u_i^2] \\ &= \sum_i \mathbb{E}[w_i^2]\mathbb{E}[u_i^2] \\ &= n\mathrm{Var}(w_i)\mathrm{Var}(u_i) \\ &= n\frac{2^{2d}}{12}\kappa(4\delta_1 + 16\delta_2) \\ &= \frac{2^{2d}}{3}(\delta_1 + 4\delta_2)\kappa n.\end{aligned}$$

Distribution of a. Let us denote by $\mathbf{a}^{(1)}$ the vector $\mathbf{z}_1\bar{\mathbf{c}}$, and by $\mathbf{a}^{(2)}$, the vector $2^d\mathbf{z}_2^\dagger\bar{\mathbf{c}}$, so that $\mathbf{a}$ can be rewritten as $\mathbf{a} = (\mathbf{a}^{(1)}|\mathbf{a}^{(2)})$. We first consider the distribution of $\mathbf{a}^{(1)}$.

The rejection sampling ensures that the coefficients of $\mathbf{z}_1$ are independent and distributed according to a discrete Gaussian D_α, of standard deviation σ. Furthermore, $\mathbf{c}$ is a random polynomial with coefficients in $\{0,1\}^n$ of Hamming weigh $wt(\mathbf{c}) = \kappa$, and $\mathbf{c}^*$ has a similar shape, up to the sign of coefficients. It follows that the coefficients of $\mathbf{z}_1\bar{\mathbf{c}}$ are all linear combinations with ± 1 coefficients of exactly κ independent samples from D_α^n, and the signs clearly do not affect the resulting distribution.

Therefore, we denote by χ_a the distribution obtained by summing κ independent samples from D_α. The coefficients of $\mathbf{a}^{(1)}$ follow χ_a. It is not exactly correct that $\mathbf{z}_1\bar{\mathbf{c}}$ follows χ_a^n, as the coefficients are not rigorously independent, but we heuristically ignore that subtlety and pretend it does. Note that χ_a has a variance of:

$$\sigma_a^2 = \kappa\mathrm{Var}(D_\alpha) = \kappa\sigma^2.$$

As $\mathbf{z}_2^\dagger$ is essentially the $(\log q - d)$ most significant bit of $\mathbf{z}_2$, the distribution of $2^d\mathbf{z}_2^\dagger$ is close to that of $\mathbf{z}_2$. The distributions cannot coincide exactly however, since all the coefficients of $2^d\mathbf{z}_2^\dagger$ are multiples of 2^d (which is not the case for $\mathbf{z}_2$). The difference does not matter much, for our purpose, and we will heuristically assume that the entire vector $\mathbf{a}$ is distributed as χ_a^{2n}.

14.3.2 Experimental Distributions

A number of approximations which cannot be precisely satisfied in practice, were made in the description of the distributions given in the previous section. In order to verify our claims, we did the following experiment: we carried out a numerical simulation, comparing our estimates for the standard deviation of e, $\mathbf{a}_1$, and $\mathbf{a}_2$ with the actual standard deviation obtained from the actual rejection sampling leakage in BLISS.

These simulations were carried in Python using the numpy package. We collected 10000 BLISS traces for each set of parameters, and computed the corresponding e values (that we stored in a table T_e) as well as the $\mathbf{a}^{(1)}$ and $\mathbf{a}^{(2)}$ vectors, and stored each of their coefficients respectively in

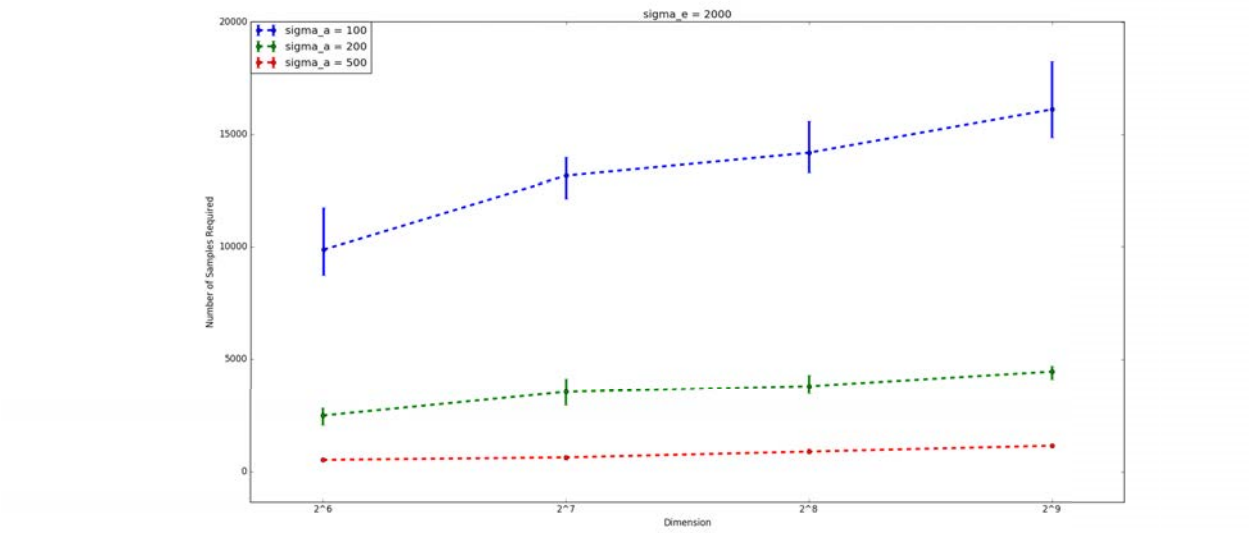


Figure 14.1 – Results for $\sigma_e = 2000$.

tables T_1 and T_2. We then asked numpy to compute the standard deviation of each table. We summarise these results in Table 14.3. As we can see, the experimental values match the heuristic estimate quite closely overall.

14.4 Experiments

We are now going to present experimental results for recovering ILWE secrets using linear regression. We first focus ILWE instances in dimension n with discrete Gaussian distributions χ_a and χ_e of respective standard deviation σ_a and σ_a, for various values of n, σ_a, σ_e. Then, we consider instances arising from BLISS side-channel leakage, leading to BLISS secret key recovery.

14.4.1 Plain ILWE

Recall that the ILWE problem is parametrised by $n, m \in \mathbb{Z}$ and probability distributions χ_a and χ_e. Samples are computed as $\mathbf{b} = A\mathbf{s} + \mathbf{e}$, where $\mathbf{s} \in \mathbb{Z}^n$, $\mathbf{b} \in \mathbb{Z}^m$, A being an m-by-n matrix with entries drawn from χ_a, and $\mathbf{e} \in \mathbb{Z}^m$ with entries drawn from χ_e. Choosing χ_a and χ_e as discrete Gaussian distributions with standard deviations σ_a and σ_e respectively, we investigated the number of samples, m required to recover ILWE secret vectors $\mathbf{s} \in \mathbb{Z}^n$ for various concrete values of n, σ_a and σ_e. We sampled sparse secret vectors $\mathbf{s}$ uniformly at random from the set of vectors with $\lceil 0.15n \rceil$ entries set to ± 1, $\lceil 0.15n \rceil$ entries set to ± 2, and the rest zero. Choosing $\mathbf{s}$ according to this distribution is not entirely an arbitrary decision. Indeed, it is quite similar to the distribution of the secret in BLISS, (choosing $\delta_1 = \delta_2 = 0.15$) up to a 2 constant.

We present two types of experimental results for plain ILWE. In our first experiment, we first estimated the number of samples m required to recover the secret perfectly, for different values of n, σ_a, and σ_e. Then, fixing m, we ran several tests (10 per triplet (n, σ_a, σ_e)) and checked if we were indeed able to fully recover $\mathbf{s}$ over the random choices of $\mathbf{s}$, A and e. In Table 14.4, we summarise the results obtained.

The second experiment consisted in investigating the distribution of the minimum value of m required to recover the secret perfectly, over the random choices of $\mathbf{s}$, A, and $\mathbf{e}$, when the linear regression method was run to completion. In other words, for fixed n, σ_a, and σ_e, more and more samples were generated until the secret could be perfectly recovered.

The results obtained for $\sigma_e = 2000$ are plotted in Figure 14.1. The figure plots the dimension n against the interquartile mean number of samples m required to recover the secret, for $\sigma_a = 100$, 200, and 500. The error bars show the upper and lower quartiles for the number of samples required.

Table 14.4 – Practical results of the experiments on ILWE.

n	σ_a	σ_e	m	Success
128	100	1000	3300	6/10
	100	2000	11500	6/10
	100	5000	65000	4/10
	200	1000	900	5/10
	200	2000	4000	7/10
	200	5000	17000	4/10
	300	1000	550	10/10
	300	2000	1890	8/10
	300	5000	9000	7/10
	400	1000	350	8/10
	400	2000	800	5/10
	400	5000	5750	7/10
	500	1000	350	10/10
	500	2000	700	6/10
	500	5000	3300	4/10
256	100	1000	5600	9/10
	100	2000	14500	6/10
	100	5000	95000	7/10
	200	1000	1300	6/10
	200	2000	4700	8/10
	200	5000	23000	6/10
	300	1000	900	9/10
	300	2000	1800	5/10
	300	5000	12000	8/10
	400	1000	550	10/10
	400	2000	1300	9/10
	400	5000	6000	5/10
	500	1000	450	7/10
	500	2000	950	8/10
	500	5000	4200	5/10
512	100	1000	5100	7/10
	100	2000	16000	4/10
	200	1000	1600	9/10
	200	2000	5200	7/10
	300	1000	1000	8/10
	300	2000	2600	8/10
	400	1000	900	10/10
	400	2000	1500	4/10
	500	1000	800	10/10
	500	2000	1250	8/10
1024	100	1000	5950	10/10
	100	2000	19000	5/10
	200	1000	2250	6/10
	200	2000	5900	6/10
	300	1000	1550	7/10
	300	2000	3350	6/10
	400	1000	1350	9/10
	400	2000	2300	7/10
	500	1000	1500	10/10
	500	2000	1900	8/10

Table 14.5 – Estimated value of S for $n = 1024$, depending on the ratio σ_e/σ_a.

σ_e/σ_a	10	5	10/3	5/2	2
S	0.01	0.05	0.1	0.18	0.23

Table 14.6 – Number of samples required to recover the secret key.

	# Trials	Min	LQ	IQM	UQ	Max
BLISS–0	12	1203	1254	1359.5	1515	1641
BLISS–I	12	14795	18648	20382.9	21789	24210
BLISS–II	8	19173	20447	22250.3	24482	29800

The results of our second experiment are consistent with the theoretical results given in Section 14.2.1. Indeed, according to Equation 14.6, we require

$$m \geq C' \cdot \frac{\sigma_e^2}{\sigma_a^2} \ln n$$

samples in order to recover the secret correctly. The dimension n on the horizontal axis of each graph is plotted on a logarithmic scale. Therefore, theory predicts that we should observe a straight line, which the graph confirms.

The LP modelling is disappointing. For the values of n, σ_a, σ_e given in Table 14.4, the LP modelling does not pay off. Indeed, for the same value of m, we were never able to fully recover the secret with Gurobi using the following model:

$$\begin{aligned} &\text{Find } \tilde{\mathbf{s}} \text{ such that:} \\ &-\sigma_e\sigma_a\sqrt{2m\ln n} \leq \left[{}^tA(\mathbf{b} - A\tilde{\mathbf{s}})\right]_i \leq \sigma_e\sigma_a\sqrt{2m\ln n}. \end{aligned}$$

Adding constraints on $\|\tilde{\mathbf{s}}\|_1$ as in the program given in Equation 14.16, did not help.

Another interesting aspect would have been to investigate how the sparse be compressed-sensing methods behave in real-life. The problem is that, when $\sigma_e > \sigma_a$, they can only be applied for large values of n. Indeed, from Equation 14.17, for known values of n, σ_e, σ_a, it is possible to give an upper bound on the number of non-zero coefficients the secret must have for these methods to work. A quick simulation for $n = 1024$ proves that S would be too small for practical experiment. We detail the value of S thus obtained in table 14.5. We tried to run tests for $n = 16384, S = 3, m = 15000$ and $\sigma_e/\sigma_a = 2$. As the generation of that amount of samples took a lot of time, we try to do it on a cluster. Unfortunatetly, our job was killed after a week.

14.4.2 BLISS leakage

The tests we performed were simulations. We did not actually mount a real side channel attack. The attack was implemented using SAGEMath, and ran on a laptop with 2.60GHz processor.

Table 14.7 – Typical timings for secret key recovery.

	Typical ILWE sample gen.	Typical time for regression
BLISS–0	$\approx$ 2min	$\approx$ 5sec
BLISS–I	$\approx$ 10min	$\approx$ 2min
BLISS–II	$\approx$ 10min	$\approx$ 2min

Considering an instance of the ILWE problem which arises from a BLISS side-channel leakage, we used linear regression to recover BLISS secret keys. Several trials were performed. For each of them more and more ILWE samples were generated until the secret key could be recovered. For BLISS–0, the secret key was fully recovered using only linear regression. For BLISS–I and BLISS–II, the secret key could not be entirely recovered because of memory issues. However, we noticed that in practice, we could recover the first half of the secret key correctly using far fewer samples. Since the two halves of the secret key are related by the public key, this is sufficient to compute the entire secret key. Therefore, for BLISS–I and BLISS–II, we stopped generating samples as soon as the least squares estimator correctly recovered the first half of the secret.

For these two different scenarios, we obtained the results displayed on Table 14.6, which gives information on the range (Min and Max), Lower (LQ) and Upper (UQ) quartiles, and interquartile mean (IQM) of the number of samples required. Typical timings for the attacks are displayed in Table 14.7. Timings are in the orders of minutes and seconds. By comparison, some of the attacks from [EFGT17] took hours, or even days, of CPU time.

Conclusion

We focused mainly on two variants of the LWE problem: the (Ring-)LWR problem and the Integer-LWE problem.

We first proposed a new PRG based on the RLWR problem, belonging to the SPRING family, and called SPRING-RS. This new scheme is faster than previous SPRING instantiation: SPRING-BCH and SPRING-CRT, and we claim that it is not less secure. Furthermore, we proposed experimentation which shows that our SPRING variant is a very efficient stream-cipher. This leads us to thinks that lattice-based cryptography can be used with high performance.

The concrete security of SPRING however is hard to estimate. Unlike the BPR family, SPRING do not enjoy reduction to worst-case lattice problem. As such our security estimate only relies on the best known attacks up to now. It could be interesting to see if SPRING-RS is vulnerable to other kind of attacks than the one we described, and if so, if these attacks could be back-ported to the other SPRING instantiations, and (possibly with some adaptation) to other LWR/LWE based cryptographic primitive.

The second part of our work was to study a variant of the LWE problem "without the modulus", that we called ILWE (for Integer-LWE). We first show that this problem is not hard, and we gave relatively tight bounds on the number of samples that have to be used to recover the secret of such an instance. This work was motivated by a side-channel attack against the BLISS signature scheme. In this setting, we were effectively able to recover BLISS secret key in the case of BLISS–0, BLISS–I and BLISS–II.

We would like to stress that the methods we present to attack ILWE are not new, and some are even widely used in numerical analysis or statistical learning theory. A concrete analysis of this problem was, to our mind, quite interesting however. Finally, it could be interesting to see if this attack could work for other schemes that rely on the "Fiat-Shamir with abort" paradigm, typically the Dilithium signature proposed as a candidate to the NIST competition.

General Conclusion

"All's well that ends better."

– J.R.R. Tolkien –

This thesis focuses on the algorithmic aspect of various problems related to linear algebra and cryptography. In this last chapter, we summarise all the results that have been presented here, and present some open problems.

In the first part of this thesis, we proposed a new sparse gaussian elimination algorithm, which is always faster than both the classical (right-looking) sparse gaussian elimination algorithm, and the (left-looking) GPLU algorithm. We also proposed new pivots selection heuristics, which prove to be very efficient in some cases. Finally, we showed, that in some cases, our algorithm is also much faster than the iterative Wiedemann Algorithm. In the future, it could be interesting to see if this algorithm could be utilised as a subroutine in factorisation or discrete logarithm problems.

The second part focused on the 3XOR problem. We proposed a new algorithm, which is a generalisation of a previous algorithm from Joux, which can recover all solutions to the problem and works for all size of input lists. It is in theory n times faster than the quadratic algorithm, with input lists $2^{n/3}$, where n is the bit-length of the entries of the lists. We also proposed an adaptation of Baran Demain and Pătraşu 3SUM algorithm to the 3XOR setting. We showed that, in our context, this algorithm is unpractical. In the future, it could be interesting to see whether our algorithm can be applied modulo some changes to other groups (e.g. 3SUM over $\mathbb{Z}$). An other interesting aspect would be to know if it can be utilised to improve the kXOR problem where k is not a power of two.

Finally, the third and last part of this thesis is dedicated to problems related to LWE. We mainly focused on two variants of this problems: the (Ring-)LWR problem and the Integer-LWE problem. We first presented a new PRG based on the RLWR problem. Although this PRG does not enjoy security reduction to hard lattice problems, we claim that it seems to be secure. In particular the underlying RLWR seems hard to solve using the state of the art LWE algorithms. The second contribution is the study of the ILWE problem. We showed that this problem is in fact much simpler than LWE, and can be solve using the least squares method. Finally, we presented an application of this problem in a side-channel attack against the BLISS signature scheme. In the future, It could be interesting to see whether this attack could also work on other lattice-based signatures which relies on the same "Fiat-Shamir with abort" paradigm, typically the Dilithium candidate to the NIST post-quantum cryptography contest.

Bibliography

[AB12] Martin Albrecht and Gregory Bard. *The M4RI Library – Version 20121224*. The M4RI Team, 2012. 105, 109

[ACF+15] Martin Albrecht, Carlos Cid, Jean-Charles Faugere, Robert Fitzpatrick, and Ludovic Perret. On the complexity of the bkw algorithm on lwe. *Designs, Codes and Cryptography*, 74(2):325–354, 2015. 142

[ACPS09] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In *Advances in Cryptology-CRYPTO 2009*, pages 595–618. Springer, 2009. 123

[ADD96] Patrick Amestoy, Timothy Davis, and Iain Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Analysis Applications*, 17(4):886–905, 1996. 33

[ADKL01] Patrick Amestoy, Ian Duff, Jacko Koster, and Jean-Yves L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001. 36

[AFFP14] Martin Albrecht, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. Lazy modulus switching for the bkw algorithm on lwe. In *International Workshop on Public Key Cryptography*, pages 429–445. Springer, 2014. 123

[AFM+13] Guiseppe Ateniese, Giovanni Felici, Luigi Mancini, Angelo Spognardi, Antonio Villani, and Domenico Vitali. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. *CoRR*, 2013. ix, x

[AG11] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In *International Colloquium on Automata, Languages, and Programming*, pages 403–415. Springer, 2011. 123, 142, 155

[AGL+87] Cleveland Ashcraft, Roger Grimes, John Lewis, Barry Peyton, Horst Simon, and Petter Bjørstad. Progress in sparse matrix methods for large linear systems on vector supercomputers. *Int. J. High Perform. Comput. Appl.*, 1(4):10–30, December 1987. 36

[AJPS17] Divesh Aggarwal, Antoine Joux, Anupam Prakash, and Miklos Santha. A new public-key cryptosystem via mersenne numbers. Technical report, Cryptology ePrint Archive, Report 2017/481, 2017. http://eprint. iacr. org/2017/481, 2017. 123

[Ajt96] Miklós Ajtai. Generating hard instances of lattice problems. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 99–108. ACM, 1996. 138

[AKPW13] Joël Alwen, Stephan Krenn, Krzysztof Pietrzak, and Daniel Wichs. Learning with rounding, revisited. In *Advances in Cryptology–CRYPTO 2013*, pages 57–74. Springer, 2013. 124, 137

[AKS01] Miklós Ajtai, Ravi Kumar, and Dandapani Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 601–610. ACM, 2001. 139

[AKS02] Miklós Ajtai, Ravi Kumar, and Dandapani Sivakumar. Sampling short lattice vectors and the closest lattice vector problem. In *Computational Complexity, 2002. Proceedings. 17th IEEE Annual Conference on*, pages 53–57. IEEE, 2002. 139

[Alb17] Martin Albrecht. On dual lattice attacks against small-secret lwe and parameter choices in helib and seal. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 103–129. Springer, 2017. 123

[APS15] Martin Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015. 125, 156

[AR13] Giulia Alberini and Alon Rosen. Efficient rounding procedures of biased samples, 2013. 147

[ASA16] Jacob Alperin-Sheriff and Daniel Apon. Dimension-preserving reductions from lwe to lwr. *IACR Cryptology ePrint Archive*, 2016(589), 2016. 137

[Bab86] László Babai. On lovász'lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986. 139, 140

[BBL+14] Abhishek Banerjee, Hai Brenner, Gaëtan Leurent, Chris Peikert, and Alon Rosen. Spring: Fast pseudorandom functions from rounded ring products. In *International Workshop on Fast Software Encryption*, pages 38–57. Springer, 2014. viii, 124, 126, 144, 146, 147, 148, 151, 153, 155, 158, 159

[BBS86] Lenore Blum, Manuel Blum, and Mike Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on computing*, 15(2):364–383, 1986. 143

[BBSJ16] Ahmad Boorghany, Siavash Bayat-Sarmadi, and Rasool Jalili. Efficient lattice-based authenticated encryption: A practice-oriented provable security approach. Cryptology ePrint Archive, Report 2016/268, 2016. `https://eprint.iacr.org/2016/268`. 124

[BCCN96] Egon Balas, Sebastian Ceria, Gérard Cornuéjols, and N. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19(1):1–9, 1996. 178

[BCG16] Eric Blais, Clément Louis Canonne, and Tom Gur. Alice and bob show distribution testing lower bounds (they don't talk to each other anymore.). *Electronic Colloquium on Computational Complexity (ECCC)*, 23(168), 2016. 130

[BCM+13] Battista. Biggio, Igino. Corona, Davide. Maiorca, Blaine. Nelson, Nedim. Šrndić, Pavel. Laskov, Giorgio. Giacinto, and Fabio. Roli. Evasion attacks against machine learning at test time. In *Machine Learning and Knowledge Discovery in Databases: European Conference, Proceedings, Part III*, pages 387–402. Springer Berlin Heidelberg, 2013. ix

[BCP97] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993). 36

[BD16] Charles Bouillaguet and Claire Delaplace. Sparse gaussian elimination modulo p: an update. In *International Workshop on Computer Algebra in Scientific Computing*, pages 101–116. Springer, 2016. iv, v, x, 8, 9, 39, 40, 44

[BDE+18] Jonathan Bootle, Claire Delaplace, Thomas Espitau, Pierre-Alain Fouque, and Mehdi Tibouchi. Lwe without modular reduction and improved side-channel attacks against bliss, 2018. Accepted in ASIACRYPT 2018. viii, x

[BDF18] Charles Bouillaguet, Claire Delaplace, and Pierre-Alain Fouque. Revisiting and improving algorithms for the 3xor problem. *IACR Transactions on Symmetric Cryptology*, 2018(1):254–276, 2018. vi, x, 66, 93, 111

[BDFK17] Charles Bouillaguet, Claire Delaplace, Pierre-Alain Fouque, and Paul Kirchner. Fast lattice-based encryption: Stretching spring. In *International Workshop on Post-Quantum Cryptography*, pages 125–142. Springer, 2017. vii, x, 124, 151, 156

[BDP05] Ilya Baran, Erik Demaine, and Mihai Pătraşcu. Subquadratic algorithms for 3SUM. In *Workshop on Algorithms and Data Structures*, pages 409–421. Springer, 2005. vii, 66, 111, 112

[BDPVA09] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3(30), 2009. 143

[BDV17] Charles Bouillaguet, Claire Delaplace, and Marie-Emilie Voge. Parallel sparse pluq factorization modulo p. In *Proceedings of the International Workshop on Parallel Symbolic Computation*, PASCO 2017, pages 8:1–8:10. ACM, 2017. v, x, 8, 9, 37, 44, 47, 51

[BEF+16] Brice Boyer, Christian Eder, Jean-Charles Faugère, Sylvian Lachartre, and Fayssal Martani. GBLA - gröbner basis linear algebra package. *CoRR*, abs/1602.06097, 2016. 37, 44

[Ber07] Daniel Bernstein. Better price-performance ratios for generalized birthday attacks, 2007. vi, 66, 89

[BFS04] Magali Bardet, Jean-Charles Faugere, and Bruno Salvy. On the complexity of gröbner basis computation of semi-regular overdetermined algebraic equations. In *Proceedings of the International Conference on Polynomial System Solving*, pages 71–74, 2004. 158

[BG14] Shi Bai and Steven Galbraith. Lattice decoding attacks on binary lwe. In *Australasian Conference on Information Security and Privacy*, pages 322–337. Springer, 2014. 123

[BGL+14] Hai Brenner, Lubos Gaspar, Gaëtan Leurent, Alon Rosen, and François-Xavier Standaert. Fpga implementations of spring. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 414–432. Springer, 2014. 124, 146, 155

[BGM+16] Andrej Bogdanov, Siyao Guo, Daniel Masny, Silas Richelson, and Alon Rosen. On the hardness of learning with rounding over small modulus. In *Theory of Cryptography Conference*, pages 209–224. Springer, 2016. 124, 137

[BGPW16] Johannes Buchmann, Florian Göpfert, Rachel Player, and Thomas Wunderer. On the hardness of lwe with binary error: revisiting the hybrid lattice-reduction and meet-in-the-middle attack. In *International Conference on Cryptology in Africa*, pages 24–43. Springer, 2016. 123

[BHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload–a cache attack on the bliss lattice-based signature scheme. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 323–345. Springer, 2016. 125

[BJMM12] 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. In *EUROCRYPT*, pages 520–536. Springer, 2012. 100, 101

[BKW03] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *Journal of the ACM (JACM)*, 50(4):506–519, 2003. 65, 142, 155

[BLN+09] Daniel Bernstein, Tanja Lange, Ruben Niederhagen, Christiane Peters, and Peter Schwabe. FSBday: Implementing Wagner's Generalized Birthday Attack. In *INDOCRYPT*, pages 18–38, 2009. 66, 89, 108

[BLP+13] Zvika Brakerski, Adeline Langlois, Chris Peikert, Oded Regev, and Damien Stehlé. Classical hardness of learning with errors. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 575–584. ACM, 2013. 123, 135

[Blu99] Norbert Blum. *A simplified realization of the Hopcroft Karp approach to maximum matching in general graphs*. Inst. für Informatik, 1999. 19

[BM17] Leif Both and Alexander May. The approximate k-list problem. *IACR Transactions on Symmetric Cryptology*, 2017(1):380–397, 2017. 73

[BM18] Leif Both and Alexander May. Decoding linear codes with high error rate and its impact for lpn security. In *International Conference on Post-Quantum Cryptography*, pages 25–46. Springer, 2018. 101

[BMVT78] Elwyn Berlekamp, Robert McEliece, and Henk Van Tilborg. On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information Theory*, 24(3):384–386, 1978. 100

[Bor13] Knut Borg. Real time detection and analysis of pdf-files. Master's thesis, Gjovik University College, 2013. ix

[BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 719–737. Springer, 2012. iii, vii, 124, 136, 137, 144, 145, 146, 147, 155

[Bra02] Ulrik Brandes. Eager *st*-ordering. In *Proceedings of the 10th Annual European Symposium on Algorithms*, ESA '02, pages 247–256, London, UK, UK, 2002. Springer-Verlag. 49

[BTX18] Jonathan Bootle, Mehdi Tibouchi, and Keita Xagawa. Cryptanalysis of compact-lwe. In *Cryptographers' Track at the RSA Conference*, pages 80–97. Springer, 2018. 123

[BW09] Nikhil Bansal and Ryan Williams. Regularity lemmas and combinatorial algorithms. In *Foundations of Computer Science, 2009. FOCS'09. 50th Annual IEEE Symposium on*, pages 745–754. IEEE, 2009. 78

[CAD15] CADO-NFS, an implementation of the number field sieve algorithm, 2015. Release 2.2.0. iii, 36, 44

[Cav00] Stefania Cavallar. Strategies in filtering in the number field sieve. In *Algorithmic Number Theory, 4th International Symposium, ANTS-IV, Leiden, The Netherlands, July 2-7, 2000, Proceedings*, pages 209–232, 2000. 36

[CC79] Guy Chaty and Mikael Chein. Ordered matchings and matchings without alternating cycles in bipartite graphs. *Utilitas Mathematica*, 16:183 – 187, January 1979. 49

[CC98] 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. on Information Theory*, 44(1):367–378, 1998. 101

[CDDV18] Bonan Cuan, Aliénor Damien, Claire Delaplace, and Mathieu Valois. Malware detection in PDF files using machine learning. In *Proceedings of the 15th International Joint Conference on e-Business and Telecommunications, ICETE 2018 - Volume 2: SECRYPT, Porto, Portugal, July 26-28, 2018.*, pages 578–585, 2018. ix, x

[CDHR08] Yanqing Chen, Timothy Davis, William Hager, and Sivasankaran Rajamanickam. Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. *ACM Trans. Math. Softw.*, 35(3):22:1–22:14, October 2008. 36

[CG90] John Coffey and Rodney Goodman. The complexity of information set decoding. *IEEE Transactions on Information Theory*, 36(5):1031–1037, 1990. 101

[Cha15] Timothy Chan. Speeding up the four russians algorithm by about one more logarithmic factor. In *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 212–217. Society for Industrial and Applied Mathematics, 2015. 78

[Che13] Yuanmi Chen. *Réduction de réseau et sécurité concrete du chiffrement completement homomorphe*. PhD thesis, Paris 7, 2013. 141

[CJM02] Philippe Chose, Antoine Joux, and Michel Mitton. Fast correlation attacks: An algorithmic point of view. In *EUROCRYPT*, pages 209–221. Springer, 2002. 73

[CLRS01] Thomas Cormen, Charles Eric Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to algorithms*, volume 6. MIT press Cambridge, 2001. 16, 18

[CN11] Yuanmi Chen and Phong Nguyen. Bkz 2.0: Better lattice security estimates. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 1–20. Springer, 2011. 141

[Con13] Contagio: Malware dump. `http://contagiodump.blogspot.fr/2013/03/16800-clean-and-11960-malicious-files.html`, 2013. x

[Cop94] Don Coppersmith. Solving homogeneous linear equations over $\mathbb{F}_2$ via block wiedemann algorithm. *Mathematics of Computation*, 62(205):333–350, 1994. 7, 27, 28, 29

[CT06] Emmanuel Candès and Terence Tao. Near-optimal signal recovery from random projections: Universal encoding strategies? *IEEE transactions on information theory*, 52(12):5406–5425, 2006. 179

[CT07] Emmanuel Candès and Terence Tao. The dantzig selector: Statistical estimation when p is much larger than n. *The Annals of Statistics*, 35(6):2313–2351, 2007. viii, 126, 179, 180

[CVE17] CVEDetails. Adobe vulnerabilities statistics. `https://www.cvedetails.com/product/497/Adobe-Acrobat-Reader.html`, 2017. ix

[Dan51] George Dantzig. Maximization of a linear function of variables subject to linear inequalities. *New York*, 1951. 178

[Dav04] Timothy Davis. Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Transactions On Mathematical Software*, 30(2):196–199, June 2004. 36

[Dav06] Timothy Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006. 11, 31, 37

[DDKS12] Itai Dinur, Orr Dunkelman, Nathan Keller, and Adi Shamir. Efficient dissection of composite problems, with applications to cryptanalysis, knapsacks, and combinatorial search problems. In *CRYPTO*, pages 719–740. Springer, 2012. 73

[DDLL13] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In *Advances in Cryptology–CRYPTO 2013*, pages 40–56. Springer, 2013. vii, 125, 161, 162, 163, 164

[DEG+99] James Demmel, Stanley Eisenstat, John Gilbert, Xiaoye Li, and Joseph Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999. 36

[DEGU07] Jean-Guillaume Dumas, Philippe Elbaz-Vincent, Pascal Giorgi, and Anna Urbanska. Parallel computation of the rank of large sparse matrices from algebraic k-theory. In *Parallel Symbolic Computation, PASCO 2007, International Workshop, 27-28 July 2007, University of Western Ontario, London, Ontario, Canada*, pages 43–52, 2007. 57, 58

[DER89] Ian Duff, Albert Erisman, and John Reid. *Direct Methods for Sparse Matrices*. Numerical Mathematics and Scientific Computation. Oxford University Press, USA, first paperback edition edition, 1989. 36

[DGK+10] Yevgeniy Dodis, Shafi Goldwasser, Yael Tauman Kalai, Chris Peikert, and Vinod Vaikuntanathan. Public-key encryption schemes with auxiliary inputs. In *Theory of Cryptography Conference*, pages 361–381. Springer, 2010. 123

[DH76] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976. 3

[DL13] Léo Ducas and Tancrède Lepoint. A proof-of-concept implementation of bliss. *Available under the CeCILL License at http://bliss.ens.di.fr*, page 9, 2013. 164

[DMQ13] Nico Döttling and Jörn Müller-Quade. Lossy codes and a new variant of the learning-with-errors problem. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 18–34. Springer, 2013. 123

[DN10] Timothy Davis and Ekanathan Palamadai Natarajan. Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems. *ACM Trans. Math. Softw.*, 37(3), 2010. 36

[DR79] Ian Duff and John Reid. Some design features of a sparse matrix code. *ACM Trans. Math. Softw.*, 5(1):18–35, March 1979. 36

[DR83] Ian Duff and John Reid. The multifrontal solution of indefinite sparse symmetric linear. *ACM Trans. Math. Softw.*, 9(3):302–325, September 1983. 36

[DR99] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael, 1999. 143

[DSW18] Martin Dietzfelbinger, Philipp Schlag, and Stefan Walzer. A subquadratic algorithm for 3xor. *arXiv preprint arXiv:1804.11086*, 2018. 73, 79, 80, 113

[dt16] The FPLLL development team. fplll, a lattice reduction library. Available at `https://github.com/fplll/fplll`, 2016. 141

[DTV15] Alexandre Duc, Florian Tramer, and Serge Vaudenay. Better algorithms for lwe and lwr. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 173–202. Springer, 2015. 142

[Dum91] Ilya Dumer. On minimum distance decoding of linear codes. In *Proc. 5th Joint Soviet-Swedish Int. Workshop Inform. Theory*, pages 50–52, 1991. 101

[Dum12] Jean-Guillaume Dumas. Sparse integer matrices collection, 2012. `http://hpac.imag.fr`. iv, v, 8, 9, 13, 29, 34, 40, 44, 57, 205

[DV02] Jean-Guillaume Dumas and Gilles Villard. Computing the rank of sparse matrices over finite fields. In Victor G. Ganzha, Ernst W. Mayr, and Evgenii V. Vorozhtsov, editors, *CASC'2002, Proceedings of the fifth International Workshop on Computer Algebra in Scientific Computing, Yalta, Ukraine*, pages 47–62. Technische Universität München, Germany, September 2002. iv, 7, 8, 9, 29

[Dvo61] Aryeh Dvoretzky. Some results on convex bodies and banach spaces. In *International Symposium Linear Spaces, Academic Press, Jerusalem*, pages 123–160, 1961. 177

[EFGT17] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Side-channel attacks on bliss lattice-based signatures: Exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1857–1874. ACM, 2017. vii, viii, 125, 126, 164, 165, 166, 186

[EHK+18] Andre Esser, Felix Heuer, Robert Kübler, Alexander May, and Christian Sohler. Dissection-bkw. In *CRYPTO*. Springer, 2018. 73

[EJ16] Thomas Espitau and Antoine Joux. Ialatred. Available at `https://almasty.lip6.fr/~espitau/latred.html`, 2016. 141

[EK97] Wayne Eberly and Erich Kaltofen. On randomized lanczos algorithms. In *Proceedings of the 1997 international symposium on Symbolic and algebraic computation*, pages 176–183. ACM, 1997. 28

[Eri95] Jeff Erickson. Lower Bounds for Linear Satisfiability Problems. In *SODA*, pages 388–395, 1995. 73

[Fau99] Jean-Charles Faugère. A new efficient algorithm for computing gröbner bases (f 4). *Journal of pure and applied algebra*, 139(1):61–88, 1999. 37, 158

[Fau02] 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, pages 75–83, New York, NY, USA, 2002. ACM. 37

[FF56] Lester Ford and Delbert Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8(3):399–404, 1956. 19

[FFL14] The FFLAS-FFPACK group. *FFLAS-FFPACK: Finite Field Linear Algebra Subroutines / Package*, v2.0.0 edition, 2014. `http://linalg.org/projects/fflas-ffpack`. 44

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with o(1) worst case access time. *J. ACM*, 31(3):538–544, June 1984. 74

[FL10] Jean-Charles Faugère and Sylvain Lachartre. Parallel gaussian elimination for gröbner bases computations in finite fields. In *PASCO*, pages 89–97, 2010. v, 9, 37, 40

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology—CRYPTO'86*, pages 186–194. Springer, 1986. 162

[FW93] Michael L Fredman and Dan E Willard. Surpassing the information theoretic bound with fusion trees. *Journal of computer and system sciences*, 47(3):424–436, 1993. 96

[Gal11] Steven Galbraith. Algorithms for the closest and shortest vector problem. *Mathematics of Public Key Cryptography*, 2011. 123, 139

[Gal13] Steven Galbraith. Space-efficient variants of cryptosystems based on learning with errors. *url: https://www. math. auckland. ac. nz/˜ sgal018/compact-LWE. pdf*, 2013. 178

[Geo73] Alan George. Nested dissection of a regular finite element mesh. *j-SIAM*, 10(2):345–363, April 1973. 33

[GGM84] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. In *Foundations of Computer Science, 1984. 25th Annual Symposium on*, pages 464–479. IEEE, 1984. 143, 145

[GHL01] Martin. Golumbic, Tirza. Hirst, and Moshe Lewenstein. Uniquely restricted matchings. *Algorithmica*, 31(2):139–154, 2001. 4, 41, 47, 49

[Gil52] Edgar N Gilbert. A comparison of signalling alphabets. *Bell System Technical Journal*, 31(3):504–522, 1952. 100

[GKPV10] Shafi Goldwasser, Yael Tauman Kalai, Chris Peikert, and Vinod Vaikuntanathan. Robustness of the learning with errors assumption. In *Innovations in Computer Science - ICS 2010*. Tsinghua University Press, 2010. 123

[GLP12] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 530–547. Springer, 2012. 163

[GNR10] Nicolas Gama, Phong Nguyen, and Oded Regev. Lattice enumeration using extreme pruning. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 257–278. Springer, 2010. 139, 141

[GO95] Anka Gajentaan and Mark Overmars. On a class of $\mathcal{O}(n^2)$ problems in computational geometry. *Computational geometry*, 5(3):165–185, 1995. 73

[GO18] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2018. ix, 177

[Gom58] Ralph Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical society*, 64(5):275–278, 1958. 178

[Gon81] Gaston Gonnet. Expected length of the longest probe sequence in hash code searching. *Journal of the ACM (JACM)*, 28(2):289–304, 1981. 69

[GP88] John Gilbert and Tim Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM Journal on Scientific and Statistical Computing*, 9(5):862–874, 1988. iv, 25, 29, 31

[Ham17] Mike Hamburg. Post-quantum cryptography proposal: Threebears (draft), 2017. 123

[Hås99] Johan Håstad. Clique is hard to approximate withinn 1- ε. *Acta Mathematica*, 182(1):105–142, 1999. 20

[HGS04] Nick Howgrave-Graham and Mike Szydlo. A method to solve cyclotomic norm equations. In *International Algorithmic Number Theory Symposium*, pages 272–279. Springer, 2004. 165

[Hin13] Pieter Hintjens. *ZeroMQ: messaging for many applications*. O'Reilly, Sebastopol, CA, 2013. 110

[HK73] John Hopcroft and Richard Karp. An n^5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973. 19, 51

[HKZ12] Daniel Hsu, Sham Kakade, and Tong Zhang. Tail inequalities for sums of random matrices that depend on the intrinsic dimension. *Electron. Commun. Probab.*, 17:13 pp., 2012. 173

[HM17] Gottfried Herold and Alexander May. Lp solutions of vectorial integer subset sums–cryptanalysis of galbraith's binary matrix lwe. In *IACR International Workshop on Public Key Cryptography*, pages 3–15. Springer, 2017. 123, 178

[HPS11] Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Algorithms for the shortest and closest lattice vector problems. In *International Conference on Coding and Cryptology*, pages 159–190. Springer, 2011. 139, 141

[HPWZ17] Jeffrey Hoffstein, Jill Pipher, William Whyte, and Zhenfei Zhang. A signature scheme from learning with truncation. Technical report, Cryptology ePrint Archive, Report 2017/995, 2017. 123

[HS93] Daniel Hershkowitz and Hans Schneider. Ranks of zero patterns and sign patterns. *Linear and Multilinear Algebra*, 34(1):3–19, 1993. 47, 49

[Hua15] Huacheng Yu. An Improved Combinatorial Algorithm for Boolean Matrix Multiplication. *CoRR*, abs/1505.06811, 2015. 78

[Jou09] Antoine Joux. *Algorithmic cryptanalysis*. CRC Press, 2009. vi, 66, 83, 86, 88

[JV13] Zahra Jafargholi and Emanuele Viola. 3sum, 3xor, triangles. *CoRR*, abs/1305.3827, 2013. 73

[KADF70] Alexander Kronrod, Vladimir Arlazarov, Yefim Dinic, and I Faradzev. On economic construction of the transitive closure of a direct graph. In *Sov. Math (Doklady)*, volume 11, pages 1209–1210, 1970. 78

[KAF+10] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen Lenstra, Emmanuel Thomé, Joppe Bos, Pierrick Gaudry, Alexander Kruppa, Peter Montgomery, Dag Arne Osvik, Herman te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. In *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, pages 333–350, 2010. iv, 29

[Kah60] Jean-Pierre Kahane. Propriétés locales des fonctions à séries de Fourier aléatoires. *Stu. Math.*, 19:1–25, 1960. 131

[Kal95] Erich Kaltofen. Analysis of coppersmith's block wiedemann algorithm for the parallel solution of sparse linear systems. *Mathematics of Computation*, 64(210):777–806, 1995. 27

[Kan83] Ravi Kannan. Improved algorithms for integer programming and related lattice problems. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 193–206. ACM, 1983. 141

[Kan87] Ravi Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of operations research*, 12(3):415–440, 1987. 139

[Kar84] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311. ACM, 1984. 178

[KDL+17] Thorsten Kleinjung, Claus Diem, Arjen Lenstra, Christine Priplata, and Colin Stahlke. Computation of a 768-bit prime field discrete logarithm. In *Advances In Cryptology-Eurocrypt 2017, Pt I*, volume 10210, pages 185–201. Springer International Publishing Ag, 2017. iv

[KF15] Paul Kirchner and Pierre-Alain Fouque. An improved bkw algorithm for lwe with applications to cryptography and lattices. In *Annual Cryptology Conference*, pages 43–62. Springer, 2015. 142

[Kit11] Jarle Kittilsen. Detecting malicious pdf documents. Master's thesis, Gjovik University College, 2011. ix

[KJJR11] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011. 164

[KM70] Victor Klee and George J Minty. How good is the simplex algorithm. Technical report, WASHINGTON UNIV SEATTLE DEPT OF MATHEMATICS, 1970. 178

[Knu98] Donald 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. 74, 76, 80

[KPP16] Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3sum conjecture. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 1272–1287. Society for Industrial and Applied Mathematics, 2016. 73

[KS91] Erich Kaltofen and David Saunders. On wiedemann's method of solving sparse linear systems. In *International Symposium on Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes*, pages 29–38. Springer, 1991. 28

[Lan14] Adeline Langlois. *Lattice - Based Cryptography - Security Foundations and Constructions*. PhD thesis, Informatique Lyon, École normale supérieure, 2014. Thèse de doctorat dirigée par Stehlé, Damien. 136

[LB88] Pil Joong Lee and Ernest Brickell. An observation on the security of McEliece's public-key cryptosystem. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 275–280. Springer, 1988. 100, 101

[LBF08] Gaëtan Leurent, Charles Bouillaguet, and Pierre-Alain Fouque. SIMD Is a Message Digest. Submission to NIST, 2008. 158

[Lep14] Tancrède Lepoint. *Design and implementation of lattice-based cryptography*. PhD thesis, Ecole Normale Supérieure de Paris-ENS Paris, 2014. 135, 164

[LF06] Éric Levieil and Pierre-Alain Fouque. An improved LPN algorithm. In *Security and Cryptography for Networks, 5th International Conference, SCN 2006, Maiori, Italy, September 6-8, 2006, Proceedings*, pages 348–359, 2006. 73

[LG14] François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th international symposium on symbolic and algebraic computation*, pages 296–303. ACM, 2014. iii

[Lin08] The LinBox Group. *LinBox – Exact Linear Algebra over the Integers and Finite Rings, Version 1.1.6*, 2008. iv, 9

[Liu17] Dongxi Liu. Compact-lwe for lightweight public key encryption and leveled iot authentication. In *ACISP*, volume 17, 2017. 123

[LLKN17] Dongxi Liu, Nan Li, Jongkil Kim, and Surya Nepal. Compact-lwe: Enabling practically lightweight public key encryption for leveled iot device authentication, 2017. 123

[LLL82] Arjen Lenstra, Hendrik Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982. 141

[LLPX18] Haoyu Li, Renzhang Liu, Yanbin Pan, and Tianyuan Xie. Ciphertext-only attacks against compact-lwe submitted to nist pqc project. Cryptology ePrint Archive, Report 2018/020, 2018. `https://eprint.iacr.org/2018/020`. 123

[LLS14] Fabien Laguillaumie, Adeline Langlois, and Damien Stehlé. Chiffrement avancé à partir du problème learning with errors, 2014. 135

[LM06] Vadim Lyubashevsky and Daniele Micciancio. Generalized compact knapsacks are collision resistant. In *International Colloquium on Automata, Languages, and Programming*, pages 144–155. Springer, 2006. 138

[LMPR08] Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. Swifft: A modest proposal for fft hashing. In *International Workshop on Fast Software Encryption*, pages 54–72. Springer, 2008. 144, 147, 158

[LO90] Brian LaMacchia and Andrew Odlyzko. Solving large sparse linear systems over finite fields. In *Advances in Cryptology - CRYPTO '90, 10th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1990, Proceedings*, pages 109–133, 1990. 36

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010. 123, 135, 136

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM (JACM)*, 60(6):43, 2013. 123

[LPSS17] San Ling, Duong Hieu Phan, Damien Stehlé, and Ron Steinfeld. Hardness of k-lwe and applications in traitor tracing. *Algorithmica*, 79(4):1318–1352, 2017. 123

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015. 123, 136

[Lyu08] Vadim Lyubashevsky. Lattice-based identification schemes secure under active attacks. In *International Workshop on Public Key Cryptography*, pages 162–179. Springer, 2008. 162

[Lyu09] Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 598–616. Springer, 2009. 125, 161

[Lyu12] Vadim Lyubashevsky. Lattice signatures without trapdoors. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 738–755. Springer, 2012. 162

[Mar57] Harry Markowitz. The elimination form of the inverse and its application to linear programming. *Manage. Sci.*, 3(3):255–269, April 1957. 33, 36

[McE78] Robert J McEliece. A public-key cryptosystem based on algebraic. *Coding Theory*, 4244:114–116, 1978. 100

[MGC12] Davide Maiorca, Giorgio Giacinto, and Ignio Corona. *A Pattern Recognition System for Malicious PDF Files Detection,*, pages 510–524. Springer Berlin Heidelberg, 2012. ix

[Mit96] Michael David Mitzenmacher. *The power of two random choices in randomized load balancing*. PhD thesis, PhD thesis, Graduate Division of the University of California at Berkley, 1996. vi, 68, 69, 84

[Mit98] John E Mitchell. Branch-and-cut algorithms for integer programming, 1998. 178

[MMT11] Alexander May, Alexander Meurer, and Enrico Thomae. Decoding Random Linear Codes in $\tilde{\mathcal{O}}(2^{0.054n})$. In *EUROCRYPT*, pages 107–124. Springer, 2011. 100, 101

[MO15] Alexander May and Ilya Ozerov. On computing nearest neighbors with applications to decoding of binary linear codes. In *EUROCRYPT*, pages 203–228, 2015. 100, 101

[MP13] Daniele Micciancio and Chris Peikert. Hardness of sis and lwe with small parameters. In *Advances in Cryptology–CRYPTO 2013*, pages 21–39. Springer, 2013. 123

[MR04] Daniele Micciancio and Oded Regev. Worst-case to average-case reductions based on gaussian measures. In *FOCS, IEEE Computer Society Press*, pages 372–381. IEEE, 2004. 169

[MR07] Daniele Micciancio and Oded Regev. Worst-case to average-case reductions based on gaussian measures. *SIAM Journal on Computing*, 37(1):267–302, 2007. 138

[MS12] Lorenz Minder and Alistair Sinclair. The extended k-tree algorithm. *Journal of cryptology*, 25(2):349–382, 2012. 65

[Nan14] Mridul Nandi. Xls is not a strong pseudorandom permutation. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 478–490. Springer, 2014. 71

[Nan15] Mridul Nandi. Revisiting Security Claims of XLS and COPA. *IACR Cryptology ePrint Archive*, 2015:444, 2015. iii, 65, 71

[NS15] Ivica Nikolić and Yu Sasaki. Refinements of the k-tree Algorithm for the Generalized Birthday Problem. In *ASIACRYPT*, pages 683–703. Springer, 2015. vi, 66, 73, 83, 89, 95

[Pat10] Mihai Patrascu. Towards polynomial lower bounds for dynamic problems. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 603–610, 2010. 73

[PBY17] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To bliss-b or not to be: Attacking strongswan's implementation of post-quantum signatures. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1843–1855. ACM, 2017. 125

[POG15] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers. In *International Conference on Cryptology and Information Security in Latin America*, pages 346–365. Springer, 2015. 164

[PR01] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *European Symposium on Algorithms*, pages 121–133. Springer, 2001. 74

[PR06] Chris Peikert and Alon Rosen. Efficient collision-resistant hashing from worst-case assumptions on cyclic lattices. In *Theory of Cryptography Conference*, pages 145–166. Springer, 2006. 138

[Pra62] Eugene Prange. The use of information sets in decoding cyclic codes. *IRE Transactions on Information Theory*, 8(5):5–9, 1962. 100

[PVZ17] Grigoris Paouris, Petros Valettas, and Joel Zinn. Random version of Dvoretzky's theorem in ℓ_p^n. *Stochastic Processes and their Applications*, 127(10):3187–3227, 2017. 177

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *STOC*, pages 84–93. ACM, 2005. vii, 123, 134, 138

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):34, 2009. 135

[Rot06] Ron Roth. *Introduction to Coding Theory*. Cambridge University Press, New York, NY, USA, 2006. 100

[RSA78] Ronald Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. 3

[RSSS17] Miruna Roşca, Amin Sakzad, Damien Stehlé, and Ron Steinfeld. Middle-product learning with errors. In *Annual International Cryptology Conference*, pages 283–297. Springer, 2017. 123, 136

[RSW18] Miruna Rosca, Damien Stehlé, and Alexandre Wallet. On the ring-lwe and polynomial-lwe problems. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 146–173. Springer, 2018. 123, 136

[RT75] Donald Rose and Endre Tarjan. Algorithmic aspects of vertex elimination. In *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*, STOC '75, pages 245–254, New York, NY, USA, 1975. ACM. 4, 32

[S+17] Andreas Steffen et al. strongSwan: the open source IPsec-based VPN solution (version 5.5.2), March 2017. 164

[Sag13] The Sage Developers. *Sage Mathematics Software (Version 5.7)*, 2013. `http://www.sagemath.org`. 36

[Sch89] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *Conference on the Theory and Application of Cryptology*, pages 239–252. Springer, 1989. 161

[Sch13] Jens Schmidt. A simple test on 2-vertex- and 2-edge-connectivity. *Information Processing Letters*, 113(7):241 – 244, 2013. 49

[SE94] Claus-Peter Schnorr and Martin Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical programming*, 66(1-3):181–199, 1994. 141

[Sho99] Peter Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999. 165

[Sho09] Victor Shoup. *A computational introduction to number theory and algebra*. Cambridge university press, 2009. 68

[SS81] Richard Schroeppel and Adi Shamir. A $T = \mathcal{O}(2^{n/2})$, $S = \mathcal{O}(2^{n/4})$ Algorithm for Certain NP-Complete Problems. *SIAM journal on Computing*, 10(3):456–464, 1981. 65

[SSTX09] Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. Efficient public key encryption based on ideal lattices. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 617–635. Springer, 2009. 123, 136

[Sta84] Wolfgang Stadje. An inequality for ℓ_p-norms with respect to the multivariate normal distribution. *Journal of Mathematical Analysis and Applications*, 102(1):149 – 155, 1984. 177

[Ste88] Jacques Stern. A method for finding codewords of small weight. In *Coding Theory and Applications*, pages 106–113. Springer, 1988. 100, 101

[Ste06] Didier Stevens. Didier Stevens Blog. `https://blog.didierstevens.com/`, 2006. ix

[SY09] David Saunders and Bryan Youse. Large matrix, small rank. In *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation*, ISSAC '09, pages 317–324, New York, NY, USA, 2009. ACM. 40

[TIR78] Steven Tanimoto, Alon Itai, and Michael Rodeh. Some matching problems for bipartite graphs. *J. ACM*, 25(4):517–525, October 1978. 49

[Tro12] Joel Tropp. User-friendly tail bounds for sums of random matrices. *Foundations of Computational Mathematics*, 12(4):389–434, Aug 2012. 173

[Var57] Rom Varshamov. Estimate of the number of signals in error correcting codes. *Dokl. Akad. Nauk SSSR*, 117(5):739–741, 1957. 100

[Vau05] Serge Vaudenay. *A Classical Introduction to Cryptography: Applications for Communications Security*. Springer-Verlag, Berlin, Heidelberg, 2005. 69

[vH14] Ramon van Handel. Probability in high dimension. Technical report, Princeton University, 2014. 177

[Vio12] Emanuele Viola. Reducing 3xor to listing triangles, an exposition. Technical report, Northeastern University, College of Computer and Information Science, May 2012. Available at `http://www.ccs.neu.edu/home/viola/papers/xxx.pdf`. 73

[vN51] John von Neumann. 13. various techniques used in connection with random digits. *Appl. Math Ser*, 12(36-38):5, 1951. 130

[VV14] Gregory Valiant and Paul Valiant. An automatic inequality prover and instance optimal identity testing. In *Proceedings of the 2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 51–60. IEEE Computer Society, 2014. 130

[Wag02] David Wagner. A generalized birthday problem. In *CRYPTO*, pages 288–304, 2002. vi, 65, 73

[Wie86] Douglas Wiedemann. Solving sparse linear equations over finite fields. *IEEE Trans. Information Theory*, 32(1):54–62, 1986. 7, 27

[XY18] Dianyan Xiao and Yang Yu. Cryptanalysis of compact-lwe and related lightweight public key encryption. *Security and Communication Networks*, 2018, 2018. 123

[Yan81] Mihalis Yannakakis. Computing the Minimum Fill-In is NP-Complete. *SIAM Journal on Algebraic Discrete Methods*, 2(1):77–79, 1981. v, 4, 32

[ZJW16] Bin Zhang, Lin Jiao, and Mingsheng Wang. Faster algorithms for solving lpn. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 168–195. Springer, 2016. 73

List of Figures

List of Tables

List of Algorithms

Resumé

Dans cette thèse, nous discutons d'aspects algorithmiques de trois différents problèmes, en lien avec la cryptographie.

La première partie est consacrée à l'algèbre linéaire creuse. Nous y présentons un nouvel algorithme de pivot de Gauss pour matrices creuses à coefficients exacts, ainsi qu'une nouvelle heuristique de selection de pivots, qui rend l'entière procédure particulièrement efficace dans certains cas.

La deuxième partie porte sur une variante du problème des anniversaires, avec trois listes. Ce problème, que nous appelons problème 3XOR, consiste intuitivement à trouver trois chaînes de caractères uniformément aléatoires de longueur fixée, telles que leur XOR soit la chaîne nulle. Nous discutons des considérations pratiques qui émanent de ce problème et proposons un nouvel algorithme plus rapide à la fois en théorie et en pratique que les précédents.

La troisième partie est en lien avec le problème *learning with errors* (LWE). Ce problème est connu pour être l'un des principaux problèmes difficiles sur lesquels repose la cryptographie à base de réseaux euclidiens. Nous introduisons d'abord un générateur pseudo-aléatoire, basé sur la variante dé-randomisé *learning with rounding* de LWE, dont le temps d'évaluation est comparable avec celui d'AES. Dans un second temps, nous présentons une variante de LWE sur l'anneau des entiers. Nous montrons que dans ce cas le problème est facile à résoudre et nous proposons une application intéressante en re-visitant une attaque par canaux auxiliaires contre le schéma de signature BLISS.

Abstract

In this thesis, we discuss algorithmic aspects of three different problems, related to cryptography.

The first part is devoted to sparse linear algebra. We present a new Gaussian elimination algorithm for sparse matrices whose coefficients are exact, along with a new pivots selection heuristic, which make the whole procedure particularly efficient in some cases.

The second part treats with a variant of the Birthday Problem with three lists. This problem, which we call 3XOR problem, intuitively consists in finding three uniformly random bit-strings of fixed length, such that their XOR is the zero string. We discuss practical considerations arising from this problem, and propose a new algorithm which is faster in theory as well as in practice than previous ones.

The third part is related to the *learning with errors* (LWE) problem. This problem is known for being one of the main hard problems on which lattice-based cryptography relies. We first introduce a pseudorandom generator, based on the de-randomised *learning with rounding* variant of LWE, whose running time is competitive with AES. Second, we present a variant of LWE over the ring of integers. We show that in this case the problem is easier to solve, and we propose an interesting application, revisiting a side-channel attack against the BLISS signature scheme.