Embedded lattice-based cryptography

Simon Montoya

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2022IPPAX089

Thèse de doctorat

Embedded lattice-based cryptography

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à École polytechnique

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat: Informatique, données, IA

Thèse présentée et soutenue à Palaiseau, le 12/10/2022, par

SIMON MONTOYA

Composition du Jury :

Alain Couvreur Directeur de recherche, Centre de recherche INRIA Saclay	Président
Louis Goubin Professeur des universités, Université de Versailles-St-Quentin-en-Yvelines	Rapporteur
Laurent Imbert Directeur de recherche, LIRMM	Rapporteur
Cécile Dumas Ingénieure de recherche, CEA de Grenoble	Examinatrice
Francois-Xavier Standaert Professeur, Université catholique de Louvain	Examinateur
Alexandre Wallet Chargé de recherche, INRIA Rennes	Examinateur
Guénaël Renault Professeur chargé de cours, École polytechnique, Agence Nationale de la Sécurité des Systèmes d'Information	Directeur de thèse
Aurélien Greuet Docteur, IDEMIA France	Co-encadrant

Remerciements

Je souhaite commencer ce manuscrit en remerciant Aurélien et Guénaël sans qui ce travail n'aurait pas été possible. Pendant ces trois années vous avez été à l'écoute, vous m'avez conseillé et beaucoup appris. Malgré les nombreuses relectures que je vous ai demandées, les nombreux rebondissements durant la thèse, le Covid et bien d'autres aventures, vous avez toujours été là pour m'aider et me soutenir. Sans vous ces lignes et toutes les suivantes n'auraient pas existé. Pour tout cela, Aurélien, Guénaël, je vous dis mille mercis.

Je tiens sincèrement à remercier Louis Goubin et Laurent Imbert d'avoir accepté d'être rapporteurs pour ma thèse. Mes remerciements vont également vers Alain Couvreur, Cécile Dumas, François-Xavier Standaert et Alexandre Wallet qui ont accepté d'être examinateurs. Je remercie également Martin R. Albrecht et renouvelle mes remerciements à Alexandre Wallet de m'avoir relu et conseillé durant la soutenance de mi-parcours.

Durant ces trois années, j'ai pu côtoyer deux superbes équipes. La première est l'équipe crypto d'IDEMIA. J'ai énormément appris en votre compagnie et apprécié travailler avec chacun d'entre vous. J'ai passé de formidables moments. Pour cela, Amaury, Aurélien, Clémence, Franck, Julien, Linda, Luk, Matthias, Nathan, Rina, Roch et Stéphane je vous dis merci. Je remercie IDEMIA d'avoir financé cette thèse et je souhaite bon courage aux (futurs) doctorants.

La seconde équipe que j'ai eu la chance de côtoyer est l'équipe GRACE du LIX. Malgré le fait que j'ai été peu présent, venir vous voir a toujours été un grand plaisir. J'ai passé de très bons moments et pu avoir des conversations très enrichissantes avec vous, merci.

Je tiens également à remercier l'ensemble de mes co-auteurs de l'ANSSI et de l'Université du Luxembourg. Travailler avec vous a été un plaisir.

Depuis de longues années, je peux compter sur le soutien indéfectible de mes amis. Que ce soit mes "bros" de Tours, mes amis de Bordeaux, de Rennes ou encore de Paris. J'ai passé de nombreuses soirées, vacances et bien d'autres moments incroyables avec vous. Merci pour votre amitié et d'être toujours à mes côtés.

Je tiens spécialement à remercier Maxime pour ces années de collocation. Bien qu'il y ait eu quelques (nombreux) confinements, j'ai passé d'agréables moments en ta compagnie.

Je ne peux finir ces remerciements sans une pensée pour ma famille. Depuis toujours, je peux compter sur leur soutien permanent. Je tiens à vous remercier profondément. Enfin, je souhaite remercier Anne-Sophie d'être là pour moi. Ta bonne humeur et ton soutien au quotidien me sont d'une grande aide. Tout l'amour que tu m'apportes m'a aidé à réaliser cette thèse.

Contents

Chapter 1

Introduction

La sécurité de la cryptographie à clé publique repose sur la difficulté de résoudre certains problèmes mathématiques. Parmi les plus connus, nous avons le problème de factorisation et du logarithme discret. La difficulté de ces deux problèmes assure la sécurité de schémas cryptographiques tels que RSA ou ceux basés sur les courbes elliptiques. En dépit des progrès dans le domaine arithmétique et informatique, ces problèmes garantissent toujours un haut niveau de sécurité. Cependant, dans les années à venir, l'avènement d'un ordinateur quantique suffisamment puissant pourrait changer la donne. En effet, l'algorithme quantique de Shor [Sho97] permet de résoudre le problème de la factorisation et du logarithme discret avec une complexité polynomiale. Ainsi lorsqu'un tel ordinateur verra le jour, les cryptosystèmes asymétriques actuellement déployés ne garantiront plus un niveau de sécurité suffisant. Néanmoins, un tel ordinateur prendra plusieurs années à être construit. En effet, les meilleurs ordinateurs quantiques actuels manipulent quelques dizaines de qubits alors que l'on estime que plusieurs milliers sont nécessaires pour pouvoir casser les cryptosystèmes actuels [BSI20b]. En réponse à cette problématique, la communauté internationale élabore des systèmes cryptographiques résistants à un attaquant disposant de la puissance de calcul quantique: la cryptographie *post-quantique*. La sécurité de ces nouveaux schémas est assurée par la difficulté de résoudre des problèmes mathématiques définis sur les réseaux euclidiens, les codes correcteurs d'erreurs, etc. A l'heure actuelle, ces problèmes ne sont pas menacés par un algorithme quantique qui pourrait les résoudre en temps polynomial.

Les agences gouvernementales ont commencé à étudier les schémas post-quantiques (e.g. [ANS; BSI; Moo16; fCry18]) et ont initié des processus de standardisation pour ces schémas [Moo16; fCry18]. La standardisation la plus suivie par la communauté est celle du National Institute of Standards and Technology (NIST), qui a été lancée en 2016 [Moo16]. L'objectif de cette standardisation est de réunir la communauté scientifique afin de déterminer les futurs standards pour les *Key Encapsulation Mechanisms* (KEMs) et les signatures. Depuis Juillet 2020, la standardisation du NIST en est à son troisième tour de sélection avec sept finalistes restants [MAA+20]. Au sein de ces finalistes, cinq basent leur sécurité sur les réseaux euclidiens. L'engouement autour des schémas basés sur les réseaux euclidiens est du à leur bon compromis entre efficacité, sécurité et taille des clés. Cependant malgré les bonnes caractéristiques, ces schémas et de manière générale les schémas post-quantiques, sont moins performants et/ou plus coûteux en mémoire que les schémas actuels.

Les futurs schémas standardisés seront déployés dans de nombreux cas d'usages et de nombreux composants. Parmi eux, il y aura les composants embarqués. Ces composants sont utilisés dans de nombreux objets du quotidien: carte bancaire, passeport, IoT, etc. Cependant ces composants sont limités en terme de RAM et de puissance de calcul. En effet, la fréquence de leur processeur se compte en méga hertz et leur capacité de RAM en kilo octets alors que sur un ordinateur ces valeurs se comptent en giga. De part leur faible puissance de calcul, ces composants embarquent généralement des coprocesseurs supplémentaires conçus pour réaliser les opérations coûteuses de la cryptographie. De plus, les composants embarqués sont exposés aux attaques physiques. Lors d'une attaque physique, un attaquant peut perturber le bon fonc-

tionnement de la puce ou sonder des valeurs physiques (la consommation de courant, le temps d'exécution, etc.) afin d'apprendre de l'information sur les secrets manipulés. Ainsi dans certains contextes, les implémentations embarquées doivent être sécurisées contre ce type d'attaque. Pour toutes ces raisons, déployer la cryptographie post-quantique sur les composants embarqués est un véritable défi.

1.1 Problématiques

De part les limitations et les contraintes de sécurité des composants embarqués, deux axes majeurs de recherche sont nécessaires pour le déploiement de la cryptographie post-quantique: l'optimisation de ces cryptosystèmes et la sécurisation contre les attaques physiques. Nous pouvons décliner ces deux axes en trois problématiques:

- Optimiser les cryptosystèmes post-quantiques pour les composants embarqués.
  - Comment optimiser les cryptosystèmes post-quantiques avec les instructions CPU ?
  - Comment réutiliser les coprocesseurs actuels pour optimiser les schémas post-quantiques ?
  - Quelle architecture matérielle choisir pour les futurs coprocesseurs post-quantiques ?
- Sécuriser les implémentations post-quantiques contre les attaques physiques.
- Évaluer la résilience des implémentations post-quantiques contre les attaques physiques.

Dans cette thèse nous apportons des réponses à ces problématiques pour les cryptosystèmes post-quantiques basés sur les réseaux euclidiens. Plus particulièrement, nous étudions les KEMs et les signatures basés sur les réseaux euclidiens proposés lors de la standardisation du NIST [Moo16]. Nous nous sommes focalisés sur les schémas basés sur les réseaux euclidiens car, comme dit précédemment, parmi les primitives post-quantiques, ces cryptosystèmes présentent le meilleur compromis entre temps d'exécution, consommation RAM et sécurité. Ainsi, pour des environnements restreints, les schémas basés sur les réseaux euclidiens apparaissent comme des candidats prometteurs et donc à étudier en premier lieu.

Une première partie de cette thèse est consacrée à l'optimisation de ces schémas à l'aide de coprocesseurs dédiés. Plus précisément, nous modifions l'utilisation des coprocesseurs asymétriques existants pour optimiser ces schémas. La réutilisation de ces coprocesseurs permet d'obtenir des optimisations applicables à de nombreux composants embarqués. La deuxième partie de cette thèse s'intéresse à la sécurité de ces schémas contre les attaques physiques. En particulier, nous déterminons la résilience de plusieurs schémas contre plusieurs types d'attaques physiques (canaux auxiliaires, injection de faute). De plus, nous proposons des contremesures pour se prémunir de certaines de ces attaques.

1.2 Algorithmique des cryptosystèmes basés sur les réseaux euclidiens

La première partie de cette thèse s'intéresse à l'optimisation, sur les composants embarqués, des cryptosystèmes basés sur les réseaux euclidiens. Pour cela, dans un premier temps, nous décrivons à haut niveau la structure de ces schémas post-quantiques.

1.2.1 Problème LWE et variantes

L'un des problèmes fondateurs utilisés pour assurer la sécurité des schémas basés sur les réseaux euclidiens est le problème Learning With Errors (LWE). Ce problème a été introduit par O. Regev dans l'article [Reg09]. Soient $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$, $(a_i, s) \in \mathbb{Z}_q^{n \times n}$, $e_i \in \mathbb{Z}_q$, où a_i est tiré uniformément dans $\mathbb{Z}_q^n$ et chaque coefficients de s_i et e_i sont tirés selon une certaine distribution sur $\mathbb{Z}_q$. Soit $b_i \in \mathbb{Z}_q$ tel que $b_i =< a_i, s > +e_i$. Le problème de recherche LWE assure qu'il est difficile de retrouver s en connaissant plusieurs échantillons

(a_i,b_i). Initialement ce problème est utilisé pour construire des systèmes de chiffrement. La limitation principale de ces schémas est l'utilisation de vecteurs de taille n pour chiffrer 1 bit de message. Ainsi si nous voulons envoyer plusieurs bits de message, le système de chiffrement requiert des calculs matrice-vecteur, ce qui est assez coûteux.

Par la suite, des variantes structurées du problème LWE ont été proposées afin de rendre les schémas plus compacts et plus efficaces. Les objets mathématiques manipulés sont alors des anneaux ou des modules de polynômes sur un corps fini [LPR13; LS15a]. Dans ce cas, nous parlons du problème Ring-LWE (RLWE) ou Module-LWE (MLWE). Par exemple, dans la compétition du NIST, l'anneau de polynômes le plus utilisé est $R_{q,1} = \mathbb{Z}_q[X]/(X^N + 1)$ où $q \geq 2, N \geq 1$. Plus généralement, la majorité des schémas basés sur les réseaux euclidiens du NIST sont basés sur les problèmes RLWE/MLWE ou sur une variante de ces problèmes: RLWR/MLWR. Le problème LWR (Learning With Rounding) est une variante du problème LWE où les vecteurs d'erreurs e_i sont remplacés par des arrondis [BPR12].

Les schémas que nous étudions dans la suite de cette thèse ont ainsi leurs vecteurs définis dans des anneaux/modules de polynômes sur un corps fini. A haut niveau, un système de chiffrement utilisant le problème RLWE peut être décrit par les algorithmes 1, 2, 3.

Algorithm 1 KEY GEN RLWE

Output: Key pair (p_k, s_k)

1: $a \leftarrow$ aléatoire uniforme dans $R_{q,1}$
2: $s, e \leftarrow$ aléatoire avec une distrib. χ dans $R_{q,1}$
3: $b \leftarrow a \cdot s + e \in R_{q,1}$
4: **return** $(p_k, s_k) = ((a, b), s)$

Algorithm 2 ENCRYPT RLWE$(p_k, m \in \{0,1\}^N)$

Output: Ciphertext $c = (c_1, c_2)$

1: $r, e_1, e_2 \leftarrow$ aléatoire avec une distrib. χ dans $R_{q,1}$
2: $c_1 \leftarrow ar + e_1 \in R_{q,1}$
3: $c_2 \leftarrow br + e_2 + \lfloor \frac{q}{2} \rceil m \in R_{q,1}$
4: **return** $c = (c_1, c_2)$

Algorithm 3 DECRYPT RLWE$(s_k = s, c = (c_1, c_2))$

Output: Plaintext m

1: $M \leftarrow c_2 - (c_1 \cdot s) \in R_{q,1}$
2: **for** $i = 0$ to $N - 1$ **do**
3: **if** $\frac{q}{4} \leq M_i < \frac{3q}{4}$ **then** $m_i \leftarrow 1$ **else** $m_i \leftarrow 0$
4: **end for**
5: **return** m

Pour ces trois routines l'arithmétique utilisée est une arithmétique modulaire de polynômes. Les opérations coûteuses sont:

- La génération d'aléa.
- La multiplication de polynômes.

De manière générale, les signatures et les KEMs de la standardisation du NIST utilisent de l'arithmétique modulaire de polynômes. De plus, les opérations les plus coûteuses sont les mêmes que celles dans l'exemple précédent.

1.2.2 La génération d'aléa

Dans les schémas de signature et les KEMs du NIST, les polynômes sont générés à partir d'une `seed` et d'une eXtendable-Output Function (XOF) basée sur Keccak [BDP+15] (SHAKE128 ou SHAKE256). Une XOF est une fonction cryptographique de hash qui peut retourner une sortie de taille arbitraire. La XOF est alors utilisée comme un générateur de nombres pseudo-aléatoires.

Dans les schémas basés sur les réseaux euclidiens, la génération d'aléa a un impact significatif sur les performances. Les opérations effectuées dans Keccak sont des opérations bit à bit ou des permutations. Ainsi, les optimisations possibles sont limitées. Une solution pour accélérer la génération d'aléa est d'embarquer un coprocesseur Keccak dédié.

1.2.3 Arithmétique modulaire de polynômes

Comme décrit plus haut, les schémas basés sur les réseaux euclidiens utilisent de l'arithmétique modulaire de polynômes. Les principales opérations sont l'addition, la soustraction et la multiplication de polynômes. Cette dernière est l'opération arithmétique la plus coûteuse. En effet, naïvement, cette opération a une complexité asymptotique en $\mathcal{O}(N^2)$ alors que l'addition et la soustraction ont une complexité en $\mathcal{O}(N)$, où N est le degré des polynômes.

Parmi les cryptosystèmes finalistes de la standardisation du NIST, certains spécifient l'utilisation d'algorithmes de multiplications polynomiales avec une meilleure complexité asymptotique que la méthode naïve. Ainsi certains cryptosystèmes utilisent dans leurs implémentations de référence l'algorithme de Karatsuba, Toom-Cook ou encore la Number Theoretic Transform (NTT). Ce dernier possède la meilleure complexité asymptotique qui est en $\mathcal{O}(N \log N)$. Cependant, la complexité asymptotique ne reflète pas nécessairement la réalité pratique. En effet, les paramètres des cryptosystèmes ne sont pas suffisamment grands pour avoir des gains asymptotiques significatifs. De plus, en fonction du CPU, certains algorithmes de multiplication peuvent être plus ou moins efficaces. Dans les systèmes embarqués, trois axes de recherche se distinguent pour optimiser la multiplication polynomiale:

- Optimiser les algorithmes existants avec des instructions CPU spécifiques. Ces optimisations sont destinées à des composants avec un jeu d'instructions CPU conséquent et rapide. Par exemple, celui des CPU cortex-M3 et cortex-M4.
- Concevoir un coprocesseur dédié à l'arithmétique polynomiale. Les coprocesseurs conçus sont destinés à de futurs composants et non ceux actuels.
- Ré-utiliser les coprocesseurs asymétriques RSA/ECC existants. Ces coprocesseurs effectuent de l'arithmétique sur les entiers. Cette réutilisation permet d'obtenir des optimisations pour des composants possédant un faible CPU. Dans la suite nous détaillons cet axe de recherche.

Dans cette thèse, nous nous intéressons à la réutilisation des coprocesseurs asymétriques existants afin d'optimiser la cryptographie basée sur les réseaux euclidiens. Nous avons fait ce choix car, comme l'ont indiqué plusieurs agences gouvernementales (ANSSI [ANS22], BSI [BSI20a]), les premiers déploiements de la cryptographie post-quantique se feront de manière hybride. La cryptographie hybride consiste en une combinaison de la cryptographie post-quantique et classique. Ainsi les communications sont à la fois sécurisées:

- Contre les attaques quantiques grâce à la cryptographie post-quantique.
- Contre les attaques classiques, avec un niveau de sécurité au moins égal à celui apporté par la cryptographie actuelle.

Le maintien de la cryptographie actuelle est nécessaire de par la jeunesse de la cryptographie post-quantique. En effet, nous ne disposons pas d'assez de recul sur la cryptographie post-quantique pour être sûrs de sa sécurité à long terme. Ainsi, en la combinant avec de la cryptographie classique, nous nous assurons de la non-régression du niveau de sécurité actuel.

En réutilisant les coprocesseurs existants, nous embarquons un seul coprocesseur asymétrique pour optimiser ces deux types de cryptographie. Ceci a un intérêt en terme de coût, de facilité de déploiement et afin de proposer des optimisations qui s'appliquent à une large gamme de composants. La réutilisation des

coprocesseurs asymétriques existants, pour la cryptographie basée sur les réseaux euclidiens, a été initiée par Albrecht *et al.* dans l'article [AHH+19]. Dans ce travail, les auteurs implémentent le KEM Kyber [BDK+18] sur un composant embarqué ayant un coprocesseur asymétrique qui effectue de l'arithmétique sur les entiers. Afin d'optimiser la multiplication polynomiale, ils utilisent la substitution introduite par Kronecker [Kro82] qui permet de transformer une multiplication de polynômes en une multiplication d'entiers. Pour ce faire cette substitution évalue les polynômes en un point afin d'obtenir des entiers. Par la suite ces entiers sont multipliés. Enfin, le résultat de cette multiplication est converti en un polynôme. Afin de gagner en performance, la multiplication d'entiers est effectuée avec le coprocesseur asymétrique. Dans l'article [WGY20], les auteurs utilisent également la substitution de Kronecker avec un coprocesseur RSA/ECC pour améliorer les performances du schéma Saber [BMD+21].

Par la suite dans l'article [BRvV22], les auteurs généralisent la substitution de Kronecker utilisée par Albrecht *et al.* [AHH+19]. Cette généralisation permet des compromis entre le nombre de multiplications d'entiers, la taille des entiers à multiplier et le nombre d'évaluations polynomiales. En fonction des spécifications du composant et du coprocesseur, cette généralisation permet d'obtenir des compromis permettant d'avoir une multiplication polynomiale plus performante.

1.2.4 Contributions sur les optimisations des cryptosystèmes à l'aide des coprocesseurs asymétriques existants

Dans cette thèse nous nous sommes concentrés sur l'optimisation des cryptosystèmes basés sur les réseaux euclidiens, à l'aide des coprocesseurs asymétriques (RSA/ECC) actuels. Plus précisément, notre objectif est d'optimiser la multiplication modulaire de polynômes à l'aide de ces coprocesseurs. Dans la majorité des schémas basés sur les réseaux euclidiens, cette multiplication modulaire se fait sur l'anneau de polynômes $R_{q,\delta} = \mathbb{Z}_q[X]/(X^N + \delta)$, où $\delta \in \{-1, 1\}$. Comme dit précédemment, afin de réutiliser les coprocesseurs existants pour la multiplication de polynômes, nous utilisons la *substitution de Kronecker*. En utilisant cette substitution, la multiplication modulaire sur $R_{q,\delta}$ se divise en quatre étapes:

1. Évaluation des polynômes en un point d'évaluation bien choisi. Cette étape permet de transformer les polynômes en entiers.

2. Multiplication modulaire des entiers obtenus. Dans les travaux précédents, uniquement cette étape est effectuée en réutilisant les coprocesseurs existants.

3. Conversion du résultat de la multiplication d'entiers en un polynôme. L'étape 2 assure que le résultat est réduit mod $X^N + \delta$. Dans la suite nous nommons cette étape la *radix conversion*.

4. Réduction des coefficients modulo q.

Nos contributions ont pour objectif d'améliorer l'ensemble de ces étapes à l'aide des coprocesseurs asymétriques actuels. Dans la suite nous les présentons en trois parties.

Résumé des contributions

Contribution 1: Alternative à la multiplication d'entier lors de la substitution de Kronecker. Dans ce travail nous optimisons la multiplication polynomiale de certains KEMs du NIST à l'aide des coprocesseurs actuels. Plus particulièrement, nous introduisons deux variantes à la substitution de Kronecker: *Kronecker substitution variant* et *Shift & Add*. Ces variantes exploitent la structure particulière des polynômes. Soient $f(X), g(X) \in R_{q,1}$ deux polynômes que l'on veut multiplier. Dans les KEMs du NIST, $f(X)$ a des coefficients uniformément distribués dans $\{0, \ldots, q-1\}$ et $g(X)$ a des coefficients proches de 0. Les variantes exploitent le fait que $g(X)$ a des coefficients proches de 0.

Substitution de Kronecker:

1. Évaluation des polynômes $f(X)$ et $g(X)$.
2. Multiplication d'entiers entre les polynômes évalués.
3. Radix conversion.

Variantes:

1. Évaluation du polynôme $f(X)$. On obtient alors un entier f'.
2. Opérations sur f' dépendantes des coefficients de $g(X)$. Les opérations sont des shifts, additions, soustractions et des multiplications qui sont réalisées avec le coprocesseur asymétrique.
3. Radix conversion.

Les opérations effectuées sur f' diffèrent d'une variante à l'autre. En fonction des spécifications du composant, ces deux variantes peuvent être plus rapide que la substitution de Kronecker.

Dans ce travail nous proposons également une méthodologie pour qu'un développeur puisse choisir rapidement entre la substitution de Kronecker et les variantes. Pour cela nous donnons, à chacun de ces trois algorithmes, une complexité théorique en fonction d'opérations basiques telles que des additions, shifts et multiplications. Ainsi un développeur, en regardant le coût de chacune de ces opérations dans la spécification du coprocesseur, peut choisir rapidement quel algorithme utiliser.

Par la suite nous évaluons nos variantes, pour plusieurs paramètres de sécurité proposés lors de la compétitions du NIST, sur un composant embarqué. Nous montrons alors que nos complexités théoriques sont en accords avec les évaluations pratiques. De plus sur ce composant, au moins l'une de nos variantes est plus efficace que la substitution de Kronecker.

Contribution 2: Réduction Modulaire Quotient Approximation. La réduction modulaire est une opération cruciale pour beaucoup de cryptosystèmes, dont les schémas basés sur les réseaux euclidiens. En effet, les polynômes sont définis sur $R_{q,\delta}$, donc, après chaque opération arithmétique, nous devons réduire l'ensemble des coefficients modulo q. Soient $a, q \in \mathbb{N}$, tels que nous voulons calculer $a \mod q$. Pour cela, nous devons déterminer la valeur $\texttt{quo} = \left\lfloor \frac{a}{q} \right\rfloor$. Ainsi, nous obtenons $a \mod q = a - \texttt{quo} \cdot q$. Calculer quo nécessite une division, ce qui est coûteux.

La réduction modulaire *Quotient Approximation Reduction* détermine une approximation de quo, nommée $\texttt{quo}_{approx}$, en calculant une somme de shifts de a. A l'aide de cette approximation, nous obtenons une réduction partielle $a - \texttt{quo}_{approx} \cdot q = a \mod q + t \cdot q$. Nous prouvons que $t \leq \sum_{i=0}^{\ell-1} \left\{ \frac{2^i}{q} \right\}$ où ℓ est la taille en bits de a. Grâce à un algorithme standard de division, nous pouvons alors obtenir une réduction complète de a avec au plus $\lfloor \log(t) \rfloor + 1$ soustractions.

Comme nous venons de le voir, t est dépendant du module q et de la taille de la valeur à réduire a. Pour certains paramètres utilisés dans les schémas basés sur les réseaux euclidiens, la valeur t est petite. Cela permet alors d'avoir une réduction modulaire efficace.

L'algorithme *Quotient Approximation Reduction* permet une réduction partielle ou complète des coefficients à l'aide d'opérations basiques (additions, soustractions, shifts, multiplications). De plus, la réduction partielle peut être ajustée pour réduire plus ou moins efficacement un coefficient. Cette flexibilité est très intéressante dans le cas de la substitution de Kronecker et de la réutilisation des coprocesseurs existants. Dans un autre travail, nous présentons comment utiliser cette réduction dans le cadre de la réutilisation des coprocesseurs existants.

Contribution 3: Multiplication polynomiale modulaire à l'aide des coprocesseurs RSA/ECC. Dans ce travail, avec un coprocesseur asymétrique actuel, nous effectuons l'ensemble des opérations arithmétiques lors d'une multiplication polynomiale modulaire sur $R_{q,\delta}$, où $\delta \in \{-1, 1\}$. Effectuer l'ensemble des opérations avec un coprocesseur a deux avantages:

- Optimiser la multiplication polynomiale pour les composants disposant d'un faible CPU.
- Éviter la manipulation des coefficients un à un. Ceci peut permettre d'éviter certaines attaques physiques.

Pour pouvoir effectuer l'ensemble des opérations arithmétiques avec un coprocesseur, nous intervertissons l'étape 3 et 4 de la multiplication polynomiale modulaire utilisant la substitution de Kronecker:

3. Réduction des coefficients modulo q. Ces réductions sont effectuées sur l'entier obtenu après la multiplication d'entiers.
4. Radix conversion.

Dans les travaux précédant, les opérations arithmétiques effectuées lors de l'évaluation et la radix conversion sont réalisées à l'aide du CPU. Les opérations à effectuer sont dues à certains polynômes qui ont des coefficients représentés négativement sur $R_{q,\delta}$. En effet, cela implique que lors de l'évaluation et de la radix conversion, des retenues sont à propager. Dans ce travail, nous effectuons la propagation des retenues à l'aide du coprocesseur RSA/ECC.

Dans l'état de l'art, les réductions modulaires modulo q sont effectuées après la radix conversion coefficient par coefficient. Dans ce travail, nous adaptons des algorithmes de réductions, tels que Barrett ou Quotient Approximation Reduction, afin de réaliser ces réductions après la multiplication modulaire d'entiers. Ainsi, les coefficients sont réduits simultanément modulo q à l'aide du coprocesseur.

Pour pouvoir réaliser l'ensemble de ces étapes, le coprocesseur doit pouvoir effectuer des additions/soustractions, des shifts, des multiplications modulaires et non modulaires et des ET logiques. Cette dernière opération est moins courante sur les coprocesseurs actuels. Cependant ajouter cette fonctionnalité à une architecture existante est plus facile et moins coûteux que de concevoir un nouveau coprocesseur pour la multiplication de polynôme.

1.3 Attaques physiques et contremesures des schémas basés sur les réseaux euclidiens

La deuxième partie de la thèse s'intéresse à la sécurisation et l'évaluation de la résilience des cryptosystèmes basés sur les réseaux euclidiens. Dans la suite, nous introduisons les notions en lien avec les attaques physiques et les contremesures que nous présentons dans nos contributions.

Les composants embarqués sont menacés par les attaques physiques. Les premières recherches sur ce sujet sont menés par Kocher *et al.* à partir de l'année 1996 [Koc96; KJJ99]. Nous distinguons deux types d'attaques physiques:

- Les injections de fautes. Ces attaques ont pour but de perturber le bon fonctionnement d'un algorithme. En étudiant le résultat obtenu après la perturbation, un attaquant peut apprendre de l'information sur les secrets manipulés. Les premières injections de fautes sur des cryptosystèmes datent de l'année 1997 [BDL97; BS97].
- Les attaques par canaux auxiliaires. Lors de ces attaques, l'attaquant sonde des grandeurs physiques qui émanent du composant. En étudiant les valeurs obtenues, l'attaquant peut apprendre de l'information sur les secrets manipulés. Les premières attaques par canaux auxiliaires sont celles menés par Kocher *et al.* à partir de 1996 [Koc96; KJJ99].

La perturbation d'un algorithme ou l'écoute des grandeurs physiques sont dépendantes du composant et/ou de l'implémentation. Ainsi, attaquer et sécuriser les cryptosystèmes basés sur les réseaux euclidiens doit se faire au cas par cas. De par la diversité du domaine des attaques physiques, nous nous focalisons dans la suite sur des travaux qui sont en lien avec les recherches effectuées au cours de la thèse.

1.3.1 Attaques physiques

Lors de la standardisation du NIST, les candidats ont été évalués contre différents types d'attaques par canaux auxiliaires. Nous pouvons les ranger dans deux catégories:

- Les attaques non supervisées. L'attaquant a uniquement accès au composant qu'il attaque. Ainsi, il ne peut pas faire d'apprentissage préalable sur un composant similaire.

- Les attaques supervisées. Dans ce cas, l'attaquant a accès à un composant similaire à la cible avec la même implémentation du cryptosystème. L'attaquant peut complètement contrôler ce composant et ainsi faire de l'apprentissage préalable. Par la suite, à l'aide de son apprentissage, l'attaquant réalise son attaque sur le composant cible.

De nombreuses attaques par canaux auxiliaires ont été effectuées sur les candidats basés sur les réseaux euclidiens du NIST. Les implémentations ciblées sont majoritairement celles de référence et donc sans contremesure contre les attaques physiques. Cependant, comme le montrent les articles [HHP+21; NDJ21], des attaques ont également été effectuées sur des implémentations sécurisées.

Les injections de fautes permettent d'altérer la bonne exécution d'un algorithme. Les principales altérations qu'elles peuvent effectuer sont:

- Sauter une instruction.

- Mettre une valeur aléatoire dans un registre ou en RAM.

- Mettre l'ensemble des bits d'un registre ou d'une partie de la RAM à 1 ou 0.

- Inverser la valeur d'un bit.

Dans la suite, nous nous intéresserons plus particulièrement aux attaques par fautes *safe-error*. Ces attaques exploitent le fait qu'une faute peut ou non modifier la sortie d'un algorithme et ainsi, nous apprendre de l'information sur le secret. A notre connaissance, un seul travail de recherche s'intéresse aux attaques *safe-error* contre les schémas basés sur les réseaux euclidiens. Dans cet article [PP21], les auteurs attaquent par fautes le déchiffrement des KEMs Kyber [ABD+21] et Newhope [AAB+19]. Si la sortie du déchiffrement n'est pas modifiée après l'injection, alors l'attaquant apprend de l'information sur le secret. L'attaque est répétée plusieurs fois afin de retrouver l'ensemble des clés secrètes.

1.3.2 Contremesures

En fonction de l'utilisation et du contexte de déploiement, les implémentations embarquées doivent être sécurisées contre les attaques physiques. Les contremesures les plus répandues sont:

- Le masquage. L'idée est de partager, en au moins deux parties, une donnée sensible s telle que $s = s_0 + \ldots + s_{\alpha-1}$. Chacun des $s_i, 0 \leq i < \alpha$, est appelé un *share*. Par la suite, chacun des *shares* est manipulé séparément afin que l'attaquant n'apprenne de l'information que sur l'un d'entre eux à la fois. Ainsi, pour pouvoir retrouver de l'information sur s, l'attaquant doit combiner de l'information sur chacun des s_i. Plus le nombre de *shares* est important, plus la combinaison d'information est compliquée. Cette contremesure est principalement utilisée pour se prémunir contre les attaques par canaux auxiliaires.

- Le shuffling. L'idée de cette contremesure est de rendre aléatoire l'ordre d'exécution des opérations manipulant une donnée sensible. Cette contremesure est principalement utilisée pour se prémunir contre les attaques par canaux auxiliaires.

- La redondance de code. L'objectif est de dupliquer une opération sensible afin de vérifier la bonne exécution de celle-ci. Par exemple lors d'un chiffrement symétrique d'un message m, le composant peut vérifier le bon déroulement du chiffrement en déchiffrant le chiffré et comparer le résultat avec m. Cette contremesure est utilisée pour se prémunir contre les attaques par fautes.

Chacune de ces contremesures a un surcoût. Pour le masquage, cela nécessite de stocker et de faire les mêmes opérations α fois. De plus, le masquage nécessite un grand nombre de génération d'aléa. A l'instar du masquage, le shuffling requiert de la génération d'aléa. D'autre part, l'utilisation du shuffling peut empêcher l'utilisation de certaines optimisations. Enfin, la redondance de code duplique des opérations. En raison des surcoûts, les contremesures implémentées dépendent du contexte de l'attaquant et de la sécurité ciblée. Pour plus de détails sur ces contremesures, se référer à [MOP08].

Dans la suite de la thèse nous nous intéressons plus particulièrement au masquage de certains schémas basés sur les réseaux euclidiens. Ces schémas possèdent une algorithmique commune, ce qui permet d'appliquer les mêmes contremesures de masquage à plusieurs cryptosystèmes. Par exemple, la génération d'aléa se fait sur des distributions similaires. Cependant, chaque cryptosystème utilise des fonctions qui diffèrent, tels que les fonctions d'encodage, de compression, etc. ce qui demande, pour le masquage, des adaptations au cas par cas. Pour prouver que les contremesures de masquage sont sûres, plusieurs modèles théoriques existent. Celui que nous utiliserons est le *probing model*, introduit par Ishai, Sahai et Wagner dans l'article [ISW03]. Ce modèle considère qu'un attaquant peut sonder au plus t valeurs d'un circuit Booléen. Tout algorithme est traduit, après compilation, en un circuit Booléen. Ainsi dans cet article, les auteurs montrent comment partager la donnée sensible en $\alpha = 2t+1$ *shares* pour rendre le circuit sécurisé. Par la suite, dans l'article [RP10], les auteurs améliorent le résultat précédant en sécurisant n'importe quel circuit Booléen en partageant la donnée sensible en $\alpha = t+1$ *shares*. Ces articles prouvent la sécurité du circuit Booléen dans son entièreté. Donc, le moindre changement dans l'algorithme nécessite de prouver la sécurité de l'entièreté du nouveau circuit. Dans l'article [BBD+16], les auteurs introduisent deux notions de sécurité pour le *probing model*: *Strong Non-Interference* et *Non-Interference*. Ces notions permettent de prouver la sécurité d'un algorithme par bloc et ainsi par composition en déduire la sécurité du circuit. Dans la suite, lorsqu'un algorithme est sûr contre un attaquant qui peut sonder $\alpha - 1$ valeurs, alors nous dirons que cet algorithme est sûr à l'ordre $\alpha - 1$.

Plusieurs finalistes, basés sur les réseaux euclidiens de la compétition du NIST, ont vu leurs cryptosystèmes masqués et prouvés dans le *probing model*. En effet, les KEMs Kyber et Saber sont masqués dans les articles [BGR+21; BDK+20], la signature Dilithium dans [MGT+19] et une variante de la signature Falcon dans [EFG+21].

1.3.3 Contributions sur l'évaluation de la résilience des implémentations contre les attaques physiques

A l'instar de la sécurisation des implémentations, l'évaluation de la résilience des implémentations contre les attaques physiques dépend de beaucoup de paramètres. En effet, en fonction du modèle d'attaquant, de l'implémentation et du composant ciblé, les résultats d'une attaque physique peuvent être différents. Dans cette thèse nous évaluons la résilience de plusieurs cryptosystèmes contre des attaques physiques. Dans la suite, nous présentons nos résultats en deux parties.

Résumé des contributions

Contribution 4: Attaque safe-error contre les schémas basés sur les réseaux euclidiens. Dans ce travail, nous évaluons la résilience des cryptosystèmes Dilithium, Kyber, NTRU et Saber contre des attaques *safe-error*. L'objectif est d'exploiter la distribution des coefficients des polynômes secrets. En effet, les distributions utilisées impliquent que les polynômes secrets ont un nombre important de coefficients égaux à 0.

Lors de nos attaques, nous ciblons par fautes des opérations impliquant les coefficients des polynômes secrets. Ainsi, si la sortie n'est pas modifiée alors nous avons ciblé un coefficient valant 0. Afin de retrouver l'ensemble des coefficients égaux à 0, nous répétons l'attaque de manière itérative sur l'ensemble des coefficients secrets.

Pour évaluer la perte de sécurité entrainée par la connaissance de ces coefficients, nous utilisons l'outil [DDG+20]. Cet outil nous permet de déterminer la sécurité théorique des cryptosystèmes avant et après l'attaque. Excepté pour le KEM Kyber, ces attaques entrainent une perte significative de sécurité.

Contribution 5: Exploitation des attaques physiques contre le KEM LAC. LAC [XYD+19] est un KEM présent jusqu'au deuxième tour de la standardisation du NIST. De plus, il est l'un des vainqueurs de la standardisation post-quantique chinoise [fCry20a]. Ce KEM a deux versions qui ont une sécurité sémantique différente [Sak11]:

- Une version IND-CPA (INDistinguishability under Chosen Plaintext Attack): Un schéma IND-CPA est sûr contre les attaques à textes clairs choisis. Un KEM ayant uniquement cette propriété de sécurité doit rafraichir ses clés de chiffrement à chaque encapsulation de clé.
- Une version IND-CCA (INDistinguishability under Chosen Ciphertext Attack): Un schéma IND-CCA est sûr contre les attaques à chiffrés choisis. Cette notion de sécurité est plus robuste et est nécessaire pour pouvoir utiliser les mêmes clés de chiffrement pour plusieurs encapsulations de clé.

Dans ce travail, dans un premier temps, nous attaquons la version IND-CPA du KEM. Pour cela, nous supposons que l'encapsulation de clé est implémentée d'une mauvaise manière. Plus précisément, l'implémentation réutilise la même clé secrète pour plusieurs encapsulations de clé. Ainsi, nous forgeons des chiffrés tels que la clé de session établie nous apprend de l'information sur la clé secrète. Avec cette attaque, nous retrouvons l'entièreté de la clé secrète en au plus 8192 encapsulations de clé.

L'attaque précédente ne s'applique que si l'implémentation IND-CPA réutilise la même clé secrète. La version IND-CCA de LAC permet de conserver la même clé secrète pour plusieurs encapsulations sans être menacée par l'attaque précédente. Ainsi dans une deuxième partie, nous expliquons comment combiner l'attaque précédente et des attaques physiques afin de retrouver la clé secrète de la version IND-CCA.

1.3.4 Contributions sur la sécurisation des implémentations contre les attaques physiques

La sécurisation des implémentations contre les attaques physiques est un vaste domaine. En effet, en fonction du contexte d'utilisation, du modèle d'attaquant et du schéma à sécuriser, les contremesures à appliquer peuvent être différentes. Dans cette thèse, nous nous sommes intéressés à des contremesures de masquage contre un attaquant qui peut sonder $\alpha - 1$ variables. De plus, nous appliquons ces contremesures aux KEMs Kyber et Saber. Enfin, ces contremesures sont prouvées sûres dans le *probing model*. Dans la suite nous présentons nos résultats en deux parties.

Résumé des contributions

Contribution 6: Algorithme de conversion de masque sûr à l'ordre $\alpha-1$. La sixième contribution est une extension de l'algorithme de conversion par table introduit dans [CT03]. L'algorithme que nous proposons permet de calculer pour n'importe quels groupes G et H, à l'aide de re-calcul de table, la fonction $f : G \to H$. Aucune propriété particulière n'est requise sur f. Supposons que nous avons une donnée x partagée en α *shares* sur G:

$$x = x_1 + \ldots + x_\alpha \in G$$

et que nous voulons la partager en α *shares* sur H. Avec notre algorithme nous pouvons calculer:

$$y_1 + \ldots + y_\alpha = f(x_1 + \ldots + x_\alpha) \in H$$

Dans cette contribution, nous prouvons que notre algorithme de re-calcul de table qui permet de calculer la fonction f est sûr, dans le *probing model*, à l'ordre $\alpha - 1$.

La complexité de cet algorithme dépend de la taille du groupe G: $\mathcal{O}(\alpha^2 \cdot |G|)$. De plus, un surcoût en mémoire est à prévoir. En effet, il faut stocker l'ensemble de la table, qui a $|G|$ entrées de la taille des éléments de H.

Par la suite nous donnons des cas d'utilisation de l'algorithme de conversion pour les KEMs Kyber et Saber. Dans ce contexte, les groupes G et H sont de taille raisonnable. Ainsi, l'algorithme de conversion de masque est efficace et a un surcoût mémoire faible.

Contribution 7: Masquage de Kyber et Saber à l'ordre $\alpha - 1$. Dans un premier temps, nous introduisons trois tests d'égalités à 0 prouvés sûrs à l'ordre $\alpha - 1$. Ces tests ont pour objectif de vérifier, de manière sécurisée, si une valeur partagée en α *shares* est égale à 0 modulo q. Cependant, ces tests ne peuvent vérifier l'égalité que d'une valeur partagée. Or, dans le contexte des schémas basés sur les réseaux euclidiens, nous avons besoin de vérifier, de manière sécurisée, l'égalité à 0 de l'ensemble des coefficients partagés en α *shares* d'un polynôme. Ainsi, par la suite, nous adaptons ces tests d'égalités pour qu'ils fonctionnent dans le cas polynomial. Les tests de comparaison de polynômes que nous introduisons nous permettent d'avoir un gain en performance, par rapport à l'état de l'art, pour les schémas Kyber et Saber.

Dans un second temps, nous implémentons les applications de la contribution 6, les comparaisons de polynômes et des algorithmes de masquage de l'état de l'art afin de proposer une implémentation sûre à l'ordre $\alpha - 1$ de Kyber et Saber. De plus, nous embarquons cette implémentation sur un composant embarqué et nous en évaluons les performances.

1.4 Perspectives

Dans ce manuscrit, nous avons apporté des réponses aux problématiques énoncées. Cependant, nos recherches n'apportent qu'un partie des réponses. Dans la suite, nous donnons des axes d'améliorations et des suggestions de nouvelles pistes de recherche.

Optimisations de la multiplication polynomiale. Nos travaux se sont concentrés sur l'optimisation de la multiplication polynomiale modulaire à l'aide des coprocesseurs existants RSA/ECC. La réutilisation de ces coprocesseurs est intéressante pour un déploiement à court terme. A long terme, concevoir des composants spécifiques à la cryptographie post-quantique pourrait être nécessaire. Ainsi, un axe de recherche peut être la conception de composants dédiés à cette cryptographie. Quel type d'architecture CPU ou quel coprocesseur choisir pour optimiser les schémas basés sur les réseaux euclidiens ?

Évaluation et sécurisation des implémentations contre les attaques physiques. Les évaluations sécuritaires que nous avons menées se sont concentrées sur un nombre restreint de candidats et sur un nombre limité d'attaques. Les travaux actuels peuvent être étendus à d'autres schémas et à d'autres types d'attaques.

Dans cette thèse, la sécurité contre les attaques par canaux auxiliaires a été abordée par le masquage dans le *probing model*. Plus précisément sur les KEMs Kyber et Saber. Des poursuites éventuelles à ces travaux pourrait être:

- Masquer d'autres schémas dans le *probing model*.
- Évaluer la sécurité pratique de ces implémentations masquées en les soumettant à des attaques physiques.
- S'intéresser à d'autres contremesures que le masquage.

Étendre les recherches aux autres primitives post-quantiques. Nos recherches se sont concentrées sur la cryptographie basée sur les réseaux euclidiens. Cependant, d'autres primitives cryptographiques sont très prometteuses. Nous pouvons penser à la cryptographie basée sur les codes, les isogénies, les systèmes polynomiaux multivariés, etc. Ainsi une poursuite des recherches serait d'optimiser et sécuriser ces cryptosystèmes dans le contexte de la cryptographie embarquée.

1.5 Organisation

Le manuscrit est divisé en deux parties. La première partie est consacrée à l'optimisation des schémas basés sur les réseaux euclidiens à l'aide de coprocesseurs asymétriques actuels. Cette partie comporte trois chapitres:

Chapitre 2: Ce chapitre introduit l'arithmétique polynomiale utilisée dans les schémas basés sur les réseaux euclidiens. De plus, ce chapitre introduit la substitution de Kronecker qui permet de réutiliser les coprocesseurs asymétriques actuels pour l'arithmétique de polynômes.

Chapitre 3: Ce chapitre introduit le travail sur les variantes de la substitution de Kronecker. Après la présentation de ces algorithmes, nous détaillons la méthodologie pour choisir rapidement quel algorithme est le plus efficace en fonction des spécifications du composant. Enfin, nous donnons des résultats pratiques de nos implémentations.

Les résultats présentés viennent d'une collaboration avec Aurélien Greuet et Guénaël Renault. Ces résultats ont mené à une publication [GMR21].

Chapitre 4: Ce chapitre, dans un premier temps, introduit la réduction modulaire *Quotient Approximation Reduction*. Par la suite, ce chapitre présente le travail sur la multiplication polynomiale modulaire sur $R_{q,\delta}$ à l'aide des coprocesseurs asymétriques actuels. Plus précisément, nous adaptons l'évaluation et la radix conversion afin d'effectuer ces opérations avec les coprocesseurs. Nous adaptons également la réduction de Barrett et Quotient Approximation Reduction pour les utiliser dans ce contexte.

Les résultats présentés viennent d'une collaboration avec Aurélien Greuet et Clémence Vermeersch. Ces résultats ont mené à deux pré-publications [GMV22a; GMV22b].

La deuxième partie du manuscrit est consacrée à l'évaluation et à la sécurisation des schémas basés sur les réseaux euclidiens contre les attaques physiques. Cette partie comporte quatre chapitres:

Chapitre 5: Ce chapitre introduit les attaques physiques et les contremesures que nous allons utiliser dans la suite. De plus, nous introduisons le *probing model* et certains algorithmes prouvés sécurisés dans ce modèle. Enfin, de part le fait que les contremesures nécessitent beaucoup de génération d'aléa, nous discutons de cette génération dans les systèmes embarqués.

Chapitre 6: Dans ce chapitre nous évaluons la résilience de plusieurs cryptosystèmes contre des attaques *safe-error*. Plus précisement, nous nous intéressons aux cryptosystèmes Dilithium, Kyber, NTRU et Saber.

Les résultats présentés viennent d'une collaboration avec Luk Bettale et Guénaël Renault. Ces résultats ont mené à une publication [BMR21].

Chapitre 7: Dans ce chapitre, nous proposons une attaque à chiffrés choisis contre une mauvaise implémentation de la version IND-CPA du KEM LAC. Par la suite, en combinant cette attaque avec des attaques physiques nous attaquons la version IND-CCA du KEM LAC.

Les résultats présentés viennent d'une collaboration avec Aurélien Greuet et Guénaël Renault. Ces résultats ont mené à une publication [GMR20].

Chapitre 8: Ce chapitre présente le travail sur la sécurisation dans le *probing model* des KEMs Kyber et Saber. Pour cela, nous introduisons un nouvel algorithme de conversion de masques basé sur le re-calcul de table. Par la suite, nous introduisons plusieurs algorithmes de comparaison de polynômes sécurisés dans le *probing model*. Enfin, en utilisant l'algorithme de conversion, ceux de comparaison et certains de l'état de l'art, nous sécurisons les KEMs Kyber et Saber dans le *probing model*.

Les résultats présentés viennent d'une collaboration avec Jean-Sébastien Coron, François Gérard et Rina Zeitoun. Ces résultats ont mené à une publication [CGM+21a] et une pré-publication [CGM+21b].

Part I

On using RSA/ECC coprocessor for lattice-based cryptography

Chapter 2

Arithmetic operation of ideal lattice-based schemes

Ideal lattice-based cryptography is believed to be a promising direction to provide efficient and secure post-quantum schemes. Indeed, among the seven finalists of the NIST call for proposal, five are based on ideal lattice-based or assimilated (NTRU) primitives. Moreover, these schemes provide the best trade-off between efficiency, security and compactness than those based on other post-quantum primitives.

In this part we focus on optimizing the ideal lattice-based cryptosystems for the embedded devices. More precisely, we repurpose existing hardware coprocessor in order to optimize modular polynomial multiplication, which is a core operation of the ideal lattice-based schemes.

2.1 Overview of ideal lattice-based schemes operations

Most of the ideal lattice-based schemes define a polynomial ring $R_{q,\delta} = \mathbb{Z}_q[X]/(X^N + \delta)$, where $\delta \in \{-1, 1\}$. In this context the vectors manipulated are polynomials. Hence, these schemes perform polynomial arithmetic like polynomial multiplication, polynomial addition, etc. and modular reductions.

2.1.1 Polynomial multiplication

Among the polynomial operations, the polynomial multiplication is the most expensive in terms of performance. Indeed, the complexity of a schoolbook multiplication is $\mathcal{O}(N^2)$, where $N-1$ is the polynomial degree. Fortunately, optimized polynomial multiplication like Karatsuba, Toom-cook and Number theoretic transform (NTT) algorithms have a better asymptotic complexity. For example, the NTT algorithm has a asymptotic complexity $\mathcal{O}(N \log N)$ [HvD21]. Most of the lattice-based cryptosystems of the NIST call for proposal specify the use of one or more of these three algorithms to optimize polynomial multiplication.

Karatsuba & Toom-Cook. Originally Karatsuba and Toom-Cook [GG03] are algorithms to perform fast integer multiplication. However, these algorithms can be straightforwardly adapted to polynomial multiplication.

The idea of Karatsuba algorithm is to replace polynomial multiplication between two polynomials of degree N by several polynomial multiplications and polynomial additions between polynomials of degree $\frac{N}{2}$. Let $f(X) = f_I(X) + f_S(X)X^{N/2}$ and $g(X) = g_I(X) + g_S(X)X^{N/2}$, where f_I,f_S, g_I and g_S have degree $< N/2$. Naively to multiply these two polynomials:

$$f(X)g(X) = f_I(X)g_I(X) + (f_I(X)g_S(X) + f_S(X)g_I(X))X^{N/2} + f_S(X)g_S(X)X^N$$

However, this technique is less efficient than schoolbook polynomial multiplication. Indeed, this technique requires $\left(4 \times \left(\frac{N}{2}\right)^2\right) = N^2$ integer multiplications plus 3 polynomial additions whereas the schoolbook

technique requires only N^2 integer multiplications. Karatsuba performs fewer multiplications at the cost of extra additions, subtractions and memory usage:

$$\begin{aligned} f(X)g(X) =& f_I(X)g_I(X) + f_S(X)g_S(X)X^N \\ &+ \left[(f_I(X) + f_S(X))(g_I(X) + g_S(X)) - f_I(X)g_I(X) - f_S(X)g_S(X)\right] X^{N/2} \end{aligned}$$

This technique can be called recursively. The asymptotic complexity is $\mathcal{O}(N^{1.58})$.

Toom-Cook algorithm is a generalization of Karatsuba. Instead of splitting the polynomials in two, one can divide them in k. In the context of the NIST lattice-based cryptography, the most used value is $k = 3$. In this case the asymptotic complexity is $\mathcal{O}(N^{1.42})$.

Number Theoretic Transform (NTT). NTT is an algorithm allowing to perform fast polynomial multiplication in $R_{q,\delta}$ [Nus82]. Given a and $b \in R_{q,\delta}$, $a \times b$ is computed as $\mathrm{NTT}^{-1}\left(\mathrm{NTT}\left(a\right) \circ \mathrm{NTT}\left(b\right)\right)$, where $\circ$ is the coefficient-wise multiplication. Asymptotically, the complexity of the NTT is $\mathcal{O}(N \log N)$, where $N - 1$ is the polynomial degree.

Theoretically, NTT has the best asymptotic complexity for polynomial multiplication in $R_{q,\delta}$. However, this algorithm can be used only in a ring which contains n-th roots of unity, where N is a multiple of n. This implies in $R_{q,\delta}$, constraints on the parameters N and q. However, some of lattice-based finalists do not use parameters compliant with the use of the NTT polynomial multiplication. Hence, NTT cannot be used to speed-up all the lattice-based schemes.

2.1.2 Modular reduction

As mentioned previously, in the context of the NIST call of proposal, the polynomials are defined over a modular ring $R_{q,\delta}$. Therefore, the coefficients require a reduction modulo q and the polynomials require a reduction modulo $X^N + \delta$, $\delta \in \{-1, 1\}$.

The polynomial reduction modulo $X^N + \delta$ is straightforwardly a subtraction or an addition of the coefficients of degree greater or equal to N into the coefficients of degree lower than N.

Two cases are to be distinguished for the modular reduction modulo q:

- q is a power of two. Then, the modular reduction is performed with a logical `AND` between a mask and the value to reduce.
- q is a prime number. In the context of NIST lattice-based cryptography, the most used algorithms are Barrett [Bar86] or Montgomery [Mon85] reduction. In Chapter 4 Section 4.1 we present a new modular reduction which finds applications in lattice-based cryptography.

2.2 Hardware accelerator for modular polynomial arithmetic

Constrained environments like smart cards can be very limited in terms of CPU frequency or amount of RAM, especially when compared to regular computers. Moreover, the CPU instruction set can be limited. For example some CPU do not have a native integer multiplication between two registers. Then despite algorithms with good asymptotic complexities, the implementations can still be slow.

In order to accelerate the costly computations of symmetric and asymmetric cryptography, the constrained devices may embed dedicated hardware coprocessors. Most of the industrial coprocessors offer security features (hardware and software countermeasures against faults and various leaks) and are Common Criteria EAL5+ or EAL6+ certified [Lom16]: they are not subject to "obvious" leaks, e.g. single trace attacks without prior learning phase should be hard in practice.

Re-purposing these current coprocessors for the lattice-based schemes allows a faster and cheaper deployment of the post-quantum cryptography. In addition, as several national agencies have stated (ANSSI

[ANS22], BSI [BSI20a]), the first deployments of the post-quantum cryptography will be done in an hybrid approach. This means that systems requiring cryptography must be able to perform classical asymmetric cryptography (RSA/ECC) and post-quantum cryptography. Therefore, re-purposing current coprocessors allows to have a single hardware accelerator for classical and post-quantum cryptography.

The current asymmetric coprocessor are designed to speed-up RSA or elliptic curves cryptosystems. To do so, these components provide a range of (modular) integer operations like multiplication, modular multiplication, addition, subtraction, shift, etc. Therefore, they are not directly adapted for lattice-based cryptography.

In the following paragraph, we describe a well known technique coming from computer algebra that allows to perform polynomial multiplication as integer multiplication. Therefore, as first shown by Albrecht *et al.* in [AHH+19], such technique allows to repurpose existing hardware to speed-up lattice-based cryptography.

2.2.1 Kronecker substitution

The main idea of this substitution, introduced by Kronecker in [Kro82] and first applied in the univariate polynomial context by Schönhage in [Sch82], is to transform polynomial multiplication to an integer multiplication by evaluating the operands and to get back to the result using a radix conversion. More precisely if $f(X) = \sum_{i=0}^{N-1} f_i X^i$ and $g(X) = \sum_{i=0}^{N-1} g_i X^i$ are two polynomials with non-negative coefficients, by considering the product of their evaluations at an integer B we obtain the integer

$$\tilde{f}\tilde{g} = \left(\sum_{i=0}^{N-1} f_i B^i\right)\left(\sum_{i=0}^{N-1} g_i B^i\right),$$

that can be expressed in the base B by radix conversion. Then one obtains the evaluation of fg at B,

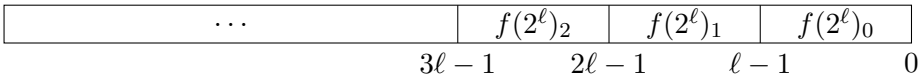
$$\tilde{f}\tilde{g} = \sum_{j=0}^{2N-2}\left(\sum_{i=0}^{j} f_i g_{j-i}\right) B^j, \text{where } f_i = 0, g_j = 0 \text{ for } i, j > N-1$$

and thus deduces the value of each coefficient of the corresponding polynomial $f(X)g(X)$. This radix conversion is possible only if each of the resulting coefficients are smaller than B. The integer B is usually chosen as a power b^ℓ where b is equal to 2 or 10 depending on the context. In all cases, ℓ has to be chosen sufficiently large in order to make the radix conversion effective. More precisely, ℓ could be chosen as the smallest integer such that b^ℓ is greater than the maximum size of the coefficients of the resulting polynomial.

When some of the polynomial coefficients are negative, we have to adapt this reconstruction. In the following algorithms, we present the evaluation process and the radix conversion for a general polynomial of degree $N-1$. We take care of the sign of the coefficients by storing this information in a variable that is given as one of the outputs of the first algorithm. We explain more precisely this procedure in the sequel.

In the following, we consider that the integers are represented in machine with their two's complement. Let $a \in \mathbb{Z}$ such that $-2^{\ell-1} \leq a < 2^{\ell-1}$ then the machine representation of a is $2^\ell + a \mod 2^\ell$.

The integer obtained after evaluation at 2^ℓ of a polynomial of degree $N-1$, can be viewed as the concatenation of N integers of bitsize ℓ. In the following we denote $f(2^\ell)_i = (f(2^\ell) >> (i\ell))\&(2^\ell - 1)$.



Evaluation Algorithm 4 always returns a non-negative integer. Indeed, if the highest degree coefficient of a polynomial $g(X)$ is negative then the algorithm returns the evaluation of $-g(X)$. The parameter `neg` indicates, for the latter radix conversion, if the evaluation algorithm returned the evaluation of $g(X)$ or $-g(X)$.

Algorithm 4 EVALUATION

Input: $f(X) \in \mathbb{Z}[X]$ of degree $N-1$, $\ell \in \mathbb{N}$
Output: $(-1)^{\mathtt{neg}} \times f(2^\ell), \mathtt{neg} \in \{0,1\}$

$\mathtt{borrow} \leftarrow 0$
$f(2^\ell) \leftarrow 0$
if $f_{N-1} < 0$ **then**
  $\mathtt{neg} \leftarrow 1$ //Determine if the last coefficient is negative
else
  $\mathtt{neg} \leftarrow 0$
end if
for $i = 0$ to $N-1$ **do**
  $\mathtt{tmp} \leftarrow (-1)^{\mathtt{neg}} f_i + \mathtt{borrow} \mod 2^\ell$
  if $\mathtt{tmp} > 2^{\ell-1}$ **then**
    $\mathtt{borrow} \leftarrow -1$ //If the coeff is negative then a borrow is propagated to the next coeff
  else
    $\mathtt{borrow} \leftarrow 0$
  end if
  $f(2^\ell) \leftarrow f(2^\ell) + \mathtt{tmp} \times 2^{i\ell}$
end for

RADIX CONVERSION Algorithm 5 converts an integer to a polynomial. More precisely, this algorithm supposes that the input integer comes from a polynomial evaluation. Then, it propagates back carries in order to compensate the -1 added during the EVALUATION. This ensures that $f(X) =$ RADIX CONVERSION(EVALUATION$(f(X), \ell)$).

Algorithm 5 RADIX CONVERSION

Input: $r(2^\ell) = (r(2^\ell)_0, r(2^\ell)_1, \ldots, r(2^\ell)_{N-1})$ and $\mathtt{neg} \in \mathbb{N}$
Output: $r(X) = r_0 + r_1 x + \ldots + r_{N-1} X^{N-1} \in \mathbb{Z}[X]$

$\mathtt{carry} \leftarrow 0$
for $i = 0$ to $N-1$ **do**
  $r_i \leftarrow r(2^\ell)_i + \mathtt{carry} \mod 2^\ell$
  if $r_i > 2^{\ell-1}$ **then**
    $r_i \leftarrow (-1)^{\mathtt{neg}+1}(2^\ell - r_i)$ //Two's complement representation to integer one
    $\mathtt{carry} \leftarrow 1$ //Propagate back carries
  else
    $r_i \leftarrow (-1)^{\mathtt{neg}} r_i$
    $\mathtt{carry} \leftarrow 0$
  end if
end for

The following example presents Kronecker Substitution by using a decimal radix.

Example 1. *Let $f(X) = 8X^2 + 3X + 2$ and $g(X) = -5X^2 - 4X + 1$. Note that the coefficients r_i of the result are such that $-\frac{10^2}{2} \leq r_i \leq \frac{10^2}{2}$. Hence, we evaluate f and g at 10^2 in order to compute $f(X) \times g(X)$ using integer multiplication:*

$$(f(10^2) = 080302, \mathtt{neg}_f = 0) \leftarrow \textit{EVALUATION}(f(X), 10^2)$$
$$(-g(10^2) = 050399, \mathtt{neg}_g = 1) \leftarrow \textit{EVALUATION}(g(X), 10^2)$$

After this evaluation, these two integers can be multiplied:

$$r = f(10^2) \times (-g(10^2)) = 080302 \times 050399 = 4047140498$$

Since the evaluation was done at 10^2*, the resulting polynomial can be interpolated by reading the coefficients 2 digits by 2 digits. The two first digits are* $98 \geq \frac{10^2}{2}$*, that represents the negative number* $-(10^2 - 98) = -2$ *and propagates a carry for the next coefficient. After that, we got* $4 < 50$ *plus the previous carry to obtain* 5*. And so on for the other coefficients. However,* $\mathtt{neg}_g = 1$*, then reading the coefficients like this gives* $-(f(X)g(X))$*.*

$$\begin{aligned} -(f(X) \times g(X)) &= 40X^4 + 47X^3 + 14X^2 + (4+1)X - (10^2 - 98) \\ &= 40X^4 + 47X^3 + 14X^2 + 5X - 2 \\ f(X) \times g(X) &= -40X^4 - 47X^3 - 14X^2 - 5X + 2 = \textsc{Radix Conversion}(r, \mathtt{neg}_f \oplus \mathtt{neg}_g) \end{aligned}$$

The Kronecker Substitution is described in Algorithm 6.

Algorithm 6 KRONECKER SUBSTITUTION

Input: $f(X) \in \mathbb{Z}[X], g(X) \in \mathbb{Z}[X]$ and $\ell \in \mathbb{N}$
Output: $r(X) = f(X)g(X) \in \mathbb{Z}[X]$
1: $(f', \mathtt{neg}_f), (g', \mathtt{neg}_g) \leftarrow$ evaluation of $f(X), g(X)$ at 2^ℓ
2: $r' \leftarrow f'g'$
3: $r(X) \leftarrow$ radix conversion of $(r', \mathtt{neg}_f \oplus \mathtt{neg}_g)$

The integer multiplication at Line 2 can be performed with the asymmetric coprocessor.

The Algorithm 6 performs a polynomial multiplication in $\mathbb{Z}[X]$. To obtain a polynomial result in $R_{q,\delta}$, the reduction modulo $X^n + \delta$ and modulo q can be performed on the output polynomial of the Kronecker substitution. In [AHH+19; BRvV22], the reduction modulo $X^n + \delta$ is performed during the integer multiplication at Line 2.

In Chapter 3, we introduce variants of the Kronecker substitution. These variants use the specific structure of the secret polynomials of the lattice-based cryptosystems. Depending of the coprocessor specification, our variants can be faster than the Kronecker substitution.

Afterwards in Chapter 4, we repurpose contemporary coprocessor to compute the evaluation and the radix conversion. Moreover, we perform the modular reductions modulo q using such coprocessor.

Chapter 3

Polynomial multiplication using RSA/ECC coprocessor

Contents

The results presented in this chapter are from a joint work with Aurélien Greuet and Guénaël Renault in [GMR21].

In this chapter we pursue the seminal work of Albrecht *et al.* in [AHH+19] by introducing two algorithms that perform polynomial multiplication using contemporary coprocessor. Moreover, we asses these algorithms on a smart card component.

Motivation & previous works Polynomial multiplication is one of the most costly operation for ideal lattice-based algorithms. A lot of research has been done on the design of efficient hardware to speed-up polynomial multiplication, see e.g. [ZZY+20; DFA+20; SB20]. However, the transition period should rely on hybrid mechanisms, mixing both classical and post-quantum asymmetric cryptography. Thus, both large modular arithmetic and operations related with post-quantum cryptography, like polynomial multiplication, have to be handled.

Nowadays, hardware accelerating large modular arithmetic are designed and deployed. Then, re-purposing these coprocessors to optimize polynomial multiplication is relevant in terms of costs and ease of deployment for an hybrid cryptography world.

The previous work of Albrecht *et al.* in [AHH+19] optimizes Kyber 1st round algorithm with a RSA/ECC coprocessor, which handles large-integer arithmetic. They use and adapt techniques introduced

in [Har07] which transform polynomial multiplication to an integer multiplication using Kronecker substitution. The work of Wang *et al.* in [WGY20] re-use the Kronecker substitution with such a coprocessor to optimize Saber algorithm.

An independent work of Bos *et al.* in [BRvV22] introduced `Kronecker+`, a generalization of the Kronecker substitution used by Albrecht *et al.* in [AHH+19]. This generalization allows trade-off between number of integer multiplications, size of the integers and the number of polynomial evaluations. Depending on the component and coprocessor specifications, `Kronecker+` allows a faster polynomial multiplication than Kronecker substitution. This article was published at the same time as our work. Therefore `Kronecker+` is not compared with our algorithms.

Contribution. In this chapter we follow the approach initiated in [AHH+19] to improve polynomial multiplication for lattice-based KEMs using a RSA/ECC coprocessor. Such coprocessors usually provide a few basic operations on large integers: multiplication, addition, subtraction, right/left shift.

Our work focuses on the core operation: unreduced polynomial multiplication, i.e. without reduction mod q or $X^N + 1$. Indeed, optimizations for modular reductions can be used on top of any unreduced polynomial multiplication.

More precisely, our work focuses on the unreduced polynomial multiplication when one of the operands has small coefficients. To take advantage of this structure, we introduce a variant of Kronecker substitution and an adaptation of the schoolbook multiplication, called Shift&Add. Both methods allow to handle polynomial multiplication with operations on large integers.

Compared to Kronecker substitution, its variant replaces large integer multiplications with additions, shifts and multiplications between a large integer and a coefficient. For small coefficients, the latter is expected to be cheaper than a regular multiplication. Shift&Add handles polynomial multiplications with only integer additions and shifts. With Shift&Add, the smaller the coefficients are, the fewer operations are performed.

Thereafter, we propose a methodology to help the comparison between Kronecker substitution, its variant and Shift&Add, for a given KEM and a given coprocessor. To this end, we give theoretical complexity estimates for the three algorithms, expressed in terms of basic operations like addition, multiplication, shift and evaluation. Then, by measuring the performance of these basic operations, a developer can determine the fastest algorithm without having to fully develop each algorithm.

Finally, we verify that practical results are in accordance with this methodology for seven parameter sets from the NIST PQC process. In particular, we show that for a given secure comparable component, Shift&Add and Kronecker substitution variant are faster than Kronecker substitution as used in [AHH+19]. We also compare our results with reference software implementations for information purposes only: it is not our aim to compare the efficiency of our implementation with algorithms using specific CPU instructions set as one could find in e.g. [GKS20; CHK+20]. Hence, we are here interested by the challenge to re-purpose secure certified coprocessor deployed in several real-life components for an hybrid transition approach.

Organization. In Section 3.1 we introduce some notations and describe the two new algorithms that we use to perform polynomial multiplication with a coprocessor. Section 3.2 is devoted to discuss the side-channel aspects of the proposed algorithms. In Section 3.3, we show how to determine the evaluation point and establish the complexity of our two algorithms plus the Kronecker substitution in terms of basic coprocessor operations. Finally, in Section 3.4, we assess our algorithms with different set of parameters and show that our practical results are consistent with our theoretical study.

3.1 Algorithms

In this section we present the algorithmic material used in this work. We first detail some notations and well known algorithmic techniques that will be developed in our contributions presented at the end of this section.

3.1.1 Notation and preliminaries

The arithmetic that we study comes naturally from the definition of the ideal lattice-based cryptography and is given as follows.

Rings. For an integer $q \geq 1$, let $\mathbb{Z}_q$ be the residue class group modulo q such that $\mathbb{Z}_q$ can be represented as $\{0, \ldots, q-1\}$. We define $R_{q,1}$ being the polynomial ring $R_{q,1} = \mathbb{Z}_q[X]/(X^N + 1)$.

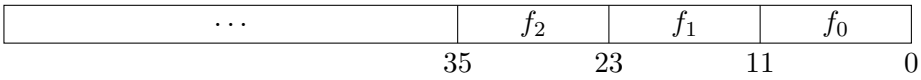
Modular reduction. Let $a, b \in \mathbb{N}$, we denote by $a \mod^{(+)} b$ the unique integer $a' \equiv a \mod b$ such that $0 \leq a' < b$ and $a \mod^{(-)} b$ the unique integer $a' \equiv a \mod b$ such that $-\frac{b}{2} \leq a' < \frac{b}{2}$. In the following, we denote by $a \mod b = a \mod^{(+)} b$

Polynomials A polynomial in $R_{q,1}$ is represented by a polynomial of degree at most $(N-1)$ with coefficients in $\mathbb{Z}_q$. Given $f \in R_{q,1}$, we denote by f_i the coefficient associated with the monomial X^i.

Polynomial representation. Polynomials are represented as byte strings. Let $f(X)$ be a polynomial of degree $N-1$ with all its coefficients $0 \leq f_i \leq \beta$. Then, each coefficient is encoded on $\lfloor \log_2(\beta) \rfloor + 1$ bits. The coefficients are packed as a string of size $N(\lfloor \log_2(\beta) \rfloor + 1)$ bits to represent $f(X)$.

Let $g(X)$ be a polynomial of degree $N-1$ with all its coefficients $-\frac{\delta}{2} \leq g_i \leq \frac{\delta}{2}$. Then, each coefficient is encoded on $\lfloor \log_2(\delta) \rfloor + 1$ such that it is represented as $g_i \mod^{(+)} 2^{\lfloor \log_2(\delta) \rfloor + 1}$. As previously, all the coefficients are packed as a string of size $N(\lfloor \log_2(\delta) \rfloor + 1)$ bits to represent $g(X)$.

Example 2. *Let $f(X)$ be a polynomial with non-negative coefficients lower or equal to $q - 1 = 3328$, then each coefficient is encoded on $\lfloor \log_2(3328) \rfloor + 1 = 12$ bits:*



Large integer operations. In the remainder of this chapter , we introduce two algorithms: Kronecker substitution variant (Alg. 7) and Shift&Add (Alg. 8). For these algorithms the integer operations (multiplication, addition, subtraction, shift) are implicitly performed on large integers and correspond to the operations provided by a hardware coprocessor.

In the following Kronecker substitution or KS refer to the Algorithm 6.

3.1.2 Polynomial multiplication using the structure

The polynomials arising in ideal lattice based cryptosystems are structured (the coefficients follow a centered binomial distribution), we show in the sequel how to gain in efficiency by using them.

Notations To perform an arithmetic operation using a hardware accelerator, the operands and an opcode for the operation to perform must be set into the coprocessor. When the choice of the operation depends on the value of a secret, we denote these sequences of instructions by $\texttt{c} \leftarrow \texttt{Op}\,(\texttt{a}, \texttt{b})$, where $\texttt{Op} \in \{\texttt{Add}, \texttt{Sub}\}$ represents the opcode. This notation allows to simplify the constant-time implementation of an algorithm (see Section 3.1.2 below).

In the following algorithm descriptions, `@var` denotes the address of the variable `var`. Given a pointer `ptr`, `*ptr` stands for the value stored at address `ptr`. The bitwise exclusive-or is denoted by $\oplus$, the bitwise logical `and` by `&` and $a >> \ell$ (resp. $a << \ell$) stands for the logical right shift (resp. logical left shift) by ℓ bits of the value a.

Kronecker substitution variant Classical Kronecker substitution multiplies two integers of length $b^\ell \times N$, where b^ℓ is the evaluation point B and $N-1$ the degree of the polynomials. As mentioned in Section 2.2.1 page 19, ℓ is determined by the maximum coefficient value of the result. In this variant, N multiplications are done on an integer of length $b^\ell \times N$ by an integer of length b^k with $k < \ell$.

A multiplication of two large integers is replaced by N multiplications of a large integer by a small coefficient and N additions/subtractions and shifts. This technique is of interest when considering the multiplication of a polynomial with small coefficients by a generic polynomial. Such multiplications are used in some lattice-based key exchanges. Algorithm 7 multiplies two polynomials of degree $N-1$ such that $(|fg(X)|)_i < 2^\ell$.

Algorithm 7 KRONECKER SUBSTITUTION VARIANT

Input: $f(X) \in \mathbb{N}[X], g(X) \in \mathbb{Z}[X]$ and $N, \ell \in \mathbb{N}$
Output: $r(X) = f(X)g(X)$

1: $(f(2^\ell), \texttt{neg}_f) \leftarrow$ evaluation of $f(X)$ at 2^ℓ
2: **for** $i = N-1$ to 0 **do**
3: **if** $g_i \geq 0$ **then**
4: $\texttt{Op} \leftarrow \texttt{Add}$ // Addition will be performed line 9
5: **else**
6: $\texttt{Op} \leftarrow \texttt{Sub}$ // Subtraction will be performed line 9
7: **end if**
8: $c \leftarrow |g_i|$
9: $r(2^\ell) \leftarrow \texttt{Op}\left(r(2^\ell), c \times f(2^\ell)\right)$
10: $r(2^\ell) \leftarrow r(2^\ell) << \ell$
11: **end for**
12: $r(X) \leftarrow$ radix conversion of $(r(2^\ell), \texttt{neg}_f)$

Example 3. *Let* $f(X) = 8X^2 + 3X + 2$ *and* $g(X) = g_2X^2 + g_1X + g_0 = 5X^2 + 4X + 1$.

$$\begin{aligned} \textit{Then, } f(10^2)g(10^2) &= (f(10^2) \times g_2)(10^2)^2 + (f(10^2)g_1)10^2 + f(10^2)g_0 \\ &= (080302 \times 5)(10^2)^2 + (080302 \times 4)10^2 + 080302 \times 1 \\ &= 4015100000 + 32120800 + 080302 = 4047301102 \end{aligned}$$

The resulting polynomial is recovered with radix conversion like in classical Kronecker.

Shift&Add We now present an adaptation of the schoolbook polynomial multiplication, denoted SHIFT&ADD, where polynomials are represented as integers, after a Kronecker-like evaluation. It relies only on additions and left shifts. This technique is of interest when one of the operands has small coefficients.

The basic idea is explained in Example 4 while a full description is given in Algorithm 8.

Example 4. *Let* $f(X) = 9X^2 + 8X + 3$ *and* $g(X) = g_2X^2 + g_1X + g_0 = 2X^2 - 1$. *Let* $r = 0$. *The computation of* $f(X) \times g(X)$ *is done as follows:*

Step 1. *Evaluate* f*:* $f(10^3) = 009008003$

Step 2. *Since* $g_2 = 2$*:*

1. $r \leftarrow r + f(10^3) \times (10^3)^2$;
2. $r \leftarrow r + f(10^3) \times (10^3)^2$;

Step 3. *Since* $g_1 = 0$*, do nothing;*

Step 4. *Since* $g_0 = -1$, $r \leftarrow r - f(10^3) \times (10^3)^0$;

This leads to

$$\begin{aligned} f(10^3)g(10^3) &= 2f(10^3)(10^3)^2 - f(10^3) \\ &= 2(009008003 \times (10^3)^2) - 009008003 = 18015996991997 \end{aligned}$$

By radix conversion,

$$\begin{aligned} f(X)g(X) &= 18X^4 + (15+1)X^3 - (10^3 - (996+1))X^2 - (10^3 - (991+1))X - (10^3 - 997) \\ &= 18X^4 + 16X^3 - 3X^2 - 8X - 3 \end{aligned}$$

Algorithm 8 Shift&Add

Input: $f(X) \in \mathbb{N}[X], g(X) \in \mathbb{Z}[X]$ with all $g_i \in \{-\frac{\delta}{2}, \ldots, 0, \ldots, \frac{\delta}{2}\}$ and $N, \ell, q \in \mathbb{N}$
Output: $r(X) = f(X)g(X)$

1: $(f(2^\ell), \texttt{neg}_f) \leftarrow$ evaluation of $f(X)$
2: `tmp` $\leftarrow [\,]$ // dummy buffer for constant time implementation
3: **for** $i = N - 1$ to 0 **do**
4: **if** $g_i \geq 0$ **then**
5: `Op` $\leftarrow$ `Add` // addition will be done line 15
6: **else**
7: `Op` $\leftarrow$ `Sub` // subtraction will be done line 15
8: **end if**
9: **for** $j = 0$ to $\frac{\delta}{2} - 1$ **do**
10: **if** $j < |g_i|$ **then**
11: `buff` $\leftarrow$ `@`$r(2^\ell)$ // `Op` in line 15 will be kept
12: **else**
13: `buff` $\leftarrow$ `@tmp` // `Op` in line 15 will be discarded
14: **end if**
15: `*buff` $\leftarrow$ `Op` $\left(r(2^\ell), f(2^\ell)\right)$
16: **end for**
17: $r(2^\ell) \leftarrow r(2^\ell) << \ell$
18: **end for**
19: $r(X) \leftarrow$ radix conversion of $(r(2^\ell), \texttt{neg}_f)$

Isochronous implementations. As it will be explained in Section 3.2, side-channel attacks must be taken into consideration. Hence, Algorithms 7 and 8 are intended to be isochronous: the execution time does not depend on a secret value. In the sequel we assume that for a given operands size, additions and subtractions have same execution time.

At each loop iteration, the same number of additions or subtractions and shifts are performed. However, the time taken to execute conditional assignments (lines 3 to 7 in Algo. 7, lines 4 to 8 and lines 10 to 14 in Algo. 8) can depend on the condition, that itself depends on a secret. Likewise, the computation of absolute value (line 8 in Algo. 7 and line 10 in Algo. 8) must be handled carefully to be isochronous.

We show in Algorithm 9 how to achieve the pointer selection for lines 10 to 14 in Algorithm 8. This pointer selection is done without branches and without table accesses. Thus, its execution time depends neither on any secret value nor on cache access. Since Algorithm 9 computes an absolute value and performs a conditional assignment, the same techniques can be used to make Algorithms 7 and 8 isochronous.

Algorithm 9 Isochronous pointers selection

Input: Coefficient $g_i \in \left\{-\frac{\delta}{2}, \ldots, \frac{\delta}{2}\right\}$ encoded on $k = \lfloor \log_2(\delta) \rfloor + 1$ bits, $j \in \mathbb{N}$, $R =$ bitsize of CPU registers.

Output: $\texttt{buff} \leftarrow \begin{cases} \texttt{@}r(2^\ell) \text{ if } |g_i| < j \\ \texttt{@tmp} \text{ else.} \end{cases}$

1: $s \leftarrow g_i >> (k-1)$ // 0 if $g_i \geq 0$, 1 else
2: $t \leftarrow (s \oplus 1) - 1$ // 0 if $g_i \geq 0$, $2^R - 1 =$ `0xFF...FF` else
3: $t \leftarrow t$ `&` $\left(2^k - 1\right)$ // previous result on k bits
4: $\texttt{abs} = (t \oplus g_i) + s$ // $|g_i| = g_i$ if $g_i \geq 0$, $\overline{g_i} + 1 = \left(g_i \oplus \left(2^k - 1\right)\right) + 1$ else
5: $t \leftarrow \left((\texttt{abs} - j) >> (R-1)\right) - 1$ // 0 if $|g_i| < j$, $2^R - 1 =$ `0xFF...FF` else
6: $\texttt{switch} = \texttt{@}r(2^\ell) \oplus \texttt{@tmp}$
7: $\texttt{buff} = (\texttt{switch \& } t) \oplus \texttt{\&}r(2^\ell)$ // $\texttt{@}r(2^\ell)$ if $|g_i| < j$, `@tmp` else

3.2 Considerations on side-channel attacks

This work focuses on implementations on embedded devices, thus the side-channel aspect must be considered.

Simple Attacks. To avoid simple attacks like SPA, we make our multiplication algorithms isochronous: the execution time does not depend on any secret. For most of the hardware accelerators, large-integer arithmetic timings depend only on the operands size. Hence, we assume that the execution time of shift, addition, subtraction and multiplication on large integers does not depend on the processed data. Moreover, exploiting secure coprocessors leaks by SPA during their computation is hard in practice. Then, we assume that an hardware addition cannot be distinguished from a hardware subtraction with a simple power consumption or EM attack. In addition, in our experiments, the CPU is set to perform the multiplication and division between registers in constant time. These instructions are used to compute modular reductions.

Under these assumptions, it is clear that a straightforward implementation of Kronecker substitution does not have any operation depending on the manipulated data.

In addition, we explained in Section 3.1.2 and showed in Algorithm 7 and 8 how to compute isochronous KSV and Shift&Add, based on techniques described in Algorithm 9. With such techniques and since by assumption, addition cannot be distinguished from a subtraction, from an attacker point of view, the execution of the same instructions are performed at each loop iteration in Algorithm 8 regardless of the secret. Thus, SPA-like attack cannot reveal secret information.

Differential/Correlation Attacks. Several physical attacks against post-quantum cryptosystems have been studied [CCA+20]. Among them, some correlation attacks have been done on polynomial multiplication, see e.g. [EFG+17; OSP+18a] [RJH+18; RdCR+16; RRD+16] and references therein.

These attacks are based on the fact that power consumption or electromagnetic emissions are correlated with the data being manipulated. Such attack targets an intermediate variable of the form $s \times m$, where s is a small part of the secret and m a known input, like a message or a public key. Since s is small and m is known, the attacker can make a guess on s, compute $s \times m$ and predict, for a given leakage model, the expected consumptions or emissions for a series of different m's. Then, using physical measurements, like power consumption traces, and statistical tools, the correct key guess can be found: it is likely to be the one for which the correlation between predictions and real measurements is the strongest.

Masking is the classical countermeasure against such attacks [MOP08]. The sensitive data is split in two shares, each share being manipulated individually. For multiplication in lattice-based cryptography, the secret polynomial s has a structure, e.g. small coefficients, that can be exploited for performance optimization. Hence, to keep this structure, one can split the known part: for a given public value m, consider a random m_1 and set $m_2 = m - m_1$. Then $s \times m$ can be computed as $(s \times m_1) + (s \times m_2)$, each $(s \times m_i)$ being processed independently. Since m_1 and m_2 are unknown and appear to be uniformly distributed, the computation of $(s \times m_i)$ can not be correlated to $s \times m$, so that order 1 attacks will fail [MOP08]. With this countermeasure, the overhead is roughly an extra polynomial addition and an extra polynomial multiplication.

In [RdCR+16], the additively-homomorphic property of R-LWE schemes is used to mask, by adding the encryption of a random message before the decryption process. This random message encryption can be precomputed, so that the overhead is only an extra addition.

Recall that our goal is to determine, given a lattice-based scheme and a device, the fastest polynomial multiplication. For both kind of masking, the overhead does not depend on the choice of the multiplication algorithm: if unmasked multiplication A is faster than unmasked multiplication B, then the same result holds for their masked versions. Then in the sequel, we focus on comparing the basic unmasked versions of polynomial multiplication algorithms.

3.3 Complexity

This section is devoted to compare the performance of Kronecker substitution (KS), Kronecker substitution variant (KSV) and Shift&Add when using an existing RSA/ECC hardware accelerator. Hence, the complexity is given in terms of basic arithmetic operations performed by such accelerators: addition, multiplication and multiplication by a power of 2 (left shift) on large integers. Moreover, the number of evaluation differs between KS, KSV and Shift&Add. Then, the evaluation step is considered in the following complexities. Since the cost of these operations depends on the operand sizes, we first determine the minimal value for the parameter ℓ.

Our work is focused on lattice-based key exchange submitted to the NIST PQC standardization process. The polynomial multiplication of these key exchanges is over $R_{q,1} = \mathbb{Z}_q[X]/(X^N + 1)$. To compute such a multiplication, we first multiply over $\mathbb{Z}[X]$. The result is then reduced modulo $X^N + 1$ and each coefficient is reduced modulo q. Since this reduction step is the same for the three methods, its cost is not relevant for a theoretical comparison between them. Hence, it is not included in the complexity computation. Likewise, radix conversion step is the same and is not included in the complexity. However, these costs are considered in the performance results of Section 3.4.

3.3.1 Choice of ℓ

As explained in Section 3.1, polynomial multiplication can be reduced to integer multiplication. To this end, the polynomials are evaluated at a point such that the result can be recovered by radix conversion.

This evaluation point is determined by the maximum coefficient of the multiplication result.

In the following, we suppose that the polynomial evaluation is done by Algorithm 4. Then, each evaluated polynomial is represented as a non-negative integer.

Proposition 1. *Let f and g be polynomials of degree $N-1$ such that for all $i \in \{0,\dots,N-1\}, 0 \leq f_i \leq \beta$ and $-\frac{\delta}{2} \leq g_i \leq \frac{\delta}{2}$. Then*

- $\forall i \in \{0,\dots,2N-2\}, |(f(X)g(X))_i| < 2^{\ell-1}$, *where* $\ell = \lfloor \log_2(N\beta\delta) \rfloor + 1$.
- *Each coefficient of $f(X)g(X)$ is encoded on at most ℓ bits.*
- $\log_2(f(2^\ell)) < N\ell$ *and* $\log_2(\left|g(2^\ell)\right|) < N\ell$.

Proof. Let $r(X) = f(X)g(X)$. Then $r(X)$ is of degree $2N-2$ and its k-th coefficient is $r_k = \sum_{i=0}^{k} f_i g_{k-i}$. To prove the first assertion, we first consider the coefficients r_k for $k \leq N-1$. Since for all i, $0 \leq f_i \leq \beta$ and $|g_i| \leq \delta/2$, we get, for $k \leq N-1$:

$$|r_k| = \left|\sum_{i=0}^{k} f_i g_{k-i}\right| \leq \sum_{i=0}^{k} |f_i|\,|g_{k-i}| \leq \sum_{i=0}^{k} \beta\frac{\delta}{2} \leq \sum_{i=0}^{N-1} \beta\frac{\delta}{2} \leq N\beta\frac{\delta}{2}.$$

For $k \geq N$, note that since f (resp. g) has degree $N-1$, $f_i = 0$ (resp. $g_i = 0$) for $i \geq N$. Hence,

$$|r_k| = \left|\sum_{i=0}^{k} f_i g_{k-i}\right| = \left|\sum_{i=0}^{N-1} f_i g_{k-i} + \sum_{i=N}^{k} f_i g_{k-i}\right| = \left|\sum_{i=0}^{N-1} f_i g_{k-i}\right| \leq N\beta\frac{\delta}{2}.$$

Thus, for $k \in \{0,\dots,2N-2\}$, $|r_k| \leq N\beta\frac{\delta}{2} < 2^{\lfloor \log_2(N\beta\frac{\delta}{2}) \rfloor + 1} = 2^{\ell-1}$. Each coefficient of the result is $-N\beta\frac{\delta}{2} \leq r_k \leq N\beta\frac{\delta}{2} < 2^{\ell-1}$, then to handle the negative case each coefficient of the result is at most encoded on ℓ bits.

We prove now the third assertion.

$$f(2^\ell) = \sum_{i=0}^{N-1} f_i 2^{i\ell} \leq (2^\ell - 1)\sum_{i=0}^{N-1} 2^{i\ell} = (2^\ell - 1)\frac{2^{N\ell}-1}{(2^\ell-1)} = 2^{N\ell} - 1 < 2^{N\ell}$$

Thus, $\log_2\left(f(2^\ell)\right) \leq N\ell$. Likewise, $\log_2\left(\left|g(2^\ell)\right|\right) \leq N\ell$. □

Remark 1. *The previous proposition applies with polynomial $f(X)$ with non-negative coefficients. In our context, the negative coefficients of a polynomial in $R_{q,1}$ can be replaced with their non-negative equivalent in $\{0,\dots,q-1\}$.*

3.3.2 Complexity estimates

In this section, we estimate the complexity of our multiplication algorithms. We express them in terms of the following basic operations. Let $E(N)$ be the evaluation complexity function for a polynomial of degree $N-1$. Let $M(x,y)$ and $A(x,y)$ be the multiplication and addition (or subtraction) of integers complexity functions depending on the bitsize of the inputs.

Example 5. *Let a and b be two integers, let $x = \lfloor \log_2(a) \rfloor + 1$ and $y = \lfloor \log_2(b) \rfloor + 1$. Then the cost of computing $a \times b$ (resp. $a + b$) is $M(x,y)$ (resp. $A(x,y)$).*

Likewise, $S(x,s)$ denotes the shift complexity function where x is the bitsize of the integer to shift on s bits.

Kronecker Substitution (KS)

Proposition 2. *Let $f(X)$ and $g(X)$ be two polynomials of degree $N-1$. Each coefficient of $f(X)$ is defined over $\mathbb{N}$ and each coefficient of $g(X)$ is defined over $\mathbb{Z}$. If for a given ℓ every coefficient $|(f(X)g(X))_i|$ is lower than $2^{\ell-1}$, then the multiplication complexity of $f(X)g(X)$ with Kronecker substitution is $2E(N) + M(N\ell, N\ell)$.*

Proof. Let $f(X)$ and $g(X)$ be polynomials of degree $N-1$. To compute $f(X) \times g(X)$ with Kronecker substitution, the following steps are performed:

1. Evaluation of $f(X)$ and $g(X)$ at 2^ℓ. According to Proposition 1, $\log_2\left(f(2^\ell)\right) \leq N\ell$ and $\log_2\left(\left|g(2^\ell)\right|\right) \leq N\ell$.
2. Multiplication of two integers of bitsize $N\ell$.
3. Radix conversion of the coefficients. As mentioned above, this step is omitted in the complexity.

Then, Kronecker substitution complexity is $2E(N) + M(N\ell, N\ell)$. □

Kronecker Substitution Variant (KSV)

Proposition 3. *Let $f(X)$ and $g(X)$ be polynomials of degree $N-1$. Each coefficient of $f(X)$ are defined over $\mathbb{N}$ and each coefficient of $g(X)$ are defined over $\mathbb{Z}$. If for a given ℓ every coefficient $|(f(X)g(X))_i|$ is lower than $2^{\ell-1}$ and all coefficients of $g(X)$ fit on k bits, then the polynomial multiplication complexity of $f(X)g(X)$ with Kronecker substitution variant is $E(N) + N(M(N\ell, k) + S(N\ell, \ell) + A(N\ell, N\ell))$.*

Proof. Let $f(X)$ and $g(X)$ be polynomials of degree $N-1$, such that each bit representation for g_i fits on k bits. Computing $f(X) \times g(X)$ with Kronecker substitution variant is done by doing:

1. The evaluation of $f(X)$ at 2^ℓ. Then, $\log_2\left(f(2^\ell)\right) \leq N\ell$.
2. A "`for`" loop with N iterations, each step being:
   - A multiplication between an integer of bitsize $N\ell$ and a coefficients of bitsize k,
   - An addition or subtraction between two integers of size $N\ell$,
   - A ℓ-shift of an integer of bitsize $N\ell$.
3. A radix conversion of the coefficients. As mentioned above, this step is omitted in the complexity.

Then the complexity of the multiplication using Kronecker substitution variant is $E(N) + N(M(N\ell, k) + S(N\ell, \ell) + A(N\ell, N\ell))$. □

Shift&Add

Proposition 4. *Let $f(X)$ and $g(X)$ be polynomials of degree $N-1$ with coefficients in $\mathbb{Z}$. If for a given ℓ every coefficient $|(f(X)g(X))_i|$ is lower than $2^{\ell-1}$ and all coefficients of $g(X)$ belong to $\{-\frac{\delta}{2}, \ldots, 0, \ldots, \frac{\delta}{2}\}$, then the polynomial multiplication in Shift&Add costs $E(N) + N(S(N\ell, \ell) + \frac{\delta}{2}A(N\ell, N\ell))$.*

Proof. Let $f(X)$, $g(X)$ be polynomials of degree $N-1$. Algorithm Shift&Add computes the multiplication as follow:

1. Evaluate $f(X)$ at 2^ℓ. Then, $\log_2\left(f(2^\ell)\right) \leq N\ell$.
2. A "`for`" loop called N times, each step being:

- $\frac{\delta}{2}$ additions or subtractions between two integers of size $N\ell$.
- A ℓ-shift of an integer of bitsize $N\ell$.

3. Radix conversion of the coefficients. As mentioned above, this step is omitted in the complexity.

Then Shift&Add complexity is $E(N) + N(S(N\ell, \ell) + \frac{\delta}{2}A(N\ell, N\ell))$. □

Complexities comparison Let $f(X)$ and $g(X)$ be polynomial of degree $N-1$. Assume that the coefficients of $g(X)$ belong to $\{-\frac{\delta}{2}, \ldots, 0, \ldots, \frac{\delta}{2}\}$ and that for all $i \in \{0, \ldots, 2N-1\}, |(f(X)g(X))_i| \leq 2^{\ell-1}$. To choose the most efficient algorithm for polynomial multiplication we need to compare the three following complexities, depending on the component specification.

- Shift&Add: $E(N) + N(S(N\ell, \ell) + \frac{\delta}{2}A(N\ell, N\ell))$.
- Kronecker substitution: $2E(N) + M(N\ell, N\ell)$.
- Kronecker substitution variant: $E(N) + N(M(N\ell, k) + S(N\ell, \ell) + A(N\ell, N\ell))$.

In Section 3.4.2, we explain how to instantiate the different basic complexities in order to compare the above estimations. We focus our study on the execution time and do not provide memory consumption estimates.

3.3.3 Time-memory trade-offs

The amount of RAM in embedded devices can be very limited. However, some devices allow a larger RAM consumption which can be utilized to speed-up our algorithms.

Polynomial representation. In Section 3.1 we describe our compact polynomial representation. This representation is useful to optimize our memory consumption but not to access to the polynomial coefficients. The evaluation and radix conversion require a lot of accesses to the coefficients, thus representing these coefficients as a machine word (e.g 32-bit) improves significantly the performance of these algorithms. Moreover, for some components, using a machine word representation allows to replace shift by pointers arithmetic.

Precomputation. In our context, polynomial multiplication is between $f(X)$ which is a random polynomial over $R_{q,1}$ and $g(X)$ which has coefficients in $\{-\frac{\delta}{2}, \ldots, \frac{\delta}{2}\}$, where δ is close to 0. Then, we can precompute $\frac{\delta}{2} - 1$ multiples of $f(X)$: $2 \times f(X), \ldots, \frac{\delta}{2} \times f(X)$ to reduce the number of operation to one addition/subtraction and one shift of each iteration of Shift&Add loop "`for`".

Positive case. As mentioned above, one of the polynomials can have negative coefficients. That implies to handle carry propagation during the evaluation, radix conversion or the subdivision. The carry propagation requires an important amount of software implementation to be handled. However, KSV and Shift&Add can perform the polynomial multiplication without negative coefficient and then without carry propagation. Indeed, as the cost of a supplementary evaluation/storage of $-f(X) \mod q$, which is the computation $q - f_i$ for all the coefficients of f, KSV (resp. Shift&Add) multiplies (resp. add) $f(X)$ when the coefficient of $g(X)$ is non-negative and $-f(X) \mod q$ when the coefficient is negative.

3.3.4 Polynomial subdivisions

RSA/ECC coprocessors perform large integers arithmetic with data in buffer whose size has a fixed limit. In our algorithms, after polynomial evaluation, the resulting integer is generally too large to fit in these buffers. In this case, a subdivision is performed on the polynomials before evaluation. Let $f(X) = f_I + f_S X^{N/2}$ and $g(X) = g_I + g_S X^{N/2}$, where f_I,f_S, g_I and g_S have degree $< N/2$. We consider the following methods to subdivide:

Naive: $f(X)g(X) = f_I g_I + (f_I g_S + f_S g_I)X^{N/2} + f_S g_S$

Karatsuba: $f(X)g(X) = f_I g_I + ((f_I + f_S)(g_I + g_S) - f_I g_I - f_S g_S)X^{N/2} + f_S g_S X^N$

Karatsuba performs fewer multiplications at the cost of extra additions, subtractions and memory usage. Depending on the coprocessor specification, it can be slower than the naive subdivision.

Impact on ℓ Subdividing d times divides the value of N by 2^d. However, for the naive subdivision it does not reduce the value of $\ell = \lfloor \log_2(N\beta\delta) \rfloor + 1$. Indeed, after recombination the result's bitsize is the same as a multiplication without subdivision.

Karatsuba requires the multiplication $(f_I + f_S)(g_I + g_S)$, which increases the value of the evaluation point by one. In fact, let f_I, f_S have coefficients in $\{0, \ldots, \beta\}$ and g_I, g_S have coefficients in $\{-\frac{\delta}{2}, \ldots, \frac{\delta}{2}\}$. Then, $(f_I + f_S)$ have coefficients in $\{0, \ldots, 2\beta\}$ and $(g_I + g_S)$ have coefficients in $\{-\delta, \ldots, \delta\}$. Thus, $\ell' = \lfloor \log_2(\frac{N}{2} \cdot 2\beta \cdot 2\delta) \rfloor + 1 = \ell + 1$. More generally, if d subdivisions are required, then $\ell' = \ell + d$ must be used instead of ℓ.

Impact on the complexities Subdividing allows to perform a large integer multiplication by few multiplications, additions and subtractions on smaller integers. KS can require one more subdivision than KSV and Shift&Add to avoid a lot of load, store and carry propagation done in software. Hence, to determine the most efficient algorithm, with the requirement of d subdivisions for KSV and Shift&Add and d' subdivisions for KS, we only need to compare the following complexities, depending on the component specification.

- Shift&Add: $E(N) + x^d \left(\frac{y}{x} A(\frac{N}{2^d}\ell, \frac{N}{2^d}\ell) + \frac{N}{2^d}(S(\frac{N}{2^d}\ell, \ell) + \frac{\delta}{2} A(\frac{N}{2^d}\ell, \frac{N}{2^d}\ell)) \right)$.
- Kronecker substitution: $2E(N) + x^{d'} \left(\frac{y}{x} A(\frac{N}{2^d}\ell, \frac{N}{2^d}\ell) + M(\frac{N}{2^{d+1}}\ell, \frac{N}{2^{d+1}}\ell) \right)$.
- Kronecker substitution variant: $E(N) + x^d \left(\frac{y}{x} A(\frac{N}{2^d}\ell, \frac{N}{2^d}\ell) + \frac{N}{2^d}[M(\frac{N}{2^d}\ell, k) + S(\frac{N}{2^d}\ell, \ell) + A(\frac{N}{2^d}\ell, \frac{N}{2^d}\ell)] \right)$.

The values x and y are, respectively, the number of sub-multiplications and the number of additions (or subtractions) required by the subdivision method. Hence, for the naive subdivision $x = 4$ and $y = 3$ and for Karatsuba $x = 3$ and $y = 6$.

In this chapter we use the naive or Karatsuba to subdivide but the subdivision can be achieved with other methods.

3.4 Assessment

3.4.1 Context

We evaluate three lattice-based algorithms: LAC, Kyber and Saber. They have been submitted to the NIST PQC standardization [BMD+21; XYD+19; BDK+18]. Saber and Kyber passed the 2nd round and they are finalists of the 3rd round. LAC did not pass the NIST 2nd round but is one of the winners of the Chinese cryptographic competition and thus remains relevant to study.

Parameters In the following, the Kyber 1st round specifications are considered for Kyber512R1 and Kyber1024R1, in order to compare our results with the previous work in [AHH+19]. For the other schemes, 2nd round specifications are considered. However, Saber 3rd round parameters and the last two Kyber's 3rd round security levels parameters are the same as 2nd round. Our results come from a device with dedicated hardware coprocessor for large-integer operations (multiplication, addition, subtraction, right/left shift).

In the following results we consider the multiplication of $f(X)$ by $g(X)$ over $R_{q,1} = \mathbb{Z}_q[X]/(X^N + 1)$, where:

1. $f(X)$ is of degree $N-1$ with coefficients in $\{0, \ldots, q-1\}$.
2. $g(X)$ is of degree $N-1$ with coefficients in $\left\{\frac{-\delta}{2}, \ldots, \frac{\delta}{2}\right\}$.
3. The evaluation point is 2^ℓ, where the value of ℓ is given by Proposition 1.

Parameters for each candidate are represented in the following table.

Set/Param	N	q	δ	ℓ
Kyber512R1	256	7681	10	25
Kyber1024R1	256	7681	6	24
KyberR2	256	3329	4	22
Light Saber	256	2^{13}	10	25
Fire Saber	256	2^{13}	6	24
Lac128	512	251	2	18
Lac256	1024	251	2	19

Target Assessments are done on a smart card component. Due to intellectual properties reasons, the component name or a detailed description cannot be given. However, details on our analysis are given, allowing it to be reproduced on any component embedding a similar coprocessor designed for large integer arithmetic. This component is used in real-life products like bank cards, passports, secure elements, etc. It embeds hardware accelerators for asymmetric cryptography computations, including large-integer arithmetic.

In the following, the chip is referred as "Component A".

Component A. The Component A is a ARM 32-bit architecture. Its asymmetric coprocessor can handle 2048-bit operands. The addition of two 2048-bit integers is done in less than ten cycles, while the multiplication takes several thousand cycles. Since the addition is several thousand times cheaper than the multiplication, Shift&Add is expected to be faster than Kronecker substitution when one operand has coefficients close to 0.

The coprocessor can also multiply a coefficient (of bit size lower than 32 bits) to a 2048-bit operand, that is of interest with Kronecker substitution variant. Then, KSV is expected to be faster than Shift&Add and Kronecker substitution when one operand has small coefficients but not too close to 0.

3.4.2 From theory to practice: a methodology

In this section, we propose a methodology to determine, with a minimal amount of implementation, which polynomial multiplication algorithm is the fastest on a given component and for a given set of parameters. This is done by measuring the timings of basic operations (integer multiplication, addition and subtraction, shift and evaluation) and plugging them into the complexities from Section 3.3. Hence, this ease the algorithm choice without coding all the multiplication algorithms. This can help a developer to quickly decide which algorithm is the best choice.

We detail the methodology for Kyber512R1 parameters on component A. We focus on the comparison between Kronecker substitution and Shift&Add and between Kronecker substitution variant and Shift&Add.

Component A's coprocessor can handle operands size of 2048 bits. For Kyber512R1 parameters $N = 256$ and $\ell = 25$, the integers after polynomial evaluation are of size $N\ell = 6400$ bits. Thus, we subdivide the polynomial to perform our algorithms (see Section 3.3.4).

On this component, the naive subdivision is more efficient than Karatsuba. For Kronecker substitution we subdivide until the result $f(X)g(X)$ can fit in the coprocessor. That requires 3 subdivisions to get $\frac{N}{2^3}\ell = 800$ bitsize per operand. For Shift&Add and KSV, we subdivide until the subdivisions of polynomial $f(X)$ can fit in the coprocessor. That requires 2 subdivisions of size 1600-bit. Then, KS requires one more subdivision than KSV and Shift&Add to avoid a lot of load, store and addition in software. Indeed, KS result doubles the size of the operand while each iteration of KSV and Shift&Add increases only by ℓ the result size, which can be handled without an important amount of software manipulation.

Naive subdivision transforms large arithmetic operations to 4^d smaller arithmetic operations, where d is the required depth. This leads to the following expressions for Shift&Add, KS and KSV complexities:

$$\begin{aligned}
\text{S\&A} &: 4^2\left(\frac{N}{2^2}[\frac{\delta}{2}A(1600,1600) + S(1600,\ell)]\right) + 12A(1600,1600) + E(N)\\
\text{KS} &: 4^3M(800,800) + 48A(1600,1600) + 2E(N)\\
\text{KSV} &: 4^2\left(\frac{N}{2^2}[M(1600,\ell) + S(1600,\ell) + A(1600,1600)]\right) + 12A(1600,1600) + E(N)
\end{aligned}$$

The value $12A(1600,1600)$ and $48A(1600,1600)$ are due to the recombination of the naive subdivision but they are negligible. Let $\mathcal{C}(S\&A) = \frac{N}{2^2}\left(\frac{\delta}{2}A(1600,1600) + S(1600,\ell)\right) + 3A(1600,1600)$. We measure the execution time corresponding to $\mathcal{C}(S\&A)$ as a reference. Then we measure $E(N)$, $M(800,800)$ and $M(1600,\ell) + S(1600,\ell) + A(1600,1600)$ and express them in terms of $\mathcal{C}(S\&A)$.These measurements are obtained with an emulator. We get that:

$$\begin{aligned}
E(N) &\simeq 1.34 \times \mathcal{C}(S\&A)\\
M(800,800) &\simeq 0.20 \times \mathcal{C}(S\&A)\\
\frac{N}{2^2}(M(1600,\ell) + S(1600,\ell) + A(1600,1600)) &\simeq 0,86 \times \mathcal{C}(S\&A).
\end{aligned}$$

It follows the following estimations for Shift&Add, KS and KSV:

$$\begin{aligned}
\text{S\&A: } & 4^2\mathcal{C}(S\&A) + E(N) \simeq 17.34 \times \mathcal{C}(S\&A)\\
\text{KS: } & 4^3M(800,800) + 2E(N) \simeq 15.48 \times \mathcal{C}(S\&A)\\
\text{KSV: } & 4^2(0.86 \times \mathcal{C}(S\&A)) + E(N) \simeq 15.1 \times \mathcal{C}(S\&A)
\end{aligned}$$

Hence, in this configuration (Kyber512R1 parameters), KSV is expected to be the fastest algorithm for these parameters.

Following this methodology, we get the expected ratios in Table 3.1 between Kronecker substitution and Shift&Add and between KSV and Shift&Add, all measurements being obtained with emulators.

	KS/S&A	KSV/S&A		KS/S&A	KSV/S&A
Kyber512R1	0.89	0.87	Fire Saber	1.17	1.1
Kyber1024R1	1.17	1.1	Lac128	1.45	1.37
KyberR2	1.26	1.2	Lac256	1.41	1.42
Light Saber	0.89	0.87			

Table 3.1: Expected ratio based on basic operations performances for component A

3.4.3 Experiments

In the sequel, assessments are done using an emulator and we measure the performance of the following:

- Algorithms relying on hardware coprocessors (KS, KSV, Shift&Add)
- Software implementation of schoolbook multiplication (Saber, Lac)
- Software implementation of NTT (Kyber)

The NTT implementation used for Kyber512R1 and Kyber1024R1 is detailed in [LN16]. For KyberR2, the reference implementation of Kyber 2nd round is used [BDK+18]. We measure NTT performance with the requirement of frequency transformations of both polynomials (w/ NTT(A)) or with only one frequency transformation (w/o NTT(A)).

Our hardware polynomial multiplications consider that the inputs are not in the NTT domain. Hence, in the case of a Kyber specification compliant implementation additional inverses NTT would have to be performed, which implies that a slower hardware polynomial multiplication than a NTT one.

For LAC software naive multiplication, we report the result for a C implementation and an optimized version in assembly (asm).

Software implementation results are given for information purposes and are not specifically optimized. Indeed, our objective is to provide algorithms which can be applied on many as possible components using a RSA/ECC coprocessors. Therefore, optimization for software polynomial multiplication using specific instructions set is out of our scope.

Our results are obtained by computing a complete polynomial multiplication over $R_{q,1}$ with a compact representation as mentioned in Section 3.1. The following timings take into account the computation done by the CPU and the coprocessor. Moreover, any optimization of the reduction modulo q and $X^N + 1$ are done. For the same reason, we avoid any optimization which requires a specific software instructions set. However, software optimization for modular reductions can be used on top of any polynomial multiplication algorithm.

Param/Algo	KS	KSV	S&A	NTT (w / w/o NTT(A))	Naive
Kyber512R1	588k	**556k**	594k	1139k / 793k	N/A
Kyber1024R1	572k	539k	**500k**	1139k / 793k	N/A
KyberR2	535k	512k	**441k**	998k / 704k	N/A
Light Saber	580k	**546k**	585k	N/A	11691k
Fire Saber	563k	530k	**493k**	N/A	11440k
Lac128	1594k	1586k	**1285k**	N/A	15560k / 1683k (asm)
Lac256	6209k	6310k	**4980k**	N/A	62340k / 7494k (asm)

Table 3.2: Polynomial multiplication over $R_{q,1}$ cycle count on component A

Practical results are given in Table 3.2. As expected for the component A regarding Table 3.1, Shift&Add is the fastest algorithm for 5/7 parameter sets and KSV is the fastest algorithm for the 2 others. Hence, for this component and these parameters, Shift&Add or KSV are faster than the hardware multiplication introduced in [AHH+19].

Note that the theoretical ratios from Table 3.1 are not the exact same ratios between the practical results. This is because radix conversion and reduction over $R_{q,1}$ are not taken into account in the theoretical complexity (see Section 3.3), while these operations are part of the timings in Table 3.2. Nevertheless, the fastest algorithm is always the expected one and proves that our methodology introduced in Section 3.4.2 is relevant.

Positive time/memory trade-off. The positive time/memory trade-off is presented in Section 3.3.3. This trade-off ensures that any carry propagation must be handled during polynomial multiplication, at cost of a supplementary storage ($N\ell$ bits). However, this trade-off only applies to KSV and S&A algorithms.

The Table 3.3 shows the practical results obtained on component A, with the highest security parameters of Kyber, Saber and LAC. Furthermore, $\text{KSV}_{\geq 0}$ (resp. $\text{S\&A}_{\geq 0}$) denotes the algorithm Kronecker substitution variant (resp. Shift&Add) using the positive trade-off. For the three parameters sets, a significant performance gain on hardware polynomial multiplication is achieved (at least 1.35), at cost of an additional storage of $N\ell$ bits.

Param/Algo	$\text{KSV}_{\geq 0}$	$\text{S\&A}_{\geq 0}$	$\text{KSV}/\text{KSV}_{\geq 0}$	$\text{S\&A}/\text{S\&A}_{\geq 0}$
KyberR2	362k	**326k**	1.41	1.35
Fire Saber	363k	**354k**	1.46	1.39
Lac256	4437k	**3455k**	1.42	1.44

Table 3.3: Positive trade-off polynomial multiplication over $R_{q,1}$ cycle count on component A

Chapter 4

Modular polynomial multiplication using RSA/ECC coprocessor

Contents

The results presented in this chapter are from joint works with Aurélien Greuet and Clémence Vermeersch in [GMV22a; GMV22b].

In Chapter 3 and in the previous works mentioned in this chapter [AHH+19; WGY20; BRvV22], the polynomial multiplication is performed using the Kronecker substitution or variants of this algorithm. These algorithms allow to convert a polynomial multiplication to an integer one. Then, the operations are performed on integers which allows to repurpose current asymmetric coprocessor. In $R_{q,\delta} = \mathbb{Z}_q[X]/(X^N + \delta)$, where $\delta \in \{-1, 1\}$, these algorithms can be summarized in four steps:

1. Convert polynomials in $R_{q,\delta}$ to integers in $\mathbb{N}$ of bit size `bitsize`. When polynomials have coefficients with a negative representation, this conversion requires additional operations; see Section 2.2.1 page 19.

2. Modular integer multiplication modulo $2^{\texttt{bitsize}} + \delta$ of the obtained integers. The modular reduction ensures that after Step 3 the polynomial result is reduced modulo $X^N + \delta$. In Chapter 3, we focus on the integer multiplication without modular reduction. However, we claim that this reduction can be done on top of the algorithms that we have introduced.

3. Convert back integer multiplication result to a polynomial in $\mathbb{Z}[X]/(X^N + \delta)$. Like Step 1, if the initial polynomials have coefficients with a negative representation this conversion requires additional operations; see Section 2.2.1 page 19.

4. Reduce the coefficients modulo q to have result over $R_{q,\delta}$.

The previous works re-purpose the coprocessor only to optimize Step 2. All the other steps are implemented in software without the use of coprocessor instruction.

In this chapter, we purse the previous works and we repurpose the coprocessor for most of the previous operations. Our work focuses on three main contributions:

- Introduction of a new modular reduction (Quotient Approximation Reduction). This modular reduction finds application in lattice-based cryptography; see Section 4.1.
- Handle negative evaluation and radix conversion using RSA/ECC coprocessor (Step 1 and 3); see Section 4.2.
- Perform modular reduction of the coefficients modulo q with a RSA/ECC coprocessor (Step 4); see Section 4.2.

The last two improvements are possible only if the coprocessor can handle the following integer operations: addition/subtraction, logical `AND`, logical shift, multiplication and modular multiplication. Except the logical `AND` operation, most of current asymmetric coprocessors handle these operations. The logical `AND` is less common on the current RSA/ECC coprocessor. However adding this operation to an existing architecture is easier and cheaper than designing a new one for polynomial multiplication.

4.1 Quotient Approximation Modular Reduction

In this section we introduce a modular reduction named Quotient Approximation Reduction. For this, in the following, we suppose that we want to reduce a coefficient modulo q. However, compared to the previous chapter we are not in the context of repurposing the coprocessor. Later on in Section 4.2, we adapt this reduction to be performed during the Kronecker substitution.

4.1.1 Context and background

Throughout this section, all integers are considered as non-negative ones.

Notations, definitions

Logical operations We denote by:

- "`>>`" the logical right shift operation: let a be an integer of bit-length ℓ, so that $a = \sum_{i=0}^{\ell-1} a_i \cdot 2^i$. Then for $0 \leq s \leq \ell - 1$, $(a >> s) = \sum_{i=s}^{\ell-1} a_i \cdot 2^{i-s}$. If $s \geq \ell$, then $(a >> s) = 0$.
- "`<<`" the logical left shift operation.
- "`&`" the logical bitwise `AND`.

Integer and fractional parts Given a non-negative real number x, the integer part of x, that is the largest integer $\leq x$, is denoted by $\lfloor x \rfloor$. Its fractional part $x - \lfloor x \rfloor$ is denoted by $\{x\}$.

For the sequel, let $q \geq 2$ be a fixed integer.

Modular Reduction Given $a \in \mathbb{N}$, the modular reduction of a modulo q is the integer $r \in \{0, 1, \ldots, q-1\}$ such that $a = \lfloor a/q \rfloor \cdot q + r$. It is denoted by $a \mod q$.

Partial Modular Reduction Given $a > 2 \cdot q$, a *partial reduction* of a modulo q is an integer a' such that $a' < a$ and $a' \mod q = a \mod q$.

Problem Statement and Motivations

We consider the following problems:

1. Given a fixed modulus q and a bound ℓ, compute a *modular reduction* of any input of bit-length $\leq \ell$.
2. Given a fixed modulus q and a bound ℓ, compute a *partial reduction* of any input of bit-length $\leq \ell$.

Solving efficiently these two problems is motivated by lattice-based post-quantum cryptography. In these cryptosystems, polynomials with coefficients modulo q, where q is fixed and generally fits in a machine word, are multiplied (see e.g. [BDK+18; BDL+21; BMD+21; CDH+19]). Hence, the modular multiplication between coefficients is a core operation.

Most schemes work with a modulus q such that polynomial multiplication is computed using the Number Theoretic Transform (see e.g. [PG12]). After a transformation to the NTT domain, polynomial multiplication is handled by point-wise multiplication. Finally, the result is transformed back from the NTT domain. In this setting, the conversion to and from Montgomery representation can be done for free in the transformations to and from the NTT domain, see e.g. the reference implementations of [BDK+18; BDL+21]. Thus, the point-wise modular multiplications are naturally handled with Montgomery multiplication.

However, some specific devices like smartcards may have a slow CPU multiplication, while having a coprocessor handling large integers multiplication. In this context, it can be faster to transform the polynomial multiplication into a large integer one, thanks to the Kronecker substitution. Then, the large integer arithmetic is handled by the coprocessor, as presented in [AHH+19; BRvV22; GMR21]. Nevertheless, it computes the polynomial multiplication over the integers. Hence, each coefficient must be reduced modulo q. Thus, solving efficiently Problem 1 is of interest in this context.

In general, only the final values have to be fully reduced. Then, it can be sufficient to just control the size of intermediate values, to avoid overflows. In this case, efficient algorithms to solve Problem 2 are adapted.

Contributions

Let a and q be non negative integers, let ℓ be an upper bound on the bit-length of a. We provide an algorithm that computes an approximation of $\lfloor a/q \rfloor$ as a sum of $(a >> j)$'s in Section 4.1.2.

A partial reduction modulo q is deduced from this algorithm in Section 4.1.2. Such a partial reduction is a $a \mod q + t \cdot q$, for a non-negative integer t. We prove that $t \leq \sum_{i=0}^{\ell-1} \left\{ \frac{2^i}{q} \right\}$.

In addition, a *relaxed* version is given in Section 4.1.2. It follows the same idea as the previous partial reduction, with a lower accuracy, leading to a faster algorithm.

In Section 4.1.2, we show that a full reduction can easily be obtained from the partial ones, e.g. by performing a standard division algorithm. A standard division algorithm leads to $a \mod q$, from $a \mod q + t \cdot q$, with at most $\lfloor \log(t) \rfloor + 1$ subtractions.

Section 4.1.3 is devoted to a use case study. Coming from post-quantum cryptography, the modulus q is such that $\sum_{i=0}^{\ell-1} \left\{ \frac{2^i}{q} \right\}$ is small. We analyse the cost of our reduction and compare it to prior art algorithms.

State of the Art

The division is a costly operation on some CPU. In order to reduce the cost of this operation, the division can be achieved by a multiplication with an integer constant. One of the first uses of this technique is done by Jacobsohn in [Jac73] (1973) or Granlund and Montgomery in [GM94] (1994). In the context of modular reduction, this techniques can be used to compute the required quotient or an approximation of it.

In [BSJ14] (Section 3.1) the authors introduce a modular reduction which approximates the quotient using shifts and additions. This approximation is described for three use cases. The modular reduction presented in this section is equivalent in terms of complexity. However, our work describes a generic modular reduction. Then, our work can be viewed as a generalization.

In this part we present standard modular reduction algorithms. The first two work with any modulus. The two others are designed for modulus with a special shape. For a more complete bibliography on modular reduction, see [GG03; BZ10] and references therein.

Montgomery Reduction Montgomery Reduction is introduced in [Mon85]. Given a modulus q, a radix R coprime to q and an integer $0 \leq a < R \cdot q$ to reduce, it computes $\tilde{a} = a \cdot R^{-1} \mod q$. Montgomery multiplication is a combination of a multiplication and a Montgomery Reduction.

To get $a \mod q$ from the Montgomery reduction $\tilde{a}$, either a pre-computation (input $a \cdot R \mod q$ instead of a) or a post-computation (output $\tilde{a} \cdot R \mod q$ instead of $\tilde{a}$) has to be done.

When several Montgomery multiplications are performed, the relative cost of pre or post-computation becomes negligible. In the context of NTT multiplication, even if only one multiplication by coefficient is done, pre and post-computations can be mixed into the transformations to and from the NTT domain, so that their cost becomes free.

However in our context, we assume that the value to reduce comes from a Kronecker substitution followed by a large integer multiplication. Thus, only one reduction is performed, so that the cost of pre or post-computation, that requires another reduction modulo q, remains significant. Hence, we consider that Montgomery reduction and Montgomery multiplication are out of scope for this work.

Barrett Reduction Barrett reduction, introduced in [Bar86] and described in a more modern way in [MVV18], is a partial reduction algorithm that does not require any assumption on the modulus. While it is often presented assuming that the value to reduce modulo q is less than q^2, we give here a more general presentation.

Let q and a be integers, let $k = \texttt{bitlen}(q)$. Barrett reduction computes a partial reduction of a modulo q as follow: let ℓ be an integer such that $2^k < a < 2^\ell$ and let $m = \left\lfloor 2^\ell / q \right\rfloor$. Then a partial reduction of a is given by $a' = a - ((a \cdot m) >> \ell) \cdot q$. In addition, a' is either the modular reduction $a \mod q$ or $a \mod q + q$.

During Barrett reduction, the computation $a \cdot m$ can exceed the size of a register. Therefore, additional operation are required to handle this temporary result on 2 registers. However, some variants of Barrett algorithm allow to limit the size of temporary variables [MVV18; Kon10]. The idea is to perform Barrett reduction using information only on the highest bits of a instead of the whole bits. To do so:

- The precomputed value: $m = \left\lfloor 2^{k+\alpha} / q \right\rfloor$
- Barrett reduction: $a' = a - q \cdot \texttt{quo}$ where $\texttt{quo} = [(a >> (k+\beta)) \cdot m] >> (\alpha - \beta)$ and α, β are integers.

This variant is a trade-off between temporary variables size and final subtraction requirement. Indeed, this partial reduction requires at best one final subtraction at cost of a slight increase in bit size or several final subtractions without increasing the bit size.

Generalized Mersenne Number Reduction Generalized Mersenne numbers are integers of the form $q = f\left(2^k\right)$, where f is a polynomial with small coefficients and low degree. A method to compute their modular reduction is presented in [Sol99; Sol11].

It is not a general modular reduction algorithm. Instead, it allows to find, given a modulus q, a sequence of operations leading to the modular reduction. These operations are logical shifts, logical `AND`, modular additions and subtractions.

As an illustration, we instantiate the first example in [Sol99] with $k = 5$: the reduction of $a < 2^{30}$ modulo $q = 2^{15} - 2^5 + 1$ is given by $a \mod q = T + S_1 + S_2 - D_1 - D_2$, where $+$ and $-$ are modular operations and $T = a \,\&\, \texttt{0x7FFF}$, $S_1 = (a >> 10) \,\&\, \texttt{0x7FE0}$, $S_2 = (a >> 20) \,\&\, \texttt{0x3E0}$, $D_1 = (a >> 15)$ and $D_2 = (a >> 25)$.

We refer to [Sol99; Sol11] for the general presentation of the method.

Pseudo Mersenne Number Reduction Pseudo Mersenne Numbers are integers of the form $q = 2^k - c$, where c is "small". An algorithm to computhe their modular reduction is introduced by Crandall in the patent [Cra27].

Let $q = 2^k - c$, with $c < 2^{k-1}$. This reduction relies on the identity $2^k = c \mod q$: if $a = a_1 2^k + a_0$ then $a \mod q = c \times a_1 + a_0 \mod q$. Hence, a recursive computation of $c \times a_1 + a_0$ is done, until the result is fully reduced.

Algorithm 10 CRANDALL

Input: $a, q = 2^k - c$
1: **while** $a \geq 2q$ **do**
2: $\quad a_0 = a \,\&\, (2^k - 1)$
3: $\quad a_1 = a >> k$
4: $\quad a = c \times a_1 + a_0$
5: **end while**
6: **return** a

In the following we present a modular reduction named Quotient Approximation Reduction and afterwards we compare it complexity with the previous algorithms in the case of post-quantum cryptography.

4.1.2 Quotient Approximation Reduction

Overview

Let q and a be integers, let $\ell = \texttt{bitlen}\,(a)$, so that $a = \sum_{i=0}^{\ell-1} a_i \cdot 2^i$, where a_i is the i-th bit of a.

The Quotient Approximation aims to compute efficiently a value close to $\lfloor a/q \rfloor$. Since $a \mod q = a - \lfloor a/q \rfloor \cdot q$, this is a first step to a partial reduction.

A first approximation is to compute $\sum_{i=0}^{\ell-1} a_i \cdot \left\lfloor \frac{2^i}{q} \right\rfloor$ rather than $\lfloor a/q \rfloor$. This sum is expected to be "close" to $\lfloor a/q \rfloor$. Indeed,

$$\left\lfloor \frac{a}{q} \right\rfloor = \left\lfloor \sum_{i=0}^{\ell-1} a_i \cdot \frac{2^i}{q} \right\rfloor = \left\lfloor \sum_{i=0}^{\ell-1} a_i \cdot \left\lfloor \frac{2^i}{q} \right\rfloor + \sum_{i=0}^{\ell-1} a_i \cdot \left\{ \frac{2^i}{q} \right\} \right\rfloor = \sum_{i=0}^{\ell-1} a_i \cdot \left\lfloor \frac{2^i}{q} \right\rfloor + \left\lfloor \sum_{i=0}^{\ell-1} a_i \cdot \left\{ \frac{2^i}{q} \right\} \right\rfloor,$$

where by definition, for each i, $\{2^i/q\} < 1$.

The sum $\sum_{i=0}^{\ell-1} a_i \cdot \left\lfloor \frac{2^i}{q} \right\rfloor$ is then computed replacing the computation of the $\lfloor 2^i/q \rfloor$'s with the computation of some $(a >> j)$'s, in order to avoid divisions. The idea is that if $\texttt{bitlen}\,(q) = k$ and q is "close to" a power of two, then $2^i/q$ is "close to" $2^i >> k$.

Section 4.1.2 is devoted to the computation of $\sum a_i \cdot \lfloor 2^i/q \rfloor$ as a sum of $(a >> j)$'s. Then, a partial reduction algorithms relying on this quotient approximation are presented in Section 4.1.2. In Section 4.1.2, two methods to get a modular reduction from the partial reduction are given. Finally, we present in Section 4.1.3 some specific usecases where our method is more efficient than the state of the art.

Computing $\sum a_i \cdot \lfloor 2^i/q \rfloor$ with shifts and adds only

Let q and a be integers, $k = \texttt{bitlen}(q)$ and $\ell = \texttt{bitlen}(a)$. Since the computation of $\lfloor 2^i/q \rfloor$ without division is straightforward if q is a power of 2, we assume in the sequel that q is not a power of 2. Let a_i be the i-th bit of a, so that $a = \sum_{i=0}^{\ell-1} a_i \cdot 2^i$. Notice that for each j, $\lfloor 2^j/q \rfloor$ is either $2 \cdot \lfloor 2^{j-1}/q \rfloor$ or $2 \cdot \lfloor 2^{j-1}/q \rfloor + 1$.

Let J_ℓ be the set of integers $J_\ell = \{1 \leq j \leq \ell - 1, \lfloor 2^j/q \rfloor = 2 \cdot \lfloor 2^{j-1}/q \rfloor + 1\}$.

Remark 2. *The smallest element of J_ℓ is $k = \texttt{bitlen}(q)$.*

Lemma 1. *Let i_0, i_1 such that $]i_0, i_1] \cap J_\ell = \emptyset$. Then $\lfloor 2^{i_1}/q \rfloor = 2^{i_1 - i_0} \cdot \lfloor 2^{i_0}/q \rfloor$.*

Proof. Since $]i_0, i_1] \cap J_\ell = \emptyset$, for any $\ell \in]i_0, i_1]$, $\left\lfloor 2^\ell/q \right\rfloor = 2 \cdot \left\lfloor 2^{\ell-1}/q \right\rfloor$. Hence,

$$\left\lfloor 2^{i_1}/q \right\rfloor = 2 \cdot \left\lfloor 2^{i_1-1}/q \right\rfloor = 2^2 \cdot \left\lfloor 2^{i_1-2}/q \right\rfloor = \cdots = 2^{i_1-i_0} \cdot \left\lfloor 2^{i_1-(i_1-i_0)}/q \right\rfloor = 2^{i_1-i_0} \cdot \left\lfloor 2^{i_0}/q \right\rfloor.$$

□

Proposition 5. *Let $J_\ell = \{1 \leq j \leq \ell - 1, \lfloor 2^j/q \rfloor = 2 \cdot \lfloor 2^{j-1}/q \rfloor + 1\}$. For all a such that $\texttt{bitlen}(a) \leq \ell$,*

$$\sum_{i=k}^{\ell-1} a_i \left\lfloor 2^i/q \right\rfloor = \sum_{j \in J_\ell} (a >> j).$$

Proof. Let t be the cardinal of J_ℓ and let $j_1 = k \leq j_2 \leq \cdots \leq j_t$ be its elements. We prove the following statement by backward induction: for all $2 \leq s \leq t$,

$$\sum_{i=j_s}^{\ell-1} a_i \left\lfloor 2^i/q \right\rfloor = \sum_{i=j_s}^{\ell-1} a_i \cdot 2^{i-j_{s-1}} \left\lfloor 2^{j_{s-1}}/q \right\rfloor + \sum_{j \in J_\ell,\, j \geq j_s} (a >> j). \tag{4.1}$$

First, let's prove the case $s = t$. Applying Lemma 1 with $i_0 = j_t$ and $i_1 = i$ to $\lfloor 2^i/q \rfloor$ leads to

$$\sum_{i=j_t}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor = \sum_{i=j_t}^{\ell-1} a_i \cdot 2^{i-j_t} \cdot \left\lfloor 2^{j_t}/q \right\rfloor.$$

Since $j_t \in J_\ell$, $\lfloor 2^{j_t}/q \rfloor = 2 \cdot \lfloor 2^{j_t-1}/q \rfloor + 1$. Replacing in the above equation gives

$$\sum_{i=j_t}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor = \sum_{i=j_t}^{\ell-1} a_i \cdot 2^{i-j_t} \cdot \left(2 \cdot \left\lfloor 2^{j_t-1}/q \right\rfloor + 1\right) = \sum_{i=j_t}^{\ell-1} a_i \cdot 2^{i-j_t+1} \cdot \left\lfloor 2^{j_t-1}/q \right\rfloor + \sum_{i=j_t}^{\ell-1} a_i \cdot 2^{i-j_t}. \tag{4.2}$$

In the first term of the right-hand side, $\lfloor 2^{j_t-1}/q \rfloor$ can be replaced with $2^{j_t - j_{t-1} - 1} \lfloor 2^{j_{t-1}}/q \rfloor$, using Lemma 1 with $i_0 = j_{t-1}$ and $i_1 = j_t - 1$.

In addition, the second term of the right-hand side is $(a >> j_t)$. Since j_t is the greatest element in J_ℓ, $(a >> j_t)$ can be written $\sum_{j \in J_\ell,\, j \geq j_t} (a >> j)$.

Reporting these equalities in Equation 4.2 leads to

$$\begin{aligned}\sum_{i=j_t}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor &= \sum_{i=j_t}^{\ell-1} a_i \cdot 2^{i-j_t+1} \cdot 2^{j_t-j_{t-1}-1} \left\lfloor 2^{j_{t-1}}/q \right\rfloor + \sum_{j\in J_\ell,\, j\geq j_t} (a >> j) \\ &= \sum_{i=j_t}^{\ell-1} a_i \cdot 2^{i-j_{t-1}} \left\lfloor 2^{j_{t-1}}/q \right\rfloor + \sum_{j\in J_\ell,\, j\geq j_t} (a >> j)\,,\end{aligned}$$

that proves the case $s = t$.

We now assume that the induction hypothesis (Equation 4.1) is true for a given s and we prove it also holds for $s-1$. We split the sum $\sum_{i=j_{s-1}}^{\ell-1} a_i \cdot \lfloor 2^i/q \rfloor$ in two parts to get

$$\begin{aligned}\sum_{i=j_{s-1}}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor &= \sum_{i=j_{s-1}}^{j_s-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor + \sum_{i=j_s}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor \\ &= \sum_{i=j_{s-1}}^{j_s-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor + \left(\sum_{i=j_s}^{\ell-1} a_i \cdot 2^{i-j_{s-1}} \left\lfloor 2^{j_{s-1}}/q \right\rfloor + \sum_{j\in J_\ell,\, j\geq j_s} (a >> j) \right), \qquad (4.3)\end{aligned}$$

where the second equality comes from the induction hypothesis.

Applying again Lemma 1, with $i_0 = j_{s-1}$ and $i_1 = i$, one gets

$$\sum_{i=j_{s-1}}^{j_s-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor = \sum_{i=j_{s-1}}^{j_s-1} a_i \cdot 2^{i-j_{s-1}} \left\lfloor 2^{j_{s-1}}/q \right\rfloor,$$

thus, replacing this expression in Equation 4.3,

$$\begin{aligned}\sum_{i=j_{s-1}}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor &= \sum_{i=j_{s-1}}^{j_s-1} a_i \cdot 2^{i-j_{s-1}} \left\lfloor 2^{j_{s-1}}/q \right\rfloor + \left(\sum_{i=j_s}^{\ell-1} a_i \cdot 2^{i-j_{s-1}} \left\lfloor 2^{j_{s-1}}/q \right\rfloor + \sum_{j\in J_\ell,\, j\geq j_s} (a >> j) \right) \\ &= \sum_{i=j_{s-1}}^{\ell-1} a_i \cdot 2^{i-j_{s-1}} \left\lfloor 2^{j_{s-1}}/q \right\rfloor + \sum_{j\in J_\ell,\, j\geq j_s} (a >> j)\,. \qquad (4.4)\end{aligned}$$

Since $j_{s-1} \in J_\ell$, $\lfloor 2^{j_{s-1}}/q \rfloor = 2 \cdot \lfloor 2^{j_{s-1}-1}/q \rfloor + 1$. Hence,

$$\begin{aligned}\sum_{i=j_{s-1}}^{\ell-1} a_i \cdot 2^{i-j_{s-1}} \left\lfloor 2^{j_{s-1}}/q \right\rfloor &= \sum_{i=j_{s-1}}^{\ell-1} a_i \cdot 2^{i-j_{s-1}} \left(2 \cdot \left\lfloor 2^{j_{s-1}-1}/q \right\rfloor + 1 \right) \\ &= \sum_{i=j_{s-1}}^{\ell-1} a_i \cdot 2^{i-j_{s-1}+1} \cdot \left\lfloor 2^{j_{s-1}-1}/q \right\rfloor + \sum_{i=j_{s-1}}^{\ell-1} a_i \cdot 2^{i-j_{s-1}}\end{aligned}$$

According to Lemma 1 applied with $i_0 = j_{s-2}$ and $i_1 = j_{s-1} - 1$, $\lfloor 2^{j_{s-1}-1}/q \rfloor = 2^{j_{s-1}-1-j_{s-2}} \lfloor 2^{j_{s-2}}/q \rfloor$, thus above equation becomes

$$\begin{aligned}\sum_{i=j_{s-1}}^{\ell-1} a_i \cdot 2^{i-j_{s-1}} \left\lfloor 2^{j_{s-1}}/q \right\rfloor &= \sum_{i=j_{s-1}}^{\ell-1} a_i \cdot 2^{i-j_{s-1}+1} \cdot 2^{j_{s-1}-1-j_{s-2}} \left\lfloor 2^{j_{s-2}}/q \right\rfloor + \sum_{i=j_{s-1}}^{\ell-1} a_i \cdot 2^{i-j_{s-1}} \\ &= \sum_{i=j_{s-1}}^{\ell-1} a_i \cdot 2^{i-j_{s-2}} \cdot \left\lfloor 2^{j_{s-2}}/q \right\rfloor + \sum_{i=j_{s-1}}^{\ell-1} a_i \cdot 2^{i-j_{s-1}}.\end{aligned}$$

Replacing above equation in Equation 4.4 and noticing that $\sum_{i=j_{s-1}}^{\ell-1} a_i \cdot 2^{i-j_{s-1}} = a >> j_{s-1}$, it comes

$$\begin{aligned}\sum_{i=j_{s-1}}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor &= \sum_{i=j_{s-1}}^{\ell-1} a_i \cdot 2^{i-j_{s-2}} \cdot \left\lfloor 2^{j_{s-2}}/q \right\rfloor + (a >> j_{s-1}) + \sum_{j \in J_\ell, j \geq j_s} (a >> j) \\ &= \sum_{i=j_{s-1}}^{\ell-1} a_i \cdot 2^{i-j_{s-2}} \cdot \left\lfloor 2^{j_{s-2}}/q \right\rfloor + \sum_{j \in J_\ell, j \geq j_{s-1}} (a >> j)\,,\end{aligned}$$

that proves the backward induction step.

To conclude, we prove the statement of the proposition, starting from

$$\sum_{i=j_1}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor = \sum_{i=j_1}^{j_2-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor + \sum_{i=j_2}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor . \tag{4.5}$$

According to Lemma 1 applied with $i_0 = j_1 = k$ and $i_1 = i$, the first term of the right-hand side can be written

$$\sum_{i=j_1}^{j_2-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor = \sum_{i=j_1}^{j_2-1} a_i \cdot 2^{i-j_1} \left\lfloor 2^{j_1}/q \right\rfloor .$$

The second term is re-written thanks to Equation 4.1 for $s = 2$, that is true from the previous backward induction:

$$\sum_{i=j_2}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor = \sum_{i=j_2}^{\ell-1} a_i \cdot 2^{i-j_1} \left\lfloor 2^{j_1}/q \right\rfloor + \sum_{j \in J_\ell, j \geq j_2} (a >> j)\,.$$

Replacing these two expressions in Equation 4.5 leads to:

$$\begin{aligned}\sum_{i=j_1}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor &= \sum_{i=j_1}^{j_2-1} a_i \cdot 2^{i-j_1} \left\lfloor 2^{j_1}/q \right\rfloor + \sum_{i=j_2}^{\ell-1} a_i \cdot 2^{i-j_1} \left\lfloor 2^{j_1}/q \right\rfloor + \sum_{j \in J_\ell, j \geq j_2} (a >> j) \\ &= \sum_{i=j_1}^{\ell-1} a_i \cdot 2^{i-j_1} \left\lfloor 2^{j_1}/q \right\rfloor + \sum_{j \in J_\ell, j \geq j_2} (a >> j)\end{aligned}$$

Since $j_1 = k$, $\lfloor 2^{j_1}/q \rfloor = 1$, so that the first term of above right-hand side is $\sum_{i=j_1}^{\ell-1} a_i \cdot 2^{i-j_1}$, that is $(a >> j_1)$.

This means that

$$\sum_{i=j_1}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor = (a >> j_1) + \sum_{j \in J_\ell, j \geq j_2} (a >> j) = \sum_{j \in J_\ell, j \geq j_1} (a >> j) = \sum_{j \in J_\ell} (a >> j)$$

□

Proposition 5 allows to compute, without division, an approximation of the quotient $\lfloor a/q \rfloor$. In the next section, we deduce a partial reduction algorithm, based on this quotient approximation.

Partial Modular Reductions

Let q be a modulus of bit-length k, let ℓ be the maximum bit-length of the numbers to reduce. Let J_ℓ be the set of integers $J_\ell = \{1 \leq j \leq \ell - 1, \lfloor 2^j/q \rfloor = 2 \cdot \lfloor 2^{j-1}/q \rfloor + 1\}$.

Quotient Approximation Partial Reduction. The following algorithm computes a partial reduction modulo q of any input a of bit-length at most ℓ and such that $a \geq 2^k$.

Algorithm 11 QAPartialRed(a, q, J_ℓ): Quotient Approximation Partial Reduction

Input: $a = \sum_{i=0}^{\ell-1} a_i \cdot 2^i \geq 2^k$, q, J_ℓ defined as above
Output: $r = a - \left(\sum_{i=k}^{\ell-1} a_i \cdot \lfloor 2^i/q \rfloor\right) \cdot q$
1: `quo_approx` $\leftarrow 0$
2: **for** each $j \in J_\ell$ **do**
3: `quo_approx` $\leftarrow$ `quo_approx` $+ (a >> j)$
4: **end for**
5: $r \leftarrow a -$ `quo_approx` $\cdot q$
6: **return** r

Proposition 6. *Algorithm 11 is correct and it outputs a partial reduction of a modulo q, that is $r < a$ and $r \mod q \equiv a \mod q$.*

Proof. It is clear that at the end of the inner loop, `quo_approx` $= \sum_{j \in J_\ell} (a >> j)$. According to Proposition 5, $\sum_{j \in J_\ell} (a >> j) = \sum_{i=k}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor$. Hence, at the end of the algorithm,

$$r = a - \left(\sum_{i=k}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor\right) \cdot q,$$

so that it outputs the expected value.

In addition, `quo_approx` $\neq 0$: since $a \geq 2^k$, at least one index $i \geq k$ is such that a's i-th bit a_i is not 0. Thus, the sum `quo_approx` $= \sum_{i=k}^{\ell-1} a_i \cdot \lfloor 2^i/q \rfloor$ is necessarily non-zero. Hence, $r = a -$ `quo_approx` $\cdot q < a$. As the subtraction of a and a multiple of q, $r \mod q \equiv a \mod q$, and r is a partial reduction of a modulo q. □

The following Proposition gives a bound on the difference between the modular reduction $a \mod q$ and the partial reduction given by Algorithm 11.

Proposition 7. *Let r be the output of Algorithm 11 for an input a of bit-length ℓ. Then*

$$0 \leq r - (a \mod q) \leq \left\lfloor \sum_{i=0}^{\ell-1} \left\{ \frac{2^i}{q} \right\} \right\rfloor \cdot q.$$

Proof. Let a be a ℓ-bit integer and r the corresponding output of Algorithm 11. According to Proposition 6, denoting by a_i the i-th bit of a,

$$r = a - \left(\sum_{i=k}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor\right) \cdot q. \tag{4.6}$$

Moreover,

$$\left\lfloor \frac{a}{q} \right\rfloor = \left\lfloor \sum_{i=0}^{\ell-1} a_i \cdot \frac{2^i}{q} \right\rfloor = \left\lfloor \sum_{i=0}^{\ell-1} a_i \cdot \left\lfloor \frac{2^i}{q} \right\rfloor + \sum_{i=0}^{\ell-1} a_i \cdot \left\{ \frac{2^i}{q} \right\} \right\rfloor = \sum_{i=0}^{\ell-1} a_i \cdot \left\lfloor \frac{2^i}{q} \right\rfloor + \left\lfloor \sum_{i=0}^{\ell-1} a_i \cdot \left\{ \frac{2^i}{q} \right\} \right\rfloor$$

Or $\sum_{i=0}^{l-1} a_i \cdot \left\lfloor \frac{2^i}{q} \right\rfloor = 0$, then

$$\left\lfloor \frac{a}{q} \right\rfloor = \sum_{i=k}^{\ell-1} a_i \cdot \left\lfloor \frac{2^i}{q} \right\rfloor + \left\lfloor \sum_{i=0}^{\ell-1} a_i \cdot \left\{ \frac{2^i}{q} \right\} \right\rfloor \tag{4.7}$$

Since $a \mod q = a - \lfloor a/q \rfloor \cdot q$, using it follows from Equations 4.6 and 4.7 that

$$\begin{aligned} r - (a \mod q) &= \left(a - \left(\sum_{i=k}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor\right) \cdot q\right) - (a - \lfloor a/q \rfloor \cdot q) \\ &= \left(\lfloor a/q \rfloor - \sum_{i=k}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor\right) \cdot q \\ &= \left(\sum_{i=k}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor + \left\lfloor \sum_{i=0}^{\ell-1} a_i \cdot \left\{2^i/q\right\} \right\rfloor - \sum_{i=k}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor\right) \cdot q \\ &= \left\lfloor \sum_{i=0}^{\ell-1} a_i \cdot \left\{2^i/q\right\} \right\rfloor \cdot q \leq \left\lfloor \sum_{i=0}^{\ell-1} \left\{2^i/q\right\} \right\rfloor \cdot q, \end{aligned}$$

the last inequation holding because each a_i is a bit, so that $a_i \leq 1$. It is also clear that $\left\lfloor \sum_{i=0}^{\ell-1} a_i \cdot \{2^i/q\} \right\rfloor \geq 0$, as a sum of non-negative elements, thus so is $r - (a \mod q)$. □

Remark 3. *Practically, when working with a fixed modulus, it is more efficient to "unroll" the loop in Algorithm 11, as shown in the next toy example.*

Example 6. *Let $q = 2^4 - 2 = 14$, so that $k = 4$. Let $\ell = 10$ be the maximum bit-length of the number to reduce. The table*

i	0	1	2	3	4	5	6	7	8	9
$\lfloor 2^i/q \rfloor$	0	0	0	0	1	2	4	9	18	36
$\{2^i/q\}$	1/14	1/7	2/7	4/7	1/7	2/7	4/7	1/7	2/7	4/7

ensures that $J_8 = \{4, 7\}$. Then, Algorithm 11 can be written:

- `quo_approx` $\leftarrow (a >> 4) + (a >> 7)$
- *return* $a -$ `quo_approx` $\cdot\, q$

In addition, Proposition 7 ensures that for any 10-bit integer a, the corresponding result r is such that:

$$0 \leq r - (a \mod q) \leq \left\lfloor \sum_{i=0}^{10-1} \left\{\frac{2^i}{q}\right\} \right\rfloor \cdot q = \left\lfloor \frac{43}{14} \cdot q \right\rfloor = 3 \cdot q.$$

Finally, we give the number of operations to perform the Quotient Approximation partial reduction.

Proposition 8. *Let q be a modulus of bit-length k, let ℓ be the maximum bit-length of the numbers to reduce. Let J_ℓ be the set of integers $J_\ell = \{1 \leq j \leq \ell - 1, \lfloor 2^j/q \rfloor = 2 \cdot \lfloor 2^{j-1}/q \rfloor + 1\}$ and let $n = \#J_\ell$ be its cardinality. Then for all a of bit-length at most ℓ, $\mathsf{QAPartialRed}(a, q, J_\ell)$ is computed with*

- *n right shifts on ℓ-bit elements,*
- *$n - 1$ additions of $(\ell - k)$-bit elements,*
- *1 multiplication between an element of bit-length at most $\ell - k + 1$ and one k-bit element,*
- *1 subtraction between two elements of size at most ℓ.*

Proof. The for loop computes the sum of n elements, each element being a $(a >> j)$ with $j \geq k$. Hence, this can be done with n shifts on ℓ-bit elements and $n - 1$ additions on $(\ell - k)$-bit elements. Since `quo_approx` $\leq \lfloor a/q \rfloor$, it has bit-length at most $\ell - k + 1$.

Thus, the final step $a -$ `quo_approx` $\cdot\, q$ requires 1 multiplication between an element of bit-length at most $\ell - k + 1$ and one k-bit element. Since the result is non-negative according to Proposition 7, the result of the multiplication is necessarily of bit-length at most ℓ. The subtraction is then between two elements of size at most ℓ. □

Quotient Approximation Partial Reduction Relaxed. The Algorithm 11 determines an approximation of the quotient by iterating over all the elements of J_ℓ. However in some cases we do not need such a precision for the approximated quotient. Hence, one can perform the Algorithm 11 in a subset of J_ℓ.

Let $J_\ell = J'_\ell \cup \bar{J}'_\ell$ where $J'_\ell \cap \bar{J}'_\ell = \emptyset$ and the first element of J_ℓ is included in J'_ℓ. The Algorithm 12 computes a partial reduction modulo q of any input a of bit-length at most ℓ such that $a \geq 2^k$ and the inner loop iterates in $J'_\ell \subset J_\ell$.

Algorithm 12 QAPartialRedRelaxed(a, q, J'_ℓ): Quotient Approximation Partial Reduction Relaxed

Input: $a = \sum_{i=0}^{\ell-1} a_i \cdot 2^i \geq 2^k$, q, J'_ℓ defined as above
Output: $r = a - \left(\sum_{i=k}^{\ell-1} a_i \cdot \lfloor 2^i/q \rfloor + \sum_{j \in \bar{J}'_\ell} \left\lfloor (2^\ell - 1)/2^j \right\rfloor\right) \cdot q$

```
quo_approx ← 0
for each j ∈ J'_ℓ do
    quo_approx ← quo_approx + (a >> j)
end for
r ← a − quo_approx · q
return r
```

Proposition 9. *Algorithm 12 is correct and it outputs a partial reduction of a modulo q, that is $r < a$ and $r \mod q = a \mod q$.*

Proposition 10. *Let r be the output of Algorithm 12 for an input a of bit-length ℓ. Then*

$$0 \leq r - (a \mod q) \leq \left\lfloor \sum_{i=0}^{\ell-1} \left\{ \frac{2^i}{q} \right\} \right\rfloor + \sum_{j \in \bar{J}'_\ell} \left\lfloor (2^\ell - 1)/2^j \right\rfloor \cdot q.$$

Proof. By definition $J_\ell = J'_\ell \cup \bar{J}'_\ell$ where $J'_\ell \cap \bar{J}'_\ell = \emptyset$ and the first element of J_ℓ is included in J'_ℓ. Then,

$$\sum_{j \in J'_\ell} (a >> j) = \sum_{j \in J_\ell} (a >> j) - \sum_{j \in \bar{J}'_\ell} (a >> j) = \sum_{i=k}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor - \sum_{j \in \bar{J}'_\ell} \left\lfloor a/2^j \right\rfloor > 0 \tag{4.8}$$

Hence at the end of Algorithm 12:

$$r = a - \left(\sum_{i=k}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor - \sum_{j \in \bar{J}'_\ell} \left\lfloor a/2^j \right\rfloor \right) \cdot q \leq a - \left(\sum_{i=k}^{\ell-1} a_i \cdot \left\lfloor 2^i/q \right\rfloor - \sum_{j \in \bar{J}'_\ell} \left\lfloor (2^\ell - 1)/2^j \right\rfloor \right) \cdot q \tag{4.9}$$

Using equations 4.8 and 4.9, the proofs of Propositions 9 and 10 can be deduced from the proofs of Propositions 6 and 7. □

The Proposition 8 holds for the set J'_ℓ. A use case of Algorithm 12 is described in Section 4.1.3.

From Partial Reduction to Modular Reduction

In this section, two methods are proposed to get a modular reduction from the Quotient Approximation partial reduction.

Partial Reduction Iterations. A first straightforward way to get a complete modular reduction is to iterate the partial reduction algorithm until the result is $< 2^k$. Up to a final subtraction with q, the result is a full modular reduction. This leads to Algorithm 13.

Algorithm 13 QARed: Quotient Approximation Reduction

Input: $a = \sum_{i=0}^{\ell-1} a_i \cdot 2^i$, q, J_ℓ
Output: $r = a \mod q$

```
r ← a
J ← J_ℓ
while r ≥ 2^k do
    r ← QAPartialRed(r, q, J)
    J ← J_bitlen(r)
end while
if r ≥ q then
    r ← r − q
end if
return r
```

Proposition 11. *Algorithm 13 is correct.*

Proof. Given $r \geq 2^k$, $\mathsf{QAPartialRed}(r)$ is a partial reduction modulo q according to Proposition 6. In particular, $\mathsf{QAPartialRed}(r) < r$. Thus, the while loop ends and $r \mod q = a \mod q$ always holds.

In addition, if $r < 2^k$, then either it is already less than q (and then already $= a \mod q$) or it is between q and $2 \cdot q$. Indeed, q has bit-length equal to k, so that $2 \cdot q \geq 2^k > r$. In the latter case, $0 \leq r - q < q$, thus $r - q = a \mod q$. □

Remark 4. *A closed-form formula for the complexity in the general case seems hard to get and outside the scope of this work. However, for a fixed modulus q, the maximum number of iteration of the while loop can be computed thanks to the bound given in Proposition 7.*

Example 7 (Example 6 continued)**.** *In Example 6, after one iteration, $r - (a \mod q) \leq 3 \cdot q$. Hence, $r \leq (a \mod q) + 3 \cdot q \leq 4 \cdot q = 56 < 2^6$ is at most 6-bit long.*

Then for the second iteration, `quo_approx` $= (r >> 4)$ *and* $r \leq (a \mod q) \cdot q + \left\lfloor \sum_{i=0}^{6-1} \{2^i/q\} \right\rfloor \cdot q \leq 2 \cdot q$, *thus it is at most 5-bit long.*

For the third iteration, `quo_approx` $= (r >> 4)$. *If r was $\geq 2^4$ before this third step, since it was necessarily $\leq 2 \cdot q < 2^5$, then* `quo_approx` $= 1$. *In that case, the new r is $r = r - q \leq q < 2^4$. In the other case,* `quo_approx` $= 0$ *but r was $< 2^4$ already.*

Finally, with the last subtraction, the full algorithm can be unrolled as follow:

1. `quo_approx` $\leftarrow (a >> 4) + (a >> 7)$
2. $r \leftarrow a -$ `quo_approx` $\cdot q$
3. `quo_approx` $\leftarrow (r >> 4)$
4. $r \leftarrow r -$ `quo_approx` $\cdot q$
5. `quo_approx` $\leftarrow (r >> 4)$
6. $r \leftarrow r -$ `quo_approx` $\cdot q$
7. *if* $r \geq q$ *then* $r \leftarrow r - q$
8. *return* r

Division Algorithm. A second way to get a modular reduction from a Quotient Approximation partial reduction is to perform a standard division algorithm until the full reduction is reached.

Algorithm 14 QARedDiv: Quotient Approximation Reduction with Divisions

Input: $a = \sum_{i=0}^{\ell-1} a_i \cdot 2^i$, q, J_ℓ
Output: $r = a \mod q$
1: $r \leftarrow \mathsf{QAPartialRed}(a, q, J_\ell)$
2: $s \leftarrow \texttt{bitlen}(r)$
3: **while** $s \geq k$ **do**
4: **if** $r - (q << (s-k)) \geq 0$ **then**
5: $r \leftarrow r - (q << (s-k))$
6: **end if**
7: $s \leftarrow s - 1$
8: **end while**
9: **return** r

The correctness of Algorithm 14 is straightforward as it is QARedDiv followed by a standard binary division algorithm, that computes the remainder of r in its division by q.

Likewise, according to Proposition 7, $r \leq (a \mod q) + \left\lfloor \sum_{i=0}^{\ell-1} \{2^i/q\} \right\rfloor \cdot q$, that is $r = (a \mod q) + \delta \cdot q$, where $0 \leq \delta \leq \left\lfloor \sum_{i=0}^{\ell-1} \{2^i/q\} \right\rfloor$. Then the while loop actually computes $r - \delta \cdot q$, that is done with $\texttt{bitlen}(\delta)$ iterations with a standard binary algorithm. Hence, at most $\texttt{bitlen}\left(\left\lfloor \sum_{i=0}^{\ell-1} \{2^i/q\} \right\rfloor\right)$ iterations of the while loop are performed.

Remark 5. *Practically, for a fixed modulus, the algorithm is unrolled, as shown in the following example.*

Example 8 (Example 6 continued)**.** *In Example 6, after one iteration,* $r - (a \mod q) \leq 3 \cdot q$. *Hence,* $a \mod q$ *can be recovered from* r *with a most 2 subtractions. The unrolled algorithm is then:*

1. $\texttt{quo\_approx} \leftarrow (a >> 4) + (a >> 7)$
2. $r \leftarrow a - \texttt{quo\_approx} \cdot q$
3. *if* $r - (q << 1) \geq 0$ *then* $r \leftarrow r - (q << 1)$
4. *if* $r - q \geq 0$ *then* $r \leftarrow r - q$
5. *return* r

4.1.3 Application: CRYSTALS-Dilithium

Dilithium [BDL+21] is a lattice-based signature, finalist of the NIST call for Post-Quantum Cryptography [MAA+20]. It relies on multiplication of polynomials of degree 256, modulo $q = 8380417 = 2^{23} - 2^{13} + 1$.

Key Generation We first consider multiplications in Key Generation for Dilithium2. In this context, 2 polynomials of degree $N = 256$ are multiplied. The first one, says f, has coefficients in $[0, q[$ while the second one, say g, has coefficients in $[-\eta, \eta]$, with $\eta = 2$. To deal with non-negative coefficients, one can compute $f \cdot g$ as $f \cdot g^+ - f \cdot g^-$, where g^+ has coefficients in $[0, \eta]$, g^- in $]0, \eta]$ and $g = g^+ - g^-$.

Hence, without loss of generality, we consider the polynomial multiplication between f and a polynomial with coefficients in $[0, \eta]$. Each coefficient a of the result is such that $0 \leq a \leq N \cdot q \cdot \eta < 2^{32}$.

To apply Quotient Approximation reduction with $\ell = 32$, we compute $J_\ell = \{23\}$ and $\left\lfloor \sum_{i=0}^{\ell-1} \{2^i/q\} \right\rfloor = \left\lfloor \frac{12574208}{8380417} \right\rfloor = 1$. It follows that at most 1 subtraction by q is needed to get $a \mod q$ from $\mathsf{QAPartialRed}(a, q, J_\ell)$. Hence, our modular reduction becomes:

1. $\texttt{quo\_approx} \leftarrow (a >> 23)$
2. $r \leftarrow a - \texttt{quo\_approx} \cdot q$
3. if $r - q \geq 0$ then $r \leftarrow r - q$
4. return r

In the following we assess Quotient Approximation Reduction, a variant of Barrett, Crandall and Solinas algorithms in the case of Dilithium parameters.

For the Barrett algorithm, we use the parameters $\alpha = 10, \beta = -2$ and $m = \lfloor 2^{33}/q \rfloor$. These parameters ensure that the temporary variable fits in a register and at most one final subtraction is required.

For Crandall algorithm, the modular reduction is done with one iteration in the "while" loop and a final subtraction.

For Solinas algorithm, the reduction of $a < 2^{32}$ modulo $q = 2^{23} - 2^{13} + 1$ is given by $a \mod q = T + S_1 + S_2 - D$, where $+$ and $-$ are modular operations and $T = a \,\&\, \texttt{7FFFFF}$, $S_1 = (a \,\&\, \texttt{0x7FE0}) >> 10$, $S_2 = (a \,\&\, \texttt{0x3E0}) >> 20$, $D = (a \,\&\, \texttt{0x1ff800000}) >> 15$.

The Table 4.1 describes the operations count for each reduction algorithm. The result of the operations fits in a register.

	Operation on 32 bits				
	Mult.	Add/Sub	Shift	AND	Cond. Sub
$\texttt{Barrett}_{10,-2}$	2	1	2	0	1
Solinas	0	2	2	3	1
Crandall	1	1	1	1	1
QARed	1	1	1	0	1

Table 4.1: Complexity of the reduction algorithms

In this context Quotient Approximation Reduction is more efficient than Barrett and Crandall algorithms. The comparison between Solinas and QARed algorithms depends on the component. Indeed, if a multiplication where both the result and each operand fit in a register costs less than 3 AND, 1 add and 1 shift, then Quotient Approximation Reduction is faster than Solinas. Otherwise Solinas is faster.

Signature In the Dilithium signature computation, one polynomial has coefficients in $[0, q[$. The other one has coefficients in $[0, 2^{\gamma_1}]$, where $\gamma_1 = 2^{17}$ or 2^{19}. Hence, each coefficient a of the result is such that $0 \leq a \leq N \cdot q \cdot 2^{\gamma_1} < 2^{50}$.

Here, for $\ell = 50$, we get $J_\ell = \{23, 33, 44, 45, 46\}$ and $\left\lfloor \sum_{i=0}^{\ell-1} \{2^i/q\} \right\rfloor = 5$. It follows that at most $\texttt{bitlen}(5) = 3$ subtractions by q are needed to get $a \mod q$ from $\mathsf{QAPartialRed}(a, q, J_\ell)$:

1. $\texttt{quo\_approx} \leftarrow (a >> 23) + (a >> 33) + (a >> 44) + (a >> 45) + (a >> 46)$
2. $r \leftarrow a - \texttt{quo\_approx} \cdot q$
3. if $r - (q << 2) \geq 0$ then $r \leftarrow r - (q << 2)$
4. if $r - (q << 1) \geq 0$ then $r \leftarrow r - (q << 1)$
5. if $r - q \geq 0$ then $r \leftarrow r - q$
6. return r

However, one can perform the modular reduction by firstly use QAPartialRedRelaxed and afterwards use QAPartialRed. For $\ell = 50$, we take $J_\ell = \{23, 33, 44, 45, 46\}$, $J'_\ell = \{23, 33\}$ and $\bar{J}'_\ell = \{44, 45, 46\}$. First we apply the algorithm $r \leftarrow \mathsf{QAPartialRedRelaxed}(a, q, J'_\ell)$:

1. $\texttt{quo\_approx} \leftarrow (a >> 23) + (a >> 33)$
2. $r \leftarrow a - \texttt{quo\_approx} \cdot q$
3. return r

Due to Proposition 10 we got

$$r \leq \left\lceil \left\lfloor \sum_{i=0}^{\ell-1} \left\{ \frac{2^i}{q} \right\} \right\rfloor + \sum_{j \in \bar{J}'_\ell} \left\lfloor (2^\ell - 1)/2^j \right\rfloor \right\rceil \cdot q = 114 \cdot q < 2^{32}$$

After this partial reduction our result is lower than 2^{32}. Then, we re-define J_ℓ as $J_\ell = \{23, 33\}$ and by applying, like in the previous key generation example, $\mathsf{QAPartialRed}(r, q, J_\ell)$ and a final subtraction we got the expected reduction. In the following we denote by $\mathsf{QARedRelaxed}$ the combination of these reductions.

As previously, we compare the Quotient Approximation Reduction Algorithm with the Barrett, Crandall and Solinas modular reduction.

For Crandall algorithm, the modular reduction is done with three iterations in the "while" loop and a final subtraction. After the second iteration, the result fits on 32 bits.

For the Barrett algorithm, we use the parameters $\alpha = 28, \beta = -2$ and $m = \lfloor 2^{51}/q \rfloor$. These parameters ensure that the temporary variables are encoded on 58 bits rather than 50 bits but fits on 2 machine words. The reduction requires at most one final subtraction.

For Solinas algorithm, the reduction of $a < 2^{50}$ modulo $q = 2^{23} - 2^{13} + 1$ is given by $a \mod q = T + S_1 + S_2 + S_3 + S_4 - (D_1 + D_2 + D_3 + D_4)$, where $+$ and $-$ are modular operations and $T = a \,\&\, \texttt{0x7fffff}$, $S_1 = (a \,\&\, \texttt{0x1ff800000}) >> 10$, $S_2 = (a \,\&\, \texttt{0x7fe00000000}) >> 20$, $S_3 = (a \,\&\, \texttt{0x7f80000000000}) >> 30$, $S_4 = (a \,\&\, \texttt{0x7c00000000000}) >> 46$, $D_1 = (a \,\&\, \texttt{0x3fffff800000}) >> 23$, $D_2 = (a \,\&\, \texttt{0x7c00000000000}) >> 32$, $D_3 = (a \,\&\, \texttt{0x3ffe00000000}) >> 33$, $D_4 = (a \,\&\, \texttt{0x7f80000000000}) >> 43$. In practice, S_4 and D_4 are computed as $S_4 = D_2 >> 14, D_4 = S_3 >> 13$. Most of the masks are sparse (especially lower bytes), therefore operations can be done on one machine word instead of two.

The Table 4.2 describes the number of operations required for each reduction algorithm. The operations are performed on a machine word (32 bits) or 2 machine words (64 bits). For the multiplication on 64 bits we consider that we multiply two integers encoded on a word and the result is encoded on two machine words.

	Operation on 64 bits				Operation on 32 bits				
	Mult.	Add/Sub	Shift	And	Mult.	Add/Sub	Shift	And	Cond. Sub
$\texttt{Barrett}_{28,-2}$	2	1	2	0	0	0	0	0	1
`Solinas`	0	0	2	2	0	8	6	5	2
`Crandall`	1	1	2	0	2	2	1	3	1
`QARed`	1	1	1	0	0	4	4	0	3
`QARedRelaxed`	1	1	1	0	1	2	2	0	1

Table 4.2: Complexity of the reduction algorithms

In order to compare the reduction algorithms we convert all operations on 64 bits to ones on 32 bits.

- A multiplication with result on 64 bits requires 4 multiplications and 3 additions on 32 bits.
- Add/sub on 64 bits requires 2 add/sub on 32 bits (addition/subtraction of the carry/borrow and the two most/least significant words can generally be performed with a single instruction, at the same cost as a simple add/sub)
- A shift on 64 bits requires 2 shifts and 1 addition on 32 bits. Generally, a shift on 64 bits requires 2 additions on 32 bits rather than 1. However, in our context the shift operation ensures that the result fits on a machine word. Therefore, only one addition is required.
- An `AND` on 64 bits requires 2 `AND` on 32 bits.

The complexities on 32 bits are described in Table 4.3.

	Operation on 32 bits				
	Mult.	Add/Sub	Shift	And	Cond. Sub
`Barrett`$_{28,-2}$	8	10	4	0	1
`Solinas`	0	10	10	9	2
`Crandall`	6	9	5	3	1
`QARed`	4	10	6	0	3
`QARedRelaxed`	5	8	4	0	1

Table 4.3: Complexity with machine word operations of the reduction algorithms

In this context Quotient Approximation Reduction (relaxed version) is more efficient than Barrett and Crandall algorithms. The comparison between Solinas and QARed algorithms depends on the component. Indeed, if 5 multiplications where both the result and each operand fit in a register costs less than 9 `AND`, 2 add, 1 conditional subtraction and 6 shifts, then Quotient Approximation Reduction is faster than Solinas. Otherwise Solinas is faster.

4.2 Modular polynomial multiplication using RSA/ECC coprocessor

In this section we pursue the previous works in [AHH+19; WGY20; BRvV22] and in Chapter 3 to repurpose existing RSA/ECC coprocessor to speed-up modular polynomial multiplication in $R_{q,\delta}$. More specifically, the contribution of this section is to:

- Handle negative evaluation and radix conversion using RSA/ECC coprocessor.
- Perform modular reduction of the coefficients modulo q with a RSA/ECC coprocessor.

These improvements are possible only if the coprocessor can handle the following integer operations: addition/subtraction, bitwise `AND`, logical shift, multiplication and modular multiplication. Except the logical `AND` operation, most of current asymmetric coprocessors handle these operations. The logical `AND` is less common on the current RSA/ECC coprocessor. However adding this operation to an existing architecture is easier and cheaper than designing a new one for polynomial multiplication

For the sake of clarity, in this section we reintroduce some notations or some algorithms already presented in the previous chapters.

4.2.1 Background

Integers representation. Let $a \in \mathbb{N}$ such that $0 \leq a < 2^{\ell}$. In the following, we say that a is represented over ℓ bits to mean that a is stored in a machine buffer of ℓ bits.

Let $b \in \mathbb{Z}$ such that $-2^{\ell'-1} < b < 2^{\ell'-1}$. Let $\tilde{b}$ be the two's complement representation of b over ℓ' bits, defined by:

$$\tilde{b} = 2^{\ell'} + b \mod 2^{\ell'} \in \mathbb{N}$$

In the following, we say that b is represented over ℓ' bits to mean that the two's complement representation of b is stored in a machine buffer of ℓ' bits.

Let r be a $N\ell$-bit natural number. We denote by r_i the i-th digit of r in base 2^{ℓ}. In other words, $r = \sum_{i=0}^{N-1} r_i 2^{i\ell}$ with $0 \leq r_i < 2^{\ell}$. We use the following notation $r = (r_0, r_1, \ldots, r_{N-1})_{\ell}$.

Polynomial representation. Let $F(X) = f_0 + f_1X + \ldots + f_{N-1}X^{N-1} \in \mathbb{Z}[X]$ of degree at most $N-1$. Let $\tilde{f_i}$ be a two's complement representation of a coefficient f_i.

Array representation. The usual machine representation of $F(X)$ is an array where the i-th item is $\tilde{f_i}$. To ease the reading, we denote in the following f_i or $f[i]$ the coefficient associated to the i-th item. Moreover, unless otherwise specified, a polynomial is represented as an array.

Packed integer representation. A packed integer representation of $F(X)$ is the concatenation of all the $\tilde{f_i}$ into a buffer.

$$f = \tilde{f}_{N-1}|\ldots|\tilde{f_1}|\tilde{f_0} \in \mathbb{N}$$

In this work, this representation is used to represent polynomials into a natural number. Afterwards, the polynomial arithmetic is carried out with operations on this natural number.

Rings. Let q be an integer. Denote by $R_{q,\delta}$ the polynomial ring $\frac{\mathbb{Z}_q[X]}{(X^N+\delta)}$, where $\delta \in \{-1, 1\}$. We represent an element $F(X) \in R_{q,\delta}$ as a polynomial of degree at most $N-1$ with coefficients in $\{0, \ldots, q-1\}$. $R^-_{q,\delta}$ denotes the elements of $R_{q,\delta}$ represented by a polynomial of degree at most $N-1$ with coefficients in $\{-\frac{q}{2}-1, \ldots, \frac{q}{2}\}$.

Integer operations. In the sequel, the algorithms are described using the following notations. Their purpose is to clarify the size of the manipulated operands.

- Let **add**(a,b,`bitlen`) (resp. **sub**(a,b,`bitlen`)) be the addition (resp. subtraction) between a and b. The values a and b are represented over `bitlen` bits.
- Let **lshift**(a,k,`bitlen`) (resp. **rshift**(a,k,`bitlen`)) be the left (resp. right) shift $a << k$ (resp. $a >> k$) over `bitlen` bits.
- Let **and**(a,b,`bitlen`) be the AND operation $a\&b$ over `bitlen` bits.
- Let **mult**(a,b,$\texttt{bitlen}_a$,$\texttt{bitlen}_b$) be the integer multiplication $a \times b$ where a (resp. b) is represented on $\texttt{bitlen}_a$ (resp. $\texttt{bitlen}_b$) bits.
- Let **modMult**(a,b,$\texttt{bitlen}_a$,$\texttt{bitlen}_b$,p) be the integer modular multiplication $a \times b \mod p$ where a (resp. b) is represented on $\texttt{bitlen}_a$ (resp. $\texttt{bitlen}_b$) bits.

Concatenation. Let $(\ell, k, N) \in \mathbb{N}^3$ with $\ell \leq k$ and $m \in \mathbb{N}$ represented over ℓ bits. In the following we denote by **concat**(m, k, N) the function that represents m on k bits and concatenates this new representation N times. Formally:

$$\textbf{concat}(m, k, N) = \sum_{j=0}^{N-1} m2^{jk} \in \mathbb{N}$$

Example 9. *Let $m = 1$ then* **concat**$(m, 8, 3) = 0x10101$.

Integer to polynomial. Let $(\ell, k, N) \in \mathbb{N}^3$, $\ell > k$ and $F(X) = f_0 + \ldots + f_{N-1}X^{N-1} \in \mathbb{Z}[X]$. For all i, let $\tilde{f_i}$ be the two's complement representation of f_i over k bits. We denote by:

$$f = \textbf{polyToN}(F(X), k, \ell) = \sum_{i=0}^{N-1} \tilde{f_i}2^{i\ell}, f \in \mathbb{N}$$

Let $g = (g_0, g_1, \ldots, g_{N-1})_\ell \in \mathbb{N}$ a $N\ell$-bit number.

$$G(X) = \textbf{NtoPoly}(g, \ell) = \sum_{i=0}^{N-1} g_iX^i$$

The obtained polynomial $G(X)$ belongs to $\mathbb{N}[X]$ and its degree is at most $N-1$.

Example 10. *Let* $F(X) = f_2X^2 + f_1X + f_0 = 2X^2 + 4X - 2$. *Let* $\tilde{f}_0 = 0$x$E, \tilde{f}_1 = 0$x$4, \tilde{f}_2 = 0$x2, *be representations of all* f_i *over* 4 *bits. Then,* $f = \mathbf{polyToN}(F(X), 4, 8) = 0x02040E$ *and* $\mathbf{NtoPoly}(f, 8) = 2X^2 + 4X + 14$

4.2.2 Multiplication in $\mathbb{N}[X]$ using Kronecker substitution

The Kronecker substitution was first introduced in [Kro82]. We give here the main steps of this substitution; for more details see Section 2.2.1 page 19. The idea of this substitution is to transform a polynomial multiplication to an integer one by evaluating the polynomials and get back to the result using a radix conversion. In the context of embedded devices, this transformation is of interest to perform polynomial multiplication by using the RSA/ECC coprocessor. Indeed, such coprocessor handles multiplication on integers.

In this section we assume that our polynomials are defined over $\mathbb{N}[X]$.

Kronecker substitution

The Kronecker substitution multiplies two polynomials $F(X)$ and $G(X)$ using an integer multiplication. This substitution can be summarized in three steps:

1. Evaluation of $F(X)$ and $G(X)$ at 2^ℓ. The value ℓ is chosen such that all the coefficients after the polynomial multiplication are lower than 2^ℓ.
2. Integer multiplication $r = F\left(2^\ell\right) G\left(2^\ell\right), r \in \mathbb{N}$.
3. Get back to polynomial $R(X) \in \mathbb{N}[X]$ using radix conversion on r.

Evaluation. The first step of the Kronecker substitution is the polynomial evaluation at 2^ℓ. Since $F(X)$ has coefficients in $\mathbb{N}$ represented over k bits:

$$\textsc{Evaluation}_{\geq 0}(F(X), k, \ell) \coloneqq F\left(2^\ell\right) = \mathbf{polyToN}(F(X), k, \ell) \tag{4.10}$$

Example 11. *Let* $F(X) = 2X^2 + X + 3$ *then* $F(2^8) = 0$x$020103 = \textsc{Evaluation}_{\geq 0}(F(X), 2, 8)$

Evaluation point. Let $R(X) = F(X)G(X)$ where $F(X), G(X) \in \mathbb{N}[X]$ of degree at most $N - 1$. The evaluation point 2^ℓ is chosen such that for all $i \leq 2(N-1)$:

$$r_i \leq \max_{j \in \{0,\ldots,N-1\}}(f_j) \max_{j \in \{0,\ldots,N-1\}}(g_j) N < 2^\ell$$

By the fact that all the coefficients are non-negative, this evaluation is only a representation of all the f_i over ℓ bits. Then in an implementation, the evaluation does not require arithmetic operations.

Radix Conversion. Radix conversion aims to transform an integer into a polynomial.

Let $f = (f_0, \ldots, f_{N-1})_\ell \in \mathbb{N}$, then:

$$F(X) = f_0 + \ldots + f_{N-1}X^{N-1} \coloneqq \textsc{Radix Conversion}_{\geq 0}(f) = \mathbf{NtoPoly}(f, \ell) \tag{4.11}$$

Example 12. *Let* $f = 0$x020103 *then* $F(X) = 2X^2 + X + 3 = \textsc{Radix Conversion}_{\geq 0}(f)$

The radix conversion converts a packed integer representation to an array one. Like the evaluation algorithm, in an implementation, the radix conversion does not require arithmetic operation.

Example of Kronecker substitution.

Example 13. *Let* $F(X) = 2X^2 + X + 3$ *and* $G(X) = X^2 + 1$. *Then,*

$$F(2^8) = 0x020103 = \textsc{Evaluation}_{\geq 0}(F(X), 2, 8), G(2^8) = 0x010001 = \textsc{Evaluation}_{\geq 0}(G(X), 2, 8)$$

Afterwards we multiply the evaluated polynomials $r = F(2^8)G(2^8) = 0x201050103$. *Finally we obtain* $R(X) = \textsc{Radix Conversion}_{\geq 0}(r) = 2X^4 + X^3 + 5X^2 + X + 3$.

4.2.3 Multiplication in $R_{q,\delta}$ using Kronecker substitution

In the previous section we perform polynomial multiplication as an integer one with polynomials in $\mathbb{N}[X]$. However, in the lattice-based schemes some polynomials, mainly the secret ones, have coefficients with a negative representation close to 0. Moreover, the reduction modulo $X^N + 1$ can also bring negative coefficients. Then in this section we focus on polynomial multiplication in $R_{q,\delta} = \mathbb{Z}_q[X]/(X^N + \delta)$. In $R_{q,\delta}$, the polynomial multiplication using Kronecker substitution is achieved as follows:

- Evaluation of polynomials considering negative coefficients.
- Integer multiplication modulo $2^{N\ell} + \delta$. The modular reduction ensures that after radix conversion the polynomial result is reduced modulo $X^N + \delta$.
- Radix conversion to obtain a polynomial in $\mathbb{Z}[X]/(X^N + \delta)$.
- Reduction modulo q of the polynomial coefficients.

The Algorithms 4 and 5 in Chapter 2 already achieve the evaluation and the radix conversion with negative coefficients. However, these algorithms are done using array representations. In this section we describe a way to realize these algorithms when the coefficients are on a packed integer representation. The main advantage of this representation is that it allows to repurpose existing coprocessor.

Negative representation. Our goal is to perform polynomial multiplication over $R_{q,\delta}$. Then, a way to avoid the negative coefficients is to represent them with a non-negative representation over $R_{q,\delta}$. However, the negative coefficients are close to 0, then the closest non-negative representation is nearby q. This involves that the evaluation point must be higher and then the integer operations are done on much larger integers; see Chapter 3 for more details on the impact on the evaluation point. Thus, for the sake of efficiency we use, when possible, our algorithms with the negative representation.

Evaluation with negative coefficients.

Let $F(X) = f_0 + f_1X + \ldots + f_{N-1}X^{N-1} \in R^-_{q,\delta}$ and $\tilde{f}_i$ be the two's complement representation over k bits of f_i. Our goal is to evaluate $F(X)$ at 2^ℓ where $\ell > k$, then for $i = 0$ to $N - 1$:

- If $f_i \geq 0$, then we only have to represent it on ℓ bits (as in Section 4.2.1).
- If $f_i < 0$, then we have to represent it with a two's complement over ℓ bits and propagate a borrow to the next coefficient. To obtain a two's complement representation from k bits to ℓ bits, we compute:

$$\tilde{f}_i + (2^\ell - 2^k) = 2^k + f_i + (2^\ell - 2^k) = 2^\ell + f_i$$

The Algorithm 15 computes the two's complement representation of the polynomial evaluation when the coefficients are in $\mathbb{Z}$. More precisely, this evaluation is done using arithmetic operations on a packed integers representation. To do so, we first represent the polynomial coefficients into a packed integers

form, as defined in Equation 4.10. Afterwards, we use arithmetic operations in order to convert the two complement's representation from k to ℓ bits and to propagate the required borrows.

Algorithm 15 EVALUATION

Input: $F(X) \in R^{-}_{q,\delta}$, $k, \ell \in \mathbb{N}$ where $\ell > k$.

Output: $\tilde{f} \in \mathbb{N}$ the two's complement representation of $F\left(2^{\ell}\right) \mod 2^{N\ell}$

1: $\texttt{mask} \leftarrow \textbf{concat}(1, \ell, N)$ //Precomputed
2: $\tilde{f} \leftarrow \textbf{polyToN}(F(X), k, \ell)$
3: $\texttt{neg} \leftarrow \textbf{rshift}(\tilde{f}, k-1, N\ell)$
4: $\texttt{neg} \leftarrow \textbf{and}(\texttt{neg}, \texttt{mask}, N\ell)$ // Detect negative coefficients
5: $\texttt{tmp} \leftarrow \textbf{mult}(\texttt{neg}, 2^{\ell} - 2^{k}, N\ell, 32)$
6: $\tilde{f} \leftarrow \textbf{add}(\tilde{f}, \texttt{tmp}, N\ell)$ // Two's complement representation of each coeff over ℓ bits
7: $\texttt{neg} \leftarrow \textbf{lshift}(\texttt{neg}, \ell, N\ell)$
8: $\tilde{f} \leftarrow \textbf{sub}(\tilde{f}, \texttt{neg}, N\ell)$ // Borrow propagation
9: **return** $\tilde{f}$

Remark 6. *The value* ***mask*** *is always the same for a fixed scheme. Then, this integer can be precomputed and stored in Non-Volatile Memory (NVM).*

Remark 7. *The* EVALUATION *(Algorithm 15) returns the two's complement representation of* $F\left(2^{\ell}\right) \mod 2^{N\ell}$. *This implies:*

- *If* $F\left(2^{\ell}\right) \geq 0$, *then the returned value is equal to* $F\left(2^{\ell}\right)$.
- *Otherwise, the returned value is not equal to* $F\left(2^{\ell}\right)$. *This case occurs when the latest non-zero coefficient of* $F(X)$ *is negative.*

To obtain the expected result after the Kronecker Substitution, the last case requires additional operations before the radix conversion. These additional operations are described in Section 4.2.3 paragraph ***Two's complement representation of the evaluated polynomial****.*

Example 14. *Let* $F(X) = 3X^2 - 2X + 2$, *where all the coefficients are encoded with a two's complement representation over* $k = 4$ *bits. Let* $N = 3$ *and* $\ell = 8$. *The expected result is* $F(2^8) = 0x02FE02$. *This is obtained with* EVALUATION$(F(X), k, \ell)$*:*

1. $\texttt{mask} \leftarrow \textbf{concat}(1, 8, 3) = 0x010101$
2. $\tilde{f} \leftarrow \textbf{polyToN}(F(X), 4, 8) = 0x030E02$
3. $\texttt{neg} \leftarrow \textbf{rshift}(\tilde{f}, 4-1, 3 \times 8) = 0x0061C0$
4. $\texttt{neg} \leftarrow \textbf{and}(\texttt{neg}, \texttt{mask}, 3 \times 8) = 0x000100$
5. $\texttt{tmp} \leftarrow \textbf{mult}(\texttt{neg}, 2^8 - 2^4, 3 \times 8, 32) = 0x00F000$
6. $\tilde{f} \leftarrow \textbf{add}(\tilde{f}, \texttt{tmp}, 3 \times 8) = 0x03FE02$
7. $\texttt{neg} \leftarrow \textbf{lshift}(\texttt{neg}, 8, 3 \times 8) = 0x010000$
8. $F(2^8) \leftarrow \textbf{sub}(\tilde{f}, \texttt{neg}, 3 \times 8) = 0x02FE02$

Evaluation point. Let $R(X) = F(X)G(X)$ where $F(X), G(X) \in R_{q,\delta}$. The evaluation point 2^ℓ is chosen such that for all $i \leq 2(N-1)$:

$$r_i \leq \max_{j\in\{0,\dots,N-1\}}(|f_j|) \max_{j\in\{0,\dots,N-1\}}(|g_j|)N < 2^{\ell-1}$$

Radix Conversion with negative coefficient representation.

As mentioned in [AHH+19] or in Chapter 2, the radix conversion has to be adapted since some coefficients have negative representations. Two issues arise with the negative coefficients:

1. The evaluation and the integer multiplication propagate borrow between the polynomial coefficients.
2. The negative evaluation algorithm returns two's complement representation over $N\ell$ bits.

Borrow between the coefficients. The evaluation converts a polynomial to a packed integers representation. In the following of the Kronecker substitution, the obtained natural numbers are manipulated regardless the original polynomial structure. Therefore, borrows can be propagated between the coefficients. However in order to retrieve the expected polynomial result, the radix conversion must compensate the propagated borrows by propagating back carries.

Let $\tilde{r} = (\tilde{r}_0, \tilde{r}_1, \dots, \tilde{r}_{N-1})_\ell \in \mathbb{N}$ be the integer that we want to convert to a polynomial, where for all i, $\tilde{r}_i$ is a two's complement representation over ℓ bits of an integer $-2^{\ell-1} < r_i < 2^{\ell-1}$. In order to propagate back the carries, we transform the negative coefficients to non-negative ones by adding a multiple of our modulus q: `maxValue`. More precisely, `maxValue` is the smallest multiple of q such that for all i, $-\texttt{maxValue} \leq r_i < \texttt{maxValue}$. Moreover with the parameters that we use in Section 4.2.5, we have $\texttt{maxValue} < 2^{\ell-1}$. Then, by adding `maxValue` we got:

- If $r_i < 0$, then $2^\ell \leq \tilde{r}_i + \texttt{maxValue} = 2^\ell + r_i + \texttt{maxValue} < 2^{\ell+1}$. Therefore a carry is propagated to $\tilde{r}_{i+1}$.
- If $r_i \geq 0$, then $\tilde{r}_i + \texttt{maxValue} = r_i + \texttt{maxValue} < 2^\ell$.

After adding `maxValue`, the values r_i are considered as natural numbers represented over ℓ bits. Then, the expected polynomial is obtained by using the radix conversion algorithm defined in Equation 4.11 on $\tilde{r}$.

This negative to non-negative conversion is possible because the polynomial multiplication is done over $R_{q,\delta}$. Indeed after reduction modulo q, the added value `maxValue` is equal to 0.

Two's complement representation of the evaluated polynomial. The second issue is due to the two's complement representation of the evaluated polynomial.

Let $F(X) = f_0 + \ldots + f_{N-1}X^{N-1} \in R^-_{q,\delta}$ of degree $N-1$ and $\ell \in \mathbb{N}$. Then Algorithm 15 returns the integer $f \leftarrow \textsc{Evaluation}(F(X), k, \ell)$, that is the two's complement representation of $F\left(2^\ell\right) \mod 2^{N\ell}$. Two cases are to be distinguished:

- $f_{N-1} > 0$, then $f = F\left(2^\ell\right) \in \mathbb{N}$.
- $f_{N-1} < 0$, then $f = 2^{N\ell} + F\left(2^\ell\right)$ is the two's complement of $F\left(2^\ell\right)$ modulo $2^{N\ell}$.

Only the second case will lead to a wrong result after the modular multiplication. Indeed, let $g \in \mathbb{N}$ and $f = 2^{N\ell} + F\left(2^\ell\right)$ we got:

$$r \mod \left(2^{N\ell}+\delta\right) = fg \mod \left(2^{N\ell}+\delta\right) = 2^{N\ell}g + F\left(2^\ell\right)g \mod \left(2^{N\ell}+\delta\right) \neq F\left(2^\ell\right)g \mod \left(2^{N\ell}+\delta\right)$$

Then in this case, before the radix conversion we must add or subtract g to r, depending on δ:

- $\delta = 1 : 2^{N\ell}g \mod \left(2^{N\ell}+1\right) = -g \mod \left(2^{N\ell}+1\right)$, then

$$r + g \mod \left(2^{N\ell}+1\right) = F\left(2^{\ell}\right) g \mod \left(2^{N\ell}+1\right)$$

- $\delta = -1 : 2^{N\ell}g \mod \left(2^{N\ell}-1\right) = g \mod \left(2^{N\ell}-1\right)$, then

$$r - g \mod \left(2^{N\ell}-1\right) = F\left(2^{\ell}\right) g \mod \left(2^{N\ell}-1\right)$$

Previously, we supposed that at most one polynomial can have negative coefficients. In case of lattice-based schemes, this is always the case.

Algorithm 16 Radix Conversion

Input: $r, g, \texttt{maxValue} \in \mathbb{N}$, and $\texttt{sign} \in \{0, 1\}$
Output: $R(X) \in \mathbb{N}[X]/(X^N + \delta)$

1: $\texttt{max} \leftarrow$ **concat**$(\texttt{maxValue}, \ell, N)$ //Can be precomputed
2: **if** sign eq 1 **then**
3: **if** δ eq 1 **then**
4: $r \leftarrow$ **add**$(r, g, N\ell)$ // To handle negative last coeff
5: **else**
6: $r \leftarrow$ **sub**$(r, g, N\ell)$ // To handle negative last coeff
7: **end if**
8: **else**
9: **if** δ eq 1 **then**
10: $\texttt{dummy} \leftarrow$ **add**$(r, g, N\ell)$ // For isochrony
11: **else**
12: $\texttt{dummy} \leftarrow$ **sub**$(r, g, N\ell)$ // For isochrony
13: **end if**
14: **end if**
15: $r \leftarrow$ **add**$(r, \texttt{max}, N\ell)$ // Add maxValue to each coefficient
16: $R(X) \leftarrow$ Radix Conversion$_{\geq 0}(r)$

Multiplication in $R_{q,\delta}$ using coprocessor

The Sections 4.2.3 and 4.2.3 are used to obtain a polynomial multiplication algorithm in $R_{q,\delta}$ using, mainly, a packed integer representation. More precisely, except for the modular reductions modulo q, the operations are done using this representation.

All operations performed on the packed integers representation can be achieved with coprocessor as defined in Section 4.2.1.

The Polynomial Multiplication in $R_{q,\delta}$ algorithm is described in Algorithm 17.

Algorithm 17 POLYNOMIAL MULTIPLICATION IN $R_{q,\delta}$

Input: $(F(X), G(X)) \in (R^{-}_{q,\delta}, R_{q,\delta})$ of degree $N-1$. Let $k, \ell, q \in \mathbb{N}$ where $\ell > k$, and `maxValue` defined as above.
Output: $R(X) = F(X)G(X) \in R_{q,\delta}$

1: $f \leftarrow$ EVALUATION$(F(X), k, \ell)$
2: $G\left(2^{\ell}\right) \leftarrow$ EVALUATION$_{\geq 0}(G(X), k, \ell)$
3: $r \leftarrow$ **modMult**$(f, G\left(2^{\ell}\right), N\ell, N\ell, 2^{N\ell} + \delta)$
4: $b \leftarrow$ `sign`$(F[N-1])$ // if $F_{N-1} < 0$ then $b = 1$, otherwise $b = 0$.
5: $R(X) \leftarrow$ RADIX CONVERSION$(r, G\left(2^{\ell}\right), \texttt{maxValue}, b)$
6: $R(X) \leftarrow R(X) \mod q$ // Any modular reduction
7: **return** $R(X)$

In the following section we determine how to perform modular reductions modulo q using packed integers representation.

4.2.4 Reducing coefficients modulo q

In Section 4.2.3, we perform polynomial multiplication in $R_{q,\delta}$. However, the reduction modulo q is done after the radix conversion on a polynomial representation. In this section we show how to perform reduction modulo q using packed integers representation. As mentioned previously, such representation allows to repurpose existing RSA/ECC coprocessor.

Let $r = (r_0, \ldots, r_{N-1})_{\ell} \in \mathbb{N}$. In our context, r is obtained after the two first steps of the Kronecker substitution: polynomial evaluation and modular integer multiplication. Moreover, we have added `maxValue` like in Section 4.2.3. Then, each r_i is such that for all i: $0 \leq r_i < 2\texttt{maxValue}$.

In the following we denote by *simultaneous reduction*, the fact of reducing all the $r_i \mod q$ by performing operations on r.

Power-of-two modulus

Some of lattice-based schemes, like Saber [BMD+21] and NTRU [CDH+19], use a power-of-two modulus. In this context, the simultaneous reduction is easy and fast. Indeed, the simultaneous reduction is achieved by the computation:

$$r \& \textbf{concat}(q-1, \ell, N)$$

Prime modulus

Kyber [BDK+18] and Dilithium [BDL+21] are lattice-based cryptosystems which perform polynomial multiplication over $R_{q,\delta}$, where q is a prime number. In this section we adapt two modular reductions: Quotient Approximation Reduction (cf. Section 4.1 page 40) and Barrett [Bar86], to perform simultaneous reduction.

Quotient Approximation Reduction

Quotient Approximation Reduction is introduced in Section 4.1 page 40. Let $a \in \mathbb{N}$ be an integer to reduce modulo $q \in \mathbb{N}$. Quotient Approximation Reduction computes an approximation `quo` of $\left\lfloor \frac{a}{q} \right\rfloor$ and then computes $a' = a - \texttt{quo} \times q$. The obtained result is $a' = a \mod q + tq$, where $0 \leq t < \left\lfloor \frac{a}{q} \right\rfloor$. A tight upper bound of t is given in Section 4.1. The value t is expected to be close to 0. In order to finish the reduction, several subtractions by q may be required.

Let ℓ be the bit-length of a and J_ℓ be the set of integers:

$$J_\ell = \{1 \leq j \leq \ell - 1, \lfloor 2^j/q \rfloor = 2\lfloor 2^{j-1}/q \rfloor + 1\}$$

Let $J'_\ell \subseteq J_\ell$, where at least the first element of J_ℓ is in J'_ℓ. The Quotient Approximation Reduction algorithm is described in Algorithm 18.

Algorithm 18 QUOTIENT APPROXIMATION REDUCTION

Input: a, q, J'_ℓ defined as above.
Output: $a' = a \mod q + tq$ where $0 \leq t < \left\lfloor \frac{a}{q} \right\rfloor$

1: `quo_approx` $\leftarrow 0$
2: **for** each $j \in J'_\ell$ **do**
3: `quo_approx` $\leftarrow$ `quo_approx` $+ (a >> j)$
4: **end for**
5: $a' \leftarrow a -$ `quo_approx` $\cdot q$
6: **return** a'

Remark 8. *The Algorithm 18 is named in 4.1 Quotient Approximation Reduction Relaxed.*

Simultaneous modular reduction. Applying Algorithm 18 to an integer obtained with Kronecker substitution is not sufficient to get a simultaneous reduction. Indeed, because of the shift at line 3, noise coming from coefficient $i+1$ can overflow on the coefficient i. We show in Algorithm 19 how to remove this noise, by applying a bitmask.

Algorithm 19 SIMULT. QUOTIENT APPROXIMATION REDUCTION

Input: $r = (r_0, \ldots, r_{N-1})_\ell \in \mathbb{N}$, $q \in \mathbb{N}$, J'_ℓ defined as above.
Output: $r' = (r'_0, \ldots, r'_{N-1})_\ell \in \mathbb{N}$, where all r_i are reduced with QUOTIENT APPROXIMATION REDUCTION algorithm.

1: `quo_approx` $\leftarrow 0$
2: **for** each $j \in J'_\ell$ **do**
3: `mask` $\leftarrow$ **concat**$(2^{\ell-j} - 1, \ell, N)$
4: `tmp` $\leftarrow$ **rshift**$(r, j, N\ell)$
5: `tmp` $\leftarrow$ **and**(`tmp`, `mask`, $N\ell$)
6: `quo_approx` $\leftarrow$ **add**(`quo_approx`, `tmp`, $N\ell$)
7: **end for**
8: `quo_approx` $\leftarrow$ **mult**(`quo_approx`, q, $N\ell$, 32) // Mult between a word and a large integer
9: **return** r'

Remark 9. *The set J'_ℓ is fixed for a given modulus. Then the values* `mask` *can be precomputed and stored in NVM.*

In our application context (cf. Section 4.2.5), the Algorithm 19 reduces the r_i such that $r'_i = r_i \mod q + t_i q$, where $t_i \in \{0, 1\}$. In order to completely reduce the r'_i modulo q, in Section 4.2.4 we explain how to perform simultaneous conditional subtraction by q.

Barrett

The Barrett reduction is introduced in [Bar86]. The main idea is to precompute an approximation of a division and use it to perform modular reduction. Let $\alpha, \beta \in \mathbb{Z}$ and $a \in \mathbb{N}$ be an integer to reduce modulo

$q \in \mathbb{N}$ of bit-length $k \in \mathbb{N}$. Barrett reduction precomputes $m = \left\lfloor \frac{2^{k+\alpha}}{q} \right\rfloor$ and computes:

$$a' = a - [((a >> (k+\beta)) \cdot m) >> (\alpha - \beta)]\, q$$

A special case is when $\alpha = \beta$, therefore the computation is

$$a' = a - [a >> (k+\beta)] \cdot m \cdot q$$

In this case, only one shift and one multiplication is performed ($m \cdot q$ is precomputed).

Depending on the parameters $(\alpha, \beta), a' = a \mod q + tq$ where $0 \leq t < \left\lfloor \frac{a}{q} \right\rfloor$. Further details on the Barrett algorithms are given in [MVV18].

Simultaneous modular reduction. To adapt this reduction to simultaneous reduction we need to perform logical `AND` after the shift operations. Indeed, like Quotient Approximation Reduction the shift operations spread some noise between the coefficients. The Algorithm 20 describes the simultaneous Barrett reduction.

Algorithm 20 SIMULT. BARRETT$_{\alpha,\beta}$

Input: $r = (r_0, \ldots, r_{N-1})_\ell \in \mathbb{N}$. Let $q \in \mathbb{N}$ of bit-length k and $m = \left\lfloor \frac{2^{k+\alpha}}{q} \right\rfloor$.
Output: $r' = (r'_0, \ldots, r'_{N-1})_\ell \in \mathbb{N}$, where all r_i are reduced with BARRETT reduction.

1: `mask` $\leftarrow$ **concat**$(2^{\ell-\alpha+\beta} - 1, \ell, N)$
2: `mask'` $\leftarrow$ **concat**$(2^{\ell-k-\beta} - 1, \ell, N)$
3: `tmp` $\leftarrow$ **rshift**$(r, k+\beta, N\ell)$
4: `tmp` $\leftarrow$ **and**$(\texttt{tmp}, \texttt{mask}', N\ell)$
5: `tmp` $\leftarrow$ **mult**$(\texttt{tmp}, m, N\ell, 32)$ // Mult between a word and a large integer
6: `tmp` $\leftarrow$ **rshift**$(\texttt{tmp}, \alpha - \beta, N\ell)$
7: `tmp` $\leftarrow$ **and**$(\texttt{tmp}, \texttt{mask}, N\ell)$
8: `tmp` $\leftarrow$ **mult**$(\texttt{tmp}, q, N\ell, 32)$ // Mult between a word and a large integer
9: $r' \leftarrow$ **sub**$(r, \texttt{tmp}, N\ell)$
10: **return** r'

Remark 10. *The values* `mask` *and* `mask'` *can be precomputed.*

In our application context (cf. Section 4.2.5), the Algorithm 20 reduces the r_i such that $r'_i = r_i \mod q + t_i q$, where $t_i \in \{0, 1, 2\}$. In the following, we explain how to completely reduce simultaneously the r'_i.

Final reduction

Using the simultaneous reduction Algorithms 19 and 20, the returned result $r' = (r'_0, \ldots, r'_{N-1})_\ell \in \mathbb{N}$ is such that, for all i, $r'_i = r'_i \mod q + t_i q$. With the parameters sets that we use in Section 4.2.5, for all i, $t_i \in \{0, 1, 2\}$.

Let k and c such that $q = 2^k - c$. Then $r'_i \geq 2q$ if and only if $r'_i + 2c$ has its $(k+1)$-th bit equal to one. This fact is used in Algorithm 21 to detect and subtract q to coefficients $\geq 2q$ in a packed integers representation.

Algorithm 21 SIMULT. CONDITIONAL SUBTRACTION

Input: $r' = (r'_0, \ldots, r'_{N-1})_\ell$ with all $0 \leq r'_i < 3q$, where $q = 2^k - c$, $\ell, N \in \mathbb{N}$.
Output: $r'' = (r''_0, \ldots, r''_{N-1})_\ell$ with all $0 \leq r''_i < 2q$

1: $(\mathtt{C}, \mathtt{mask}) \leftarrow (\mathbf{concat}(2c, \ell, N), \mathbf{concat}(1, \ell, N))$ //Precomputed
2: $\mathtt{tmp} \leftarrow \mathbf{add}(r', \mathtt{C}, N\ell)$ //Raised the $k+1$-th bit in each coeff
3: $\mathtt{tmp} \leftarrow \mathbf{rshift}(\mathtt{tmp}, k+1, N\ell)$ // Move the $k+1$-th bit to position 0 in each coeff
4: $\mathtt{tmp} \leftarrow \mathbf{and}(\mathtt{tmp}, \mathtt{mask}, N\ell)$ // Detect the coeff $\geq 2q$
5: $\mathtt{tmp} \leftarrow \mathbf{mult}(\mathtt{tmp}, q, N\ell, 32)$ // Mult between a word and a large integer
6: $r'' \leftarrow \mathbf{sub}(r', \mathtt{tmp}, N\ell)$ // Subtract q to each coeff $\geq 2q$
7: **return** r''

Remark 11. *The values* `C` *and* `mask` *are fixed for a given scheme. Then, these integers can be precomputed and stored in NVM.*

After using the Algorithm 21, the r''_i are bounded by $2q$. In that case, this algorithm can be adapted replacing $2c$ by c (line 1) and $k+1$ by k (line 3). It follows that q is subtracted from each $r''_i \geq q$. Afterwards, each r''_i is necessary lower than q.

Modular polynomial multiplication using coprocessor

The Algorithm 22 performs polynomial multiplication in $R_{q,\delta}$ using operations on packed integers representation. All operations performed on this representation can be achieved with coprocessor as defined in Section 4.2.1.

The MODULAR POLYNOMIAL MULTIPLICATION Algorithm 22 works as follows:

1. Line 2: Polynomial evaluations defined in Equation 4.10 and Algorithm 15.

2. Line 3: Modular integer multiplication modulo $2^{N\ell} + \delta$ of the evaluated polynomials.

3. Line 4 to 11: Handle the two's complement representation of the evaluated polynomial; see Section 4.2.3.

4. Line 12: Convert the negative representation to non negative one; see Section 4.2.3. This operation allows to perform simultaneous reduction mod q and radix conversion.

5. Line 13 to 19: Perform simultaneous reduction mod q. This ensures that the polynomial result has coefficients reduced mod q.

6. Line 20: Radix conversion defined in Equation 4.11 to obtain a polynomial result.

Algorithm 22 MODULAR POLYNOMIAL MULTIPLICATION

Input: $(F(X), G(X)) \in (R^{-}_{q,\delta}, R_{q,\delta})$ of degree $N-1$. Let $k, \ell, q \in \mathbb{N}$ where $\ell > k$, and `maxValue` defined as above.
Output: $R(X) = F(X)G(X) \in R_{q,\delta}$

1: `max` $\leftarrow$ **concat**(`maxValue`, ℓ, N) // Precomputed
2: $\left(f, G\left(2^{\ell}\right)\right) \leftarrow$ (EVALUATION$(F(X), \ell)$, EVALUATION$_{\geq 0}(G(X), \ell)$)
3: $r \leftarrow$ **modMult**$(f, G\left(2^{\ell}\right), N\ell, N\ell, 2^{N\ell} \pm \delta)$
4: $b \leftarrow$ `sign`$(f[N-1])$
5: **if** b eq 1 **then**
6: **if** δ eq 1 **then** $r \leftarrow$ **sub**$(r, G\left(2^{\ell}\right), N\ell)$ // To handle negative last coeff
7: **else** $r \leftarrow$ **add**$(r, G\left(2^{\ell}\right), N\ell)$
8: **else**
9: **if** δ eq 1 **then** `dummy` $\leftarrow$ **sub**$(r, G\left(2^{\ell}\right), N\ell)$ // For isochrony
10: **else** `dummy` $\leftarrow$ **add**$(r, G\left(2^{\ell}\right), N\ell)$
11: **end if**
12: $r \leftarrow$ **add**$(r,$ `max`$, N\ell)$ //Negative to non negative representation for all r'_i
13: **if** q eq 2^k **then**
14: `mask`$' \leftarrow$ **concat**$(2^k - 1, \ell, N)$
15: $r \leftarrow$ **and**$(r,$ `mask`$', N\ell)$
16: **else**
17: $r \leftarrow$SIMULT. QA REDUCTION(r) // Or SIMULT. BARRETT$_{\alpha,\beta}$ or a combination of them
18: $r \leftarrow$ SIMULT. CONDITIONAL SUBTRACTION(r, ℓ, N) // Can be applied twice if some $r_i \geq 2q$
19: **end if**
20: $R(X) \leftarrow$ RADIX CONVERSION$_{\geq 0}(r)$
21: **return** $R(X)$

4.2.5 Applications and Results

In this section, after some preliminaries, we present the component on which we performed our experiments and the results obtained by implementing the `Modular Polynomial Multiplication` (`MPM`) Algorithm 22 and another polynomial multiplication depending of the evaluated scheme. The evaluated lattice-based algorithms are: `Kyber`, `Dilithium`, `NTRU`, and `Saber`.

Background

NTT

NTT is an algorithm allowing to perform fast polynomial multiplication in $R_{q,1}$ [Nus82]. Given a and $b \in R_{q,1}$, $a \times b$ is computed as $\mathrm{NTT}^{-1}(\mathrm{NTT}(a) \circ \mathrm{NTT}(b))$, where $\circ$ is the coefficient-wise multiplication.

Theoretically, NTT has the best asymptotic complexity for multiplication in $R_{q,1}$. However, in constrained environments (e.g. smart cards), devices may have dedicated hardware to perform fast large-integer arithmetic. In this context, NTT can be outperformed by an algorithm relying on integer arithmetic, even if its theoretical complexity is worse than NTT.

Subdivision

RSA/ECC coprocessors perform integer arithmetic with data in buffer size has a fixed limit. In our context after polynomial evaluation, the resulting integer is generally too large to fit in these buffers. In that case

we use multiple-precision arithmetic. This arithmetic consists of dividing the manipulated integers into several smaller ones and then perform operations on these smaller integers.

In the case of integer multiplication we use two techniques to divide the integer multiplication into smaller ones: Karatsuba and Schoolbook. Let $f = f_I + f_S 2^{N\ell/2}$ and $g = g_I + g_S 2^{N\ell/2}$, where f_I, f_S, g_I, g_S are lower than $2^{N\ell/2}$.

Schoolbook: $fg = f_I g_I + (f_I g_S + f_S g_I)2^{N\ell/2} + 2^{N\ell} f_S g_S$

Karatsuba: $fg = f_I g_I + ((f_I + f_S)(g_I + g_S) - f_I g_I - f_S g_S)2^{N\ell/2} + 2^{N\ell} f_S g_S$

These techniques can be applied recursively in order to obtain a targeted integer size. Later on when presenting the results, we specify in a column named *subdivision* the multi-precision method that we use for the integer multiplication.

Evaluation point. In our context the Karatsuba subdivision requires to increase the size of the evaluation point by 1 bit at each subdivision. It is due to the computation $(f_I + f_S)(g_I + g_S)$. Indeed, this computation is performed on integers of length twice as small but with values twice as large.

In the following results, the evaluation point is chosen to take into account the negative coefficients and the Karatsuba subdivisions.

Polynomial distribution

The following polynomial multiplications are performed between a polynomial $G(X) \in R_{q,\delta}$ and $F(X) \in R^{-}_{q,\delta}$. More precisely, the coefficients of $G(X)$ are sampled uniformly in $\{0, \ldots, q-1\}$ and the coefficients of $F(X)$ are sampled in a distribution $\mathcal{D}_\sigma$. Using a distribution $\mathcal{D}_\sigma$, the coefficients are represented in $\{-\sigma, \ldots, 0, \ldots, \sigma\}$.

Masked secret polynomial. Most of the time the polynomial using the distribution $\mathcal{D}_\sigma$ is the secret polynomial. In some use cases, an embedded implementation must be strongly secured against side-channel attacks. One way to do this is to mask the secret data. To do so, we split the sensitive data into shares $x = x_1 + x_2 \mod q$, where x_1, x_2 belongs to $\{0, \ldots, q-1\}$, and then we process the operations on each share separately. In our context the value q is much larger than the secret distribution. Therefore, that implies we will manipulate larger secret data and then it increases the evaluation point. For some assessments, in order to consider this security requirement, we suppose that the polynomial $F(X)$ is defined over $R_{q,\delta}$ and its coefficients are sampled uniformly in $\{0, \ldots, q-1\}$. In the following results, we denote this case by $\mathcal{U}_q$ distribution.

In the following results, we only specify the distribution of $F(X)$.

Target

Assessments are done on a smart card component using a 32-bit architecture. In the following we refer to this device as Component B. Due to intellectual properties reasons, the component name or a detailed description cannot be given. Then, we only give the main characteristics of the component B:

- Standards 32-bit instructions (add, sub, shifts, bitwise and, xor, or, etc.).
- No CPU multiplication and division.
- A coprocessor which handles: logical `AND`, addition, subtraction, shifts, modular integer multiplication and the non-modular one.

The following results take into account a complete modular reduction. Moreover like the previous works [AHH+19; WGY20; BRvV22] or in Chapter 3, we assume that the inputs are already in the appropriate machine representation. This implies that the inputs are in:

- Polynomial representation for NTT, Karatsuba and schoolbook polynomial multiplication.
- Packed integers representation for the `MPM` algorithm.

Kyber

Kyber [BDK+18] is a lattice-based KEM finalist of the NIST standardization. The polynomial ring defined in Kyber is $R_{q,1} = \mathbb{Z}_q[X]/(X^N + 1)$, where $q = 3329$ and $N = 256$. The polynomial multiplication used in the specification is the NTT algorithm. In this context, we have implemented two polynomial multiplications:

- A NTT multiplication. It is adapted from the reference implementation, in order to use the hardware Montgomery multiplication. Tables of roots of unity have been recomputed to handle the Montgomery arithmetic with $R = 2^{32}$, the smallest handled by the coprocessor, instead of $R = 2^{16}$. In addition, the multiplication followed by a Montgomery reduction is replaced by a call to the coprocessor Montgomery multiplication. In Table 4.4 we present timings from the NTT's implementation.

	`NTT`	`Pointwise`	$\texttt{NTT}^{-1}$
`Cycles`	98k	40k	106k

Table 4.4: Kyber NTT cycles on Component B

- The modular polynomial multiplication (`MPM`) described in Algorithm 22. For this algorithm we consider two distributions for the polynomial $F(X)$:
  - $\mathcal{D}_3$. In this case the modular reduction modulo q is done using $\texttt{Simult.Barrett}_{11,0}$. In order to completely reduce the coefficients we perform 2 final subtractions using the technique described in Section 4.2.4.
  - $\mathcal{U}_q$. In this case the modular reduction modulo q is done using $\texttt{Simult.Barrett}_{10,10}$ and then an application of $\texttt{Simult.Barrett}_{13,-2}$. Afterwards, a final subtraction is performed using the technique described in Section 4.2.4.

In Table 4.5, we describe the parameters used for `MPM` algorithms. More precisely, we describe ℓ such that the evaluation point is 2^ℓ, the maximum value to convert negative coefficients to non-negative ones, the subdivision used and the obtained cycles.

Distribution	ℓ	`maxValue`	`Subdivision`	`Cycles MPM`
$\mathcal{D}_3$	23	$3qn$	None	50k
$\mathcal{U}_q$	34	q^2n	2 calls to Karatsuba	67k

Table 4.5: Parameters and cost of one multiplication in $R_{q,1}$ for Kyber parameters

Comparison. The previous results take into account one execution of `MPM` algorithm and each NTT routine. In order to compare NTT and `MPM` algorithms, we must not only compare `pointwise` routine with `MPM` algorithm. Indeed, we must also take into account calls to the `NTT` and $\texttt{NTT}^{-1}$ routines. Then, in order to compare the two polynomial multiplication methods we must determine how many times each algorithm is called.

The Table 4.6 describes the number of calls to `NTT`, `pointwise` multiplication and $\texttt{NTT}^{-1}$ during the `Key Generation`, `Encrypt` and `Decrypt` routines. The number of calls depends on the Kyber's security parameters which are $k = 2/3/4$. Note that the number of pointwise matches the number of `MPM` calls.

	NTT	Pointwise/MPM	$\texttt{NTT}^{-1}$
Key Gen.	$2k$	k^2	0
Encrypt	k	$k^2 + k$	$k + 1$
Decrypt	k	k	1

Table 4.6: Number of call to NTT routines in Kyber

In order to fairly compare NTT and `MPM` algorithms we use:

- The official specification of Kyber for the NTT algorithm. The private and public keys are stored in the NTT domain.
- A tweaked version of Kyber for the `MPM` algorithm. The private and public keys are not stored in the NTT domain. Therefore, we do not need to apply $\texttt{NTT}^{-1}$ to perform `MPM` algorithm.

The `MPM` algorithm is called with the $\mathcal{U}_q$ distribution parameters.

	Total cycles NTT	Total cycles MPM	Ratio (NTT/MPM)
$k = 2$			
Key Gen.	552k	**268k**	2
Encrypt	754k	**402k**	1.9
Decrypt	382k	**134k**	2.9
$k = 3$			
Key Gen.	948k	**603k**	1.6
Encrypt	1198k	**804k**	1.5
Decrypt	520k	**201k**	2.6
$k = 4$			
Key Gen.	1424k	**1072k**	1.3
Encaps	1722k	**1340k**	1.3
Decrypt	658k	**268k**	2.5

Table 4.7: Cycle count for all multiplications in Kyber for the $\mathcal{U}_q$ distribution parameters

Dilithium

Dilithium [BDL+21] is a lattice-based signature finalist of the NIST standardization. The polynomial ring defined in Dilithium is $R_{q,1} = \mathbb{Z}_q[X]/(X^N + 1)$, where $q = 8380417$ and $N = 256$. Like Kyber, the specification specified the use of NTT polynomial multiplication. Therefore, we have implemented two polynomial multiplications:

- A NTT multiplication. To this end, the reference implementation has been adapted. Tables of roots of unity have been recomputed to get non-negative values. Moreover, multiplications followed by a Montgomery reduction, in the reference code, are replaced by a call to the coprocessor Montgomery multiplication.

	NTT	Pointwise	$\texttt{NTT}^{-1}$
Cycles	114k	15k	128k

Table 4.8: Dilithium NTT cycles on Component B

- MPM algorithm. For this algorithm, we consider two distributions for the polynomial $F(X)$:
  - $\mathcal{D}_1$. The polynomial sampled in this distribution is always not sensitive to side-channel attacks. Therefore, we never need to mask it. The modular reduction modulo q is done by calling SIMULT. QUOTIENT APPROXIMATION REDUCTION with $J'_\ell = \{23\}$ and a final subtraction is performed using the technique described in Section 4.2.4.
  - $\mathcal{U}_q$. The modular reduction modulo q is done by calling SIMULT. QUOTIENT APPROXIMATION REDUCTION algorithm with $J'_\ell = \{23, 33\}$ and afterwards by calling it twice with $J'_\ell = \{23\}$. A final subtraction is performed using the technique described in Section 4.2.4.

Many other distributions are used in Dilithium. For the sake of clarity, we describe only the worst ones for MPM algorithm.

Distribution	ℓ	maxValue	Subdivision	Cycles MPM
$\mathcal{D}_1$	32	$60q$	None	48k
$\mathcal{U}_q$	57	q^2n	3 calls to Karatsuba	146k

Table 4.9: Parameters and cost of one multiplication in $R_{q,1}$ for Dilithium parameters

Comparison. Like Kyber, not every NTT-based multiplication uses all the three algorithms NTT, pointwise and $\texttt{NTT}^{-1}$.

The Table 4.10 presents the number of calls to NTT, pointwise multiplication and $\texttt{NTT}^{-1}$ depending on the Dilithium's security parameters which are $(k, l) = (4, 4)/(6, 5)/(8, 7)$. In this operation count we suppose that during the sign algorithm there is no rejection sampling. Note that the number of pointwise matches the number of MPM calls. The pointwise operations in boldface correspond to the polynomial multiplication with one polynomial in $\mathcal{D}_1$.

	NTT	Pointwise/MPM	$\texttt{NTT}^{-1}$
Key Gen.	l	lk	k
Sign	$2l + 2k + 1$	$lk + \mathbf{l} + \mathbf{2k}$	$l + 3k$
Verify	$l + k + 1$	$lk + \mathbf{k}$	k

Table 4.10: Number of call to NTT routines in Dilithium

Like Kyber, in order to compare fairly NTT and MPM algorithms we use:

- The official specification of Dilithium for the NTT algorithm. The public key is stored in the NTT domain.
- A tweaked version of Dilithium for the MPM algorithm. The public keys is not stored in the NTT domain. Therefore, we do not need to apply NTT^{-1} to perform MPM algorithm.

Moreover, for the MPM algorithm, the multiplication in boldface implies a polynomial sampled in $\mathcal{D}_1$ and the other ones are performed with a polynomial sampled in a $\mathcal{U}_q$ distribution.

	Total cycles NTT	Total cycles MPM	Ratio (NTT/MPM)
$(k,l) = (4,4)$			
Key Gen.	**1208k**	2336k	0.5
Sign	4406k	**2912k**	1.5
Verify	**1838k**	2528k	0.7
$(k,l) = (6,5)$			
Key Gen.	**1788k**	4380k	0.4
Sign	6271k	**5196k**	1.2
Verify	**2676k**	4668k	0.6
$(k,l) = (8,7)$			
Key Gen.	**2662k**	8176k	0.3
Sign	**8687k**	9280k	0.9
Verify	**3808k**	8560k	0.4

Table 4.11: Cycle count for all multiplications in Dilithium for the $\mathcal{U}_q$ distribution parameters

In this context, MPM algorithm is almost less efficient than NTT. However, we can combine NTT and MPM algorithms to obtain a faster Sign and Verify routines. Indeed, one can perform the multiplication which implies a polynomial sampled $\mathcal{D}_1$ using the MPM algorithm and the others multiplication using NTT algorithm. By combining these two multiplications, we avoid a lot of NTT and NTT^{-1} transformation which ensures an efficient polynomial multiplication. Moreover, this combination is achieved without changing the Dilithium specification.

In Table 4.12 the ratio is between the best algorithm in Table 4.11 (result in bold face) over the combination of NTT and MPM algorithms.

	Best in Table 4.11	Total cycles NTT + MPM	Ratio
$(k,l) = (4,4)$			
Sign	2912k	**1784k**	1.6
Verify	1838k	**1400k**	1.3
$(k,l) = (6,5)$			
Sign	5196k	**2604k**	2
Verify	2676k	**2076k**	1.3
$(k,l) = (8,7)$			
Sign	8687k	**3766k**	2.3
Verify	3808k	**3046k**	1.25

Table 4.12: Cycle count for all multiplications in Dilithium using the $\mathcal{U}_q$ and $\mathcal{D}_1$ distribution parameters

Saber & NTRU

Saber. Saber [BMD+21] is a lattice-based KEM finalist of the NIST standardization. The polynomial ring used in Saber is $R_{q,1} = \mathbb{Z}_q[X]/(X^N + 1)$, where $N = 256$ and $q = 8192 = 2^{13}$. In this work we consider two distributions for the polynomial $F(X)$:

- $\mathcal{D}_5$. Other distributions are used in Saber. However we only describe the worst one for the MPM algorithm.
- $\mathcal{U}_q$.

Since the modulus is a power of two, the reductions are achieved using a logical AND with the appropriate mask.

NTRU. NTRU [CDH+19] is also a KEM finalist of the NIST competition. The polynomial ring used in NTRU is $R_{q,-1} = \frac{\mathbb{Z}_q[X]}{(X^N-1)}$. The modulus q and the value N depends on the security parameters. In this work we only consider NTRU HPS 1 parameters, where $N = 509$ and $q = 2048 = 2^{11}$.

The value of N does not allow to easily make subdivisions. To overcome this issue, we work on polynomials with $\tilde{N} = 512$ coefficients where the latest coefficients are equal to 0.

In this work, we consider only a $\mathcal{U}_q$ distribution. Since q is a power of two, the modular reductions are performed with a logical `AND`.

Comparison. The Saber and NTRU `MPM` algorithms are compared with the polynomial multiplication used in their reference implementations.

- `Saber`: A combination of a 4-way Toom-Cook and Karatsuba algorithms.
- `NTRU`: A schoolbook multiplication.

The polynomial multiplication of the reference implementations are achieved with the 32 bits coprocessor multiplication. The Table 4.13 describes the obtained results on Component B.

Distribution	ℓ	`maxValue`	`Subdivision`	Cycl.`MPM`	Cycl. ref.
Saber					
$\mathcal{D}_5$	25	$5qn$	None	47k	1405k
$\mathcal{U}_q$	36	q^2n	2 calls to Karatsuba	61k	1405k
NTRU					
$\mathcal{U}_q$	34	q^2n	3 calls to Karatsuba	173k	17256k

Table 4.13: Parameters and cost of one multiplication in $R_{q,\delta}$ for Saber and NTRU parameters

Part II

Physical security of lattice-based schemes

Chapter 5

Physical attacks, countermeasures and probing model

Contents

The embedded devices are threatened by physical attacks. These attacks are intended to learn about secret information using the physical properties of the component which executes a code.

Cryptosystems are mathematically proven secure under hardness assumption of a problem. In the case of lattice-based cryptography, the underlying problem is LWE [Reg09] or variants of this problem. The security proofs do not take into account the potential physical perturbations or eavesdrops during an execution. Hence, the physical attacks can be devastating.

Physical security can be approached from two point of views: offensive and defensive. Hence, in this chapter, we firstly describe different kind of physical attacks. Afterwards, we present countermeasures which mitigate these attacks. Finally, we describe a well-known theoretical model which allows to prove that an algorithm is secure, in this model, against side-channel attacks.

5.1 Physical attacks

Physical attacks can be divided in two main categories: side-channel attacks and fault injection. The first side-channel attacks against cryptosystems are carried out by Kocher *et al.* from the years 1996 [Koc96; KJJ99]. One year later the first fault injections against cryptosystems are conducted [BDL97; BS97].

5.1.1 Side-channel attacks

The side-channel attacks use the fact that during an execution, a component uses power, produces electromagnetic emanations, heat, etc. These quantities are component dependent and are linked with the executed algorithm. Therefore, by analyzing these physical quantities an attacker can learn information about secret data. The most well-known physical quantities used are: timing, power consumption and electromagnetic emanation. A side-channel attack can be summarized in two major steps:

1. Acquisition of one or multiple traces of the executed algorithms.
2. Analysis of these traces to deduce information. Statistical tools may be used for the analysis.

5.1.2 Fault injection

The fault injections use the fact that an electric current flow through the transistors during an algorithm. The idea is to disrupt this flow to interfere with the good execution of the algorithm. Afterwards, by analyzing the impact of the interference an attacker can learn information about secret data. The most common alterations during fault injection are:

- Instruction skip. For example, an attacker can skip a verification done in a *if* statement.
- Randomize variable. The definition of variable depends on the CPU architecture and the attacker model. For example a variable could be 8 bits or 32 bits.
- Stuck all the bits of a variable at 1 or 0.
- Bit flipping.

Performing a fault may alter the expected output which can leak information on the manipulated data.

In Chapter 6 we assess the security of some lattice-based cryptosystems against Safe-Error Attack (SEA). Such attacks exploit the fact that a fault injection may or may not lead to a faulty output depending on a sensitive data.

5.2 Countermeasures

In the following we describe three well-known countermeasures against some physical attacks [MOP08].

5.2.1 Masking

Side-channel attacks are based on the fact that physical measurements are correlated with the manipulated sensitive data. Therefore, the idea of masking countermeasure is to split a sensitive data s into at least two shares $s = s_1 + \ldots + s_\alpha$ and the shares are manipulated separately. Moreover, the shares must be uniformly distributed all along the algorithm which requires a significant amount of randomness.

The attacker must combine information on s_1, s_2 to s_α to retrieve any information about s. However, combining information is much more difficult than learning information about one variable. Then, the more information there is to combine, the more difficult the attack is. However, splitting into α shares multiplies at least by α the number of operations.

Therefore, a secure implementation using masking countermeasure must take into account the targeted security and the targeted efficiency to select the best security/efficiency trade-off.

In Chapter 8 we generalize a masking countermeasures which apply for α shares and we implement it in order to secure lattice-based key encapsulation mechanism.

5.2.2 Shuffling

Shuffling countermeasure aims to randomize the execution order of the manipulated secret data. Therefore, it is more difficult for an attacker to deduce which part of the secret is manipulated.

An example is the polynomial multiplication which is a core operation of ideal lattice-based schemes. Let $s(X) = s_0 + \ldots + s_{N-1}X^{N-1}$ be a secret polynomial and $f(X)$ be a public one. The schoolbook multiplication computes in the following order: $s_0 f(X), s_1 f(X)X, \ldots, s_{N-1} f(X)X^{N-1}$. Hence, an attacker knows which coefficient is manipulated. A shuffled polynomial multiplication computes all $s_i f(X)X^i$ in a random order. The random order is modified at each execution.

Shuffling countermeasure requires a lot of random generation and can hinder arithmetic improvement.

5.2.3 Code redundancy

Code redundancy countermeasure duplicates a sensitive operation in order to verify the correct execution of this operation. For example, a polynomial multiplication that implies a sensitive data can be computed twice. Thus, by comparing the results we can detect if at least one of the polynomial multiplication has been tampered. Such a countermeasure is used against fault injections.

5.2.4 Random generation

The masking and shuffling countermeasures require a significant amount of random generation. Hence, we recall how the random generation works on embedded devices.

Embedded devices random generation. The micro-controllers may embed additional hardware to perform sensitive or costly operation. Most of the embedded devices which performs cryptographic operations have a TRNG (True Random Number Generation) hardware. This generator produces byte or multiple bytes of uniformly random numbers (*e.g.* 32 bits of random). Moreover, the TRNG can be used in combination with a Deterministic Random Number Generator (DRNG). A DRNG requires a random seed, which is generated by the TRNG, to generate random numbers.

Random generation modulo q. In the following chapters we need to generate uniform random numbers modulo q which is not a power of two. Hence, we explain our methodology to derive, from a k-bit random number, a random number modulo q. The example $q = 3329$ corresponds to the modulo used in the lattice-based KEM Kyber [BDK+18].

To generate a random integer in $\mathbb{Z}_q$, one can generate a k-bit random number, using the TRNG or DRNG generator, where the gap $2^k - i \cdot q$ is small for some i. Until the random value is not in $[0, i \cdot q[$ we reject it and we sample a new one. This step is called rejection sampling. Using the rejection sampling technique we obtain a uniformly distributed integer modulo q, with rejection probability $1 - i \cdot q/2^k$. For example, with $q = 3329$, one can take $k = 16$ and $i = 19$, with rejection probability 0.035.

In order to decrease the rejection probability, we can also use the trick described in [Lum13, Section 3]. It consists in generating a random integer modulo q^2, which enables to extract two random integers modulo q; we can of course use higher powers of q. As previously, we generate a k-bit random number such that the gap $2^k - i \cdot q^2$ is small. The rejection probability is then $1 - i \cdot q^2/2^k$.

For example, with $q = 3329$, we can use $k = 25$ and $i = 3$, and the rejection probability is 0.009, so we are using 12.5 bits per random integer modulo q, with rejection probability 0.0046 per random integer. We can also use $k = 32$ and $i = 387$, which gives 16 bits per random integer as previously, but with rejection probability 0.0007 per random integer. We describe the pseudo-code below, to be run with parameters $(i, k, q) = (387, 32, 3329)$.

Algorithm 23 randomModq

Input: Parameters i, k, q such that $i \cdot q^2 < 2^k$.
Output: r_1, r_2 uniformly distributed in $\mathbb{Z}_q$.

1: $r := 2^k$
2: **while** $r \geq i \cdot q^2$ **do**
3: $\quad r \xleftarrow{\$} \{0,1\}^k$
4: **end while**
5: $r := r \bmod q^2$
6: **return** $(r \bmod q, \lfloor r/q \rfloor)$

The random generation is also threatened by physical attacks. Then, the previous algorithm and the k-bit random number generation must be securely implemented. In the sequel, we suppose that the random generation is secured.

5.3 Probing model

The probing model is a theoretical view for protecting circuit against adversaries who can probe a number of wires. More precisely, this model is introduced in [ISW03] by Ishai, Sahai and Wagner. They consider that an adversary can probe at most t wires in a circuit. This means that the adversary can learn information about t bits during the algorithm execution. Afterwards, they show how to transform any Boolean circuit into a secure one by splitting the secret information into $\alpha = 2t + 1$ shares. Such a model can be used to prove the security of an implementation against side-channel attacks. Indeed, the t wires can be seen as the side-channel leakage points of the algorithm.

In the article [RP10], the authors explain that a wire can be extended over any field of characteristic 2. Moreover the authors improve the previous result by showing that $\alpha = t + 1$ shares is enough to obtain the targeted security. In the following we consider that an attacker can learn information about t variables.

Later on in [BBD+16] Barthe *et al.* the authors introduce, in the probing model, two security notions: Strong Non-Interference (SNI) and Non-Interference (NI), which facilitate the security proofs. In the following we describe the NI and SNI definitions.

5.3.1 Security definitions: NI and SNI.

The following definitions use the term *gadget*. A gadget is a probabilistic algorithm which means it is an algorithm using a source of randomness during it execution. Moreover, the output depends on the sampled random numbers. Examples of gadgets are described in Algorithms 24, 25, 26.

When a gadget is proven t-NI or t-SNI secure, then an attacker can learn information about the secret only if he can probe at least $t+1$ variables. This means that any set of at most t intermediates variables is independent from at least one share of the secret. In this case we say that the t variables can be perfectly simulated.

Definition 1 (t-NI security)**.** *Let G be a gadget taking as input $(x_i)_{1 \leq i \leq \alpha}$ and outputting the vector $(y_i)_{1 \leq i \leq \alpha}$. The gadget G is said t-NI secure if for any set of $t_1 \leq t$ intermediate variables, there exists a subset I of input indices with $|I| \leq t_1$, such that the t_1 intermediate variables can be perfectly simulated from $x_{|I}$.*

Definition 2 (t-SNI security)**.** *Let G be a gadget taking as input α shares $(x_i)_{1 \leq i \leq \alpha}$, and outputting n shares $(z_i)_{1 \leq i \leq \alpha}$. The gadget G is said to be t-SNI secure if for any set of t_1 probed intermediate variables and any subset $\mathcal{O}$ of output indices, such that $t_1 + |\mathcal{O}| \leq t$, there exists a subset I of input indices that satisfies $|I| \leq t_1$, such that the t_1 intermediate variables and the output variables $z_{|\mathcal{O}}$ can be perfectly simulated from $x_{|I}$.*

The SNI definition is stronger than NI in that the number of input shares required for the simulation only depends on the number of internal probes, and not on the number of output shares that must be simulated. The main benefit of t-SNI security definition is that it allows proof by composition. Indeed, by proving the t-SNI property of individual gadgets, we obtain that the full circuit is secure against t probes using $\alpha = t + 1$ shares.

If a gadget only satisfies the NI definition, we can apply some SNI mask refreshing (like Algorithm 5.3.4) on the outputs and then the resulting gadget becomes SNI (see [BBD+16]). In the sequel all our gadgets will be proven either NI or SNI.

In the following we describe three well-known gadgets that we use in the next chapters.

5.3.2 The SecAnd algorithm

The SecAnd Algorithm 5.3.2 enables to compute the logical AND between two Boolean masked values. Let $x = x_1 \oplus \ldots \oplus x_\alpha$ and $y = y_1 \oplus \ldots \oplus y_\alpha$. Then the SecAnd computes $x\&y$ using operation on the values x_i and y_i.

Algorithm 24 SecAnd

Input: $k \in \mathbb{N}$, $x_1, \ldots, x_\alpha \in \{0,1\}^k$, $y_1, \ldots, y_\alpha \in \{0,1\}^k$

Output: $z_1, \ldots, z_\alpha \in \{0,1\}^k$, with $\bigoplus_{i=1}^{\alpha} z_i = (\bigoplus_{i=1}^{\alpha} x_i) \wedge (\bigoplus_{i=1}^{\alpha} y_i)$

1: for $i = 1$ **to** α **do** $z_i \leftarrow x_i \wedge y_i$
2: **for** $i = 1$ to α **do**
3: **for** $j = i + 1$ to α **do**
4: $r \xleftarrow{\$} \{0,1\}^k$
5: $r' \leftarrow (r \oplus (x_i \wedge y_j)) \oplus (x_j \wedge y_i)$
6: $z_i \leftarrow z_i \oplus r$
7: $z_j \leftarrow z_j \oplus r'$
8: **end for**
9: **end for**
10: **return** $z_1, \ldots, z_\alpha$

This algorithm is a variant with k-bit words of the original ISW algorithm [ISW03]. Its asymptotic complexity is $\mathcal{O}(\alpha^2)$ with a number of operations:

$$\mathcal{C}_{\mathsf{SecAnd}}(\alpha) = \alpha(7\alpha - 5)/2$$

Lemma 2 ([BBD+16]). *The* SecAnd *algorithm is* $(\alpha - 1)$-SNI.

5.3.3 Secure multiplication modulo q

The SecMult Algorithm 5.3.3 computes a modular multiplication modulo a prime q between two arithmetically masked integers. Let $x = x_1 + \ldots + x_\alpha \mod q \in \mathbb{Z}_q, y = y_1 + \ldots + y_\alpha \mod q \in \mathbb{Z}_q$, then $\mathsf{SecMult}((x_1, \ldots, x_\alpha), (y_1, \ldots, y_\alpha)) = (z_1, \ldots, z_\alpha)$ where $z_1 + \ldots + z_\alpha \mod q = xy \mod q$. This algorithm is introduced in [SPO+19].

Algorithm 25 SecMult

Input: $x_1, \ldots, x_\alpha \in \mathbb{Z}_q$, $y_1, \ldots, y_\alpha \in \mathbb{Z}_q$
Output: $z_1, \ldots, z_\alpha \in \mathbb{Z}_q$ such that $\sum_i z_i = (\sum_i x_i) \cdot (\sum_i y_i) \bmod q$.

for $i = 1$ **to** α **do** $z_i \leftarrow x_i \cdot y_i \bmod q$
for $i = 1$ to α **do**
  for $j = i + 1$ to α **do**
    $r \xleftarrow{\$} \mathbb{Z}_q$
    $r' \leftarrow (r + x_i \cdot y_j \bmod q) + x_j \cdot y_i \bmod q$
    $z_i \leftarrow z_i - r \bmod q$
    $z_j \leftarrow z_j + r' \bmod q$
  end for
end for

Note that the number of operations of SecMult is $\alpha \cdot (7\alpha - 5)/2$ by considering random generation in $\mathbb{Z}_q$, addition and multiplication modulo q as a single operation.

Lemma 3 ([SPO+19]). *The* SecMult *algorithm is* $(\alpha - 1)$-SNI.

5.3.4 Mask refreshing

The RefreshMasks Algorithm 26 allows to refresh the shares of a masked value over any finite field $\mathbb{F}$. Let $a = a_1 + \ldots + a_\alpha$, $\mathsf{RefreshMasks}(a_1, \ldots, a_\alpha) = c_1 + \ldots + c_\alpha$ where $c_1 + \ldots + c_\alpha = a$. Such algorithm can be used on the outputs of a NI gadget to make it SNI.

Algorithm 26 RefreshMasks

Input: $a_1, \ldots, a_\alpha$
Output: $c_1, \ldots, c_\alpha$ such that $\sum_{i=1}^{\alpha} c_i = \sum_{i=1}^{\alpha} a_i$

For $i = 1$ **to** α **do** $c_i \leftarrow a_i$
for $i = 1$ to α **do**
  for $j = i + 1$ to α **do**
    $r \xleftarrow{\$} \mathbb{F}$
    $c_i \leftarrow c_i + r$
    $c_j \leftarrow c_j - r$
  end for
end for
return $c_1, \ldots, c_\alpha$

The asymptotic complexity of Algorithm 26 is $\mathcal{O}(\alpha^2)$, with a number of operations:

$$\mathcal{C}_{\mathsf{refresh}}(\alpha) = 3\alpha(\alpha - 1)/2$$

Lemma 4 ([BBD+16]). *The* RefreshMasks *algorithm is* $(\alpha - 1)$-SNI.

Chapter 6

Safe-error analysis of post-quantum cryptography mechanisms

Contents

The results presented in this chapter are from a joint work with Luk Bettale and Guénaël Renault in [BMR21].

In order to protect the implementations against side-channel attacks, the NIST asks for constant-time implementation. This may be enough for implementations on a server, but insufficient for embedded devices. Contrarily to cryptography deployed in a distant server, embedded devices can be physically accessed by an attacker. The attacker could easily perform side-channel analysis [Koc96; KJJ99; BCO04], or fault injection [BDL97; BS97] in order to recover the embedded secrets.

In this chapter we assess the security of some lattice-based schemes against safe-error attack.

Safe-error analysis of post-quantum cryptography mechanism.

Fault attacks have been very useful to break embedded cryptosystems. Many powerful fault attacks have been described in the literature [JT12]. Safe-Error Attack (denoted SEA) [YJ00; BCG12; Cla12] is a powerful way to exploit fault injection, especially on constant-time implementation such as the ones proposed to the NIST. Such attacks exploit the fact that a specific fault may or may not lead to a faulty output depending on a secret value.

Previous works. An independent work in [PP21] performs SEA during the decryption algorithm of some lattice-based candidates (Kyber and Newhope). More precisely, they perform a fault injection during the reconciliation step. If the outcome is not modified, then they learn information depending on the secret

key. By gathering such information they retrieve the entire secret key. Their SEA attack focuses on the error distribution while our work focuses on the secret distribution. Our work proposes a different way of performing SEA applied to more schemes than [PP21] and thus it is complementary.

To the best of our knowledge, no other public researches on fault attacks in the PQC context (e.g. [KY11; KY13; VOG+18; VPR19; BBK16]) consider SEA. In the following we investigate the security of PQC against such attacks.

Our contribution. In the Sequel we provide an overall analysis of the resilience of NIST lattice-based finalist schemes (Dilithium, Kyber, NTRU, Saber) against safe-error attacks. More precisely, we describe an attack path to retrieve the 0 coefficients of the secret keys using fault injection and we determine the security impact of such knowledge.

6.1 Framework description

6.1.1 Attacker model

For a safe-error attack to be effective, an attacker has to know first how a specific fault impacts an implementation. This knowledge is usually acquired by characterising his attack on a similar device as the victim. To obtain the best characterisation possible, the attacker needs devices where he can *know/set* the manipulated secrets. For example, this assumption could be obtained by using an *open sample*. Then, the attacker has to find a fault that will lead to a faulty output only if some conditions on a secret value are fulfilled.

Safe-error attacks are particularly useful against constant-time implementations that are designed to resist timing side-channel attacks. A typical example is a regular implementation of a square-and-multiply algorithm used for modular exponentiation in RSA:

Algorithm 27 `Square and multiply`

Input: $m, N, d = \sum_{i=0}^{n-1} d_i\, 2^i$
Output: $r = m^d \bmod N$
1: **for** $i = n-1$ **to** 0 **do**
2: $\quad r_1 \leftarrow r_1^2 \bmod N$
3: $\quad r_{d_i} \leftarrow r_1 \times m \bmod N$
4: **end for**
5: **return** r_1

The operation at line 3 is useful only if d_i equals 1. More precisely, it is just a dummy operation if d_i equals 0 and does not contribute to the output computation. Thus, faulting line 3 will lead to a faulty ciphertext only if $d_i = 1$. Hence, if a fault can skip this operation, the attacker can, step by step, detect all the zeroes of the exponent d.

The fault has not to be necessarily an instruction skip. For instance, an attacker could modify a value during a RAM access. In the previous example, if the attacker can set to 0 the value of d_i when it is read, one obtains the same result.

Thus, throughout this section, we assume that the attacker can:

- Precisely set the fault to a target operation ;
- Skip an instruction or function call
- Or set a variable to 0.

To keep our assumptions closed from practical fault attacks, we do not suppose that the attacker can set a variable to a chosen value different from zero.

We will see that in our PQC context, these assumptions are sufficient to let an attacker retrieve the positions of all the null coefficients in a secret key.

6.1.2 Safe-error attack on lattice-based cryptography

The central operation in lattice-based cryptography is the polynomial multiplication. Such operation usually involves a secret polynomial with coefficients close to 0. Such operation must be performed in a timing-resistant manner to be secure against timing attacks. As described earlier, this context is usually a perfect fit for safe-error attacks.

Moreover, most of the lattice-based NIST finalist candidates use a centered binomial distribution or a uniform distribution in $\{-1, 0, 1\}$ ensuring that 0-coefficients are numerous. Hence, fault injections setting a secret coefficient to 0 at a precise operation, reveal information about secret 0-coefficients positions. Our study focuses on the NIST lattice-based Key Exchange Mechanisms (denoted KEMs) and a signature. Then, the fault injection is performed during a decryption or a signature algorithm. Therefore, if the obtained plain text or signature is correct despite of a fault injection on a secret coefficient then, that ensures this coefficient is a 0.

Finding the positions of the zeroes may not be enough to recover the key, but it surely reduce the security of the scheme. In order to precisely measure the security loss, we need to use an estimation tool and compare our result with the claimed security.

6.1.3 Security analysis of lattice-based cryptography

The NIST call for post-quantum safe cryptography has mentioned 5 security categories: 1 to 5. The categories 1/3/5 corresponding to a claimed security equivalent, in terms of computational capabilities, of key search on a block cipher AES-128/192/256. The categories 2/4 corresponding to a claimed security equivalent, in terms of computational capabilities, of collision search on a 256/384-hash functions as SHA3-256/384.

The main goal of these categories is to facilitate security comparison between all the PQC proposals. However, some candidates under-estimate and others over-estimate these security categories. Then, the security comparison between these algorithms is an arduous task. In the following, we refer to these categories only to estimate how proposals securities downgrade due to safe-error attacks.

6.1.4 Security estimation loss

In the following, the attacker performs SEA during the execution of some lattice-based KEM or signatures to learn some information about the secret key. To estimate the security loss, we use the toolkit introduced in [DDG+20]. This tool is a security estimation framework for lattice-based schemes under lattice reduction attack when an attacker obtains some "*hints*" about the secret key. The "*hints*" can be obtained from side-channel attacks or, in our case, from fault injection.

Let s be a secret vector and v, l, k be parameters known by the attacker. This tools can handle 4 types of hints:

- Perfect hints: $\langle s, v \rangle = l$.
- Modular hints: $\langle s, v \rangle = l \mod k$.
- Approximate hints: $\langle s, v \rangle = l + e$, where e is an error following a known distribution.
- Short vector hints: $v \in \Lambda$, where Λ is a lattice related to the secret s.

These hints can be used to estimate cryptosystems security relying on LWE, LWR and NTRU problems. The security estimation is done using a unit called *bikz*, denoted by β. This unit corresponding to the BKZ-β used to solve the Distorted Bounded Distance Decoding (DBDD) instance associated to the secret. As mentioned in the toolkit article [DDG+20], there is no *bikz*-to-bit exact conversion. Then, in the following, security estimations are done in terms of *bikz* β or NIST security categories.

To estimate the cryptosystems security, this tool requires to instantiate a DBDD instance related to our secret vector. Hence, some parameters are required: the secret's length and distribution $(N, \mathcal{D}_\sigma)$, the error's length and distribution $(m, \mathcal{D}_{\sigma_e})$, and the modulo q. Our script using this toolkit is available at [Monb].

In Sect. 6.2, we present an attack description as well as the required parameters to instantiate the tool. We use it to provide a security loss estimation for each attacked cryptosystems.

6.2 Application on post-quantum cryptography

6.2.1 NTRU

NTRU [CDH+19] is a KEM based on the eponymous *Nth degree Truncated Polynomial Ring Units* problem which is assimilated to a lattice-based one. The secret polynomial f has coefficients belonging to $\{-1, 0, 1\}$ uniformly distributed and the public polynomial $h = (3gf_q) \mod (q, \phi_1\phi_N)$ has coefficients modulo q, where $\phi_1 = X - 1, \phi_N = (X^n - 1)/(X - 1)$. The polynomials manipulated in NTRU are defined in $R_{q,-1} = \mathbb{Z}_q[X]/(X^N - 1)$ with $N \in \{509, 677, 821, 701\}$ and $q \in \{2048, 4096, 8192\}$ depending on the security level. In this proposal, we focus the decryption function:

Algorithm 28 `NTRU Decryption`

Input: $((f, f_p, h_q), c)$
Output: r, m
1: if $c \not\equiv 0 \mod (q, \phi_1)$ return$(0, 0, 1)$
2: $a \leftarrow (c \cdot f) \mod (q, \phi_1\phi_N)$
3: $m \leftarrow (a \cdot f_p) \mod (3, \phi_N)$
4: $m' \leftarrow \text{Lift}(m)$
5: $r \leftarrow ((c - m') \cdot h_q) \mod (q, \phi_N)$
6: **if** $(r, m) \in \mathcal{L}_r \times \mathcal{L}_m$ **then**
7: **return** $(r, m, 0)$
8: **else**
9: **return** $(0, 0, 1)$
10: **end if**

SEA application Our objective is to learn the zero coefficients positions of f during the computation at line 2: $a \leftarrow (c \cdot f) \mod (q, \phi_1\phi_N)$. In the reference implementation, the secret coefficients are stored in a 16-bit integer (`uint16_t`) and the polynomial multiplication is done with the algorithm described hereafter:

Algorithm 29 `NTRU Polynomial multiplication`

Input: a, c, f {all $a[i], c[i], f[i]$ are `uint16_t`}
Output: a

1: **for** $k = 0$ to N **do**
2: $a[k] \leftarrow 0$
3: **for** $i = 1$ to N **do**
4: $a[k] \leftarrow a[k] + c[k + i] \times f[N - i]$
5: **end for**
6: **for** $i = 0$ to $k + 1$ **do**
7: $a[k] \leftarrow a[k] + c[k - i] \times f[i]$
8: **end for**
9: **end for**
10: **return** a

The fault injection is performed during the computation $a[k] \leftarrow a[k] + c[k + i] \times f[N - i]$. If the decryption succeeds then the coefficient $N - i$ is equal to 0. This attack is performed during several decryption procedures to recover all the coefficients of f that are equal to 0.

Security impact Using the tool introduced in Section 6.1 we can determine the security impact of this knowledge.

The column "Classical" determines the security level in terms of "bikz β" of the NTRU cryptosystem without any knowledge. The "Classical" NTRU DBDD instance is done with the following parameters:

- The length of f: N
- The distribution of f: coefficients uniform over $\{-1, 0, 1\}$
- The distribution for g (see NTRU article [CDH+19])

The column "Attacked" determines the security level of NTRU with the knowledge of $N/3$ secret coefficients. Indeed, we suppose that the secret coefficients are well distributed, then f has $N/3$ coefficients equal to 0. The "Attacked" NTRU DBDD instance is done with the following parameters:

- The secret vector length without the known coefficients: $N = N - N/3$.
- A uniform distribution over two elements.
- The same distribution for g.

	Classical	Attacked
NTRU HPS 1 $N = 509, q = 2048$	Dim $= 1018$ $\beta = 172.15$	Dim $= 680$ $\beta = 95.53$
NTRU HPS 2 $N = 677, q = 2048$	Dim $= 1354$ $\beta = 249.95$	Dim $= 904$ $\beta = 146.20$
NTRU HPS 3 $N = 821q, q = 4096$	Dim $= 1642$ $\beta = 308.42$	Dim $= 1096$ $\beta = 183.35$
NTRU HRSS $N = 701, q = 8192$	Dim $= 1402$ $\beta = 236.30$	Dim $= 936$ $\beta = 135.96$

In average the safe-error attack provides, in terms of *bikz*, a 42% security loss. The NTRU specification [CDH+19] mentioned that the NTRU HPS 1 security is lower than an actual AES-128 security. However, for all parameters the SEA brings the bikz security under NTRU HPS 1 original security. Therefore, with this attack an attacker can reduce the security of NTRU cryptosystem under 2^{128} security bits in a pre-quantum world and under 2^{64} security bits in a quantum world.

6.2.2 Saber

Saber [BMD+21] is a MLWR lattice-based KEM. The secret vector $S = (s_1, \ldots, s_k)$, where s_i's are polynomials following a centered binomial distribution $\mathcal{D}_\sigma$ in $R_{q,1} = \mathbb{Z}_q[X]/(X^N + 1)$. Depending on the security level, the Saber parameters are: $q = 2^{13}, p = 2^{10}, N = 256, \sigma \in \{5, 4, 3\}$ and $k \in \{2, 3, 4\}$. We focus the decryption function for applying SEA.

Algorithm 30 `Saber Decryption`

Input: $S, c = (c_m, b')$
Output: m'
1: $v = b'^T \cdot (S \mod p) \in R_{p,1}$
2: $m' = ((v - 2^{\epsilon_p - \epsilon_T} c_m + h_2) \mod p) >> (\epsilon_p - 1) \in R_{2,1}$
3: **return** m'

SEA application Our objective is to determine the positions of the zero coefficients in S. One way is to perform a fault injection during the computation in line 1: $v = b'^T \cdot (S \mod p) \in R_p$. However, as the polynomial multiplication is computed with a 4 way Toom-cook algorithm, the secret coefficients are combined between each other. This makes the attack difficult in practice.

To overcome this issue, the attack can be done during the conversion of the secret input. Indeed, the coefficients of the secret polynomial are stored as a compact byte string and it is manipulated as a polynomial structure which encodes each secret coefficients to an `uint16_t`. Our attack focus on the byte-to-poly conversion.

Algorithm 31 `Saber Byte-to-Poly function`

Input: (`uint8_t` $* sk$, `uint16_t` $* S$)
Output: S
1: **for** $j = 0$ to $j < N/8$ **do**
2: os_sk, os_s $= 13\,j,\ 8\,j$
3: S[os_s] $= sk$[os_sk] | ((sk[os_sk + 1]&0x1F) << 8)
4: S[os_s + 1] $= \ldots$
5: $\ldots$
6: S[os_s + 7] $= \ldots$
7: **end for**
8: **return** s

We present here, for the sake of clarity, only a sketch of the reference algorithm. The main idea is that each coefficient of S is determined by multiple bytes from sk. Thus, the fault injection can be done during this transformation. If the decryption algorithm succeeded then, the faulted coefficient is equal to 0. As in NTRU, this attack is performed during several decryption procedure over different coefficients to recover all the 0-coefficients positions.

Security impact Depending of the centered binomial distribution parameters σ, the secret vector has more or less 0. As previously, we suppose that our secret is well distributed. Hence, for the three possible values of σ (5,4 or 3 respectively), the ratio of secret coefficients equal to 0 is $24,6\%$, $27,3\%$, and $31,25\%$ respectively.

The "Classical" LWR DBDD instance is done with the following parameters:

- The secret vector length: N,
- The secret vector centered binomial distribution: $\mathcal{D}_\sigma$,

- The length of the rounding: $m = N$.

The "Attacked" LWR DBDD instance is done with the following parameters:

- The secret vector length without the known coefficients: $N = N - aN$, where a is 0.246, 0.273 or 0.3125.
- The previous distribution without the 0-coefficients,
- The length of the rounding m (unchanged).

The following table shows the security impact of such knowledge using the tool introduced in Section 6.1.

	Classical	Attacked
Light Saber $N, m = 512, \sigma = 5$	Dim $= 1025$ $\beta = 404.38$	Dim $= 900$ $\beta = 292.05$
Saber $N, m = 768, \sigma = 4$	Dim $= 1537$ $\beta = 648.72$	Dim $= 1328$ $\beta = 462.57$
Fire Saber $N, m = 1024, \sigma = 3$	Dim $= 2049$ $\beta = 892.21$	Dim $= 1729$ $\beta = 613.26$

In average the safe-error attack provides, in terms of *bikz*, a 30% security loss. The SABER specification [BMD+21] describes the security level in terms of AES-128/192/256. The following table present the specified security and the obtained one after SEA:

	Security claimed	SEA security
Light Saber	AES-128	$\leq$ AES-128
Saber	AES-192	$\approx$ AES-128
Fire Saber	AES-256	$\approx$ AES-192

Remark 12. *The NTRU HPS 3 claimed an equivalent AES-192 security as Saber. However, the NTRU HPS 3 value of β is 308.42 whereas Saber's value of β is 648.72. This main difference is due to the secret coefficients distribution. Indeed, NTRU cryptosystem underestimates the theoretical security by lake of concrete attack, rather than Saber cryptosystem overestimate it.*

6.2.3 Dilithium

Dilithium [BDL+21] is a lattice-based signature, based on the MLWE problem. The secret is given by (S_1, S_2) two vectors of l and k polynomials respectively. Their polynomials have coefficients following a centered binomial distribution $\mathcal{D}_\sigma$ in $R_{q,1} = \mathbb{Z}_q[X]/(X^N + 1)$. Depending of the security level, $q = 8380417, N = 256, \sigma \in \{2, 4\}, (k, l) \in \{(4, 4), (6, 5), (8, 7)\}$

SEA attack The polynomial multiplication is performed with the NTT algorithm. However, for compactness, the secret vectors are not stored in the NTT domain. One way to attack the secret coefficients, is to perform fault injection during the NTT computation. However as for Saber, it is easier to attack the secret coefficients during the unpacking process:

Algorithm 32 `Dilithium unpack`

Input: $(\texttt{uint8\_t} * sk, \texttt{uint16\_t} * s, \eta)$
Output: s

1: **for** $j = 0$ to $j < N/8$ **do**
2: $\quad s[8\,i] = (sk[3\,i] >> 3)\&7$
3: $\quad s[8\,i+1] = (sk[3\,i] >> 3)\&7$
4: $\quad \ldots$
5: $\quad s[8\,i+7] = (sk[3\,i+2] >> 5)\&7$
6: $\quad s[8\,i] = \eta - s[8\,i]$
7: $\quad s[8\,i+1] = \eta - s[8\,i+1]$
8: $\quad \ldots$
9: $\quad s[8\,i+7] = \eta - s[8\,i+7]$
10: **end for**
11: **return** s

Here, the fault injection is performed as of line 6, depending of the targeted secret coefficient. For example a fault injection at line 7 ensures that the $8\,i+1$ coefficient is stored as a 0. The attack on the previous algorithm is similar to the one on Saber. The fault injection is repeated on each coefficient during several signature procedures. If the signature is not different than the original one, then the coefficient is 0. This attack is performed on S_1 and S_2.

Security impact The number of 0 depends on the value σ. Hence, for $\sigma = 4$ (resp. $\sigma = 2$), $27,3\%$ (resp. $37,5\%$) of the secret coefficients of S_1 and S_2 are equal to 0. The following table shows the security impact of such knowledge using the tool introduced in Sect. 6.1. The parameter N (resp. m) corresponds to the number of coefficients in S_1 (resp. S_2). The "Classical" LWE DBDD instance is done with the following parameters:

- The length of S_1: N.
- The distribution of S_1: centered binomial distribution $\mathcal{D}_\sigma$.
- The length of S_2: m,
- The distribution of S_2: centered binomial distribution $\mathcal{D}_\sigma$ (same as S_1).

The "Attacked" LWE DBDD instance is done with the following parameters:

- The length of S_1 without the known coefficients: $N = N - aN.$ where a = 0.273 or 0.375.
- The distribution of S_1 and S_2 without the 0-coefficients.
- The length of S_2 without the known coefficients: $m = m - am.$ where a is 0.273 or 0.375.

The following table shows the security impact of such knowledge using the tool introduced in Section 6.1.

	Classical	Attacked
Dilithium 1 $(N, m) = (1024, 1024)$ $\sigma = 2$	Dim $= 2049$ $\beta = 348.84$	Dim $= 1281$ $\beta = 192.84$
Dilithium 2 $(N, m) = (1280, 1536)$ $\sigma = 4$	Dim $= 2817$ $\beta = 499.65$	Dim $= 2049$ $\beta = 340.06$
Dilithium 3 $(N, m) = (1792, 2048)$ $\sigma = 2$	Dim $= 3841$ $\beta = 717.52$	Dim $= 2401$ $\beta = 411.13$

Due to small centered binomial distributions, the security impact of SEA is consequent for Dilithium. The Dilithium specification [BDL+21] describes security level in terms of equivalence of SHA3-256/AES-192/AES-256. The following table presents the specified security and the one obtained after SEA:

	Security claimed	SEA security
Dilithium 1	SHA3-256	$\leq$ SHA3-256
Dilithium 2	AES-192	$\approx$ SHA3-256
Dilithium 3	AES-256	$\leq$ AES-192

6.2.4 Kyber

Kyber [BDK+18] is a MLWE lattice-based KEM. The secret vector $S = (s_1, \ldots, s_k)$, where, s_i is a polynomial with coefficients following a centered binomial distribution $\mathcal{D}_\sigma$ in $R_{q,1} = \mathbb{Z}_q[X]/(X^N + 1)$. In this proposal $q = 3329$, $N = 256$, $\sigma \in \{3, 2\}$ and $k \in \{2, 3, 4\}$ depending of the security level. Again we focus its decryption function:

Algorithm 33 `Kyber Decryption`

Input: sk, c
Output: Message m
1: $u \leftarrow \text{Decompress}_q(\text{Decode}(c), d_u)$
2: $v \leftarrow \text{Decompress}_q(\text{Decode}(c + \text{offset}), d_u)$
3: $\hat{s} \leftarrow \text{Decode}(sk)$
4: $\tilde{c} \leftarrow v - \text{NTT}^{-1}(\hat{s}^T \cdot \text{NTT}(\text{u}))$
5: $m \leftarrow \text{Encode}(\text{Compress}(\tilde{c}), 1)$
6: **return** m

However, in this scheme the secret coefficients of all the s_i's are stored in the NTT domain. Then, the structure of the binomial distribution cannot be exploited to perform a stuck-at 0 during the computation of $\hat{s}^T \circ \text{NTT}(\text{u})$. In fact, the NTT domain ensures that the coefficients are values in $\{0, \ldots, q-1\}$ rather than $\{-\sigma, \ldots, \sigma\}$. Hence, by design, Kyber decryption algorithm is safe against our SEA at a cost of extra memory storage.

6.3 Countermeasures

The lattice-based cryptosystems are vulnerable to safe-error attacks in general. Indeed, most of their distribution are small and the null coefficients are numerous. Being able to determine their positions provide an important advantage to an attacker.

One way to protect these schemes is to mask the distribution as in Kyber proposal. However, all the lattice-based algorithm are not compliant with the NTT domain. Another way would be to mask the secret coefficients by a uniform random in $\mathbb{Z}_q$. Then, the manipulated secret coefficients would be random in $\{0, \ldots, q\}$ rather than in $\{-\sigma, \ldots, \sigma\}$, where σ is close to 0. The main drawbacks of masking are the extra storage for the coefficients, and the larger cost of the central operation: polynomial multiplication.

Another possible countermeasure would be the shuffling. Indeed, this countermeasure ensures, in theory, that the positions of the null coefficients would not leak. However this technique requires a significant amount of random generation.

Chapter 7

Exploiting physical attacks

Contents

The results present in this chapter are from a joint work with Aurélien Greuet and Guénaël Renault in [GMR20].

The lattice-based Key Encapsulation Mechanisms (KEMs) of the NIST standardization are proven secure according two semantic security notions [Sak11]:

- INDinstiguishability under Chosen Plaintext Attack (IND-CPA).
- INDinstiguishability under Chosen Ciphertext Attack (IND-CCA).

Chosen plaintext attack considers that an attacker can choose plaintext to be encrypted and he can obtain the corresponding ciphertext. The IND-CPA security notion ensures that an attacker cannot learn information about the secret key using chosen plaintext attack. When a KEM is only proven IND-CPA secure, then the key pair must be refreshed at each encapsulation.

Chosen cipher attack considers that an attacker can have access to plaintexts obtained by decryption of chosen ciphertexts. The IND-CCA security notion ensures that an attacker cannot learn information about the secret key using such attack. This security notion is stronger than the IND-CPA one. When a KEM is IND-CCA secure, then the key pair can be used for several encapsulations.

In 1999 E. Fujisaki and T. Okamoto [FO99] introduce a transformation that allows to convert a CPA secure cryptosystem to a CCA secure one. This transformation or variants of this one, are widely used in the lattice-based schemes to convert a CPA secure KEM to a CCA secure one.

In this chapter we study the importance of Fujisaki Okamoto (FO) [FO99] transformation to ensure CCA security of lattice-based schemes. More precisely we study this transformation in the context of LAC [XYD+19], a lattice-based key encapsulation mechanism. In the following we evaluate the transformation security in two scenarios:

- In a key misuse situation. We suppose that an implementer does not respect the CPA key exchange implementation guideline such that he does not refresh the secret key at each key exchange. Therefore, we mount an attack using forged ciphertext to retrieve the secret key. (Section 7.1)

- In a physical attack context. We suppose that the CCA key exchange is implemented without countermeasure to protect the FO transformation. Therefore in order to apply the previous attack, we determine which side-channel information an attacker needs. (Section 7.2)

LAC scheme and misuse situation. In this chapter we study LAC [XYD+19], a RLWE candidate to the NIST standardization process until round 2. Moreover, it is a winner of the Chinese post-quantum competition [fCry20b]. It differs from other RLWE KEMs by its small key and ciphertext sizes, for an equivalent security level. Such small sizes can be an advantage, particularly in constrained environments and embedded systems. We focus on LAC.KE, a KEM based on the CPA secure public-key cryptosystem LAC.CPA. In constrained environments it's interesting to determine the impact of key caching to evaluate the requirement of random generation. Furthermore, the specification of CCA version of LAC uses a static secret key due to security provided by the Fujisaki Okamoto (FO) transformation [FO99]. However, as shown in [OSP+16; BGR+19] without a secure implementation of FO transformation, a physical attack can bypass security provided by FO and modifies a CCA version to a CPA one with a static secret key. Our study is inspired by previous works in [BGR+19; LZZ18; QCD19], which evaluate the resilience in a misuse context offered by two other NIST KEM candidates. Here we propose to pursue this evaluation with another NIST candidate to determine which one is the more resilient against this kind of attack.

Previous works. The seminal work of Menezes and Ustaoglu [MU10] paved the way for active attacks on KE protocols. The idea of key mismatch attack on LWE based key exchange was first proposed by Fluhrer in [Flu16; DFR18]. In a key mismatch attack, a participant's secret key is reused for several key establishments, and his private key can be recovered by comparing the shared secret key of the two participants.

Some lattice-based KEM of the NIST competition were analysed in the key reused context using a key mismatch oracle. In [BDH+19], Baetu *et al.* proposed a generic attack for several algorithms using the same structure called meta-algorithm. However, most of the algorithms attacked in [BDH+19] did not pass the first round of the submission, except Frodo-640 and NewHope512. However in [HV20], Huguenin-Dumittan *et al.* pursue the work of generic attack for round 2 candidates. The security of NewHope1024 CPA algorithm in this misuse scenario is analyzed by Bauer *et al.* in [BGR+19] and an improvement is proposed in [LZZ18]. More recently, in the same context, an attack on Kyber CPA KEM is proposed by Ding *et al.* [QCD19].

In [GJY19], Guo *et al.* presented an attack against the CCA version of LAC. This attack is theoretically stronger than ours since it does not rely on a misuse hypothesis but it requires 2^{162} pre-computations that cannot be achieved in practice.

Our contribution. In Section 7.1, we investigate the resilience of the LAC KEM under a misuse case: we assume that the same secret key is reused for multiple key establishment and we assume that an attacker can use a key mismatch oracle as introduced in [BGR+19].

Since LAC uses encoding and compression functions different from a classical RLWE scheme, Fluhrer's attack [Flu16] cannot be applied directly. Furthermore, these functions are different from those used in NewHope or Kyber, so we cannot apply straightforwardly the attacks described in [BGR+19; LZZ18; QCD19]. A recent independent work in [HV20] attacks several round 2 candidates using the generic structure of these schemes. Their attack is applied to the first security level of LAC but is focused on the theoretical aspect. Our work complete this work by bringing a practical aspect and an extension to the others security levels.

The main idea of these attacks is to send forged ciphertexts to a victim, ensuring that its decryption will leak partial information of his static secret key. LAC algorithms use two encoding functions including an error-correction code BCH that can correct a limited number of errors. If a message exceeds the number

of errors that the error-correction code can correct, then a decryption failure occurs. Thus, we propose to use this failure to provide leaks about the static secret key.

More precisely, we propose a deterministic key mismatch attack on LAC KE for the first two security levels: LAC-128 and LAC-192, which required at most 2 queries per coefficient of the secret key. Afterwards, we adapt our attack to the highest security level LAC-256 which is still deterministic but we need at most 8 queries per coefficient of the secret key.

We experimented our attack with the reference and optimized implementation in `C` provided by the LAC team [XYD+19] with parameters described in Section 7.1.1. The code of our attack is available in [Mona].

In Section 7.2 we suppose that there is no misuse. Hence, due to the CCA security brings by the Fujisaki-Okamoto transformation, the previous attack cannot applied straightforwardly . Therefore, we analyze leakage points which can be exploited to mount a side-channel attack to thwart the FO security and then retrieve secret coefficients.

7.1 Attack on LAC CPA key exchange in misuse situation

7.1.1 Preliminaries

Notation

Ring definition. For an integer $q \geq 1$, let $\mathbb{Z}_q$ be the residue class group modulo q such that $\mathbb{Z}_q$ can be represented as $\{0, \ldots, q-1\}$. We define $R_{q,1}$ being the polynomial ring $R_{q,1} = \mathbb{Z}_q[X]/(X^N + 1)$.

Polynomial. A polynomial in $R_{q,1}$ is of degree at most $(N-1)$ with coefficients in $\mathbb{Z}_q$. Given $P \in R_{q,1}$, we denote by $P[i]$ or P_i the coefficient associated with the monomial X^i. P can also be represented as a vector with N coordinates. In the following, the notation $(a)_{l_v}$ ($l_v \in \mathbb{N}$), where a is a vector (or a polynomial) of dimension $N > l_v$, means we keep the first l_v coordinates of a.

Message space. Let the message space $\mathcal{M}$ be $\{0,1\}^{l_m}$ and the space of random seeds $\mathcal{S}$ be $\{0,1\}^{l_s}$, where l_m and l_s are two integer values.

Random distribution. Let ψ_σ be the centred binomial distribution on the set $\{-1, 0, 1\}$. We denote the centred binomial distribution for N independent coordinates by ψ_σ^N i.e. for a vector a of dimension N each coefficient is sampled with the centred binomial distribution. In LAC algorithms we use:

1. $\psi_1 : Pr(x = 0) = \frac{1}{2}$, $Pr(x = -1) = \frac{1}{4}$, $Pr(x = 1) = \frac{1}{4}$
2. $\psi_{\frac{1}{2}} : Pr(x = 0) = \frac{3}{4}$, $Pr(x = -1) = \frac{1}{8}$, $Pr(x = 1) = \frac{1}{8}$

Given a set A, $U(A)$ is the uniform distribution over A. We denote by H a hash function and $\texttt{Samp}(D, seed)$ an algorithm which samples a random variable according to a distribution D with a given seed.

Error correction code. We denote by $[n', k, d]$ a set of parameters of an error-correction code (in our case a binary BCH code). n' denotes the length of the codewords, k is the dimension and d is the minimal Hamming distance of the code.

LAC

LAC is a Ring-LWE based public key encryption scheme over $R_{q,1}$. In order to balance performance and size, LAC team chose $q = 251$, that fits on one byte. This choice of a small modulus implies a lower security or a higher decryption error rate. To overcome these issues, an error-correction code is used, allowing to

keep a low decryption error rate and maintain the same security level than schemes using larger modulus. Three security levels are proposed for LAC: LAC-128, LAC-192 and LAC-256. In this section, we describe the four algorithms `CPA.KeyGen`, `CPA.Encrypt`, `CPA.Decrypt`, `CPA.Decrypt256` of the CPA version of LAC, the four subroutines `BCHEncode`, `BCHDecode`, `Compress` and `Decompress` and the CPA-KEM scheme.

Note that `CPA.KeyGen` and `CPA.Encrypt` are common to the three security levels. However, the decryption depends on the security level: Algorithm 36 is the decryption process for LAC-128 and LAC-192. The decryption routine for LAC-256 is described in Algorithm 37.

Algorithm 34
`CPA.KeyGen`

Output: Key pair (p_k, s_k)
1: $seed_a \leftarrow U(\mathcal{S})$
2: $a \leftarrow \texttt{Samp}(U(R_{q,1}), seed_a)) \in R_{q,1}$
3: $s \leftarrow \psi_\sigma^N$
4: $e \leftarrow \psi_\sigma^N$
5: $b \leftarrow a \times s + e \in R_{q,1}$
6: **return** $(p_k, s_k) = ((seed_a, b), s)$

Algorithm 35
`CPA.Encrypt`$(p_k, m, seed)$

Output: Ciphertext $c = (c_1, c_2)$
1: $(seed_a, b) \leftarrow p_k$
2: $a \leftarrow \texttt{Samp}(U(R_{q,1}), seed_a) \in R_{q,1}$
3: $\widehat{m} \leftarrow \texttt{BCHEncode}(m) \in \{0,1\}^{l_v}$
4: $r \leftarrow \texttt{Samp}(\psi_\sigma^N, seed)$
5: $e_1 \leftarrow \texttt{Samp}(\psi_\sigma^N, seed)$
6: $e_2 \leftarrow \texttt{Samp}(\psi_\sigma^{l_v}, seed)$
7: $c_1 \leftarrow ar + e_1 \in R_{q,1}$
8: $c_2 \leftarrow (br)_{l_v} + e_2 + \lfloor \frac{q}{2} \rceil \widehat{m} \in \mathbb{Z}_q^{l_v}$
9: **if** LAC-256
10: $c_2 \leftarrow c_2 || c_2$ //D2 encoding
11: **end if**
12: $c_2 \leftarrow \texttt{Compress}(c_2)$
13: **return** $c = (c_1, c_2)$

Algorithm 36
`CPA.Decrypt`$(s_k, c = (c_1, c_2))$

Output: Plaintext m
1: $c_2 \leftarrow \texttt{Decompress}(c_2)$
2: $\widehat{M} \leftarrow c_2 - (c_1 s_k)_{l_v} \in \mathbb{Z}_q^{l_v}$
3: **for** $i = 0$ to $l_v - 1$ **do**
4: **if** $\frac{q}{4} \leq \widehat{M}_i < \frac{3q}{4}$ **then**
5: $\widehat{m}_i \leftarrow 1$
6: **else**
7: $\widehat{m}_i \leftarrow 0$
8: **end if**
9: **end for**
10: $m \leftarrow \texttt{BCHDecode}(\widehat{m})$
11: **return** m

Algorithm 37
`CPA.Decrypt256`$(s_k, c = (c_1, c_2))$

Output: Plaintext m
1: $c_2 \leftarrow \texttt{Decompress}(c_2)$
2: $\widehat{M} \leftarrow c_2 - (c_1 s_k)_{2l_v} \in \mathbb{Z}_q^{2l_v}$
3: **for** $i = 0$ to $l_v - 1$ **do** //D2 Decoding
4: $tmp_1, tmp_2 := \widehat{M}[i], \widehat{M}[i + l_v]$
5: **if** $tmp_1 < \frac{q}{2}$
6: $tmp_1 \leftarrow q - tmp_1$
7: **else if** $tmp_2 < \frac{q}{2}$
8: $tmp_2 \leftarrow q - tmp_2$
9: **end if**
10: **if** $tmp_1 + tmp_2 - q < \frac{q}{2}$
11: $\widehat{m}_i \leftarrow 1$
12: **else**
13: $\widehat{m}_i \leftarrow 0$
14: **end if**
15: **end for**
16: $m \leftarrow \textbf{\textit{BCHDecode}}(\widehat{m})$
17: **return** m

Subroutines

`BCHEncode` and `BCHDecode`. The function `BCHEncode` takes as input a message m of length l_m, pads it with $(k - l_m)$ zeros, where k is the dimension of the BCH code, and returns the corresponding value c on

the code. The function **BCHDecode** takes as input a message $\widehat{c}$ of length $N-1$, retrieves the codeword c closest to $\widehat{c}$ and returns m such that $c = mG$, where G is the generator matrix of the code.

Compress and Decompress. The function **Compress** takes as input a variable $c = (c_0, \ldots, c_{len_c})$ where each coefficient c_i is a 8-bits number and returns $c' = (c'_0, \ldots, c'_{len_c})$ where each c'_i is a 4 bits number obtained by keeping the highest 4 bits of c_i.

The function **Decompress** takes as input a variable $c' = (c'_0, \ldots, c'_{len_c})$ where each coefficient c'_i is a 4-bit number, and returns $\tilde{c} = (\tilde{c}_0, \ldots, \tilde{c}_{len_c})$ where each $\tilde{c}_i$ is a 8 bits number obtained by padding each coefficient c'_i with 4 zero bits.

D2 encoding. $D2$ encoding duplicates the coordinate of a vector: $\texttt{D2Enc}(c_2) = (c_2||c_2)$. The use of this encoding allows to decrease decoding errors.

Parameters Recall that LAC is a RLWE public-key encryption scheme on $R_{q,1} = \mathbb{Z}_q[X]/(X^N+1)$, with input messages of length l_m.
LAC uses different parameters for its three algorithms:

Name	N	q	Distrib	l_m	l_v	Code(BCH) $[n', k, d]$	D2
LAC-128	512	251	ψ_1	256	$l_m + 144$	$[511, 367, 33]$	No
LAC-192	1024	251	$\psi_{\frac{1}{2}}$	256	$l_m + 72$	$[511, 439, 17]$	No
LAC-256	1024	251	ψ_1	256	$l_m + 144$	$[511, 367, 33]$	Yes

The value l_v depends on the BCH code. Let G be a generator matrix of the BCH code C. By the construction of LAC, G is on systematic form $G = (Id_k|A_{n'-k})$. In fact, we cannot keep only l_v bits of a codeword without this condition. The **BCHEncode** function takes as input a message m of length l_m and pads it with $(k - l_m)$ zeros. We obtain

$$(m_1, \ldots, m_{l_m}, 0_1, \ldots, 0_{k-l_m})G = (m_1, \ldots, m_{l_m}, 0_1, \ldots, 0_{k-l_m}|mA_{n'-k}) = c$$

We omit the $(k - l_m)$ zeros of c then $l_v = l_m + (n' - k)$.

LAC Key Exchange We describe the LAC Key Exchange introduced in [XYD+19], based on the CPA version of the LAC public-key encryption scheme. In the following we denote the secret key sk by s.

Alice		Bob
$(pk, s) \leftarrow$ **CPA.KeyGen**		
	$\xrightarrow{pk}$	
		$r \leftarrow U(\{0,1\}^{l_m})$
		$c \leftarrow$ **CPA.Encrypt**(pk, r)
		$Key_B \leftarrow H(pk, r) \in \{0,1\}^{l_k}$
	$\xleftarrow{c}$	
$r' \leftarrow$ **CPA.Decrypt**(s, c)		
$Key_A \leftarrow H(pk, r') \in \{0,1\}^{l_k}$		

If Key Exchange succeeds then $r' = r$ and $Key_B = Key_A$.

7.1.2 Attack on LAC key exchange

We start by defining the attack scenario by introducing the oracle defined in [BGR+19].

Attack Model

Suppose that Alice does a misuse of the Key Exchange Mechanism by caching her secret s. More precisely:

Assumption 1. *Alice keeps her secret key constant for several CPA key establishments requests.*

Eve is a malicious active adversary who acts as Bob and can cheat and generate c that is not the encryption of a random r. To mount the active attack, we suppose that Eve has access to a session key mismatch oracle defined as follow.

Definition 3. *A* key mismatch oracle *outputs a bit of information on the possible mismatch at the end of the key encapsulation mechanism. In the LAC context, this oracle, denoted $\mathcal{O}$, takes any message c and any session key guess μ as input and outputs:*

$$\mathcal{O}(c,\mu) = \begin{cases} 1 & \text{if } H(pk, \texttt{CPA.Decrypt}(s,c)) = \mu \\ -1 & \text{otherwise} \end{cases}$$

This oracle can also be used by Bob during an honest key exchange with Alice, when he verifies the match between his session key and Alice's one.

The idea of the attack mounted by Eve is to send forged ciphertexts to Alice to ensure that she obtains information on some coefficients of Alice's secret key. As Eve knows that $c = (c_1, c_2)$ and s are used during the decryption algorithms (s is multiplied by c_1), she will mount an attack using this fact and following four mains steps:

- Choose a session key μ.
- Construct c_1 such that some coefficients of the secret key are exposed.
- Construct c_2 depending of μ such that the result of Alice's decryption can be monitored as a function of the key guess.
- Call to the oracle $\mathcal{O}$ to obtain information about our key guesses.

The following section shows how to choose appropriate (c, μ) to retrieve information on s. We assume that Eve has access to the oracle $\mathcal{O}$.

Attack on LAC-128-KE and LAC-192-KE

First, we use a simplified version where we do not consider `Compress` and `Decompress` functions. We follow the different steps of the decryption algorithm 36.

Simplified version In this first result, we show how one can forge a LAC ciphertext in order to impose which plaintext will be obtained after decryption. To do so, we need to forge c such that the impact of the secret key during the decryption is under our control.

Proposition 12. *Assume that Eve forges $c = (c_1, c_2)$ such that :*

- $c_1 = -aX^{N-w}$ *where w is an integer $0 \leq w < N$ and $0 \leq a < \frac{q}{4}$*
- $c_2 = (\alpha_0, \ldots, \alpha_{l_v-1})$ *where $\alpha_i = \frac{q}{2}$ or 0 for all i in $[0, l_v - 1]$.*

Then she can determine the plaintext m that Alice will obtain after decryption.

Proof. When Alice deciphers Eve's ciphertext she:

1. Computes $\widehat{M} = c_2 - (c_1 s)_{l_v}$

2. Compares each coefficient of $\widehat{M}$ to $\frac{q}{4}$ and $\frac{3q}{4}$ to define $\widehat{m}$
3. Retrieves m using **BCHDecode** algorithm on $\widehat{m}$

Let $c_1 = -aX^{N-w}$ and $s = s_0 + s_1X^1 + \ldots + s_{N-1}X^{N-1}$ then

$$c_1 s = as_w + as_{w+1}X + \ldots + as_{N-1}X^{N-w-1} - as_0X^{N-w} - \ldots - as_{w-1}X^{N-1}$$

and the polynomial $c_1 s$ can be represented as the vector $(as_w, \ldots, -as_{w-1})$.
During the computation of $\widehat{M}$, two cases are possible:

- $w < l_v$ then $\widehat{M} = c_2 - (c_1 s)_{l_v} = (\alpha_0 - as_w, \ldots, \alpha_w + as_0, \ldots, \alpha_{l_v-1} + as_{w+l_v-1})$
- $w \geq l_v$ then $\widehat{M} = c_2 - (c_1 s)_{l_v} = (\alpha_0 - as_w, \alpha_1 - as_{w+1}, \ldots, \alpha_{l_v-1} - as_{w+l_v-1})$

After this computation each coefficient of $\widehat{M}$ is compared to $\frac{q}{4} \leq \widehat{M}_i < \frac{3q}{4}$.
Recall that since $s \leftarrow \psi_\sigma^N$, each of its coefficients belongs to $\{-1, 0, 1\}$. Let i be an integer such that $0 \leq i < N$ and $j \equiv N - w + i \mod N$.

If $\alpha_i = \frac{q}{2}$ one gets:

$$\alpha_i \mp as_j = \begin{cases} \frac{q \mp 2a}{2} & \text{if } s_j = \pm 1 \\ \frac{q}{2} & \text{if } s_j = 0 \\ \frac{q \pm 2a}{2} & \text{if } s_j = \mp 1 \end{cases}$$

If $\alpha_i = 0$ one gets:

$$\alpha_i \mp as_j = \begin{cases} \pm a & \text{if } s_j = \mp 1 \\ 0 & \text{if } s_j = 0 \\ \mp a & \text{if } s_j = \pm 1 \end{cases}$$

In the first case, the three possible values for $\alpha_i \mp as_j$ lie in $\left[\frac{q}{4}, \frac{3q}{4}\right[$ if $0 \leq a \leq \frac{q}{4}$.
In the case $\alpha_i = 0$, the three possible values do not lie in $\left[\frac{q}{4}, \frac{3q}{4}\right[$ when $0 \leq a < \frac{q}{4}$ or $\frac{3q}{4} \leq a \leq q$.
Thus, Eve can choose $a < \frac{q}{4}$ and $\alpha_i = \frac{q}{2}$ or 0 to determine what Alice will obtain on the first l_v coordinates of $\widehat{m}$. Then, Eve can deduce, by applying BCH decoding, what Alice obtains at the end of the decryption procedure. □

The next example explains how one can use Proposition 12.

Example 15. *Suppose that Eve wants that Alice will obtain, after decryption, the message* $m =$***BCHDecode***$(1, 0, 1, 1, 0, \ldots, 0)$. *Then she forges* $c = (c_1, c_2)$ *such that:*

- $c_1 = -\frac{q}{5}X^N = \frac{q}{5}$ *on* $R_{q,1}$. *In fact Eve can take any* c_1 *such that* $c_1 = -aX^{N-w}$ *with* $0 \leq a < \frac{q}{4}$
- $c_2 = (\frac{q}{2}, 0, \frac{q}{2}, \frac{q}{2}, 0, \ldots, 0)$

From c *Alice first computes:*

$$\begin{aligned} \widehat{M} &= c_2 - (c_1 s)_{l_v} \\ &= \left(\frac{q}{2}, 0, \frac{q}{2}, \frac{q}{2}, 0, \ldots, 0\right) - \frac{q}{5}\left(s_0, s_1, \ldots, s_{l_v}\right),\ s_i \text{ belongs to } \{-1, 0, 1\} \\ &= \left(\frac{q}{2} - \frac{q}{5}s_0, -\frac{q}{5}s_1, \frac{q}{2} - \frac{q}{5}s_2, \frac{q}{2} - \frac{q}{5}s_3, -\frac{q}{5}s_4, \ldots, -\frac{q}{5}s_{l_v}\right) \end{aligned}$$

Then, Alice compares each coefficients of $\widehat{M}$ *to* $\frac{q}{4}$ *and* $\frac{3q}{4}$. *She obtains (see proof of Proposition 12):*

$$\widehat{m} = (1, 0, 1, 1, 0, \ldots, 0)$$

At the end, Alice obtains m *by applying* ***BCHDecode*** *algorithm to* $\widehat{m}$. *Thus, Eve had forged* c *such that Alice has* $m =$ ***BCHDecode***$(1, 0, 1, 1, 0, \ldots, 0)$.

With Proposition 12 we construct a ciphertext such that the secret key has no impact during decryption. Now Eve needs to construct forged ciphertexts that allow a key guessing strategy in order to retrieve the secret key. Thus, we need that the secret key has an impact during decryption if and only if we did a good key guess.

Proposition 13. *Let s'_w be a guess done by Eve on the w-th coefficient of the secret key s, where $0 \leq w < N$. Assume $s_w = 1$ or -1. If Eve forges $c = (c_1, c_2)$ as given in Proposition 12 and modifies the first coordinate of c_2 such that:*

- $c_2 = (as'_w, \alpha_1, \ldots, \alpha_{l_v-1})$ *with* $\frac{q}{8} < a < \frac{q}{4}$.

Then she can verify her key guess from the plaintext computed by Alice from c.

Proof. Suppose that Eve wants to retrieve the w-th coefficient of s. When Alice will decipher Eve ciphertext she first computes:

$$\widehat{M} = c_2 - (c_1 s)_{l_v} = (as'_w - as_w, \alpha_1 - as_{w+1}, \ldots)$$

According to Proposition 12, Eve can determine what Alice will obtain for every coefficient different from her guess s'_w. Let see what happens with this coefficient by analysing $as'_w - as_w$.

$$as'_w - as_w = \begin{cases} 0 & \text{if } s'_w = s_w \\ 2a & \text{if } s'_w = 1 \text{ and } s_w = -1 \\ -2a & \text{if } s'_w = -1 \text{ and } s_w = 1 \\ \mp a & \text{if } s'_w = 0 \text{ or } s_w = 0 \end{cases}$$

Let $\frac{q}{8} < a < \frac{q}{4}$ then $\frac{q}{4} < 2a < \frac{q}{2}$ and $-2a \equiv q - 2a \mod q$ satisfies $\frac{q}{2} < q - 2a < \frac{3q}{4}$.

The key guess is good (resp. wrong) when a 1 (resp. 0) is returned at the first coordinate of $\widehat{m}$. Hence Eve can effectively determines what Alice obtained by applying `BCHDecode` algorithm to $\widehat{m}$ and thus deterministically verifies her key guess from $\widehat{m}$. □

Proposition 13 ensures that if Eve guessed the good key then Alice will obtains $m =$ `BCHDecode`$(1, \ldots)$. Otherwise, she will obtain $m =$ `BCHDecode`$(0, \ldots)$. Computational details are given in the following example.

Example 16. *Suppose that Eve wants to learn information about the first bit of Alice's secret key. Eve forges $c = (c_1, c_2)$ such that:*

- $c_1 = -\frac{q}{5}X^N = \frac{q}{5}$ *on* $R_{q,1}$.
- $c_2 = (\frac{q}{5}s'_0, 0, \frac{q}{2}, \frac{q}{2}, 0, \ldots, 0)$ *where s'_w is Eve's key guess.*

As in Example 15, Alice first computes $\widehat{M} = c_2 - (c_1 s)_{l_v} = (\frac{q}{5}s'_0 - \frac{q}{5}s_0, -\frac{q}{5}s_1, \frac{q}{2} - \frac{q}{5}s_2, \frac{q}{2} - \frac{q}{5}s_3, -\frac{q}{5}s_4, \ldots, -\frac{q}{5}s_{l_v})$ *where s_i belongs to $\{-1, 0, 1\}$. Then, Alice compares each coefficients of $\widehat{M}$ to $\frac{q}{4}$ and $\frac{3q}{4}$. She obtains (see proof of Proposition 13):*

$$\widehat{m} = (1, 0, 1, 1, 0, \ldots, 0) \text{ if } s'_0 = -s_0 \text{ and } s_0 \neq 0$$
$$\widehat{m} = (0, 0, 1, 1, 0, \ldots, 0) \text{ otherwise}$$

At the end, Alice obtains m by applying `BCHDecode` *algorithm to $\widehat{m}$. Thus, Eve did the good key guess if Alice gets $m =$* `BCHDecode`$(1, 0, 1, 1, 0, \ldots, 0)$.

Proposition 13 already gives interesting information to Eve but it is not enough to mount an attack since Eve needs a way to verify if Alice obtains:

- either $m = \texttt{BCHDecode}(1, 0, 1, 1, 0, \ldots, 0)$
- or $m = \texttt{BCHDecode}(0, 0, 1, 1, 0, \ldots, 0)$

without knowing m. Moreover, most of the time $\texttt{BCHDecode}(1, 0, 1, 1, 0, \ldots, 0)$ will not differ from $\texttt{BCHDecode}(0, 0, 1, 1, 0, \ldots, 0)$.

To overcome these issues we need to instantiate precisely the oracle given in Definition 3 using Proposition 13.

Instantiation of the Oracle. The oracle defined in Definition 3 gives information about the success of a key session establishment between Alice and Bob. Eve can use such an oracle with the help of Proposition 13 and the BCH code decryption failure to overcome issues mentioned above.

In the sequel, we show how Eve can practically mount an attack by forging specific inputs to this oracle and deduce information on Alice's secret key. The following theorem and its proof detail this construction. The Algorithm 38 and Algorithm 39 formally describe the attack.

Theorem 1. *Let $s'_w \in \{-1, 1\}$ be the guessed value of s_w $(0 \leq w < N)$ done by Eve. If Eve takes a session key $\mu_{s'_w}$ then she can forge $c_{s'_w} = (c_1, c_2)$ depending of $\mu_{s'_w}$ by using properties given in Proposition 13 such that by calling $\mathcal{O}(c_{s'_w}, \mu_{s'_w})$ with $s'_w \in \{-1, 1\}$, she retrieves the w-th coefficient of s. In consequence, Eve needs at most 2 calls to the oracle in order to retrieve a coefficient of Alice's secret key.*

Proof. According to Proposition 13, Eve can monitor Alice's decryption procedure if she does the good key guess.

An error-correction code can correct at most $\frac{d-1}{2}$ errors (where d is the minimal Hamming distance of the BCH code). The idea is that after comparison with $\frac{q}{4}$ and $\frac{3q}{4}$, $\widehat{m}$ is a codeword with $\frac{d}{2}$ errors if Eve did the wrong key guess, causing a decoding error. Suppose Eve wants to retrieve the w-th coefficient of s:

1. Eve chooses a codeword called *cdword* with a 1 at the first coordinate such that $cdword = mG$ where G is the generator matrix of the BCH code
2. Eve injects $\frac{d-1}{2}$ errors to *cdword* at any coordinate except the first one
3. Eve chooses a verifying $\frac{q}{8} < a < \frac{q}{4}$ according to Proposition 13
4. Eve constructs c_1, c_2 with her key guess at the first bit of c_2: $c_2[0] = as'_w$ and such that after comparison with $\frac{q}{4}$ and $\frac{3q}{4}$, Alice retrieves *cdword* with $\frac{d-1}{2}$ errors or *cdword* with $\frac{d}{2}$ errors
5. Eve sends $c = (c_1, c_2)$ to Alice

With this construction, Alice obtains a codeword with $\frac{d}{2}$ errors if Eve provides a wrong key guess. At this point, Eve's session key is $sess_E = H(pk, m)$ and Alice's session key $sess_A$ depends on Eve's key guess. Eve can verify whether she did the correct key guess with the oracle as follow:

If $s'_w = 1$ and $\mathcal{O}(c, sess_E) = 1$ then $s_w = -1$ and $sess_A = sess_E$
Else If $s'_w = -1$ and $\mathcal{O}(c, sess_E) = 1$ then $s_w = 1$ and $sess_A = sess_E$
Otherwise $s_w = 0$

□

Algorithm 38 and Algorithm 39 are based on the construction described in the proof of Theorem 1. Here, we fix the constant a to $\frac{q}{7}$ in the construction of c_1.

Algorithm 38 forge(hyp,bit)

Output: Forge ciphertext $c = (c_1, c_2)$

$c_1 := -\frac{q}{7} X^{N-bit}$
$m := [0 : \textbf{for } i := 0 \text{ to } 255]$
$m[0] := 1$
$codeword := (m||0..0)G$
Add $\frac{d-1}{2}$ errors to codeword (but not on codeword[0])
For $i = 0$ **to** Len($codeword$) :
  if $i == 0$:
    $c_2[0] \leftarrow \text{hyp} \times \frac{q}{7}$
  else if $codeword[i] == 1$:
    $c_2[i] \leftarrow \frac{q}{2}$
  else
    $c_2[i] \leftarrow 0$
  end if
end for
Return$(m, c = (c_1, c_2))$

Algorithm 39 recoverOneBit(bit)

Output: A bit of s

$m, c :=$ **forge**$(-1, bit)$
If $\mathcal{O}(c, m) == 1$:
  Return 1
end if
$m, c :=$ **forge**$(1, bit)$
If $\mathcal{O}(c, m) == 1$:
  Return -1
end if
Return 0

Using Theorem 1, a key of length N can be fully recovered with at most $2 \times N$ requests to the oracle. LAC-128 works with keys of length $N = 512$ and LAC-192 with length $N = 1024$.

Full version The subroutine `Compress` removes the 4 lowers bits of each coeff of c_2. They are replaced by 4 zero-bit when the subroutine `Decompress` is applied at the beginning of the decryption process. Thus, each coefficient of c_2 can be only equal to 16, 32, 64, 128 and any sum of theses values.
For c_2 in our attack, we only consider the values $\frac{q}{7}$, $-\frac{q}{7}$ and $\frac{q}{2}$. In our implementation [Mona] we approximate $\frac{q}{7} \approx 32$, $-\frac{q}{7} \approx 128 + 64 + 16 = 210$ and $\frac{q}{2} \approx 128$. Proposition 13 is still verified and we still retrieve s with at most $2 \times N$ requests to the oracle by the Theorem 1.

In comparison of the recent work of Huguenin-Dumittan *et al.* in [HV20], our upper-bound for LAC128 is 2 times less than theirs. Indeed, they need at most 2^{11} queries to retrieve the entire secret key, while we need at most 2^{10} queries.

Implementation results We have developed a `C` implementation of the attack (see [Mona]). To assess its efficiency we use the reference code of LAC [XYD+19] as a target. In the following, we present practical results on the average of 1000 attacks launched on 1000 random secret keys for LAC-128 and LAC-192. Timing results have been evaluated on core i5-8350U at 1.90GHz.

	Nb of coeff of sk	Average oracle requests	Average time
LAC-128	512	896	$2,94$ ms
LAC-192	1024	1920	$15,53$ ms

The size of LAC-192 secret key is 2 times larger than LAC-128 one, but the number of required request to retrieve sk is more than 2 times larger. This is due to a different probability distribution between these two levels of security.
In average we need $1,75 \times 512$ oracle requests for LAC-128 and $1,875 \times 1024$ requests for LAC-192. For both cases, the practical result is less than the upper bound of $2 \times N$ where $N = 512$ or 1024.

7.1.3 Attack on LAC-256 key exchange

Attack on LAC-256 key exchange

Since LAC-256 encryption uses $D2$ encoding , the decryption procedure is slightly different. Let $c = (c_1, c_2)$, $D2$ encoding duplicates the coordinate of c_2: $c_2 = (c_2||c_2)$. The use of this encoding allows to decrease decoding errors.

In the attack on LAC-128/192 we forged c_1 as a monomial to avoid linear combination between coefficients of s during computation of $c_1 s$. This allows to do key guess on only one coefficient of s. But, despite the use of a monomial for c_1, $D2$ encoding ensures that each coefficient of $c_1 s$ is a linear combination of at least 2 coefficients of s. It implies that we need to do key guesses on two coefficients of s. In this section, we adapt our previous attack to allow to do two key guesses rather than one. The attack procedure is the same as previously:

- Choose a session key μ.
- Construct c_1 such that some coefficients of the secret key are exposed.
- Construct c_2 depending of μ such that the result of Alice's decryption can be monitored as a function of the key guess.
- Call to the oracle $\mathcal{O}$ to obtain information about our key guesses.

`CPA.Decrypt256` description The first step of the decryption it's to compute $\widehat{M} = c_2 - (c_1 s)_{2l_v}$ as previously. However the comparison is different for LAC-256. The decryption algorithm considers two cases:

Case 1. If $\widehat{M}[i]$ and $\widehat{M}[i + l_v] < \frac{q}{2}$ or $\widehat{M}[i]$ and $\widehat{M}[i + l_v] \geq \frac{q}{2}$ then algorithm `CPA.Decrypt256` checks whether: $\frac{\widehat{M}[i]+\widehat{M}[i+l_v]}{2} \in]\frac{q}{4}, \frac{3q}{4}[$

Case 2. If $\widehat{M}[i] < \frac{q}{2}$ and $\widehat{M}[i + l_v] \geq \frac{q}{2}$ or $\widehat{M}[i] \geq \frac{q}{2}$ and $\widehat{M}[i + l_v] < \frac{q}{2}$ then `CPA.Decrypt256` checks whether $\frac{|\widehat{M}[i]-\widehat{M}[i+l_v]|}{2} \in]0, \frac{q}{4}[$

In the following we notice when we are in the case 1 or 2.

Attack on LAC-256 key exchange simplified

As previously we first use a simplified version where we do not consider `Compress` and `Decompress` subroutines.

Proposition 14. *Assume that Eve forges $c = (c_1, c_2)$ such that:*

- $c_1 = -aX^{N-w}$ *where w is an integer $0 \leq w < (N - l_v)$ and $0 \leq a < \frac{q}{4}$*
- $c_2 = (\alpha_0, \ldots, \alpha_{l_v-1}, \alpha_{l_v}, \ldots, \alpha_{2l_v-1})$ *where $\alpha_i = \frac{q}{2}$ or 0 for all i in $[0, 2l_v - 1]$*

Then she can determine the plaintext m that Alice obtains after decryption.

Proof. Assuming Alice receives $c = (c_1, c_2)$ then she:

1. Computes $\widehat{M} = c_2 - (c_1 s)_{2l_v}$
2. Compares $\frac{q}{4} < \frac{\widehat{M}[i]+\widehat{M}[i+l_v]}{2} < \frac{3q}{4}$ or $0 < \frac{|\widehat{M}[i]-\widehat{M}[i+l_v]|}{2} < \frac{q}{4}$ for $i = 0$ to $l_v - 1$ to define each coefficient of $\widehat{m}$

3. Retrieves m using `BCHDecode` algorithm on $\widehat{m}$

Let $c_1 = -aX^{N-w}$ and $s = s_0 + s_1X^1 + \ldots + s_{N-1}X^{N-1}$ then

$$c_1 s = as_w + as_{w+1}X + \ldots + as_{N-1}X^w - as_0X^{w-1} - \ldots - as_{w-1}X^{N-1}$$

$c_1 s$ can be represented as a vector: $(as_w, \ldots, -as_{w-1})$. During the computation of $\widehat{M}$ two cases are possible:

- $w < 2l_v$ then $\widehat{M} = c_2 - (c_1 s)_{2l_v} = (\alpha_0 - as_w, \alpha_1 - as_{w+1}, \ldots, \alpha_w + as_0, \ldots, \alpha_{2l_v-1} + as_{(2l_v-1+w \mod N)})$
- $w \geq 2l_v$ then $\widehat{M} = c_2 - (c_1 s)_{2l_v} = (\alpha_0 - as_w, \alpha_1 - as_{w+1}, \ldots, \alpha_{2l_v-1} - as_{(2l_v-1+w \mod N)})$

Recall that since $s \leftarrow \psi_\sigma^N$, each of its coefficients belongs to $\{-1, 0, 1\}$. Let i be an integer such that $0 \leq i < l_v$ and $j \equiv i + w \mod N$. For decryption there are the three following cases. (We cannot have the case where $\widehat{M}[i] = \alpha_i + as_j$ and $\widehat{M}[i + l_v] = \alpha_{i+l_v} - as_{j+l_v}$ because that implies $j + l_v \leq w + l_v$ and $w < j$ with $l_v > 0$ and $j \geq 0$.)

1. $\widehat{M}[i] = \alpha_i - as_j$ **and** $\widehat{M}[i + l_v] = \alpha_{i+l_v} - as_{j+l_v}$
If $\alpha_i = \frac{q}{2}$ one gets:

- If $s_j = s_{j+l_v}$ or $s_j + s_{j+l_v} = -1$ we are in the Case 1 described in 7.1.3, where $\alpha_i - as_j = \widehat{M}[i]$. Then

$$\alpha_i = \alpha_{i+l_v} = \frac{q}{2}, \frac{(\alpha_i - as_j) + (\alpha_{i+l_v} - as_{j+l_v})}{2} = \begin{cases} \frac{q-2a}{2} & \text{if } s_j = s_{j+l_v} = 1 \\ \frac{q+2a}{2} & \text{if } s_j = s_{j+l_v} = -1 \\ \frac{q}{2} & \text{if } s_j = s_{j+l_v} = 0 \\ \frac{q+a}{2} & \text{if } s_j + s_{j+l_v} = -1 \end{cases}$$

  These 3 values lie in $\left]\frac{q}{4}, \frac{3q}{4}\right[$ if $0 \leq a < \frac{q}{4}$.

- Otherwise we are in the Case 2 described in Paragraph 7.1.3, where $\alpha_i - as_j = \widehat{M}[i]$:

$$\alpha_i = \alpha_{i+l_v} = \frac{q}{2}, \frac{|(\alpha_i - as_j) - (\alpha_{i+l_v} - as_{j+l_v})|}{2} = \begin{cases} \frac{a}{2} & \text{if } s_j + s_{j+l_v} = 1 \\ a & \text{if } s_j = -1, s_{j+l_v} = 1 \\ & \text{or } s_j = 1, s_{j+l_v} = -1 \end{cases}$$

  These values lie in $[0, \frac{q}{4}[$ if $0 \leq a < \frac{q}{4}$.

Then for both cases, if $c_1 = -aX^{N-w}$ with $\alpha_i, \alpha_{i+l_v} = \frac{q}{2}$, we can ensure that we have a 1 after comparison. If $\alpha_i = 0$ then we are in the Case 1 described in Paragraph 7.1.3, where $\alpha_i - as_j = \widehat{M}[i]$:

$$\frac{(\alpha_i - as_j) + (\alpha_{i+l_v} - as_{j+l_v})}{2} = \begin{cases} a & \text{if } s_j = s_{j+l_v} = -1 \\ 0 & \text{if } s_j = -s_{j+l_v} \text{ or } s_j = s_{j+l_v} = 0 \\ -a & \text{if } s_j = s_{j+l_v} = 1 \\ \pm\frac{a}{2} & \text{otherwise} \end{cases}$$

Then these 3 values do not lie in $]\frac{q}{4}, \frac{3q}{4}[$ for $0 \leq a < \frac{q}{4}$.

2. $\widehat{M}[i] = \alpha_i + as_j$ **and** $\widehat{M}[i + l_v] = \alpha_{i+l_v} + as_{j+l_v}$**.** The proof is the same as above. We give here the different decryption cases:

- If $\alpha_i = \frac{q}{2}$ then two cases are possible: if $s_j = s_{j+l_v}$ or $s_j + s_{j+l_v} = 1$ then we are in the decryption Case 1 otherwise in the Case 2.
- If $\alpha_i = 0$ then we are in the decryption Case 1.

3. $\widehat{M}[i] = \alpha_i - as_j$ **and** $\widehat{M}[i+l_v] = \alpha_{i+l_v} + as_{j+l_v}$**.** The proof is the same as above. We give here the different decryption cases:

- If $\alpha_i = \frac{q}{2}$ then two cases are possible: if $s_j = -s_{j+l_v}$ or $s_j = 0, s_{j+l_v} = 1$ or $s_j = -1, s_{j+l_v} = 0$ then we are in the decryption Case 1, otherwise in the Case 2.
- If $\alpha_i = 0$ then we are in the decryption Case 1.

□

Example 17. *Suppose that Eve wants that Alice obtains, after decryption, the plaintext*

$$m = \texttt{BCHDecode}(1, 1, 0, 1, 0, \ldots, 0)$$

. Eve forges $c = (c_1, c_2)$ such that:

- $c_1 = -\frac{q}{5}X^N = \frac{q}{5}$ *on* $R_{q,1}$.
- $c_2 = (\frac{q}{2}, \frac{q}{2}, 0, \frac{q}{2}, 0 \ldots, 0 || \frac{q}{2}, \frac{q}{2}, 0, \frac{q}{2}, 0 \ldots, 0)$. *The symbol || delimits the l_v first part to the l_v second part of c_2 (we duplicate c_2 due to D2 encoding in Algorithm 35). The two parts are symmetric.*

When Alice deciphers c, she computes $\widehat{M} = c_2 - (c_1 s)_{2l_v}$ and uses the comparison procedure describes in Algorithm 37 to obtain $\widehat{m}$ of length l_v. If c_1 and c_2 are constructed according to Proposition 14, then (cf Proof 7.1.3):

- *If $c_2[i] = c_2[i + l_v] = \frac{q}{2}$ then $\widehat{m}[i] = 1$*
- *If $c_2[i] = c_2[i + l_v] = 0$ then $\widehat{m}[i] = 0$*

Then, with our $c_2 = (\frac{q}{2}, \frac{q}{2}, 0, \frac{q}{2}, 0 \ldots, 0 || \frac{q}{2}, \frac{q}{2}, 0, \frac{q}{2}, 0 \ldots, 0)$ Alice obtains $\widehat{m} = (1, 1, 0, 1, 0, \ldots, 0)$. Thus, Alice retrieves $m = \texttt{BCHDecode}(1, 1, 0, 1, 0, \ldots, 0)$.

So Eve can choose $a < \frac{q}{4}$ and $\alpha_i = \frac{q}{2}$ or 0 to know what Alice obtains on the l_v coordinates of $\widehat{m}$ and then Eve can deduce what Alice obtains at the end of decryption for m. Eve needs to construct forged ciphertexts which allow to verify her key guesses.

Proposition 15. *Let s'_w and s'_{w+l_v} be guesses done by Eve on the w-th and $w + l_v$ coefficients of the secret key s. Assume $s_w, s_{w+l_v} = 1$ or -1. If Eve forges $c = (c_1, c_2)$ as given in Proposition 14 and modifies the first and l_v-th coordinates of c_2 such that:*

- $c_2 = (as'_w, \alpha_1, \ldots, \alpha_{l_v-1}, as'_{w+l_v}, \ldots, \alpha_{l_{v-1}})$ *with* $\frac{q}{8} < a < \frac{q}{4}$.

Then she can verify her key guesses from the plaintext computed by Alice from c.

Proof. According to Proposition 14 Eve can determine what Alice obtains at the end of the decryption procedure for every coefficient different from the key guesses. Assume that Eve wants to retrieve the w-th and $(w + l_v)$-th coefficients of s. Let $\widehat{M} = c_2 - (c_1 s)_{2l_v}$, due to $0 \leq w < (N - l_v)$ the only case to consider is $\widehat{M}[0] = as'_w - as_w$ and $\widehat{M}[l_v] = as'_{w+l_v} - as_{w+l_v}$.

Let $s'_w = s'_w = 1$ and $\frac{q}{8} < a < \frac{q}{4}$, so we are in the Case 1 described in Paragraph 7.1.3. Let see what happens with $\frac{\widehat{M}_0 + \widehat{M}_{l_v}}{2} = \frac{as'_w - as_w + as'_{w+l_v} - as_{w+l_v}}{2}$:

$$\frac{a - as_w + a - as_{w+l_v}}{2} = \begin{cases} 2a & \text{if } s_w = s_{w+l_v} = -1 \\ 0 & \text{if } s_w = s_{w+l_v} = 1 \\ \frac{3a}{2} & \text{if } s_w = 0,\ s_{w+l_v} = -1 \\ & \text{or } s_w = -1,\ s_{w+l_v} = 0 \\ \frac{a}{2} & \text{otherwise} \end{cases}$$

Then only the case $\frac{a - as_w + a - as_{w+l_v}}{2} = \frac{3a}{2}$ can put a 1 to $\widehat{m}_0$ if $\frac{q}{8} < a < \frac{q}{4}$.
With the same condition on a and with the same method Eve can have :

- If $s'_w = s'_{w+l_v} = 1$ and $\widehat{m_0} = 1$ then $s_w = s_{w+l_v} = -1$
- If $s'_w = s'_{w+l_v} = -1$ and $\widehat{m_0} = 1$ then $s_w = s_{w+l_v} = 1$
- If $s'_w = 1$, $s'_{w+l_v} = -1$ and $\widehat{m_0} = 1$ then $s_w = -1$ and $s_{w+l_v} = 1$
- If $s'_w = -1$, $s'_{w+l_v} = 1$ and $\widehat{m_0} = 1$ then $s_w = 1$ and $s_{w+l_v} = -1$

□

Example 18. *Suppose that Eve wants to learn information about the first and the l_v-th bit of Alice's secret key. Eve forges $c = (c_1, c_2)$ such that:*

- $c_1 = -\frac{q}{5}X^N = \frac{q}{5}$ *on* $R_{q,1}$.
- $c_2 = (\frac{q}{5}s'_0, \frac{q}{2}, 0, \frac{q}{2}, 0 \ldots, 0 || \frac{q}{5}s'_{l_v}, \frac{q}{2}, 0, \frac{q}{2}, 0 \ldots, 0)$ *where s'_0 and s'_{l_v} are key guesses*

When Alice deciphers c she computes $\widehat{M} = c_2 - (c_1 s)_{2l_v}$ and uses the comparison procedure describes in Algorithm 37 to obtain $\widehat{m}$ of length l_v. If c_1, c_2, s'_0 and s'_{l_v} are constructed according to Proposition 15, then (cf Proof 7.1.3):

- *If $s'_0 = -s_0$ and $s'_{l_v} = -s_{l_v}$ then $\widehat{m}[0] = 1$*
- *Else $\widehat{m}[0] = 0$*
- *The value of the others coefficients of $\widehat{m}$ are determined as in the previous example*

If Eve does correct key guesses then Alice obtains $\widehat{m} = (1, 1, 0, 1, 0, \ldots, 0)$. Otherwise, Alice obtains $\widehat{m} = (0, 1, 0, 1, 0, \ldots, 0)$

Proposition 15 ensures that Eve can know what Alice obtains if Alice's secrets coefficients are different from 0. Let see what happens when one of the two coefficient is equal to 0.

Proposition 16. *Let s'_w and s'_{w+l_v} be guesses done by Eve on the w-th and $w+l_v$ coefficients of the secret key s. Assume $s_w = 0$ or $s_{w+l_v} = 0$. If Eve forges $c = (c_1, c_2)$ as given in Proposition 14 and modify the first and l_v-th coordinates of c_2 such that:*

- $c_2 = (as'_w, \alpha_1, \ldots, \alpha_{l_v-1}, as'_{w+l_v}, \ldots, \alpha_{l_{v-1}})$ *with* $\frac{q}{6} < a < \frac{q}{4}$.

Then she can verify her key guesses from the plaintext computed by Alice from c.

Proof. Assume that Eve wants to retrieve the w-th and $(w + l_v)$-th coefficients of s.
As Proof 7.1.3 the only case to consider is $\widehat{M}[0] = as'_w - as_w$ and $\widehat{M}[l_v] = as'_{w+l_v} - as_{w+l_v}$. Suppose $\frac{q}{6} < a < \frac{q}{4}$, $s'_w = 1$ and $s'_{w+l_v} = 1$. Let see what happens with $\frac{\widehat{M}_0 + \widehat{M}_{l_v}}{2} = \frac{as'_w - as_w + as'_{w+l_v} - as_{w+l_v}}{2}$ (Case 1 described in Paragraph 7.1.3):

$$\frac{a - as_w + a - as_{w+l_v}}{2} = \begin{cases} \frac{3a}{2} & \text{if } s_w = -1 \text{ and } s_{w+l_v} = 0 \\ & \text{or } s_w = 0 \text{ and } s_{w+l_v} = -1 \\ \frac{a}{2} & \text{if } s_w = 0 \text{ and } s_{w+l_v} = 1 \\ & \text{or } s_w = 1 \text{ and } s_{w+l_v} = 0 \\ a & \text{if } s_w = s_{w+l_v} = 0 \end{cases}$$

With $\frac{q}{6} < a < \frac{q}{4}$ then only the case where the result is $\frac{3a}{2}$ can put a 1 to $\widehat{m}_w$. However Eve needs to determine if $s_w = -1$ or $s_{w+l_v} = -1$.

Suppose $a < \frac{q}{4}$, $s'_w = -1$ and $s'_{w+l_v} = 1$, $s_w = -1$ and $s_{w+l_v} = 0$ or $s_w = 0$ and $s_{w+l_v} = -1$. Here, we need to consider the both decryption cases described in Paragraph 7.1.3. Let see what happens:

- If $s_w = -1$ and $s_{w+l_v} = 0$ we are in Case 1 thus $\frac{q}{4} < a < \frac{3q}{4}$.
- If $s_w = 0$ and $s_{w+l_v} = -1$ we are in Case 2 thus $0 < \frac{|-a-2a|}{2} < \frac{q}{4}$ which implies $0 < a < \frac{3q}{8}$.

However $a < \frac{q}{4}$, then only one case can put a 1 to $\widehat{m}_0$.

With the same condition on a and with the same method, Eve can retrieve the other values:

- If $s'_w = 1$, $s'_{w+l_v} = 1$ and $\widehat{m_0} = 1$ then $s_w = -1$, $s_{w+l_v} = 0$ or $s_w = 0$, $s_{w+l_v} = -1$
  - If $s'_w = -1$, $s'_{w+l_v} = 1$ and $\widehat{m_0} = 1$ then $s_w = 0$, $s_{w+l_v} = -1$ else $s_w = -1$, $s_{w+l_v} = 0$
- If $s'_w = -1$, $s'_{w+l_v} = -1$ and $\widehat{m_0} = 1$ then $s_w = 1$, $s_{w+l_v} = 0$ or $s_w = 0$, $s_{w+l_v} = 1$
  - If $s'_w = 1$, $s'_{w+l_v} = -1$ and $\widehat{m_0} = 1$ then $s_w = 0$, $s_{w+l_v} = 1$ else $s_w = 1$, $s_{w+l_v} = 0$

□

Proposition 16 works like Proposition 15 but for the case where one of the two targeted coefficient is equal to 0. However, as previously, Proposition 15 and Proposition 16 are not enough to mount an attack for the same reasons:

- Eve needs a way to verify what Alice obtains.
- A bit of difference on $\widehat{m}$ is corrected by the BCH code. Thus, at the end of the decryption procedure Alice and Eve have the same plaintext.

Nonetheless, Eve can use Proposition 15 and Proposition 16, the BCH code decryption failure and the oracle to overcome these issues.

Theorem 2. *Let $s'_w, s'_{w+l_v} \in \{-1, 1\}$ be the guessed values of s_w and s_{w+l_v} done by Eve. If Eve takes a session key $\mu_{s'_w, s'_{w+l_v}}$ then she can forge $c_{s'_w, s'_{w+l_v}} = (c_1, c_2)$ depending of $\mu_{s'_w, s'_{w+l_v}}$ by using properties given in Proposition 13 such that by calling $\mathcal{O}(c_{s'_w, s'_{w+l_v}}, \mu_{s'_w, s'_{w+l_v}})$ with $s'_w, s'_{w+l_v} \in \{-1, 0, 1\}$, she retrieves the w-th and $w + l_v$-th coefficients of s. In consequence, Eve needs at most $8 \times (N - l_v)$ calls to the oracle in order to retrieve two coefficients of Alice's secret key.*

Proof. The idea is the same as LAC-128 and 192, Eve takes c_2 to ensure, after comparison in `CPA.Decrypt256`, that $\widehat{m}$ is a codeword with $\frac{d}{2}$ errors if she did a wrong key guess. Since at most $\frac{d-1}{2}$ errors can be corrected, a decoding errors occurs.
According to Proposition 15 and Proposition 16, Eve can monitor Alice's decryption procedure if she does the good key guess.
Suppose Eve wants to retrieve the w-th and the $(w + l_v)$-th coefficients of s:

1. Eve chooses a codeword called *cdword* with a 1 at the first coordinate such that $cdword = mG$ where G is the generator matrix of the BCH code
2. Eve injects $\frac{d-1}{2}$ errors to *cdword* at any coordinate except the first one
3. Eve chooses a verifying $\frac{q}{8} < a < \frac{q}{4}$ if she is on the case of Proposition 15 or $\frac{q}{6} < a < \frac{q}{4}$ if she is on the case of Proposition 16
4. Eve constructs c_1 and c_2 with her key guesses at the first and l_v-th coefficient of c_2: $c_2[0] = as'_w$ and $c_2[l_v] = as'_{w+l_v}$ and such that after comparison, Alice retrieves *cdword* with $\frac{d-1}{2}$ errors or *cdword* with d errors
5. Eve sends $c = (c_1, c_2)$ to Alice

With this construction Alice obtains a codeword with $\frac{d}{2}$ errors if Eve does a wrong key guess. At this point, Eve's session key is $sess_E = H(pk, m)$ and Alice's session key $sess_A$ depends on Eve's key guesses. Eve can verify if she did a good key guess with the oracle.
First Eve determines if s_w and s_{w+l_v} are different from 0 (see Proposition 15):

If $s'_w = 1, s'_{w+l_v} = 1$ and $\mathcal{O}(c, sess_E) = 1$ then $s_w = -1$ and $s_{w+l_v} = -1$
Else If $s'_w = -1, s'_{w+l_v} = -1$ and $\mathcal{O}(c, sess_E) = 1$ then $s_w = 1$ and $s_{w+l_v} = 1$
Else If $s'_w = 1, s'_{w+l_v} = -1$ and $\mathcal{O}(c, sess_E) = 1$ then $s_w = -1$ and $s_{w+l_v} = 1$
Else If $s'_w = -1, s'_{w+l_v} = 1$ and $\mathcal{O}(c, sess_E) = 1$ then $s_w = 1$ and $s_{w+l_v} = -1$

If the oracle does not return 1, then Eve determines which coefficient is equal to 0 (see Proposition 16):

If $s'_w = 1, s'_{w+l_v} = 1$ and $\mathcal{O}(c, sess_E) = 1$ then $s_w = -1$ and $s_{w+l_v} = 0$
or $s_w = 0$ and $s_{w+l_v} = -1$
  If $s'_w = -1, s'_{w+l_v} = 1$ and $\mathcal{O}(c, sess_E) = 1$ then $s_w = 0$ and $s_{w+l_v} = -1$
  Else If $\mathcal{O}(c, sess_E) = -1$ then $s_w = -1$ and $s_{w+l_v} = 0$
Else If $s'_w = -1, s'_{w+l_v} = -1$ and $\mathcal{O}(c, sess_E) = 1$ then $s_w = 1$ and $s_{w+l_v} = 0$
or $s_w = 0$ and $s_{w+l_v} = 1$
  If $s'_w = 1, s'_{w+l_v} = 1$ and $\mathcal{O}(c, sess_E) = 1$ then $s_w = 0$ and $s_{w+l_v} = 1$
  Else if $\mathcal{O}(c, sess_E) = -1$ then $s_w = 1$ and $s_{w+l_v} = 0$
Otherwise $s_w = 0$ and $s_{w+l_v} = 0$

Eve can applies this procedure for $0 \leq w < (N - l_v)$ to retrieve the entire secret key. □

To recover the entire key we need at most $8 \times (N - l_v)$ requests to the oracle due to Theorem 2, where $l_v = 400$ and $N = 1024$.

Full version The subroutine `Compress` removes the 4 lowers bits of each coeff of c_2. They are replaced by 4 zero-bit when the subroutine `Decompress` is applied at the beginning of the decryption process. So each coefficient of c_2 can be only equal to 16, 32, 64, 128 and any sum of these values.

For c_2 in our attack we choose $a \approx \frac{q}{7}$ for Proposition 15 and $a \approx \frac{q}{5}$ for Proposition 16. Then, we only consider the values $\frac{q}{7}$, $-\frac{q}{7}$, $\frac{q}{5}$, $-\frac{q}{5}$ and $\frac{q}{2}$. In our implementation [Mona] we approximate $\frac{q}{7} \approx 32$, $-\frac{q}{7} \approx 128 + 64 + 16 = 210$ or $-\frac{q}{7} \approx 128 + 64 + 32 = 224$ (we use two different values to compensate the approximation), $\frac{q}{5} \approx 16 + 32 = 48$, $-\frac{q}{5} \approx 128 + 64 = 192$ and $\frac{q}{2} \approx 128$. Proposition 15 and Proposition 16 are still verified and we still retrieve s with at most $8 \times (N - l_v)$ requests to the oracle by the Theorem 2.

Implementation results We assess our attack implementation [Mona] plug in the reference code of LAC. Following results are the average of 1000 attacks launched on 1000 random secret keys for LAC-256. Timing results have been evaluated on core i5-8350U at 1.90GHz.

	Nb of coeff of sk	Average oracle requests	Average time
LAC-256	1024	3355	30, 31 ms

In average we need $5,4 \times (1024 - 400)$ oracle requests that is much less than the upper bound of $8 \times (N - l_v)$ requests.

7.2 Attack on LAC CCA key exchange using side-channel leakage

In the previous section we have shown an attack on a misuse situation. Hence, with an implementation complying the specification, this forgery attack is not effective. Most of the lattice based key encapsulation mechanisms base their CCA security on the Fujisaki-Okamoto (FO) transformation. From the CPA version of a scheme, this transformation puts additional checks to ensure CCA security. More precisely, the transformation prevents chosen ciphertext attack during the decapsulation routine. The CCA decapsulation can be summarized in three steps:

1. Decipher the received ciphertext c using the IND-CPA decrypt algorithm.
2. Encrypt the plaintext obtained at Step 1 to obtain a cipertext c'.
3. Compare c and c'. Accept if $c = c'$.

This transformation ensures the IND-CCA security thanks to the re-encryption and the verification. The LAC specification states that the same secret key can be used for several CCA key exchanges. However, the FO transformation supposes that an attacker does not have access to information during the three steps. As mentioned in Chapter 5, embedded devices are threatened by physical attacks that allow to learn information about intermediate values. In the following we determine which side-channel leakage or fault injection allow to bypass the CCA security in order to apply the attack presented in Section 7.1.

7.2.1 Physical attacks against LAC CCA key exchange

The `CCA.Decapsulation` routine, described in Algorithm 40, manipulates the secret key during a `CPA.Decrypt` or `CPA.Decrypt256` routine like in Section 7.1. After the whole CPA computation the ciphertext is then verified. Due to this verification the oracle instantiation of Definition 3 page 96 cannot be achieved. Then in this section, we investigate new ways to instantiate the oracle using physical attacks.

For sake of clarity, in the following we suppose that the `CCA.Decapsulation` routine uses only `CPA.Decrypt` algorithm. The following statements apply similarly to `CPA.Decrypt256` algorithm.

Algorithm 40 `CCA.Decapsulation`$(s_k, c = (c_1, c_2))$

Output: An encapsulated key K.

1: $m \leftarrow$ `CPA.Decrypt`(s_k, c) or `CPA.Decrypt256`(s_k, c)
2: $H \leftarrow H(m, c)$
3: seed $\leftarrow G(m)$
4: $c' \leftarrow$ `CPA.Encrypt`(p_k, m, seed)
5: **if** $c' \neq c$ **then**
6: $\quad K \leftarrow H(H(s_k), c)$
7: **end if**
8: **return** K

Algorithm 41
`CPA.Decrypt`$(s_k, c = (c_1, c_2))$

Output: Plaintext m

1: $c_2 \leftarrow$ `Decompress`(c_2)
2: $\widehat{M} \leftarrow c_2 - (c_1 s_k)_{l_v} \in \mathbb{Z}_q^{l_v}$
3: **for** $i = 0$ to $l_v - 1$ **do**
4: $\quad$ **if** $\frac{q}{4} \leq \widehat{M}_i < \frac{3q}{4}$ **then**
5: $\quad\quad \widehat{m}_i \leftarrow 1$
6: $\quad$ **else**
7: $\quad\quad \widehat{m}_i \leftarrow 0$
8: $\quad$ **end if**
9: **end for**
10: $m \leftarrow$ `BCHDecode`$(\widehat{m})$
11: **return** m

In Section 7.1, we forge a ciphertext such that the key guess is validated or not by the first coordinate of the plaintext. Therefore, in the following we determine the leakage points which can bring information about this first coordinate.

Side-channel attacks inside `CPA.Decrypt` algorithm

Computation of $\hat{M}$. The forged ciphertext described in Section 7.1 ensures that during the computation $\hat{M} \leftarrow c_2 - (c_1 s_k)_{l_v}$ (Algorithm 41 Line 2), only one coefficient (two coefficients in the case of LAC 256) of

s_k is implied in each coordinate of $\hat{M}$. Hence, the secret coefficients are more likely to leak in side-channel. More precisely, an attacker can perform correlation power analysis (CPA) by sending various ciphertexts to learn information about the secret coefficients.

Reconciliation step. The forged ciphertext is chosen such that the first coordinate of $\hat{m}_i$ is equal to 1 if the attacker does the good key guess and 0 otherwise. Therefore, as previously, the attacker can send various ciphertext in order to mount CPA attack which focuses on the reconciliation step (Line 3 to 9 of Algorithm 41) to determine whether the first coefficient is equal to 1 or not.

BCH decoding. Another attack path is the BCH decoding algorithm. Indeed, if the attacker does the wrong key guess this implies that the retrieved $\hat{m}$ ensures a decoding failure. If the BCH decoding is not implemented in constant time (see [WBB+19]) the attacker can determine whether he does the right key guess or not.

Physical attacks outside `CPA.Decrypt` algorithm

Hash functions. In order to retrieve the first coefficient of the plaintext m the attacker can focus on the hash functions which manipulate it. LAC's reference implementation uses SHA-3 as a hash function, which is based on Keccak. Single trace side channel attacks on Keccak have already been carried out [KPP20]. Such attacks allow the attacker to find the manipulated inputs. Then, such attacks can be mounted against the hash operation at Line 2 and 3 of Algorithm 40 to learn information about m.

Fault injection against the verification. The previous attack paths focus on instantiate the oracle describes in Section 7.1 using side-channel information. However, if an attacker can bypass the comparison at Line 4 then the attacker can apply straightforwardly the CPA key exchange attack.

Chapter 8

High-order masking of lattice-based KEM

Contents

The results presented in this chapter are from a joint work with Jean Sébastien-Coron, François Gérard and Rina Zeitoun. The articles which refer to these results are [CGM+21a; CGM+21b].

In this chapter we are interested in protecting the lattice-based KEMs Kyber and Saber against side-channel attacks. More precisely, we provide high-order masking countermeasures proven secure in the probing model and we implement them in order to propose high-order secure decapsulation implementations Kyber and Saber. The decapsulation routine can be described at a high level in three stages:

1. IND-CPA decryption of the ciphertext c to obtain a message m
2. Re-encryption of m into a ciphertext c'.
3. Polynomial comparison between c and c'.

To obtain a fully masked implementation, all three steps must be masked, otherwise this can lead to a CCA attack.

Masking countermeasures require, most of the time, conversion algorithms to switch an arithmetic masking to a Boolean one and conversely. This is notably the case for Kyber and Saber, which use arithmetic and Boolean operations. Our first contribution is a generic high-order table-based conversion algorithm which finds applications to protect Kyber and Saber. However, the conversion algorithm is not enough to completely mask these schemes. Therefore, our second contribution is high-order polynomial comparisons and the high-order implementations of Kyber and Saber.

In the following we provide a state of the art of the conversion algorithms and the masked implementations of Kyber and Saber.

First-order conversions algorithms.

The first conversion algorithms were proposed by Goubin in [Gou01], with security against first-order attacks. The Boolean to arithmetic conversion is efficient and has an optimal complexity $\mathcal{O}(1)$. The conversion from arithmetic modulo 2^k to Boolean masking is less efficient as its complexity is $\mathcal{O}(k)$.

This was later improved to $\mathcal{O}(\log k)$ in [CGT+15]; however in practice for $k = 32$ the number of operations was similar.

A table-based conversion from arithmetic to Boolean masking was described in [CT03], for first-order security only. For a small value of k, the conversion can be done by a simple table look-up, using a pre-computed table. This is similar to the classical first-order randomized SBox table countermeasure [CJR+99]. More precisely, the algorithm uses a randomized pre-computed table $T : \mathbb{Z}_{2^k} \to \{0,1\}^k$ which is initialized as follows. First, one generates a random mask $r \xleftarrow{\$} \mathbb{Z}_{2^k}$. As a second step, one computes $T[v] = (v + r) \oplus r$ for all $v \in \mathbb{Z}_{2^k}$. Then, given an arithmetically masked value $A = x - r \mod 2^k$, one obtains a Boolean masking x' of x by simply reading the table T at index A, *i.e.* $x' = T[A]$; this gives $x' = (A + r) \oplus r = x \oplus r$ as required. The same randomized table can be used multiple times; therefore once the table has been initialized for all possible values in $\mathbb{Z}_{2^k}$, each conversion is a simple table look-up.

The authors also showed how to extend the technique to convert variables of $k = \delta \cdot \ell$ bits, by propagating the carry by blocks of ℓ bits. However there was a flaw in their algorithm: they computed the carry table modulo 2^ℓ only, instead of modulo $2^{k-\ell}$; therefore the algorithm is incorrect for $\delta > 2$; this mistake was identified and corrected by Debraize in [Deb12]. In [Deb12], the author described multiple first-order conversion algorithms, but one of them was recently found insecure in [BDV21], where two corrected algorithms are described.

High-order conversion algorithms.

The first conversion algorithms secure against high-order attacks were described in [CGV14], with complexity $\mathcal{O}(\alpha^2 \cdot k)$ for α shares and k-bit variables. The authors described conversions algorithms in both directions, with a security proof in the probing model. Using the technique from [CGT+15], the complexity can be improved to $\mathcal{O}(\alpha^2 \cdot \log k)$ as in the first-order case, however in practice the number of operations for $k = 32$ is also similar. The technique described in [CGV14; CGT+15] considers arithmetic masking modulo 2^k. This was later extended in [BBE+18] to arithmetic masking modulo any integer q, in the context of masking the GLP lattice-based signature scheme; the complexity is also $\mathcal{O}(\alpha^2 \cdot k)$ or $\mathcal{O}(\alpha^2 \cdot \log k)$ for a k-bit integer q. More precisely, the authors provided the extension of [CGV14; CGT+15] to arbitrary modulus with cubic complexity in α; the extension with quadratic complexity in α is provided in [SPO+19].

The approach used in [CGV14] to perform the Boolean to arithmetic conversion requires to first perform an arithmetic to Boolean conversion. An alternative, more direct approach is described in [SPO+19], also with complexity $\mathcal{O}(\alpha^2 \cdot k)$. It also works with arithmetic masking modulo an arbitrary q. The technique is based on a 1-bit Boolean to arithmetic masking conversion with complexity $\mathcal{O}(\alpha^2)$. Such 1-bit Boolean

to arithmetic conversion is interesting in the context of lattice-based cryptography, for masking the re-encryption of the message, and for masking the binomial sampling.

Finally, a high-order Boolean to arithmetic conversion algorithm was described in [Cor17], and later simplified in [BCZ18], with complexity $\mathcal{O}(2^\alpha)$; that is, the complexity is independent from the size of the k-bit variable that must be converted. The technique can be seen as a high-order extension of the original first-order Boolean to arithmetic algorithm from [Gou01]. Although the complexity is exponential in α, for small values of α the algorithm is at least one order of magnitude faster than [CGV14; CGT+15]. However for algorithms in [Cor17; BCZ18], the arithmetic masking is modulo 2^k only; we do not know how to extend the technique to any modulus q.

First and high-order implementations of Kyber and Saber.

In [BGR+21], the authors described the first completely masked implementation of Kyber, secure against first-order and higher-order attacks. For the IND-CPA decryption (Step 1), the authors consider the threshold function $\mathsf{th}(x)$ outputting 0 if $x < q/2$ and 1 otherwise. They show that

$$\mathsf{th}(x) = x_{11} \oplus (\neg x_{11} \cdot x_{10} \cdot x_9 \cdot (x_8 \oplus (\neg x_8 \cdot x_7)))$$

where x_i is the i-th bit of x; namely this corresponds to a binary comparison with the threshold $\lfloor q/2 \rfloor$. This implies that the high-order computation of $\mathsf{th}(x)$ can be performed by first converting the masking of x from arithmetic modulo q to Boolean, using [BBE+18]; then $\mathsf{th}(x)$ can be computed with high-order secure implementations of the And and Xor gadgets. For the high-order polynomial comparison (Step 3), the technique of [BGR+21] consists in performing the comparison with uncompressed ciphertexts. The advantage of this approach is that the ciphertext compression from Kyber does not need to be explicitly masked. Given the masked uncompressed polynomials obtained from re-encryption, the ciphertext comparison requires to check that every coefficient belongs to a certain public range modulo q, instead of checking for equality.

In [SPO+19], the authors described an efficient technique for high-order masking the binomial sampling in the re-encryption of m at Step 2 above, based on a 1-bit Boolean to arithmetic conversion modulo q with complexity $\mathcal{O}(\alpha^2)$; their technique is an extension of a first-order algorithm from [OSP+18b].

First contribution: high-order table-based conversion.

Our first contribution is to extend the table-based conversion algorithm between Boolean and arithmetic masking of [CT03] from first-order to any order. For this we extend the high-order table recomputation countermeasure from [Cor14]. Namely we observe that in [Cor14], the incremental shifting of the rows of the table T can be performed according to any additive group G, not only for the xor operation in $\{0,1\}^k$. For example we can work modulo 2^k as input, which automatically gives a high-order conversion from arithmetic to Boolean masking. Similarly, the α-encoding of the rows of T as output can be according to any group law, not only for the xor in $\{0,1\}^k$. This implies that we can easily convert from Boolean to arithmetic masking modulo any integer q, which is useful in the context of lattice-based cryptography (see below).

More generally, our extended table recomputation countermeasure allows computing any function $f : G \to H$, for any group G as input and any group H as output. Given as input an α-sharing of $x = x_1 + \cdots + x_\alpha \in G$, we can compute α outputs shares $y_i \in H$ such that $y_1 + \cdots + y_\alpha = f(x_1 + \cdots + x_\alpha)$, while being secure in the ISW probing model against $t = \alpha - 1$ probes. By selecting the right groups G and H, we can therefore obtain high-order secure conversion algorithms between Boolean and arithmetic masking. To convert from Boolean to arithmetic masking modulo 2^k, we take $G = \{0,1\}^k$ and $H = \mathbb{Z}_{2^k}$ and we obtain $y_1 + \cdots + y_\alpha = x_1 \oplus \cdots \oplus x_\alpha \mod 2^k$ as required. Similarly for arithmetic modulo 2^k to Boolean conversion we take $G = \mathbb{Z}_{2^k}$ and $H = \{0,1\}^k$, and we obtain $y_1 \oplus \cdots \oplus y_\alpha = x_1 + \cdots + x_\alpha \mod 2^k$ as required.

The main advantage of the table-based approach for conversions is its flexibility, as we can choose any groups G and H and any function $f : G \to H$. However the running time complexity is $\mathcal{O}(\alpha^2 \cdot |G|)$. This implies that for k-bit Boolean or arithmetic masking, the generic complexity is $\mathcal{O}(\alpha^2 \cdot 2^k)$.

Moreover, we show how to efficiently compute a threshold function th from arithmetic masking modulo 2^k, whose result is a 1-bit Boolean masking. This corresponds to the decryption function in lattice-based cryptosystems which is useful for Kyber and Saber. In that case, our optimization consists in putting each column of the table in a single register; the resulting complexity is $\mathcal{O}(\alpha^2)$ only, assuming that we have access to 2^k-bit registers. In practice, for this optimization we obtain at least an order of magnitude improvement compared to the techniques in [CGV14; BBE+18].

Second contribution: high-order masking of lattice-based encryption schemes.

Our second contribution is to provide a fully masked implementation of Kyber and Saber. To do so, we first apply our table-based conversion algorithms to mask the IND-CPA encryption and decryption. Secondly, we provide high order polynomial comparisons to fully mask the IND-CCA decryption.

We first consider the IND-CPA decryption of the ciphertext c (Step 1 page 109). For ring-LWE encryption the ciphertext $c = (c_1, c_2)$ is decrypted with the private key s using $m = \mathsf{th}(c_1 - s \cdot c_2)$, where th is the threshold function $\mathsf{th} : \mathbb{Z}_q \to \{0, 1\}$ where $\mathsf{th}(x) = 1$ if $x \in [q/4, 3q/4[$ and $\mathsf{th}(x) = 0$ otherwise. The threshold function is actually applied independently on each coefficient of the polynomial $u = c_1 - s \cdot c_2$ modulo q. When the private key s is arithmetically masked modulo q with α shares, we obtain α shares for $u = u_1 + \cdots + u_\alpha \mod q$, and we must therefore convert from an arithmetically masked u modulo q into a 1-bit Boolean masked $m = m_1 \oplus \cdots \oplus m_\alpha = \mathsf{th}(u)$. For this one could use our generic table-based approach with the function $f = \mathsf{th}$ and $f : \mathbb{Z}_q \to \{0, 1\}$. However the complexity would be $\mathcal{O}(\alpha^2 \cdot q)$, which is prohibitive for large q. Therefore we describe an optimization in which we first perform a modulus switching from masking modulo q to masking modulo 2^k (for a small k), while maintaining a negligible probability of decryption error, as required to achieve CCA-security [DNR04]. We can then convert from arithmetic masking modulo 2^k into 1-bit Boolean masking, which recovers the Boolean masked message m. This optimization has complexity $\mathcal{O}(\alpha^2 \cdot \log \alpha)$, instead of $\mathcal{O}(\alpha^2 \log q)$ with [BBE+18]. In practice we obtain an order of magnitude improvement in the IND-CPA decryption of Kyber.

We also consider the masking of re-encryption of m into a ciphertext c', and the masking of the binomial sampling (Step 2 page 109). To encrypt a Boolean masked message $m \in \{0, 1\}$, we can use our generic table-based Boolean to arithmetic modulo q conversion algorithm.

In that case the complexity is $\mathcal{O}(\alpha^2)$ as in [SPO+19]. The same holds for the masking of the binomial sampling, which is easily computed as the sum of independent 1-bit Boolean to arithmetic modulo q conversions, as in [SPO+19]. In practice we obtain a similar level of efficiency as in [SPO+19], and an order of magnitude improvement compared to [BBE+18].

Afterwards, we focus on the high-order polynomial comparison (Step 3 page 109). We consider several techniques, firstly for zero testing a single coefficient, and then zero testing a set of polynomials at once. As an application, we consider the high-order polynomial comparison in Kyber and Saber for IND-CCA decryption.

Finally, we provide a detailed description of the masking of the full IND-CCA decryption of the Kyber and Saber schemes at any order. We also describe the practical results of a C implementation of the full high-order masking of Kyber and Saber. The source code is public and can be found at

https://github.com/fragerar/HOTableConv/tree/main/Masked_KEMs

8.1 High-order table-based conversion and applications

8.1.1 High-order table-based conversion algorithm

In this section we introduce our generic high-order table-based conversion algorithm, as an extension of the table recomputation countermeasure from [Cor14]. We consider two additive groups G and H and a function $f : G \to H$. Our algorithm takes as input α shares $x_1, \ldots, x_\alpha \in G$ and outputs α shares $y_1, \ldots, y_\alpha \in H$ such that:

$$y_1 + \cdots + y_\alpha = f(x_1 + \cdots + x_\alpha)$$

We stress that the function f does not need to have any special property, except being efficiently computable. In particular it need not be a group homomorphism, as in general the groups G and H will not be homomorphic. Note that the high-order SBox computation algorithm from [Cor14] is a particular case with $G = H = \{0,1\}^k$ and $f(x) = S(x)$.

The algorithm consists in progressively shifting a randomized table T, using the input shares $x_1, \ldots, x_{\alpha-1}$ for the successive shifts. The randomized table T has $|G|$ rows, and each row is a vector of α shares, which encodes over H the function $f(x)$, but progressively shifted by $x_1, \ldots, x_{\alpha-1} \in G$. Eventually one reads the table at index x_α, which gives an α-sharing (y_i) over H of $f(x_1 + \cdots + x_\alpha)$ as required. Between every shift, the α shares of every row are refreshed using the same mask refreshing as in [RP10], but over the group H. This mask refreshing is not SNI contrary to the one presented in Algorithm 26 (page 80). However, it is more efficient and its lower security is enough for the security proof, see Theorem 3.

As we will see in more details in Section 8.1.2, for a Boolean to arithmetic conversion algorithm, one will take $G = \{0,1\}^k$ and $H = \mathbb{Z}_{2^k}$. Then by identifying k-bit strings and integers modulo 2^k and taking f the identity function, we obtain $y_1 + \cdots + y_\alpha \bmod 2^k = x_1 \oplus \cdots \oplus x_\alpha$ as required. Similarly, for an arithmetic to Boolean conversion, one takes $G = \mathbb{Z}_{2^k}$ and $H = \{0,1\}^k$ and obtains $y_1 \oplus \cdots \oplus y_\alpha = x_1 + \cdots + x_\alpha \bmod 2^k$ as required; see Section 8.1.3 for more details.

Algorithm 42 $\mathsf{Convert}_{G,H,f}$

Input: $x_1, \ldots, x_\alpha \in G$
Output: $y_1, \ldots, y_\alpha \in H$ such that $y_1 + \cdots + y_\alpha = f(x_1 + \cdots + x_\alpha)$

1: **for all** $u \in G$ **do** $T(u) \leftarrow (f(u), 0, \ldots, 0) \in H^\alpha$
2: **for** $i = 1$ to $\alpha - 1$ **do**
3: **for all** $u \in G$ **do** $T'(u) \leftarrow T(u + x_i)$
4: **for all** $u \in G$ **do** $T(u) \leftarrow \mathsf{Refresh}_H(T'(u))$
5: **end for**
6: $(y_1, \ldots, y_\alpha) \leftarrow \mathsf{Refresh}_H(T(x_\alpha))$
7: **return** $y_1, \ldots, y_\alpha$

Algorithm 43 $\mathsf{Refresh}_H$

Input: $x_1, \ldots, x_\alpha \in H$
Output: $y_1, \ldots, y_\alpha \in H$ such that $y_1 + \cdots + y_\alpha = x_1 + \cdots + x_\alpha$

1: $y_\alpha \leftarrow x_\alpha$
2: **for** $j = 1$ to $\alpha - 1$ **do**
3: $r_j \xleftarrow{\$} H$
4: $y_j \leftarrow x_j + r_j$
5: $y_\alpha \leftarrow y_\alpha - r_j$
6: **end for**
7: **return** $y_1, \ldots, y_\alpha$

We provide a pseudocode description in Algorithm 42 above. The algorithm uses two temporary tables T and T' in RAM, with $|G|$ rows, where each row contains a vector of α elements in H. The table T is initialized at Line 1 with $T(u) \leftarrow (f(u), 0, \ldots, 0) \in H^\alpha$. Given an encoding $\vec{v} = (v_1, \ldots, v_\alpha)$ with α shares in H, we denote by

$$\sum(\vec{v}) = v_1 + \cdots + v_\alpha$$

the encoded element in H. This implies that initially we have $\sum(T(u)) = f(u)$ for all rows $u \in G$. For the first index $i = 1$, the table is shifted at Line 3 by x_1 into T', which gives $\sum(T'(u)) = f(u + x_1)$ for all $u \in G$. Note that the shift is performed according to the group law in G. The rows are then refreshed at Line 4 using Algorithm 43, and we still have $\sum(T(u)) = f(u + x_1)$. More generally, after the shift by $x_1, \ldots, x_i$ we obtain at Line 4:

$$\sum(T(u)) = f(u + x_1 + \cdots + x_i)$$

for all $u \in G$, and after all the input shares $x_1, \ldots, x_{\alpha-1}$ have been processed we have:

$$\sum(T(u)) = f(u + x_1 + \cdots + x_{\alpha-1})$$

Therefore from the final look-up table $(y_1, \ldots, y_\alpha) \leftarrow \mathsf{Refresh}_H(T(x_\alpha))$, we obtain that $\sum(\vec{y}) = \sum(T(x_\alpha)) = f(x_\alpha + x_1 + \cdots + x_{\alpha-1})$. This gives $y_1 + \cdots + y_\alpha = f(x_1 + \cdots + x_\alpha)$, which proves the correctness of the algorithm.

Lemma 5. *Let $(y_i)_{1 \le i \le \alpha}$ be the input and let $(z_i)_{1 \le i \le \alpha}$ be the output of $\mathsf{Refresh}_H$. Any subset of $\alpha - 1$ output variables z_i is uniformly and independently distributed in H.*

Complexity. In this chapter we provide complexity for several algorithms. Complexities are used to compare different algorithms asymptotically. Therefore, we only consider Boolean or arithmetic operation. Moreover, we assume that a group operation in G and H takes unit time, as well as randomness generation and table transfer. For α shares, the number of operations of $\mathsf{Refresh}_H$ is $3\alpha - 3$. The time complexity of Algorithm 42 is therefore:

$$\begin{aligned} \mathcal{C}_{\mathsf{convert}} &= |G| \cdot (\alpha + (\alpha - 1) \cdot (1 + \alpha + 3\alpha - 3)) + 3\alpha - 3 \\ &= |G| \cdot (4\alpha^2 - 5\alpha + 2) + 3\alpha - 3 \simeq 4 \cdot |G| \cdot \alpha^2 \end{aligned}$$

The asymptotic complexity is therefore $\mathcal{O}(|G| \cdot \alpha^2)$. The memory complexity is $\mathcal{O}(|G| \cdot \alpha)$. The algorithm requires $(\alpha - 1) \cdot (|G| \cdot (\alpha - 1) + 1)$ random elements.

Security. We prove that our algorithm achieves the t-SNI definition (Definition 2). One can therefore use our algorithm inside a more complex construction and achieve security against t probes with $\alpha = t + 1$ shares.

Theorem 3 ($(\alpha - 1)$-SNI of $\mathsf{Convert}_{G,H,f}$)**.** *For any subset $O \subset [1, \alpha]$ and any t_1 intermediate variables with $|O| + t_1 < \alpha$, the output variables $y_{|O}$ and the t_1 intermediate variables can be perfectly simulated from the input variables $x_{|I}$, with $|I| \le t_1$.*

Proof. The proof of Theorem 3 is relatively similar to the proof of the table-based countermeasure in [Cor14]. Given $u \in G$, we denote by $T(u)[j]$ and $T'(u)[j]$ the j-th component of the vectors $T(u)$ and $T'(u)$ respectively, for $1 \le j \le \alpha$. We denote by *Part i* the computation performed within the main for loop, that is between lines 2 and 5 of Algorithm 42, and by *Part n* the computation performed at line 6. We describe hereafter the construction of two index sets I and J, both initially empty.

- For every probed input variable x_i or any intermediate variable $u + x_i$ (for any $1 \le i \le \alpha$), we add i to I.

- For every probed intermediate variable $T'(u)[j] = T(u + x_i)[j]$, or r_j or y_j in $\mathsf{Refresh}_H$ (for any $1 \leq i \leq \alpha$) in Part i, or $T(u)[j]$ in line 4 of Part i, we add i to I and j to J.

- For every output y_j such that $j \in O$, we add j to J.

Since for every probed variable we added at most one index in I, we have $|I| \leq t_1$ as required. Similarly for J we must have $|J| \leq |O| + t_1 < \alpha$.

We now show that any set of t_1 variables and the output shares $y_{|O}$ can be perfectly simulated from $x_{|I}$. This is clear for the probed input variables x_i and intermediate variables $u + x_i$ (for any $1 \leq i \leq \alpha$ and all $u \in G$), since by construction $i \in I$. It remains to perfectly simulate all probed variables of the form $T(u)[j]$, $T'(u)[j]$ and $T(u + x_i)[j]$, including the output variables $y_{|O}$. We proceed by induction on i. Namely we show that at the beginning of each part i, we can perfectly simulate all variables $T(u)[j]$ for all $j \in J$ and all $u \in G$. This holds for the case $i = 1$, since at the beginning of Part 1, the vector $T(u) = (f(u), 0, \ldots, 0)$ is publicly known. At the beginning of Part i, we distinguish two cases:

Case $i \in I$. If $i \in I$ then knowing x_i we can perfectly simulate all intermediate variables with column index $j \in J$ in Part i, as knowing x_i we can propagate the simulation for all variables with column index j and perfectly simulate $T(u+x_i)[j]$, $T'(u)[j]$ and the resulting $T(u)[j]$ at Line 4, and similarly the variables y_j at Line 6 if $i = \alpha$; in particular the r_j variables within $\mathsf{Refresh}_H$ are simulated exactly as in the $\mathsf{Refresh}_H$ procedure.

Case $i \notin I$. If $i \notin I$ then no variable in Part i has been probed, including variables in $\mathsf{Refresh}_H$. Since $|J| < \alpha$, using Lemma 5 we can therefore perfectly simulate all intermediate variables $T(u)[j]$ for $j \in J$ and $u \in G$ at the output of $\mathsf{Refresh}_H$ at Line 4, or similarly all y_j for $j \in J$ at the output of $\mathsf{Refresh}_H$ at Line 6 when $i = \alpha$, simply by generating uniform and independent values.

As a conclusion, we have shown that for all i the induction step is verified, which means that all $T(u)[j]$, $T'(u)[j]$ and $T'(u+x_i)[j]$ for $j \in J$ can be perfectly simulated from $x_{|I}$, including the output variables $y_{|O}$. Therefore all probes can also be perfectly simulated. This terminates the proof of Theorem 3. □

8.1.2 Table-based high-order Boolean to arithmetic conversion

In this section we consider the case of Boolean to arithmetic conversion. We describe the straightforward application of the generic table-based conversion algorithm from G to H from Section 8.1.1. In this section we consider the easiest case with the conversion from Boolean to arithmetic masking. We consider the other direction in Section 8.1.3.

Direct approach

We consider the straightforward application of Algorithm 42 to high-order Boolean to arithmetic conversion, which can be used for small values of k. We consider an integer q. We identify k-bit strings with integers in the interval $[0, 2^k[$. Algorithm 44 below describes a Boolean to arithmetic masking conversion algorithm such that given $x_1, \ldots, x_\alpha \in \{0,1\}^k$ as input, we obtain $y_1, \ldots, y_\alpha \in \mathbb{Z}_q$ as output, with, $x_1 \oplus \cdots \oplus x_\alpha = y_1 + \cdots + y_\alpha \mod q$. The $(\alpha - 1)$-SNI security follows directly from Theorem 3.

Algorithm 44 BooleanToArithmetic

Input: $k \in \mathbb{Z}$ and $x_1, \ldots, x_\alpha \in \{0,1\}^k$
Output: $y_1, \ldots, y_\alpha \in \mathbb{Z}_q$ such that $y_1 + \cdots + y_\alpha \bmod q = x_1 \oplus \cdots \oplus x_\alpha$

1: **for all** $u \in \{0,1\}^k$ **do** $T(u) \leftarrow (u \bmod q, 0, \ldots, 0)$
2: **for** $i = 1$ to $\alpha - 1$ **do**
3: **for all** $u \in \{0,1\}^k$ **do** $T'(u) \leftarrow T(u \oplus x_i)$
4: **for all** $u \in \{0,1\}^k$ **do** $T(u) \leftarrow \mathsf{Refresh}_{\mathbb{Z}_q}(T'(u))$
5: **end for**
6: $(y_1, \ldots, y_\alpha) \leftarrow \mathsf{Refresh}_{\mathbb{Z}_q}(T(x_\alpha))$
7: **return** $y_1, \ldots, y_\alpha$

Algorithm 45 $\mathsf{Refresh}_{\mathbb{Z}_q}$

Input: $x_1, \ldots, x_\alpha \in \mathbb{Z}_q$
Output: $y_1, \ldots, y_\alpha \in \mathbb{Z}_q$ such that $y_1 + \cdots + y_\alpha = x_1 + \cdots + x_\alpha \mod q$

1: $y_\alpha \leftarrow x_\alpha$
2: **for** $j = 1$ to $\alpha - 1$ **do**
3: $r_j \xleftarrow{\$} \mathbb{Z}_q$
4: $y_j \leftarrow x_j + r_j \bmod q$
5: $y_\alpha \leftarrow y_\alpha - r_j \bmod q$
6: **end for**
7: **return** $y_1, \ldots, y_\alpha$

Complexity. We do not take into account the reductions modulo q. The operation count is the same as for Algorithm 42, with $|G| = 2^k$, which gives:

$$\mathcal{C}_{\mathsf{BA}}(k, \alpha) = 2^k \cdot (4\alpha^2 - 5\alpha + 2) + 3\alpha - 3 \simeq 2^{k+2} \cdot \alpha^2$$

The memory complexity is $\mathcal{O}(2^k \cdot \alpha)$. The algorithm requires $(\alpha - 1) \cdot (2^k \cdot (\alpha - 1) + 1)$ random elements in $\mathbb{Z}_q$.

Comparison with existing techniques

1-bit Boolean to arithmetic modulo 2^k conversion. The 1-bit Boolean to arithmetic conversion is useful in the context of ring-LWE encryption. Here we use $k = 13$, since this corresponds to the binomial sampling for Saber, which can be written as a sum modulo 2^k of 1-bit Boolean to arithmetic modulo 2^k conversions, as in [SPO+19]. We see in Table 8.1 that our operation count is comparable to [SPO+19], both methods having complexity $\mathcal{O}(\alpha^2)$. Our operation count is an order of magnitude faster than [CGV14], which has complexity $\mathcal{O}(k \cdot \alpha^2)$. Namely the approach in [CGV14] requires to perform an arithmetic to Boolean conversion first, which has complexity $\mathcal{O}(k \cdot \alpha^2)$, so one cannot really take advantage of the 1-bit Boolean masking as input.

B $\to$ **A mod** 2^{13}	Security order t								
	1	2	3	4	5	6	8	10	12
Goubin [Gou01]	7								
[BCZ18]		49	123	277	591	1 225	5 053	20 401	81 829
[CGV14] $1 \to 13$		884	1 605	2 837	4 261	5 859	10 037	15 476	21 839
[SPO+19] $1 \to 13$		39	71	112	162	221	366	547	764
Algorithm 44, $1 \to 13$		52	101	166	247	344	586	892	1 262

Table 8.1: Operation count for 1-bit Boolean to arithmetic modulo 2^k conversion algorithms, up to security order $t = 12$, with $\alpha = t + 1$ shares, for $k = 13$.

1-bit Boolean to arithmetic modulo q conversion. We use $q = 3329$, as this corresponds to the encryption of Kyber, and to the binomial sampling of Kyber. For [BBE+18], we must use a word size k such that $2q < 2^k$, so we take $k = 13$. As previously, our complexity is comparable to [SPO+19], and more than an order of magnitude faster than [BBE+18].

B $\to$ **A mod** q	Security order t								
	1	2	3	4	5	6	8	10	12
[BBE+18] $1 \to \text{mod } q$	755	2 111	3 875	6 522	9 577	13 235	22 445	34 152	48 076
[SPO+19] $1 \to \text{mod } q$	16	39	71	112	162	221	366	547	764
Algorithm 44, $1 \to \text{mod } q$	19	52	101	166	247	344	586	892	1 262

Table 8.2: Operation count for 1-bit Boolean to arithmetic modulo q conversion algorithms, up to security order $t = 12$, with $\alpha = t + 1$ shares, for prime $q = 3329$.

8.1.3 Table-based high-order arithmetic to Boolean conversion

Direct approach for arithmetic modulo q

We consider the direct application of Algorithm 42 to high-order arithmetic to Boolean conversion. Given $x_1, \ldots, x_\alpha \in \mathbb{Z}_q$ as input, we obtain $y_1, \ldots, y_\alpha \in \{0,1\}^k$ as output, with $x_1 + \cdots + x_\alpha \bmod q = y_1 \oplus \cdots \oplus y_\alpha$. For this we have to assume that $q \leq 2^k$, since the sum $x_1 + \cdots + x_\alpha \bmod q$ needs at least $\lceil \log_2 q \rceil$ bits for its representation. We provide the pseudocode description below.

Algorithm 46 ArithmeticToBoolean

Input: $q \in \mathbb{Z}$ and $x_1, \ldots, x_\alpha \in \mathbb{Z}_q$
Output: $y_1, \ldots, y_\alpha \in \{0,1\}^k$ such that $y_1 \oplus \cdots \oplus y_\alpha = x_1 + \cdots + x_\alpha \bmod q$

1: **for all** $u \in \mathbb{Z}_q$ **do** $T(u) \leftarrow (u, 0, \ldots, 0)$
2: **for** $i = 1$ to $\alpha - 1$ **do**
3: **for all** $u \in \mathbb{Z}_q$ **do** $T'(u) \leftarrow T(u + x_i \bmod q)$
4: **for all** $u \in \mathbb{Z}_q$ **do** $T(u) \leftarrow \mathsf{Refresh}_{\{0,1\}^k}(T'(u))$
5: **end for**
6: $(y_1, \ldots, y_\alpha) \leftarrow \mathsf{Refresh}_{\{0,1\}^k}(T(x_\alpha))$
7: **return** $y_1, \ldots, y_\alpha$

Algorithm 47 $\mathsf{Refresh}_{\{0,1\}^k}$

Input: $x_1, \ldots, x_\alpha \in \{0,1\}^k$
Output: $y_1, \ldots, y_\alpha \in \{0,1\}^k$ such that $y_1 \oplus \cdots \oplus y_\alpha = x_1 \oplus \cdots \oplus x_\alpha$
1: $y_\alpha \leftarrow x_\alpha$
2: **for** $j = 1$ to $\alpha - 1$ **do**
3: $\quad r_j \xleftarrow{\$} \{0,1\}^k$
4: $\quad y_j \leftarrow x_j \oplus r_j$
5: $\quad y_\alpha \leftarrow y_\alpha \oplus r_j$
6: **end for**
7: **return** $y_1, \ldots, y_\alpha$

The operation count is the same as for Algorithm 42, with $|G| = q$, which gives:

$$\mathcal{C}_{\mathsf{AB}}(q, \alpha) = q \cdot (4\alpha^2 - 5\alpha + 2) + 3\alpha - 3 \simeq 4q \cdot \alpha^2$$

The memory complexity is $\mathcal{O}(q \cdot \alpha)$. The number of random elements is $(\alpha - 1) \cdot (q \cdot (\alpha - 1) + 1)$. The t-SNI security follows directly from Theorem 3.

Optimization with table in registers

We describe an optimization of Algorithm 46, where the j-th column of the table is stored in a single register R_j for $1 \leq j \leq \alpha$. The cyclic shift of the rows of the table by input share x_i then corresponds to a simple rotation of each register R_j. In the following we consider the arithmetic to Boolean conversion with k bits as input and 1 bit as output, as will be used in Section 8.1.4 for the IND-CPA decryption of lattice-based encryption.

More precisely we consider the computation of a function $f : \mathbb{Z}_{2^k} \to \{0,1\}$. One can consider for example the threshold function $f(x) = \lfloor x/2^{k-1} \rfloor$. Given as input α arithmetic shares $x_1, \ldots, x_\alpha \in \mathbb{Z}_{2^k}$, our goal is to compute 1-bit Boolean shares $y_1, \ldots, y_\alpha \in \{0,1\}$ such that $y_1 \oplus \cdots \oplus y_\alpha = f(x_1 + \cdots + x_\alpha \bmod 2^k)$.

Since we must store every column of the table with 2^k rows in a single register, each register must have 2^k bits. We denote by $R_j[u]$ the u-th bit of register R_j, for $0 \leq u < 2^k$ and $1 \leq j \leq \alpha$. Then Line 1 of Algorithm 46 becomes $R_1[u] = f(u)$ for $0 \leq u < 2^k$, and $R_j = 0$ for $2 \leq j \leq \alpha$. The rotation of the table at Line 3 becomes a rotation of all registers R_j by x_i positions to the right. The refreshing of the rows of the table at Line 4 becomes a mask refreshing of the shares $(R_1, \ldots, R_\alpha)$ with 2^k-bit random elements. Eventually we must read and refresh the row x_α of the table (Line 6 of Algorithm 46), so we simply read the x_α-th bit of each register R_j. We refer to Algorithm 48 for a formal description. We denote by $\mathsf{ROR}[a](R)$ the cyclic rotation of a 2^k-bit register R by a bits to the right.

Algorithm 48 ArithmeticToBoolean, register optimization (ABreg)

Input: $x_1, \ldots, x_\alpha \in \mathbb{Z}_{2^k}$
Output: $y_1, \ldots, y_\alpha \in \{0,1\}$ such that $y_1 \oplus \cdots \oplus y_\alpha = f(x_1 + \cdots + x_\alpha \bmod 2^k)$

1: **for all** $u \in \mathbb{Z}_{2^k}$ **do** $R_1[u] \leftarrow f(u)$
2: **for all** $2 \le j \le \alpha$ **do** $R_j \leftarrow 0$.
3: **for** $i = 1$ to $\alpha - 1$ **do**
4: **for** $j = 1$ to α **do** $R_j \leftarrow \mathsf{ROR}[x_i](R_j)$
5: **for** $j = 1$ to $\alpha - 1$ **do**
6: $r \xleftarrow{\$} \{0,1\}^{2^k}$, $R_j \leftarrow R_j \oplus r$, $R_\alpha \leftarrow R_\alpha \oplus r$
7: **end for**
8: **end for**
9: $(y_1, \ldots, y_\alpha) \leftarrow \mathsf{Refresh}_{\{0,1\}}(R_1[x_\alpha], \ldots, R_\alpha[x_\alpha])$
10: **return** $y_1, \ldots, y_\alpha$

Operation count. We do not count the 2^k operations of Line 1, since the value eventually stored in the register R_1 can be pre-computed. The number of operations is given by:

$$\begin{aligned}\mathcal{C}_{\mathsf{ABreg}}(\alpha) &= (\alpha-1)(\alpha+3(\alpha-1)) + \alpha + 3(\alpha-1)\\ &= 4\alpha^2 - 3\cdot\alpha \simeq 4\cdot\alpha^2\end{aligned}$$

Using 2^k-bit registers, the complexity of the countermeasure is therefore $\mathcal{O}(\alpha^2)$, assuming that generating a 2^k-bit random also takes unit time[1]. The memory complexity is $\alpha + 1$ registers of 2^k bits.

Obviously this optimization can only work for small values of k. In the comparison with existing techniques (sections 8.1.3 and 8.1.4), we use the following more realistic estimate of operation count, assuming a 32-bit processor. We assume that a register operation (or random generation) takes 1 operation for 32-bit ($k = 5$), and more generally 2^{k-5} operations for 2^k bits, for $k \ge 5$. The time complexity then becomes $\mathcal{C}'_{\mathsf{ABreg}}(\alpha, k) = 2^{k-5} \cdot (4\alpha^2 - 3\alpha)$. The number of 32-bit random elements is $2^{k-5} \cdot (\alpha-1)^2 + \alpha - 1$ for $k \ge 5$.

For $k = 5$, the implementation only requires $\alpha + 1$ registers of 32-bits. More generally, for $k \ge 5$, the memory complexity is $(\alpha + 1) \cdot 2^{k-5}$ registers of 32 bits.

Security. We prove below the $(\alpha - 1)$-SNI property of Algorithm 48. We stress that we do not put two shares from the same encoding into the same register. Otherwise the attacker could obtain information from multiple shares of the same encoding using a single probe on a given register, which would break the $(\alpha - 1)$-SNI property.

Theorem 4 ($(\alpha-1)$-SNI of ABreg)**.** *For any subset $O \subset [1, \alpha]$ and any t_1 intermediate variables with $|O| + t_1 < \alpha$, the output variables $y_{|O}$ and the t_1 intermediate variables can be perfectly simulated from the input variables $x_{|I}$, with $|I| \le t_1$.*

Proof. The proof is essentially the same as the proof of Theorem 3. The only difference is that by probing a register R_j, the adversary gets the full j-th column of the table, instead of a single cell only. Such probe is simulated in the same way by putting the index j in J for every such probe. □

Extensions. The technique is easily extended to arithmetic masking modulo any q as input, not only $q = 2^k$. In that case, one must perform two shifts for each register, instead of a single rotation for $q = 2^k$. Moreover, the technique is easily extended to k-bit Boolean masking as output, instead of 1-bit. In that case, one must use registers of size $k \cdot 2^k$ bits instead of 2^k.

[1] Obviously, one must be careful when expressing complexities with registers of exponential size. For example, Shamir described in [Sha79] an algorithm for factoring a k-bit RSA modulus in time $\mathcal{O}(k)$ only, but with exponentially large registers.

Comparison with existing techniques

Arithmetic modulo 2^k to 1-bit Boolean conversion, for small k. As we will see in Section 8.1.4, arithmetic modulo 2^k to 1-bit Boolean conversion is interesting in the context of ring-LWE IND-CPA decryption, in order to compute the threshold function $\mathsf{th} : \mathbb{Z}_{2^k} \to \{0,1\}$, with $\mathsf{th}(x) = 1$ if $x \in [2^{k-2}, 3 \cdot 2^{k-2}[$ and $\mathsf{th}(x) = 0$ otherwise.

Such threshold function th can be computed directly using our Algorithm 48 from the previous section, since the algorithm works for any function f. Alternatively, to compute th with [CGV14], we write $\mathsf{th}(x) = \mathsf{th}'(x - 2^{k-2})$ where $\mathsf{th}'(x) = 1$ if $x \in [0, 2^{k-1}[$ and 0 otherwise. Thus, $\mathsf{th}'(x)$ is the complement of the most significant bit of x. Therefore we first subtract 2^{k-2} to the first arithmetic share of x, and perform the arithmetic to Boolean conversion from [CGV14]. Finally we extract the most significant bit of each Boolean share, and complement the first share. We see in Table 8.3 that for $k = 6$, we obtain a significant improvement compared to [CGV14]. The value $k = 6$ is chosen such that our algorithm find an application to Saber.

A mod 2^6 → B	Security order t								
	1	2	3	4	5	6	8	10	12
Goubin [Gou01]	38								
[CGV14] $6 \to 1$		226	411	786	1 207	1 663	2 895	4 531	6 416
Algorithm 48	20	54	104	170	252	350	594	902	1 274

Table 8.3: Operation count for arithmetic modulo 2^k to 1-bit Boolean conversion algorithms, up to security order $t = 12$, with $\alpha = t + 1$ shares and $k = 6$.

8.1.4 Application to threshold function

In order to mask the ring-LWE IND-CPA decryption, we must compute a threshold function th over arithmetic shares modulo q, with 1-bit Boolean shares as output. Namely for Kyber we must compute the threshold function $\mathsf{th} : \mathbb{Z}_q \to \{0,1\}$ with

$$\mathsf{th}(x) = \begin{cases} 0 & \text{if } (x \bmod^{\pm} q) \in [-q/4, q/4[\\ 1 & \text{otherwise.} \end{cases} \tag{8.1}$$

More precisely, given as input arithmetic shares $x_1, \ldots, x_\alpha \in \mathbb{Z}_q$, we must compute 1-bit Boolean shares $y_1, \ldots, y_\alpha \in \{0,1\}$ such that

$$y_1 \oplus \cdots \oplus y_\alpha = \mathsf{th}(x_1 + \cdots + x_\alpha \bmod q)$$

The computation of the threshold function for Saber is similar.

For computing the threshold function th, we could apply our generic conversion algorithm from Section 8.1.1 with $G = \mathbb{Z}_q$, $H = \{0,1\}$ and the function $f : \mathbb{Z}_q \to \{0,1\}$ with $f = \mathsf{th}$. In that case, the time complexity is $\mathcal{O}(q \cdot \alpha^2)$, and the memory consumption is $\mathcal{O}(q \cdot \alpha)$. Both can be prohibitive for large q, for example with $q = 3329$ in Kyber, or with $q = 2^{10}$ in Saber.

We describe in the following an optimized technique based on modulus switching to a smaller modulus 2^ℓ, which enables to apply the fast table-based variant from Section 8.1.3. For this we slightly modify the decryption algorithms of Kyber, with a negligible increase in the decryption failure probability. We explain in the following why the security proof for Kyber remains perfectly valid. Namely the IND-CCA security proof only depends on the decryption failure probability, and not on the specific decryption algorithm used. In other words, one can use any decryption algorithm, as long as the decryption failure probability remains negligible. We maintain a total decryption failure probability $\delta \leq 2^{-128}$ to guarantee the same level of security as in the original scheme, against both classical attacks and quantum attacks.

Threshold arithmetic modulo q to 1-bit Boolean

Our goal is to compute the function $\mathsf{th} : \mathbb{Z}_q \to \{0,1\}$ given by (8.1) but this time we require a correct computation of $\mathsf{th}(x)$ only for a large subset of $\mathbb{Z}_q$, not necessarily for the full $\mathbb{Z}_q$. More precisely, we require correct computation only for values of x which are not too close to the thresholds $\pm q/4$, by a relative factor Δ. More precisely we require correct decryption for $x \in R_{q,1,\Delta}$ with:

$$R_{q,1,\Delta} = \left\{ x \in \mathbb{Z}_q,\ |x \bmod^{\pm} q| < q \cdot \left(\frac{1}{4} - \Delta\right) \text{ or } |x \bmod^{\pm} q| > q \cdot \left(\frac{1}{4} + \Delta\right) \right\} \tag{8.2}$$

This means that there will be a small subset of $\mathbb{Z}_q$ for which the computation of the function th can be incorrect. We will see that for lattice-based schemes such as Kyber and Saber, the probability that $x \notin R_{q,1,\Delta}$ is negligible for small enough Δ, and therefore the decryption error will remain negligible for these two schemes.

Our goal is therefore to compute output shares $b_1, \ldots, b_\alpha \in \{0,1\}$, such that when given input shares $x_1, \ldots, x_\alpha \in \mathbb{Z}_q$ such that $x = x_1 + \cdots + x_\alpha \in R_{q,1,\Delta}$, we are guaranteed to obtain a correct result, that is:

$$b_1 \oplus \cdots \oplus b_\alpha = \mathsf{th}(x_1 + \cdots + x_\alpha)$$

For this our strategy is to first perform a modulus switching into an arithmetic masking modulo a smaller 2^ℓ, and then to perform the (easier) conversion from arithmetic masking modulo 2^ℓ to Boolean masking via a threshold function f over $\mathbb{Z}_{2^\ell}$. More precisely, we first perform a modulus switching of all input shares x_i, by computing $y_i = \lfloor x_i \cdot 2^\ell / q \rceil$ for all $1 \le i \le \alpha$. Note that this modulus switching can be computed by writing

$$y_i = \left\lfloor \frac{x_i \cdot 2^\ell}{q} + \frac{1}{2} \right\rfloor = \left\lfloor \frac{x_i \cdot 2^{\ell+1} + q}{2q} \right\rfloor$$

and therefore y_i is the quotient of the Euclidean division of $x_i \cdot 2^{\ell+1} + q$ by $2q$. We obtain:

$$\sum_{i=1}^{\alpha} y_i = \sum_{i=1}^{\alpha} \left\lfloor \frac{2^\ell \cdot x_i}{q} \right\rceil = \sum_{i=1}^{\alpha} \frac{2^\ell \cdot x_i}{q} + \varepsilon_i$$

where $|\varepsilon_i| \le 1/2$ for all $1 \le i \le \alpha$. This gives:

$$y = \sum_{i=1}^{\alpha} y_i = \frac{2^\ell \cdot x}{q} + \varepsilon \mod 2^\ell, \quad \text{where } |\varepsilon| \le \alpha/2 \tag{8.3}$$

Therefore we have obtained an arithmetic masking of $y \in \mathbb{Z}_{2^\ell}$ where $y = 2^\ell \cdot x/q + \varepsilon \mod 2^\ell$ and the error $\varepsilon \in \mathbb{R}$ is such that $|\varepsilon| \le \alpha/2$. In the second step, we apply the Convert Algorithm 42 page 113 with $G = \mathbb{Z}_{2^\ell}$, $H = \{0,1\}$ and the function $f : \mathbb{Z}_{2^\ell} \to \{0,1\}$ where

$$f(y) = \begin{cases} 0 & \text{if } (y \bmod^{\pm} 2^\ell) \in (-2^{\ell-2}, 2^{\ell-2}) \\ 1 & \text{otherwise.} \end{cases}$$

Our algorithm is formally described in Algorithm 49 below.

Algorithm 49 Arithmetic modulo q to 1-bit Boolean conversion (ThresholdAtoB)

Input: $x_1, \ldots, x_\alpha \in \mathbb{Z}_q$
Output: $b_1, \ldots, b_\alpha \in \{0,1\}$ such that $b_1 \oplus \cdots \oplus b_\alpha = \mathsf{th}(x)$ for $x = x_1 + \cdots + x_\alpha \in R_{q,1,\Delta}$

1: **for** $i = 1$ to α **do** $y_i \leftarrow \lfloor x_i \cdot 2^\ell / q \rceil$
2: $(b_1, \ldots, b_\alpha) \leftarrow \mathsf{Convert}_{\mathbb{Z}_{2^\ell}, \{0,1\}, \mathsf{f}}(y_1, \ldots, y_\alpha)$
3: **return** $b_1, \ldots, b_\alpha$

Correctness and complexity. The following lemma proves the correctness of Algorithm 49 when the threshold function $\mathsf{th}(x)$ is computed on $x \in R_{q,1,\Delta}$, under the condition $\alpha \leq 2^{\ell+1} \cdot \Delta$.

Lemma 6. *Assume that $\alpha \leq 2^{\ell+1} \cdot \Delta$. The output of Algorithm 49 is correct if $x_1 + \cdots + x_\alpha \in R_{q,1,\Delta}$.*

Proof. Assume that $x \in R_{q,1,\Delta}$ and let represent x in $]-q/2, q/2[$. Assume that $|x| < q \cdot (1/4 - \Delta)$. This implies:

$$\left|\frac{2^\ell}{q} \cdot x\right| < 2^{\ell-2} - \Delta \cdot 2^\ell \leq 2^{\ell-2} - \frac{\alpha}{2}$$

From Equation (8.3), this implies that $|y \bmod^{\pm} 2^\ell| < 2^{\ell-2}$ and therefore $f(y) = \mathsf{th}(x) = 0$ as required.

Similarly, if $|x| \geq q \cdot (1/4 + \Delta)$, then $|x \cdot 2^\ell / q| > 2^{\ell-2} + \Delta \cdot 2^\ell \geq 2^{\ell-2} + \alpha/2$ and therefore $|y \bmod^{\pm} 2^\ell| > 2^{\ell-2}$, which implies $f(y) = \mathsf{th}(x) = 1$ as required. This proves the correctness of Algorithm 49. □

From Lemma 6, it suffices to select an intermediate modulus 2^ℓ with

$$\ell = \lceil \log_2(\alpha/\Delta) \rceil - 1 \tag{8.4}$$

to ensure correct computation of $\mathsf{th}(x)$ for $x \in R_{q,1,\Delta}$. The complexity of the arithmetic to Boolean conversion at Line 2 is therefore $\mathcal{O}(2^\ell \cdot \alpha^2) = \mathcal{O}(\alpha^3)$ using the generic conversion (Algorithm 46 page 117).

The memory complexity is $\mathcal{O}(\alpha)$ Finally, using the optimization with table in registers from Section 8.1.3, the complexity is $\mathcal{O}(\alpha^2)$ only, assuming that operations on registers of size 2^ℓ take unit time.

Security. The previous algorithm achieves the $(\alpha - 1)$-SNI property, thanks to the $(\alpha - 1)$-SNI property of the Convert algorithm.

Application to ring-LWE IND-CPA decryption

In this section we show how to efficiently mask the IND-CPA decryption of ring-LWE schemes. We explain how to tune the value Δ used in the definition of $R_{q,1,\Delta}$ in (8.2) so that the decryption error remains negligible for Kyber.

In order to mask the ring-LWE IND-CPA decryption, the secret-key $s \in \mathcal{R}$ is initially masked with α shares using $s = s_1 + \cdots + s_\alpha \mod q$ where $s_i \in \mathcal{R}_q$ for all $1 \leq i \leq \alpha$. Given as input a ciphertext (c_1, c_2), instead of computing $u = c_2 - s \cdot c_1$ and then $m = \mathsf{th}(u)$ coefficient-wise, in the first step we compute $u_1 = c_2 - s_1 \cdot c_1$ and $u_i = -s_i \cdot c_1$ for all $2 \leq i \leq \alpha$, which gives an arithmetic sharing of $u = u_1 + \cdots + u_\alpha \in \mathcal{R}_q$.

Therefore, in the second step, by applying Algorithm 49 coefficient-wise on the polynomial shares $u_i \in \mathcal{R}_q$, we obtain α boolean shares m_i of the message $m = m_1 \oplus \cdots \oplus m_\alpha$ such that $m_1 \oplus \cdots \oplus m_\alpha = \mathsf{th}(u_1 + \cdots + u_\alpha)$, as required. To ensure a negligible decryption error, we must therefore ensure that all coefficients of u belong to the set $R_{q,1,\Delta}$ considered in the previous section, except with negligible probability.

Application to Kyber. The authors of the Kyber submission provide a Python script computing a tight upper bound on the decryption error probability δ. Following [HHK17], we say that $\mathsf{PKE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is $(1 - \delta)$ correct if $\mathbb{E}[\max_{m \in \mathcal{M}} \Pr[\mathsf{Dec}(sk, \mathsf{Enc}(pk, m)) = m]] \geq 1 - \delta$, where the probability is over the randomness of Enc, and the expectation is over $(pk, sk) \leftarrow \mathsf{KeyGen}()$. More precisely, for an encryption of 0, the authors compute an upper-bound on the probability that any coefficient of $u = c_2 - \vec{s}^T \cdot \vec{c}_1$ is greater than $q/4$ in absolute value. From the definition of the set $R_{q,1,\Delta}$ in (8.2), it suffices to rerun the script with the bound $q \cdot (1/4 - \Delta)$ instead, in order to obtain the new decryption failure probability.

For our implementations, we choose to take $\Delta = 0.02$ for the recommended parameters of Kyber; this gives a decryption failure probability $\delta' = 2^{-137}$, instead of 2^{-164} originally (see Table 8.5). We argue in

Section 8.1.4 that Kyber remains secure with this increased decryption failure probability. We provide in Table 8.4 page 123 the value of the register size ℓ as a function of the number of shares α for $\Delta = 0.02$, according to Condition (8.4).

α	2	3	4	5	6	7	8	9	10
ℓ	6	7	7	7	8	8	8	8	8

Table 8.4: Value of ℓ as a function of α with $\Delta = 0.02$ for Kyber and Saber.

Moreover, we show in Table 8.5 that the decryption failure probability is easily decreased by modifying the compression parameters (d_u, d_v), which does not affect the security analysis of Kyber[2]. More precisely, by using the same compression parameters $(d_u, d_v) = (11, 5)$ as for Kyber1024, we obtain for $\Delta = 0.02$ a decryption error probability $\delta' = 2^{-192}$, that is smaller than originally in Kyber768.

	N	k	q	η_1	η_2	(d_u, d_v)	δ	δ'
Kyber768	256	3	3329	2	2	(10,4)	2^{-164}	2^{-137}
Kyber768'	256	3	3329	2	2	(11,5)	2^{-228}	2^{-192}

Table 8.5: Parameter set for Kyber, with the original failure probability δ, and the failure probability δ' for $\Delta = 0.020$.

Application to Saber. For Saber, the original decryption failure probability is 2^{-136}, and with $\Delta = 0.02$ the failure probability becomes 2^{-112}. We can reach $\delta' = 2^{-128}$ with $\Delta = 0.007$, but in that case there is no performance improvement. However, we can slightly modify the scheme parameters to reach a decryption failure probability $\delta' \leq 2^{-128}$, still with $\Delta = 0.02$. More precisely, we can increase the parameter $T = 2^4$ to $T = 2^6$ as in the more secure FireSaber[3]. This enables to reach a decryption failure probability $\delta' = 2^{-138}$ for $\Delta = 0.02$; see Table 8.6. In that case, we can use the same values as in Kyber for the register size ℓ as a function of the number of shares α (see Table 8.4).

	N	l	q	p	T	μ	δ	δ'
Saber	256	3	2^{13}	2^{10}	2^4	8	2^{-136}	2^{-112}
Saber'	256	3	2^{13}	2^{10}	2^6	8	2^{-164}	2^{-138}

Table 8.6: Parameter sets for Saber, with the original failure probability δ, and the failure probability δ' for $\Delta = 0.02$.

Security impact for ring-LWE IND-CCA encryption

In this section we consider the security impact of increasing the decryption failure probability δ. Namely, as illustrated in Table 8.5, for the Kyber768 parameters the decryption failure probability becomes $\delta' = 2^{-137}$ instead of $\delta = 2^{-164}$, so we must explain why the Kyber scheme remains secure. For this we follow closely the analysis from [ABD+21, Section 5.5].

[2]Namely, the classical and quantum core-SVP-hardness computed in the analysis script `Kyber.py` available at https://github.com/pq-crystals/kyber/tree/master/scripts/, do not depend on the compression parameters (d_u, d_v).

[3]As for Kyber, the security level computed by the Python script provided in the submission package, does not depend on the parameter T.

Classical security. We recall the CCA security of Kyber against classical adversaries, based on the Fujisaki-Okamoto transform, with a security bound that includes the decryption failure probability δ.

Theorem 5 (CCA security of Kyber [ABD+21])**.** *Suppose* XOF*, H, and G are random oracles. For any classical adversary $\mathcal{A}$ that makes at most q_{RO} many queries to random oracles* XOF*, H and G, there exist adversaries $\mathcal{B}$ and $\mathcal{C}$ of roughly the same running time as that of $\mathcal{A}$ such that* $\mathsf{Adv}^{\mathsf{cca}}_{\mathsf{Kyber.CCAKEM}} \leq 2\mathsf{Adv}^{\mathsf{mlwe}}_{k+1,k,\eta}(\mathcal{B}) + \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{PRF}}(\mathcal{C}) + 4q_{RO} \cdot \delta$

We note that the above security bound does not depend on the specific decryption algorithm used. This means that modifying the decryption algorithm (as we did in the previous section) does not invalidate the security proof of Kyber, as long as the decryption failure probability δ remains negligible. From the above security bound, with $\delta = 2^{-137}$, the best strategy to generate a decryption failure is to make $\simeq 2^{137}$ decryption or random oracle queries. This makes a classical attack completely unpractical.

Quantum security and failure boosting. In the quantum random oracle model, the security bound is non-tight and includes a term $q_{RO}^2 \cdot \delta$, see [ABD+21, Theorem 3]. Namely in the quantum setting the search for a m provoking a decryption failure can be quadratically accelerated using Grover's algorithm. In [ABD+21], the authors consider a failure boosting attack strategy that uses Grover's algorithm in an offline phase to search for a polynomial pair $(\vec{e}_1, \vec{r})$ with a larger norm, so that it is more likely to produce a decryption error. Below we use the same reasoning to estimate the quantum complexity of the attack, with decryption failure probability $\delta = 2^{-137}$ instead of 2^{-164}.

The polynomial pair $(\vec{e}_1, \vec{r})$ is seen as a vector in $\mathbb{Z}^{1536}$ distributed as a discrete Gaussian with standard deviation $\sigma = \sqrt{\eta_1/2} = 1$. We have that a m-dimensional vector $\vec{v}$ under such distribution satisfies for any $\kappa > 1$:

$$\Pr[\|\vec{v}\| > \kappa \cdot \sigma\sqrt{m}] < \kappa^m \cdot \exp(m(1-\kappa^2)/2)$$

Grover's algorithm is used to search this space with a quadratic speed-up, so with complexity $\kappa^{-m/2} \cdot \exp(m(\kappa^2-1)/4)$. In the second step, a decryption failure occurs if $\langle \vec{z}, \vec{v} \rangle$ is large enough for the secret vector $\vec{z}$. If $\vec{z}$ is distributed as a Gaussian with standard deviation σ', then for any λ, we have $\Pr[\langle \vec{z}, \vec{v} \rangle > \lambda\sigma'\|\vec{v}\|] \leq 2\exp(-\lambda^2/2)$. For a vector $\vec{v}$ without the failure boosting, we therefore have $\delta \simeq 2\exp(-\lambda^2/2)$, which gives $\lambda \simeq 13.8$ for $\delta = 2^{-137}$. Thanks to the failure boosting, we get a $\vec{v}$ whose norm is larger by a factor κ, so we can use $\lambda' = \lambda/\kappa$ instead of λ. The improved decryption failure probability after Grover's search then becomes $2\exp(-(\lambda/\kappa)^2/2)$, which gives a total complexity $\kappa^{-m/2} \cdot \exp(m(\kappa^2-1)/4 + (\lambda/\kappa)^2/2)$. For $\delta = 2^{-137}$, this is minimized for $\kappa = 1.1$, with total complexity 2^{124} (instead of 2^{150} for $\delta = 2^{-164}$). Therefore the attack remains completely unpractical. We refer to [ABD+21] for a discussion on the more recent attacks based on decryption failure [BS20; DRV20]; their overall running time for Kyber are no better than the above attack. In particular, the multi-target attack considered in [DGJ+19] is prevented in Kyber by hashing the public key pk into $\vec{r}$ and $\vec{e}_1$.

Comparison with existing techniques

We consider the computation of the threshold function th used in IND-CPA decryption for Saber and Kyber, and we provide a comparison of the operation count of our new technique (Algorithm 49 page 121) with existing techniques [BBE+18; BGR+21].

Threshold function mod 2^k based on the A2B conversion algorithm from [CGV14]. We show how to compute a threshold function $\mathsf{th} : \mathbb{Z}_{2^k} \to \{0,1\}$ where $\mathsf{th}(x) = 0$ if $x \in [0, 2^{k-1} - 1]$ and $\mathsf{th}(x) = 1$ otherwise, as used in Saber with $k = 10$. Note that $\mathsf{th}(x)$ is equal to the most significant bit of the k-bit representation of x. Starting from $x = x_1 + \cdots + x_\alpha \mod 2^k$, we perform an arithmetic to Boolean conversion of $(x_i)_{1 \leq i \leq \alpha} \in \mathbb{Z}_{2^k}$ into $(z_i)_{1 \leq i \leq \alpha} \in \{0,1\}^k$. We then let $b_i \in \{0,1\}$ be the most significant bit of z_i for $1 \leq i \leq \alpha$. We obtain $b_1 \oplus \cdots \oplus b_\alpha = \mathsf{th}(x_1 + \cdots + x_\alpha)$ as required. The operation count is therefore:

$$\mathcal{C}_{\mathsf{th}}(\alpha,k) = \mathcal{C}_{\mathsf{AB}}(\alpha,k) + \alpha$$

Threshold function modulo q, based on the A2B from [BBE+18]. We consider an integer q such that $q = 1 \mod 4$, as in Kyber with $q = 3329$. We show how to compute the threshold function $\mathsf{th} : \mathbb{Z}_q \to \{0,1\}$ where $\mathsf{th}(x) = 0$ if $x \in]-q/4, q/4[$ and $\mathsf{th}(x) = 1$ otherwise, where x is represented in $[-(q+1)/2, (q-1)/2]$. We first compute $y = x + (q-1)/4 \in \mathbb{Z}_q$ and we consider the function $\mathsf{th}'(y) = \mathsf{th}(x)$. We obtain $\mathsf{th}'(y) = 0$ if $y \in [0, (q-1)/2]$ and $\mathsf{th}'(y) = 1$ otherwise, where y is represented in $[0, q-1]$.

We consider k such that $q < 2^k$. We now consider y over $\mathbb{Z}$ with $0 \leq y < q < 2^k$. We let $z = y - (q+1)/2 \mod 2^k$. If $0 \leq y \leq (q-1)/2$, then $-2^{k-1} \leq -(q+1)/2 \leq y - (q+1)/2 < 0$. In this case we have $z = y - (q+1)/2 + 2^k$, which gives $2^{k-1} \leq z < 2^k$, and therefore the most significant bit of z is 1. Otherwise, if $(q+1)/2 \leq y < q$, then $0 \leq y - (q+1)/2 < (q-1)/2 \leq 2^{k-1}$, and therefore $z = y - (q+1)/2$, which gives $0 \leq z < 2^{k-1}$ and therefore the most significant bit of z is 0. Letting b be the most significant bit of z, we have $\mathsf{th}(x) = \neg b$.

Starting from $x = x_1 + \cdots + x_\alpha \mod q$, we compute $y_1 = x_1 + (q-1)/4 \in \mathbb{Z}_q$ and $y_i = x_i$ for $2 \leq i \leq \alpha$. We then perform an arithmetic modulo q to Boolean conversion of $(y_i)_{1\leq i\leq\alpha} \in \mathbb{Z}_q$ into $(u_i)_{1\leq i\leq\alpha} \in \{0,1\}^k$. We then perform a SecAdd between $(u_i)_{1\leq i\leq\alpha}$ and $(v_i)_{1\leq i\leq\alpha}$ with $v_1 = 2^k - (q+1)/2$ and $v_i = 0$ for $2 \leq i \leq \alpha$. We obtain $(z_i)_{1\leq i\leq\alpha}$ and we let b_i be the most significant bit of z_i for $1 \leq i \leq \alpha$. We let $b_1 \leftarrow \neg b_1$. Eventually we obtain $b_1 \oplus \cdots \oplus b_\alpha = \mathsf{th}(x_1 + \cdots + x_\alpha)$ as required. The operation count is therefore:

$$\mathcal{C}_{\mathsf{thModp}}(\alpha,k) = \mathcal{C}_{\mathsf{ABModp}}(\alpha,k) + \mathcal{C}_{\mathsf{SecAdd}}(\alpha,k) + \alpha + 2$$

The high-order threshold decryption technique from [BGR+21] for $q = 3329$ is based on computing:

$$\mathsf{Compress}_q^s(x) = x_{11} \oplus (\neg x_{11} \cdot x_{10} \cdot x_9 \cdot (x_8 \oplus (\neg x_8 \cdot x_7)))$$

where x_i is the i-th bit of x. The number of operations is therefore.

$$\mathcal{C}_{\mathsf{Comp1}}(\alpha) = \mathcal{C}_{\mathsf{ABModp}}(\alpha, 12) + 4 \cdot \mathcal{C}_{\mathsf{SecAnd}}(\alpha) + 2 \cdot \mathcal{C}_{\mathsf{refresh}}(\alpha) + 2\alpha$$

Comparison. For our Algorithm 49, we use the register optimization (Algorithm 48 page 119) to perform the arithmetic modulo 2^ℓ to 1-bit Boolean conversion, according to the values of ℓ from Table 8.4. As in Section 8.1.3, we assume that a register operation takes 1 operation for 32-bit ($\ell = 5$), and $2^{\ell-5}$ operations for 2^ℓ-bit, for $\ell \geq 5$.

We see in Table 8.7 that for Kyber, we obtain more than an order of magnitude improvement in IND-CPA decryption compared to [BBE+18] and [BGR+21].

	A mod $q \to$ 1-bit B	Security order t							
		1	2	3	4	5	6	8	9
Saber	[Gou01]	58							
	[CGV14]		366	667	1 286	1 979	2 731	4 767	6 051
	Algorithm 49 (Saber')	26	117	220	355	1 026	1 421	2 403	2 990
Kyber	[BBE+18]	511	1 515	2 792	4 783	7 047	9 765	16 642	20 801
	[BGR+21]	395	1 267	2 362	4 121	6 103	8 489	14 552	18 229
	Algorithm 49	26	117	220	355	1 026	1 421	2 403	2 990

Table 8.7: Operation count for arithmetic modulo q to 1-bit Boolean conversion algorithms, up to security order $t = 10$, with $\alpha = t + 1$ shares, for Kyber and Saber, with $q = 3329$ for Kyber and $q = 2^{10}$ for Saber. For Algorithm 49, we use the values of ℓ from Table 8.4 corresponding to $\Delta = 0.02$.

8.1.5 Application to binomial sampling

In this section, we show that our techniques enable to efficiently mask the re-encryption of ring-LWE encryption schemes. In the second step, under a simplified version of the FO transform for IND-CCA decryption, the message m is re-encrypted using error polynomials $(e_1, e_2, e_3) = H_1(m)$ to get a new ciphertext c'. To encode a Boolean masked message $m \in \{0,1\}$ like in (Equation 8.9 page 138), we can use our generic table-based conversion algorithm, with the function $f : \{0,1\} \to \mathbb{Z}_q$ with $f(x) = \lfloor q/2 \rceil \cdot x \bmod q$. In that case the complexity is $\mathcal{O}(\alpha^2)$ as in [SPO+19].

Consider a single error e, which we write $e = H(m)$ for some hash function H; for simplicity we focus on a single component $e \in \mathbb{Z}$. The error e is actually computed using binomial sampling, with $(\alpha, \beta) = H(m)$ and then $e = h(\alpha) - h(\beta)$, where $\alpha, \beta \in \{0,1\}^k$ and h is the Hamming weight function. The message m is Boolean masked, and therefore the variables α and β are Boolean masked, while the error e must be arithmetically masked modulo q.

To mask the binomial sampling we must therefore mask the Hamming weight computation, with Boolean masking as input and arithmetic masking modulo q as output. Our approach is similar to [SPO+19]: we start from our 1-bit Boolean to arithmetic masking modulo q algorithm from Section 8.1.2 (with $k = 1$), and for $\alpha \in \{0,1\}^k$, the Hamming weight of α is computed as the sum of k independent 1-bit Boolean to arithmetic masking modulo q conversions. Starting from a Boolean masked message $m = m_1 \oplus \cdots \oplus m_\alpha$, we then obtain an arithmetically masked ciphertext with α shares modulo q. Since our table-based approach has a similar level of efficiency as the technique from [SPO+19] (see Table 8.2 page 117 for a comparison), for the binomial sampling we obtain a similar level of efficiency as in [SPO+19], and an order of magnitude improvement compared to [BBE+18].

In the following, we describe in more details the technique to securely compute the Hamming weight and the binomial sampling, and we show how to perform masked IND-CPA encryption.

Masked Hamming weight computation

We consider the Hamming weight function $h : \{0,1\}^k \to \mathbb{Z}$ where $h(x)$ is the sum over $\mathbb{Z}$ of the bits of x, and the function $h_q : \{0,1\}^k \to \mathbb{Z}_q$ where this sum is computed modulo q, that is $h_q(x) = h(x) \bmod q$. Given as input $x_1, \ldots, x_\alpha \in \{0,1\}^k$, our goal is to compute arithmetic shares $a_1, \ldots, a_\alpha \in \mathbb{Z}_q$ such that:

$$a_1 + \cdots + a_\alpha = h_q(x_1 \oplus \cdots \oplus x_\alpha) \mod q$$

We let $x = x_1 \oplus \cdots \oplus x_\alpha$ and we write $x^{(j)}$ the j-th bit of x for $0 \le j < k$, which gives $h_q(x) = \sum_{j=0}^{k-1} x^{(j)}$. We also denote by $x_i^{(j)}$ the j-th bit of each share x_i. We obtain:

$$h_q(x) = \sum_{j=0}^{k-1} \bigoplus_{i=1}^{\alpha} x_i^{(j)} \bmod q$$

We now perform an independent table-based Boolean to arithmetic conversion for each of the k variables $x^{(j)}$, namely we write for each $0 \le j < k$:

$$x^{(j)} = \bigoplus_{i=1}^{\alpha} x_i^{(j)} = \sum_{i=1}^{\alpha} y_i^{(j)} \mod q$$

This gives:

$$h_q(x) = \sum_{j=0}^{k-1} \sum_{i=1}^{\alpha} y_i^{(j)} = \sum_{i=1}^{\alpha} \sum_{j=0}^{k-1} y_i^{(j)} \mod q$$

and therefore letting $a_i := \sum_{j=0}^{k-1} y_i^{(j)}$ for all $1 \le i \le \alpha$, we obtain $h_q(x_1 \oplus \cdots \oplus x_\alpha) = a_1 + \cdots + a_\alpha \mod q$ as required. The algorithm is formally described in Algorithm 50 below. The complexity of the algorithm is $\mathcal{O}(k \cdot \alpha^2)$.

The $(\alpha-1)$-SNI property follows from the $(\alpha-1)$-SNI of each of the k independent table-based conversions (Theorem 3 page 114). Namely the corresponding output shares $y_i^{(j)}$ are combined independently for each share index $1 \leq i \leq \alpha$.

Algorithm 50 Hamming weight

Input: $x_1, \ldots, x_\alpha \in \{0,1\}^k$
Output: $a_1, \ldots, a_\alpha \in \mathbb{Z}_q$ such that $a_1 + \cdots + a_\alpha = h(x_1 \oplus \cdots \oplus x_\alpha) \mod q$

1: **for** $i = 1$ to α **do** $a_i \leftarrow 0$
2: **for** $j = 0$ to $k-1$ **do**
3: **for** $i = 1$ to α **do** $z_i \leftarrow (x_i \gg j)\ \&\ 1$
4: $(y_1^{(j)}, \ldots, y_\alpha^{(j)}) \leftarrow \mathsf{BooleanToArithmetic}(1, z_1, \ldots, z_\alpha)$
5: **for** $i = 1$ to α **do** $a_i \leftarrow a_i + y_i^{(j)} \mod q$
6: **end for**
7: **return** $a_1, \ldots, a_\alpha$

Application to binomial sampling and masked IND-CPA encryption

We consider the high-order masking of ring-LWE IND-CPA encryption, using error polynomials $(e_1, e_2, e_3) = H_1(m)$ from a message $m \in R$ with binary coefficients:

$$c_1 = a \cdot e_1 + e_2$$
$$c_2 = t \cdot e_1 + e_3 + \lfloor q/2 \rceil \cdot m$$

We are given as input a Boolean masked message $m = m_1 \oplus \cdots \oplus m_\alpha \in R$ and we must output an arithmetically masked ciphertext modulo q. Applying our generic conversion algorithm with the function $f : \{0,1\} \to \mathbb{Z}_q$ with $f(x) = \lfloor q/2 \rceil \cdot x \bmod q$ on each coefficient separately, we obtain arithmetic shares $M_1, \ldots, M_\alpha \in R_{q,1}$ such that:

$$m \cdot \lfloor q/2 \rceil = \sum_{i=1}^{\alpha} M_i \mod q$$

Similarly, each component $e \in \mathbb{Z}$ of the error polynomials e_1, e_2, e_3 is equal to $e = h_q(\alpha) - h_q(\beta) \bmod q$, where $\alpha, \beta \in \{0,1\}^k$ and h_q is the Hamming weight function modulo q. Starting from α-shared Boolean masking of α and β, we can therefore apply Algorithm 50 to generate α arithmetic shares for e modulo q. Eventually, we obtain arithmetically masked error polynomials $E_{ji} \in \mathcal{R}_q$ for $j = 1, 2, 3$ such that

$$e_j = \sum_{i=1}^{\alpha} E_{ji} \mod q$$

Finally, we can compute the α shares of the ciphertext:

$$c_{1,i} = a \cdot E_{1,i} + E_{2,i}$$
$$c_{2,i} = t \cdot E_{1,i} + E_{3,i} + M_i$$

and we have $\sum_{i=1}^{\alpha} c_{1,i} = c_1 \mod q$ and $\sum_{i=1}^{\alpha} c_{2,i} = c_2 \mod q$ as required. Therefore we have obtained a masked ciphertext with α shares modulo q. The complexity is $\mathcal{O}(\alpha^2)$ for α shares.

8.2 High-order polynomial comparison

8.2.1 High-order zero testing

In the IND-CCA decryption of lattice-based schemes, according to the Fujisaki-Okamoto transform, we must perform a comparison between the input ciphertext $\tilde{c}$, and the re-encrypted ciphertext c. In the

context of the masking countermeasure, the re-encrypted ciphertext c is masked with α shares, so we must perform this comparison over arithmetic or Boolean shares. Moreover, the coefficients of the polynomials $\tilde{c}$ and c must be compared all at once. Otherwise the leaking of partial comparison results can leak information about the secret key, as demonstrated in [BDH+21].

In this section, for simplicity, we consider the zero-testing of a single coefficient. We will then show in Section 8.2.2 how to test multiple coefficients at once. With arithmetic shares, comparing two individual coefficients x and y in $\mathbb{Z}_q$ is equivalent to zero testing $x-y \in \mathbb{Z}_q$. Similarly, with Boolean shares, comparing two coefficients $x, y \in \{0,1\}^k$ is equivalent to zero testing $x \oplus y$. Therefore, in the rest of this section, we focus on zero-testing.

For a single coefficient x, we are therefore given as input the α Boolean shares of $x = x_1 \oplus \cdots \oplus x_\alpha \in \{0,1\}^k$, or the α arithmetic shares of $x = x_1 + \cdots + x_\alpha \bmod q$, and we must output a bit b, with $b = 1$ if $x = 0$ and $b = 0$ if $x \neq 0$, without revealing more information about x. This means that an adversary with at most $t = \alpha - 1$ probes will learn nothing about x, except if $x = 0$ or not.

From Boolean shares over $\{0,1\}^k$, one can perform a zero-test with complexity $\mathcal{O}(\alpha^2 \cdot \log k)$; we recall the technique in ZeroTestBoolLog Algorithm 52). From arithmetic shares modulo q, the simplest technique is to first perform an arithmetic to Boolean conversion, and then apply the zero-testing on the Boolean share (ZeroTestAB Algorithm 53). However in our context, we have an arithmetic sharing modulo a prime q. Hence, we describe a new zero-testing algorithm for such modulo: ZeroTestMult Algorithm 54. This technique is based on converting from arithmetic masking to multiplicative masking, so that one can distinguish between $x = 0$ and $x \neq 0$ without revealing more information about x. We will see in Section 8.2.2 that for zero testing ℓ coefficients at once, this technique is much more efficient than arithmetic to Boolean conversion. We refer to Table 8.8 for a summary.

	Technique	Masking	Complexity
ZeroTestBoolLog	Secure And	Boolean	$\mathcal{O}(\alpha^2 \cdot \log k)$
ZeroTestAB	A $\rightarrow$ B conversion	mod $q, 2^k$	$\mathcal{O}(\alpha^2 \cdot \log k)$
ZeroTestMult	Mult. masking	mod q	$\mathcal{O}(\alpha^2)$

Table 8.8: Complexities of zero testing a single value with α arithmetic shares, and a modulus 2^k or a k-bit prime q.

For the ciphertext comparison in Kyber, we will describe in Section 8.3.2 a hybrid approach in which the first part of the re-encrypted ciphertext is arithmetically masked modulo q, while the remaining part is Boolean masked. Therefore, we will use the ZeroTestBoolLog algorithm for the second part, and for the first part ZeroTestMult. For Saber, the re-encrypted ciphertext is completely Boolean shared, so we will use ZeroTestBoolLog. Finally, the ZeroTestAB algorithm will not be used in our constructions, but we keep this algorithm anyway for comparison with the ZeroTestMult algorithms.

Boolean zero testing in $\{0,1\}^k$

We first consider the zero-testing of $x \in \{0,1\}^k$ from its Boolean shares. We consider the k bits of $x = x^{(k-1)} \cdots x^{(0)}$. The zero-testing of x computes a bit b with $b = 1$ if $x = 0$, and $b = 0$ otherwise; therefore:

$$b = \overline{x^{(k-1)} \vee \cdots \vee x^{(0)}} = \overline{x^{(k-1)}} \wedge \cdots \wedge \overline{x^{(0)}}$$

Starting from the α Boolean shares of $x = x_1 \oplus \cdots \oplus x_\alpha$, the right-hand side of the above equation can be computed by a sequence of $k-1$ secure And. For simplicity we actually perform k iterations of SecAnd, the first one being a SecAnd with encoded input 1, to avoid an explicit mask refreshing at the beginning. The shares $b_1, \ldots, b_\alpha$ are eventually recombined after a mask refreshing. We obtain Algorithm 51 below.

Algorithm 51 ZeroTestBool

Input: $k \in \mathbb{N}$ and $x_1, \ldots, x_\alpha \in \{0,1\}^k$
Output: $b \in \{0,1\}$ such that $b = 1$ if $\oplus x_i = 0$, and $b = 0$ otherwise.
1: $(y_1, \ldots, y_\alpha) \leftarrow (\overline{x_1}, x_2, \ldots, x_\alpha)$
2: $(b_1, \ldots, b_\alpha) \leftarrow (1, 0, \ldots, 0)$
3: **for** $j = 0$ to $k-1$ **do**
4: $\quad (b_1, \ldots, b_\alpha) \leftarrow \mathsf{SecAnd}(1, (b_1, \ldots, b_\alpha), ((y_1 \gg j)\ \&\ 1, \ldots, (y_\alpha \gg j)\ \&\ 1))$
5: **end for**
6: $(b_1, \ldots, b_\alpha) \leftarrow \mathsf{RefreshMasks}(b_1, \ldots, b_\alpha)$
7: **return** $b_1 \oplus \cdots \oplus b_\alpha$

Theorem 6. *The* ZeroTestBool *is* $(\alpha-1)$-NI*, when* b *is given to the simulator.*

Proof. The ZeroTestBool algorithm up to Line 5 is $(n-1)$-SNI since it is the composition of SecAnd operations, which are $(n-1)$-SNI. Thanks to the final RefreshMasks, the ZeroTestBool algorithm up to Line 6 is $(n-1)$-SNI. This implies that the full ZeroTestBool is $(n-1)$-NI, when b is given to the simulator. □

Boolean zero-test with complexity $\mathcal{O}(\alpha^2 \cdot \log k)$

Let $x \in \{0,1\}^k$ and let $x = x_1 \oplus \cdots \oplus x_\alpha$ a Boolean sharing of x. We describe a procedure to zero-test x in $\mathcal{O}(\alpha^2 \cdot \log k)$ operations on k-bit registers, instead of $\mathcal{O}(\alpha^2 \cdot k)$ with the previous approach. The technique is as follows. We write

$$x = x^{(k-1)} \cdots x^{(0)}$$

the k bits of x. Let $m = \lceil \log_2 k \rceil$. If k is not a power of two, then we set the most significant bits of x to 1 until the next power of two, which is 2^m. Let $f_i(x) = x \wedge (x \gg 2^i)$. We prove below that we have:

$$x^{(k-1)} \wedge \cdots \wedge x^{(0)} = \mathsf{LSB}\left((f_{m-1} \circ \cdots \circ f_0)(x)\right) \tag{8.5}$$

Therefore to zero-test x, we can compute:

$$\overline{x^{(k-1)} \vee \cdots \vee x^{(0)}} = \mathsf{LSB}\left((f_{m-1} \circ \cdots \circ f_0)(\bar{x})\right)$$

We describe in Algorithm 52 below the high-order computation of the previous equation with α shares, using the SecAnd and RefreshMasks algorithms.

Algorithm 52 ZeroTestBoolLog

Input: $k \in \mathbb{Z}$ and $x_1, \ldots, x_\alpha \in \{0,1\}^k$
Output: $b \in \{0,1\}$ with $b = 1$ if $\oplus_{i=1}^{\alpha} x_i = 0$ and $b = 0$ otherwise
1: $m \leftarrow \lceil \log_2 k \rceil$
2: $y_1 \leftarrow \overline{x_1}$ or $(2^{2^m} - 2^k)$
3: **for** $i = 2$ to α **do** $y_i \leftarrow x_i$
4: **for** $i = 0$ to $m-1$ **do**
5: $\quad (z_1, \ldots, z_\alpha) \leftarrow \mathsf{RefreshMasks}(y_1 \gg 2^i, \ldots, y_\alpha \gg 2^i)$
6: $\quad (y_1, \ldots, y_\alpha) \leftarrow \mathsf{SecAnd}(m, (y_1, \ldots, y_\alpha), (z_1, \ldots, z_\alpha))$
7: **end for**
8: $(b_1, \ldots, b_\alpha) \leftarrow \mathsf{RefreshMasks}(y_1\ \&\ 1, \ldots, y_\alpha\ \&\ 1)$
9: **return** $b_1 \oplus \cdots \oplus b_\alpha$

Theorem 7 (Soundness)**.** *Given as input* $x_1, \ldots, x_\alpha \in \{0,1\}^k$*, the* ZeroTestBoolLog *algorithm outputs* $b = 1$ *if* $\oplus_{i=1}^{\alpha} x_i = 0$ *and* $b = 0$ *otherwise.*

Proof. We first consider the case where $\alpha = 1$, that is $x = x_1$ and $y = y_1 = \overline{x}$. For simplicity, we first assume that k is a power of two, that is $k = 2^m$. At Step 6 of the ZeroTestBoolLog algorithm, we compute $(f_{m-1} \circ \cdots \circ f_0)(y)$ where $f_i(y) = y \wedge (y \gg 2^i)$. To ease reading, we denote by $F_i(y)$ the value $(f_i \circ \cdots \circ f_0)(y)$. In the following we prove by induction on i for $i < m$ that the j-th bit of $F_i(y)$ is

$$(F_i(y))^{(j)} = y^{(j+2^{i+1}-1)} \wedge \cdots \wedge y^{(j)} \ , \tag{8.6}$$

for $j \leq 2^m - 2^{i+1}$.

For the base case $i = 0$, we have

$$(F_0(y))^{(j)} = (f_0(y))^{(j)} = ((y \gg 1) \wedge y)^{(j)} = (y \gg 1)^{(j)} \wedge y^{(j)} = y^{(j+1)} \wedge y^{(j)}$$

which satisfies the induction hypothesis. Now we show that

$$(F_{i+1}(y))^{(j)} = y^{(j+2^{i+2}-1)} \wedge \cdots \wedge y^{(j)} \ .$$

Indeed, we have

$$(F_{i+1}(y))^{(j)} = (f_{i+1}(F_i(y)))^{(j)} = (F_i(y) \wedge ((F_i(y) \gg 2^{i+1})))^{(j)} = (F_i(y))^{(j)} \wedge (F_i(y))^{(j+2^{i+1})} \ .$$

By using the induction hypothesis in Equation (8.6), we get

$$\begin{aligned}(F_{i+1}(y))^{(j)} &= (y^{(j+2^{i+1}-1)} \wedge \cdots \wedge y^{(j)}) \wedge (y^{((j+2^{i+1})+2^{i+1}-1)} \wedge \cdots \wedge y^{(j+2^{i+1})}) \\ &= y^{(j+2^{i+2}-1)} \wedge \cdots \wedge y^{(j)} \ ,\end{aligned}$$

which terminates the recursive proof.

In particular, for $i = m - 1$, we can use Equation (8.6) for $j \leq 2^m - 2^{i+1} = 0$. Thus, by keeping only the LSB part (that is $j = 0$) as done in Step 8 of Algorithm ZeroTestBoolLog, we have:

$$(F_{m-1}(y))^{(0)} = \mathsf{LSB}\left((f_{m-1} \circ \cdots \circ f_0)(y)\right) = y^{(2^m-1)} \wedge \cdots \wedge y^{(0)} = y^{(k-1)} \wedge \cdots \wedge y^{(0)} \ ,$$

as specified in equation (8.5). Note that this is also true in the case where k is not a power of two since in this case, the most significant bits of y are initially set to 1. From $y = \overline{x}$, we obtain as required:

$$\mathsf{LSB}\left((f_{m-1} \circ \cdots \circ f_0)(\overline{x})\right) = \overline{y^{(k-1)}} \vee \cdots \vee \overline{y^{(0)}} = \overline{x^{(k-1)} \vee \cdots \vee x^{(0)}} \ .$$

Eventually, the result also holds for $x = x_1 \oplus \cdots \oplus x_\alpha$ with $\alpha > 1$, since the same operations are performed on all shares, which proves the theorem. □

Complexity. We have used $\mathcal{C}_{\mathsf{refresh}}(\alpha) = 3\alpha(\alpha-1)/2$ and $T_{\mathsf{SecAnd}}(\alpha) = \alpha(7\alpha-5)/2$

$$\begin{aligned}\mathcal{C}_{\mathsf{ZeroTestBoolLog}}(k, \alpha) &= 2 + \lceil \log_2 k \rceil \cdot (\alpha + T_{\mathsf{refresh}}(\alpha) + \mathcal{C}_{\mathsf{SecAnd}}(\alpha)) + \alpha + T_{\mathsf{refresh}}(\alpha) + \alpha - 1 \\ &\simeq 5\alpha^2 \lceil \log_2 k \rceil\end{aligned}$$

Theorem 8. *The* ZeroTestBoolLog *algorithm is* $(\alpha-1)$-NI*, when* b *is given to the simulator.*

Proof. It is easy to see that the composition of steps 5 and 6 is $(\alpha-1)$-SNI secure, from the $(\alpha-1)$-SNI security of SecAnd and RefreshMasks. Therefore, the ZeroTestBoolLog algorithm up to Line 7 satisfies the $(\alpha-1)$-SNI property, and thanks to the last RefreshMasks, the full algorithm is $(\alpha-1)$-NI when b is given to the simulator.

□

In our context we only need the NI property for the zero tests or the polynomial zero tests. Indeed, the SNI property considers the outputs into the simulation. However, the comparison outputs are used in an unmasked way. Then, the NI property is enough.

Zero testing modulo q via arithmetic to Boolean conversion

We now consider the zero-testing of an element $x \in \mathbb{Z}_q$ from its arithmetic shares. Given as input the α arithmetic shares of $x = x_1 + \cdots + x_\alpha \bmod q$, we must output a bit b, with $b = 1$ if $x = 0$ and $b = 0$ if $x \neq 0$, without revealing more information about x. For $q \leq 2^k$, we first perform an arithmetic to Boolean conversion, which gives the Boolean shares $y_1, \ldots, y_\alpha \in \{0,1\}^k$, with $x = y_1 \oplus \cdots \oplus y_\alpha$. We then apply the Boolean zero-testing algorithm from the previous section. We obtain the pseudo-code below.

Algorithm 53 ZeroTestAB

Input: $q \in \mathbb{Z}$, $k \in \mathbb{Z}$ with $q \leq 2^k$, and $x_1, \ldots, x_\alpha \in \mathbb{Z}_q$
Output: $b \in \{0,1\}$ with $b = 1$ if $\sum_i x_i = 0 \mod q$ and $b = 0$ otherwise
1: $(y_1, \ldots, y_\alpha) \leftarrow \mathsf{ArithmeticToBoolean}(q, (x_1, \ldots, x_\alpha))$
2: **return** $\mathsf{ZeroTestBoolLog}(k, (y_1, \ldots, y_\alpha))$

The arithmetic to Boolean conversion step has complexity $\mathcal{O}(\alpha^2 \cdot k)$ for $q = 2^k$, using [CGV14] or the table recomputation approach from Section 8.1. We can also obtain an improved $\mathcal{O}(\alpha^2 \cdot \log k)$ complexity using the improved arithmetic to Boolean conversion from [CGT+15]. The technique actually works for arithmetic masking modulo any integer q, since we can use [BBE+18; SPO+19] to convert from arithmetic modulo q to Boolean masking, with complexity $\mathcal{O}(\alpha^2 \cdot \log \log q)$. In the second step, one can use the improved algorithm ZeroTestBoolLog with complexity $\mathcal{O}(\alpha^2 \cdot \log k)$. Therefore the overall complexity is $\mathcal{O}(\alpha^2 \cdot \log k)$, where $k = \lceil \log_2 q \rceil$, with a number of operations:

$$\mathcal{C}_{\mathsf{ZeroTestAB}}(k, \alpha) = \mathcal{C}_{\mathsf{AB}}(k, \alpha) + \mathcal{C}_{\mathsf{ZeroTestBoolLog}}(k, \alpha)$$

where $\mathcal{C}_{\mathsf{AB}}(k, \alpha)$ is the complexity of the arithmetic to Boolean conversion for a k-bit modulus q.

Theorem 9. *The* ZeroTestAB *algorithm is* $(\alpha - 1)$*-NI, when* b *is given to the simulator.*

Proof. The result follows from Theorem 6, with the ArithmeticToBoolean algorithm which is assumed to be $(\alpha - 1)$-NI. □

Zero testing modulo a prime q via multiplicative masking

Our new technique works for prime q only. It is based on converting from arithmetic masking modulo q to multiplicative masking. When the secret value x is 0, the multiplicatively masked value remains 0, whereas for $x \neq 0$, we obtain a random non-zero masked value. This enables to distinguish the two cases, without leaking more information about x.

More precisely, given as input the shares x_i of $x = x_1 + \cdots + x_\alpha \mod q$, we convert the arithmetic masking into a multiplicative masking. For this we generate a random $u_1 \in \mathbb{Z}_q^*$ and we compute:

$$u_1 \cdot x = u_1 \cdot x_1 + \cdots + u_1 \cdot x_\alpha \mod q$$

by computing the corresponding shares $x'_i = u_1 \cdot x_i \bmod q$ for all $1 \leq i \leq \alpha$. We then perform a linear mask refreshing of the arithmetic shares x'_i. Such linear mask refreshing is not SNI, but it is NI and its property is that any subset of $\alpha - 1$ output shares is uniformly and independently distributed, as in the mask refreshing from [RP10], see Algorithm 55.

We proceed similarly with the multiplicative shares $u_2, \ldots, u_\alpha \in \mathbb{Z}_q^*$. Eventually we obtain an arithmetic sharing $(B_i)_{1 \leq i \leq \alpha}$ satisfying:

$$u_1 \cdots u_\alpha \cdot x = B_1 + \cdots + B_\alpha \mod q$$

Thanks to the α multiplicative shares u_i, we can now safely decode the arithmetic sharing $(B_i)_{1 \leq i \leq \alpha}$ without revealing more information about x. More precisely, we compute $B = B_1 + \ldots + B_\alpha \mod q$, and we obtain:

$$u_1 \cdots u_\alpha \cdot x = B \mod q$$

Recall that $u_i \in \mathbb{Z}_q^*$ for all $1 \leq i \leq \alpha$. Therefore if $x \neq 0$, we must have $B \neq 0$, and if $x = 0$, we have $B = 0$. This gives a zero-test of x.

We provide below a pseudocode description of the ZeroTestMult algorithm taking as input the shares x_i of $x = x_1 + \cdots + x_\alpha \mod q$ and outputting a bit b with $b = 1$ if $x = 0$ and $b = 0$ otherwise.

Algorithm 54 ZeroTestMult

Input: $x_1, \ldots, x_\alpha \in \mathbb{Z}_q$ for prime q.
Output: $b \in \{0, 1\}$ with $b = 1$ if $\sum_i x_i = 0 \mod q$ and $b = 0$ otherwise

1: $(B_1, \ldots, B_\alpha) \leftarrow (x_1, \ldots, x_\alpha)$
2: **for** $j = 1$ to α **do**
3: $\quad u_j \xleftarrow{\$} \mathbb{Z}_q^*$
4: $\quad (B_1, \ldots, B_\alpha) \leftarrow (u_j \cdot B_1 \bmod q, \ldots, u_j \cdot B_\alpha \bmod q)$
5: $\quad (B_1, \ldots, B_\alpha) \leftarrow \mathsf{LinearRefreshMasks}(q, B_1, \ldots, B_\alpha)$
6: **end for**
7: $B \leftarrow B_1 + \cdots + B_\alpha \bmod q$
8: **if** $B = 0$ **then**
9: $\quad$ **return** 1
10: **else**
11: $\quad$ **return** 0
12: **end if**

Algorithm 55 LinearRefreshMasks

Input: $q \in \mathbb{Z}$ and $x_1, \ldots, x_\alpha \in \mathbb{Z}_q$
Output: $y_1, \ldots, y_\alpha \in \mathbb{Z}_q$ such that $y_1 + \cdots + y_\alpha = x_1 + \cdots + x_\alpha \mod q$

1: $y_\alpha \leftarrow x_\alpha$
2: **for** $j = 1$ to $\alpha - 1$ **do**
3: $\quad r_j \xleftarrow{\$} \mathbb{Z}_q$
4: $\quad y_j \leftarrow x_j + r_j \bmod q$
5: $\quad y_\alpha \leftarrow y_\alpha - r_j \bmod q$
6: **end for**
7: **return** $y_1, \ldots, y_\alpha$

Note that as opposed to the techniques described in the previous sections, we obtain an unmasked bit b. This means that when zero testing multiple coefficients at once, we can not keep an α-shared bit b and high-order combine the results of individual zero-testing, as with the previous ZeroTestBool and ZeroTestAB algorithms. Therefore, to test multiple coefficients at once, we will have to proceed differently (see Section 8.2.2).

Complexity. For simplicity we ignore the reductions modulo q in the operation count. The complexity of LinearRefreshMask is $3(\alpha - 1)$ operations. We obtain:

$$\mathcal{C}_{\mathsf{ZeroTestMult}}(\alpha) = \alpha \cdot (1 + \alpha + 3(\alpha - 1)) + \alpha = \alpha \cdot (4\alpha - 1) \simeq 4\alpha^2$$

The technique has therefore complexity $\mathcal{O}(\alpha^2)$ for a single coefficient. That is, as opposed to the previous techniques, the complexity is independent from the size of the modulus q, assuming that arithmetic operations in $\mathbb{Z}_q$ take unit time. We will see in Section 8.2.1 that for zero testing a single coefficient, the technique is much faster than the other techniques.

Theorem 10 (($\alpha-1$)-NI of ZeroTestMult)**.** *The* ZeroTestMult *takes as input α arithmetic shares x_i for $1 \leq i \leq \alpha$ and outputs a bit b with $b = 1$ if $\sum_{i=1}^{\alpha} x_i = 0 \mod q$ and $b = 0$ otherwise. Any t probes can be perfectly simulated from $x_{|I}$ and b, with $|I| \leq t$.*

Proof. We describe hereafter the construction of the set $I \subset [1, \alpha]$ of indices. Initially, I is empty. For every probed input variable x_i and for any probed intermediate variable B_i at Loop j between Steps 3 and 5, for $1 \leq i \leq \alpha$, we add index i to I. By construction of the set I, we have $|I| \leq t$ as required.

We now show that any t probes of Algorithm ZeroTestMult can be perfectly simulated from $x_{|I}$ and b. Since the number of probes t is such that $t < \alpha$, we deduce that at least one entire loop (Steps 3 to 5) has not been probed. Let $j^\star$ be the index of this non-probed loop. For all probed variables B_i between Steps 3 and 5 in loop indices $j < j^\star$, we have $i \in I$ and the simulation is straightforward from the input shares $x_{|I}$.

It remains to simulate all probed variables between Steps 3 and 5 in loop indices $j \geq j^\star$, and all probed variables at Step 7. To this aim, we consider two cases whether the output $b = 0$ or $b = 1$ (recall that b is given to the simulator).

If $b = 1$, then we know that $\sum_{i=1}^{\alpha} B_i = 0 \mod q$ at the end of each for loop. At the end of loop $j^\star$, since LinearRefreshMasks has not been probed, we can perfectly simulate all variables B_i, by generating random B_i's for $1 \leq i \leq \alpha$ such that $\sum_{i=1}^{\alpha} B_i = 0 \mod q$.

Similarly, if $b = 0$, we use the fact that $u_{j^\star}$ has not been probed and acts as a multiplicative one-time pad in $\mathbb{Z}_q^*$. This implies that the value encoded by the B_i's is randomly distributed in $\mathbb{Z}_q^*$. We can therefore perfectly simulate all shares B_i for $1 \leq i \leq \alpha$ at the end of loop $j^\star$ by generating random B_i's under the condition $\sum_{i=1}^{\alpha} B_i \neq 0 \mod q$.

In both cases, one can propagate the simulation until the end of the for loop, that is until $j = \alpha$, and from the knowledge of the B_i shares at the end of the for loop, one can compute all probed intermediate variables at Step 7 as in the real algorithm. We therefore conclude that the ZeroTestMult algorithm is ($\alpha-1$)-NI, when b is given to the simulator. □

Comparison of zero-test algorithms

We provide a comparison of the 2 zero-test algorithms that work modulo q, with $q = 3329$ as in Kyber. We see in Table 8.9 that for testing a single value, ZeroTestMult is more than one order of magnitude faster than ZeroTestAB .

zero-testing mod q	Security order t								
	1	2	3	4	5	6	8	10	12
ZeroTestAB	439	1 393	2 593	4 514	6 681	9 289	15 913	24 386	34 428
ZeroTestMult	14	33	60	95	138	189	315	473	663

Table 8.9: Operation count for zero testing with arithmetic masking modulo q, with $\alpha = t + 1$ shares and $q = 3329$.

8.2.2 High-order polynomial comparison

In this section we consider the zero-testing of multiple coefficients at once. For this we extend the zero-testing techniques from Section 8.2.1 to multiple coefficients.

For Boolean masked coefficients (PolyZeroTestBool) the extension is straightforward: we can simply keep the results of individual zero-testing in α-shared form (instead of recombining the shares), and high-order compute an iterated And between those results; only at the end do we recombine the shares to output a bit b. The approach is the same for zero testing multiple coefficients arithmetically masked modulo 2^k, when using arithmetic to Boolean conversion (PolyZeroTestAB). However, in the following we do not describe the algorithm PolyZeroTestAB. Indeed in our context of prime modulo, ZeroTestAB is much less efficient than ZeroTestMult therefore the polynomial version is also much less efficient.

When working modulo a prime q, it is very advantageous to first apply the technique from [BDH+21] that reduces the zero-testing of ℓ coefficients to the zero-testing of $\kappa \ll \ell$ coefficients, with $\kappa = \lceil \lambda / \log_2 q \rceil$, where λ is the security parameter, via random linear combinations. Namely the coefficients of the linear combinations can be computed without being masked, and the complexity of this first step is only $\mathcal{O}(\alpha)$ instead of $\mathcal{O}(\alpha^2)$.

The remaining κ coefficients must then be zero tested all at once. When the zero-testing is based on multiplicative masking (ZeroTestMult), we obtain the unmasked bit b of an individual zero-testing, so we must proceed differently. Before applying the zero-testing, we first compute random linear combinations as in [BDH+21], but this time the coefficients of the linear combination must be masked with α shares. For each linear combination, we perform a zero-test of the result. If all coefficients are 0, the linear combination will be 0, and the algorithm will return $b = 1$ as required. If at least one of the coefficients is non-zero, the linear combination will be non-zero and the algorithm will return $b = 0$, except with error probability $1/q$. As previously, by repeating the procedure κ times, we can decrease the error probability to $2^{-\lambda}$ with $\kappa = \lceil \lambda / \log_2 q \rceil$.

Polynomial comparison of Boolean masked coefficients

We are given as input a set of $\ell \cdot \alpha$ shares $(x^{(j)})_i \in \{0,1\}^k$ for $1 \le j \le \ell$ and $1 \le i \le \alpha$, corresponding to ℓ coefficients:

$$x^{(j)} = x_1^{(j)} \oplus \cdots \oplus x_\alpha^{(j)}$$

and we must output a single bit b such that $b = 1$ if $x^{(j)} = 0$ for all $1 \le j \le \ell$, and $b = 0$ otherwise. The simplest approach is to perform a Boolean zero-test of each $x^{(j)}$ as in Section 8.2.1, keeping each resulting bit $b^{(j)}$ in Boolean α-shared form, and then to perform a sequence of SecAnds between the bits $b^{(j)}$, and to eventually recombine the shares into a bit b. The complexity of this approach is then $\mathcal{O}(\ell \cdot \alpha^2 \cdot \log k)$. A slightly better approach is to high-order compute:

$$y = \overline{\bigwedge_{j=1}^{\ell} \overline{x^{(j)}}} \in \{0,1\}^k$$

Then $y = 0$ iff $x^{(j)} = 0$ for all $1 \le j \le \ell$, so we eventually perform a single zero-test of y. In this approach we take advantage of computing the SecAnds over k bits instead of a single bit. The complexity is then $\mathcal{O}(\ell \cdot \alpha^2 + \alpha^2 \cdot \log k)$. We obtain the pseudo-code below.

Algorithm 56 PolyZeroTestBool

Input: $k \in \mathbb{Z}$, and $(x_i^{(j)}) \in \{0,1\}^k$ for $1 \le i \le \alpha$ and $1 \le j \le \ell$.
Output: $b \in \{0,1\}$ with $b = 1$ if $\oplus_i x_i^{(j)} = 0$ for all $1 \le j \le \ell$, and $b = 0$ otherwise

1: **for** $j = 1$ to ℓ **do** $x_1^{(j)} \leftarrow \overline{x_1^{(j)}}$
2: $(y_1, \ldots, y_\alpha) \leftarrow (1, 0, \ldots, 0)$
3: **for** $j = 1$ to ℓ **do** $(y_1, \ldots, y_\alpha) \leftarrow \mathsf{SecAnd}(k, (y_1, \ldots, y_\alpha), (x_1, \ldots, x_\alpha))$
4: $y_1 \leftarrow \overline{y_1}$
5: **Return** $\mathsf{ZeroTestBoolLog}(k, y_1, \ldots, y_\alpha)$

The number of operations is:

$$\mathcal{C}_{\mathsf{PolyZeroTestBool}}(k, \ell, \alpha) = \ell \cdot (1 + \mathcal{C}_{\mathsf{SecAnd}}(\alpha)) + 1 + \mathcal{C}_{\mathsf{ZeroTestBoolLog}}(k, \alpha)$$

The following theorem shows that the adversary does not learn more than the output bit b of the comparison. The proof is obtained by composition of the SNI gadget SecAnd and the NI gadget ZeroTestBoolLog

Theorem 11. *The* PolyZeroTestBool *algorithm is* $(\alpha - 1)$*-NI, when* b *is given to the simulator.*

Polynomial comparison modulo prime q: reduction step

When working modulo a prime q, we can first apply the technique from [BDH+21] that efficiently reduces the zero-testing of ℓ coefficients to the zero-testing of $\kappa \ll \ell$ coefficients, with $\kappa = \lceil \lambda / \log_2 q \rceil$, where λ is the security parameter. Given as input ℓ coefficients $x^{(j)} \in \mathbb{Z}_q$ with arithmetic shares $x_i^{(j)}$, the technique consists in computing κ linear combinations:

$$y^{(k)} = \sum_{j=1}^{\ell} a_{kj} \cdot x^{(j)} \bmod q \tag{8.7}$$

for $1 \le k \le \kappa$, with randomly distributed coefficients $a_{kj} \in \mathbb{Z}_q$. The above equation is actually high-order computed using the arithmetic shares $x_i^{(j)}$ of each $x^{(j)}$, and we obtain the arithmetic shares $y_i^{(k)}$ of each coefficient $y^{(k)}$. We obtain the pseudo-code below.

Algorithm 57 PolyZeroTestRed [BDH+21]

Input: $q \in \mathbb{Z}$, a parameter κ and $(x_i^{(j)}) \in \mathbb{Z}_q$ for $1 \le i \le \alpha$ and $1 \le j \le \ell$.
Output: $(y_i^{(k)}) \in \mathbb{Z}_q$ for $1 \le i \le \alpha$ and $1 \le k \le \kappa$.

1: **for** $k = 1$ to κ **do**
2: **for** $i = 1$ to α **do** $y_i^{(k)} \leftarrow 0$
3: **for** $j = 1$ to ℓ **do**
4: $a_{kj} \xleftarrow{\$} \mathbb{Z}_q$
5: **for** $i = 1$ to α **do** $y_i^{(k)} \leftarrow y_i^{(k)} + a_{kj} \cdot x_i^{(j)}$
6: **end for**
7: **end for**
8: **return** $(y_i^{(k)})_{1 \le k \le \kappa,\, 1 \le i \le \alpha}$

Now if $x^{(j)} = 0$ for all $1 \le j \le \ell$, then $y^{(k)} = 0$ for all $1 \le k \le \kappa$. If $x^{(j)} \neq 0$ for some $1 \le j \le \ell$, then for each $1 \le k \le \kappa$, we have $y^{(k)} \neq 0$, except with probability $1/q$. Therefore we must have $y^{(k)} \neq 0$ for some $1 \le k \le \kappa$, except with error probability $q^{-\kappa}$. We have therefore reduced the zero-testing of ℓ coefficients to the zero-testing of $\kappa \ll \ell$ coefficients. To reach an error probability $\le 2^{-\lambda}$ for security parameter λ, one must take $\kappa = \lceil \lambda / \log_2 q \rceil$.

We stress that after this reduction step we cannot zero-test the coefficients $y^{(k)}$ separately. Otherwise, since the coefficients a_{kj} in Equation (8.7) are computed without mask, knowing that $y^{(k)} = 0$ for some k would leak an equation over the coefficients $x^{(j)}$, which would leak information about the $x^{(j)}$ with fewer than α probes. Instead, the remaining κ coefficients $y^{(k)}$ must be zero-tested all at once. For this we describe in the next sections two efficient techniques.

This reduction technique is quite efficient because the random coefficients a_{kj} in (8.7) are non-masked, which implies that each multiplication $a_{kj} \cdot x^{(j)}$ can be computed in time $\mathcal{O}(\alpha)$ for α shares, instead of $\mathcal{O}(\alpha^2)$ for a fully masked multiplication. The total complexity of this first step is therefore $\mathcal{O}(\ell \cdot \kappa \cdot \alpha)$, with a number of operations:

$$T_{\mathsf{PolyZeroTestRed}}(\kappa, \ell, \alpha) = \kappa \cdot \ell \cdot (2\alpha + 1)$$

Theorem 12 ([BDH+21]). *The* PolyZeroTestRed *algorithm is* $(\alpha - 1)$*-NI.*

Polynomial comparison modulo q via multiplicative masking

As explained previously, when zero testing a value x modulo q using the multiplicative masking technique (Section 8.2.1), we obtain the unmasked resulting bit b, so we cannot zero-test the coefficients iteratively as in the previous techniques. Instead, we first compute a random linear combination of the individual

coefficients modulo q, and we then perform a zero-test of the result. This approach is similar to [BDH+21], except that we must compute the coefficients $a^{(j)}$ in the linear combination in α-shared form, as otherwise this can leak information on the coefficients $x^{(j)}$ and then leads to a CCA attack.

As previously, we consider as input an arithmetic masking of ℓ coefficients $x^{(j)}$, that is $x^{(j)} = x_1^{(j)} + \cdots + x_\alpha^{(j)} \mod q$ for all $1 \leq j \leq \ell$. We first apply the reduction algorithm PolyZeroTestRed described previously. In the second step, we must therefore zero-test the set of coefficients $y^{(j)}$ with arithmetic shares $y_i^{(j)}$ modulo q, that is $y^{(j)} = y_1^{(j)} + \cdots + y_\alpha^{(j)} \mod q$ for all $1 \leq j \leq \kappa$.

For this, we generate random coefficients $a^{(j)} \in \mathbb{Z}_q$, and we high-order compute the linear combination:

$$z = \sum_{j=1}^{\kappa} a^{(j)} \cdot y^{(j)} \bmod q \tag{8.8}$$

If $y^{(j)} = 0$ for all $1 \leq j \leq \kappa$, then $z = 0$. If $y^{(j)} \neq 0$ for some $1 \leq j \leq \kappa$, then we have $z \neq 0$, except with probability $1/q$. We can therefore perform a zero-test of z. The procedure can be repeated a small number of times to have a negligible probability of error. Namely, for κ repetitions with randomly generated $a^{(j)}$, the error probability becomes $q^{-\kappa}$.

Equation (8.8) is high-order computed using the arithmetic shares $y_i^{(j)}$ of the coefficients $y^{(j)}$. Similarly the random coefficients $a^{(j)}$ are generated via α random shares $a_i^{(j)}$ in $\mathbb{Z}_q$. This is the main difference with the linear combination used in Section 8.2.2 for the reduction step, in which the coefficients a_{kj} in (8.7) were computed without mask. We stress that this time, the coefficients $a^{(j)}$ must be computed in α-shared form, and the multiplication $a^{(j)} \cdot y^{(j)}$ computed with SecMult, since otherwise an equation over the $x^{(j)}$ could be leaked with fewer than α probes. From the high-order computation of (8.8), we obtain the α shares z_i of the linear combination z. We then apply the zero-test procedure from Section 8.2.1 on the shares z_i, which outputs a bit b such that $b = 1$ if $z = 0$ and $b = 0$ otherwise. The procedure is repeated κ times, and if we always obtain $b = 1$ from the zero-test, we output 1, otherwise we output 0.

Algorithm 58 PolyZeroTestMult

Input: $q \in \mathbb{Z}$, a parameter κ and $(x_i^{(j)}) \in \mathbb{Z}_q$ for $1 \leq i \leq \alpha$ and $1 \leq j \leq \ell$.
Output: $b \in \{0,1\}$ with $b = 1$ if $\sum_i x_i^{(j)} = 0 \mod q$ for all $1 \leq j \leq \ell$ and $b = 0$ otherwise

1: $(y_i^{(k)})_{1 \leq k \leq \kappa,\, 1 \leq i \leq \alpha} \leftarrow$ PolyZeroTestRed$((x_i^{(j)})_{1 \leq j \leq \ell,\, 1 \leq i \leq \alpha})$
2: $b \leftarrow 0$
3: **for** $k = 1$ to κ **do**
4: **for** $i = 1$ to α **do** $z_i \leftarrow 0$
5: **for** $j = 1$ to ℓ **do**
6: **for** $i = 1$ to α **do** $a_i^{(j)} \xleftarrow{\$} \mathbb{Z}_q$
7: $(z_1^{(j)}, \ldots, z_\alpha^{(j)}) \leftarrow$ SecMult$((a_1^{(j)}, \ldots, a_\alpha^{(j)}), (y_1^{(j)}, \ldots, y_\alpha^{(j)}))$
8: **for** $i = 1$ to α **do** $z_i \leftarrow z_i + z_i^{(j)} \bmod q$
9: **end for**
10: $b_k \leftarrow$ ZeroTestMult$(z_1, \ldots, z_\alpha)$
11: $b \leftarrow b + b_k$
12: **end for**
13: **if** $b = \kappa$ **then** $b \leftarrow 1$ **else** $b \leftarrow 0$
14: **return** b

The complexity of the PolyZeroTestMult algorithm is

$$\begin{aligned}\mathcal{C}_{\mathsf{PolyZeroTestMult}}(q,\kappa,\ell,\alpha) = \; & \mathcal{C}_{\mathsf{PolyZeroTestRed}}(\kappa,\ell,\alpha)+ \\ & \kappa\cdot(\kappa\cdot(\mathcal{C}_{\mathsf{SecMult}}(\alpha)+2\alpha)+\mathcal{C}_{\mathsf{ZeroTestMult}}(\alpha)))\end{aligned}$$

For security level λ, the error probability must satisfy $q^{-\kappa} \leq 2^{-\lambda}$, so we can take $\kappa = \lceil \lambda / \log_2 q \rceil$ repetitions. Therefore, the complexity of the second step is $\mathcal{O}(\kappa^2\alpha^2)$. The total complexity is therefore $\mathcal{O}(\kappa\ell\alpha + \kappa^2\alpha^2)$.

Theorem 13 (Soundness)**.** *The* PolyZeroTestMult *outputs the correct answer, except with probability at most* $q^{-\kappa}$*.*

Proof. We denote by PolyZeroTestMultLoop, one loop iteration on k of the PolyZeroTestMult algorithm, namely, going from line 4 to 11. We start by showing that PolyZeroTestMultLoop computes the correct answer b_k, except with probability at most $1/q$. Indeed, in PolyZeroTestMultLoop, one securely computes the value $z = \sum_{j=1}^{\ell} a^{(j)} \cdot y^{(j)} \mod q$ where the random values $a^{(j)}$ are uniformly distributed in $\mathbb{Z}_q$. Thus, if $z \neq 0$, then at least one coefficient $y^{(j)}$ is not zero and the output $b_k = 0$ is always correct.

However, if $z = 0$, two cases arise: either all coefficients $y^{(j)}$ are null in which case the algorithm outputs $b_k = 1$ which is correct, or at least one coefficient $y^{(j)}$ is such that $y^{(j)} \neq 0$ but with $\sum_{j=1}^{\ell} a^{(j)} \cdot y^{(j)} = 0 \mod q$ and the output $b_k = 1$ in this case is incorrect. Since the $a^{(j)}$ values are uniformly distributed in $\mathbb{Z}_q$, the result of the linear combination of the $y^{(j)} \neq 0$ with the values $a^{(j)}$ is also uniform in $\mathbb{Z}_q$. Therefore the probability that $\sum_{j=1}^{\ell} a^{(j)} \cdot y^{(j)} = 0 \mod q$ is $1/q$ for each iteration of PolyZeroTestMultLoop.

Hence by iterating PolyZeroTestMultLoop κ times with fresh random values $a_\kappa^{(j)}$ as done in PolyZeroTestMult, the probability that $\sum_{j=1}^{\kappa} a_\kappa^{(j)} \cdot y^{(j)} \mod q = 0$ for all κ iterations, with at least one coefficient $y^{(j)} \neq 0$, is $(1/q)^\kappa = q^{-\kappa}$. □

Theorem 14. *The* PolyZeroTestMult *algorithm is* $(\alpha-1)$*-NI, when* b *is given to the simulator.*

Proof. As before, we denote by PolyZeroTestMultLoop, one loop iteration on k of the PolyZeroTestMult algorithm (line 4 to 11). We write $y^{(j)} = \sum_{i=1}^{\alpha} y_i^{(j)} \mod q$. We distinguish two cases: either $y^{(j)} = 0$ for all $1 \leq j \leq \ell$, or $y^{(j)} \neq 0$ for some j. We show that the simulator can perform a perfect simulation in both cases. Moreover, by assumption the simulator eventually receives the bit b. This means that the simulator can distinguish the two cases, except with error probability at most $q^{-\kappa}$. Therefore the error probability of the simulator will be at most $q^{-\kappa}$.

$y^{(j)} = 0$ for all $1 \leq j \leq \ell$. This is the easy case. Namely in that case, we know that $b_k = 1$ for all k. The computation of the shares z_i at Line 8 is $(\alpha-1)$-SNI. Knowing b_k, the algorithm ZeroTestMult at Step 10 is $(\alpha-1)$-NI from Theorem 10. Therefore the global PolyZeroTestMultLoop algorithm remains $(\alpha-1)$-NI.

$y^{(j)} \neq 0$ for some $1 \leq j \leq \ell$. We consider a sequence of games.

Game_0: we generate all variables as in the algorithm. We assume that we know all input shares $y_i^{(j)}$. We can therefore perform a perfect simulation of all probes. Moreover, we have that $\Pr[b_k = 1] = 1/q$ for all $1 \leq k \leq \kappa$, and the variables b_k are independently distributed.

Game_1: we modify the way the variables are generated. Instead of generating all variables $a_i^{(j)}$ uniformly and independently, we first generate the bits b_k independently with $\Pr[b_k = 1] = 1/q$. Then for each $1 \leq k \leq \kappa$, if $b_k = 1$ then we generate the shares $a_i^{(j)}$ such that $\sum_{j=1}^{\ell} a^{(j)} y^{(j)} = 0 \mod q$, where $a^{(j)} = \sum_{i=1}^{\alpha} a_i^{(j)} \mod q$. Otherwise, we generate the shares $a_i^{(j)}$ such that $\sum_{j=1}^{\ell} a^{(j)} y^{(j)} \neq 0 \mod q$. The distribution of the variables is the same as in the previous game. Therefore, we can still perform a perfect simulation of all probed variables.

Game_2: we show that we can still perform a perfect simulation as in Game_1, but only with the input shares $y^{(j)}_{|I}$ for a subset $|I| \leq t$. This will prove that the algorithm is $(\alpha-1)$-NI.

Firstly, from the $(\alpha-1)$-SNI property of SecMult and the $(\alpha-1)$-NI property of ZeroTestMult knowing b_k, the simulation of all probes can be performed from the knowledge of a subset $y^{(j)}_{|I}$ of the input shares for $|I| \leq t$, and a subset $a^{(j)}_{|J}$ of the shares of the values $a^{(j)}$, for $|J| \leq t \leq \alpha-1$. Secondly, the constraints on $\sum_{j=1}^{\ell} a^{(j)}y^{(j)}$ from Game_2 can be satisfied by generating all shares $a^{(j)}_i$ for $i \neq i^\star$ uniformly at random, and by fixing $a^{(j)}_{i^\star}$, without changing the distribution of the shares $a^{(j)}_i$, for some $i^\star \notin J$. Finally, since the knowledge of $a^{(j)}_{i^\star}$ is not needed for the simulation, we can perform a perfect simulation of all probes from $y^{(j)}_{|I}$. This concludes the proof. □

8.3 Fully masked implementation of Kyber

Kyber is a lattice-based encryption scheme and a finalist of the third round of the NIST competition [BDK+18; ABD+21]. Its security is based on the hardness of the module learning-with-errors (M-LWE) problem. The IND-CCA secure key establishment mechanism (KEM) is obtained by applying the Fujisaki-Okamoto transform [FO99; HHK17]. The Kyber submission provides three parameters sets Kyber512, Kyber768 and Kyber1024, with claimed security level equivalent to AES-128, AES-192 and AES-256 respectively. The three parameter sets share the common parameters $N = 256$, $q = 3329$ and $\eta_2 = 2$, while the security level is defined by setting the module rank $k = 2, 3, 4$, and the parameters η_1, d_t, d_u and d_v (see Table 8.10).

In the following, we start with an overview of ring-LWE encryption [LPR10], and then recall the definition of the Kyber scheme. We then describe the evaluation of the Kyber decapsulation mechanism, secure at any order, using the techniques from the previous sections.

8.3.1 The Kyber Key Encapsulation Mechanism (KEM)

Ring-LWE IND-CPA encryption. Let $\mathcal{R}$ and $\mathcal{R}_q$ denote the rings $\mathbb{Z}[X]/(X^N+1)$ and $\mathbb{Z}_q[X]/(X^N+1)$ respectively, for some $N \in \mathbb{N}$ and an integer q. Let $a \in R_{q,1}$ be a public random polynomial. Let χ be a distribution outputting "small" elements in $\mathcal{R}$, and let $s, e \leftarrow \chi$. The public key is $t = as + e \in \mathcal{R}_q$, while the secret key is s. To CPA-encrypt a message $m \in \mathcal{R}$ with binary coefficients, one computes the ciphertext (c_1, c_2) where

$$\begin{aligned} c_1 &= a \cdot e_1 + e_2 \\ c_2 &= t \cdot e_1 + e_3 + \lfloor q/2 \rceil \cdot m \end{aligned} \tag{8.9}$$

with $e_1, e_2, e_3 \leftarrow \chi$. To decrypt a ciphertext (c_1, c_2), one first computes $u = c_2 - s \cdot c_1$, which gives:

$$\begin{aligned} u &= (a \cdot s + e) \cdot e_1 + e_3 + \lfloor q/2 \rceil \cdot m - s \cdot a \cdot e_1 - s \cdot e_2 \\ &= \lfloor q/2 \rceil \cdot m + e \cdot e_1 + e_3 - s \cdot e_2 \end{aligned}$$

Since the ring elements e, e_1, e_2, e_3 and s are small, and the message $m \in \mathcal{R}$ has binary coefficients, we can recover m by rounding. Namely, for each coefficient of the above polynomial u, we decode to 0 if the coefficient is closer to 0 than $\lfloor q/2 \rceil$, and to 1 otherwise. More precisely, we decode the message m as $m = \mathsf{th}(c_2 - s \cdot c_1)$, where th applies coefficient-wise the threshold function $\mathsf{th} : \mathbb{Z}_q \to \{0, 1\}$:

$$\mathsf{th}(x) = \begin{cases} 0 & \text{if } x \in [0, q/4] \cup]3q/4, q[\\ 1 & \text{if } x \in]q/4, 3q/4] \end{cases} \tag{8.10}$$

The Kyber IND-CPA encryption. The Kyber scheme is based on the module learning-with-errors problem (M-LWE) in module lattices [LS15b]. For a modulo rank k, we use a public random $k \times k$ matrix $\mathbf{A}$ with elements in $\mathcal{R}_q$. We set χ_η as the centered binomial distribution with support $\{-\eta, \ldots, \eta\}$, and extended to the distribution of polynomials of degree N with entries independently sampled from χ_η. The public key is $\mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e} \in \mathcal{R}_q^k$ and the secret key is $\mathbf{s}$, where $\mathbf{s}, \mathbf{e} \leftarrow \chi_{\eta_1}^k$ for some parameter η_1. To CPA-encrypt a message $m \in \mathcal{R}$ with binary coefficients, one computes $(\mathbf{c}_1, c_2) \in \mathcal{R}_q^k \times \mathcal{R}_q$ such that $\mathbf{c}_1 = \mathbf{A}^T \cdot \mathbf{r} + \mathbf{e}_1$ and $c_2 = \mathbf{t}^T \cdot \mathbf{r} + e_2 + \lfloor q/2 \rceil \cdot m$, where $\mathbf{r} \leftarrow \chi_{\eta_1}^k$, $\mathbf{e}_1 \leftarrow \chi_{\eta_2}^k$ and $e_2 \leftarrow \chi_{\eta_2}$, for some parameter η_2. To decrypt a ciphertext $(\mathbf{c}_1, c_2)$, one computes as previously:

$$u = c_2 - \mathbf{s}^T \cdot \mathbf{c}_1 = \mathbf{e}^T \cdot \mathbf{r} + e_2 - \mathbf{s}^T \cdot \mathbf{e}_1 + \lfloor q/2 \rceil \cdot m \approx \lfloor q/2 \rceil \cdot m$$

Kyber instantiates the M-LWE-based encryption scheme above with $N = 256$ and a prime $q = 3329$; see Table 8.10 for the other parameters. We recall the pseudo-code from [BDK+18] below. For simplicity we omit the NTT transform for fast polynomial multiplication. The NTT is indeed a linear operation, so it is easily masked with arithmetic masking modulo q.

Algorithm 59 Kyber.CPA.KeyGen()

1: $\rho, \sigma \xleftarrow{\$} \{0,1\}^{256}$
2: $\mathbf{A} \leftarrow \mathcal{R}_q^{k \times k} := \mathsf{Sam}(\rho)$
3: $\mathbf{s}, \mathbf{e} \leftarrow \chi_{\eta_1}^k \times \chi_{\eta_1}^k := \mathsf{Sam}(\sigma)$
4: $\mathbf{t} := \mathsf{Compress}_{q,d_t}(\mathbf{A}\mathbf{s} + \mathbf{e})$
5: **return** $pk := (\mathbf{t}, \rho)$, $sk := \mathbf{s}$

Algorithm 60 Kyber.CPA.Dec($sk, c = (\vec{u}, v)$)

1: $\vec{u} := \mathsf{Decompress}_{q,d_u}(\vec{u})$
2: $v := \mathsf{Decompress}_{q,d_v}(v)$
3: **return** $\mathsf{Compress}_{q,1}(v - \vec{s}^T \vec{u})$

Algorithm 61 Kyber.CPA.Enc(pk, m)

1: $r \xleftarrow{\$} \{0,1\}^{256}$
2: $\mathbf{t} := \mathsf{Decompress}_{q,d_t}(\mathbf{t})$
3: $\mathbf{A} \leftarrow \mathcal{R}_q^{k \times k} := \mathsf{Sam}(\rho)$
4: $\mathbf{r}, \mathbf{e}_1, e_2 \leftarrow \chi_{\eta_1}^k \times \chi_{\eta_2}^k \times \chi_{\eta_2} := \mathsf{Sam}(r)$
5: $\vec{u} := \mathsf{Compress}_{q,d_u}(\mathbf{A}^T \mathbf{r} + \mathbf{e}_1)$
6: $v := \mathsf{Compress}_{q,d_v}(\mathbf{t}^T \mathbf{r} + e_2 + \lfloor q/2 \rceil \cdot m)$
7: **return** $c := (\vec{u}, v)$

The Kyber CCA-secure KEM. The Kyber scheme provides a CCA-secure key encapsulation mechanism, based on the Fujisaki-Okamoto transform [FO99]. We recall the pseudo-code from [BDK+18] below. It requires two different hash functions H and G. The main principle of the Fujisaki-Okamoto transform is to check the validity of a ciphertext by performing a re-encryption with the same randomness (see the variable r' at Line 3 of Algorithm 63 below), and a comparison with the original ciphertext.

Note that the Kyber.Decaps algorithm does not output $\perp$ for invalid ciphertexts, as originally in the FO transform. Instead, it outputs a pseudo-random value from the hash of a secret seed z and the ciphertext c. This variant of the FO transform was proven secure in [HHK17]. However, the variant remains secure even if the adversary is given the result of the ciphertext comparison, under the condition that the IND-CPA scheme is γ-spread, which essentially means that ciphertexts have sufficiently large entropy (see [HHK17]), which is the case in Kyber. Therefore, in the high-order masking of Kyber, the bit b of the comparison can be computed without mask (as in [BGR+21]), because for the simulation of the probes the bit b can be given for free to the simulator.

Algorithm 62 Kyber.Encaps(pk)

1: $m \xleftarrow{\$} \{0,1\}^{256}$
2: $(\hat{K}, r) := G(H(pk), m)$
3: $c := \mathsf{Kyber.CPA.Enc}(pk, m; r)$
4: $K = H(\hat{K}, H(c))$
5: **return** c, K

Algorithm 63 Kyber.Decaps($sk = (\vec{s}, z, \vec{t}, \rho), c = (\vec{u}, v)$)

1: $m' := \mathsf{Kyber.CPA.Dec}(\vec{s}, c)$
2: $(\hat{K}', r') := G(H(pk), m')$
3: $(\vec{u'}, v') := \mathsf{Kyber.CPA.Enc}((\vec{t}, \rho), m'; r')$
4: **if** $(\vec{u'}, v') = (\vec{u}, v)$ **then**
return $K := H(\hat{K}', H(c))$
5: **else return** $K := H(z, H(c))$

8.3.2 Polynomial comparison for Kyber

In this section, we focus on the polynomial comparison in Kyber [BDK+18]. The parameters that we take as examples are those of Kyber768. We made this choice because our experiments are done on this version. However, all the following properties apply to the other Kyber parameters. The table Table 8.10 describes all the parameters for the different security level.

Recall that computations in Kyber are performed in the ring $R_{q,1} = \mathbb{Z}_q[X]/(X^N + 1)$ with $N = 256$ and $q = 3329$. To reduce the ciphertext size, the coefficients of the ciphertext are compressed from modulo q to d bits using the function:

$$\mathsf{Compress}_{q,d}(x) := \left\lfloor (2^d/q) \cdot x \right\rceil \bmod 2^d$$

and are decompressed using the function $\mathsf{Decompress}_{q,d}(c) := \left\lfloor (q/2^d) \cdot c \right\rceil$. For example according to the Kyber768 parameters, $d = d_u = 10$ for the first part of the ciphertext and $d = d_v = 4$ for the second part.

In the IND-CCA decryption based on the Fujisaki-Okamoto transform [FO99], we must perform a polynomial comparison between two compressed ciphertexts: the input ciphertext $\tilde{c}$, and the re-encrypted ciphertext c. The $\mathsf{Compress}_{q,d}$ function is applied coefficient-wise, so for simplicity we first consider a single coefficient. Let x be the re-encrypted coefficient modulo q before compression, and let c be the resulting compressed coefficient, that is $c = \mathsf{Compress}_{q,d}(x)$. We must therefore perform the comparison with the input ciphertext $\tilde{c}$ modulo 2^d:

$$\tilde{c} \stackrel{?}{=} \mathsf{Compress}_{q,d}(x) \mod 2^d \tag{8.11}$$

There are two possible approaches to perform this comparison. The first approach consists in performing the comparison as in (8.11). Since the re-encrypted coefficient x is arithmetically masked modulo q, we show how to high-order compute $\mathsf{Compress}_{q,d}(x)$ with arithmetically masked input modulo q, and Boolean masked output in $\{0,1\}^d$. We can then perform the high-order polynomial comparison over Boolean shares, using the technique from Section 8.2.2. We describe in Section 8.3.2 the high-order computation of the Compress function.

A second approach is to avoid the computation of the Compress function, as already used in [BGR+21]. Namely instead of performing the comparison over $\{0,1\}^d$ as in (8.11), one can equivalently compute the set of candidates $\tilde{x}_i$ such that $\tilde{c} = \mathsf{Compress}_{q,d}(\tilde{x}_i)$. One must then determine whether the re-encrypted coefficient x is equal to one of the (public) candidates $\tilde{x}_i$, using the α arithmetic shares of x modulo q. Our contribution compared to [BGR+21] is to describe an alternative, faster technique when the number of candidates $\tilde{x}_i$ is small, which is the case for $d = d_u = 10$ (see Section 8.3.2).

Finally, we argue that the best approach is hybrid: for the first $\ell_1 = 768$ coefficients of the ciphertext with $d_u = 10$, we do not compute the Compress function and apply our faster technique with the small number of candidates $\tilde{x}_i$, and for the remaining $\ell_2 = 256$ coefficients with $d_v = 4$, we high-order compute the Compress function. We describe this hybrid approach in Section 8.3.2.

	N	k	q	η_1	η_2	(d_u, d_v)	δ
Kyber512	256	2	3329	3	2	(10,4)	2^{-139}
Kyber768	256	3	3329	2	2	(10,4)	2^{-164}
Kyber1024	256	4	3329	2	2	(11,5)	2^{-174}

Table 8.10: Parameter sets for Kyber.

High-order computation of the Compress function

We provide the first description of the high-order computation of the Compress function of Kyber. Our technique can be seen as a generalization of the first-order technique of [FBR+21], based on modulus

switching: it consists in first using more precision, so that the error induced by the modulus switching can be completely eliminated, after a logical shift.

The Compress function is defined as:

$$\mathsf{Compress}_{q,d}(x) = \left\lfloor \frac{2^d \cdot x}{q} \right\rceil \bmod 2^d$$

We are given as input an arithmetic sharing of $x = x_1 + \ldots + x_\alpha \mod q$ and we want to compute a Boolean sharing of $y = \mathsf{Compress}_{q,d}(x) = y_1 \oplus \cdots \oplus y_\alpha \in \{0,1\}^d$. We stress that in [BGR+21], the authors only described the high-order masking of the Compress function with 1-bit output, which corresponds to the IND-CPA decryption function of Kyber. Here we high-order mask Compress for any number of output bits d (for example $d = d_u = 10$ or $d = d_v = 4$ in Kyber768). For the special case $d = 1$ there are more efficient techniques, see for example [BGR+21] and Section 8.1.

We proceed as follows. We first perform a modulus switching of the input coefficients x_i but with more precision; that is we work modulo $2^{d+\alpha}$ for some parameter $\alpha > 0$ and compute:

$$z_1 = \left\lfloor \frac{x_1 \cdot 2^{d+\alpha}}{q} \right\rceil + 2^{\alpha-1} \bmod 2^{d+\alpha}, \quad z_i = \left\lfloor \frac{x_i \cdot 2^{d+\alpha}}{q} \right\rceil \bmod 2^{d+\alpha} \quad \text{for } 2 \le i \le \alpha$$

The rounding can be computed by writing:

$$\left\lfloor \frac{x_i \cdot 2^{d+\alpha}}{q} \right\rceil = \left\lfloor \frac{x_i \cdot 2^{d+\alpha}}{q} + \frac{1}{2} \right\rfloor = \left\lfloor \frac{x_i \cdot 2^{d+\alpha+1} + q}{2q} \right\rfloor$$

which is the quotient of the Euclidean division of $x_i \cdot 2^{d+\alpha+1} + q$ by $2q$.

We then perform an arithmetic to Boolean conversion of the arithmetic shares $z_1, \ldots, z_\alpha$, followed by a logical shift by α bits. This can be done with complexity $\mathcal{O}((d+\alpha) \cdot \alpha^2)$ using [CGV14]. By definition we obtain:

$$y_1 \oplus \cdots \oplus y_\alpha = \left\lfloor \left(\sum_{i=1}^{\alpha} z_i \right) / 2^\alpha \right\rceil \mod 2^d \tag{8.12}$$

and eventually we output the Boolean shares $y_1, \ldots, y_\alpha$. We show below that we indeed have $\mathsf{Compress}_{q,d}(x) = y_1 \oplus \cdots \oplus y_\alpha$ as required, under the condition $2^\alpha > q \cdot \alpha$. This condition determines the number α of bits of precision as a function of the number of shares α. We provide the pseudocode in Algorithm 64 below.

Algorithm 64 HOCompress

Input: $x_1, \ldots, x_\alpha \in \mathbb{Z}_q$
Output: $y_1, \ldots, y_\alpha \in \{0,1\}^d$ such that $y_1 \oplus \cdots \oplus y_\alpha = \mathsf{Compress}_{q,d}(x_1 + \cdots + x_\alpha)$

1: $\alpha \leftarrow \lceil \log_2(q \cdot \alpha) \rceil$
2: $z_1 \leftarrow \lfloor (x_1 \cdot 2^{d+\alpha+1} + q)/(2q) \rfloor + 2^{\alpha-1} \bmod 2^{d+\alpha}$
3: **for** $i = 2$ to α **do** $z_i \leftarrow \lfloor (x_i \cdot 2^{d+\alpha+1} + q)/(2q) \rfloor \bmod 2^{d+\alpha}$
4: $(c_1, \ldots, c_\alpha) \leftarrow \mathsf{ArithmeticToBoolean}(d+\alpha, (z_1, \ldots, z_\alpha))$
5: **for** $i = 1$ to α **do** $y_i \leftarrow c_i \gg \alpha$
6: **return** $y_1, \ldots, y_\alpha$

Theorem 15 (Soundness). *Given $x_1, \ldots, x_\alpha \in \mathbb{Z}_q$ as input for odd $q \in \mathbb{N}$, the algorithm* HOCompress *computes $y_1, \ldots, y_\alpha \in \{0,1\}^d$ such that $y_1 \oplus \cdots \oplus y_\alpha = f(x_1 + \cdots + x_\alpha)$ where $f(x) = \lfloor x \cdot 2^d / q \rceil \bmod 2^d$.*

Proof. Given $x \in \mathbb{Z}_q$, we have:

$$f(x) = \left\lfloor \frac{x \cdot 2^d}{q} \right\rceil \bmod 2^d = \left\lfloor \frac{x \cdot 2^d}{q} + \frac{1}{2} \right\rfloor \bmod 2^d = \left\lfloor \frac{x \cdot 2^{d+1} + q}{2q} \right\rfloor \bmod 2^d$$

We write the Euclidean division $x \cdot 2^{d+1} + q = y \cdot (2q) + \delta$ with $y, \delta \in \mathbb{Z}$ and $0 \leq \delta < 2q$. Therefore $f(x) = y \mod 2^d$. Moreover we must have $\delta \neq 0$, since otherwise $x = 0 \mod q$, which gives $y = 1/2$, a contradiction. Therefore $0 < \delta < 2q$.

We have for some $|e| \leq \alpha/2$:

$$\begin{aligned}\sum_{i=1}^{\alpha} z_i &= \sum_{i=1}^{\alpha} \left\lfloor \frac{x_i \cdot 2^{d+\alpha}}{q} \right\rceil + 2^{\alpha-1} = \sum_{i=1}^{\alpha} \frac{x_i \cdot 2^{d+\alpha}}{q} + 2^{\alpha-1} + e \mod 2^{d+\alpha} \\ &= \frac{x \cdot 2^{d+\alpha}}{q} + 2^{\alpha-1} + e = 2^{\alpha} \cdot \frac{x \cdot 2^{d+1} + q}{2q} + e \mod 2^{d+\alpha} \\ &= 2^{\alpha} \cdot \left(y + \frac{\delta}{2q} \right) + e = 2^{\alpha} \cdot y + 2^{\alpha} \left(\frac{\delta}{2q} + e \cdot 2^{-\alpha} \right) \mod 2^{d+\alpha}\end{aligned}$$

From (8.12), if we ensure that $0 \leq \delta/(2q) + e \cdot 2^{-\alpha} < 1$, then we must have $y = f(x) = y_1 \oplus \cdots \oplus y_\alpha$ as required. Since $0 < \delta < 2q$, it is sufficient to ensure that $e \cdot 2^{-\alpha} < 1/(2q)$, and therefore a sufficient condition is $\alpha \cdot 2^{-\alpha} < 1/q$. Therefore it is sufficient to ensure $2^\alpha > q \cdot \alpha$. □

Complexity of HOCompress. The number of operations of the HOCompress algorithm above is:

$$\mathcal{C}_{\mathsf{HOComp}}(\alpha, d, q) = 5\alpha + 1 + \mathcal{C}_{\mathsf{AB}}(d + \alpha, \alpha)$$

We refer to [CGV14] for the operation count of arithmetic to Boolean conversion, with $\mathcal{C}_{\mathsf{AB}}(d+\alpha, \alpha) = \mathcal{O}((d+\alpha) \cdot \alpha^2)$. With $d < \log_2 q$ and $\alpha = \lceil \log_2(q \cdot \alpha) \rceil$, the total complexity of HOCompress is therefore $\mathcal{O}(\alpha^2 \cdot (\log q + \log \alpha))$.

Security. The following theorem shows that the HOCompress achieves the $(\alpha-1)$-NI property. The proof follows from the $(\alpha-1)$-NI property of the ArithmeticToBoolean algorithm, and the fact that the perfect simulation of z_i requires the knowledge of the input x_i only.

Theorem 16 ($(\alpha-1)$-NI security)**.** *The* HOCompress *algorithm achieves the* $(\alpha-1)$*-NI property.*

Polynomial comparison with Compress. Recall that we must perform the comparison $\tilde{c} \stackrel{?}{=} \mathsf{Compress}_{q,d}(x)$, where for simplicity we consider a single coefficient $\tilde{c}$. By applying the HOCompress algorithm, we obtain α Boolean shares such that $c = c_1 \oplus \cdots \oplus c_\alpha$. We must therefore zero-test the value $(c_1 \oplus \tilde{c}) \oplus c_2 \oplus \cdots \oplus c_\alpha$, which can be done using the ZeroTestBool algorithm from Section 8.2.1.

For multiple coefficients, we apply the HOCompress algorithm separately on each coefficient $x^{(j)}$ of the re-encrypted uncompressed ciphertext. We obtain the compressed ciphertext c masked with α Boolean shares. As previously, we xor each coefficient of the input ciphertext $\tilde{c}$ with the first share of the corresponding coefficient in c, and we apply the PolyZeroTestBool algorithm from Section 8.2.2 to perform the comparison.

Polynomial comparison for Kyber without Compress

In this section we describe an alternative technique for ciphertext comparison, already used in [BGR+21], that performs the comparison on uncompressed ciphertexts, and avoids the high-order computation of the $\mathsf{Compress}_{q,d}(x)$ function. As previously, we first consider for simplicity a single polynomial coefficient. Given a compressed input ciphertext $\tilde{c}$ and an uncompressed re-encrypted ciphertext x, we must check that $\tilde{c} = \mathsf{Compress}_{q,d}(x)$, where x is arithmetically masked with α shares modulo q. For this we use the equivalence:

$$\tilde{c} = \mathsf{Compress}_{q,d}(x) \iff x \in \mathsf{Compress}_{q,d}^{-1}(\tilde{c})$$

Given $\tilde{c}$ as input, we must therefore compute the list of candidates $\mathsf{Compress}_{q,d}^{-1}(\tilde{c})$, and check whether x belongs to the set of candidates. Given $0 \leq a < b < q$, we denote by $[a, b]_q$ the discrete interval

$\{a, a+1, \ldots, b\}$; similarly given $0 \leq b < a < q$, we denote by $[a, b]_q$ the discrete interval $\{a, a+1, \ldots, q-1, 0, 1, \ldots, b\}$. In the following, we show that we always have $\mathsf{Compress}_{q,d}^{-1}(\tilde{c}) = [a, b]_q$ for some a, b.

We can then distinguish two cases. If the number of candidates is small, we can perform individual comparisons. More precisely, letting $\{\tilde{x}_1, \ldots, \tilde{x}_m\} = \mathsf{Compress}_{q,d}^{-1}(\tilde{c})$ be the list of candidates, we must test that $x = \tilde{x}_i$ for some $1 \leq i \leq m$. Recall that x is arithmetically masked with α shares modulo q. Therefore we can high-order compute $z = \prod_{i=1}^{m}(x - \tilde{x}_i) \bmod q$ and then apply a high-order zero-test of z modulo q. Alternatively, for a large number of candidates, the authors of [BGR+21] describe a high-order algorithm checking that $x \in [a, b]_q$ by performing two high-order comparisons. We describe the two methods in more details in the next subsections.

Computing the set of candidates

Given a compressed coefficient $\tilde{c}$, we must compute the list of candidates $\mathsf{Compress}_{q,d}^{-1}(\tilde{c})$. From [BDK+18], we know that for any $x \in \mathbb{Z}_q$ such that $\tilde{c} = \mathsf{Compress}_{q,d}(x)$, letting the value $y = \mathsf{Decompress}_{q,d}(\tilde{c})$ we must have:

$$|y - x \bmod^{\pm} q| \leq B_{q,d} := \left\lfloor \frac{q}{2^{d+1}} \right\rceil$$

Therefore the number of candidates is upper-bounded by $2B_{q,d} + 1$. The following lemma shows that there are always at least $2B_{q,d} - 1$ candidates around the decompressed value y, with possibly 2 additional candidates to test with $\mathsf{Compress}$.

Lemma 7. *Assume $d < \lceil \log_2 q \rceil$. Let $\tilde{c} \in \mathbb{Z}_{2^d}$ and let $y = \mathsf{Decompress}_{q,d}(\tilde{c})$. We have $[y - B_{q,d} + 1, y + B_{q,d} - 1]_q \subset \mathsf{Compress}_{q,d}^{-1}(\tilde{c}) \subset [y - B_{q,d}, y + B_{q,d}]_q$.*

Proof. We first show that if $x \in \mathsf{Compress}_{q,d}^{-1}(\tilde{c})$, then we must have $|x - y \bmod^{\pm} q| \leq B_{q,d}$, where $y = \mathsf{Decompress}_{q,d}(\tilde{c})$. This will imply $\mathsf{Compress}_{q,d}^{-1}(\tilde{c}) \subset [y - B_q, \ldots, y + B_q]_q$. Namely in this case we have $\tilde{c} = \mathsf{Compress}_{q,d}(x)$, and therefore we have

$$y = \mathsf{Decompress}_{q,d}(\mathsf{Compress}_{q,d}(x)) = \left\lfloor \frac{q}{2^d} \cdot \left(\left\lfloor \frac{2^d}{q} \cdot x \right\rceil \bmod 2^d \right) \right\rceil$$

We write $\left\lfloor (2^d/q) \cdot x \right\rceil = (2^d/q) \cdot x + \varepsilon$ for some $|\varepsilon| \leq 1/2$. We obtain:

$$\begin{aligned}
|x - y \bmod^{\pm} q| &= \left| x - \left\lfloor \frac{q}{2^d} \cdot \left(\left\lfloor \frac{2^d}{q} \cdot x \right\rceil \bmod 2^d \right) \right\rceil \bmod^{\pm} q \right| \\
&= \left| x - \left\lfloor \frac{q}{2^d} \cdot \left(\frac{2^d}{q} \cdot x + \varepsilon \bmod 2^d \right) \right\rceil \bmod^{\pm} q \right| \\
&= \left| x - \left\lfloor x + \frac{q}{2^d} \cdot \varepsilon \bmod q \right\rceil \bmod^{\pm} q \right| = \left| \left\lfloor \frac{q}{2^d} \cdot \varepsilon \right\rceil \right| \leq B_{q,d}
\end{aligned}$$

Conversely, we show that if $|x - y \bmod^{\pm} q| \leq B_{q,d} - 1$, then we must have $x \in \mathsf{Compress}_{q,d}^{-1}(\tilde{c})$. This will imply $[y - B_q + 1, \ldots, y + B_q - 1]_q \subset \mathsf{Compress}_{q,d}^{-1}(\tilde{c})$. Namely we write again $\left\lfloor (2^d/q) \cdot x \right\rceil = (2^d/q) \cdot x + \varepsilon$ for some $|\varepsilon| \leq 1/2$, and we can write:

$$\begin{aligned}
\left| \tilde{c} - \mathsf{Compress}_{q,d}(x) \bmod^{\pm} 2^d \right| &= \left| \tilde{c} - \left(\left\lfloor \frac{2^d}{q} \cdot x \right\rceil \bmod 2^d \right) \bmod^{\pm} 2^d \right| \\
&= \left| \tilde{c} - \left(\frac{2^d}{q} \cdot x + \varepsilon \right) \bmod^{\pm} 2^d \right| \\
&= \frac{2^d}{q} \cdot \left| \frac{q}{2^d} \cdot \tilde{c} - x - \frac{q}{2^d} \cdot \varepsilon \bmod^{\pm} q \right|
\end{aligned}$$

We write $y = \left\lfloor (q/2^d) \cdot \tilde{c} \right\rceil = (q/2^d) \cdot \tilde{c} + \varepsilon'$ for some $|\varepsilon'| \leq 1/2$, which gives:

$$\begin{aligned} \left| \tilde{c} - \mathsf{Compress}_{q,d}(x) \bmod^{\pm} 2^d \right| &= \frac{2^d}{q} \cdot \left| y - \varepsilon' - x - \frac{q}{2^d} \cdot \varepsilon \bmod^{\pm} q \right| \\ &\leq \frac{2^d}{q} \cdot \left(B_{q,d} - 1 + \frac{q}{2^{d+1}} + \frac{1}{2} \right) < 1 \end{aligned}$$

For the last inequality we use $B_{q,d} < q/(2^{d+1}) + 1/2$ for odd q. This implies $\tilde{c} = \mathsf{Compress}_{q,d}(x)$, which proves the lemma. □

Generating the list of candidates. From the above lemma, to generate the list of candidates $\mathsf{Compress}_{q,d}^{-1}(\tilde{c})$, it suffices to consider the set $[a, b]_q$ with $a = y - B_{q,d}$ and $b = y + B_{q,d}$ and to test whether the two elements at the border belong to the set, that is we check whether $\mathsf{Compress}_{q,d}(a) = c$ and $\mathsf{Compress}_{q,d}(b) = c$. We provide the pseudocode in Algorithm 65 below.

Algorithm 65 CompressInv

Input: $\tilde{c} \in \mathbb{Z}_{2^k}$
Output: $a, b \in \mathbb{Z}$ such that $\mathsf{Compress}_{q,d}^{-1}(\tilde{c}) = [a, b]_q$.

1: $B_{q,d} \leftarrow \left\lfloor \frac{q}{2^{d+1}} \right\rceil$
2: $y \leftarrow \mathsf{Decompress}_{q,d}(\tilde{c})$
3: $a \leftarrow y - B_{q,d} \bmod q$, $b \leftarrow y + B_{q,d} \bmod q$
4: **if** $\mathsf{Compress}_{q,d}(a) \neq \tilde{c}$ **then** $a \leftarrow a + 1 \bmod q$
5: **if** $\mathsf{Compress}_{q,d}(b) \neq \tilde{c}$ **then** $b \leftarrow b - 1 \bmod q$
6: **return** a, b

Number of candidates. We provide in Table 8.11 the value of the upper-bound $2B_{q,d} + 1$ on the number of candidates, and the maximum number of candidates N_{max}, for $q = 3329$. We see that individual comparisons are feasible only for $d = 10, 11$, while we must use range comparison for $d = 4, 5$. We describe the two techniques in the next section.

	$d = 4$	$d = 5$	$d = 10$	$d = 11$
$2 \cdot B_{q,d} + 1$	209	105	5	3
N_{max}	209	105	4	2

Table 8.11: Upper-bound on the number of candidates, for $q = 3329$.

Individual comparison

Letting $\{\tilde{x}_1, \ldots, \tilde{x}_m\} = \mathsf{Compress}_{q,d}^{-1}(\tilde{c})$ be the list of candidates, we must test whether $x = \tilde{x}_i$ for some $1 \leq i \leq m$. For this, given an arithmetically masked x with α shares with $x = x_1 + \cdots + x_\alpha \mod q$, we high-order compute the value

$$z = \prod_{i=1}^{m} (x - \tilde{x}_i) \bmod q \tag{8.13}$$

and we have that $z = 0 \mod q$ if and only if $x = \tilde{x}_i$ for some $1 \leq i \leq m$. We provide in Algorithm 66 the pseudocode description of the SecMultList algorithm, computing the α shares of z in (8.13), from the input shares x_i of x. For m candidates, the number of operations for high-order computing z is therefore at most:

$$\mathcal{C}_{\mathsf{SecMultList}}(d, \alpha) = (2 \cdot B_{q,d} + 1) \cdot \mathcal{C}_{\mathsf{SecMult}}(\alpha)$$

As a second step, one can apply a high-order zero-test of z modulo q, as the ZeroTestMult algorithm from Section 8.2.1.

Algorithm 66 SecMultList

Input: $x_1, \ldots, x_\alpha \in \mathbb{Z}_q$ s.t. $\sum_i x_i \mod q = x$, a prime q, an index m and $a_1, \ldots a_m \in \mathbb{Z}_q$
Output: $z_1, \ldots, z_\alpha \in \mathbb{Z}_q$ such that $\sum_{i=1}^{\alpha} z_i = \prod_{i=1}^{m}(x - a_i) \mod q$
1: $(z_1, z_2, \ldots, z_\alpha) \leftarrow (1, 0, \ldots 0)$
2: **for** $i = 1$ to m **do**
3: $\quad (z_1, z_2, \ldots, z_\alpha) \leftarrow \mathsf{SecMult}((z_1, \ldots, z_\alpha), (x_1 - a_i \bmod q, x_2, \ldots, x_\alpha))$
4: **end for**
5: **return** $(z_1, \ldots, z_\alpha)$

Theorem 17. *The* SecMultList *algorithm is* $(\alpha - 1)$-SNI.

Proof. The SecMultList algorithm is $(\alpha-1)$-SNI since it is the composition of m iterations of the $(\alpha-1)$-SNI secMult algorithm. We stress that the first multiplication by 1 (initialized in line 1) is here on purpose since it is equivalent to an $(\alpha - 1)$-SNI masks refreshing, which ensures the independence between both subsequent inputs.

□

The above applies for a single coefficient x. In reality we must compare ℓ coefficients, so for each coefficient $x^{(j)}$ whose compressed value must be compared to the coefficient $\tilde{c}_j$ of the input ciphertext $\tilde{c}$, we compute the corresponding list of candidates from $\tilde{c}_j$, and then the corresponding arithmetically masked $z^{(j)}$. Then a polynomial zero-test is applied to the set of arithmetically masked $z^{(j)}$'s modulo q, for example the PolyZeroTestMult algorithm from Section 8.2.2.

Polynomial comparison with range test [BGR+21]

When the number of candidates in $\mathsf{Compress}_{q,d}^{-1}(\tilde{c})$ is too large (which is the case for $d = 4, 5$), we cannot perform individual comparisons as in the previous section. Instead, we must test whether $x \in [a, b]_q = \mathsf{Compress}_{q,d}^{-1}(\tilde{c})$ by performing two high-order comparisons with the interval bounds a and b. We recall the technique from [BGR+21].

We let $k = \lfloor \log_2 q \rfloor$, so that $2^k < q < 2^{k+1}$. We write $\Delta := q - 2^k - 1$. For Kyber with $q = 3329$, we get $k = 11$ and $\Delta = 1280$. We assume that the bounds of the interval $[a, b]_q$ satisfy $b - a \bmod^{\pm} q \leq \Delta$. Recall that we have the upper-bound $b - a \bmod^{\pm} q \leq 2 \cdot B_{q,d} + 1 = 2 \cdot \left\lfloor q/2^{d+1} \right\rceil + 1$. Therefore the assumption is satisfied for Kyber for $d \geq 2$.

Taking as input the shares x_i of $x = x_1 + \cdots + x_n \mod q$, we want to output a α-shared bit u such that $u = 1$ if $x \in [a, b]_q$ and $u = 0$ otherwise. For this we use:

$$x \in [a, b]_q \iff ((2^k + x - a \bmod q) \geq 2^k) \wedge ((x - b - 1 \bmod q) \geq 2^k) \tag{8.14}$$

Namely we have using $\Delta = q - 2^k - 1$:

$$((2^k + x - a \bmod q) \geq 2^k) \iff x \in [a, a + \Delta]_q$$
$$((x - b - 1 \bmod q) \geq 2^k) \iff x \in [b - \Delta, b]_q$$

Since by assumption $b - a \bmod^{\pm} q \leq \Delta$, we have $[a, b]_q \subset [a, a + \Delta]_q$ and similarly $[a, b]_q \subset [b - \Delta, b]_q$. Moreover from $2\Delta < q$ we must have $[b, a+\Delta]_q \cap [b-\Delta, a]_q = \emptyset$. This implies $[a, b]_q = [a, a+\Delta]_q \cap [b-\Delta, b]_q$, which proves (8.14).

From (8.14), we perform a high-order arithmetic modulo q to Boolean conversion of the two values $2^k + x - a$ and $x - b - 1$ modulo q using [BBE+18], and we perform a high-order And (with SecAnd) of the most significant bit of the two results (using the Boolean shares). The number of operations is therefore:

$$\mathcal{C}_{\mathsf{range}}(k, \alpha) = 2 \cdot \mathcal{C}_{\mathsf{AB}}(k+1, \alpha) + \mathcal{C}_{\mathsf{SecAnd}}(\alpha)$$

Algorithm 67 RangeTestShares

Input: $x_1, \ldots, x_\alpha \in \mathbb{Z}_q$ for prime q with $\sum_i x_i = x \mod q$, $k = \lfloor \log_2 q \rfloor$, bounds a and b s.t. $b - a \bmod^{\pm} q \leq q - 2^k - 1$.

Output: $u_1, \ldots, u_\alpha \in \mathbb{Z}_q$ with $\sum_i u_i = 1 \mod q$ if $x \in [a, b]_q$ and $\sum_i u_i = 0 \mod q$ otherwise

1: $(A_1, \ldots, A_\alpha) \leftarrow (x_1 + 2^k - a \bmod q, x_2, \ldots, x_\alpha)$
2: $(B_1, \ldots, B_\alpha) \leftarrow (x_1 - b - 1 \bmod q, x_2, \ldots, x_\alpha)$
3: $(y_1, \ldots, y_\alpha) \leftarrow \mathsf{ArithmeticToBoolean}(q, (A_1, \ldots, A_\alpha))$
4: $(z_1, \ldots, z_\alpha) \leftarrow \mathsf{ArithmeticToBoolean}(q, (B_1, \ldots, B_\alpha))$
5: $(u_1, \ldots, u_\alpha) \leftarrow \mathsf{SecAnd}(1, (\mathsf{MSB}(y_1), \ldots, \mathsf{MSB}(y_\alpha)), (\mathsf{MSB}(z_1), \ldots, \mathsf{MSB}(z_\alpha)))$
6: **return** $(u_1, \ldots, u_\alpha)$

Ciphertext comparison in Kyber: hybrid approach

We first compare in Table 8.12 the efficiency of the approaches with and without Compress. For the coefficients with compression to $d_u = 10$ bits, without using the Compress function, since the number of candidates is small, we can use either the RangeTestShares algorithm from [BGR+21], or our SecMultList algorithm. We see in Table 8.12 that the latter is significantly faster. It is also faster than applying the Compress function with our HOCompress algorithm. On the other hand, for the coefficients with compression to $d_v = 4$ bits, without using the Compress function, one must use the RangeTestShares algorithm from [BGR+21]. But we see that our HOCompress is nevertheless faster. Namely, it uses only a single arithmetic to Boolean conversion with a power-of-two modulus, whereas RangeTestShares uses two arithmetic to Boolean conversions modulo q, which is more costly than with a power-of-two modulus.

In summary, from Table 8.12, we deduce that for $d = d_u = 10$, our SecMultList approach without Compress is faster, while for $d = d_v = 4$, our HOCompress algorithm is faster. Therefore, to perform the ciphertext comparison in Kyber, we use a hybrid approach, applying the Compress function only for the last $\ell_2 = 256$ coefficients of the ciphertext, for which $d = d_v = 4$.

		Security order t								
		1	2	3	4	5	6	7	8	9
$d_u = 10$	RangeTestShares [BGR+21]	707	2 318	4 314	7 577	11 225	15 620	20 400	26 809	33 603
	SecMultList	45	120	230	375	555	770	1 020	1 305	1 625
	HOCompress	131	868	1 579	3 181	4 898	6 764	8 809	11 823	15 611
$d_v = 4$	RangeTestShares [BGR+21]	707	2 318	4 314	7 577	11 225	15 620	20 400	26 809	33 603
	HOCompress	101	658	1 195	2 431	3 740	5 162	6 721	9 015	12 041

Table 8.12: Comparison of the RangeTestShares, SecMultList and HOCompress algorithms, in number of operations, for $q = 3329$ and $d = d_u = 10$ or $d = d_v = 4$.

Procedure for ciphertext comparison. Recall that for masking the IND-CCA decryption of Kyber, we must perform a comparison between the unmasked input ciphertext $\tilde{c}$, and the masked re-encrypted ciphertext c. Moreover, with the Kyber768 parameters, a ciphertext consists of 4 polynomials with 256 coefficients each. The coefficients of the first 3 polynomials are compressed with $d_u = 10$ bits, while

the coefficients of the last polynomial are compressed with $d_v = 4$ bits. Starting from the re-encrypted uncompressed ciphertext c_u which is masked modulo q, and given the input ciphertext $\tilde{c}$, we proceed as follows:

1. For each of the first $\ell_1 = 768$ coefficients of c_u, with compression parameter $d_u = 10$, we use the individual comparison technique from Section 8.3.2 (Algorithm SecMultList). We obtain a set of values $z^{(j)}$ arithmetically masked modulo q, that must all be equal to 0, for $1 \leq j \leq \ell_1$.

2. For each of the last $\ell_2 = 256$ coefficients of c_u, we apply the HOCompress algorithm with $d_v = 4$ bits. We obtain a set of ℓ_2 coefficients $c^{(j)}$ for $1 \leq j \leq \ell_2$, which are Boolean masked with α shares.

3. We xor each of the last ℓ_2 coefficients of the input ciphertext $\tilde{c}$ to the first Boolean share of each of the corresponding ℓ_2 coefficients $c^{(j)}$. This gives a vector of ℓ_2 coefficients $x^{(j)}$ for $1 \leq j \leq \ell$, which are Boolean masked with α shares, and that must all be equal to 0.

4. We apply the PolyZeroTestBool algorithm (Alg. 56) to the set of ℓ_2 coefficients $x^{(j)}$, but without recombining the shares at the end of the ZeroTestBoolLog algorithm. That is, we obtain Boolean shares b_i for $1 \leq i \leq \alpha$, with $b' = b_1 \oplus \cdots \oplus b_\alpha$ and $b' = 1$ if the ℓ_2 coefficients $x^{(j)}$ are zero, and $b' = 0$ otherwise.

5. We take the complement of b' by taking the complement of b_1, and convert the result from Boolean to arithmetic masking modulo q. We obtain an additional coefficient $z^{(\ell_1+1)}$ arithmetically masked modulo q, and that must be equal to 0.

6. Finally, we perform a zero-test of the $\ell_1 + 1$ coefficients $z^{(i)}$ for $1 \leq i \leq \ell_1 + 1$, using the PolyZeroTestMult algorithm. We obtain a bit $b = 1$ if the two ciphertexts are equal, and $b = 0$ otherwise, as required.

The number of operations is then:

$$\begin{aligned}\mathcal{C} = \; & \ell_1 \cdot \mathcal{C}_{\mathsf{SecMultList}}(d_u, \alpha) + \ell_2 \cdot \mathcal{C}_{\mathsf{HOComp}}(d_v, \alpha) + \ell_2 + \mathcal{C}_{\mathsf{PolyZeroTestBool}}(13, \ell_2, \alpha) + \\ & \mathcal{C}_{BA}(1, \alpha) + \mathcal{C}_{\mathsf{polyZT}}(q, \ell_1 + 1, \alpha)\end{aligned}$$

Operation count and concrete running time

We provide in Table 8.13 a comparison of the operation count for the ciphertext comparison in Kyber, first using the approach from [BGR+21] without Compress, and the PolyZeroTestMult methods. We see that the hybrid approach is significantly faster, especially for high security orders. We have also performed a C implementation that confirms these results, see Table 8.14 below. We also provide in Table 8.15 the number of 32-bit random values.

Polynomial comparison in Kyber	Security order t								
	1	2	3	4	5	6	7	8	9
Without Compress [BGR+21]	786	2 574	4 792	8 413	12 465	17 346	22 657	29 770	37 313
Hybrid, with PolyZeroTestMult	121	350	603	1 065	1 575	2 144	2 778	3 629	4 697

Table 8.13: Operation count for the three proposed methods to perform ciphertext comparison (with Compress and without Compress using PolyZeroTestMult), in thousands of operations.

Polynomial comparison in Kyber	Security order t							
	1	2	3	4	5	6	7	8
Without Compress [BGR+21]	1 395	3 722	6 230	9 619	14 517	19 206	24 783	33 675
Hybrid, with PolyZeroTestMult	185	410	562	966	1 731	2 046	2 829	3 842

Table 8.14: Running time in thousands of cycles for a C implementation on Intel(R) Core(TM) i7-1065G7, for the three methods considered in Table 8.13.

Polynomial comparison in Kyber	Security order t							
	1	2	3	4	5	6	7	8
Without Compress [BGR+21]	79	309	618	1 146	1 755	2 512	3 350	4 476
Hybrid, with PolyZeroTestMult	12	31	50	94	145	202	264	354

Table 8.15: Number of calls to the rand() function (outputting a 32-bit value), in thousands of calls, rounded down to the closest thousand, for the three methods considered in Table 8.13.

8.3.3 High-order masking of Kyber

We describe the high-order masking of the Kyber.Decaps algorithm recalled in Algorithm 68, using the techniques from the previous sections.

Algorithm 68 Kyber.Decaps$(sk = (\vec{s}, z, \vec{t}, \rho), c = (\vec{u}, v))$

1: $m' := \mathsf{Kyber.CPA.Dec}(\vec{s}, c)$
2: $(\hat{K}', r') := G(H(pk), m')$
3: $(\vec{u'}, v') := \mathsf{Kyber.CPA.Enc}((\vec{t}, \rho), m'; r')$
4: **if** $(\vec{u'}, v') = (\vec{u}, v)$ **then**
   return $K := H(\hat{K}', H(c))$
5: **else return** $K := H(z, H(c))$

Algorithm 69 Kyber.CPA.Enc(pk, m)

1: $r \xleftarrow{\$} \{0,1\}^{256}$
2: $\mathbf{t} := \mathsf{Decompress}_{q,d_t}(\mathbf{t})$
3: $\mathbf{A} \leftarrow \mathcal{R}_q^{k \times k} := \mathsf{Sam}(\rho)$
4: $\mathbf{r}, \mathbf{e}_1, e_2 \leftarrow \chi_{\eta_1}^k \times \chi_{\eta_2}^k \times \chi_{\eta_2} := \mathsf{Sam}(r)$
5: $\vec{u} := \mathsf{Compress}_{q,d_u}(\mathbf{A}^T\mathbf{r} + \mathbf{e}_1)$
6: $v := \mathsf{Compress}_{q,d_v}(\mathbf{t}^T\mathbf{r} + e_2 + \lfloor q/2 \rceil \cdot m)$
7: **return** $c := (\vec{u}, v)$

Keccak. The Kyber routines use hash functions based on keccak. Keccak algorithm performs linear boolean operations expect the logical `and`. Therefore, to obtain a high-order secure implementation of Keccak we only have to implement a `secAnd` algorithm as the one described in Algorithm 24 page 79.

1. We consider Line 1 of Algorithm 68, with the IND-CPA decryption as the first step. We assume that the secret key $\vec{s} \in \mathcal{R}^k$ is initially masked with α shares, with $\vec{s} = \vec{s}_1 + \cdots + \vec{s}_\alpha \mod q$, where $\vec{s}_i \in (\mathcal{R}_q)^k$ for all $1 \leq i \leq \alpha$. Therefore, at Line 3 of the Kyber.CPA.Dec algorithm, we obtain a value $v - \vec{s}^T\vec{u}$ that is arithmetically α-shared modulo q. We must therefore compute the $\mathsf{Compress}_{q,1}$ function on this value, which is the same as the threshold function th from (8.10). For this we use the modulus switching and table recomputation technique from Section 8.1.4, which outputs a Boolean masked message $m' = m_1 \oplus \cdots \oplus m_\alpha = \mathsf{th}(v - \vec{s}^T\vec{u})$.

2. At Line 2 of Algorithm 68, starting from the Boolean masked m', we use an α-shared Boolean implementation of the hash function G, and obtain as output the Boolean α-shared values $\hat{K}'$ and r'.

3. At Line 3 of Algorithm 68, we start with Line 4 of Algorithm 69 which is the masked binomial sampling. Starting from the Boolean α-shared r', we must obtain values $\mathbf{r}$, $\mathbf{e}_1$ and e_2 which are arithmetically α-shared modulo q. For this we use the α-shared binomial sampling from Section

8.1.5, based on Boolean to arithmetic modulo q conversion (based on table recomputation). We use the random generation modulo q described in Section 5.2.4.

4. We proceed with lines 5 and 6 of Algorithm 69. We obtain the values $\mathbf{A}^T \cdot \mathbf{r} + \mathbf{e}_1$ and $\mathbf{t}^T \cdot \mathbf{r} + e_2 + \lfloor q/2 \rceil \cdot m$ arithmetically α-shared modulo q. In particular, the α-shared value $\lfloor q/2 \rceil \cdot m$ is obtained using the table-based Boolean to arithmetic modulo q conversion from Section 8.1.3.

5. At Line 6 of Algorithm 69, the α-shared value $\mathbf{t}^T \cdot \mathbf{r} + e_2 + \lfloor q/2 \rceil \cdot m$ is high-order compressed into v' using the HOCompress algorithm from Section 8.3.2. The value v' is therefore Boolean α-shared in $\{0,1\}^{d_v}$. On the other hand, the vector $\vec{u}'$ at Line 5 is left uncompressed.

6. For the ciphertext comparison at Line 4 of Algorithm 68, we use the hybrid technique from Section 8.3.2 with the arithmetically masked modulo q uncompressed vector $\vec{u}'$, and the Boolean masked compressed value v'. We obtain a bit b in the clear.

7. Finally, if $b = 1$, we use the Boolean α-shared $\hat{K}'$ to obtain a Boolean α-shared session key K, using an α-shared implementation of H. Similarly, if $b = 0$, we use the Boolean α-shared secret z to obtain the Boolean α-shared session key K.

We describe in Section 8.5 the implementation results of the fully masked Kyber.Decaps.

8.4 Fully masked implementation of Saber

8.4.1 The Saber Key Encapsulation Mechanism (KEM)

Saber [BMD+21] is based on the hardness on the module learning-with-rounding (M-LWR) problem. The difference with Kyber is that instead of explicitly adding error terms $\mathbf{e}, \mathbf{e}_1, e_2$ from a "small" distribution, the errors are deterministically added by applying a rounding function mapping $\mathbb{Z}_q$ to $\mathbb{Z}_p$ with $p < q$. For Saber, both p and q are powers of two; therefore the rounding function is a shift extracting the $\log_2(p)$ most significant bits of its input.

The Saber submission provides three parameters sets LightSaber, Saber and FireSaber with claimed security level equivalent to AES-128, AES-192 and AES-256 respectively; see Table 8.16. We recall the pseudocode below. The constants h_1, h_2 and $\vec{h}$ are needed to center the errors introduced by rounding around 0. We write $q = 2^{\epsilon_q}$ and $p = 2^{\epsilon_p}$.

Algorithm 70 Saber.CPA.KeyGen()

1: $\rho, \sigma \xleftarrow{\$} \{0,1\}^{256}$
2: $\mathbf{A} \leftarrow \mathcal{R}_q^{k \times k} := \mathsf{Sam}(\rho)$
3: $\mathbf{s} \leftarrow \chi_\mu^k := \mathsf{Sam}(\sigma)$
4: $\mathbf{t} := (\mathbf{A}^T \mathbf{s} + \mathbf{h} \bmod q) \gg (\epsilon_q - \epsilon_p) \in \mathcal{R}_p^k$
5: **return** $pk := (\mathbf{t}, \rho)$, $sk := \mathbf{s}$

	N	k	q	p	T	μ
LightSaber	256	2	2^{13}	2^{10}	2^3	5
Saber	256	3	2^{13}	2^{10}	2^4	4
FireSaber	256	4	2^{13}	2^{10}	2^6	3

Table 8.16: Parameter set for Saber.

Algorithm 71 Saber.CPA.Enc(pk, m)

1: $r \xleftarrow{\$} \{0,1\}^{256}$
2: $\mathbf{A} \leftarrow \mathcal{R}_q^{k \times k} := \mathsf{Sam}(\rho)$
3: $\mathbf{r} \leftarrow \chi_\mu^k := \mathsf{Sam}(r)$
4: $\vec{u} := (\mathbf{A}\mathbf{r} + \mathbf{h} \bmod q) \gg (\epsilon_q - \epsilon_p) \in \mathcal{R}_p^k$
5: $v' := \mathbf{t}^T (\mathbf{r} \bmod p) \in \mathcal{R}_p$
6: $c_m := (v' + h_1 - 2^{\epsilon_p - 1} m \bmod p) \gg (\epsilon_p - \epsilon_T) \in \mathcal{R}_T$
7: **return** $c := (\vec{u}, c_m)$

Algorithm 72 Saber.CPA.Dec($sk, c = (\vec{u}, c_m)$)

1: $v := \vec{u}^T (\vec{s} \bmod p) \in \mathcal{R}_p$
2: $m := (v - 2^{\epsilon_p - \epsilon_T} c_m + h_2 \bmod p) \gg (\epsilon_p - 1) \in \mathcal{R}_2$
3: **return** m

The Saber CCA-secure key encapsulation mechanism is similar to that of Kyber. We recall the pseudocode below.

Algorithm 73 Saber.Encaps(pk)

1: $m \xleftarrow{\$} \{0,1\}^{256}$
2: $(\hat{K}, r) := G(H(pk), m)$
3: $c :=$ Saber.CPA.Enc$(pk, m; r)$
4: $K = H(\hat{K}, c)$
5: **return** c, K

Alg. 74 Saber.Decaps$(sk = (\vec{s}, z, \vec{t}, \rho), c)$

1: $m' :=$ Saber.CPA.Dec$(\vec{s}, c)$
2: $(\hat{K}', r') := G(H(pk), m')$
3: $c' :=$ Saber.CPA.Enc$((\vec{t}, \rho), m'; r')$
4: **if** $c = c'$ **then return** $K := H(\hat{K}', c)$
5: **else return** $K := H(z, c)$

8.4.2 High-order masking of Saber

The high-order masking of Saber is quite similar to that of Kyber. The main difference is that we work with power-of-two moduli. We describe the high-order masking of the Saber.Decaps algorithm recalled above (Algorithm 74), using the techniques from the previous sections.

1. We consider Line 1 of Algorithm 74. As previously, we assume that the secret key $\vec{s} \in \mathcal{R}_q^k$ is initially arithmetically masked with α shares. By modular reduction, we obtain α shares in $\mathcal{R}_p^k$. Therefore, at Line 1 of the Saber.CPA.Dec algorithm, we obtain a value v that is arithmetically α-shared modulo p. At Line 2, we obtain α Boolean shares of the message m by arithmetic to Boolean conversion modulo $p = 2^{\epsilon_p}$ using [CGV14], and taking the MSB of each share.

2. At Line 2 of Algorithm 74, starting from the Boolean masked m', we use an α-shared Boolean implementation of the hash function G, and obtain as output the Boolean α-shared values $\hat{K}'$ and r'.

3. At Line 3 of Algorithm 74, we start with Line 3 of Algorithm 71 which is the masked binomial sampling. Starting from the Boolean α-shared r', we must obtain a value $\mathbf{r}$ which is arithmetically α-shared modulo q. For this we use the α-masked binomial sampling from Section 8.1.5, based on Boolean to arithmetic modulo q conversion (based on table recomputation).

4. We proceed with Line 4 of Algorithm 71. The vector $\mathbf{Ar}+\mathbf{h}$ mod q is α-shared modulo q. We convert from arithmetic to Boolean masking using [CGV14], and then perform a right shift of all Boolean shares by $\epsilon_q - \epsilon_p$. The vector $\vec{u}'$ as output of Line 4 of Algorithm 71 is therefore Boolean masked with α shares.

5. At Line 5, the value $\vec{r}$ is arithmetically masked modulo p by modular reduction modulo p of the shares modulo q. The value v' is therefore arithmetically masked modulo p. This enables to compute the value $v' + h_1 - 2^{\epsilon_p - 1} m$ mod p at Line 6 with α shares modulo p. As previously the shift by $\epsilon_p - \epsilon_T$ bits is computed via arithmetic to Boolean conversion. At Line 3 of Algorithm 74, the vector $\vec{u}'$ and the value c'_m of the ciphertext $c' = (\vec{u}', c'_m)$ are therefore both in Boolean masked form.

6. For the ciphertext comparison at Line 4, we use the same technique as in Section 8.3.2 for the ciphertext comparison of Kyber, for the second part of the ciphertext with the Compress function (lines 3 and 4). Eventually we recombine the shares and we obtain a bit b in the clear, with $b = 1$ if the two ciphertexts match.

7. Finally, as in Kyber, if $b = 1$, we use the Boolean α-shared $\hat{K}'$ to obtain a Boolean α-shared session key K, using an α-shared implementation of H. Similarly, if $b = 0$, we use the Boolean α-shared secret z to obtain the Boolean α-shared session key K.

8.5 Practical implementation

We have implemented in C a high-order version of the Kyber.Decaps and Saber.Decaps algorithms, following the description of sections 8.3.3 and 8.4.2 respectively. For both schemes, we have targeted the parameter set corresponding to NIST security category 3 (parameters Kyber768 and Saber, see tables 8.10 and 8.16). We have run our implementation on a laptop and an embedded component. We provide the source code of the laptop implementation at:

https://github.com/fragerar/HOTableConv/tree/main/Masked_KEMs

For the embedded component, we have used a 100 MHz ARM Cortex-M3 architecture with 48k of RAM, which also includes a hardware accelerator for secure 32-bit random generation. Such component is used in real-life products like bank cards, passports, secure elements, etc.[4] The embedded code is almost the same as for the laptop implementation, except for the random generation which uses the hardware accelerator. We have also performed some RAM optimization in order to reduce the number of temporary variables without changing the number of operations.

We see that for both Kyber and Saber the performance gap between the unmasked and the order 1 versions is fairly large. This is because we have used generic gadgets only, with no optimization at order 1. In practice, for first-order security, a significantly lower penalty factor could be obtained via some optimizations. In particular, all techniques based on table recomputation are much more efficient at order 1, since in that case the table can be randomized once and read multiple times.

8.5.1 Kyber

Our high-order implementation of Kyber.Decaps follows the description from Section 8.3.3. To generate random integers modulo q, we use the technique described in Algorithm 23 (see Section 5.2.4), starting from a 32-bit random number generator. The timings are summarized in Table 8.17. For the embedded implementation, we can reach at most a security order of 3, due to RAM limitation.

	Security order t							
	0	1	2	3	4	5	6	7
Intel i7	133	1 164	2 225	4 723	6 613	11 177	14 174	19 806
ARM Cortex-M3	3 173	21 492	39 539	69 348	-	-	-	-

Table 8.17: Kyber.Decaps cycles counts on Intel(R) Core(TM) i7-1065G7 and ARM Cortex-M3, in thousands of cycles.

8.5.2 Saber

Our high-order implementation of Saber.Decaps follows the description from Section 8.4.2. As for Kyber, the embedded implementation can reach at most a security order 3, due to RAM limitation. The timings are summarized in Table 8.18.

	Security order t							
	0	1	2	3	4	5	6	7
Intel i7	100	352	933	1 585	2 828	4 208	5 621	7 251
ARM Cortex-M3	5 682	17 805	42 662	67 389	-	-	-	-

Table 8.18: Saber.Decaps cycles counts on Intel(R) Core(TM) i7-1065G7 and ARM Cortex-M3, in thousands of cycles.

[4] Due to intellectual properties reasons, the component name or detailed description cannot be given.

Bibliography

References

[AAB+19] Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Thomas Pöppelmann, Peter Schwabe, and Douglas Stebila. NewHope, 2019. Available at https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions (cited on page 8).

[ABD+21] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber (version 3.02) – submission to round 3 of the NIST post-quantum project. Specification document (update from August 2021). 2021-08-04. 2021 (cited on pages 8, 123, 124, 138).

[AHH+19] Martin R Albrecht, Christian Hanser, Andrea Hoeller, Thomas Pöppelmann, Fernando Virdia, and Andreas Wallner. Implementing RLWE-based Schemes Using an RSA Co-Processor. *IACR Transactions on Cryptographic Hardware and Embedded Systems*:169–208, 2019 (cited on pages 5, 19, 21, 23, 24, 34, 36, 39, 41, 54, 59, 67).

[ANS] ANSSI. Technical position paper - ANSSI views on the Post-Quantum Cryptography transition. en. Available at https://www.ssi.gouv.fr/publication/anssi-views-on-the-post-quantum\-cryptography-transition/ (cited on page 1).

[ANS22] ANSSI. ANSSI views on the post-quantum cryptography transition, 2022. URL: https://www.ssi.gouv.fr/en/publication/anssi-views-on-the-post-quantum-cryptography-transition/#note17 (cited on pages 4, 19).

[Bar86] Paul Barrett. Implementing The Rivest Shamir And Adleman Public Key Encryption On A Standard Digital Signal Processor. *CRYPTO' 86. CRYPTO 1986. Lecture Notes in Computer Science, vol 263. Springer, Berlin, Heidelberg.*:1156–1158, 1986 (cited on pages 18, 42, 61, 62).

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 116–129, 2016. Publicly available at https://eprint.iacr.org/2015/506.pdf. (cited on pages 9, 78–80).

[BBE+18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. Masking the GLP lattice-based signature scheme at any order. In *Advances in Cryptology - EUROCRYPT 2018 - Proceedings, Part II*, pages 354–384, 2018 (cited on pages 110–112, 117, 124–126, 131, 146).

[BBK16] Nina Bindel, Johannes Buchmann, and Juliane Krämer. Lattice-based signature schemes and their sensitivity to fault attacks. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography FDTC*, pages 63–77. IEEE Computer Society, 2016 (cited on page 82).

[BCG12] Alexandre Berzati, Cécile Canovas-Dumas, and Louis Goubin. A survey of differential fault analysis against classical RSA implementations. In Marc Joye and Michael Tunstall, editors, *Fault Analysis in Cryptography*, Information Security and Cryptography. Springer, 2012 (cited on page 81).

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, volume 3156, pages 16–29. Springer, 2004 (cited on page 81).

[BCZ18] Luk Bettale, Jean-Sébastien Coron, and Rina Zeitoun. Improved high-order conversion from boolean to arithmetic masking. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):22–45, 2018 (cited on pages 111, 117).

[BDH+19] Ciprian Băetu, F Betül Durak, Loïs Huguenin-Dumittan, Abdullah Talayhan, and Serge Vaudenay. Misuse Attacks on Post-Quantum Cryptosystems. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 747–776. Springer, 2019 (cited on page 92).

[BDH+21] Shivam Bhasin, Jan-Pieter D'Anvers, Daniel Heinz, Thomas Pöppelmann, and Michiel Van Beirendonck. Attacking and defending masked polynomial comparison for lattice-based cryptography. Cryptology ePrint Archive, Report 2021/104, 2021. https://eprint.iacr.org/2021/104 (cited on pages 128, 134–136).

[BDK+18] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - kyber: A CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 353–367, 2018 (cited on pages 5, 33, 36, 41, 61, 67, 77, 89, 138–140, 143).

[BDK+20] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A side-channel resistant implementation of SABER. *IACR Cryptol. ePrint Arch.*, 2020:733, 2020. URL: https://eprint.iacr.org/2020/733 (cited on page 9).

[BDL+21] Shi Bai, Léo Ducas, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium, 2021. Available at https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-3-Submissions (cited on pages 41, 51, 61, 68, 87, 89).

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults. In Walter Fumy, editor, *Advances in Cryptology - EUROCRYPT '97*, volume 1233, pages 37–51. Springer, 1997 (cited on pages 7, 75, 81).

[BDP+15] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. SHA-3 standard: permutation-based hash and extendable-output functions. In 2015. URL: https://keccak.team/keccak_specs_summary.html (cited on page 3).

[BDV21] Michiel Van Beirendonck, Jan-Pieter D'Anvers, and Ingrid Verbauwhede. Analysis and comparison of table-based arithmetic to boolean masking. Cryptology ePrint Archive, Report 2021/067, 2021. https://eprint.iacr.org/2021/067 (cited on page 110).

[BGR+19] Aurélie Bauer, Henri Gilbert, Guénaël Renault, and Mélissa Rossi. Assessment of the Key-Reuse Resilience of NewHope. In *Cryptographers' Track at the RSA Conference*, pages 272–292. Springer, 2019 (cited on pages 92, 95).

[BGR+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking Kyber: first- and higher-order implementations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):173–214, 2021. https://eprint.iacr.org/2021/483 (cited on pages 9, 111, 124, 125, 139–143, 145–148).

[BMD+21] Andrea Basso, Jose Maria Bermudo Mera, Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Michiel Van Beirendonck, and Frederik Vercauteren. Saber: Mod-LWR based KEM (round 3 submission), 2021. https://www.esat.kuleuven.be/cosic/pqcrypto/saber/files/saberspecround3.pdf (cited on pages 5, 33, 41, 61, 70, 86, 87, 149).

[BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 719–737, Berlin, Heidelberg. Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-29011-4 (cited on page 3).

[BRvV22] Joppe W. Bos, Joost Renes, and Christine van Vredendaal. Post-Quantum cryptography with contemporary Co-Processors: beyond kronecker, Schönhage-Strassen & nussbaumer, Boston, MA, August 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/bos (cited on pages 5, 21, 24, 39, 41, 54, 67).

[BS20] Nina Bindel and John M. Schanck. Decryption failure is more likely after success. In *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020, Paris, France, April 15-17, 2020, Proceedings*, pages 206–225, 2020 (cited on page 124).

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski Jr., editor, *Advances in Cryptology - CRYPTO '97*, volume 1294, pages 513–525. Springer, 1997 (cited on pages 7, 75, 81).

[BSI] BSI. Migration zu Post-Quanten-Kryptografie - Handlungsempfehlungen des BSI. de (cited on page 1).

[BSI20a] BSI. Migration zu Post-Quanten-Kryptografie, 2020. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Krypto/Post-Quanten-Kryptografie.pdf (cited on pages 4, 19).

[BSI20b] BSI. Status of quantum computer development, 2020. URL: hhttps://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/Quantencomputer/P283_QC_Studie-V_1_2.html (cited on page 1).

[BSJ14] Ahmad Boorghany, Siavash Bayat Sarmadi, and Rasool Jalili. On constrained implementation of lattice-based cryptographic primitives and schemes on smart cards. Cryptology ePrint Archive, Paper 2014/514, 2014. URL: https://eprint.iacr.org/2014/514. https://eprint.iacr.org/2014/514 (cited on page 42).

[BZ10] Richard P. Brent and Paul Zimmermann. Modern computer arithmetic (version 0.5.1). *CoRR*, abs/1004.4710, 2010. arXiv: 1004.4710. URL: http://arxiv.org/abs/1004.4710 (cited on page 42).

[CCA+20] Sreeja Chowdhury, Ana Covic, Rabin Yu Acharya, Spencer Dupee, Fatemeh Ganji, and Domenic Forte. Physical Security in the Post-quantum Era: A Survey on Side-channel Analysis, Random Number Generators, and Physically Unclonable Functions. *arXiv preprint arXiv:2005.04344*, 2020. https://arxiv.org/abs/2005.04344 (cited on page 29).

[CDH+19] Cong Chen, Oussama Damba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, and Zhenfei Zhang. NTRU: algorithm specifications and supporting documentation. *Brown University and Onboard security company, Wilmington USA*, 2019 (cited on pages 41, 61, 71, 84, 85).

[CGT+15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from arithmetic to boolean masking with logarithmic complexity. In *Proceedings of FSE 2015*, pages 130–149, 2015 (cited on pages 110, 111, 131).

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between boolean and arithmetic masking of any order. In *Proceedings of CHES 2014*, pages 188–205, 2014 (cited on pages 110–112, 116, 117, 120, 124, 125, 131, 141, 142, 150).

[CHK+20] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. Ntt multiplication for ntt-unfriendly rings. Cryptology ePrint Archive, Report 2020/1397, 2020. https://eprint.iacr.org/2020/1397 (cited on page 24).

[CJR+99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO*, 1999 (cited on page 110).

[Cla12] Christophe Clavier. Attacking block ciphers. In Marc Joye and Michael Tunstall, editors, *Fault Analysis in Cryptography*, Information Security and Cryptography. Springer, 2012 (cited on page 81).

[Cor14] Jean-Sébastien Coron. Higher order masking of look-up tables. In *Proceedings of EUROCRYPT 2014*, pages 441–458, 2014 (cited on pages 111, 113, 114).

[Cor17] Jean-Sébastien Coron. High-order conversion from boolean to arithmetic masking. In *Proceedings of CHES 2017*, pages 93–114, 2017. Full version available at http://eprint.iacr.org/2017/252 (cited on page 111).

[Cra27] Richard E. Crandall. Method and Apparatus for Public Key Exchange in a Cryptographic System, US patent, 1992-10-27. https://patentimages.storage.googleapis.com/11/9b/b8/75aa2cab01785d/US5159632.pdf (cited on page 43).

[CT03] Jean-Sébastien Coron and Alexei Tchulkine. A new algorithm for switching from arithmetic to boolean masking. In *Proceedings of CHES 2003*, pages 89–97, 2003 (cited on pages 10, 110, 111).

[DDG+20] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. Lwe with side information: attacks and concrete security estimation. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, pages 329–358, Cham. Springer International Publishing, 2020 (cited on pages 10, 83, 84).

[Deb12] Blandine Debraize. Efficient and provably secure methods for switching from arithmetic to boolean masking. In Emmanuel Prouff and Patrick Schaumont, editors, *Proceedings of CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2012 (cited on page 110).

[DFA+20] Viet Ba Dang, Farnoud Farahmand, Michal Andrzejczak, Kamyar Mohajerani, Duc Tri Nguyen, and Kris Gaj. Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using Hardware and Software/Hardware Co-design Approaches. Cryptology ePrint Archive, Report 2020/795, 2020. https://eprint.iacr.org/2020/795 (cited on page 23).

[DFR18] Jintai Ding, Scott Fluhrer, and Saraswathy Rv. Complete Attack on RLWE Key Exchange with Reused Keys, Without Signal Leakage. In *Australasian Conference on Information Security and Privacy*, pages 467–486. Springer, 2018 (cited on page 92).

[DGJ+19] Jan-Pieter D'Anvers, Qian Guo, Thomas Johansson, Alexander Nilsson, Frederik Vercauteren, and Ingrid Verbauwhede. Decryption failure attacks on IND-CCA secure lattice-based schemes. In *Public-Key Cryptography - PKC 2019 - 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part II*, pages 565–598, 2019 (cited on page 124).

[DNR04] Cynthia Dwork, Moni Naor, and Omer Reingold. Immunizing encryption schemes from decryption errors. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, pages 342–360, 2004 (cited on page 112).

[DRV20] Jan-Pieter D'Anvers, Mélissa Rossi, and Fernando Virdia. (One) failure is not an option: bootstrapping the search for failures in lattice-based encryption schemes. In *Advances in Cryptology - EUROCRYPT 2020 - Part III*, pages 3–33, 2020 (cited on page 124).

[EFG+17] 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, 2017 (cited on page 29).

[EFG+21] Thomas Espitau, Pierre-Alain Fouque, François Gérard, Mélissa Rossi, Akira Takahashi, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu. Mitaka: a simpler, parallelizable, maskable variant of falcon. Cryptology ePrint Archive, Report 2021/1486, 2021. https://ia.cr/2021/1486 (cited on page 9).

[FBR+21] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked accelerators and instruction set extensions for post-quantum cryptography. *IACR Cryptol. ePrint Arch.*:479, 2021. URL: https://eprint.iacr.org/2021/479 (cited on page 140).

[fCry18] Chinese Association for Cryptography Research. National cryptographic algorithm design competition, 2018. Available at https://www.cacrnet.org.cn/site/content/838.html (cited on page 1).

[fCry20a] Chinese Association for Cryptography Research. Lac won first prize of the national cryptographic algorithm design competition, 2020. Available at https://m.cacrnet.org.cn/site/content/854.html (cited on page 10).

[fCry20b] Chinese Association for Cryptography Research. Lac won first prize of the national cryptographic algorithm design competition, 2020. Available at https://m.cacrnet.org.cn/site/content/854.html (cited on page 92).

[Flu16] Scott Fluhrer. Cryptanalysis of ring-LWE based key exchange with key share reuse. Cryptology ePrint Archive, Report 2016/085, 2016. Available at https://eprint.iacr.org/2016/085 (cited on page 92).

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pages 537–554, 1999 (cited on pages 91, 92, 138–140).

[GG03] Joachim Von Zur Gathen and Jurgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, USA, 2nd edition, 2003. ISBN: 0521826462 (cited on pages 17, 42).

[GJY19] Qian Guo, Thomas Johansson, and Jing Yang. A Novel CCA Attack using Decryption Errors against LAC. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 82–111. Springer, 2019 (cited on page 92).

[GKS20] Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Compact dilithium implementations on cortex-m3 and cortex-m4. Cryptology ePrint Archive, Report 2020/1278, 2020. https://eprint.iacr.org/2020/1278 (cited on page 24).

[GM94] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. *SIGPLAN Not.*, 29(6):61–72, June 1994. ISSN: 0362-1340. DOI: 10.1145/773473.178249. URL: https://doi.org/10.1145/773473.178249 (cited on page 41).

[Gou01] Louis Goubin. A sound method for switching between boolean and arithmetic masking. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 2001 (cited on pages 110, 111, 117, 120, 125).

[Har07] David Harvey. Faster polynomial multiplication via multipoint kronecker substitution, 2007. arXiv: 0712.4046 [cs.SC] (cited on page 24).

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I*, pages 341–371, 2017 (cited on pages 122, 138, 139).

[HHP+21] Mike Hamburg, Julius Hermelink, Robert Primas, Simona Samardjiska, Thomas Schamberger, Silvan Streit, Emanuele Strieder, and Christine van Vredendaal. Chosen Ciphertext k-Trace Attacks on Masked CCA2 Secure Kyber. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):88–113, 2021. DOI: 10.46586/tches.v2021.i4.88-113. URL: https://doi.org/10.46586/tches.v2021.i4.88-113 (cited on page 8).

[HV20] Loïs Huguenin-Dumittan and Serge Vaudenay. Classical Misuse Attacks on NIST Round 2 PQC. In *International Conference on Applied Cryptography and Network Security*, pages 208–227. Springer, 2020 (cited on pages 92, 100).

[HvD21] David Harvey and Joris van Der Hoeven. Integer multiplication in time O(n log n). *Annals of Mathematics*, March 2021. DOI: 10.4007/annals.2021.193.2.4. URL: https://hal.archives-ouvertes.fr/hal-02070778 (cited on page 17).

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: securing hardware against probing attacks. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, pages 463–481, 2003 (cited on pages 9, 78, 79).

[Jac73] DH. Jacobsohn. A combinatoric division algorithm for fixed-integer divisors, 1973 (cited on page 41).

[JT12] Marc Joye and Michael Tunstall, editors. *Fault Analysis in Cryptography*. Information Security and Cryptography. Springer, 2012 (cited on page 81).

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99*, volume 1666, pages 388–397. Springer, 1999 (cited on pages 7, 75, 81).

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96*, volume 1109, pages 104–113. Springer, 1996 (cited on pages 7, 75, 81).

[Kon10] Yinan Kong. Optimizing the improved barrett modular multipliers for public-key cryptography, December 2010. DOI: 10.1109/CISE.2010.5676912 (cited on page 42).

[KPP20] Matthias J. Kannwischer, Peter Pessl, and Robert Primas. Single-trace attacks on keccak. Cryptology ePrint Archive, Paper 2020/371, 2020. URL: https://eprint.iacr.org/2020/371. https://eprint.iacr.org/2020/371 (cited on page 108).

[Kro82] L. Kronecker. Grundzüge einer arithmetischen theorie der algebraischen grössen. (abdruck einer festschrift zu herrn e. e. kummers doctor-jubiläum, 10. september 1881.). ger. *Journal für die reine und angewandte Mathematik*, 92:1–122, 1882. URL: http://eudml.org/doc/148487 (cited on pages 5, 19, 56).

[KY11] Abdel Alim Kamal and Amr M. Youssef. Fault analysis of the ntruencrypt cryptosystem. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 94-A(4):1156–1158, 2011 (cited on page 82).

[KY13] Abdel Alim Kamal and Amr M. Youssef. Strengthening hardware implementations of ntruencrypt against fault analysis attacks. *J. Cryptogr. Eng.*, 3(4):227–240, 2013 (cited on page 82).

[LN16] Patrick Longa and Michael Naehrig. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. In *International Conference on Cryptology and Network Security*, pages 124–139. Springer, 2016 (cited on page 36).

[Lom16] Victor Lomne. CHES Tutorial: Common Criteria Certification of a Smartcard: a Technical Overview, 2016. https://iacr.org/workshops/ches/ches2016/presentations/CHES16-Tutorial1.pdf (cited on page 18).

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings*, pages 1–23, 2010 (cited on page 138).

[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 (cited on page 3).

[LS15a] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015 (cited on page 3).

[LS15b] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Des. Codes Cryptogr.*, 75(3):565–599, 2015 (cited on page 139).

[Lum13] Jérémie O. Lumbroso. Optimal discrete uniform generation from coin flips, and applications. *CoRR*, abs/1304.1916, 2013. arXiv: 1304.1916. URL: http://arxiv.org/abs/1304.1916 (cited on page 77).

[LZZ18] Chao Liu, Zhongxiang Zheng, and Guangnan Zou. Key Reuse Attack on NewHope Key Exchange Protocol. In *International Conference on Information Security and Cryptology*, pages 163–176. Springer, 2018 (cited on page 92).

[MAA+20] Dustin Moody, Gorjan Alagic, Daniel C Apon, David A Cooper, Quynh H Dang, John M Kelsey, Yi-Kai Liu, Carl A Miller, Rene C Peralta, Ray A Perlner, Angela Y Robinson, Daniel C Smith-Tone, and Jacob Alperin-Sheriff. Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process. Technical report, National Institute of Standards and Technology, July 2020. URL: https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf (cited on pages 1, 51).

[MGT+19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking Dilithium - efficient implementation and side-channel evaluation. In *Applied Cryptography and Network Security - 17th International Conference, ACNS 2019, Bogota, Colombia, June 5-7, 2019, Proceedings*, pages 344–362, 2019 (cited on page 9).

[Mona] Simon Montoya. LAC attack. https://github.com/ayotnomis/LACAttack (cited on pages 93, 100, 106).

[Monb] Simon Montoya. SEA PQC Script. https://github.com/paper16FDTC2021/SEAPQC (cited on page 84).

[Mon85] Peter L Montgomery. Modular Multiplication Without Trial Division, 1985 (cited on pages 18, 42).

[Moo16] Dustin Moody. Post-Quantum Cryptography NIST's Plan for the Future, 2016. Available at https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/pqcrypto-2016-presentation.pdf (cited on pages 1, 2).

[MOP08] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*, volume 31. Springer Science & Business Media, 2008 (cited on pages 9, 29, 76).

[MU10] Alfred Menezes and Berkant Ustaoglu. On Reusing Ephemeral Keys in Diffie-Hellman Key Agreement Protocols. *IJACT*, 2(2):154–158, 2010 (cited on page 92).

[MVV18] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of Applied Cryptography*. CRC press, 2018 (cited on pages 42, 63).

[NDJ21] Kalle Ngo, Elena Dubrova, and Thomas Johansson. Breaking Masked and Shuffled CCA Secure Saber KEM by Power Analysis. In Chip-Hong Chang, Ulrich Rührmair, Stefan Katzenbeisser, and Debdeep Mukhopadhyay, editors, *ASHES@CCS 2021: Proceedings of the 5th Workshop on Attacks and Solutions in Hardware Security, Virtual Event, Republic of Korea, 19 November 2021*, pages 51–61. ACM, 2021. DOI: 10.1145/3474376.3487277. URL: https://doi.org/10.1145/3474376.3487277 (cited on page 8).

[Nus82] Henri J. Nussbaumer. *Number Theoretic Transforms*. In *Fast Fourier Transform and Convolution Algorithms*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1982, pages 211–240. ISBN: 978-3-642-81897-4. DOI: 10.1007/978-3-642-81897-4_8. URL: https://doi.org/10.1007/978-3-642-81897-4_8 (cited on pages 18, 65).

[OSP+16] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-Secure and Masked Ring-LWE Implementation. Cryptology ePrint Archive, Report 2016/1109, 2016. https://eprint.iacr.org/2016/1109 (cited on page 92).

[OSP+18a] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-Secure and Masked Ring-LWE Implementation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*:142–174, 2018 (cited on page 29).

[OSP+18b] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-secure and masked ring-lwe implementation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):142–174, 2018 (cited on page 111).

[PG12] Thomas Pöppelmann and Tim Güneysu. Towards Efficient Arithmetic for Lattice-Based Cryptography on Reconfigurable Hardware. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology – LATINCRYPT 2012*, pages 139–158, Berlin, Heidelberg. Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-33481-8 (cited on page 41).

[PP21] Peter Pessl and Lukas Prokop. Fault attacks on cca-secure lattice kems, 2021. https://ia.cr/2021/064 (cited on pages 8, 81, 82).

[QCD19] Yue Qin, Chi Cheng, and Jintai Ding. An Efficient Key Mismatch Attack on the NIST Second Round Candidate Kyber. *IEEE*, 2019 (cited on page 92).

[RdCR+16] Oscar Reparaz, Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Additively Homomorphic Ring-LWE Masking. In *Post-Quantum Cryptography*, pages 233–244. Springer, 2016 (cited on page 29).

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6), September 2009. ISSN: 0004-5411. DOI: 10.1145/1568318.1568324. URL: https://doi.org/10.1145/1568318.1568324 (cited on pages 2, 75).

[RJH+18] Prasanna Ravi, Mahabir Prasad Jhanwar, James Howe, Anupam Chattopadhyay, and Shivam Bhasin. Side-channel Assisted Existential Forgery Attack on Dilithium-A NIST PQC candidate. 2018. https://eprint.iacr.org/2018/821 (cited on page 29).

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, pages 413–427, 2010 (cited on pages 9, 78, 113, 131).

[RRD+16] Oscar Reparaz, Sujoy Sinha Roy, Ruan De Clercq, Frederik Vercauteren, and Ingrid Verbauwhede. Masking Ring-LWE. *Journal of Cryptographic Engineering*, 6(2):139–153, 2016 (cited on page 29).

[Sak11] Kazue Sako. *Semantic security*. In *Encyclopedia of Cryptography and Security*. Henk C. A. van Tilborg and Sushil Jajodia, editors. Springer US, Boston, MA, 2011, pages 1176–1177. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_23. URL: https://doi.org/10.1007/978-1-4419-5906-5_23 (cited on pages 10, 91).

[SB20] Sujoy Sinha Roy and Andrea Basso. High-speed Instruction-set Coprocessor for Lattice-based Key Encapsulation Mechanism: Saber in Hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):443–466, August 2020. DOI: 10.13154/tches.v2020.i4.443-466. URL: https://tches.iacr.org/index.php/TCHES/article/view/8690 (cited on page 23).

[Sch82] A. Schönhage. Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coeficients. In *EUROCAM*, 1982 (cited on page 19).

[Sha79] Adi Shamir. Factoring numbers in o(log n) arithmetic steps. *Inf. Process. Lett.*, 8(1):28–31, 1979 (cited on page 119).

[Sho97] Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.*, 26(5):1484–1509, October 1997. ISSN: 0097-5397. URL: https://doi.org/10.1137/S0097539795293172 (cited on page 1).

[Sol11] Jerome A. Solinas. *Generalized mersenne prime*. In *Encyclopedia of Cryptography and Security*. Springer US, Boston, MA, 2011, pages 509–510. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_32. URL: https://doi.org/10.1007/978-1-4419-5906-5_32 (cited on pages 42, 43).

[Sol99] Jerome A. Solinas. Generalized Mersenne Numbers. Technical report, Dept. of C&O, University of Waterloo, 1999. Available at https://cacr.uwaterloo.ca/techreports/1999/corr99-39.pdf (cited on pages 42, 43).

[SPO+19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In *Public-Key Cryptography - PKC 2019 - 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part II*, pages 534–564, 2019 (cited on pages 79, 80, 110–112, 116, 117, 126, 131).

[VOG+18] Felipe Valencia, Tobias Oder, Tim Güneysu, and Francesco Regazzoni. Exploring the vulnerability of R-LWE encryption to fault attacks. In John Goodacre, Mikel Luján, Giovanni Agosta, Alessandro Barenghi, Israel Koren, and Gerardo Pelosi, editors, *Proceedings of the Fifth Workshop on Cryptography and Security in Computing Systems, CS2 2018*, pages 7–12. ACM, 2018 (cited on page 82).

[VPR19] Felipe Valencia, Ilia Polian, and Francesco Regazzoni. Fault sensitivity analysis of lattice-based post-quantum cryptographic components. In Dionisios N. Pnevmatikatos, Maxime Pelcat, and Matthias Jung, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation SAMOS 2019*, volume 11733. Springer, 2019 (cited on page 82).

[WBB+19] Guillaume Wafo-Tapa, Slim Bettaieb, Loic Bidoux, Philippe Gaborit, and Etienne Marcatel. A practicable timing attack against hqc and its countermeasure. Cryptology ePrint Archive, Report 2019/909, 2019. https://ia.cr/2019/909 (cited on page 108).

[WGY20] Bin Wang, Xiaozhuo Gu, and Yingshan Yang. Saber on ESP32. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *Applied Cryptography and Network Security*, pages 421–440, Cham. Springer International Publishing, 2020. ISBN: 978-3-030-57808-4 (cited on pages 5, 24, 39, 54, 67).

[XYD+19] Lu Xianhui, Liu Yamin, Jia Dingding, Xue Haiyang, He Jingnan, and Zhang Zhenfei. LAC: Lattice-based Cryptosystems, 2019. Available at https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions (cited on pages 10, 33, 91–93, 95, 100).

[YJ00] Sung-Ming Yen and Marc Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Trans. Computers*, 49, 2000 (cited on page 81).

[ZZY+20] Yihong Zhu, Min Zhu, Bohan Yang, Wenping Zhu, Chenchen Deng, Chen Chen, Shaojun Wei, and Leibo Liu. A High-performance Hardware Implementation of Saber Based on Karatsuba Algorithm. Cryptology ePrint Archive, Report 2020/1037, 2020. https://eprint.iacr.org/2020/1037 (cited on page 23).

Contributions

[BMR21] Luk Bettale, Simon Montoya, and Guénaël Renault. Safe-error analysis of post-quantum cryptography mechanisms - short paper-. In *18th Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2021, Milan, Italy, September 17, 2021*, pages 39–44. IEEE, 2021 (cited on pages 12, 81).

[CGM+21a] Jean-Sebastien Coron, François Gerard, Simon Montoya, and Rina Zeitoun. High-order table-based conversion alg. and masking lattice-based encryption, 2021 (cited on pages 13, 109).

[CGM+21b] Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. High-order polynomial comparison and masking lattice-based encryption. *IACR Cryptol. ePrint Arch.*:1615, 2021 (cited on pages 13, 109).

[GMR20] Aurélien Greuet, Simon Montoya, and Guénaël Renault. Attack on Lac Key Exchange In Misuse Situation. In *Cryptology and Network Security, CANS 2020*. Springer, 2020 (cited on pages 12, 91).

[GMR21] Aurélien Greuet, Simon Montoya, and Guénaël Renault. On using RSA/ECC coprocessor for ideal lattice-based key exchange. In Shivam Bhasin and Fabrizio De Santis, editors, *Constructive Side-Channel Analysis and Secure Design - 12th International Workshop, COSADE 2021, Lugano, Switzerland, October 25-27, 2021, Proceedings*, volume 12910 of *Lecture Notes in Computer Science*, pages 205–227. Springer, 2021 (cited on pages 12, 23, 41).

[GMV22a] Aurélien Greuet, Simon Montoya, and Clémence Vermeersch. Modular Polynomial Multiplication Using RSA/ECC Coprocessor, 2022 (cited on pages 12, 39).

[GMV22b] Aurélien Greuet, Simon Montoya, and Clémence Vermeersch. Quotient approximation modular reduction. Cryptology ePrint Archive, Paper 2022/411, 2022. URL: https://eprint.iacr.org/2022/411 (cited on pages 12, 39).

ECOLE DOCTORALE

Titre: Cryptographie basée sur les réseaux euclidiens pour les systèmes embarqués

Mots clés: Cryptographie post-quantique, systèmes embarqués, coprocesseur, attaques physiques, contremesures

Résumé: Un ordinateur quantique suffisamment puissant pour exécuter l'algorithme de Shor pourrait voir le jour et ainsi menacer la sécurité apportée par la cryptographie asymétrique actuellement déployée. En effet, un tel ordinateur pourrait casser l'ensemble de la cryptographie dont la sécurité repose sur le problème de factorisation ou du logarithme discret. Pour anticiper une telle menace, les agences gouvernementales ont commencé des processus de standardisation de cryptosystèmes résistants à la puissance quantique: la cryptographie *post-quantique*. La standardisation la plus suivie par la communauté internationale est celle du National Institute of Standards and Technology (NIST) lancée en 2016. Elle a pour objectif de définir les futurs standards post-quantiques en termes de *Key Encapsulation Mechanisms* (KEMs) et de signatures. En Juillet 2020, cette standardisation en est à son troisième tour de sélection avec sept candidats finalistes restants. Parmi les candidats restants, cinq basent leur sécurité sur des problèmes mathématiques reliés aux réseaux euclidiens. A l'heure actuelle, les cryptosystèmes basés sur les réseaux euclidiens présentent le meilleur compromis entre efficacité, sécurité et taille des clés.

Les standards ont pour objectif d'être déployés massivement et dans différents composants. Parmi eux, il y a les composants embarqués. Ces composants sont très répandus mais limités en terme de mémoire et de puissance de calcul. De plus, ils sont menacés par des attaques physiques, ce qui demande un ajout de sécurité supplémentaire. Ainsi, déployer les cryptosystèmes basés sur les réseaux euclidiens dans les composants embarqués est un véritable défi.

Dans cette thèse, nous nous intéressons au déploiement des cryptosystèmes basés sur les réseaux euclidiens dans le contexte des systèmes embarqués. Plus particulièrement, ceux présents lors de la standardisation du NIST. Dans un premier temps, nous optimisons ces schémas à l'aide des coprocesseurs asymétriques existants. Les optimisations proposées sont implémentées et évaluées sur un composant embarqué. Par la suite, nous nous intéressons à la sécurisation et l'évaluation de la résilience contre les attaques physiques de ces schémas. Nous proposons ainsi des nouvelles contremesures de masquage applicables à plusieurs KEMs. De plus, nous évaluons les implémentations de certains cryptosystèmes contre des attaques par injection de fautes et des attaques par canaux auxiliaires.

Title: Embedded lattice-based cryptography

Keywords: Post-quantum cryptography, embedded devices, coprocessor, physical attacks, countermeasures

Abstract: A quantum computer powerful enough to run Shor's algorithm could emerge and thus threaten the security provided by the currently deployed asymmetric cryptography. Such a computer can break the entire cryptography based on the hardness of integer factorization or discrete logarithm. In order to anticipate this threat, national agencies have initiated standardization processes of quantum-safe cryptography: the *post-quantum* cryptography. The most followed standardization by the international community is the one of the National Institute of Standards and Technology (NIST) launched in 2016. This one aims to determine the future *Key Encapsulation Mechanisms* (KEMs) and signatures standards. In july 2020, the third round of selection of this standardization started with seven finalists cryptosystems remaining. Among these candidates, five are based on lattice problems. Currently, lattice-based cryptography provides the best trade-off between efficiency, security and compactness.
The future post-quantum standards will be deployed on several devices, like the embedded ones. These components are widely used but they are very limited in terms of memory and computing power. Moreover, they are threatened by physical attacks, which requires additional security. Therefore, implement lattice-based cryptography in such devices is a real challenge.
In this thesis we focus on the deployment of lattice-based cryptography in embedded devices. More precisely, we are interested in the lattice-based cryptosystems introduced during the NIST standardization. In a first step, we optimize these schemes by re-purposing existing asymmetric coprocessors. Such optimizations are then implemented and assessed on an embedded device. In a second step, we investigate the security and the resilience of the lattice-based schemes against physical attacks. To do so, we introduce new masking countermeasures which find applications to several KEMs. Moreover, we assess the security of several implementations against fault injection or side-channel attacks.

Institut Polytechnique de Paris
91120 Palaiseau, France