*T*owards efficient and practical h*o*m*o*m*o*rphic arithmetics

Loris Bergerat

Normandie Université

UNIVERSITÉ CAEN NORMANDIE

THÈSE

Pour obtenir le diplôme de doctorat

Spécialité **INFORMATIQUE**

Préparée au sein de l'**Université de Caen Normandie**

Towards Efficient and Practical Homomorphic Arithmetics

Présentée et soutenue par

BERGERAT LORIS

Thèse soutenue le 12/11/2025

devant le jury composé de :

MME ROUX-LANGLOIS ADELINE	Directeur de recherche au CNRS - UCN - Université de Caen Normandie	Directeur de thèse
MME FONTAINE CAROLINE	Directeur de recherche au CNRS - École normale supérieure Paris-Saclay	Président du jury
M. ORFILA JEAN-BAPTISTE	Ingénieur - Zama	Co-directeur de thèse
M. ALBRECHT MARTIN	Professeur - King's college London	Membre du jury
M. STEHLÉ DAMIEN	Chercheur - Cryptolab	Membre du jury
M. RIVAIN MATTHIEU	Chercheur HDR - CryptoExpert	Rapporteur du jury
MME VERBAUWHEDE INGRID	Professeur - LEUVEN - KATHOLIEKE UNIVERSITEIT	Rapporteur du jury

Thèse dirigée par **ROUX-LANGLOIS ADELINE** (Groupe de recherche en informatique, image et instrumentation de Caen) et **ORFILA JEAN-BAPTISTE** (Zama)

MIIS

GREYC
Laboratoire de recherche en sciences du numérique

Towards Efficient and Practical Homomorphic Arithmetics

Contents

Résumé en Français

La cryptographie est une science visant à assurer la sécurité de l'information et des communications. L'un des plus anciens cryptosystèmes documentés est le chiffrement de César, datant du premier siècle avant J.-C., qui consiste en une permutation de l'alphabet.

Au fil du temps, la cryptographie a évolué, passant de simples chiffrements par permutation aux systèmes cryptographiques modernes utilisés quotidiennement par des milliards de personnes à travers le monde. Si la cryptographie a été initialement développée pour des usages militaires ou diplomatiques, elle n'est désormais plus restreinte à ces domaines. De nos jours, les données numériques circulent en continu sur des réseaux non fiables, faisant de la cryptographie une science fondamentale pour la protection de la vie privée et des communications quotidiennes où elle joue un rôle crucial dans la garantie de la confidentialité, de l'authenticité et de l'intégrité des messages. La confidentialité assure qu'aucune information n'est divulguée à des entités malveillantes, l'authenticité, quant à elle, garantit que la communication s'effectue bien avec les entités prévues, et enfin, l'intégrité s'assure que les données n'ont pas été modifiées par un tiers. La cryptographie est présente partout dans notre vie quotidienne. Elle est notamment intégrée dans de nombreux appareils tels que les smartphones, les montres connectées et les ordinateurs. Elle est aussi utilisée pour sécuriser des services comme les banques et les communications numériques. Des schémas cryptographiques robustes sont donc essentiels. Leur sécurité repose sur des problèmes mathématiques difficiles ainsi que sur des algorithmes cryptographiques, tous deux fondés sur des modélisations et des preuves mathématiques, qui assurent qu'un schéma cryptographique satisfait les propriétés de sécurité souhaitées. Nous pouvons ainsi distinguer deux grands paradigmes de chiffrement: le chiffrement symétrique et le chiffrement asymétrique.

Le chiffrement symétrique, également appelé cryptographie à clé secrète, désigne les schémas dans lesquels la même clé secrète est utilisée à la fois pour le chiffrement et pour le déchiffrement. Dans ce contexte, deux parties partagent la même clé, ce qui permet d'assurer une communication sécurisée et de protéger les données. Le chiffrement symétrique est par exemple employé dans le chiffrement de disque, qui permet à un utilisateur de stocker et d'accéder à ses propres données en toute sécurité. Ces schémas sont généralement très efficaces pour chiffrer de grandes quantités de données et reposent sur des clés plus courtes que celles utilisées dans les schémas asymétriques. Le principal inconvénient du chiffrement symétrique réside dans le fait que les deux parties doivent avoir accès à la même clé secrète. La distribution sécurisée de cette clé n'est pas une tâche triviale, et ce problème est généralement résolu grâce au recours au chiffrement asymétrique.

Le chiffrement asymétrique, également appelé cryptographie à clé publique, désigne les schémas qui utilisent une paire de clés. La première, la clé publique, est accessible à tous et sert à chiffrer les messages, tandis que la seconde, la clé privée, n'est connue que du destinataire et lui permet de déchiffrer les textes chiffrés. Comme mentionné précédemment, les schémas asymétriques peuvent être utilisés pour partager en toute sécurité une clé symétrique entre plusieurs parties: ce processus est appelé échange de clés. Le premier protocole de chiffrement asymétrique fut un protocole d'échange de clé créé par Diffie et Hellman en 1976 [DH76], suivi de l'algorithme RSA, développé par Rivest, Shamir et Adleman en 1978 [RSA78].

Avènement de l'ordinateur quantique. La cryptographie moderne repose aujourd'hui sur des problèmes mathématiquement difficiles, c'est-à-dire des problèmes qu'il est pratiquement impossi-

ble de résoudre sans disposer d'informations auxiliaires, comme une clé secrète. La cryptographie à clé publique s'appuie principalement sur deux grandes familles de problèmes: le problème de la factorisation, qui consiste à retrouver les facteurs premiers p et q d'un produit $N = p \cdot q$ et le problème du logarithme discret, qui consiste à retrouver l'exposant x tel que $a = b^x$ pour des a et b donnés dans un groupe cyclique. Actuellement, ces problèmes sont encore considérés comme difficiles, et réussir à les résoudre efficacement compromettrait tous les schémas cryptographiques qui en dépendent.

En 1994, Peter Shor a introduit un nouvel algorithme quantique, aujourd'hui connu sous le nom d'algorithme de Shor [Sho94], permettant de résoudre efficacement les problèmes difficiles mentionnés ci-dessus à l'aide d'un ordinateur quantique suffisamment puissant. Grâce à cet algorithme, la factorisation ou le calcul de logarithmes discrets pourraient être effectués en temps polynomial, alors qu'actuellement ces problèmes nécessitent un temps exponentiel. Cela réduirait ainsi la durée d'attaques, d'un temps impraticable à l'échelle humaine à seulement quelques jours ou heures [GE21]. En 1994, les ordinateurs quantiques n'étaient encore qu'un concept théorique, et ce type d'algorithmes n'avait donc aucun impact sur la sécurité des schémas basés sur ces problèmes, tels que RSA [RSA78] ou l'échange de clés de Diffie–Hellman [DH76]. Au cours de la dernière décennie, de grandes entreprises comme Intel, Microsoft, IBM et Google ont massivement investi dans la recherche quantique, transformant progressivement cette technologie de la théorie en réalité. Même si nous en sommes encore aux premiers stades, la menace posée par de tels algorithmes doit être prise au sérieux. La cryptographie du futur doit s'adapter et reposer sur des problèmes considérés comme résistants aux ordinateurs quantiques.

Au regard de la menace posée par les ordinateurs quantiques, le National Institute of Standards and Technology (NIST) [NIS] a lancé en 2016 un processus de normalisation de la cryptographie post-quantique. Au départ, 69 schémas de chiffrement et de signature étaient en compétition. Finalement, seuls quatre ont été retenus, dont trois reposent sur des problèmes liés aux réseaux (lattices): Kyber [BDK+18], Dilithium [DKL+18], Falcon [PFH+20]. Ces résultats font de la cryptographie basée sur les réseaux un successeur post-quantique particulièrement prometteur. Aujourd'hui, de nombreuses constructions cryptographiques reposent sur les réseaux, notamment le chiffrement public, les protocoles d'échange de clés et le chiffrement homomorphe.

Chiffrement Totalement Homomorphe

Le chiffrement totalement homomorphe (Fully Homomorphic Encryption, FHE) désigne une famille de schémas de chiffrement permettant d'effectuer des calculs directement sur des données chiffrées. Cette propriété offre la possibilité à un tiers de manipuler des textes chiffrés tout en préservant la confidentialité des données sous-jacentes. Actuellement, l'un des principaux domaines ciblés par une telle technologie est l'apprentissage automatique (machine learning). Par exemple, des réseaux de neurones peuvent être utilisés pour prédire des maladies comme le cancer à partir de dossiers médicaux. Ces programmes aident la communauté médicale à améliorer la précision des diagnostics et à renforcer la détection précoce des maladies graves. Cependant, les hôpitaux ne peuvent pas partager directement des données médicales sensibles avec des tiers non fiables. C'est là que le chiffrement totalement homomorphe constitue une solution idéale: les données sensibles peuvent être chiffrées puis envoyées à un tiers, qui effectue des calculs de manière homomorphe sans rien apprendre des données. Les résultats chiffrés sont ensuite renvoyés à la communauté médicale, qui les déchiffre pour obtenir l'analyse, tout en préservant la confidentialité des patients. Un autre domaine émergent est l'utilisation du chiffrement totalement homomorphe dans la blockchain, où il permet d'effectuer des transactions sans divulguer d'informations, tout en garantissant que les opérations sont correctement exécutées. De plus, il trouve des applications dans de nombreux autres domaines, tels que le vote électronique ou le calcul multipartite sécurisé.

Le concept de chiffrement homomorphe a été introduit pour la première fois en 1978 par Rivest, Adleman et Dertouzos [RAD+78], qui ont posé la question de l'existence d'un schéma de chiffrement permettant de manipuler des données chiffrées sans rien révéler, et tel que le déchiffrement du texte chiffré manipulé retourne le même résultat que si l'on avait appliqué les opérations directement

sur les données en clair. Depuis, de nombreux schémas ont été développés pour satisfaire cette propriété, et un champ entier de la cryptographie s'est construit autour de ce concept.

On distingue trois grandes familles de schémas de chiffrement homomorphe, les schémas partiellement homomorphes (Partial Homomorphic Encryption, PHE), qui permettent un seul type d'opération (addition ou multiplication uniquement), les schémas presque homomorphes (Somewhat Homomorphic Encryption, SHE), qui autorisent l'addition et la multiplication mais en nombre limité, et enfin les schémas totalement homomorphes (Fully Homomorphic Encryption, FHE), qui permettent d'effectuer des opérations arbitraires sans contrainte de profondeur de calcul.

Dans ce travail, nous nous concentrons uniquement sur les schémas totalement homomorphes. Mais à titre d'exemple de schémas partiellement homomorphes, on peut citer RSA [RSA78] ou ElGamal [ElG85], qui permettent un nombre quelconque de multiplications entre textes chiffrés mais pas d'autres opérations, ce qui limite leur applicabilité. De même, le schéma de Paillier [Pai99] autorise un nombre quelconque d'additions entre textes chiffrés, mais ne permet pas d'effectuer d'autres opérations. Un des premiers schémas presque homomorphes a été proposé par Boneh, Goh et Nissim [BGN05], permettant l'addition et la multiplication, mais seulement un nombre restreint de fois.

La première solution pratique pour construire un schéma totalement homomorphe a été présentée en 2009, lorsque Craig Gentry publia son travail "Fully Homomorphic Encryption using Ideal Lattices" [Gen09]. Ce travail fondateur, comme beaucoup d'autres par la suite, base sa sécurité sur des éléments aléatoires, appelés bruit ou erreur, présents dans chaque texte chiffré. Or, effectuer des opérations sur des données chiffrées fait croître ce bruit. Une fois un certain seuil dépassé, le bruit accumulé empêche de correctement déchiffrer le message. Ainsi, sans contrôle de la croissance du bruit, aucun schéma dont la sécurité repose sur ces termes d'erreur ne peut prétendre être totalement homomorphe.

L'une des contributions majeures de Craig Gentry fut de résoudre la limitation liée à la croissance du bruit en proposant un nouvel algorithme appelé bootstrapping, qui réduit le bruit présent dans un texte chiffré. Cette technique permet, en théorie, d'évaluer un nombre illimité d'opérations sur des données chiffrées, rendant possible la construction de schémas totalement homomorphes basant leur sécurité sur des termes d'erreur présents dans chacun des textes chiffrés. Même si ce premier bootstrapping était extrêmement lent — prenant de 30 secondes à 30 minutes par opération [GH11] — il a ouvert la voie à de nombreux autres schémas de chiffrement totalement homomorphe et reste une pierre angulaire dans la conception des constructions modernes.

Les schémas totalement homomorphes reposent sur différents problèmes difficiles. Par exemple, le schéma proposé dans [VDGHV10] est basé sur le problème de l'approximation du plus grand commun diviseur (GCD), ceux de [LATV12, BIP$^+$22] reposent sur le problème NTRU (Nth-degree Truncated Polynomial Ring Unit) [HPS98]. De nos jours, les schémas FHE modernes s'appuient sur ce travail initial et reposent principalement sur l'hypothèse d'apprentissage avec erreurs (Learning With Errors, LWE) introduite dans [Reg05], ainsi que sur ses variantes, le Ring-LWE (RLWE) introduit dans [SSTX09, LPR10] ainsi que le General-LWE (GLWE) introduit dans [BGV12, LS15]. Il est important de noter que l'hypothèse LWE et ses variantes sont conjecturées résistantes aux attaques proposées par les ordinateurs quantiques. Par conséquent, tous les schémas cryptographiques qui s'appuient sur ces hypothèses sont considérés comme post-quantiques. Dans la même dynamique que le NIST, l'Organisation Internationale de Normalisation (ISO) [ISO] a entrepris, depuis 2023, la normalisation des principaux schémas de chiffrement homomorphe, incluant BGV [BGV12], B/FV [FV12, Bra12], CKKS [CKKS17], FHEW [DM17] et TFHE [CGGI16a].

Alors que tous ces schémas implémentent un algorithme de bootstrapping, dans le cas de BGV [BGV12], B/FV [FV12, Bra12] et CKKS [CKKS17], la stratégie consiste à l'éviter, car il demeure une opération trop coûteuse. Ces schémas adoptent alors une approche dite *leveled*, c'est-à-dire qu'ils utilisent des paramètres cryptographiques assez grands pour supporter un nombre fixe d'opérations. Cela implique de choisir des paramètres assurant que l'erreur reste suffisamment faible après ce nombre d'opérations. Cette stratégie est efficace dans certains cas, mais elle montre des limites: supporter un grand nombre d'opérations requiert d'augmenter les paramètres, ce qui

peut conduire à des tailles considérables. De plus, concevoir ces paramètres nécessite de connaître à l'avance le circuit à évaluer.

La deuxième stratégie pour évaluer un circuit homomorphe consiste à utiliser le bootstrapping comme opération centrale. Cette approche est principalement employée par FHEW [DM15] et TFHE [CGGI16a], où le bootstrapping a été fortement optimisé, passant de moins d'une seconde dans les premières implémentations de FHEW à seulement quelques millisecondes dans les versions récentes de TFHE. Cette amélioration provient essentiellement des optimisations introduites par TFHE dans une opération clé du bootstrapping. Comparé aux autres approches, le bootstrapping FHEW/TFHE est non seulement efficace mais également programmable, ce qui permet d'évaluer toute fonction univariée durant le processus. L'inconvénient majeur de cette approche réside toutefois dans l'espace de messages réduit, généralement limité à des valeurs inférieures à 8 bits [BBB+22]. Effectuer un bootstrapping sur des messages plus larges devient rapidement inefficace.

Même si le chiffrement totalement homomorphe, et le bootstrapping en particulier, ont connu des améliorations significatives en termes de performance au cours de la dernière décennie, ces algorithmes restent beaucoup plus lents que les opérations en clair. Ce manque d'efficacité demeure un frein majeur à une adoption massive des schémas totalement homomorphes, et la réduction de la latence globale reste un défi crucial. Dans ce manuscrit, nous nous concentrons exclusivement sur TFHE et présentons une étude détaillée de ce schéma, en introduisant plusieurs améliorations visant à réduire la latence et à lever certaines de ses principales limitations.

Nos contributions.

TFHE est bien connu pour l'efficacité de son bootstrapping, qui non seulement réduit le bruit mais permet également l'évaluation de fonctions univariées arbitraires, faisant ainsi du bootstrapping l'une des opérations centrales de ce schéma. Nativement, TFHE supporte des messages booléens ou de petits entiers (typiquement inférieurs à 8 bits), alors que l'informatique traditionnelle repose sur des types de données élémentaires tels que les entiers 32 bits, 64 bits ou les nombres à virgule flottante. De plus, les opérations homomorphes sont significativement plus lentes que leurs équivalents en clair, avec un surcoût généralement d'un facteur d'au moins 10^6.

Tout au long de ce manuscrit, nous cherchons à améliorer le schéma TFHE en répondant à la question centrale suivante:

Comment exploiter TFHE pour calculer homomorphiquement des types de données élémentaires, afin de réduire l'écart entre le calcul chiffré et le calcul en clair ?

Pour répondre à cette question générale, le manuscrit est structuré en deux parties distinctes, chacune dédiée à une sous-question:

Quelles stratégies peuvent être mises en œuvre pour accélérer les opérations de base de TFHE ?

Comment représenter et calculer efficacement des types de données élémentaires en utilisant TFHE ?

Avant d'apporter des réponses à ces questions, le Chapitre 2 introduit l'ensemble des concepts fondamentaux nécessaires à la compréhension du chiffrement totalement homomorphe et, plus particulièrement, de TFHE. Ce chapitre commence par présenter les problèmes difficiles qui garantissent la sécurité de TFHE. Il décrit ensuite les différents textes chiffrés utilisés dans TFHE, ainsi que les opérations de base du schéma. Il expose également certaines limitations connues de TFHE, telles que la taille réduite de l'espace des messages, la taille importante des éléments publics et des chiffrés ou encore l'efficacité limitée de certains algorithmes. Enfin, ce chapitre présente les techniques employées pour générer les différents ensembles de paramètres, indispensables afin de garantir à la fois la sécurité du schéma et la correction des opérations. Le Chapitre 3 se concentre

sur l'état de l'art, en présentant des algorithmes classiques et avancés qui répondent à certaines des limitations introduites dans le chapitre précédent.

Les Chapitres 4, 5 et 6 constituent la première partie de la thèse et visent à répondre à la première sous-question. Dans le Chapitre 4, nous présentons une amélioration du bootstrapping qui réduit le nombre d'opérations nécessaires à l'exécution de l'algorithme. Cette amélioration réduit la latence du bootstrapping ainsi que celle de l'évaluation globale de n'importe quel circuit homomorphe, ce qui en fait l'une des techniques les plus efficaces pour évaluer des textes chiffrés de précision moyenne (messages entre 6 et 10 bits). Nous montrons aussi comment cette procédure peut être améliorée en modifiant une autre partie du bootstrapping, et comment l'algorithme global peut être parallélisé efficacement.

Le Chapitre 5 propose deux distributions alternatives pour les clés secrètes. Comme ces clés impactent directement la sécurité de TFHE, nous fournissons une analyse démontrant comment les utiliser de manière sûre. Nous présentons également les avantages liés à ces nouvelles distributions, d'abord étudiées séparément, puis nous examinons les bénéfices offerts par la combinaison des deux distributions. Ces distributions offrent plusieurs atouts, permettant la conception de nouveaux algorithmes et une meilleure gestion de la propagation du bruit, ce qui conduit à une accélération globale de TFHE, qu'elles soient utilisées individuellement ou conjointement.

Ensuite, le Chapitre 6 propose un nouvel algorithme qui résout simultanément plusieurs limitations de TFHE. En particulier, cette opération se révèle efficace pour travailler avec des précisions plus élevées que d'ordinaire et permet d'évaluer des opérations multivariées. À l'issue d'une étude détaillée, menée selon une méthodologie rigoureuse, nous montrons que, pour des précisions importantes (supérieures à 9 bits), cette technique surpasse les approches de l'état de l'art présentées au Chapitre 3.

Enfin, les Chapitres 7 et 8 constituent la seconde partie de la thèse et visent à répondre à la deuxième sous-question. Contrairement aux chapitres précédents, davantage centrés sur l'amélioration d'opérations isolées et la résolution de limitations spécifiques, ces chapitres se concentrent sur les représentations et les encodages. Les améliorations étudiées dans les chapitres précédents peuvent être adaptées et utilisées dans ces nouvelles constructions.

Dans le Chapitre 7, nous étudions comment représenter efficacement de grands entiers avec TFHE, afin de correspondre aux types de données élémentaires tels que les entiers 32 bits et 64 bits. En raison des limitations de TFHE, il n'est pas possible d'encoder directement de grands entiers. Nous les divisons donc en parties plus petites, en utilisant soit une représentation en base (radix), soit le théorème des restes chinois (Chinese Remainder Theorem, CRT). Ces petits entiers sont ensuite chiffrés séparément, ce qui introduit de nouvelles contraintes, que nous surmontons en proposant de nouveaux algorithmes efficaces.

Enfin, le Chapitre 8 propose d'étendre le travail du chapitre précédent en introduisant des représentations homomorphes efficaces des nombres à virgule flottante. En particulier, nous présentons deux nouvelles méthodes. La première, basée sur les opérations du Chapitre 6, est très efficace pour de petites précisions, mais devient inefficace lorsque la précision augmente. La seconde, fondée sur la représentation du Chapitre 7, permet d'atteindre des précisions plus élevées. Comme au chapitre précédent, nous y présentons de nouveaux algorithmes afin de répondre de manière efficace aux contraintes introduites par ces deux nouvelles représentations.

Publications. Nous listons ici les contributions scientifiques et publications réalisées durant cette thèse et présentées dans ce manuscrit, ainsi que les chapitres auxquels elles sont associées.

[BBB+23] **Parameter Optimization and Larger Precision for (T)FHE.**
Co-auteurs: Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila et Samuel Tap. Publié dans le *Journal of Cryptology.* Présenté au Chapitre 6 et Chapitre 7.

[BCL+23] **New Secret Keys for Enhanced Performance in (T)FHE.**
Co-auteurs: Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, Adeline Roux-Langlois et Samuel Tap. Publié à la conférence CCS 2024. Présenté au Chapitre 5.

[BCL+25] **TFHE Gets Real: an Efficient and Flexible Homomorphic Floating-Point Arithmetic.**

Co-auteurs: Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila et Samuel Tap. Publié à la conférence CHES 2025. Présenté au Chapitre 8.

[BORT25] **Accelerating TFHE with Sorted Bootstrapping Techniques.**

Co-auteurs: Jean-Baptiste Orfila, Adeline Roux-Langlois et Samuel Tap. Publié à la conférence Asiacrypt 2025. Présenté au Chapitre 4.

Autre Publication. Durant cette thèse, nous avons également rédigé un article non inclus dans ce manuscrit: **Sharing the mask: TFHE bootstrapping on packed messages** [BBC+25]. Dans cet article, nous présentons un nouveau format de texte chiffré, nommé common mask ciphertext, qui repose sur l'utilisation d'un masque partagé permettant de chiffrer plusieurs messages sous différentes clés secrètes. Le premier avantage de ce type de texte chiffré réside dans la réutilisation du masque, ce qui permet une compression en éliminant le besoin d'un masque distinct pour chaque message chiffré. En plus de cet avantage, nous montrons que tous les textes chiffrés FHEW/TFHE connus, les éléments de clé publique et l'ensemble des opérations peuvent être naturellement étendus pour supporter ce format de masque commun. Enfin, à travers une série de benchmarks, nous démontrons que ce nouveau format de texte chiffré améliore significativement les performances dans le cadre des opérations amorties. Ce travail a été publié à la conférence CHES 2025.

Évolution du domaine. Depuis l'introduction des premières techniques de bootstrapping proposées par Gentry en 2009 [Gen09], qui ont démontré la faisabilité du chiffrement totalement homomorphe, de nombreux schémas FHE ont vu le jour. En moins d'une décennie, avec [DM15] puis [CGGI16a], la procédure de bootstrapping est passée de plus de 30 minutes (pour de larges valeurs de paramètres) à seulement quelques millisecondes. Même si l'algorithme est devenu beaucoup plus rapide au fil des années, évaluer un circuit homomorphe reste encore bien plus lent que d'effectuer les opérations correspondantes en clair. Pour rendre le FHE pratique, des efforts considérables ont été consacrés, au cours de la dernière décennie, à l'accélération du bootstrapping et, plus généralement, du chiffrement totalement homomorphe. Cette thèse a débuté à une période où la plupart des premières idées avaient déjà été largement explorées et optimisées. Cependant, au cours de ces trois années, nous avons réussi à découvrir de nouvelles techniques et algorithmes qui ont contribué à l'amélioration du domaine. De plus, puisque de nombreux schémas totalement homomorphes reposent sur les mêmes problèmes que TFHE, certaines des améliorations proposées pourraient être généralisées à d'autres schémas.

Au fil de ces trois années de doctorat, nous avons vu le chiffrement totalement homomorphe évoluer et s'améliorer, réduisant progressivement l'écart entre le calcul homomorphe et le calcul en clair. En particulier, nous avons observé les nouvelles possibilités offertes par les avancées de cette technologie. Comme mentionné dans l'introduction, le chiffrement totalement homomorphe a été initialement conçu pour des applications en apprentissage automatique et en informatique en nuage, permettant à des parties externes de calculer sur des données privées sans en révéler le contenu sensible. Plus récemment, la blockchain a également évolué, permettant l'intégration de schémas totalement homomorphes dans son infrastructure. Le chiffrement homomorphe rend alors possibles des transactions sur la blockchain sans divulguer d'information, tout en garantissant la validité et la bonne exécution des opérations. En rendant le FHE plus pratique, de nouvelles applications concrètes pourraient émerger, faisant du chiffrement totalement homomorphe une brique incontournable de la cryptographie moderne.

Enfin, le dernier sujet exploré au cours de ces trois années a été l'émergence d'un nouveau modèle de sécurité spécifiquement conçu pour le chiffrement totalement homomorphe. Ce modèle est similaire au modèle classique IND-CPA [BDPR98], mais dans ce nouveau modèle, nommé IND-CPA$^{\mathrm{D}}$ [LM21], l'adversaire dispose d'un oracle de déchiffrement. Un tel modèle permet de

mieux cibler les particularités du chiffrement totalement homomorphe ainsi que les attaques potentielles qui lui sont propres. Alors que le chiffrement totalement homomorphe est désormais considéré comme suffisamment mature pour résister aux attaques classiques, de nouveaux types d'attaques sont récemment apparus [CCP+24, LM21, CSBB24], tous fondés sur ce nouveau modèle. Ces attaques exploitent la présence de bruit dans les textes chiffrés et la possibilité qu'un échec lors du déchiffrement divulgue des informations sur le bruit et sur la clé secrète. Par conséquent, la gestion du bruit et la probabilité d'échec deviennent des aspects essentiels dans l'étude de la sécurité d'un schéma. Dans le même temps, ces nouvelles menaces ouvrent des pistes prometteuses pour la conception de contre-mesures efficaces et l'atténuation de leur impact.

Chapter 1

Preface

Cryptography is a science designed to secure information and communications. It has existed for centuries, and one of the oldest documented cryptosystems is Caesar's cipher, dating back to the first century BC, which consists of substituting each letter of a message with another.

Over time, cryptography has evolved from simple substitution ciphers to modern cryptographic systems used globally by billions of people every day. While it was originally developed for military or diplomatic purposes, cryptography is no longer restricted to such uses. In today's era, where digital data flows continuously across untrusted networks, it has become fundamental to individual privacy and everyday communication, playing a crucial role in ensuring confidentiality, authenticity, and integrity. Confidentiality ensures that no information is leaked to malicious entities, authenticity verifies that the communication takes place with the desired entities, and finally, integrity ensures that the data has not been modified by a third party. Cryptography can be found everywhere in our daily lives. It is embedded in devices such as smartphones, smartwatches, personal computers, and is used to secure services such as online banking and digital communications. Robust cryptographic schemes are therefore essential, and the security of such schemes relies on hard problems and cryptographic algorithms, both based on mathematical modeling and proofs, which ensure that a cryptographic scheme satisfies the desired security properties. From there, we can define two types of cryptographic encryption paradigm: symmetric encryption and asymmetric encryption.

Symmetric encryption, also known as private key cryptography, refers to encryption schemes where the same secret key is used for both encryption and decryption. In this context, two parties share the same secret key, which enables secure communication and also protects data, such as in disk encryption, allowing a user to securely store and access his own data. These schemes are typically very efficient for encrypting large amounts of data, and use smaller secret keys than asymmetric encryption schemes. The main disadvantage of such schemes is that both parties must have access to the same secret key. Securely distributing this key is not a trivial task and is generally solved by using asymmetric encryption.

Asymmetric encryption, also known as public key cryptography, is family of encryption that uses a pair of keys. The first, the public key, is publicly known and used to encrypt messages, while the second, the private key, is known only by the receiver of the message and allows them to decrypt ciphertexts. As mentioned before, asymmetric encryption schemes can be used to securely share a symmetric secret key between multiple parties: this process is called key exchange. The first asymmetric encryption protocol was the Diffie–Hellman key exchange [DH76], designed in 1976, followed by the RSA algorithm developed by Rivest, Shamir, and Adleman in 1978 [RSA78].

Advent of Quantum Computer. Modern cryptography is now based on mathematically hard problems, meaning problems that are difficult to solve without access to side information, such as the secret key. The most famous public key cryptography mainly relies on two families of hard problems: the factoring problem, which consists of recovering the prime factors p and q from a

product $N = p \cdot q$, and the discrete logarithm problem, which consists of recovering the exponent x such that $a = b^x$ for given a and b in a cyclic group. Currently these problems are still considered hard, and breaking one of these problems brings down all the cryptographic schemes relying on them.

In 1994, Peter Shor introduced a quantum algorithm, now known as Shor's algorithm [Sho94], that efficiently solves the aforementioned hard problems, assuming access to a sufficiently powerful quantum computer. With this algorithm, solving the factoring problem or the discrete logarithm problem could be done in polynomial time, whereas solving these problems currently requires exponential time, reducing the computation time of the attacks from impractical duration to only a few days or hours [GE21]. In 1994, quantum computers were still a theoretical concept, and these types of algorithms had no impact on the security of schemes based on these problems, such as RSA [RSA78] and the Diffie-Hellman key exchange [DH76]. Over the past decade, major corporations such as Intel, Microsoft, IBM and Google have massively invested in quantum computers, transforming this technology from theory into reality. Even if we still are in the early stages of the quantum computer, the threat of quantum algorithm needs to be taken seriously and future cryptography needs to be based on quantum-resistant problems.

As a consequence, the National Institute of Standards and Technology (NIST) [NIS] launched in 2016 a post-quantum cryptography standardization process. At the beginning, 69 encryption and signature schemes were listed to participate in this standardization, and finally, only 4 of them were accepted, and, 3 of them are based on lattice problems: Kyber [BDK+18], Dilithium [DKL+18], and Falcon [PFH+20]. These results make lattice-based cryptography a promising post-quantum successor to current cryptography. Nowadays, many constructions are lattice-based such as public encryption, key exchanges and homomorphic encryption.

1.1 Fully Homomorphic Encryption

Fully Homomorphic Encryption (FHE) refers to a family of encryption schemes that enable computations on encrypted data. This capability allows a third party to process ciphertexts while preserving the confidentiality of the underlying data. Currently, one of the main fields targeted by such a technology is machine learning. For instance, neural networks can be used to predict diseases such as cancer by analyzing medical records. These programs can assist the medical community in improving diagnostic accuracy and enhancing early detection of serious diseases. However, hospitals cannot share sensitive medical data with untrusted third parties. This is where Fully Homomorphic Encryption (FHE) becomes an ideal solution. With FHE, sensitive data can be encrypted and sent to a third party, who performs computations homomorphically without learning any information about the data. The encrypted results are then returned to the medical community, who decrypts them to obtain the analysis while preserving data privacy. Another nascent field is the use of FHE in blockchain, permitting on-chain transactions without revealing information, while still ensuring that operations are correctly executed. In addition, FHE can be applied in many other domains, such as electronic voting and multi-party computation.

The concept of homomorphic encryption was first introduced in 1978 by Rivest, Adleman, and Dertouzos [RAD+78], who raised the question of the existence of an encryption scheme where encrypted data could be manipulated without revealing any information, and such that the decryption of the manipulated ciphertext returns the same result as applying the operations on the plaintext. Since then, many schemes have been developed to satisfy this property, and an entire field of cryptography has emerged around this concept. We can distinguish three families of homomorphic encryption schemes: Partial Homomorphic Encryption (PHE) schemes, Somewhat Homomorphic Encryption (SHE) schemes and Fully Homomorphic Encryption (FHE) schemes. Partial homomorphic encryption schemes allow only a single type of operation (only addition or only multiplication). Then, somewhat homomorphic encryption schemes permit performing addition and multiplication but in a limited amount. Finally, FHE schemes support arbitrary operations without constraints on the computational depth.

In this work, we focus only on FHE schemes, but as examples of partial encryption schemes,

we can cite the RSA [RSA78] or the ElGamal [ElG85] cryptosystems, which allow any number of multiplications between ciphertexts but do not support other types of operations, limiting their applicability. Similarly, we can mention the Paillier [Pai99] cryptosystem, which allows any number of additions between ciphertexts but does not support other types of operations. One of the earliest somewhat homomorphic encryption schemes was proposed by Boneh, Goh, and Nissim [BGN05]. This scheme permits to perform addition and multiplication, but only a limited number of times.

The first solution proposed to create an FHE scheme was presented in 2009, when Gentry published his work "Fully homomorphic encryption using ideal lattices" [Gen09] in which he introduced the first FHE scheme. This primary work, like many that followed, based its security on small random terms, known as noise or error, present in each ciphertext. Therefore, performing operations on encrypted data cause this noise to grow. Once the noise reaches a certain threshold, too much noise is accumulated in the ciphertext, making correct decryption impossible. Consequently, without controlling the noise growth, no scheme whose security is based on errors terms can claim to be fully homomorphic. One of the most important contribution of this paper consists of solving this limitation by proposing a new algorithm, called bootstrapping, which reduces the noise present in a ciphertext. This technique theoretically enables the evaluation of an unlimited number of operations on encrypted data, making it possible to construct FHE schemes where security is based on a small error term. Even though this first bootstrapping was slow, ranging from 30 seconds to 30 minutes per bootstrapping [GH11], it paved the way for the creation of many other FHE schemes and has become a cornerstone in the design of modern FHE constructions.

FHE schemes are based on different hardness assumptions. For instance, the scheme proposed in [VDGHV10] is based on the approximate greatest common divisor problem, while those in [LATV12, BIP$^+$22] rely on the N^{th}-degree Truncated Polynomial Ring Unit (NTRU) problem [HPS98]. Nowadays, modern FHE schemes are based on this primary work and rely on the Learning With Error (LWE) assumption introduced in [Reg05] and its variant, the Ring Learning With Error (RLWE) assumption introduced in [SSTX09, LPR10] and the General Learning With Errors (GLWE) assumption introduced in [BGV12, LS15]. We note that, the LWE assumption and all its variants are conjectured to be post-quantum problems. Consequently, all cryptographic schemes based on this assumption are resilient against quantum computers. Following the same initiative as NIST, in 2023 the International Organization for Standardization (ISO) [ISO] began standardizing the main homomorphic encryption schemes, including BGV [BGV12], B/FV [FV12, Bra12], CKKS [CKKS17], FHEW [DM17], and TFHE [CGGI16a].

While all of these schemes implement a bootstrapping algorithm, in the case of BGV [BGV12], B/FV [FV12, Bra12], and CKKS [CKKS17], the strategy is to avoid it, as it remains a highly costly operation, even though these schemes can encrypt multiple messages within a single ciphertext and then bootstrap several encrypted messages with only one bootstrapping. These schemes then adopt a leveled approach, meaning they use cryptographic parameters large enough to support a fixed number of operations. This implies selecting cryptographic parameters which can guarantee an error small enough after performing the fixed number of homomorphic operations. These strategies can be applied in many use cases, but certain limitations are already apparent. Indeed, supporting a large number of operations requires increasing the cryptographic parameters to ensure a sufficiently small error, which can lead to significantly large parameter sizes. Moreover, designing such parameters requires prior knowledge of the circuit to be evaluated.

The second strategy for evaluating a homomorphic circuit follows a different approach by using bootstrapping as a core operation. This strategy is primarily employed by FHEW [DM15] and TFHE [CGGI16a], where bootstrapping is highly efficient, taking less than a second in the first implementations of FHEW [DM15] and only a few milliseconds in recent versions of TFHE [CGGI16a]. This improvement comes essentially from the improvements to FHEW introduced by TFHE, particularly in how a key operation is performed in the bootstrapping. Compared to other bootstrappings, FHEW/TFHE bootstrapping is not only efficient but also programmable, allowing any univariate function to be evaluated during the bootstrapping process. The main drawback of this approach is the small message space, which is generally limited to values smaller than 8 bits [BBB$^+$22]. Performing bootstrapping on larger messages quickly becomes inefficient and

impractical for higher precision.

Even if fully homomorphic encryption, and bootstrapping in particular, has become faster over the last decade, algorithms remain slow compared to cleartext operations. This lack of efficiency remains a major bottleneck to the massive adoption of FHE, and reducing the latency of operations remains a real challenge. In this manuscript, we focus exclusively on TFHE and present a detailed study of the scheme, introducing several improvements that reduce latency and solve several limitations of TFHE.

1.2 Our Contributions

TFHE is well known for its fast bootstrapping, which not only reduces noise but also enables the evaluation of arbitrary univariate functions, making bootstrapping one of the central operations. Natively, this scheme, supports boolean values and small integer messages (typically less than 8 bits), whereas traditional computing relies on standard data types such as 32-bit and 64-bit integers or floating-point numbers. Moreover, homomorphic operations are significantly slower than their plaintext counterparts, typically by a factor of at least 10^6. Throughout this manuscript, we aim to improve the TFHE scheme by addressing the following central question: :

How can TFHE be leveraged to homomorphically compute primitive data types, bridging the gap between encrypted and plaintext computation?

To answer this general question, the manuscript is structured into two distinct parts, each dedicated to exploring one of the following sub-questions:

What strategies can be employed to accelerate the core operations of TFHE?

How can we represent and efficiently compute primitive data types using TFHE?

Before answering these questions, Chapter 2 introduces all the foundational concepts necessary to understand fully homomorphic encryption and TFHE. This chapter starts by presenting the hard problems that guarantee the security of TFHE. Then, it presents the different ciphertexts used in TFHE based on these hard problems, along with the basic operations of the TFHE scheme. This chapter also introduces and explains some of the known limitations of the TFHE scheme, such as the small message space, the size of the public material, or the efficiency of certain algorithms. It also presents techniques used to generate the different parameter sets necessary to ensure security and correctness. Chapter 3 focuses on the state-of-the-art, presenting both classical and advanced algorithms that address some of the limitations introduced in the previous chapter.

Then, Chapters 4, 5 and 6 constitute the first part of the thesis and aim to answer the first sub-question. In Chapter 4, we present an improvement to the bootstrapping that reduces the number of operations required to execute the algorithm. This improvement reduces the latency of both the bootstrapping procedure and the overall evaluation of a homomorphic circuit, making it one of the most efficient bootstrapping techniques for evaluating medium-precision ciphertexts (messages between 6 and 10 bit) in TFHE. We also show how this new procedure can be improved by modifying another part of the bootstrapping, and how the overall algorithm can be efficiently parallelized.

Chapter 5 proposes two alternative distributions for the secret keys. As these secret keys affect the security of TFHE, we provide an analysis to demonstrate how to use them safely. We also present the different advantages of using them, first by studying them separately, and then by analyzing the impact of combining these two new secret key distributions. These new distributions offer multiple advantages, resulting in new algorithms and improved noise propagation, which leads to an overall speed-up of TFHE when such secret keys are used together or separately.

Then, Chapter 6 proposes a new algorithm solving several limitations of TFHE at the same time. In particular, this operation is efficient for working with higher precision than usual. After

a detailed study following a rigorous methodology, we show that, for large precisions (greater than 9 bits), this technique surpasses the previous state-of-the-art approaches introduced in Chapter 3.

Finally, Chapters 7 and 8 form the second part of the thesis and aim to answer the second sub-question. Compared to the previous chapters, which focused more on improving isolated operations and solving specific limitations, these chapters focus on working with different representations and encodings. The improvements studied in the previous chapters can easily be adapted and used in the following constructions.

In Chapter 7, we studied how to efficiently represent large integers using TFHE to match the primitive data types, such as 32- and 64-bit integers. Due to the limitations of TFHE, we cannot directly encode large integers, so we split these integers into smaller parts using either a radix or a Chinese Remainder Theorem (CRT) representation. These smaller integers are then each encrypted into separate ciphertexts, introducing new constraints that we overcome by proposing new efficient algorithms.

Finally, Chapter 8 proposes to extend the work presented in the previous chapter by introducing efficient homomorphic floating-point representations. Especially, we introduce two new methods. The first one is based on the operations presented in Chapter 6 and is very efficient for small precision but inefficient for higher precision. The second method is based on the representation presented in Chapter 7 and allows reaching higher precision. As in the previous chapter, we design new algorithms to efficiently address the new constraints introduced by these two representations.

Publications. Here, we list the research contributions and publications made during this thesis and presented in this manuscript, along with their associated chapters.

[BBB+23] **Parameter Optimization and Larger Precision for (T)FHE.**
Co-authors: Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Published in the *Journal of Cryptology*. Presented in Chapter 6 and Chapter 7.

[BCL+23] **New Secret Keys for Enhanced Performance in (T)FHE.**
Co-authors: Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, Adeline Roux-Langlois, and Samuel Tap. Published at the CCS 2024 conference. Presented in Chapter 5.

[BCL+25] **TFHE Gets Real: an Efficient and Flexible Homomorphic Floating-Point Arithmetic.**
Co-authors: Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Published at CHES 2025 conference. Presented in Chapter 8.

[BORT25]**Accelerating TFHE with Sorted Bootstrapping Techniques.**
Co-authors: Jean-Baptiste Orfila, Adeline Roux-Langlois, and Samuel Tap. Published at Asiacrypt 2025 conference. Presented in Chapter 4.

Other Works. I have also worked on another paper, named **Sharing the mask: TFHE bootstrapping on packed messages** [BBC+25], that is not included in this manuscript. In this paper, we present a new ciphertext format named common mask ciphertext, which consists of a shared mask and multiple message bodies. Each body encrypts a distinct message while reusing the same random mask. The first advantage of such ciphertexts is that they reuse the mask, allowing ciphertext compression by eliminating the need for one mask per encrypted message. In addition to the compression benefit, we show that all known FHEW/TFHE ciphertexts, public key materials and all operations can be naturally extended to support this common mask format.

Finally, through benchmarks, we demonstrate that this new ciphertext format leads to improved performance for amortized operations. This work will be published at the CHES 2025 conference.

Evolution of the Domain. Since the introduction of the first bootstrapping techniques proposed by Gentry in 2009 [Gen09], which demonstrated the feasibility of fully homomorphic encryption, many FHE schemes have emerged. In less than a decade, with [DM15] closely followed by [CGGI16a], the bootstrapping procedure went from more than 30 minutes (for large parameters) to only a few milliseconds. Even though the algorithm has become faster over the years, evaluating a circuit homomorphically remains much slower than performing operations on cleartext. To make FHE practical, significant effort has been focused over the last decade to accelerating bootstrapping and FHE more generally. This thesis began in a period where most straightforward ideas had already been thoroughly explored and optimized. However, during these three years, we managed to discover many new techniques and algorithms that contributed to further improving TFHE. Moreover, since the other FHE schemes are based on the same problems than TFHE, some of the proposed improvements might be used more generally in other FHE schemes.

During these three years of thesis work, we have seen fully homomorphic encryption evolve and improve, progressively bridging the gap between the homomorphic world and the cleartext domain. In particular, we have seen the new possibilities offered by the advancements in this technology. As presented in the introduction, FHE was initially designed for machine learning and cloud computing, allowing external parties to compute over private data without exposing any sensitive information. But recently, an emerging field integrating FHE schemes into its technology is blockchain, where FHE could enable on chain transactions without revealing any information, while still ensuring that operations are correctly executed. By making FHE more practical, new real-world applications may come to light, making FHE an essential solution for modern cryptography.

Finally, the last topic revealed during these three years was the emergence of a new model of security specifically designed for FHE. This new security model is similar to the classical IND-CPA model [BDPR98], but in this new model, named IND-CPA$^{\mathrm{D}}$ [LM21], the adversary has access to a decryption oracle. This new model allows better targeting of the capabilities of FHE and the underlying possible attacks. While fully homomorphic encryption is now considered mature enough to resist common attacks, we have seen the rise of new types of attacks, such as those in [CCP^{+}24, LM21, CSBB24], all based on this new model. All these attacks rely on the presence of noise inside ciphertexts and the possibility of decryption failure leaking some information about the noise or the secret key. Therefore, noise management and failure probability must be taken into account when studying the security of a scheme. At the same time, they open up promising directions for designing new countermeasures and mitigating the impact of such threats.

Part I
TFHE Background

This first part of the thesis presents the TFHE scheme.
Chapter 2 begins by establishing the essential mathematical background required to understand TFHE. It introduces the hard problems such as the Learning With Errors (LWE) and its variants, on which TFHE is based. This chapter also defines the ciphertext structures and the principal homomorphic operations, and finally presents the main limitations of TFHE.
Then, Chapter 3, based on the previous construction, goes deeper into the state-of-the-art algorithms that define the current capabilities of TFHE. It presents advanced algorithms that improve the efficiency and extend the range of applications of TFHE. It also serves as a reference for comparing our future improvements or new algorithms against the state-of-the-art.

Chapter 2

Understanding TFHE

As introduced in the previous chapter, many fully homomorphic encryption (FHE) schemes are lattice-based cryptosystems. This chapter presents the necessary mathematical background to define and understand the principles of FHE schemes, and then explores in more detail the Torus Fully Homomorphic Encryption (TFHE) scheme, which is the focus of this manuscript.

We first introduce the problems used in lattice-based FHE cryptography and briefly discuss the security of such schemes by presenting common attacks.

Then we focus in more detail on TFHE. Based on the hard problems introduced previously, we explain how to encode and encrypt a message. We present the different ciphertext types used in TFHE, mainly for data encryption and later for generating public material required for more complex operations. Finally, we show how to perform basic homomorphic operations such as addition, rotation, and multiplication by a constant, as well as more complex operations such as various forms of key switching, a central operation that enables changing the secret key of a ciphertext.

All these operations highlight one of the primary limitations of TFHE, and more broadly of FHE: the growth of noise during computation. As explained in the introduction, when noise becomes too large, decryption may fail, making noise management essential throughout the computation. This management is handled by the most important, yet most expensive, operation in FHE: the bootstrapping. After presenting all the necessary building blocks for this operation, we conclude by presenting the complete bootstrapping procedure.

To conclude the chapter, we present methodologies for parameter selection and explain how to homomorphically evaluate arbitrary computational graphs. Indeed, careful parameter selection is essential for enabling efficient circuit evaluation while maintaining low failure probability and the desired security level.

This chapter concludes by identifying the current limitations of TFHE that are studied and addressed in this manuscript.

2.1 Preliminaries

In this section, we present the notations used throughout the thesis and the mathematical background necessary to introduce TFHE and the improvements discussed in the following chapters. We then introduce the Learning With Errors (LWE) problem and its variants, which have been proven to be computationally hard to solve, and which form the fundamental basis of TFHE. Finally, we provide a general definition of what an FHE scheme is.

2.1.1 Notations and Mathematical Background

We use the notations $\mathbb{N}$ to represent the natural numbers, $\mathbb{Z}$ to represent the natural integers and $\mathbb{R}$ to represent the real numbers. We use the notation $\mathbb{N}^*$ (resp. $\mathbb{Z}^*$) to refer to $\mathbb{N} \setminus \{0\}$ (resp.

$\mathbb{Z} \setminus \{0\}$). For q a positive integer, we note $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ the quotient ring of integers modulo q. By convention, integers are denoted in lowercase x. For two integers $a \leq b$, we note by $[a, b]$ the closed interval, i.e., $[a, b] = \{k \in \mathbb{Z} : a \leq k \leq b\}$. We use $[a, b)$ to denote the interval $[a, b-1]$, $(a, b]$ for the interval $[a+1, b]$ and finally, (a, b) for the interval $[a+1, b-1]$. We write $[.]_q$ to denote the q modular reduction. The usual notations for the floor $\lfloor \cdot \rfloor$, the ceil $\lceil \cdot \rceil$ and the round $\lfloor \cdot \rceil$ functions are used.

Let N be a power of two, representing the degree of the quotient polynomial. Then, we note $\mathfrak{R}_{q,N} = \mathbb{Z}_q[X]/(X^N + 1)$ the quotient ring of polynomials modulo $X^N + 1$ with integer coefficients modulo q. When N is clear from the context, we omit it and denote the ring simply by $\mathfrak{R}_q$. Polynomials are noted in uppercase P or $P(X)$ such that $P(X) = P = \sum_{i=0}^{N-1} p_i X^i$ (We note that N is the only exception of this notation and represents a parameter, the degree of the polynomials). The vectors are written in bold $\boldsymbol{x}$, and can be detailed with the notation $(x_0, \ldots, x_k)$. Given two vectors of integers $\boldsymbol{a}$ and $\boldsymbol{b}$, we denote the dot product as $\langle \boldsymbol{a}, \boldsymbol{b} \rangle = \sum_i a_i \cdot b_i$. We use the notation $\{x_i\}_{i \in [0,k]}$ to denote the set $\{x_0, \ldots, x_k\}$.

Let $\mathfrak{D}$ be a probabilistic distribution. We denote $x \hookleftarrow \mathfrak{D}(D)$ as the random sampling according to the distribution $\mathfrak{D}$ from the support D. Specifically, we denote $\mathfrak{U}(.)$ as the uniform distribution and $\mathcal{N}_{\sigma^2}$ as the Gaussian distribution with a mean set to zero and a standard deviation set to σ. We denote $\mathsf{Var}(X)$ the variance and $\mathbb{E}(X)$ the expectation of a random variable X. Finally, we denote $\mathsf{card}(.)$ the cardinality of a given set.

We denote by λ the security level, meaning that an adversary needs to perform at least 2^λ operations to break a cryptographic primitive. Usually, to guarantee a long term security, λ is set to 128.

Definition 1 (Noise & Cost Model)**.** *FHE operators are associated with a noise model and an algorithmic cost model. A noise model is a formula used to model the noise evolution across an FHE operator. The algorithmic cost model of a homomorphic operation is used to estimate the cost of executing an algorithm. It is denoted by* $\mathsf{Cost}.$*, where an atomic operation is denoted by* $\mathbb{C}.$*.*

Definition 2 (Standard score)**.** *Let* $A \hookleftarrow \mathcal{N}(0, \sigma^2)$ *(centered normal distribution), let* $\mathsf{p_{fail}}$ *be a failure probability and let* erf *be the error function* $\mathsf{erf}(z) \mapsto \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} \mathsf{dt}$*. We define the* standard score z^* *for* $\mathsf{p_{fail}}$ *as* $z^*(\mathsf{p_{fail}}) = \sqrt{2} \cdot \mathsf{erf}^{-1}(1 - \mathsf{p_{fail}})$ *and we have:* $\mathbb{P}(A \notin (-z^*\sigma, z^*\sigma)) \leq \mathsf{p_{fail}}$*.*

Let $t \in \mathbb{R}$*, we have* $z^*(\mathsf{p_{fail}}) \cdot \sigma \leq t \Rightarrow \mathbb{P}(A \notin (-t, t)) \leq \mathsf{p_{fail}}$

Definition 3 (Lattices)**.** *A lattice* Λ *of dimension* n *is a discrete additive subgroup of* $\mathbb{R}^n$*, generated by a finite set of linearly independent vectors.*

$$\Lambda = \left\{ \sum_{i=1}^{k} a_i b_i \;\middle|\; a_i \in \mathbb{Z} \right\}$$

Where $\{b_1, \ldots, b_k\}$ *is a set of linearly independent vectors called a basis of the lattice, and* k *is the rank of the lattice. A full rank lattice in* $\mathbb{R}^n$ *is a lattice composed of* n *linearly independent vectors in* $\mathbb{R}^n$*; i.e.,* $k = n$*.*

Cyclotomic Polynomials. In lattice-based cryptography, which will be detailed later, we mostly use quotient rings of polynomials with special algebraic structures. These structures are based on cyclotomic polynomials and play a central role in ensuring the hardness assumptions on which many cryptosystems are based.

Definition 4 (Cyclotomic Polynomials)**.** *The* n^{th} *cyclotomic polynomials, denoted* Φ_n*, corresponds to the unique irreducible polynomial with integer coefficients that divides* $X^n - 1$ *but does not a divide* $X^k - 1$ *for any* $k < n$*.*

For efficiency reasons, we mainly focus on the cyclotomic polynomial $\Phi_{2N} = X^N + 1$, where N is a power of two. Indeed, the underlying quotient ring permits fast polynomial arithmetic via the Number Theoretic Transform (NTT) or Fast Fourier Transform (FFT). We note that using

the FFT adds a supplementary error due to the casting of the polynomials (64-bit integers) into floating-point numbers (double precision with a 53-bit mantissa). This error is analyzed later in Subsection 2.5.3.

Even if we only study algorithms based on the cyclotomic polynomial Φ_{2N}, with N a power of two, all the results can be adapted to any cyclotomic polynomial.

Remark 2.1 (Negacyclicity)**.** In this manuscript, we essentially work with the ring of negacyclic polynomials $\mathfrak{R}_{q,N} = \mathbb{Z}_q[X]/X^N + 1$ with N a power of two. This means that all polynomials are modulo $X^N + 1$. For any polynomials $P(X) = \sum_{i=0}^{N-1} p_i X^i$ in $\mathfrak{R}_{q,N}$, for any $k \mod 2N \in [0, N)$ we have:

$$P(X) \cdot X^k = \sum_{i=0}^{k-1} p_{N-k+i} \cdot X^i - \sum_{i=k}^{N-1} p_i \cdot X^i.$$

In particular, we have $P(X) = P(X) \cdot X^{2N}$ and $P(X) = -P(X) \cdot X^{\pm N}$.

For more details on cyclotomic polynomials, we refer the reader to standard mathematical references (e.g., [Con15]).

Radix Decomposition. In this manuscript we often refer to the radix decomposition first introduced in [BV14]. This operation consists in decomposing an integer, starting from the most significant bits, and outputting a vector containing the successive decomposition according to a given base and level. In [Joy21], authors proposed a more complete analysis of the impact of the decomposition and even proposed new techniques permitting to balance the decomposition.

Definition 5 (Radix Decomposition [BV14, CGGI16a])**.** *Let $\mathfrak{B} \in \mathbb{N}^*$ be a base decomposition and let $\ell \in \mathbb{N}^*$ be a level decomposition. Given the base and the level decomposition, the radix decomposition algorithm (denoted $\mathbf{Dec}^{\mathfrak{B},\ell}(\cdot)$) takes as input an integer $x \in \mathbb{Z}_q$ and outputs a vector of integers $(x_1 \dots x_\ell) \in \mathbb{Z}_q^\ell$ such that:*

$$\left\langle \mathbf{Dec}^{\mathfrak{B},\ell}(x), \left(\frac{q}{\mathfrak{B}}, \dots, \frac{q}{\mathfrak{B}^\ell}\right)\right\rangle = \left\lfloor x \cdot \frac{\mathfrak{B}^\ell}{q} \right\rceil \cdot \frac{q}{\mathfrak{B}^\ell} = x + \epsilon.$$

With $|\epsilon| < \frac{q}{\mathfrak{B}^\ell}$. The vector $\mathbf{Dec}^{\mathfrak{B},\ell}(x)$ is referred to as the decomposition vector of x. In [CGGI16a], the integers composing the vector $\mathbf{Dec}^{\mathfrak{B},\ell}(x) = (x_1, \dots x_\ell)$ are defined as the unique integers satisfying

$$\left\lfloor x \cdot \frac{\mathfrak{B}^\ell}{q} \right\rceil \cdot \frac{q}{\mathfrak{B}^\ell}, \text{ with } x_i \in [-\mathfrak{B}/2, \mathfrak{B}/2)\,.$$

In [Joy21], the authors highlight the benefit of using a balanced radix decomposition where the x_i are in $\left[-\frac{\mathfrak{B}}{2}, \frac{\mathfrak{B}}{2}\right]$, which results in improved behavior for FHE algorithms.

Remark 2.2. In the previous definition, the radix decomposition is defined for integers, but we note that it is also possible to apply it to any integer polynomial (respectively, vector of integers) by applying the radix decomposition algorithm independently to each coefficient. This results in a vector of integer polynomials (respectively, a vector of integer vectors).

2.1.2 Learning With Error Problems & its Variants

Defined in 2005 by Regev [Reg05], the Learning With Error (LWE) problem has become one of the standard problems of lattice-based cryptography. At a high level, the problem consists in solving a linear system modulo q, perturbed by some randomness known as the error e (also called noise), in order to extract the secret key (or the error). The hardness of the problem is directly related to the modulus q, the lattice dimension n and to the distribution of the error. If the error is close to q, the problem is too hard and if the error is close to zero, the problem becomes easier (and insecure if $e = 0$). Since the initial presentation of the LWE problem and its derivatives, many

constructions based on this hard problem have emerged, including encryption schemes [BDK+18], digital signatures [PFH+20, DKL+18], zero-knowledge proofs [LS18, YAZ+19, LNP22], and more advanced schemes [CKKS17, CGGI16a].

Definition 6 (Learning With Errors [Reg05])**.** *Let $n \in \mathbb{N}$ be the lattice dimension and $q \in \mathbb{N}$ be the ciphertext modulus. Let $\boldsymbol{s} = (s_0, \ldots, s_{n-1}) \in \mathbb{Z}_q^n$ be a secret, where s_i is sampled from a given distribution $\mathcal{D}_s \in \mathbb{Z}_q$ for all $0 \leq i < n$, and let χ be an error distribution.*

We define $\left(\boldsymbol{a}, b = \sum_{i=0}^{n-1} a_i \cdot s_i + e\right) \in \mathbb{Z}_q^{n+1}$ to be a sample from the learning with errors ($\mathrm{LWE}_{n,q,\mathcal{D}_s,\chi}$*) distribution, such that $\boldsymbol{a} = (a_0, \ldots, a_{n-1}) \hookleftarrow \mathcal{U}(\mathbb{Z}_q)^n$, meaning that all the coefficients of a_i are sampled uniformly from $\mathbb{Z}_q$, and the error (noise) $e \in \mathbb{Z}_q$ is sampled from χ.*

The **Decision** $\mathrm{LWE}_{n,q,\mathcal{D}_s,\chi}$ *problem consists in distinguishing m independent samples from $\mathcal{U}(\mathbb{Z}_q)^{n+1}$ from the same amount of samples from the* $\mathrm{LWE}_{n,q,\mathcal{D}_s,\chi}$ *distribution.*

The **Search** $\mathrm{LWE}_{n,q,\mathcal{D}_s,\chi}$ *problem consists in finding the secret $\boldsymbol{s} \in \mathbb{Z}_q^n$ given m samples of* $\mathrm{LWE}_{n,q,\mathcal{D}_s,\chi}$*.*

Theorem 2.1 (Hardness of LWE [Reg05, Pei09])**.** *The Decision* $\mathrm{LWE}_{n,q,\mathcal{D}_s,\chi}$ *problem and Search* $\mathrm{LWE}_{n,q,\mathcal{D}_s,\chi}$ *problem are equivalent. In [Reg05], Regev provided a quantum reduction from the Decision* $\mathrm{LWE}_{n,q,\mathcal{D}_s,\chi}$ *problem to the approximate Shortest Vector Problem (SVP). Later, in [Pei09], Peikert provided a classical proof of this reduction.*

Inspired from the LWE problems introduced previously, many variants were introduced adding additional structures to the original construction. Typically, in this manuscript, alongside LWE, we use the Ring Learning With Errors (RLWE) problem introduced in [SSTX09, LPR10] and the General Learning With Errors (GLWE) problem introduced in [LS15, BGV12] also called Module Learning With Errors problem.

Definition 7 (General Learning With Errors (GLWE) [LS15, BGV12])**.** *Let the quotient ring $\mathfrak{R}_{q,N}$ for a modulo $q \in \mathbb{N}$ and a polynomial size $N \in \mathbb{N}$, with N a power of two. Let $k \in \mathbb{N}$ be the GLWE dimension. Let $\boldsymbol{S} = (S_0, \ldots, S_{k-1}) \in \mathfrak{R}_{q,N}^k$ be a secret, where $S_i = \sum_{j=0}^{N-1} s_{i,j} X^j$ is sampled from a given distribution $\mathcal{D}_S \in \mathfrak{R}_{q,N}$ for all $0 \leq i < k$, and let χ be an error distribution.*

We define $\left(\boldsymbol{A}, B = \sum_{i=0}^{k} A_i \cdot S_i + E\right) \in \mathfrak{R}_{q,N}^{k+1}$ to be a sample from the general learning with errors ($\mathrm{GLWE}_{q,N,k,\mathcal{D}_S,\chi}$*) distribution, such that $\boldsymbol{A} = (A_0, \ldots, A_{k-1}) \hookleftarrow \mathcal{U}(\mathfrak{R}_{q,N})^k$, meaning that all the coefficients of A_i are sampled uniformly from $\mathbb{Z}_q$, and the error (noise) polynomial $E \in \mathfrak{R}_{q,N}$ is such that all the coefficients are sampled from the distribution χ.*

The **decision** $\mathrm{GLWE}_{q,N,k,\mathcal{D}_S,\chi}$ *problem consists in distinguishing m independent samples from $\mathcal{U}(\mathfrak{R}_{q,N})^{k+1}$ from the same amount of samples from the* $\mathrm{GLWE}_{q,N,k,\mathcal{D}_S,\chi}$ *distribution.*

The **Search** $\mathrm{GLWE}_{q,N,k,\mathcal{D}_S,\chi}$ *problem consists in finding the secret $\boldsymbol{S} \in \mathfrak{R}_{q,N}^k$ given m samples of* $\mathrm{GLWE}_{q,N,k,\mathcal{D}_S,\chi}$*.*

Theorem 2.2 (Hardness of GLWE [LS15, BGV12])**.** *The decision* $\mathrm{GLWE}_{q,N,k,\mathcal{D}_S,\chi}$ *problem and the Search* $\mathrm{GLWE}_{q,N,k,\mathcal{D}_S,\chi}$ *problem are equivalent. The hardness of solving the search version of the GLWE problem is related to the Approximate Shortest Vector Problem (α-SVP) on ideal lattices.*

Remark 2.3 (LWE, RLWE & GLWE)**.** All the problems presented above are closely related, indeed when $N = 1$, the GLWE problem corresponds to the LWE problem: $\mathrm{GLWE}_{q,1,k,\mathcal{D}_S,\chi} = \mathrm{LWE}_{k=n,q,\mathcal{D}_s,\chi}$. In this case we consider the parameter $n = k$ to be the size of the LWE secret key.

The RLWE problem is a particular case of the *GLWE* problem. When $k = 1$, the GLWE problem corresponds to the RLWE problem: $\mathrm{GLWE}_{q,N,1,\mathcal{D}_S,\chi} = \mathrm{RLWE}_{q,N,\mathcal{D}_S,\chi}$.

To encompass all the problems together, we will refer to them collectively as (G)LWE. Latter, for the same reasons we will use this notation (with a G between parenthesis) for different ciphertexts, to refer to all of them all at once.

In general, the secret key distribution $\mathcal{D}_S \in \mathfrak{R}_{q,N}$ is such that the polynomial coefficients are usually either sampled from a uniform binary distribution, a uniform ternary distribution or a small Gaussian distribution ([BJRLW23, ACPS09]).

Some works have studied specific secret key distributions that offer different trade-offs. One example is the secret keys with a fixed hamming weight, i.e., secret keys where the number of non-zero coefficients is fixed and publicly known, as presented in [KDE+23, GP25]. These types of secret keys are used in different schemes, as they offer reduced noise growth. Another example is the block secret keys [LMSS23], a type of secret key where the secret key can be expressed as a concatenation of several vectors of Hamming weight at most one. Finally, we can mention sparse secret keys [ABW10], where the number of non-zero elements is small, though the exact Hamming weight is unknown. This type of key must be used with caution, as it may introduce vulnerabilities and reduce the overall security level.

2.1.3 Attacks on (G)LWE Problems

In the previous section, we defined the LWE problem (Definition 6) and its variants (Definition 7). We highlight that the security of these problems is directly linked to various parameters such as the dimensions, the distributions, the modulus. In particular, the LWE problem can be interpreted as a lattice problem, and it is at least as hard to solve as the well known Shortest Independent Vector Problem (SIVP). Most attacks on (G)LWE schemes involve solving lattice-based problems.

Attacks on LWE. The first well known kind of attacks is the so called LWE primal attacks. This attack was first formulated in [ADPS16] and improved in [AGVW17, DSDGR20, PV21]. It consists in using lattice reduction to solve an instance of uSVP [Mic01] (the unique Shortest Vector Problem) generated from LWE samples. The most common way to perform this reduction is to use the BKZ algorithm [SE94] to reduce a lattice basis by using a SVP (Shortest Vector Problem) oracle. Based on this attack, the security of an LWE instance relies on the cost of lattice reduction for solving uSVP. In the paper [ADPS16], the authors propose to analyze the hardness of RLWE as an LWE problem. All the research on this attack tend to find the best cost of solving uSVP in order to find the closest model of security for LWE and by extension for RLWE.

The second type of attack is the LWE dual attacks. This attack is explained in [MR09] and upgraded with the dual hybrid attacks in [Alb17]. It consists in solving an instance of the SIS (Short Integer Solution) problem [Ajt96, MR07] in the dual lattice of the lattice formed by LWE samples. As for the first type of attacks, the security of an LWE instance is based on the cost of solving the problem SIS.

The third well known kind of attacks is the coded-BKW attacks, which are based on the algorithm BKW (Blum, Kalai and Wasserman [BKW03]). This attack is explained in [GJS15, KF15]. The BKW algorithm is a recursive dimension reduction for LWE instances. In [GJS15], the authors make use of these attacks against RLWE. To do that, the RLWE problem is seen as a sub problem of LWE.

Attacks on RLWE/GLWE. In the last decade, some attacks (for example [CDW17, PMHS19, BRL20, BLNRL23]) tried to take advantage of the structure of RLWE and GLWE to solve the id-SVP (ideal-Shortest Vector Problem). However, none of these attacks are as efficient as the LWE attacks presented before. Thus, to efficiently break GLWE, one actually uses LWE attacks: the security of GLWE $\in \mathfrak{R}_{N,q}^{k+1}$ is then estimated as the LWE $\in \mathbb{Z}_q^{kN+1}$ one.

Other Attacks. Some other attacks are not based on a reduction to a classical problem but on the leakage of some fraction of the coordinates of the NTT transform of the RLWE secret. It is the case in [DSGKS18] which proposes a more direct attack against RLWE under this leakage assumption.

2.1.4 Practical Estimation of (G)LWE Security

The different attacks we just detailed, need to be taken into account when defining parameter sets for (G)LWE problems. The security of all lattice-based cryptographic constructions relies on the resistance against these different attacks.

To estimate the security against the attacks aforementioned, in 2015, Martin R. Albrecht, Rachel Player and Sam Scott [APS15] presented the Lattice Estimator, a tool that estimates the cost of different attacks given parameters set. This tool has been a cornerstone in the adoption of lattice-based cryptography and is integrated into the NIST standardization process, as well as the ISO effort to standardize homomorphic encryption schemes. Using this tool, the objective is to identify parameter sets that guarantee a desired security level λ.

Remark 2.4. In this manuscript, we use the Lattice Estimator [APS15] to estimate the security of the different parameter sets. All the parameter sets yield an estimated security level of $\lambda \geq 128$. We note that this security level corresponds to the estimation at the time of the article publication. Since then, new attacks or methodologies may have emerged, potentially leading to slight reductions in the actual security.

2.1.5 Fully Homomorphic Encryption

A fully homomorphic encryption (FHE) scheme is a family of encryption schemes that allows operations to be performed over encrypted data. Similar to classical encryption schemes, an FHE scheme is composed of three core components: a key generation, an encryption, and a decryption algorithm. In addition to these traditional algorithms, FHE schemes also include an evaluation algorithm, which allows the execution of arbitrary computations on ciphertexts.

- **Key generation:** Given a security level λ, generates the private secret key (sk) and the public materials (PUB) necessary for the evaluation algorithm. In the case of asymmetric encryption, it also outputs the corresponding public key (pk).

$$\mathsf{Keygen}(1^\lambda) = (pk, sk, \mathsf{PUB}).$$

- **Encryption:** Given $\mathfrak{M}$ the message space and $\mathfrak{C}$ the ciphertext space, the encryption algorithm takes as input a message $m \in \mathfrak{M}$ and the key, which is public in asymmetric encryption (pk) and private in symmetric encryption (sk), and it outputs a ciphertext $c \in \mathfrak{C}$.

$$\mathsf{Enc}(m, pk) = c.$$

- **Decryption:** The decryption algorithm takes as input a ciphertext $c \in \mathfrak{C}$ and the private secret key sk, and returns the original message $m \in \mathfrak{M}$. It satisfies the correctness property with probability greater than $1 - \epsilon$, for $\epsilon \geq 0$.

$$\mathsf{Dec}(c, sk) = m \quad \text{such that} \quad \mathsf{Dec}(\mathsf{Enc}(m, pk), sk) = m.$$

  An FHE scheme is perfectly correct if $\epsilon = 0$.

- **Evaluation:** The evaluation algorithm enables homomorphic computation over encrypted data. It takes as input a list of ciphertexts $\{c_i\}_{i\in[0,k]} \in \mathfrak{C}^{k+1}$, where each c_i encrypts a message $m_i \in \mathfrak{M}$, the public materials (PUB) and a function f_h corresponding to the homomorphic circuit. This function corresponds to the cleartext function f representing the clear circuit. The algorithm produces a new ciphertext that encrypts the result of applying f to the clear messages:

$$\mathsf{Eval}\left(\{c_i\}_{i\in[0,k]}, \mathsf{PUB}, f_h\right) = f_h\left(\{c_i\}_{i\in[0,k]}\right).$$

  It satisfies the correctness condition:

$$\mathsf{Dec}\left(f_h\left(\{c_i\}_{i\in[0,k]}\right), sk\right) = f\left(\{m_i\}_{i\in[0,k]}\right).$$

Due to the manipulability of ciphertexts, the security model adopted for FHE schemes was IND-CPA, meaning that ciphertexts are indistinguishable under chosen-plaintext attacks. Recently, a

stronger security model, denoted IND-CPA$^{\mathrm{D}}$ [LM21], was proposed specifically for FHE schemes. This new model increases the security of the standard IND-CPA definition by granting the adversary access to a decryption oracle capable of decrypting ciphertexts that have been homomorphically evaluated. This new security model better targets the new attacks specifically designed for FHE, where evaluating ciphertexts may permit new attacks, such as the one presented in [LM21, CCP$^+$24, CSBB24].

Nowadays, many FHE schemes are based on lattice cryptography, particularly on the LWE problem and its variants. The most commonly used are BGV [BGV12], B/FV [FV12, Bra12], HEEAN [CHK$^+$18], CKKS [CKKS17], FHEW [DM15], and TFHE [CGGI16a]. Since these schemes are based on the LWE problem, their security relies on small random terms, known as noise, present in each ciphertext. When an homomorphic operation is performed on encrypted data, this noise increases, and if it reaches a certain threshold, the ciphertext can no more be correctly decrypted. Therefore, managing noise growth is essential to preserve correctness.

In 2009, Gentry [Gen09] proposed a technique to address this issue by refreshing the noise inside a ciphertext. This technique is called bootstrapping. Following this blueprint, all the aforementioned FHE schemes have their own bootstrapping procedures, each with its advantages and disadvantages. In what follows, we focus on the TFHE scheme and its bootstrapping.

2.2 TFHE Scheme

Torus Fully Homomorphic Encryption (TFHE), also known as CGGI, is a (G)LWE-based fully homomorphic encryption scheme initially presented in [CGGI16a]. It was initially presented as an improvement over the FHE scheme FHEW [DM15], specifically improving the techniques used to perform the bootstrapping (see Section 2.4), a core operation in both schemes that accounts for the majority of the total execution time of a program. In addition, TFHE supports more functionalities and enables more efficient homomorphic evaluation of any circuits. The main difference compared to other FHE schemes, is that FHEW and TFHE have a very fast bootstrapping making it a core of any computation. This advantage comes from the fact that both schemes are using small ciphertexts (compared to other FHE schemes) using native CPU type for modulus and relatively small lattices dimensions. Initially, both TFHE and FHEW were described for encoding boolean messages, but further works such as [CJP21, CLOT21] noticed and studied how to natively handle small integer messages (smaller than 10 bits).

Originally, in the first TFHE articles [CGGI16a, CGGI17, CGGI20], the message space and the ciphertexts space were both defined over the real torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$. In modern architecture, values are natively represented over 32- or 64-bits precision. This is why in the first library [CGGI16b], introduced with these articles, both the message space and the ciphertexts space are implemented using the native machine arithmetics modulo 2^{32} or 2^{64}. As detailed in [BGGJ20], working with such values is equivalent to working over the discretized torus. Indeed, there exists an isomorphism between $\mathbb{Z}_q$ and the discretized torus $\frac{1}{q}\mathbb{Z}/\mathbb{Z}$. In this document, we adopt the notation in the ring integer $\mathbb{Z}_q$.

First, we introduce how messages are encoded in TFHE. Then, we present the different types of TFHE ciphertexts that encrypt these encoded messages.

2.2.1 TFHE Encoding

In TFHE, messages must be encoded before encryption. Because the scheme is built on the (G)LWE assumptions where ciphertexts contain an error term. If this error interferes with the message during decryption, it may lead to an incorrect decryption. To prevents such loss, the message is usually shifted to the most significant bits, ensuring that the error, concentrated in the least significant bits, does not corrupt the message.

Definition 8 (Polynomial Message Encode & Decode)**.** *Let $q \in \mathbb{N}$ be a ciphertext modulus, and let $p \in \mathbb{N}$ a message modulus, and $\pi \in \mathbb{N}$ the number of bits of padding*[1]*. We have $2^\pi \cdot p \leq q$ and $\widetilde{2^\pi \cdot p}$ is the plaintext modulus. Let $M \in \mathfrak{R}_{p,N}$ be a message. We define the encoding of M as: $\widetilde{M} = \mathsf{Encode}\left(M, 2^\pi \cdot p, q\right) = \lfloor \Delta \cdot M \rceil \in \mathfrak{R}_{q,N}$ with $\Delta = \frac{q}{2^\pi \cdot p} \in \mathbb{Q}$ the scaling factor (see a visual example in Figure 2.1). To decode, we compute the following function: $M = \mathsf{Decode}\left(\widetilde{M}, 2^\pi \cdot p, q\right) = \left\lfloor \frac{\widetilde{M}}{\Delta} \right\rceil \in \mathbb{Z}_{2^\pi \cdot p}$.*

In practice $\widetilde{M}$ contains a small error term $E = \sum_{i=0}^{N-1} e_i \cdot X^i \in \mathbb{Q}[X]/(X^N + 1)$, so we can rewrite $\widetilde{M} = \Delta \cdot M + E \in \mathbb{Z}_q$. The decoding algorithm fails if and only if there is at least one $i \in [0, N-1]$ such that $|e_i| \geq \frac{\Delta}{2}$. We can note this probability as follows:

$$\mathbb{P}\left(\bigcup |e_i| \geq \frac{\Delta}{2}\right) = \mathbb{P}\left(\mathsf{Decode}\left(\widetilde{M}, 2^\pi \cdot p, q\right) \neq M\right). \tag{2.1}$$

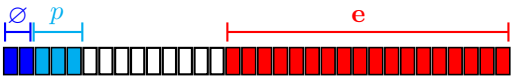

Figure 2.1: Plaintext binary representation with $p = 8 = 2^3$ (cyan), $\pi = 2$ (dark blue) such that $2^\pi \cdot p \leq q$, the error e (red). The white part is empty. The MSB are on the left and the LSB on the right.

2.2.2 TFHE Ciphertexts

In this section, we present all the different ciphertexts used in TFHE. First, we introduce the (G)LWE ciphertexts, which are the primary ciphertexts used in TFHE to encrypt the data. Then, based on these ciphertexts, we present the (G)LEV ciphertexts, which lead to the (G)GSW ciphertexts, essential for public material and for performing more advanced algorithms such as bootstrapping.

(G)LWE Ciphertexts. In TFHE, the common ciphertexts defined in [CGGI16a] are the LWE ciphertexts encrypting only small integers. The security of these ciphertexts directly relies on the LWE problem detailed in Definition 6.

Definition 9 (LWE Ciphertexts)**.** *Based on* LWE *samples* $\mathrm{LWE}_{n,q,\mathfrak{D}_s,\chi}$ *as defined in Definition 6, we can generate* LWE *ciphertexts encrypting an encoded messages $\Delta m \in \mathbb{Z}_q$.*

Let $\boldsymbol{s} = (s_0, \ldots, s_{n-1}) \in \mathbb{Z}_q^n$ be the LWE *secret key, with s_i sampled from the distribution $\mathfrak{D}_s$. Let $\boldsymbol{a} = (a_0, \ldots, a_{n-1}) \in \mathbb{Z}_q^n$ be the* LWE *mask, with $a_i \hookleftarrow \mathfrak{U}(\mathbb{Z}_q)$ for $i \in [0, n-1]$. Given an encoded message $\Delta m \in \mathbb{Z}_q$, an* LWE *ciphertext encrypts the message in the* LWE *body $b \in \mathbb{Z}_q$ such that $b = \langle \boldsymbol{a} \cdot \boldsymbol{s} \rangle + \Delta m + e$ where e is the error sampled from the distribution χ.*

We denote the LWE *ciphertext encrypting an encoded message Δm as follows:*

$$\mathsf{ct} = (\boldsymbol{a}, b = \langle \boldsymbol{a} \cdot \boldsymbol{s} \rangle + \Delta m + e) \in \mathrm{LWE}_{\boldsymbol{s}}(\Delta m) \subseteq \mathbb{Z}_q^{n+1}.$$

Remark 2.5. In this manuscript, unless clearly defined, the secret key follows a Boolean distribution. So for $i \in [0, n-1]$, we have $s_i \hookleftarrow \mathfrak{U}(\{0,1\})$, i.e., $\mathfrak{D}_s = \mathfrak{U}(\{0,1\})$.

In the same manner, unless clearly defined, errors follow a Gaussian distribution with a mean set to zero and a standard deviation set to σ, i.e., $e \hookleftarrow \mathcal{N}_\sigma$.

For algorithmic reason, TFHE also uses the GLWE ciphertexts. These ciphertexts permit encrypting polynomial small messages using the integer ring $\mathfrak{R}_{q,N} = \mathbb{Z}_q[X]/X^N + 1$ (See remark 2.1). The security of these ciphertexts directly relies on the GLWE problem detailed in Definition 7.

[1] For simplicity we use a power of 2 for the padding, but this is not a necessary condition.

Definition 10 (GLWE Ciphertexts)**.** *Based on* $\mathrm{GLWE}_{q,N,k,\mathcal{D}_\delta,\chi}$ *samples as defined in Definition 7, we can generate* GLWE *ciphertexts encrypting an encoded polynomial messages* $\Delta M \in \mathfrak{R}_{q,N}$*.*

Let $\boldsymbol{S} = (S_0, \ldots, S_{k-1}) \in \mathfrak{R}_{q,N}^k$ *be the* GLWE *secret key, where* $S_i = \sum_{j=0}^{N-1} s_{i,j} X^j$ *is sampled from the distribution* $\mathcal{D}_S$ *for all* $0 \leq i < k$*. Let* $\boldsymbol{A} = (A_0, \ldots, A_{k-1}) \hookleftarrow \mathcal{U}\left(\mathfrak{R}_{q,N}\right)^k$ *be the* GLWE *mask. Given an encoded polynomial message* $\Delta M \in \mathfrak{R}_{q,N}$*, the* GLWE *ciphertext encrypts the message in the* GLWE *body* $B \in \mathfrak{R}_{q,N}$ *such that* $B = \sum_{i=0}^{k-1} A_i \cdot S_i + \Delta M + E$ *where* $E \in \mathfrak{R}_{q,N}$ *is the polynomial error, and where each coefficient* e_i *is sampled from the distribution* χ*.*

We denote the GLWE *ciphertext encrypting an encoded polynomial message* ΔM *as follows:*

$$\mathsf{CT} = \left(\boldsymbol{A}, B = \sum_{i=0}^{k-1} A_i \cdot S_i + \Delta M + E \right) \in \mathrm{GLWE}_{\boldsymbol{S}}(\Delta M) \subseteq \mathfrak{R}_{q,N}^{k+1}.$$

The encryption procedure is detailed in Algorithm 1.

Algorithm 1: $\mathsf{CT}_M \leftarrow \mathsf{Encrypt}(\boldsymbol{S}, \Delta M)$

Context:
- N : Polynomial Size.
- k : GLWE dimension.
- q : GLWE modulus.
- χ : Noise Distribution.

Input:
- $\Delta M \in \mathfrak{R}_{q,N}$: an encoded message.
- $\boldsymbol{S} \in \mathfrak{R}_{q,N}^k$: a GLWE secret key.

Output:
- $\mathsf{CT}_M \in \mathrm{GLWE}_{\boldsymbol{S}}(\Delta M) \subseteq \mathfrak{R}_{q,N}^{k+1}$:
- a GLWE ciphertext encrypting M.

```
E ↩ χ^N
B = ΔM + E  mod ℜ_{q,N}
for i ∈ [0, k − 1] do
    A_i ↩ U(ℜ_{q,N})
    B = B + A_i · S_i  mod ℜ_{q,N}
CT_M = (A, B) = (A_0, ..., A_{k−1}, B)
return CT_M ∈ GLWE_S(ΔM) ⊆ ℜ_{q,N}^{k+1}
```

Remark 2.6 (Trivial Encryption)**.** Given an encoded message $\Delta M \in \mathfrak{R}_{q,N}$, anyone can create an unencrypted ciphertext with the characteristics necessary to perform operations with encrypted values. These ciphertexts are generated using trivial encryption, meaning they are created with both the polynomial mask and the polynomial error set to zero. As a result, these ciphertexts are not encrypted and are insecure. They are defined as follows:

$$\mathsf{CT} = (\boldsymbol{0}, B = \Delta M) \in \mathrm{GLWE}_{\boldsymbol{S}}(\Delta M) \subseteq \mathfrak{R}_{q,N}^{k+1}.$$

These types of ciphertexts are mainly used to perform operations between plaintext and ciphertext and help simplify some algorithms.

Definition 11 (Flattened Representation of a GLWE Secret Key)**.** *A GLWE secret key* $\boldsymbol{S} = \left(S_0 = \sum_{j=0}^{N-1} s_{0,j} X^j, \ldots, S_{k-1} = \sum_{j=0}^{N-1} s_{k-1,j} X^j \right) \in \mathfrak{R}_{q,N}^k$ *can be* flattened *into an LWE secret key* $\bar{\boldsymbol{s}} = (\bar{s}_0, \ldots, \bar{s}_{kN-1}) \in \mathbb{Z}^{kN}$ *in the following manner:* $\bar{s}_{iN+j} := s_{i,j}$*, for* $0 \leq i < k$ *and* $0 \leq j < N$*.*

Definition 12 ((G)LWE Decryption)**.** *Let* $\mathsf{CT} \in \mathrm{GLWE}_{\boldsymbol{S}}(\Delta M) \subseteq \mathfrak{R}_{q,N}^{k+1}$ *be a ciphertext encrypting the encoded message* $\Delta M \in \mathfrak{R}_{q,N}^{k+1}$*, under the secret key* $\boldsymbol{S} = (S_0, \ldots, S_{k-1}) \in \mathfrak{R}_{q,N}^k$ *(Definition 10).*

The decryption returns the encoded message along the noise added during the encryption. The decryption of $\mathsf{CT} = \left(\boldsymbol{A}, B = \sum_{i=0}^{k-1} A_i \cdot S_i + \Delta M + E\right)$*, is computed as follows:*

$$\widetilde{M} + E = \Delta M + E = B - \sum_{i=0}^{k-1} A_i \cdot S_i.$$

The decryption procedure is detailed in Algorithm 2.

Remark 2.7. In the following, to lighten the notation and highlight the message, when Δ is clear from context, we will omit it in the ciphertext notation and simply write $\mathrm{GLWE}_{\boldsymbol{S}}(M)$ for a GLWE ciphertext that encrypts the encoded message $\Delta M \in \mathfrak{R}_{q,N}$, or $\mathrm{LWE}_{\boldsymbol{s}}(m)$ for an LWE ciphertext that encrypts the encoded message $\Delta m \in \mathbb{Z}_q$.

Algorithm 2: $\Delta M + E \leftarrow \mathsf{Decrypt}(\boldsymbol{S}, \mathsf{CT}_M)$

Context:
- N : Polynomial Size.
- k : GLWE dimension.
- q : GLWE modulus.

Input:
- $\mathsf{CT}_M \in \mathrm{GLWE}_{\boldsymbol{S}}(M) \subseteq \mathfrak{R}_{q,N}^{k+1}$: GLWE ciphertext encrypting M.
- $\boldsymbol{S} \in \mathfrak{R}_{q,N}^{k}$: the GLWE secret key associated to the GLWE ciphertext.

Output:
- $\Delta M + E \in \mathfrak{R}_{q,N}$: The encoded message and the error of the input ciphertext.

```
res = B
for i ∈ [0, k − 1] do
    res = res − A_i · S_i  mod ℜ_{q,N}
/* res = B − Σ_{i=0}^{i=k−1} A_i S_i  mod ℜ_{q,N}                    */
return res ∈ ℜ_{q,N}
```

(G)LEV Ciphertexts. Based on the (G)LWE ciphertexts, we can build more complex ciphertexts useful to perform algorithm such as the key switch (Algorithm 4) or even more complex ciphertexts such as the (G)GSW ciphertext detailed latter. (G)LEV ciphertexts were first detailed in [CLOT21], they are composed of collection of (G)LWE ciphertexts, then their security also relies on the (G)LWE problem (Definition 7).

Definition 13 (GLEV Ciphertexts)**.** *For a given decomposition base* $\mathfrak{B} \in \mathbb{N}^*$ *and a level decomposition* $\ell \in \mathbb{N}^*$*, a* GLEV *ciphertext of a message* $M \in \mathfrak{R}_{p,N}$ *under a secret key* $\boldsymbol{S} \in \mathfrak{R}_{q,N}^{k}$ *is a ciphertext composed of* ℓ $\mathrm{GLWE}_{q,N,k,\mathcal{D}_S,\chi}$ *ciphertexts (Definition 7) encrypting the same message* M *for different scaling factors (given by the base* $\mathfrak{B}$ *and the level* ℓ*). Let* $\mathsf{CT}_i \in \mathrm{GLWE}_{\boldsymbol{S}}\left(\frac{q}{\mathfrak{B}^{i+1}} M\right) \subseteq \mathbb{R}_{q,N}^{k+1}$ *for* $i \in [0, \ell - 1]$*. Then, a* GLEV *ciphertext is denoted* $\overline{\mathsf{CT}}$ *and is composed of* ℓ GLWE *ciphertexts. He is defined as:*

$$\overline{\mathsf{CT}} = (\mathsf{CT}_0, \ldots, \mathsf{CT}_{\ell-1}) \in \mathrm{GLEV}_{\boldsymbol{S}}^{\mathfrak{B},\ell}(M) \subseteq \mathfrak{R}_{q,N}^{\ell \times (k+1)}.$$

As for the (G)LWE ciphertexts (see Remark 2.3), a GLEV ciphertexts with $N = 1$ is a LEV ciphertext and in this case we consider the parameter $n = k$ for the size of the LWE secret key. LEV ciphertexts are denoted $\overline{\mathsf{ct}}$. A GLEV ciphertext with $k = 1$ and $N > 1$ is a RLEV ciphertext.

(G)GSW Ciphertexts. The last type of ciphertexts is the (G)GSW ciphertexts, built using the (G)LEV ciphertexts introduced just before. These ciphertexts are useful in many algorithms and especially, there are used to perform the bootstrapping (described latter in Algorithm 11) the central operation of TFHE. At a high level, these messages are composed of collection of (G)LEV ciphertexts and they was first detailed in [GSW13].

Definition 14 (GGSW ciphertexts)**.** *For a given decomposition base $\mathfrak{B} \in \mathbb{N}^*$ and a level decomposition $\ell \in \mathbb{N}^*$, a* GGSW *ciphertext encrypting a message $M \in \mathfrak{R}_{p,N}$ under a secret key $\boldsymbol{S} \in \mathfrak{R}_{q,N}^k$ is composed by $(k+1)$* GLEV *ciphertexts (Definition 13) encrypting the same message M multiplied by elements of the secret key for different scaling factor (given by the base $\mathfrak{B}$ and the level ℓ). Let $\overline{\mathsf{CT}}_j \in \mathrm{GLEV}_{\boldsymbol{S}}^{\mathfrak{B},\ell}(-S_j \cdot M) \subseteq \mathfrak{R}_{q,N}^{\ell\times(k+1)}$ for $j \in [0,k)$ and $\overline{\mathsf{CT}}_k \in \mathrm{GLEV}_{\boldsymbol{S}}^{\mathfrak{B},\ell}(M) \subseteq \mathfrak{R}_{q,N}^{\ell\times(k+1)}$. Then, a* GGSW *ciphertext is denoted $\overline{\overline{\mathsf{CT}}}$, is composed of $k+1$* GLEV *ciphertexts and is defined as:*

$$\overline{\overline{\mathsf{CT}}} = \left(\overline{\mathsf{CT}}_0, \ldots, \overline{\mathsf{CT}}_k\right) \in \mathrm{GGSW}_{\boldsymbol{S}}^{\mathfrak{B},\ell}(M) \subseteq \mathfrak{R}_{q,N}^{(k+1)\ell\times(k+1)}.$$

As for the (G)LWE ciphertexts (see Remark 2.3), a GGSW ciphertexts with $N = 1$ is a GSW ciphertext and in this case we consider the parameter $n = k$ for the size of the LWE secret key. GSW ciphertexts are denoted $\overline{\overline{\mathsf{ct}}}$. A GGSW ciphertext with $k = 1$ and $N > 1$ is a RGSW ciphertext.

Remark 2.8. In Section 2.2 we mentioned that TFHE was originally defined over the torus $\mathbb{T}$. In the literature, the different ciphertexts LWE / RLWE / GLWE and GSW / RGSW / GGSW are also denoted as TLWE / TRLWE / TGLWE, and respectively TGSW / TRGSW / TGGSW to emphasize that they are defined over the torus.

2.3 TFHE Modular arithmetics

In this section, we introduce the basic operations performed over TFHE ciphertexts. We first present the addition between two (G)LWE ciphertexts and the multiplication by plaintext integers. Then we present how to perform more complex operations such as the key switch, an operation that allows changing the secret key of a ciphertext to another one, or the external product, an operation that performs multiplication between (G)LWE and (G)GSW ciphertexts.

2.3.1 Homomorphic Addition of (G)LWE ciphertexts

The first operation we introduce is the addition between two (G)LWE ciphertexts. Even though this is one of the simplest homomorphic operations, we analyze its cost and its noise, which allows us to introduce the methodology used to study any algorithm throughout the manuscript.

Remark 2.9. To study the noise growth resulting from an operation, we proceed as follows. We take two ciphertexts ct_1 and ct_2 encrypted with noise following the distributions χ_1 and χ_2, respectively, and perform the operation under study. Then, we decrypt the resulting ciphertext $\mathsf{ct}_{\mathsf{out}}$ using Algorithm 2, which outputs $\Delta m_{\mathsf{out}} + e_{\mathsf{out}}$. We finally analyze the new distribution of the output noise e_{out} with respect to χ_1 and χ_2.

Theorem 2.3 (GLWE Addition (Algorithm 3))**.** *Let $\mathsf{CT}_1 = (A_{1,0}, \ldots, A_{1,k-1}, B_1) \in \mathfrak{R}_{q,N}^{k+1}$ and $\mathsf{CT}_2 = (A_{2,0}, \ldots, A_{2,k-1}, B_2) \in \mathfrak{R}_{q,N}^{k+1}$ be two GLWE ciphertexts encrypting the message $\Delta M_1 \in \mathfrak{R}_{q,N}$ and the message $\Delta M_2 \in \mathfrak{R}_{q,N}$, respectively, under the GLWE secret key $\boldsymbol{S} = (S_0, \ldots, S_{k-1}) \in \mathfrak{R}_{q,N}^k$. The coefficients of the corresponding noise polynomials, E_1 and E_2, are statistically independent and the coefficients of E_1 (resp. E_2) follow a centred gaussian distribution $\chi_1 = \mathcal{N}_{\sigma_1^2}$ (resp. $\chi_2 = \mathcal{N}_{\sigma_2^2}$). Then, Algorithm 3 returns the new GLWE ciphertext $\mathsf{CT} = (A_0, \ldots, A_{k-1}, B)$ encrypting the message $\Delta(M_1 + M_2) \in \mathfrak{R}_{q,N}$ such that:*

$$\mathsf{CT} = \mathsf{Add}(\mathsf{CT}_1, \mathsf{CT}_2) = (A_{1,0} + A_{2,0}, \ldots, A_{1,k-1} + A_{2,k-1}, B_1 + B_2) \in \mathfrak{R}_{q,N}^{k+1}.$$

Algorithm 3: $\mathsf{CT} \leftarrow \mathsf{Add}(\mathsf{CT}_1, \mathsf{CT}_2)$

Context: $\begin{cases} \boldsymbol{S} = (S_0, \dots, S_{k-1}) \in \mathfrak{R}_{q,N}^k : \text{The input GLWE secret key.} \\ \mathsf{CT}_i = (A_{i,0}, \dots, A_{i,k-1}, B_i) \in \mathrm{GLWE}_{\boldsymbol{S}}(M_i) \subseteq \mathfrak{R}_{q,N}^{k+1} \text{ for } i \in \{0,1\}. \end{cases}$

Input: $\begin{cases} \mathsf{CT}_1 \in \mathrm{GLWE}_{\boldsymbol{S}}(M_1) \subseteq \mathfrak{R}_{q,N}^{k+1}, \text{ with the message } M_1 \in \mathfrak{R}_{q,N} \\ \mathsf{CT}_2 \in \mathrm{GLWE}_{\boldsymbol{S}}(M_2) \subseteq \mathfrak{R}_{q,N}^{k+1}, \text{ with the message } M_2 \in \mathfrak{R}_{q,N} \end{cases}$

Output: $\left\{ \mathsf{CT} \in \mathrm{GLWE}_{\boldsymbol{S}}(M_1 + M_2) \subseteq \mathfrak{R}_{q,N}^{k+1} \right.$

1 **for** $i \in [0, k-1]$ **do**
2 $\quad A_i \leftarrow A_{1,i} + A_{2,i}$
3 $B \leftarrow B_1 + B_2$
4 **return** $\mathsf{CT} = (A_0, \dots, A_{k-1}, B)$

After the addition, the noise output follows the distribution $\chi = \mathcal{N}_{\sigma_1^2+\sigma_2^2}$.

The algorithmic complexity of Algorithm 3 is $\mathsf{Cost}_{\mathsf{Add}}^{k,N} = (k+1)N \cdot \mathbb{C}_{add}$ *where* $\mathbb{C}_{add}$ *denotes the complexity of adding two elements of* $\mathbb{Z}_q$.

Remark 2.10. In the following, we use the classical notation + to denote the addition between two ciphertexts. When multiple additions are chained, we use the classical notation $\sum$. For instance, the sum of 3 GLWE ciphertexts is denoted as:

$$\sum_{i=0}^{2} \mathsf{CT}_i = \mathsf{CT}_0 + \mathsf{CT}_1 + \mathsf{CT}_2 = \mathsf{Add}(\mathsf{CT}_2, \mathsf{Add}(\mathsf{CT}_0, \mathsf{CT}_1)).$$

Proof *(Theorem 2.3) Let* $\boldsymbol{S} = (S_0, \dots, S_{k-1}) \in \mathfrak{R}_{q,N}^k$ *be the GLWE secret key. Let* CT_1 *and* CT_2 *be two ciphertexts in* $\mathfrak{R}_{q,N}^{k+1}$ *such that* $\mathsf{CT}_i = (A_{i,0}, \dots, A_{i,k-1}, B_i) \in \mathrm{GLWE}(M_i)$ *with* $B_i = \sum_{j=0}^{k-1} A_{i,j} \cdot S_j + \widetilde{M}_i + E_i$ *for* $i \in \{0,1\}$.

We suppose that the corresponding noise polynomials, E_1 *and* E_2*, are statistically independent and each of there coefficient follow two centred gaussian distribution* $\chi_1 = \mathcal{N}_{\sigma_1^2}$ *and* $\chi_2 = \mathcal{N}_{\sigma_2^2}$.

After Algorithm 3, we obtain:

$$\mathsf{CT} = \mathsf{CT}_1 + \mathsf{CT}_2 = (A_{1,0} + A_{2,0}, \dots, A_{1,k-1} + A_{2,k-1}, B_1 + B_2) \in \mathfrak{R}_{q,N}^{k+1}.$$

Let us now decrypt CT *using Definition 12.*

$$\begin{aligned} \widetilde{M} + E &= B - \sum_{i=0}^{k-1} A_i S_i = B_1 + B_2 - \sum_{i=0}^{k-1} (A_{1,i} + A_{2,i}) \cdot S_i \\ &= \sum_{i=0}^{k-1} A_{1,i} S_i + \widetilde{M}_1 + E_1 + \sum_{i=0}^{k-1} A_{2,i} S_i + \widetilde{M}_2 + E_2 - \sum_{i=0}^{k-1} (A_{1,i} + A_{2,i}) \cdot S_i \\ &= \widetilde{M}_1 + \widetilde{M}_2 + E_1 + E_2. \end{aligned}$$

So after Algorithm 3, we obtain a new noise E *equals to* $E_1 + E_2$*. As* E_1 *and* E_2 *are statistically independent, each coefficient of the output noise polynomial* E *follow a Gaussian distribution* $\chi = \mathcal{N}_{\sigma_1^2+\sigma_2^2}$. □

After performing an homomorphic addition, we see that the noise grows. As presented before, the noise growth appear in almost all the following homomorphic operations.

We note that Algorithm 3, along with Theorem 2.3 and its corresponding proof, can be trivially adapted to LWE and RLWE ciphertexts and to (G)LEV and (G)GSW ciphertexts.

In the case of addition between a cleartext and a ciphertext, it suffices to trivially encrypt the cleartext (see Remark 2.6) in order to perform the operation. In this case, no additional noise is introduced when applying Algorithm 3.

2.3.2 Rotation and Multiplication by Public Constants

Here, we present how to perform various multiplications with public constants. First, we describe how to rotate encrypted polynomials in RLWE/GLWE ciphertexts. Next, we show how to multiply any ciphertext by a constant. Finally, building on these two operations, we explain how to perform a multiplication by a known polynomial.

In the following, we use the classical notation $\cdot$, where $\mathsf{CT} \cdot w$ denotes the multiplication of each polynomials composing the ciphertext CT by $w \in \mathbb{Z}_q$.

GLWE Ciphertext Rotations. As previously mentioned, we work in the ring $\mathfrak{R}_{q,N}$, where multiplications are done modulo $X^N + 1$. To perform a rotation by $w \in \mathbb{Z}$ of a polynomial, it suffices to multiply it by X^w. Let us show how to perform a rotation of a polynomial encrypted in a GLWE ciphertext.

Theorem 2.4 (Rotation of a GLWE ciphertext)**.** *Let $\mathsf{CT} = (A_0, \ldots, A_{k-1}, B) \in \mathfrak{R}_{q,N}^{k+1}$ be a GLWE ciphertext encrypting the message $M = \sum_{i=0}^{N-1} m_i X^i \in \mathfrak{R}_{p,N}$ under the GLWE secret key $\boldsymbol{S} = (S_0, \ldots, S_{k-1}) \in \mathfrak{R}_{q,N}^k$. The coefficients of the input polynomial noise E are independently sampled from a centre Gaussian distribution $\mathcal{N}_{\sigma^2}$. Let $w \in [0, 2N-1]$ be the number of position by which the input message M is rotated. By applying a rotation to all the polynomials in $\mathsf{CT} \in \mathrm{GLWE}_{\boldsymbol{S}}(M)$, we obtain a new ciphertext, $\mathsf{CT}_{\mathsf{out}}$, that encrypts $M \cdot X^w$.*

$$\mathsf{CT}_{\mathsf{out}} = \mathsf{CT} \cdot X^w = (A_0 \cdot X^w, \ldots, A_{k-1} \cdot X^w, B \cdot X^w) \in \mathrm{GLWE}_{\boldsymbol{S}}(M \cdot X^w) \subseteq \mathfrak{R}_{q,N}^{k+1}.$$

After a rotation, the noise remains unchanged and does not increase.

Proof *(Theorem 2.4). Let $\mathsf{CT} = (A_0, \ldots, A_{k-1}, B) \in \mathfrak{R}_{q,N}^{k+1}$ be the input* GLWE *ciphertext. Let $w \in [0, 2N-1]$ be the number of position by which the input message M is rotated. After a rotation, we obtain the ciphertext $\mathsf{CT}_{\mathsf{out}} = \mathsf{CT} \cdot X^w = (A_0 \cdot X^w, \ldots, A_{k-1} \cdot X^w, B \cdot X^w)$. Let us now decrypt $\mathsf{CT}_{\mathsf{out}}$ using Definition 12:*

$$\begin{aligned}\widetilde{M}_{\mathsf{out}} + E_{\mathsf{out}} &= B \cdot X^w - \sum_{i=0}^{k-1} A_i \cdot X^w \cdot S_i = \left(\sum_{i=0}^{k-1} A_i \cdot S_i + \widetilde{M} + E\right) \cdot X^w - \sum_{i=0}^{k-1} A_i \cdot X^w \cdot S_i \\ &= \widetilde{M} \cdot X^w + E \cdot X^w.\end{aligned}$$

So the output message M_{out} equals $M \cdot X^w$, and the output noise $E_{\mathsf{out}} = E \cdot X^w$ is simply rotation of the input noise, no additional noise is introduced. □

Multiplication by Integers. To multiply a message encrypted in a (G)LWE ciphertext by a constant $w \in \mathbb{Z}_q$, the methodology is similar to the rotation operation described in Theorem 2.4. However, this operation introduces additional noise. In the following, we study how to perform this operation and analyze the resulting noise growth.

Theorem 2.5 (Multiplication by Integers)**.** *Let $\mathsf{CT} = (A_0, \ldots, A_{k-1}, B) \in \mathfrak{R}_{q,N}^{k+1}$ be a GLWE ciphertext encrypting the message $M = \sum_{i=0}^{N-1} m_i X^i \in \mathfrak{R}_{p,N}$ under the GLWE secret key $\boldsymbol{S} = (S_0, \ldots, S_{k-1}) \in \mathfrak{R}_{q,N}^k$. The coefficients of the input polynomial noise E are independently sampled from a centre Gaussian distribution $\mathcal{N}_{\sigma^2}$. Let $w \in \mathbb{Z}_q$ be the constant which multiply the input message. By multiplying all the coefficients of all the polynomials in $\mathsf{CT} \in \mathrm{GLWE}_{\boldsymbol{S}}(M)$, we obtain a new ciphertext, $\mathsf{CT}_{\mathsf{out}}$, that encrypts $M \cdot w$.*

$$\mathsf{CT}_{\mathsf{out}} = \mathsf{CT} \cdot w = (A_0 \cdot w, \ldots, A_{k-1} \cdot w, B \cdot w) \in \mathrm{GLWE}_{\boldsymbol{S}}(M \cdot w) \subseteq \mathfrak{R}_{q,N}^{k+1}.$$

After this operations each coefficient of the noise polynomial follows the Gaussian distribution $\mathcal{N}_{(\sigma \cdot w)^2}$.

Proof *(Theorem 2.5). Let* $\mathsf{CT} = (A_0, \ldots, A_{k-1}, B) \in \mathfrak{R}_{q,N}^{k+1}$ *be the input* GLWE *ciphertext. Let* $w \in \mathbb{Z}_q$ *be the constant which multiplies the input message. After the operation, we obtain the ciphertext* $\mathsf{CT}_{\mathsf{out}} = \mathsf{CT} \cdot w = (A_0 \cdot w, \ldots, A_{k-1} \cdot w, B \cdot w)$*. Let us now decrypt* $\mathsf{CT}_{\mathsf{out}}$ *using Definition 12:*

$$\begin{aligned}\widetilde{M}_{\mathsf{out}} + E_{\mathsf{out}} &= B \cdot w - \sum_{i=0}^{k-1} A_i \cdot w \cdot S_i \\ &= \left(\sum_{i=0}^{k-1} A_i \cdot S_i + \widetilde{M} + E\right) \cdot w - \sum_{i=0}^{k-1} A_i \cdot w \cdot S_i \\ &= \widetilde{M} \cdot w + E \cdot w.\end{aligned}$$

So the output message M_{out} *equals* $M \cdot w$*, and the output noise* E_{out} *equals* $E \cdot w = \sum_{i=0}^{N-1} e_i \cdot w$*. Each coefficient of the output noise is multiplied by* w*.* □

Multiplication by Polynomial. The multiplication of polynomial message encrypted in a (G)LWE ciphertext by a public polynomial $P \in \mathfrak{R}_{q,N}$, can easily be done by combining the two previous theorems (Theorem 2.4 and Theorem 2.5). In the following, we study how to perform this operation and analyze the resulting noise growth.

Theorem 2.6 (Multiplication by Polynomial)**.** *Let* $\mathsf{CT} = (A_0, \ldots, A_{k-1}, B) \in \mathfrak{R}_{q,N}^{k+1}$ *be a GLWE ciphertext encrypting the message* $M = \sum_{i=0}^{N-1} m_i X^i \in \mathfrak{R}_{p,N}$ *under the GLWE secret key* $\boldsymbol{S} = (S_0, \ldots, S_{k-1}) \in \mathfrak{R}_{q,N}^k$*. The coefficients of the input polynomial noise* E *are independently sampled from a centre Gaussian distribution* $\mathcal{N}_{\sigma^2}$*. Let* $P = \sum_{i=0}^{N-1} p_i X^i \in \mathfrak{R}_{q,N}$ *be the polynomial that multiplies the input message. By multiplying all the polynomials in* $\mathsf{CT} \in \mathrm{GLWE}_{\boldsymbol{S}}(M)$ *by* P*, we obtain a new ciphertext,* $\mathsf{CT}_{\mathsf{out}}$*, that encrypts* $M \cdot P$*.*

$$\mathsf{CT}_{\mathsf{out}} = \mathsf{CT} \cdot P = (A_0 \cdot P, \ldots, A_{k-1} \cdot P, B \cdot P) \in \mathrm{GLWE}_{\boldsymbol{S}}(M \cdot P) \subseteq \mathfrak{R}_{q,N}^{k+1}.$$

After this operation, all coefficients of the noise polynomial follow the Gaussian distribution $\mathcal{N}((\sigma \cdot w)^2)$*, where* w *corresponds to the 2-norm of the coefficients of the polynomial* P*, i.e.,* $w^2 = \sum_{i=0}^{N-1} p_i^2$*.*

Proof *(Theorem 2.6). The proof of Theorem 2.6 is a composition of the proofs of Theorems 2.3, 2.4 and 2.5.* □

Homomorphic Dot Product. Finally, let us define the dot product, which corresponds to summing several LWE ciphertexts, each multiplied by different constant. This operation corresponds to a composition of Theorem 2.3 and Theorem 2.5.

Theorem 2.7 (Homomorphic Dot Product)**.** *Let* $\mathsf{ct}_i = (a_{i,0}, \ldots, a_{i,n-1}, b_i) \in \mathbb{Z}_q^{n+1}$ *be LWE ciphertexts encrypting the message* $m_i \in \mathbb{Z}_p$ *under the LWE secret key* $\boldsymbol{s} = (s_0, \ldots, s_{n-1}) \in \mathbb{Z}_q^n$*, for* $i \in [0, \alpha - 1]$*. The errors* e_i *associated with the ciphertext* ct_i *is sampled from a centre Gaussian distribution* $\mathcal{N}_{\sigma_i^2}$ *for* $i \in [0, \alpha - 1]$*. Let* $w_i \in \mathbb{Z}_q$ *be a constant used to multiply each* ct_i*. Then, the dot product corresponds to sum of all the multiplied ciphertexts:*

$$\mathsf{ct}_{\mathsf{out}} = \sum_{i=0}^{\alpha-1} \mathsf{ct}_i \cdot w_i.$$

After this operation, the output noise e_{out} *of* $\mathsf{ct}_{\mathsf{out}}$ *follows a Gaussian distribution* $\mathcal{N}_{(\sigma \cdot w)^2}$ *with* $w^2 = \sum_{i=0}^{\alpha-1} w_i^2$*.*

Proof *(Theorem 2.7). The proof of Theorem 2.7 is an adaptation and composition of the proofs of Theorems 2.3 and 2.5.* □

Remark 2.11. As for the addition, all the previous operations (Theorem 2.4,Theorem 2.5 and Theorem 2.6) can easily be generalized for RLWE, GLEV and RLEV or GGSW and RGSW ciphertexts.

The previous operations (Theorem 2.5 and Theorem 2.7) can easily be generalized for LWE, LEV and GSW ciphertexts.

2.3.3 Key Switches

In TFHE, operations can only be performed with ciphertexts encrypted under the same secret key. Key switches (**KS**) are operations that allow switching from one secret key to another. Basically, given a ciphertext $\mathsf{CT}_{\mathsf{in}} \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{in}}}(M)$, a key switch creates a new ciphertext $\mathsf{CT}_{\mathsf{out}} \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}}(M)$ encrypting the same message under a new secret key. The output secret key $\boldsymbol{S}_{\mathsf{out}}$ can have different parameters from the input secret key $\boldsymbol{S}_{\mathsf{in}}$, allowing for changes the polynomial size N, the GLWE dimension k or the LWE dimension n (in the case of LWE key switches). Unlike the previously introduced operations, key switching is less straightforward and requires public material. This public material is named key switching key (KSK) and corresponds to an encryption of the input secret key under the output secret key.

This operation is essential as it enables optimizations and enhancements in homomorphic circuit evaluations. In the following, we will study different techniques for performing key switching. We begin by introducing the LWE key switching followed by the GLWE key switching. Finally, we present packing key switching, an operation that allows packing multiple LWE ciphertexts into a single GLWE ciphertext.

Definition 15 (Key Switching Key (KSK))**.** *The KSK is the public material required to perform a key switch from an input secret key $\boldsymbol{S}_{\mathsf{in}} = (S_{\mathsf{in},0}, \ldots, S_{\mathsf{in},k_{\mathsf{in}}-1}) \in \mathfrak{R}_{q,N}^{k_{\mathsf{in}}}$ to the output secret key $\boldsymbol{S}_{\mathsf{out}} \in \mathfrak{R}_{q,N}^{k_{\mathsf{out}}}$. Each part of the KSK is a* GLEV *ciphertext that encrypts an element of the input secret key under the output secret key with noise sampled from $\mathcal{N}_{\sigma^2_{\mathsf{KSK}}}$,*

$$\mathsf{KSK} = \left\{\mathsf{KSK}_i \in \mathrm{GLEV}^{\mathcal{B},\ell}_{\boldsymbol{S}_{\mathsf{out}}}(-S_{\mathsf{in},i})\right\}_{i\in[0,k_{\mathsf{in}}-1]} \in \mathfrak{R}_{q,N}^{(k_{\mathsf{out}}+1)\cdot\ell\cdot k_{\mathsf{in}}}.$$

We denote each component of the KSK_i by $\mathsf{KSK}_{i,j} = (\boldsymbol{A_{i,j}}, B_{i,j}) \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}}\left(\frac{q}{\mathcal{B}^{j+1}} S_{\mathsf{in},i}\right)$, for all $0 \leq i < k_{\mathsf{in}}$ and for all $0 \leq j < \ell$. As presented in Remark 2.3, the notation can be adapted depending on the types of the input and output ciphertext.

LWE Key Switch. The LWE key switch allows switching an LWE ciphertext from one secret key to another, either to a different LWE secret key or to a GLWE secret key. In the case of switching to a GLWE secret key, the LWE ciphertext is transformed into a GLWE ciphertext that encrypts the input message in the constant term of the polynomial message. All other coefficients of the polynomial are set to zero.

The LWE key switch to a GLWE ciphertext is detailed in Algorithm 4 and can be easily adapted to an LWE-to-LWE key switch.

Theorem 2.8 (LWE to GLWE Key Switch (Algorithm 4))**.** *Let $\mathsf{ct}_{\mathsf{in}} \in \mathrm{LWE}_{\boldsymbol{s}_{\mathsf{in}}}(m) \subseteq \mathbb{Z}_q^{n+1}$ be an LWE ciphertext encrypting the message $m \in \mathbb{Z}_p$ under the secret key $\boldsymbol{s}_{\mathsf{in}} = (s_{\mathsf{in},0}, \ldots, s_{\mathsf{in},n-1}) \in \mathbb{Z}_q^n$ where the noise e is sampled from a Gaussian distribution $\mathcal{N}_{\sigma^2}$. Let $\boldsymbol{S}_{\mathsf{out}} = (S_{\mathsf{out},0}, \ldots, S_{\mathsf{out},k-1}) \in \mathfrak{R}_{q,N}^k$ be a GLWE secret key. Let $\mathcal{B} \in \mathbb{Z}^*$ be the base decomposition and $\ell \in \mathbb{Z}^*$ the level decomposition. Let $\mathsf{KSK} = \left\{\mathsf{KSK}_i \in \mathrm{GLEV}^{\mathcal{B},\ell}_{\boldsymbol{S}_{\mathsf{out}}}(s_i)\right\}_{i\in[0,n-1]} \in \mathfrak{R}_{q,N}^{(k+1)\cdot\ell\cdot n}$ be the key switching key as presented in Definition 15.*

Then Algorithm 4 outputs $\mathsf{CT}_{\mathsf{out}} \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}}(m) \subseteq \mathfrak{R}_{q,N}^{k+1}$, an GLWE ciphertext encrypting the input message m under the secret key $\boldsymbol{S}_{\mathsf{out}}$. The noise variance after Algorithm 4 is:

$$\mathsf{Var}(e_{\mathbf{KS}}) = \sigma_{\mathsf{in}} + n \cdot \left(\frac{q^2}{12\mathcal{B}^{2\ell}} - \frac{1}{12}\right) \cdot \left(\mathsf{Var}(s_i) + \mathbb{E}^2(s_i)\right) + \frac{n}{4} \cdot \mathsf{Var}(s_i) + n \cdot \ell \cdot \sigma^2_{\mathsf{KSK}} \cdot \frac{\mathcal{B}^2 + 2}{12}.$$

Algorithm 4: $\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{LWEKeySwitch}(\mathsf{ct}_{\mathsf{in}}, \mathsf{KSK})$

Context:
- $\boldsymbol{s}_{\mathsf{in}} = (s_{\mathsf{in},0}, \ldots, s_{\mathsf{in},n-1}) \in \mathbb{Z}_q^n$: The input LWE secret key.
- $\boldsymbol{S}_{\mathsf{out}} = (S_{\mathsf{out},0}, \ldots, S_{\mathsf{out},k-1}) \in \mathfrak{R}_{q,N}^k$: The output GLWE secret key.
- $\Delta m \in \mathbb{Z}_q$: The encoded message.
- $\mathsf{KSK}_i \in \mathrm{GLEV}_{\boldsymbol{S}_{\mathsf{out}}}^{\mathfrak{B},\ell}(-s_{\mathsf{in},i})$: Definition 13
- $\mathfrak{B} \in \mathbb{Z}^*$: The base decomposition.
- $\ell \in \mathbb{Z}^*$: The level decomposition.

Input:
- $\mathsf{ct}_{\mathsf{in}} = (a_0, \ldots, a_{n-1}, b) \in \mathrm{LWE}_{\boldsymbol{s}_{\mathsf{in}}}(m) \subseteq \mathfrak{R}_{q,N}^{k+1}$
- $\mathsf{KSK} = \{\mathsf{KSK}_i\}_{i\in[0,n-1]}$ The key switching key from $\boldsymbol{s}_{\mathsf{in}}$ to $\boldsymbol{S}_{\mathsf{out}}$

Output: $\mathsf{CT}_{\mathsf{out}} \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}}(m) \subseteq \mathfrak{R}_{q,N}^{k+1}$

$\mathsf{CT}_{\mathsf{out}} \leftarrow (0, \ldots, 0, b) - \sum_{i=0}^{n-1} \left\langle \mathsf{KSK}_i, \mathbf{Dec}^{\mathfrak{B},\ell}(a_i) \right\rangle$

return $\mathsf{CT}_{\mathsf{out}}$

The algorithmic cost of Algorithm 4 is:

$$\mathsf{Cost}_{\mathbf{KS}}^{\ell,n,k,N} = n\mathsf{Cost}_{\mathbf{Dec}}^{\mathfrak{B},\ell} + n\ell(k+1)N\mathbb{C}_{mul} + ((\ell n - 1)(k+1)N)\mathbb{C}_{add}.$$

Where $\mathbb{C}_{add}$ denotes the complexity of adding two elements of $\mathbb{Z}_q$ and $\mathbb{C}_{mul}$ denote the complexity of multiplying two elements of $\mathbb{Z}_q$.

Proof *(Theorem 2.8). The inputs of a* LWE-*to*-GLWE *key switching (Algorithm 4) are:*

- *The input LWE ciphertext:* $\mathsf{ct}_{\mathsf{in}} = (a_{\mathsf{in},0}, \ldots, a_{\mathsf{in},n-1}, b_{\mathsf{in}}) \in \mathrm{LWE}_{\boldsymbol{s}_{\mathsf{in}}}(m) \subseteq \mathbb{Z}_q^{n+1}$*, where* $b_{\mathsf{in}} = \sum_{i=0}^{n-1} a_{\mathsf{in},i} \cdot s_{\mathsf{in},i} + \Delta m + e_{\mathsf{in}}$*, with* $e_{\mathsf{in}} \hookleftarrow \mathcal{N}_{\sigma_{\mathsf{in}}^2}$.
- *The key switch key:* $\mathsf{KSK} = (\mathsf{KSK}_0, \ldots, \mathsf{KSK}_{n-1})$ *as presented in Definition 15.*

The output of this algorithm is: $\mathsf{CT}_{\mathsf{out}} = (\boldsymbol{A}_{\mathsf{out}}, B_{\mathsf{out}}) \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}}(m) \subseteq \mathfrak{R}_{q,N}^{k+1}$*. By definition, in the decomposition described in Definition 5, we have that* $\mathbf{Dec}^{\mathfrak{B},\ell}(a_{\mathsf{in},i}) = (\widetilde{a}_{\mathsf{in},i,0}, \ldots, \widetilde{a}_{\mathsf{in},i,\ell-1})$ *such that* $\widetilde{a}_{\mathsf{in},i} = \sum_{j=0}^{\ell-1} \frac{q}{\mathfrak{B}^{j+1}} \widetilde{a}_{\mathsf{in},i,j}$*, for all* $0 \leq i < k_{\mathsf{in}}$.
Let define $\bar{a}_{\mathsf{in},i} = a_{\mathsf{in},i} - \widetilde{a}_{\mathsf{in},i}$, $|\bar{a}_{\mathsf{in},i}| \in \left[\frac{-q}{2\mathfrak{B}^\ell}, \frac{q}{2\mathfrak{B}^\ell}\right)$*. So we have that their expectations and variances are respectively* $\mathbb{E}(\bar{a}_{\mathsf{in},i}) = -\frac{1}{2}$, $\mathsf{Var}(\bar{a}_{\mathsf{in},i}) = \frac{q^2}{12\mathfrak{B}^{2\ell}} - \frac{1}{12}$, $\mathbb{E}(\widetilde{a}_{\mathsf{in},i}) = -\frac{1}{2}$ *and* $\mathsf{Var}(\widetilde{a}_{\mathsf{in},i}) = \frac{\mathfrak{B}^2-1}{12}$.

Let us now decrypt $\mathsf{CT}_{\mathsf{out}}$ *using Definition 12.*

$$\begin{aligned}
\widetilde{M}_{\mathsf{out}} + E_{\mathsf{out}} &= B_{\mathsf{out}} - \sum_{i=0}^{n-1} A_{\mathsf{out}} \boldsymbol{S}_{\mathsf{out}} = \langle (A_{\mathsf{out}}, B_{\mathsf{out}}), (-\boldsymbol{S}_{\mathsf{out}}, 1) \rangle \\
&= \left\langle (\mathbf{0}, b_{\mathsf{in}}) - \sum_{i=0}^{n-1} \mathbf{Dec}^{(\mathfrak{B},\ell)}(a_{\mathsf{in},i}) \cdot \mathsf{KSK}_i, (-\boldsymbol{S}_{\mathsf{out}}, 1) \right\rangle \\
&= b_{\mathsf{in}} - \sum_{i=0}^{n-1} \sum_{j=0}^{\ell-1} \widetilde{a}_{\mathsf{in},i,j} \langle \mathsf{KSK}_{i,j}, (-\boldsymbol{S}_{\mathsf{out}}, 1) \rangle \\
&= b_{\mathsf{in}} - \sum_{i=0}^{n-1} \sum_{j=0}^{\ell-1} \widetilde{a}_{\mathsf{in},i,j} \left(\frac{q}{\mathfrak{B}^{j+1}} s_{\mathsf{in},i} + E_{\mathsf{ksk},i,j} \right) \\
&= b_{\mathsf{in}} - \sum_{i=0}^{n-1} \widetilde{a}_{\mathsf{in},i} s_{\mathsf{in},i} - \sum_{i=0}^{n-1} \sum_{j=0}^{\ell-1} \widetilde{a}_{\mathsf{in},i,j} \cdot E_{\mathsf{ksk},i,j}.
\end{aligned}$$

Now let us focus on the constant coefficient of the previous equation:

$$b_{\mathsf{in}} - \sum_{i=0}^{n-1} \widetilde{a}_{\mathsf{in},i} s_{\mathsf{in},i} - \sum_{i=0}^{n-1}\sum_{j=0}^{\ell-1} \widetilde{a}_{\mathsf{in},i,j} \cdot e_{\mathsf{ksk},i,j,0} = b_{\mathsf{in}} - \sum_{i=0}^{n-1} (a_{\mathsf{in},i} - \bar{a}_{\mathsf{in},i}) s_{\mathsf{in},i} - \sum_{i=0}^{n-1}\sum_{j=0}^{\ell-1} \widetilde{a}_{\mathsf{in},i,j} \cdot e_{\mathsf{ksk},i,j,0}$$
$$= \widetilde{m} + e + \sum_{i=0}^{n-1} \bar{a}_{\mathsf{in},i} s_{\mathsf{in},i} - \sum_{i=0}^{n-1}\sum_{j=0}^{\ell-1} \widetilde{a}_{\mathsf{in},i,j} \cdot e_{\mathsf{ksk},i,j,0}.$$

We can now isolate the output error for the constant coefficient and remove the message coefficient. We obtain that the output error is:

$$e_{\mathsf{out}} = e + \sum_{i=0}^{n-1} \bar{a}_{\mathsf{in},i} s_{\mathsf{in},i} + \sum_{i=0}^{n-1}\sum_{j=0}^{\ell-1} \widetilde{a}_{\mathsf{in},i,j} \cdot e_{\mathsf{ksk},i,j,0}.$$

Then, we obtain:

$$\begin{aligned}
\mathsf{Var}(e_{\mathsf{out}}) &= \mathsf{Var}(e) + n\mathsf{Var}(\bar{a}_{\mathsf{in},i} s_{\mathsf{in},i}) + n\ell\mathsf{Var}(\widetilde{a}_{\mathsf{in},i,j} \cdot e_{\mathsf{ksk},i,j,0}) \\
&= \sigma_{\mathsf{in}}^2 + n(\mathsf{Var}(\bar{a}_{\mathsf{in},i}) \cdot \mathsf{Var}(s_{\mathsf{in},i}) + \mathsf{Var}(\bar{a}_{\mathsf{in},i}) \cdot \mathbb{E}^2(s_{\mathsf{in},i}) + \mathbb{E}^2(\bar{a}_{\mathsf{in},i}) \cdot \mathsf{Var}(s_{\mathsf{in},i})) \\
&\quad + n\ell(\mathsf{Var}(\widetilde{a}_{\mathsf{in},i,j})\mathsf{Var}(e_{\mathsf{ksk},i,j,0}) + \mathsf{Var}(\widetilde{a}_{\mathsf{in},i,j})\mathbb{E}^2(e_{\mathsf{ksk},i,j,0}) + \mathbb{E}^2(\widetilde{a}_{\mathsf{in},i,j})\mathsf{Var}(e_{\mathsf{ksk},i,j,0})) \\
&= \sigma_{\mathsf{in}} + n \cdot \left(\frac{q^2}{12\mathfrak{B}^{2\ell}} - \frac{1}{12}\right) \cdot \big(\mathsf{Var}(s_i) + \mathbb{E}^2(s_i)\big) + \frac{n}{4} \cdot \mathsf{Var}(s_i) + n \cdot \ell \cdot \sigma_{\mathsf{KSK}}^2 \cdot \frac{\mathfrak{B}^2 + 2}{12}.
\end{aligned}$$

□

GLWE Key Switch. The GLWE key switch allows switching a GLWE ciphertext from one secret key to another GLWE secret key. The procedure is similar to the LWE key switch presented earlier, and Theorem 2.8 can be easily adapted to the GLWE key switch algorithm. Since both algorithms are closely related, we simply refer to key switch for both of them when the context is clear enough.

Algorithm 5: $\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{GLWEKeySwitch}(\mathsf{CT}_{\mathsf{in}}, \mathsf{KSK})$

Context:
- $\boldsymbol{S}_{\mathsf{in}} = (S_{\mathsf{in},0}, \ldots, S_{\mathsf{in},k_{\mathsf{in}}-1}) \in \mathfrak{R}_{q,N}^{k_{\mathsf{in}}}$: The input GLWE secret key.
- $\boldsymbol{S}_{\mathsf{out}} = (S_{\mathsf{out},0}, \ldots, S_{\mathsf{out},k-1}) \in \mathfrak{R}_{q,N}^{k_{\mathsf{out}}}$: The output GLWE secret key.
- $\Delta M \in \mathfrak{R}_{q,N}$: The encoded message.
- $\mathsf{KSK}_i \in \mathrm{GLEV}_{\boldsymbol{S}_{\mathsf{out}}}^{\mathfrak{B},\ell}(-S_{\mathsf{in},i})$: Definition 13
- $\mathfrak{B} \in \mathbb{Z}^*$: The base decomposition.
- $\ell \in \mathbb{Z}^*$: The level decomposition.

Input:
- $\mathsf{CT}_{\mathsf{in}} = (A_0, \ldots, A_{k_{\mathsf{in}}-1}, B) \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{in}}}(M) \subseteq \mathfrak{R}_{q,N}^{k_{\mathsf{out}}+1}$
- $\mathsf{KSK} = \{\mathsf{KSK}_i\}_{i \in [0, k_{\mathsf{in}}-1]}$ The key switching key from $\boldsymbol{S}_{\mathsf{in}}$ to $\boldsymbol{S}_{\mathsf{out}}$

Output: $\mathsf{CT}_{\mathsf{out}} \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}}(M) \subseteq \mathfrak{R}_{q,N}^{k_{\mathsf{out}}+1}$

1 $\mathsf{CT}_{\mathsf{out}} \leftarrow (0, \ldots, 0, B) - \sum_{i=0}^{k_{\mathsf{in}}-1} \left\langle \mathsf{KSK}_i, \mathbf{Dec}^{\mathfrak{B},\ell}(A_i) \right\rangle$

2 **return** $\mathsf{CT}_{\mathsf{out}}$

Remark 2.12. By modifying the key switching key, one can perform a key switch (Algorithm 4 and Algorithm 5) that simultaneously evaluates a private function. This operation is

named PrivateKS. In this case, to apply a linear function f during the key switch, the $\mathsf{KSK} = \left\{\mathsf{KSK}_i \in \mathrm{GLEV}^{\mathcal{B},\ell}_{\boldsymbol{S}_{\mathsf{out}}}(-S_{\mathsf{in},i})\right\}_{i\in[0,k_{\mathsf{in}}-1]} \in \mathfrak{R}^{(k_{\mathsf{out}}+1)\cdot\ell\cdot k_{\mathsf{in}}}_{q,N}$ (Definition 15) become a Private Functional Key Switching Key $\mathsf{PKSK} = \left\{\mathsf{KSK}_i \in \mathrm{GLEV}^{\mathcal{B},\ell}_{\boldsymbol{S}_{\mathsf{out}}}(-f\cdot S_{\mathsf{in},i})\right\}_{i\in[0,k_{\mathsf{in}}-1]} \cup \left\{\mathsf{KSK}_{k_{\mathsf{in}}} \in \mathrm{GLEV}^{\mathcal{B},\ell}_{\boldsymbol{S}_{\mathsf{out}}}(f)\right\} \in \mathfrak{R}^{(k_{\mathsf{out}}+1)\cdot\ell\cdot(k_{\mathsf{in}}+1)}_{q,N}$ and we need to perform a dot product between the decomposed B and $\mathsf{KSK}_{k_{\mathsf{in}}}$.

Packing Key Switch. The packing key switch is an operation that takes as input several LWE ciphertexts and outputs a GLWE ciphertext encrypting the messages from the input ciphertexts. The signature of the packing key switch is as follows:

$$\mathsf{CT} \in \mathrm{GLWE}_{\boldsymbol{S}}\left(\sum_{i=0}^{\alpha} m_i X^{j_i}\right) \leftarrow \mathsf{PackingKS}\left(\{\mathsf{ct}_i \in \mathrm{LWE}_{\boldsymbol{s}}(m_i)\}_{i\in[0,\alpha]}, \{j_i\}_{i\in[0,\alpha]}, \mathsf{KSK}\right).$$

The packing key switch can be easily described using the previously introduced algorithm. Indeed, the process to perform a packing key switch can be summarized by first performing LWE to GLWE key switch (Algorithm 4), then a multiplication by a power of X (Theorem 2.5) to put the constant coefficient at the desired position, and finally an addition (Algorithm 3) to add all the coefficient together.

Theorem 2.9 (LWE to GLWE packing Key Switch)**.** *Let start with only one ciphertext (this case is similar than only on LWE to GLWE key switch).*

Let $\mathsf{ct}_{\mathsf{in}} \in \mathrm{LWE}_{\boldsymbol{s}_{\mathsf{in}}}(m) \subseteq \mathbb{Z}^{n+1}_q$ be an LWE ciphertext encrypting the message $m \in \mathbb{Z}_p$ under the secret key $\boldsymbol{s}_{\mathsf{in}} = (s_{\mathsf{in},0},\ldots,s_{\mathsf{in},n-1}) \in \mathbb{Z}^n_q$ where the noise e is sample from a Gaussian distribution $\mathcal{N}_{\sigma^2}$. Let $\boldsymbol{S}_{\mathsf{out}} = (S_{\mathsf{out},0},\ldots,S_{\mathsf{out},k-1}) \in \mathfrak{R}^k_{q,N}$ be a GLWE secret key. Let $\mathcal{B} \in \mathbb{Z}^*$ be the base decomposition and $\ell \in \mathbb{Z}^*$ the level decomposition. Let $\mathsf{KSK} = \left\{\mathsf{KSK}_i \in \mathrm{GLEV}^{\mathcal{B},\ell}_{\boldsymbol{S}_{\mathsf{out}}}(s_{\mathsf{in},i})\right\}_{i\in[0,n-1]} \in \mathfrak{R}^{(k+1)\cdot\ell\cdot(n-1)}_{q,N}$ be the key switching key as presented in Definition 15. Then the packing key switch outputs $\mathsf{CT}_{\mathsf{out}} \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}}(m) \subseteq \mathfrak{R}^{k+1}_{q,N}$ an GLWE ciphertext encrypting the input message m under the secret key $\boldsymbol{S}_{\mathsf{out}}$.
We can distinguish two variances, the one with the input message:

$$\mathsf{Var}(e_{\mathsf{fill}}) = \sigma_{\mathsf{in}} + n\cdot\left(\frac{q^2}{12\mathcal{B}^{2\ell}} - \frac{1}{12}\right)\cdot\left(\mathsf{Var}(s_{\mathsf{in},i}) + \mathbb{E}^2(s_{\mathsf{in},i})\right) + \frac{n}{4}\cdot\mathsf{Var}(s_{\mathsf{in},i}) + n\cdot\ell\cdot\sigma^2_{\mathsf{KSK}}\cdot\frac{\mathcal{B}^2+2}{12},$$

and the variance without the input message:

$$\mathsf{Var}(e_{\mathsf{empty}}) = n\cdot\ell\cdot\sigma^2_{\mathsf{KSK}}\cdot\frac{\mathcal{B}^2+2}{12}.$$

When we pack $1 \le \alpha < N$ coefficients, we obtain the following noise variance:

$$\mathsf{Var}(e_{\mathsf{fill}})^{\alpha} = \mathsf{Var}(e_{\mathsf{fill}}) + (\alpha-1)\mathsf{Var}(e_{\mathsf{empty}}),$$
$$\mathsf{Var}(e_{\mathsf{empty}})^{\alpha} = \alpha\left(n\cdot\ell\cdot\sigma^2_{\mathsf{KSK}}\cdot\frac{\mathcal{B}^2+2}{12}\right).$$

Finally the cost of a packing key switch is:

$$\mathsf{Cost}^{\alpha,\ell,n,k,N}_{\mathsf{PackingKS}} = \alpha\cdot\mathsf{Cost}^{\ell,n,k,N}_{\mathbf{KS}} + (\alpha-1)\mathsf{Cost}^{k,N}_{\mathsf{Add}}.$$

Proof *(Theorem 2.9). The proof of Theorem 2.9 is not provided but can easily be obtained by combining the proof of Theorem 2.8 and the proof of Theorem 2.3.* □

Remark 2.13. Some algorithms use both key switching and packing key switching. To differentiate the associated public material, the key switching key for standard key switching keeps its usual name, while the key switching key for packing key switching is called the Packing Key Switching Key, and is referred to as PKSK

2.4 Programmable Bootstrapping

In the previous section, we saw that most homomorphic operations cause noise growth, which can eventually become problematic. Indeed, after a certain number of operations, if this growth is not managed, the accumulated noise may corrupt the message, making decryption wrong.

In this section, we present the building blocks required to perform TFHE's bootstrapping. As introduced in [CGGI16a, CGGI17, CGGI20], bootstrapping is a technique used to control and reduce noise during computation. In addition to noise reduction, TFHE bootstrapping allows the evaluation of univariate functions, which is why it is often referred to as programmable (or functional) bootstrapping (PBS).

At a high level, the goal of bootstrapping is to reduce the noise by homomorphically evaluating the decryption function (i.e., this reveals no information about the message of the input ciphertext) to produce a less noisy ciphertext as output.

In the following, we describe the core building blocks of TFHE bootstrapping: Modulus Switching, Blind Rotation, and Sample Extraction. We then present the complete programmable bootstrapping procedure and conclude with a discussion on efficient encoding strategies for TFHE based on the bootstrapping strategies.

2.4.1 Programmable Bootstrapping Building Blocks

The bootstrapping introduced in [CGGI16a, CGGI17, CGGI20] is done in three distinct steps. First, the modulus of the input LWE ciphertext is adjusted by performing an operation called Modulus Switch (**MS**). Then a Blind Rotation (**BR**) is performed, this operation consisting in chaining several External Products (**EP**) (products between GLWE and GGSW ciphertexts). This step is the most expensive part of the PBS and corresponds to the linear part of the decryption (see Definition 12). Finally, the last step is Sample Extraction (**SE**), an operation that enables the extraction of an LWE ciphertext from a GLWE ciphertext.

Modulus Switch. The modulus switch takes as input an LWE ciphertext in $\mathbb{Z}_q^n$ and changes the modulus so that the output LWE ciphertext lies in $\mathbb{Z}_w^n$ while keeping the most significant bit. In the context of the PBS, $w = 2N$ where N is the polynomial size of GLWE ciphertext used during the PBS.

Algorithm 6: $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{ModulusSwitch}(\mathsf{ct}_{\mathsf{in}}, w)$

Context: $\begin{cases} w \in \mathbb{Z} \text{ with } w < q \\ \boldsymbol{s} = (s_0, \ldots, s_{n-1}) : \text{ the LWE secret key} \end{cases}$

Input: $\begin{cases} \mathsf{ct}_{\mathsf{in}} = (a_{\mathsf{in},0}, \ldots, a_{\mathsf{in},n-1}, b) \in \mathrm{GLWE}_{\boldsymbol{s}}(m) \subseteq \mathbb{Z}_q^{n+1} \\ w : \text{ the new modulus} \end{cases}$

Output: $\left\{ \mathsf{ct}_{\mathsf{out}} \in \mathrm{LWE}_{\boldsymbol{s}} \left(\left\lfloor \frac{m \cdot w}{q} \right\rceil \right) \subseteq \mathbb{Z}_w^{n+1} \right.$

for $i \in [0, n-1]$ **do**
  $a_{\mathsf{out},i} \leftarrow \left(\left\lfloor \frac{a_{\mathsf{in},i} \cdot w}{q} \right\rceil \right) \mod w$
$b_{\mathsf{out}} \leftarrow \left(\left\lfloor \frac{b_{\mathsf{in}} \cdot w}{q} \right\rceil \right) \mod w$
return $\mathsf{ct}_{\mathsf{out}} = (a_{\mathsf{out},0}, \ldots, a_{\mathsf{out},n-1}, b_{\mathsf{out}})$

Theorem 2.10 (Modulus Switch (Algorithm 6))**.** *Let $w \in \mathbb{Z}$ be the new modulus with $w < q$. Let $\mathsf{ct}_{\mathsf{in}} \in \mathrm{LWE}_{\boldsymbol{s}}(m) \subseteq \mathbb{Z}_q^{n+1}$ be an LWE ciphertext encrypting the message $m \in \mathbb{Z}_p$ under the secret key $\boldsymbol{s} = (s_0, \ldots, s_{n-1}) \in \mathbb{Z}_q^n$ where the noise e_{in} is sampled from a centered Gaussian distribution $\mathcal{N}_{\sigma^2}$. Then Algorithm 6 outputs $\mathsf{ct}_{\mathsf{out}} \in \mathrm{LWE}_{\boldsymbol{s}} \left(\left\lfloor \frac{m \cdot w}{q} \right\rceil \right) \subseteq \mathbb{Z}_w^{n+1}$.*

After the modulus switch, the noise variance of $\mathsf{ct}_{\mathsf{out}}$ *is:*

$$\mathsf{Var}(\mathsf{ModulusSwitch}) = \frac{\sigma_{\mathsf{in}}^2 w^2}{q^2} + \frac{1}{12} - \frac{w^2}{12q} + \frac{n}{24} + \frac{nw^2}{48q^2}.$$

And the cost of Algorithm 6 is:

$$\mathsf{Cost}^{n,q,w}_{\mathsf{ModulusSwitch}} = (n+1)\mathbb{C}^{q,w}_{\mathsf{Round}}.$$

Where $\mathbb{C}^{q,w}_{\mathsf{Round}}$ *denotes the complexity of performing a modulus switch from* q *to* w *on a single element of* $\mathbb{Z}_q$*.*

Proof *(Theorem 2.10). Let* $\mathsf{ct}_{\mathsf{in}} \in \mathrm{LWE}_{\boldsymbol{s}}(m) \subseteq \mathbb{Z}_q^{n+1}$ *be the input ciphertext. Let* $\boldsymbol{s}$ *be the secret key. Let* $\mathsf{ct}_{\mathsf{out}} \in \mathrm{LWE}_{\boldsymbol{s}}\left(\left\lfloor \frac{m \cdot w}{q} \right\rceil\right) \subseteq \mathbb{Z}_w^{n+1}$ *be the output ciphertext of Algorithm 6.*

We note $a_{\mathsf{out},i} = \left\lfloor \frac{a_{\mathsf{in},i} \cdot w}{q} \right\rceil = \frac{a_{\mathsf{in},i} \cdot w}{q} + \bar{a}_i$*, then we have* $a_{\mathsf{out},i} \in \mathfrak{U}\left(\left[\frac{-w}{2}, \frac{w}{2}\right)\right)$ *and* $\bar{a}_i \in \frac{w}{q}\mathfrak{U}\left(\left[\frac{-q}{2w}, \frac{q}{2w}\right)\right)$*. So we have that* $\mathsf{Var}(\bar{a}_i) = \frac{1}{12} - \frac{w^2}{12q}$ *and* $\mathbb{E}(\bar{a}_i) = \frac{-w}{2q}$*. We use the same notation for* b_{out} *and* $\bar{b}$ *follows the same distribution.*

Let us now decrypt $\mathsf{ct}_{\mathsf{out}}$ *by using Definition 12:*

$$\begin{aligned}
\widetilde{m}_{\mathsf{out}} + e_{\mathsf{out}} &= b_{\mathsf{out}} - \sum_{i=0}^{n-1} a_{\mathsf{in},i} s_i = \left\lfloor \frac{b_{\mathsf{in}} \cdot w}{q} \right\rceil - \sum_{i=0}^{n-1} \left\lfloor \frac{a_{\mathsf{in},i} \cdot w}{q} \right\rceil s_i \\
&= \frac{b_{\mathsf{in}} \cdot w}{q} + \bar{b}_i - \sum_{i=0}^{n-1} \left(\frac{a_{\mathsf{in},i} \cdot w}{q} + \bar{a}_i \right) s_i = \frac{w}{q} \left(b_{\mathsf{in}} - \sum_{i=0}^{n-1} a_{\mathsf{in},i} s_i \right) + \bar{b} - \sum_{i=0}^{n-1} \bar{a}_i s_i \\
&= \frac{w}{q} \widetilde{m} + \frac{w}{q} e_{\mathsf{in}} + \bar{b} - \sum_{i=0}^{n-1} \bar{a}_i s_i.
\end{aligned}$$

We can now isolate the error:

$$\begin{aligned}
\mathsf{Var}(e_{\mathsf{out}}) &= \mathsf{Var}\left(\frac{w}{q} e_{\mathsf{in}} + \bar{b} - \sum_{i=0}^{n-1} \bar{a}_i s_i \right) \\
&= \frac{w^2 \sigma_{\mathsf{in}}}{q^2} + \mathsf{Var}(\bar{b}) + n \cdot \mathsf{Var}(\bar{a}_i) \cdot (\mathsf{Var}(s_i) + \mathbb{E}^2(s_i)) + n \cdot \mathbb{E}^2(\bar{a}.\mathsf{Var}(s_i)) \\
&= \frac{w^2 \sigma_{\mathsf{in}}}{q^2} + \frac{1}{12} - \frac{w^2}{12q^2} + n \cdot \left(\frac{1}{12} - \frac{w^2}{12q^2} \right) \cdot \frac{1}{2} + n \cdot \frac{w^2}{4q^2} \cdot \frac{1}{4}.
\end{aligned}$$

□

External Products. The external product [GINX16] is an operation that takes as input a (G)LWE ciphertext and a (G)GSW ciphertext. Its goal is to compute a multiplication between the two encrypted messages. The procedure is similar to the one used for key switching, as described in Algorithm 5.

In this manuscript, we focus exclusively on the external product as the method for performing multiplication between two encrypted values (between a GLWE ciphertext and a GGSW ciphertext). However, in the literature, other approaches exist, such as those presented in [FV12, CLOT21], which enable multiplication between two GLWE ciphertexts. This alternative method consists of performing a tensor product followed by a relinearization (algorithm permitting to recover the input secret key). Although this technique is more costly than the external product and requires a relinearization key, it does not rely on GGSW ciphertexts. This procedure is latter detailed in Algorithm 20.

Algorithm 7: $\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{ExternalProduct}(\mathsf{CT}, \overline{\overline{\mathsf{CT}}})$

Context:
- $\boldsymbol{S} = (S_0, \ldots, S_{k-1})$: the GLWE secret key
- $\Delta M \in \mathfrak{R}_{q,N}$: The encoded message of CT
- $P \in \mathfrak{R}_{p,N}$: The encoded message of $\overline{\overline{\mathsf{CT}}}$
- $(\mathfrak{B}, \ell) \in (\mathbb{Z}^*)^2$: The (base, level) decomposition.
- $\overline{\overline{\mathsf{CT}}} \in \mathrm{GGSW}_{\boldsymbol{S}}^{\mathfrak{B},\ell}(P) \subseteq \mathfrak{R}_{q,N}^{(k+1)\ell\cdot(k+1)}$: Definition 14
- $\overline{\mathsf{CT}}_{i\in[0,k-1]} \in \mathrm{GLEV}_{\boldsymbol{S}}^{\mathfrak{B},\ell}(-P \cdot S_i) \subseteq \mathfrak{R}_{q,N}^{(k+1)\ell}$: Definition 13
- $\overline{\mathsf{CT}}_k \in \mathrm{GLEV}_{\boldsymbol{S}}^{\mathfrak{B},\ell}(P) \subseteq \mathfrak{R}_{q,N}^{(k+1)\ell}$: Definition 13
- $\mathsf{CT} \in \mathrm{GLWE}_{\boldsymbol{S}}(M) \subseteq \mathfrak{R}_{q,N}^{k+1}$

Input:
- $\mathsf{CT} = (A_0, \ldots, A_{k_{\mathsf{in}}-1}, B)$
- $\overline{\overline{\mathsf{CT}}} = (\overline{\mathsf{CT}}_0, \ldots, \overline{\mathsf{CT}}_{k-1}, \overline{\mathsf{CT}}_k)$

Output: $\mathsf{CT}_{\mathsf{out}} \in \mathrm{GLWE}_{\boldsymbol{S}}(M \cdot P) \subseteq \mathfrak{R}_{q,N}^{k+1}$

1 $\mathsf{CT}_{\mathsf{out}} \leftarrow \left\langle \overline{\mathsf{CT}}_k, \mathbf{Dec}^{\mathfrak{B},\ell}(B) \right\rangle - \sum_{i=0}^{k_{\mathsf{in}}-1} \left\langle \overline{\mathsf{CT}}_i, \mathbf{Dec}^{\mathfrak{B},\ell}(A_i) \right\rangle$

2 **return** $\mathsf{CT}_{\mathsf{out}}$

Theorem 2.11 (External Products (Algorithm 7))**.** *Let $\boldsymbol{S} = (S_0, \ldots, S_{k-1}) \in \mathfrak{R}_{q,N}^k$ be the GLWE secret key. Let $\mathsf{CT} = (A_0, \ldots, A_{k-1}, B) \in \mathrm{GLWE}_{\boldsymbol{S}}(M) \subseteq \mathfrak{R}_{p,N}^{k+1}$ be a GLWE ciphertext encrypting the message $M = \sum_{i=0}^{N-1} m_i X^i \in \mathfrak{R}_{p,N}$ with the noise sample from the centered Gaussian distribution $\mathcal{N}_{\sigma^2}$. Let $\overline{\overline{\mathsf{CT}}} = (\overline{\mathsf{CT}}_0, \ldots, \overline{\mathsf{CT}}_{k-1}, \overline{\mathsf{CT}}_k) \in \mathrm{GGSW}_{\boldsymbol{S}}^{\mathfrak{B},\ell}(P) \subseteq \mathfrak{R}_{q,N}^{(k+1)\ell\cdot(k+1)}$ be a GGSW ciphertext encrypting the message $P = \sum_{i=0}^{N-1} p_i X^i \subseteq \mathfrak{R}_{p,N}$ such that $\overline{\mathsf{CT}}_{j\in[0,k-1]} = (\mathsf{CT}_{j,0}, \ldots, \mathsf{CT}_{j,\ell-1}) \in \mathrm{GLEV}_{\boldsymbol{S}}^{\mathfrak{B},\ell}(-P \cdot S_j) \subseteq \mathfrak{R}_{q,N}^{(k+1)\ell}$ and with $\overline{\mathsf{CT}}_k = (\mathsf{CT}_{k,0}, \ldots, \mathsf{CT}_{k,\ell-1}) \in \mathrm{GLEV}_{\boldsymbol{S}}^{\mathfrak{B},\ell}(P) \subseteq \mathfrak{R}_{q,N}^{(k+1)\ell}$ where the noise sample from the centered Gaussian distribution $\mathcal{N}_{\sigma^2_{\mathrm{GGSW}}}$.*

Then Algorithm 7 returns an GLWE ciphertext $\mathsf{CT}_{\mathsf{out}} = (A_0, \ldots, A_{k-1}, B) \in \mathrm{GLWE}_{\boldsymbol{S}}(M \cdot P) \subseteq \mathfrak{R}_{q,N}^{k+1}$. The cost of this Algorithm is:

$$\begin{aligned}\mathsf{Cost}^{\ell,k,N}_{\mathsf{ExternalProduct}} &= (k+1)\cdot N \cdot \mathsf{Cost}^{\ell}_{\mathbf{Dec}} + \ell\cdot(k+1)\cdot \mathsf{Cost}^{N}_{\mathsf{FFT}} \\ &+ (k+1)\cdot\ell\cdot(k+1)\cdot N \cdot \mathsf{Cost}^{N}_{\mathsf{mulFFT}} \\ &+ (k+1)\cdot(\ell\cdot(k+1)-1)\cdot N \cdot \mathsf{Cost}^{N}_{\mathsf{addFFT}} \\ &+ (k+1)\cdot \mathsf{Cost}^{N}_{\mathsf{iFFT}}.\end{aligned}$$

Where $\mathsf{Cost}^{N}_{\mathsf{FFT}}$ and $\mathsf{Cost}^{N}_{\mathsf{iFFT}}$ denote the cost of performing a FFT (resp., an inverse FFT) of size N. $\mathsf{Cost}^{N}_{\mathsf{addFFT}}$, $\mathsf{Cost}^{N}_{\mathsf{mulFFT}}$ denotes the cost of performing addition and multiplication of size N in the Fourier domain.

In the case where the GGSW ciphertext encrypts a constant polynomial with a message uniformly in $\{0,1\}$, the variance after the external product is:

$$\begin{aligned}\mathsf{Var}(\mathsf{ExternalProduct}) &= \ell\cdot(k+1)\cdot N\cdot\frac{\mathfrak{B}^2+2}{12}\sigma^2_{\mathrm{GGSW}} + \frac{\sigma^2_{\mathsf{in}}}{2} + \frac{kN}{8}\mathsf{Var}(s_i) \\ &+ \frac{q^2-\mathfrak{B}^{2\ell}}{24\mathfrak{B}^{2\ell}}\cdot\left(1+kN\cdot\left(\mathsf{Var}(s_i)+\mathbb{E}^2(s_i)\right)\right) + \frac{1}{16}\cdot(1-kN\cdot\mathbb{E}(s_i))^2.\end{aligned}$$

In the following, we denote the external product as $\boxdot$, such that:

$$\mathsf{ExternalProduct}\left(\mathsf{CT}, \overline{\overline{\mathsf{CT}}}\right) = \mathsf{CT} \boxdot \overline{\overline{\mathsf{CT}}}.$$

Proof *(Theorem 2.11). The inputs of the external product (Algorithm 7) are:*

- *The input GLWE ciphertext:* $\mathsf{CT}_{\mathsf{in}} = (A_{\mathsf{in},0}, \ldots, A_{\mathsf{in},k-1}, B_{\mathsf{in}}) \in \mathrm{GLWE}_{\boldsymbol{S}}(M_{\mathsf{in}}) \subseteq \mathfrak{R}_{q,N}^{k+1}$, *where* $B_{\mathsf{in}} = \sum_{i=0}^{k-1} A_{\mathsf{in},i} \cdot S_{\mathsf{in},i} + \tilde{M}_{\mathsf{in}} + E_{\mathsf{in}}$, *with* $E_{\mathsf{in}} = \sum_{i=0}^{N-1} e_{\mathsf{in},i} X^i$ *such that* $e_{\mathsf{in},i} \hookleftarrow \mathcal{N}_{\sigma^2}$.

- *The input GGSW ciphertext:* $\overline{\overline{\mathsf{CT}}} = \left(\overline{\mathsf{CT}}_0, \ldots, \overline{\mathsf{CT}}_{k-1}, \overline{\mathsf{CT}}_k\right)$ *encrypting the message* $p \hookleftarrow \{0,1\}$ *such that* $\overline{\mathsf{CT}}_{j\in[0,k-1]} = (\mathsf{CT}_{j,0}, \ldots, \mathsf{CT}_{j,\ell-1}) \in \mathrm{GLEV}_{\boldsymbol{S}}^{\mathcal{B},\ell}(-p \cdot S_j) \subseteq \mathfrak{R}_{q,N}^{(k+1)\ell}$ *and with* $\overline{\mathsf{CT}}_k = (\mathsf{CT}_{k,0}, \ldots, \mathsf{CT}_{k,\ell-1}) \in \mathrm{GLEV}_{\boldsymbol{S}}^{\mathcal{B},\ell}(p) \subseteq \mathfrak{R}_{q,N}^{(k+1)\ell}$. *Each noise coefficient in the ciphertext composing the* GGSW *ciphertext follow the centered Gaussian distribution* $\mathcal{N}_{\sigma_{\mathrm{GGSW}}}$.

The output of this algorithm is: $\mathsf{CT}_{\mathsf{out}} = (\boldsymbol{A}_{\mathsf{out}}, B_{\mathsf{out}}) \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}}(Mp) \subseteq \mathfrak{R}_{q,N}^{k+1}$. *By definition, with the decomposition described in Definition 5, we have that* $\mathbf{Dec}^{\mathcal{B},\ell}(A_{\mathsf{in},i}) = \left(\widetilde{A}_{\mathsf{in},i}^0, \ldots, \widetilde{A}_{\mathsf{in},i}^{\ell-1}\right)$ *with* $\widetilde{A}_{\mathsf{in},i} = \sum_{j=0}^{N-1} \sum_{\iota=0}^{\ell-1} \widetilde{a}_{\mathsf{in},i,j}^{\iota} \frac{q}{\mathcal{B}^{j+1}} X^{\iota}$, *such that* $\mathbf{Dec}^{\mathcal{B},\ell}(a_{\mathsf{in},i,j}) = \left(\widetilde{a}_{\mathsf{in},i,j}^0, \ldots, \widetilde{a}_{\mathsf{in},i,j}^{\ell-1}\right)$ *and* $\widetilde{a}_{\mathsf{in},i,j} = \sum_{\iota=0}^{\ell-1} \widetilde{a}_{\mathsf{in},i,j}^{\iota}$, *for all* $0 \leq i < k$ *and for all* $0 \leq j < N$. *And we have that* $\mathbf{Dec}^{\mathcal{B},\ell}(B_{\mathsf{in}}) = \left(\widetilde{B}_{\mathsf{in}}^0, \ldots, \widetilde{B}_{\mathsf{in}}^{\ell-1}\right)$ *with* $\widetilde{B}_{\mathsf{in}} = \sum_{j=0}^{N-1} \sum_{\iota=0}^{\ell-1} \widetilde{b}_{\mathsf{in},j}^{\iota} \frac{q}{\mathcal{B}^{j+1}} X^{\iota}$, *such that* $\mathbf{Dec}^{\mathcal{B},\ell}(b_{\mathsf{in},j}) = \left(b_{\mathsf{in},j}^0, \ldots, b_{\mathsf{in},j}^{\ell-1}\right)$ *and* $\widetilde{b}_{\mathsf{in},j} = \sum_{\iota=0}^{\ell-1} \widetilde{b}_{\mathsf{in},j}^{\iota}$, *for all* $0 \leq j < N$.

Let define $\bar{A}_{\mathsf{in},i} = A_{\mathsf{in},i} - \widetilde{A}_{\mathsf{in},i}$ *and* $\bar{a}_{\mathsf{in},i,j} = a_{\mathsf{in},i,j} - \widetilde{a}_{\mathsf{in},i,j}$, $|\bar{a}_{\mathsf{in},i,j}| \in \left[\frac{-q}{2\mathcal{B}^{\ell}}, \frac{q}{2\mathcal{B}^{\ell}}\right)$ *and let define* $\bar{B}_{\mathsf{in}} = B_{\mathsf{in}} - \widetilde{B}_{\mathsf{in}}$ *and* $\bar{b}_{\mathsf{in},j} = a_{\mathsf{in},j} - \widetilde{b}_{\mathsf{in},j}$, $|\bar{b}_{\mathsf{in},j}| \in \left[\frac{-q}{2\mathcal{B}^{\ell}}, \frac{q}{2\mathcal{B}^{\ell}}\right)$ *So we have that their expectations and variances are respectively* $\mathbb{E}(\bar{a}_{\mathsf{in},i,j}) = -\frac{1}{2}$, $\mathsf{Var}(\bar{a}_{\mathsf{in},i,j}) = \frac{q^2}{12\mathcal{B}^{2\ell}} - \frac{1}{12} = \frac{q^2 - \mathcal{B}^{2\ell}}{12\mathcal{B}^{2\ell}}$, $\mathbb{E}(\widetilde{a}_{\mathsf{in},i,j}) = -\frac{1}{2}$ *and* $\mathsf{Var}(\widetilde{a}_{\mathsf{in},i}) = \frac{\mathcal{B}^2-1}{12}$ *and we obtain the same values for* $\bar{b}_{\mathsf{in},j}$ *and* $\widetilde{b}_{\mathsf{in},j}$.

Let us now decrypt $\mathsf{CT}_{\mathsf{out}}$ *using Definition 12.*

$$
\begin{aligned}
\widetilde{M}_{\mathsf{out}} + E_{\mathsf{out}} &= B_{\mathsf{out}} - \sum_{i=0}^{k-1} A_{\mathsf{out}} \boldsymbol{S} = \langle (A_{\mathsf{out}}, B_{\mathsf{out}}), (-\boldsymbol{S}, 1) \rangle \\
&= \left\langle \mathbf{Dec}^{(\mathcal{B},\ell)}(B_{\mathsf{in}}) \cdot \overline{\mathsf{CT}}_k - \sum_{i=0}^{k-1} \mathbf{Dec}^{(\mathcal{B},\ell)}(A_{\mathsf{in},i}) \cdot \overline{\mathsf{CT}}_{k-1}, (-\boldsymbol{S}, 1) \right\rangle \\
&= \sum_{j=0}^{\ell-1} \widetilde{B}_{\mathsf{in}}^j \langle \mathsf{CT}_{k,j}, (-\boldsymbol{S}, 1) \rangle - \sum_{i=0}^{k-1} \sum_{j=0}^{\ell-1} \widetilde{A}_{\mathsf{in},i}^j \langle \mathsf{CT}_{i,j}, (-\boldsymbol{S}, 1) \rangle \\
&= \sum_{j=0}^{\ell-1} \widetilde{B}_{\mathsf{in}}^j \left(\frac{q}{\mathcal{B}^{j+1}} p + E_{\mathrm{GGSW},k}^j \right) - \sum_{i=0}^{k-1} \sum_{j=0}^{\ell-1} \widetilde{A}_{\mathsf{in},i}^j \left(\frac{q}{\mathcal{B}^{j+1}} p \cdot S_i + E_{\mathrm{GGSW},i}^j \right) \\
&= (B_{\mathsf{in}} - \bar{B}_{\mathsf{in}}) \cdot p + \sum_{j=0}^{\ell-1} \widetilde{B}_{\mathsf{in}}^j \cdot E_{\mathrm{GGSW},k}^j - \sum_{i=0}^{k-1} \left(A_{\mathsf{in},i} - \bar{A}_{\mathsf{in},i} \right) \cdot p \cdot S_i - \sum_{i=0}^{k-1} \sum_{j=0}^{\ell-1} \widetilde{A}_{\mathsf{in},i}^j E_{\mathrm{GGSW},i}^j \\
&= \left(B_{\mathsf{in}} - \bar{B}_{\mathsf{in}} - \sum_{i=0}^{k-1} (A_{\mathsf{in},i} - \bar{A}_{\mathsf{in},i}) \cdot S_i \right) \cdot p + \sum_{j=0}^{\ell-1} \widetilde{B}_{\mathsf{in}}^j \cdot E_{\mathrm{GGSW},k}^j - \sum_{i=0}^{k-1} \sum_{j=0}^{\ell-1} \widetilde{A}_{\mathsf{in},i}^j E_{\mathrm{GGSW},i}^j \\
&= \widetilde{M}_{\mathsf{in}} \cdot p + E_{\mathsf{in}} \cdot p + p \left(-\bar{B}_{\mathsf{in}} + \sum_{i=0}^{k-1} \bar{A}_{\mathsf{in},i} \cdot S_i \right) + \sum_{j=0}^{\ell-1} \widetilde{B}_{\mathsf{in}}^j \cdot E_{\mathrm{GGSW},k}^j - \sum_{i=0}^{k-1} \sum_{j=0}^{\ell-1} \widetilde{A}_{\mathsf{in},i}^j E_{\mathrm{GGSW},i}^j.
\end{aligned}
$$

Now, let us isolate and focus on the error associated with the constant coefficient in the previous

equation (we note that the approximated result will be identical for all other coefficients):

$$e_{0,\mathsf{out}} = e_{0,\mathsf{in}} \cdot p + \underbrace{\left(-\bar{b}_{0,\mathsf{in}} + \sum_{i=0}^{k-1}\left(\bar{a}_{\mathsf{in},i,0} \cdot s_{i,0} - \sum_{j=1}^{N-1} \bar{a}_{\mathsf{in},i,j} \cdot s_{i,j}\right)\right) \cdot p}_{I}$$
$$\underbrace{+\sum_{\iota=0}^{\ell-1}\left(\widetilde{b}_{\mathsf{in},0} \cdot e^{\iota}_{\mathrm{GGSW},k,0} - \sum_{j=1}^{N-1} \widetilde{b}_{\mathsf{in},j} \cdot e^{\iota}_{\mathrm{GGSW},k,N-1-j}\right)}_{II}$$
$$\underbrace{-\sum_{i=0}^{k-1}\sum_{\iota=0}^{\ell-1}\left(\widetilde{a}^{\iota}_{\mathsf{in},i,0} \cdot e^{\iota}_{\mathrm{GGSW},i,0} - \sum_{j=1}^{N-1} \widetilde{a}^{\iota}_{\mathsf{in},i,j} \cdot e^{\iota}_{\mathrm{GGSW},i,N-1-j}\right)}_{II}.$$

Then, $\mathsf{Var}(I)$ *can be approximated as follows:*

$$\begin{aligned}\mathsf{Var}(I) &= \left(\mathsf{Var}(p) + \mathbb{E}^2(p)\right)\mathsf{Var}\left(-\bar{b}_{0,\mathsf{in}} + kN\bar{a}_{\mathsf{in},i,j} \cdot s_{i,j}\right) + \mathsf{Var}(p) \cdot \mathbb{E}^2\left(-\bar{b}_{0,\mathsf{in}} + kN\bar{a}_{\mathsf{in},i,j} \cdot s_{i,j}\right)\\ &= \frac{1}{2} \cdot \mathsf{Var}\left(-\bar{b}_{0,\mathsf{in}} + kN\bar{a}_{\mathsf{in},i,j} \cdot s_{i,j}\right) + \frac{1}{4} \cdot \mathbb{E}^2\left(-\bar{b}_{0,\mathsf{in}} + kN\bar{a}_{\mathsf{in},i,j} \cdot s_{i,j}\right).\end{aligned}$$

With:

$$\begin{aligned}\mathbb{E}\left(-\bar{b}_{0,\mathsf{in}} + kN\bar{a}_{\mathsf{in},i,j} \cdot s_{i,j}\right) &= \mathbb{E}(-\bar{b}_{0,\mathsf{in}}) + kN\mathbb{E}(\bar{a}_{\mathsf{in},i,j}) \cdot \mathbb{E}(s_{i,j})\\ &= \frac{1}{2} - kN\frac{1}{2}\mathbb{E}(s_{i,j}) = \frac{1}{2}(1 - kN\mathbb{E}(s_{i,j}))\\ \mathbb{E}^2\left(-\bar{b}_{0,\mathsf{in}} + kN\bar{a}_{\mathsf{in},i,j} \cdot s_{i,j}\right) &= \frac{1}{4}(1 - kN\mathbb{E}(s_{i,j}))^2.\end{aligned}$$

And:

$$\begin{aligned}\mathsf{Var}(-\bar{b}_{0,\mathsf{in}} + kN\bar{a}_{\mathsf{in},i,j} \cdot s_{i,j}) &= \mathsf{Var}(-\bar{b}_{0,\mathsf{in}}) + kN\mathsf{Var}(\bar{a}_{\mathsf{in},i,j} \cdot s_{i,j})\\ &= \frac{q^2 - \mathfrak{B}^{2\ell}}{12\mathfrak{B}^{2\ell}} + kN\left(\mathsf{Var}(\bar{a}_{\mathsf{in},i,j}) \cdot \left(\mathsf{Var}(s_{i,j}) + \mathbb{E}^2(s_{i,j})\right) + \mathsf{Var}(s_{i,j}) \cdot \mathbb{E}^2(\bar{a}_{\mathsf{in},i,j})\right)\\ &= \frac{q^2 - \mathfrak{B}^{2\ell}}{12\mathfrak{B}^{2\ell}} + kN\left(\frac{q^2 - \mathfrak{B}^{2\ell}}{12\mathfrak{B}^{2\ell}} \cdot \left(\mathsf{Var}(s_{i,j}) + \mathbb{E}^2(s_{i,j})\right) + \frac{1}{4}\mathsf{Var}(s_{i,j})\right)\\ &= \frac{q^2 - \mathfrak{B}^{2\ell}}{12\mathfrak{B}^{2\ell}} \cdot \left(1 + kN\left(\mathsf{Var}(s_{i,j}) + \mathbb{E}^2(s_{i,j})\right)\right) + \frac{kN}{4}\mathsf{Var}(s_{i,j}).\end{aligned}$$

Then, $\mathsf{Var}(II)$ *can be approximated as follows:*

$$\begin{aligned}&\mathsf{Var}(II) = \ell \cdot N \cdot \mathsf{Var}(\widetilde{b}_{0,\mathsf{in}} \cdot e^{\iota}_{\mathrm{GGSW},k,0}) + k\ell N\mathsf{Var}(\widetilde{a}^{\iota}_{\mathsf{in},i,0} \cdot e^{\iota}_{\mathrm{GGSW},i,0})\\ &= (k+1) \cdot \ell \cdot N \cdot \mathsf{Var}(\widetilde{a}^{\iota}_{\mathsf{in},i,0} \cdot e^{\iota}_{\mathrm{GGSW},i,0})\\ &= (k+1)\ell N\left(\mathsf{Var}(\widetilde{a}^{\iota}_{\mathsf{in},i,0})\mathsf{Var}(e_{\mathrm{GGSW},i,0}) + \mathbb{E}^2(\widetilde{a}^{\iota}_{\mathsf{in},i,0})\mathsf{Var}(e_{\mathrm{GGSW},i,0}) + \mathsf{Var}(\widetilde{a}^{\iota}_{\mathsf{in},i,0})\mathbb{E}^2(e_{\mathrm{GGSW},i,0})\right)\\ &= (k+1) \cdot \ell \cdot N\left(\frac{\mathfrak{B}^2 - 1}{12}\sigma^2_{\mathrm{GGSW}} + \frac{1}{4}\sigma^2_{\mathrm{GGSW}}\right) = \ell \cdot (k+1) \cdot N \cdot \frac{\mathfrak{B}^2 + 2}{12}\sigma^2_{\mathrm{GGSW}}.\end{aligned}$$

Finally, we obtain:

$$\begin{aligned}\mathsf{Var}(e_{\mathsf{out}}) &= \mathsf{Var}(e_{0,\mathsf{in}} \cdot p) + \mathsf{Var}(I) + \mathsf{Var}(II)\\ &= \ell \cdot (k+1) \cdot N \cdot \frac{\mathfrak{B}^2 + 2}{12}\sigma^2_{\mathrm{GGSW}} + \frac{\sigma^2_{\mathsf{in}}}{2} + \frac{kN}{8}\mathsf{Var}(s_i)\\ &+ \frac{q^2 - \mathfrak{B}^{2\ell}}{24\mathfrak{B}^{2\ell}} \cdot \left(1 + kN \cdot \left(\mathsf{Var}(s_i) + \mathbb{E}^2(s_i)\right)\right) + \frac{1}{16} \cdot (1 - kN \cdot \mathbb{E}(s_i))^2.\end{aligned}$$

□

Cmux. The CMux is a homomorphic operation that takes as input two ciphertexts and an encrypted selector, and depending on the encrypted selector, outputs one or the other ciphertext without leaking any information. It is an homomorphic selector. More precisely, given two GLWE ciphertexts encrypting respectively $M_0 \in \mathfrak{R}_{p,N}$ and $M_1 \in \mathfrak{R}_{p,N}$ and a GGSW ciphertext encrypting a bit $b \in \{0,1\}$, the Cmux returns the ciphertext corresponding to the encrypted input bit b, i.e., it returns M_b (M_0 if $b = 0$, M_1 otherwise). The main building block of the algorithm is the external product (Algorithm 7), and the goal is to homomorphically compute $(M_1 + M_0) \cdot b + M_0$.

Algorithm 8: $\mathsf{CT}_b \leftarrow \mathsf{CMux}(\mathsf{CT}_0, \mathsf{CT}_1, \overline{\overline{\mathsf{CT}}})$

Context:
- $\boldsymbol{S} = (S_0, \ldots, S_{k-1})$: the GLWE secret key
- $\Delta M_{i\in\{0,1\}} \in \mathfrak{R}_{q,N}$: The encoded message of $\mathsf{CT}_{i\in\{0,1\}}$
- $b \in \{0,1\}$: The encoded message of $\overline{\overline{\mathsf{CT}}}$
- $(\mathfrak{B}, \ell) \in (\mathbb{Z}^*)^2$: The (base, level) decomposition.
- $\overline{\mathsf{CT}}_{i\in[0,k-1]} \in \mathrm{GLEV}^{\mathfrak{B},\ell}_{\boldsymbol{S}}(-P \cdot S_i) \subseteq \mathfrak{R}^{(k+1)\ell}_{q,N}$ (Definition 13)
- $\overline{\mathsf{CT}}_k \in \mathrm{GLEV}^{\mathfrak{B},\ell}_{\boldsymbol{S}}(P) \subseteq \mathfrak{R}^{(k+1)\ell}_{q,N}$ (Definition 13)

Input:
- $\mathsf{CT}_0 \in \mathrm{GLWE}_{\boldsymbol{S}}(M_0) \subseteq \mathfrak{R}^{k+1}_{q,N}$
- $\mathsf{CT}_1 \in \mathrm{GLWE}_{\boldsymbol{S}}(M_1) \subseteq \mathfrak{R}^{k+1}_{q,N}$
- $\overline{\overline{\mathsf{CT}}} \in \mathrm{GGSW}^{\mathfrak{B},\ell}_{\boldsymbol{S}}(b) \subseteq \mathfrak{R}^{(k+1)\ell\cdot(k+1)}_{q,N}$ (Definition 14)

Output: $\mathsf{CT}_b \in \mathrm{GLWE}_{\boldsymbol{S}}(M_b) \subseteq \mathfrak{R}^{k+1}_{q,N}$

```
1 CT_b ← (CT_1 − CT_0) ⊡ CT̿ + CT_0;                    /* Algorithm 7 */
2 return CT_b
```

Theorem 2.12 (Cmux (Algorithm 8))**.** *Let $\boldsymbol{S} = (S_0, \ldots, S_{k-1}) \in \mathfrak{R}^k_{q,N}$ be the GLWE secret key. For $j \in \{0,1\}$, let $\mathsf{CT}_j = (A_{j,0}, \ldots, A_{j,k-1}, B_j) \in \mathrm{GLWE}_{\boldsymbol{S}}(M_j) \subseteq \mathfrak{R}^{k+1}_{q,N}$ be a GLWE ciphertext encrypting the message $M_j = \sum_{i=0}^{N-1} m_{j,i} X^i \in \mathfrak{R}_{p,N}$ with the noise sample from the centered Gaussian distribution $\mathcal{N}_{\sigma^2}$. Let $\overline{\overline{\mathsf{CT}}} = (\overline{\mathsf{CT}}_0, \ldots, \overline{\mathsf{CT}}_{k-1}, \overline{\mathsf{CT}}_k) \in \mathrm{GGSW}^{\mathfrak{B},\ell}_{\boldsymbol{S}}(b) \subseteq \mathfrak{R}^{(k+1)\ell\cdot(k+1)}_{q,N}$ be a GGSW ciphertext encrypting the constant message b with the noise sample from the centered Gaussian distribution $\mathcal{N}_{\sigma^2_{\mathrm{GGSW}}}$.*

Then Algorithm 8 returns an GLWE ciphertext $\mathsf{CT}_{\mathsf{out}} = (A_0, \ldots, A_{k-1}, B) \in \mathrm{GLWE}_{\boldsymbol{S}}(M_b) \subseteq \mathfrak{R}^{k+1}_{q,N}$. The cost of this algorithm is:

$$\begin{aligned}\mathsf{Cost}^{\ell,k,N}_{\mathsf{CMux}} &= \ell \cdot (k+1) \cdot N \cdot \mathsf{Cost}^{\ell}_{\mathbf{Dec}} + \ell \cdot (k+1) \cdot \mathsf{Cost}^{N}_{\mathsf{FFT}} \\ &+ (k+1) \cdot \ell \cdot (k+1) \cdot N \cdot \mathsf{Cost}^{N}_{\mathsf{mulFFT}} \\ &+ (k+1) \cdot (\ell \cdot (k+1) - 1) \cdot N \cdot \mathsf{Cost}^{N}_{\mathsf{addFFT}} \\ &+ (k+1) \cdot \mathsf{Cost}^{N}_{\mathsf{iFFT}} + 2\mathsf{Cost}^{k,N}_{add}.\end{aligned}$$

Where $\mathsf{Cost}^N_{\mathsf{FFT}}$ and $\mathsf{Cost}^N_{\mathsf{iFFT}}$ denote the cost of performing a FFT (reps., an inverse FFT) of size N. $\mathsf{Cost}^N_{\mathsf{addFFT}}$, $\mathsf{Cost}^N_{\mathsf{mulFFT}}$ denotes the cost of performing addition and multiplication of size N in the Fourier domain.

The variance of Algorithm 8 is:

$$\mathsf{Var}(\mathsf{Cmux}) = \mathsf{Var}(\mathsf{ExternalProduct}).$$

Proof *(Theorem 2.12). The proof is not provided, but can easily be retrieved from the proof of Theorem 2.11.* □

Blind Rotation. Blind rotation is the second major step of the bootstrapping. This operation consists in chaining multiple CMuxes to perform a rotation of an input polynomial message encrypted under a GLWE ciphertext. Given a GLWE ciphertext encrypting a message M, the blind rotation takes as input a set of rotations r_i, each associated with a GGSW ciphertext encrypting a selector bit b_i. Then by chaining several CMuxes taking as input the output of the previous CMux, the blind rotation outputs a ciphertext encrypting $M \cdot X^{\sum r_i b_i}$.

In the context of the PBS, the input rotations correspond to the elements of $(\boldsymbol{a}, b) \in \mathbb{Z}_q^{n+1}$ of a noisy LWE ciphertext, and the selectors correspond to the associated secret key bits. Consequently, blind rotation outputs an encryption of $M \cdot X^{b-\sum a_i s_i} = M \cdot X^{\Delta m+e}$ (Definition 12). We present the details of the blind rotation in the PBS context in Theorem 2.13 and Algorithm 9.

The polynomial $M \in \mathfrak{R}_{p,N}$, known as the Lookup Table (LUT), is designed to implement the rounding operation of the decoding process (see Definition 8). Each coefficient of the LUT corresponds to a possible decoded message of the input LWE. More generally, to evaluate a function homomorphically, the LUT can be constructed such that the i^{th} coefficient encodes the value $f(i)$, enabling functional evaluation through blind rotation and sample extraction.

Definition 16. *The Lookup Table (LUT) is a polynomial that encodes a function f such that the i^{th} coefficient corresponds to the encoding of $f(i)$.*

$$LUT = \sum_{i=0}^{N-1} \mathsf{encode}(f(i), \pi, p, q) \quad \text{, See Definition 8.}$$

In the commonly used case, when q and p are powers of two and $\pi = 1$ and with $\Delta m + e \in [0, N-1]$, the LUT is constructed as follows:

$$LUT = X^{-\Delta/2} \sum_{i=0}^{N/\Delta-1} \sum_{j=0}^{\Delta} f(i) \Delta X^{i \cdot \Delta + j} \mod X^N + 1.$$

With a such Lookup, after a rotation by $-(\Delta m + e)$, the resulting polynomial $LUT \cdot X^{-(\Delta m+e)}$, have $\Delta f(m)$ as the constant term.

We note that this lookup table can be public and trivially encrypted, as explained in Remark 2.6, or it can be encrypted within a GGSW ciphertext. In the second case, the function f remains hidden from the server.

As presented in Definition 4, this work focuses exclusively on the cyclotomic polynomial X^N+1. For other cyclotomic polynomials, the construction of the lookup table must be adapted accordingly, as detailed in [JW22].

Theorem 2.13 (Blind Rotation (Algorithm 9))**.** *Let $\boldsymbol{s} = (s_0, \ldots, s_{n-1}) \in \mathbb{Z}_q^n$ be a binary LWE secret key and $\boldsymbol{S} \in \mathfrak{R}_{q,N}^k$ be a GLWE secret key. Let $\mathsf{ct_{in}} = (a_{\mathsf{in},0}, \ldots, a_{\mathsf{in},n-1}, b_{\mathsf{in}}) \in \mathrm{LWE}_{\boldsymbol{s}}(m) \subseteq \mathbb{Z}_{2N}^{n+1}$ encrypting the message $m \in \mathbb{Z}_p$. Let n GGSW ciphertexts $\left\{ \overline{\overline{\mathsf{CT}}}_i \in \mathrm{GGSW}_{\boldsymbol{S}}^{\mathcal{R},\ell}(s_i) \subseteq \mathfrak{R}_{q,N}^{(k+1)\ell \times (k+1)} \right\}_{i \in [0,n-1]}$ be a set of GGSW ciphertexts, each encrypting an element s_i of the LWE secret key under the GLWE secret key, with noise drawn from the centered Gaussian distribution $\mathcal{N}_{\sigma^2_{\mathrm{GGSW}}}$. This collection will later corresponds to the bootstrapping key (see Definition 17). Finally let the lookup table LUT_f representing the function f as presented in Definition 16*

Then Algorithm 9 returns an GLWE ciphertext $\mathsf{CT_{out}} \in \mathrm{GLWE}_{\boldsymbol{S}}\left(\mathsf{LUT}_f \cdot X^{-(b+\sum_{i=0}^{n-1} a_i s_i)}\right) \subseteq \mathfrak{R}_{q,N}^{k+1}$. The cost of this algorithm is:

$$\mathsf{Cost}^{\ell,k,N,n}_{\mathsf{BlindRotation}} = n \cdot \mathsf{Cost}^{\ell,k,N}_{\mathsf{CMux}}.$$

Algorithm 9: $\mathsf{CT} \leftarrow \mathsf{BlindRotation}(\mathsf{ct_{in}}, \mathsf{CT}_f, \overline{\overline{\mathsf{CT}}}_{i\in[0,n-1]})$

Context:
$\begin{cases} \boldsymbol{s} = (s_0, \dots, s_{n-1}) : \text{ the LWE secret key} \\ \boldsymbol{S} = (S_0, \dots, S_{k-1}) : \text{ the GLWE secret key} \\ \Delta m \in \mathbb{Z}_{2N} : \text{The encoded message of } \mathsf{ct_{in}} \\ \mathsf{LUT}_f : \text{The Lookup table evaluating the function } m \mapsto f(m), \textit{Definition } 16 \\ \mathsf{CT}_f : \text{Trivial encryption of } \mathsf{LUT}_f \text{ (Remark 2.6)} \\ (\mathfrak{B}, \ell) \in (\mathbb{Z}^*)^2 : \text{The (base, level) decomposition.} \end{cases}$

Input:
$\begin{cases} \mathsf{ct_{in}} = (a_{\mathsf{in},0}, \dots, a_{\mathsf{in},n-1}, b_{\mathsf{in}}) \in \mathrm{LWE}_{\boldsymbol{s}}(m) \subseteq \mathbb{Z}_{2N}^{n+1} \\ \mathsf{CT}_f \in \mathrm{GLWE}_{\boldsymbol{S}}(\mathsf{LUT}_f) \subseteq \mathfrak{R}_{q,N}^{k+1} \\ \left\{\overline{\overline{\mathsf{CT}}}_i \in \mathrm{GGSW}_{\boldsymbol{S}}^{\mathfrak{B},\ell}(s_i) \subseteq \mathfrak{R}_{q,N}^{(k+1)\ell\times(k+1)}\right\}_{i\in[0,n-1]} \end{cases}$

Output: $\left\{\mathsf{CT_{out}} \in \mathrm{GLWE}_{\boldsymbol{S}}\left(\mathsf{LUT}_f \cdot X^{-(b-\sum_{i=0}^{n-1} a_i s_i)}\right) \subseteq \mathfrak{R}_{q,N}^{k+1}\right.$

1 $\mathsf{CT_{out}} \leftarrow \mathsf{CT}_f \cdot X^{-b}$
2 **for** $i \in [0, n-1]$ **do**
3 $\quad \mathsf{CT}_0 \leftarrow \mathsf{CT_{out}}$
4 $\quad \mathsf{CT}_1 \leftarrow \mathsf{CT_{out}} \cdot X^{a_i}$
5 $\quad \mathsf{CT_{out}} \leftarrow \mathsf{CMux}\left(\mathsf{CT}_0, \mathsf{CT}_1, \overline{\overline{\mathsf{CT}}}_i\right)$; `/* Algorithm 8 */`
6 **return** $\mathsf{CT_{out}}$

And for a trivially encrypted lookup table (Definition 16), the output noise variance is equal to:

$$\mathsf{Var}(\mathsf{BlindRotation}) = n \cdot \ell \cdot (k+1) \cdot N \cdot \frac{\mathfrak{B}^2+2}{12}\sigma^2_{\mathrm{GGSW}} + \frac{nkN}{8}\mathsf{Var}(s_i) + n \cdot \frac{q^2 - \mathfrak{B}^{2\ell}}{24\mathfrak{B}^{2\ell}} \cdot \left(1 + kN \cdot \left(\mathsf{Var}(s_i) + \mathbb{E}^2(s_i)\right)\right) + \frac{n}{16} \cdot (1 - kN \cdot \mathbb{E}(s_i))^2.$$

Proof *(Theorem 2.13). This proof follows the idea of Theorem 2.12; in particular, it consists in chaining n CMux operations.* □

Remark 2.14. For efficiency reasons, the blind rotation operates over polynomials in $\mathfrak{R}_{q,N}$ where N is much smaller than q. Although rotations could theoretically be done with polynomials of degree q (i.e., in $\mathfrak{R}_{q,q}$), in practice q equals 32− or 64−bits. This would result in large polynomials and extremely inefficient polynomial multiplications. Using smaller degree polynomials significantly improves the efficiency of the blind rotation algorithm.

As a result, polynomials are in $\mathfrak{R}_{q,N}$ and rotations are performed modulo $2N$, but the polynomial degree constraint the representation to only N distinct values. This constraint arises from the negacyclicity property, as explained in Remark 2.1.

To solves this limitation, two strategies can be employed:

- Evaluate a negacyclic function, satisfying $f(x) = -f(x+N) \mod 2N$ then the encoded message plus the error $\Delta m + e \in \mathbb{Z}_{2N}$ can be in the full range $[0, 2N-1]$.
- Restrict the encoded message plus the error $\Delta m + e \in \mathbb{Z}_{2N}$ to the range $[0, N-1]$.

The second approach is practically ensured by keeping the most significant bit (MSB) of the plaintext at a known value (generally equal to zero), i.e., we force the padding bit to be equal to zero. To maintain this guarantee, it is crucial to track the message growth throughout the computation to ensure that the padding bit remains unused.

Sample Extract. The final step of the PBS is sample extraction, an operation that permits obtaining an LWE ciphertext encrypting the constant coefficient from a GLWE ciphertext. The

goal is to extract the constant term of B and each coefficient of the polynomials in $\boldsymbol{A}$ to obtain, by rearranging these coefficients, an LWE ciphertext encrypted under the flattened representation of the GLWE secret key (Definition 11).

Algorithm 10: $\mathsf{ct} \leftarrow \mathsf{SampleExtract}(\mathsf{CT})$

Context:
$$\begin{cases} \boldsymbol{S} = (S_0, \ldots, S_{k-1}) \in \mathfrak{R}_{q,N}^k : \text{ the GLWE secret key} \\ \boldsymbol{s} = (s_0, \ldots, s_{kN-1}) \in \mathbb{Z}_q^{Nk} : \text{ Flattened GLWE secret key; } \texttt{/* Definition 11 */} \\ M = \sum_{i=0}^{N-1} m_i X^i \in \mathfrak{R}_{p,N} : \text{The message of } \overline{\overline{\mathsf{CT}}},\ b \in \{0,1\} \end{cases}$$

Input: $\left\{ \mathsf{CT} = (A_0, \ldots A_{k-1}, B) \in \mathrm{GLWE}_{\boldsymbol{S}}(M) \subseteq \mathfrak{R}_{q,N}^{k+1} \right.$

Output: $\left\{ \mathsf{ct} \in \mathrm{LWE}_{\boldsymbol{s}}(m_0) \subseteq \mathbb{Z}_q^{kN+1} \right.$

1 $b \leftarrow b_0$
2 **for** $i \in [0, k-1]$ **do**
3 $a_{iN} \leftarrow a_{i,0}$ **for** j *in* $[1, N-1]$ **do**
4 $a_{iN+j} \leftarrow -a_{i,N-j}$
5 **return** $\mathsf{ct}_{\boldsymbol{s}} = (a_0, \ldots a_{nk-1}, b)$

Theorem 2.14 (Sample Extract (Algorithm 10))**.** *Let* $\mathsf{CT} = (A_0, \ldots, A_{k-1}, B) \in \mathrm{GLWE}_{\boldsymbol{S}}(M) \subseteq \mathfrak{R}_{q,N}^{k+1}$ *be a GLWE ciphertext encrypting the message* $M = \sum_{i=0}^{N-1} m_i X^i \in \mathfrak{R}_{p,N}$ *with the noise sample from the centered Gaussian distribution* $\mathcal{N}_{\sigma^2}$*. After Algorithm 10, we obtain an LWE ciphertext* $\mathsf{ct} \in \mathrm{LWE}_{\boldsymbol{s}}(m_0)$ *encrypted under the secret* $\boldsymbol{s}$*, the flatten representation of* $\boldsymbol{S}$ *(Definition 11). The noise after the sample extract remains unchanged and the cost of Algorithm 10 is:*

$$\mathsf{Cost}_{\mathsf{SampleExtract}}^{k,N} = (k+1)N\mathbb{C}_{\mathsf{Copy}}.$$

where $\mathbb{C}_{\mathsf{Copy}}$ *denotes the computational cost of performing a copy operation on an element in* $\mathbb{Z}_q$*. This operation is almost negligible compared to the complexity of other algorithms.*

Proof *(Theorem 2.14). Let* $\boldsymbol{S} = (S_0, \ldots, S_{k-1}) \in \mathfrak{R}_{q,N}^k$ *be the GLWE secret key. Let* $\mathsf{CT} = (A_0, \ldots, A_{k-1}, B) \in \mathrm{GLWE}_{\boldsymbol{S}}(M) \subseteq \mathfrak{R}_{q,N}^{k+1}$ *be a GLWE ciphertext encrypting the message* $M = \sum_{i=0}^{N-1} m_i X^i \in \mathfrak{R}_{p,N}$ *with the noise sample from the centered Gaussian distribution* $\mathcal{N}_{\sigma^2}$ *such that* $A_j = \sum_{i=0}^{N-1} a_{j,i} X^i$ *for* $j \in [0, k-1]$ *and* $B = \sum_{i=0}^{N-1} b_i X^i = \sum_{j=0}^{k-1} A_j S_j + \widetilde{M} + E$*.*

According to Definition 14, we have that $b_0 = \widetilde{m}_0 + e_0 + \sum_{j=0}^{k-1} \left(a_{j,0} s_{j,0} - \sum_{i=1}^{N-1} a_{j,N-i} s_{j,i} \right)$ *which corresponds to an LWE ciphertext* $(a_{0,0}, -a_{0,N-1}, \ldots - a_{0,1}, a_{1,0}, \ldots, -a_{k-1,1}, b_0)$ *encrypting* m_0 *under the flatten representation of the secret key* $\boldsymbol{S}$ *(Definition 11).* □

The final step of the PBS consists of performing a sample extraction of the constant term from a polynomial message encrypted in a GGSW ciphertext, as described in Algorithm 10. It is also possible to extract coefficients other than the constant one. This can be achieved by properly reordering the coefficients, as described later in Algorithm 29 for another context, or by applying a rotation using Algorithm 2.4 before the sample extraction.

2.4.2 Programmable Bootstrapping

As presented in the section introduction, the TFHE bootstrapping introduced in [CGGI16a, CGGI17, CGGI20] is an operation that reduces the noise of a noisy LWE ciphertext while enabling the evaluation of any univariate function. Previously, we have described all the building blocks required to perform this operation. At a high level, the Programmable Bootstrapping (PBS) first changes the modulus of the input LWE ciphertext to satisfy the conditions required to correctly perform a rotation modulo $2N$. Then, a blind rotation is applied to a lookup table indexed by the input LWE. Finally, the first coefficient of the rotated GLWE ciphertext is extracted, to obtain the desired result in an LWE ciphertext. We now present how all these blocks interact to form the complete programmable bootstrapping algorithm.

Definition 17 (Bootstrapping key)**.** *Let $\boldsymbol{s} = (s_0, \ldots, s_{n-1})$ be an binary LWE secret key and $\boldsymbol{S'}$ be a GLWE secret key. The bootstrapping Key (BSK) is the public material required to perform the programmable bootstrapping. This secret key is composed of n* GGSW *ciphertexts, each encrypting an element s_i of the LWE secret key under the GLWE secret key, with noise drawn from the centered Gaussian distribution $\mathcal{N}_{\sigma^2_{\mathsf{BSK}}}$.*

$$\mathsf{BSK} = (\mathsf{BSK}_0, \ldots, \mathsf{BSK}_{n-1}) = \left\{\mathsf{BSK}_i \in \mathrm{GGSW}^{\mathcal{B},\ell}_{\boldsymbol{S'}}(s_i)\right\}_{i\in[0,n-1]}.$$

Remark 2.15 (Public Material)**.** To refer to all the public materials collectively, we introduce the notation PUB, which denotes the set of all public materials required throughout this work. This set includes: the Bootstrapping Key (Definition 17), the Private Function Key Switching Key (Remark 2.12), the Packing Key Switching Key (Remark 2.13) and the Key Switching Key (Definition 15).

Theorem 2.15 (Programmable Bootstrapping (Algorithm 11))**.** *Let $\boldsymbol{s} = (s_0, \ldots, s_{n-1})$ be a binary LWE secret key and $\boldsymbol{S'} \in \mathfrak{R}^k_{q,N}$ be a GLWE secret key and $\boldsymbol{s'}$ his flatten representation. Let $\mathsf{ct}_{\mathsf{in}} = (a_{\mathsf{in},0}, \ldots, a_{\mathsf{in},n-1}, b_{\mathsf{in}}) \in \mathrm{LWE}_{\boldsymbol{s}}(m) \subseteq \mathbb{Z}^{n+1}_{2N}$ encrypting the message $m \in Z_p$. Let BSK be the bootstrapping key as presented in Definition 17. Finally let the lookup table LUT_f representing the function f as presented in Definition 16.*

Then Algorithm 11 returns an LWE ciphertext $\mathsf{ct}_{\mathsf{out}} \in \mathrm{LWE}_{\boldsymbol{s'}}\left(f\left(\frac{(\Delta m+e)2N}{q} \mod 2N\right)\right) \subseteq \mathfrak{R}^{k+1}_{q,N}$. An LWE ciphertext encrypting $f\left(\frac{(\Delta m+e)2N}{q} \mod 2N\right)$ under the secret key $\boldsymbol{s'}$.

The cost of this algorithm is:

$$\mathsf{Cost}^{\ell,k,N,n,q}_{\mathbf{PBS}} = \mathsf{Cost}^{n,q,2N}_{\mathsf{ModulusSwitch}} + \mathsf{Cost}^{\ell,k,N,n}_{\mathsf{BlindRotation}} + \mathsf{Cost}^{k,N}_{\mathsf{SampleExtract}}.$$

And for a trivially encrypted lookup table (Definition 16), the output noise variance is equal to:

$$\begin{aligned}\mathsf{Var}(\mathbf{PBS}) = {} & n\cdot\ell\cdot(k+1)\cdot N\cdot\frac{\mathcal{B}^2+2}{12}\sigma^2_{\mathsf{BSK}} + \frac{nkN}{32} \\ & + n\cdot\frac{q^2-\mathcal{B}^{2\ell}}{24\mathcal{B}^{2\ell}}\cdot\left(1+\frac{kN}{2}\right) + \frac{n}{16}\cdot\left(1-\frac{kN}{2}\right)^2.\end{aligned}$$

Proof *(Theorem 2.15). The correctness of the programmable bootstrapping algorithm relies on the correctness of three algorithms: the Modulus Switch (Algorithm 6), the Blind Rotation (Algorithm 9), and the Sample Extraction (Algorithm 10). The output variance is determined by the variance introduced during the blind rotation, assuming a binary secret key. The corresponding formula for the output variance can be obtained by directly applying the proof of Theorem 2.13.* □

Remark 2.16. The main difference between TFHE bootstrapping [GINX16, CGGI16a] and FHEW bootstrapping [ASP14, DM15] lies in the way the product between ciphertexts is performed. In TFHE, this operation uses the external product, which computes the product of a GLWE ciphertext and a GGSW ciphertext (see Definition 2.11).

In contrast, in FHEW bootstrapping, the product is performed using an internal product, which is a product between two GGSW ciphertexts, outputting another GGSW ciphertext (see Definition 14). While the noise propagation is similar to that of the external product, the internal product requires several external products to be performed, resulting in a higher cost compared to using a single external product. Moreover, the internal product requires larger public material.

Since the binary distribution is the most commonly used in TFHE, Algorithm 11 is presented with binary secret keys. In Subsection 2.1.2, we introduce several alternative secret key distributions and the PBS can be performed with these alternative distributions, but it results in less efficient computation. For instance, when the LWE secret key follows a ternary distribution, blind rotation requires twice as many Cmux operations, as two chained Cmuxes are sufficient to represent one elements of the LWE ternary secret key.

Algorithm 11: $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathbf{PBS}(\mathsf{ct}_{\mathsf{in}}, \mathsf{LUT}, \mathsf{BSK})$

Context:
- $\boldsymbol{s} = (s_0, \ldots, s_{n-1})$: the LWE secret key
- $\boldsymbol{S'} = (S'_0, \ldots, S'_{k-1})$: the GLWE secret key
- $\boldsymbol{s'}$ the flatten representation of the GLWE secret key, Definition 11
- $\Delta m + e \in \mathbb{Z}_q$: The encoded message of $\mathsf{ct}_{\mathsf{in}}$, Definition 8
- LUT_f : The Lookup table evaluating the function $m \mapsto f(m), Definition\ 16$
- CT_f : Trivial encryption of LUT_f, Remark 2.6
- $(\mathfrak{B}, \ell) \in (\mathbb{Z}^*)^2$: The (base, level) decomposition.

Input:
- $\mathsf{ct}_{\mathsf{in}} = (a_{\mathsf{in},0}, \ldots, a_{\mathsf{in},n-1}, b_{\mathsf{in}}) \in \mathrm{LWE}_{\boldsymbol{s}}(m) \subseteq \mathbb{Z}_q^{n+1}$
- $\mathsf{CT}_f \in \mathrm{GLWE}_{\boldsymbol{S}}(\mathsf{LUT}_f) \subseteq \mathfrak{R}_{q,N}^{k+1}$
- $\mathsf{BSK} = \left\{\mathsf{BSK}_i \in \mathrm{GGSW}_{\boldsymbol{S'}}^{\mathfrak{B},\ell}(s_i)\right\}_{i \in [0,n-1]}$

Output: $\left\{\mathsf{ct}_{\mathsf{out}} \in \mathrm{LWE}_{\boldsymbol{s'}}\left(f\left(\Delta m + e\right)\right) \subseteq \mathbb{Z}_q^{Nk+1}\right.$

```
1 ct ← ModulusSwitch(ct_in, 2N);                    /* Algorithm 6 */
2 CT ← BlindRotation(ct_in, CT_f, BSK);             /* Algorithm 9 */
3 ct_out ← SampleExtract(CT);                       /* Algorithm 10 */
4 return ct_out
```

2.4.3 Carry and Message Space Encoding

In our work [BBB+23], we introduced an efficient method that uses an encoding scheme split into two parts: the message and the carry. This encoding enables several levels of computation to be performed before requiring a bootstrapping. This is another approach from the usual Boolean encoding from the original TFHE scheme.

This encoding is obtained by modifying the Definition 8 in order to include a *carry space* into the plaintext space. The core idea is to give enough room in a ciphertext encrypting an integer message modulo $\beta \in \mathbb{N}$ to store more than just the message but also potential carries coming from leveled operations such as addition or multiplication with a known integer.

In practice, we split the traditional plaintext space into three different parts: the *message subspace* storing an integer modulo $\beta \in \mathbb{Z}$ (we call β the base), the *carry subspace* containing information overlapping β, and a bit of padding (or more) often needed for bootstrapping. In this context, we refer to the *carry-message modulo* as the subspace including both the message subspace plus the carry subspace, and we note it $p \in \mathbb{N}$. Figure 2.2 shows a visual example.

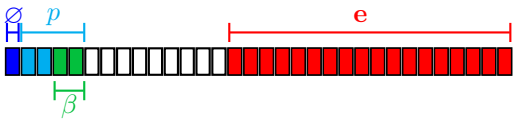

Figure 2.2: Plaintext binary representation with a base $\beta = 4 = 2^2$ (green), a carry subspace (cyan), a carry-message modulo $p = 16 = 2^{2+2}$ (cyan+green) such that $0 < \beta < p$, the error e (red), and a bit of padding is displayed in the MSB (dark blue). The white part is empty. So the plaintext modulo is $32 = 2^{2+2+1}$. This means that we have 2 bits in the carry subspace (set to 0 in a fresh ciphertext), that will contain useful data when one computes leveled operations.

In order to keep track of the worst case message in each ciphertext, i.e., check if there is still room to perform more operations or if we needs to perform a PBS to keep the padding bit empty, we use a metadata that we call *degree of fullness*.

Definition 18. *The* degree of fullness, *that we note* deg, *of an LWE ciphertext* ct *encrypting a message* $0 \leq m < p$, *is equal to* $\deg(\mathsf{ct}) = \frac{\mu}{p-1} \in \mathbb{Q}$, *where* μ *is the known worst case for* m, *i.e.,*

the biggest integer that m can be, such that $0 \leq m \leq \mu < p$. To ensure correctness, the degree of fullness should always be a quantity included between 0 and 1, where $\deg(\mathsf{ct}) = 1$ means that the carry-message subspace is full in the worst case.

We take advantage of the carry subspace to compute leveled operations and to avoid bootstrapping. In practice, *the carry subspace acts as a buffer* to contain the carry information derived from homomorphic operations and the degree of fullness acts as a measure that indicates when the buffer cannot support additional operations: once this limit is reached the carry subspace is emptied by bootstrapping. To be able to perform a leveled operation between two LWE ciphertexts of that type, they need to have the same base β, carry-message p and ciphertext modulus q.

Remark 2.17. Using PBS on values encoded in the message-and-carry representation, it is possible to apply a single PBS to process two fresh ciphertexts as input, thereby enabling the evaluation of bivariate functions (See Algorithm 16). Moreover, information can be selectively extracted from either the message or the carry component. These operations are often referred to as message extraction and carry extraction. These operations are commonly used in more complex algorithms and representations, as detailed in Chapter 7 and Chapter 8.

2.5 Optimization & Parameter Generation

The entire content of this section is derived from our article [BBB+23]. The presented algorithm altogether enables the construction of a fully homomorphic encryption scheme. First, we can perform linear operations such as additions (Algorithm 3), multiplications by a constant (Theorem 2.5), or external products (Algorithm 7). Throughout these operations, the noise grows gradually. Once it reaches a critical threshold, we must manage it using programmable bootstrapping (Algorithm 11) to avoid noise to corrupts the message. After bootstrapping, the secret key may have changed, requiring the use of a key switching (Algorithm 4) to retrieve the input secret key. This process represents the first complete graph presented in the original TFHE scheme.

Along this journey, we distinguish many different parameters, such as, the LWE and GLWE parameters such as the dimension n, the polynomial size N and the GLWE dimension k. These parameters are named the *macro parameter*. In addition to perform operations such as the key switch (Algorithm 5) or the programmable bootstrapping (Algorithm 11), introduces with new parameter such as the bases $\mathcal{B}_{\mathbf{PBS}}$ and $\mathcal{B}_{\mathbf{KS}}$, the different decomposition levels, $\ell_{\mathbf{PBS}}$ and $\ell_{\mathbf{KS}}$. These parameters are named the *micro parameter*. All the micro and macro parameters have a huge impact on the correctness and the execution time of the different algorithms. Moreover during a graph evaluation, it is crucial to track the noise growth to prevent decryption failures ($\mathsf{p}_{\mathsf{fail}}$, Definition 2) while maintaining security and maximizing efficiency.

Finding parameters that satisfy all the correctness constraints across the full computational graph is highly challenging. In this section, we present methodologies that have been proposed to determine optimal parameter sets balancing both security and execution efficiency. This methodology allows us to find parameter sets that satisfy the following three guarantees:

1. the desired *level of security*,
2. the *correctness* of the computation up to the desired correctness probability,
3. a *cost* as small as possible.

The first guarantee is easy to reach using the security oracle (already discussed in Subsection 2.1.4) that can be built using the lattice-estimator [APS15]. Indeed, one can always increase the amount of noise at encryption (or key generation) to get the desired security. Using this, one does not need to find the best encryption noise, one can simply look for the best LWE dimension (or GLWE dimension and polynomial size) and take the minimal encryption noise given by the security oracle. In the end, one is sure to provide enough security as the noise is chosen with respect to other ciphertext parameters.

To guarantee the correctness of a computation (guarantee 2), one needs to rely on the noise model of each FHE operator in the graph. With FHE schemes, there is a link between the noise inside a ciphertext and the correctness of the computation. In fact, if the noise grows too much, the message will be tampered and the decryption algorithm will not yield the correct result. In order to guarantee the correctness, one needs to track the noise at each step of the computation (using the noise model) and choose parameters in a way that the noise remains small enough.

The last guarantee is to have a cost as small as possible. For that, one needs to use the cost model and select the parameters that minimize this cost (among the ones that satisfy guarantee 2). Naturally, the more realistic the cost model is, the better the parameters will be in practice.

2.5.1 Basis for FHE Optimization

Let us start by recalling some high-level definitions.

Definition 19 (FHE & Plain operator)**.** *Any FHE operator $\mathbb{O}$ is an implementation of an FHE algorithm, on a given piece of hardware, taking as input some ciphertexts and/or plaintexts and returning one or more ciphertexts. A plain operator is a function mapping several integers into an output list of integers.*

Noise formulae and cost model. A noise formula for a given homomorphic operator takes as input the variance of the input ciphertext noises, some cryptographic parameters involved in the operator computation, as well as the plaintext values used in the operator.

The *noise of a freshly encrypted ciphertext* is a random (small) integer drawn from a given distribution $\chi(\sigma)$, where σ^2 is its variance. Variances help us quantifying noise in ciphertext, so whenever it is written that a ciphertext contains more noise than another, we mean that the noise inside the first ciphertext is drawn from a normal distribution with a larger variance than the second one.

We will always consider the *cost model* to approximate the running time on a single thread. More details on the cost model used in the experiments / benchmarks are provided later in the manuscript. Other and more complex cost models could be considered (e.g., combining complexity of operations with keys and ciphertext sizes, pieces of hardware, RAM, etc.), but it is note studied in this manuscript.

Security. The *security* of a GLWE-based scheme depends on the distribution of the secret key (for example binary, ternary or Gaussian), the product between the GLWE dimension and the polynomial size (i.e. $k \cdot N$), the noise distribution, and the ciphertext modulus (often written q). To estimate the security level offered by some given parameters one can use the LWE/Lattice-estimator [APS15] (See subsection 2.1.4). As a general rule of thumb, to keep the same security level, when increasing the product $k \cdot N$ we can decrease the minimal noise needed inside a ciphertext.

Noise Plateau. In TFHE's implementations, q is often chosen equal to 2^{32} or 2^{64}, in order to be able to work with 32 or 64 bit integers, respectively, since they are native types in the majority of machines used nowadays. As mentioned before, to keep the same security level when increasing $k \cdot N$, we can reduce the variance σ^2 of the noise. But the value of q and the fact that we work with discretized values, impose a lower bound on the variance, meaning that starting from a certain point − that we call *plateau* − we can not reduce the variance anymore, otherwise we lose security. Notice that, for LWE ciphertexts, a small increase of the value of n allows for a small decrease of the value of σ^2. But when working with polynomials, moving to a bigger power of 2 for N will lead to a large increase of the size of the secret key, from kN to $k \cdot 2N$, and so a large decrease of σ^2 when allowed. For the same security reason mentioned above, at some point we reach a limit where we cannot reduce the variance σ^2 of the noise anymore. The consequence is that to avoid having no security at all, we end up with a security level way higher than desired.

Definition 20 (Noise Plateau)**.** *The noise plateau is the threshold in the size of a ciphertext beyond which noise can no longer be reduced without compromising security. This minimum noise variance is studied in [GHS12, MR09].*

Remark 2.18. We note that the notion of a Noise Plateau can be subject to controversy, as even with very large coefficients and a small σ^2 value, errors may still be present, even with discretized values. However, since this topic is still under discussion, we chose not to decrease the noise beyond the noise plateau threshold during the parameters selection.

In the rest of this manuscript, we assume that, for each possible distributions of the secret key, we have access to the following security oracle:

Definition 21 (Security Oracle)**.** *Given the product $k \cdot N$, a level of security λ and a ciphertext modulus q, the security oracle outputs the minimal noise variance σ^2_{min} needed in a ciphertext for it to be secure with the required level of security.*

2.5.2 The TFHE Optimization Problem

As said above, one needs to choose the *macro-parameters* among a set of possible values. For example, the polynomial size N must be a power of 2. One wants to narrow it down to a finite set, and a practical yet wide enough space for TFHE-like schemes could be $\mathcal{P}_N = \left\{2^8, 2^9, \cdots, 2^{17}\right\}$. In the same manner, the LWE dimension n could be selected in $\mathcal{P}_n = [256, 2048]$ and the GLWE dimension in $\mathcal{P}_k = [1, 6]$. The $\mathcal{P}_N$ is called the *search space* of N.

Definition 22 (FHE Directed Acyclic Graph (FHE DAG))**.** *Let $\mathcal{G} = (V, L)$ be a DAG of FHE operators. We define $V = \{\mathbb{O}_i\}_{1 \leq i \leq \alpha}$ as the set of vertices, each of them being an FHE operator. We define L as the set of edges, each of them associated with the modulus p of the encrypted message i.e. $L \subset \left\{\{x, y, p\} \,|(x, y) \in V^2, p \in \mathbb{N}\right\}$. When L is not needed, we will simply write $\mathcal{G} = V$. We note $\mathsf{Cost}\mathcal{G}, x$ the cost of running the FHE graph $\mathcal{G}$ with the parameter set x.*

For a given FHE DAG $\mathcal{G}$ (definition 22), one also needs to set the *micro parameters*. For example, the logarithm of the decomposition base for a **KS** or a **PBS** $\log_2(\mathcal{B})$ can be taken in $\mathcal{P}_{\log_2(\mathcal{B})} = [1, \lfloor \log_2(q) \rfloor]$ and the level of the decomposition ℓ in $\mathcal{P}_\ell = [1, \lfloor \log_2(q) \rfloor]$. As $(\mathcal{B}, \ell)$ are used to do a radix decomposition of each integer composing the input ciphertext, we know that $\ell \cdot \log_2(\mathcal{B}) \leq \log_2(q)$ so in practice, we will consider $(\mathcal{B}, \ell)$ as one unique variable in $\mathcal{P}_{\log_2(\mathcal{B}),\ell} = \left\{(\log_2(\mathcal{B}), \ell) \in [1, \lfloor \log_2(q) \rfloor]^2,\ \ell \cdot \log_2(\mathcal{B}) \leq \log_2(q)\right\}$.

In the end, one needs to choose a set of parameters in the Cartesian product of the search spaces of all the micro and macro parameters of a graph $\mathcal{G}$. This space is noted $\mathcal{P}_\mathcal{G}$ and is called the *search space* of $\mathcal{G}$. In the rest of the manuscript, this set is simply called $\mathcal{P}$ when there is no ambiguity on the graph.

Definition 23 (Noise Bound)**.** *Let $\mathsf{CT} \in \mathrm{GLWE}_{\boldsymbol{S}}\left(\widetilde{M}\right)$ a GLWE ciphertext of an encoding $\widetilde{M}$ of M with a message modulus p and π padding bits. The noise bound $t_\alpha(\pi, p)$ for a failure probability α is the biggest integer satisfying:*

$$\sigma \leq t_\alpha(\pi, p) \Rightarrow \mathbb{P}\left(\mathsf{Decode}\left(\widetilde{M}, 2^\pi \cdot p, q\right) \neq M\right) \leq \alpha.$$

The noise bound can also depend on other values, for instance it could take the degree of fullness later defined in Definition 18.

Remark 2.19. Assuming that the input ciphertext contains a noise polynomial $E = \sum_{i=0}^{N-1} e_i X^i \in \mathfrak{R}_{q,N}$ such that $\forall i, e_i \sim \mathcal{N}\left(0, \sigma^2\right)$, we have an explicit formula for the noise bound $t_\alpha(\pi, p) = \frac{\Delta}{2 \cdot \kappa}$ with $\kappa = z^*(p_{\mathsf{fail}})$, the standard score (Definition 2) for $p_{\mathsf{fail}} = 1 - \sqrt[N]{1 - \alpha}$. Let us assume $\sigma \leq t_\alpha$.

Immediately using Definition 2 and Equation 2.1, we have $\mathbb{P}\left(|e_i| \geq \frac{\Delta}{2}\right) \leq p_{\mathsf{fail}} = 1 - \sqrt[N]{1-\alpha}$. Thus

$$\begin{aligned}
\mathbb{P}\left(\mathsf{Decode}\left(\widetilde{M}, 2^{\pi}\cdot p, q\right) \neq M\right) &= \mathbb{P}\left(\bigcup |e_i| \geq \frac{\Delta}{2}\right) \\
&= 1 - \mathbb{P}\left(\bigcap |e_i| < \frac{\Delta}{2}\right) \\
&= 1 - \prod_{i=1}^{N} \mathbb{P}\left(|e_i| < \frac{\Delta}{2}\right) \text{ by indep. of } \{e_i\}_{i\in[\![1,N]\!]} \\
&= 1 - \prod_{i=1}^{N} \left(1 - \mathbb{P}\left(|e_i| \geq \frac{\Delta}{2}\right)\right) \\
&\leq 1 - (1 - p_{\mathsf{fail}})^{\frac{1}{N}} = \alpha.
\end{aligned}$$

When the input ciphertext is an LWE ciphertext i.e. $N = 1$, we have $1 - \sqrt[N]{1-\alpha} = \alpha$.

Using the noise bound, we can guarantee a correct decoding up to a given probability using only the distribution of the noise which can be publicly estimated. The tightness of the noise model is crucial to build tight confidence intervals.

Every ciphertext in an FHE DAG must have a noise smaller than its associated noise bound in order to guarantee the correctness of the computation. With those constraints, we define the *noise feasible set*, a subset of the search space $\mathcal{P}$ where every set of parameters will guarantee a correct computation.

Definition 24 (Noise Feasible Set)**.** *Let $\mathcal{G}$, an FHE DAG such that $\mathcal{G} = (V, L)$ with $L = \{(\cdot,\cdot,p_i)\}_{i\in[1,|L|]}$, and let α be a failure probability. Let $\{\sigma_i\}_{i\in|L|}$ be the standard deviation of the noise in the ciphertexts transiting on every edge of $\mathcal{G}$. For every edge i, we must have $\sigma_i(x) \leq t_\alpha(p_i)$ which defines a subset of the search space $\mathcal{P}$: $\mathcal{S}_i = \{x \in \mathcal{P} | \sigma_i(x) \leq t_\alpha(p_i)\}$. The intersection of all those sets is the noise feasible set $\mathcal{S}$: the set of parameter sets that will lead to a correct computation. We have:*

$$\mathcal{S} = \bigcap_{i\in I} \mathcal{S}_i = \{x \in \mathcal{P} | \forall i \in [1,|L|], \sigma_i \leq t_\alpha(p_i)\}.$$

By choosing a set of parameters that is in the noise feasible set, we are sure to have a correct computation which satisfies guarantee 2. In this set, we want to find the set of parameters minimizing the cost of the FHE DAG. Formally, we want:

$$\arg\min_{x\in\mathcal{P}} \mathsf{Cost}(\mathcal{G}, x) \text{ s.t. } x \in \mathcal{S}(\mathcal{G}). \tag{2.2}$$

The problem of finding efficient and correct FHE parameters is then a minimization problem under constraints. We can naturally use optimization techniques to solve it. The issue is that the complexity of the problem is dependent on the size of the FHE DAG which can rapidly become unrealistic for large DAGs. In the next section, we present several non trivial simplifications prior to the optimization enabling to speed up the task.

Remark 2.20. As we defined a feasible set for the noise, we can also define other feasible sets for other constraints. For instance to limit the size of the public keys (key switching keys, bootstrapping keys, ...), the size of the ciphertexts (bandwidth) or even to add some constraints between parameters.

2.5.3 Pre-Optimization & Graph Transformations

To simplify the optimization problem, we present an analysis that applies to any FHE DAG. The idea is to subdivide it into subgraphs with the constraint that, to compute the noise distribution

of a ciphertext in one of these subgraphs, we do not need to know the noise distribution of a ciphertext in another subgraph.

The starting point is to note that there are some FHE operators that output ciphertexts with a noise independent of the input noise for some well-chosen parameters. This motivates us to distinguish those FHE operators from the rest:

Definition 25 (FHE operator Categories)**.** *We divide the FHE operators (definition 19) into two categories regarding their respective noise formulae:*

(i) an operator which outputs a noise independent of the input noise, such as the **PBS** *in our context;*

(ii) an operator which adds some noise to the input noise, such as a **KS** *or a dot product;*

Using this distinction, for any FHE DAG, we can identify sub-graphs that are independent from others. Now that we have several independent sub-graphs, we want to find a way to compare them together. To do so, we define the notion of *atomic pattern types* to regroup sub-graphs of FHE operators called *atomic patterns* that we know how to compare. For instance, two atomic patterns of the same type can have a different message modulus p or different number of inputs.

For each atomic pattern types, we will compare atomic patterns and identify the ones where the noise will be the highest. Those are the ones we need to take into account when trying to construct $\mathcal{S}(\mathcal{G})$.

Definition 26 (Atomic Pattern Type)**.** *An Atomic Pattern (AP) type $\mathscr{A}^{(\cdot)}$ corresponds to a sub-graph of FHE operators that outputs one or several ciphertexts with a noise independent of the input noise.*

An Atomic Pattern $\mathscr{A}$ is a particular instance of an AP type $\mathscr{A}^{(\cdot)}$. When an AP $\mathscr{A} \in \mathscr{A}^{(\cdot)}$ is instantiated with a parameter set x, we write $\mathscr{A}(x)$. From $\mathscr{A}(x)$ one can estimate the amount of noise at any edge of its FHE sub-graph and one can also estimate its total cost using a cost model.

Once we have identified the atomic pattern types in a graph $\mathcal{G} = (V, E)$, we can build an FHE DAG $\mathcal{G}' = (V', E')$ such that each FHE operator in V' is an atomic pattern i.e. $V' = \{A_i(\cdot)\}_{i \in [1, |V'|]}$. This new graph is equivalent to the input graph and we have $\mathcal{S}(\mathcal{G}) = \bigcap_{i \in [\![1, |V'|]\!]} \mathcal{S}(A_i(\cdot))$. We leverage the fact that we can compare the noise between atomic patterns of the same type to efficiently find the atomic patterns that have the smallest feasible sets. We will describe this procedure for a noise feasible set, but this can be extended to another kind of feasible set - for instance, the evaluation key sizes.

Two AP of the same type can be compared even without a given set of parameters. Hence we can introduce the notion of *domination between AP.*

Definition 27 (AP Domination)**.** *An AP A dominates A' if any $x \in \mathcal{P}(\mathcal{G})$ satisfying the noise constraints of A also satisfies the constraints of A'. More formally, we have $\mathcal{S}(A) \subset \mathcal{S}(A')$ i.e. $\mathcal{S}(A) \bigcap \mathcal{S}(A') = \mathcal{S}(A)$. A' is said to be dominated by A*

For all AP types in a graph $\mathcal{G}$, for all APs of this type, we can simply keep the ones that are not dominated by any other AP. Indeed, we can discard the APs that are dominated because their constraints will be satisfied if the constraints of one of their dominant AP are satisfied.

With TFHE, we mainly use three FHE operators: the homomorphic dot product (**DP**), the key switch and the programmable bootstrapping. The key switch is generally computed before the PBS (as in [CJP21]). We consider the noise formulae of [CLOT21] for the key switch and the bootstrapping. Because of the FFT in TFHE PBS, we had to add a corrective formula to take into account the noise added by the floating point representation. In particular, simply by casting the bootstrapping key from a *64-bit integers* to a *float* (represented with 64 bits) some of the LSB are lost. Similarly, the error grows all along computations in the Fourier domain due to the floating point arithmetic. To correct the formula accordingly, one solution is to collect data regarding the noise in many different parameter settings and use them to deduce a corrective formula that takes into account the FFT-induced error. Using this method, we found that the

following formula provides a good correction for the variance of the output of a bootstrapping: $\mathsf{FftError}_{k,N,\mathcal{B},\ell} = n \cdot 2^{\omega_1} \cdot \ell \cdot \mathcal{B}^2 \cdot N^2 \cdot (k+1)$ with $\omega_1 \approx 22 - 2.6$ (where 22 is $2 \cdot (64 - 53)$, since $q = 2^{64}$, 53 corresponds to the mantissa bits in the 64-bit floating point representation, and 2.6 is an experimental fitting).

All the experiments and benchmarks later provided in this manuscript will consider the algorithmic complexity of each FHE algorithm for the cost model. It means that we count the number of additions, multiplications, castings between integer types, and the asymptotic cost of the FFT in each algorithm and use it as a surrogate of the execution time. For instance, the operation $\sum_{i=1}^{\alpha} M_i \cdot \mathsf{CT}_i$ (where $M_i \in \mathfrak{R}_{q,N}$ are polynomials and $\mathsf{CT}_i = (A_i, B_i) \in \mathfrak{R}_{q,N}^2$ are RLWE ciphertexts) will have a cost of:

$$\underbrace{(2+1)\ \alpha\ N\log(N)}_{\text{to FFT domain}} + \underbrace{2\alpha\ N}_{\text{float }\times} + \underbrace{(\alpha-1)N}_{\text{float }+} + \underbrace{2\ N\log(N)}_{\text{to standard domain}} .$$

With this cost model, we assume the cost of a multiplication between floating point numbers or integers to be same than the cost of an addition between integers. While this hypothesis might be false in practice, it is close enough to provide efficient parameter sets. To simplify the problem, we assume the cost of the dot product to be negligible compared to the other FHE operators. Here, we assume the cost of an atomic pattern $\mathcal{A}$ to be the sum of the cost of every FHE operator inside it, i.e. the cost of a **PBS** and the cost of a **KS**.

A homomorphic dot product is a dot product between a vector of ciphertexts and a vector of integers. Notice that given some ciphertexts $\{\mathsf{ct}_i\}_{i\in I}$ with independent noises coming from $\mathcal{N}(0, \sigma^2)$ and some weights $\{\omega_i\}_{i\in I}$, the noise in the output ciphertext $\mathsf{ct}_{\mathsf{out}} = \sum_{i\in I} \mathsf{ct}_i \cdot \omega_i$ follows the distribution $\mathcal{N}(0, \nu^2\sigma^2)$ with $\nu_2^2 = \sum_{i\in I} \omega_i^2$, the squared 2-norm. Thus, given a dot product between a vector of ciphertexts with the same (normal) noise distribution and a vector of integers, we only need the 2-norm ν to characterize the output noise of a dot product.

Naturally, we define our first concrete atomic pattern type $\mathcal{A}^{(\mathrm{CJP21})}$ which is composed of a dot product, followed by a **KS** and a final **PBS** (i.e. a **MS**, a **BR** and a **SE**) as defined in [CJP21] (Figure 2.3). Here we assume every input of the dot product to be the output of a bootstrapping, hence we do not consider the fact that some of those inputs could be freshly-encrypted ciphertext. Everything we describe below is easily modifiable to take that into account. In the definition of the dot product, we saw that the 2-norm ν and the input variance are sufficient to compute the output noise of a dot product if every input ciphertext has the same normal noise distribution. Hence, an atomic pattern AP of type $\mathcal{A}^{(\mathrm{CJP21})}$ is entirely characterized by two values: the 2-norm ν and its noise bound t. We will note $A = A(\nu, t)$.

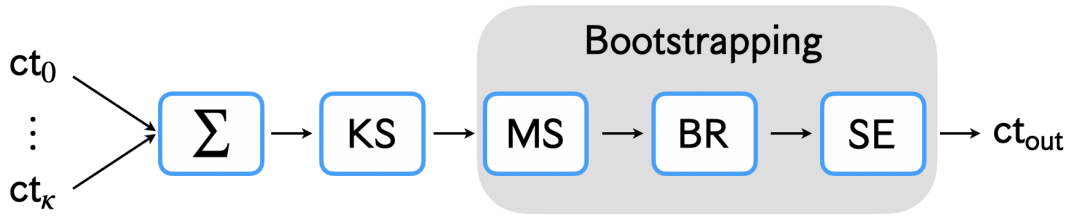

Figure 2.3: Representation of [CJP21] atomic pattern ($\mathcal{A}^{(\mathrm{CJP21})}$) where $\sum$ represents the homomorphic dot product (Theorem 2.7), **KS** the key switch (Algorithm 4), **MS** the modulus switch (Algorithm 6), **BR** the blind rotation (Algorithm 9) and **SE** the sample extract (Algorithm 10).

It is easy to compare the noise in atomic patterns of this type using the following property which is a special case of definition 27.

Theorem 2.16 (AP Domination)**.** *Let's consider $\mathcal{A}_1, \mathcal{A}_2 \in \mathcal{A}^{(\cdot)}$ two AP of a type that include a homomorphic dot product, ν_1, ν_2 two 2-norms such that $\nu_1 \leq \nu_2$ and t_1, t_2 two noise bounds where*

$t_2 \leq t_1$. *We have:* $\mathcal{S}(\mathcal{A}_2(\nu_2, t_2)) \subset \mathcal{S}(\mathcal{A}_1(\nu_1, t_1))$ *i.e.* $\mathcal{S}(\mathcal{A}_2(\nu_2, t_2)) \bigcap \mathcal{S}(\mathcal{A}_1(\nu_1, t_1)) = \mathcal{S}(\mathcal{A}_2(\nu_2, t_2))$. $\mathcal{A}_1$ *is said to be dominated by* $\mathcal{A}_2$

Proof *(Sketch).* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *share the same type. When decreasing the noise bound, i.e. going from* t_1 *to* t_2*, we have less possible solutions* x*, but all the ones that satisfy* t_2 *will satisfy* t_1*. The same reasoning works for the 2-norms. By increasing the 2-norm, i.e. going from* ν_1 *to* ν_2*, there are less possible solutions* x*, but all the solutions satisfying* ν_2 *will satisfy* ν_1*.* □

Given a graph $\mathcal{G} = \{A_i\}_{i \in I}$ of atomic patterns of type $\mathcal{A}^{(\text{CJP21})}$, we can apply the theorem above to simplify the construction of $\mathcal{S}(\mathcal{G})$. In fact, we do not need to build each $\mathcal{S}(A_i)$ as some of them are included in others. From our input graph $\mathcal{G}$, we construct a new graph $\mathcal{G}_{\mathsf{pareto}} = \mathsf{Pareto}(\mathcal{G}) = \{A'_i\}_{i \in I_{\mathsf{pareto}}}$ containing only non-dominated atomic patterns using theorem 2.16 (Pareto comes from *Pareto front*, well known in optimization). It follows that $\mathcal{G}_{\mathsf{pareto}}$ contains at most as many atomic patterns as there are different noise bounds in the graph.

An interesting property of $\mathcal{G}_{\mathsf{pareto}}$ is that $\mathcal{S}(\mathcal{G}) = \mathcal{S}(\mathcal{G}_{\mathsf{pareto}})$ i.e. if one solves the optimization problem (Eq. 2.2) using $\mathcal{S}(\mathcal{G}_{\mathsf{pareto}})$ instead of $\mathcal{S}(\mathcal{G})$, we will get the same optimal solution. This is interesting because to compute $\mathcal{S}(\mathcal{G}) = \bigcap_{i \in I} \mathcal{S}(A_i)$ we needed to build $|I|$ search spaces and with $\mathcal{G}_{\mathsf{pareto}}$, we only need to build $|I_{\mathsf{pareto}}|$ search spaces and most of the time $|I| \gg |I_{\mathsf{pareto}}|$.

Another useful observation is to notice that in an atomic pattern of type $\mathcal{A}^{(\text{CJP21})}$, the noise is strictly increasing until the end of the modulus switching step in the final PBS. As the noise bound is assumed to be constant inside one atomic pattern, we do not need to check that the noise satisfies the noise bound t after the dot product or after the key switching, we only need to do it after the modulus switch. If we note $\sigma_{\mathsf{MS},1}$, the standard deviation of the noise after the modulus switching in an atomic pattern $\mathcal{A}_1$, we have $\mathcal{S}(\mathcal{A}_1) = \{x \in \mathcal{P} | \sigma_{\mathsf{MS},1}(x) \leq t\}$.

As we assume the cost of a dot product to be negligible, the cost of an atomic pattern is only dependent on the cryptographic set of parameters and not on a particular instance of an atomic pattern of type $\mathcal{A}^{(\text{CJP21})}$.

For a graph $\mathcal{G} = \{\mathcal{A}_i\}_{i \in I}$, we have $\mathsf{Cost}\mathcal{G}, x = \sum_{i \in I} \mathsf{Cost}\mathcal{A}_i, x$ for x a solution in the search space $\mathcal{P}$ and we now that for any $(i, j) \in I^2, \mathsf{Cost}\mathcal{A}_i, x = \mathsf{Cost}\mathcal{A}_j, x$, so instead of minimizing the cost of running the total graph $\mathcal{G}$, we can settle for minimizing the cost of one atomic pattern of type $\mathcal{A}^{(\text{CJP21})}$.

To sum up, for a given graph $\mathcal{G}$, instead of solving equation 2.2, we can build a new graph $\mathcal{G}_{\mathsf{pareto}}$ as described above and solve the following which will give us the same value but will be easier to compute.

$$\underset{x \in \mathcal{P}}{\arg\min}\, \mathsf{Cost}\cdot, x \text{ s.t. } x \in \mathcal{S}(\mathcal{G}_{\mathsf{pareto}}). \tag{2.3}$$

The above problem is greatly simplified but still depends on the input graph $\mathcal{G} = \{\mathcal{A}(\nu_i, t_i)\}_{i \in I}$. It can be useful to have access to sets of parameters that work for a wide range of applications. Given a graph $\mathcal{G}$, we will be able to select the best set of parameters in those pre-computed sets.

A simple way to do that is to introduce another special graph $\mathcal{G}_{\mathsf{worst}}$, that we call the *worst case* atomic pattern. It is defined as $\mathcal{G}_{\mathsf{worst}} = \{\mathcal{A}(\max_{i \in I} \nu_i, \min_{i \in I} t_i)\}$. This graph is reduced to only one atomic pattern that may or may not be present if the input graph $\mathcal{G}$. Using theorem 2.16, we know that $\mathcal{S}(\mathcal{G}_{\mathsf{worst}}) \subset \mathcal{S}(\mathcal{G})$. So if we solve equation 2.3 on $\mathcal{G}_{\mathsf{worst}}$, we end up with a feasible solution for $\mathcal{G}$. Using this new graph, we are able to pre-compute sets of cryptographic parameters for different values of (ν, t). Given a graph $\mathcal{G}$, we will select the set of parameters for the worst case atomic pattern $\mathcal{G}_{\mathsf{worst}}$ of $\mathcal{G}$.

Above, we found a feasible solution and intuitively, this solution is close to the optimal one. To have bounds on the optimality of the solution for a graph $\mathcal{G} = \{\mathcal{A}_i\}_{i \in I}$, we can use another particular graph $\mathcal{G}_{\mathsf{best}}$ defined as $\mathcal{G}_{\mathsf{best}} = \{A(\nu^*, t^*)\}$ with $t^* = \min_{i \in I} t_i$ and $\nu^* = \max\{\nu_i | \mathcal{A}(\nu_i, t^*) \subset \mathcal{G}\}$ i.e. it is a graph composed of the atomic pattern of the graph $\mathcal{G}$ that have the smallest noise bound and the highest norm2 for this noise bound. If the worst case atomic pattern is the same as the best case atomic pattern, the method described above yields an optimal solution as $\mathcal{G}_{\mathsf{pareto}} = \mathcal{G}_{\mathsf{worst}} = \mathcal{G}_{\mathsf{best}}$. If they are different, we can deduce a bound of optimality: as $\mathcal{G}_{\mathsf{best}} \subset \mathcal{G}$,

we know that $\mathcal{S}(\mathcal{G}) \subset \mathcal{S}(\mathcal{G}_{\mathsf{best}})$. Solving equation 2.3 for $\mathcal{G}_{\mathsf{best}}$ give us a lower bound on the cost of the optimal solution of equation 2.3 for $\mathcal{G}$ and solving equation 2.3 for $\mathcal{G}_{\mathsf{worst}}$ give us an upper bound.

The atomic pattern types give us a powerful tool to compare several variants of the bootstrapping existing in the FHE literature. As different bootstrapping techniques have different cost-noise trade-offs, it is hard to compare them. By studying atomic patterns, we do not need to trouble ourselves with that, if one bootstrap yields more noise than another, it will be taken into account as the input noise of the atomic pattern will be higher.

2.5.4 Correctness and Security

As previously discussed, during the evaluation of an FHE computation graph, the noise grows until it reaches a point where bootstrapping (Algorithm 11) must be performed. The parameters derived from the methodologies explained in Section 2.5 guarantee a failure probability ($\mathsf{p}_{\mathsf{fail}}$) smaller than a predefined threshold.

Recently, the work in [LM21] showed that when errors occur in approximate FHE schemes such as CKKS, attacks can be mounted to recover elements of the secret key. This class of attacks, named IND-CPA$^{\mathrm{D}}$, leads to the definition of a new security model, in which the failure probability must be explicitly considered during parameter generation.

Even though TFHE is called an exact scheme, a small failure probability remains, and errors may still occur when the noise reaches its maximum. Later, in [CCP+24], the authors took advantage of this phenomenon to generalize this type of attack to exact FHE schemes, including TFHE. Even if an FHE scheme is called exact, an error may still occur during computation with some probability. This error is due to the noise distribution which usually follows a Gaussian distribution. There is a very small probability that the noise distribution becomes too large, compromising the message. This probability can be fixed in the same manner as security, i.e., through the selection of adequate cryptographic parameters. This new attack arises by breaking the parameter selection constraints and noise model. For example, an attacker could provide ciphertexts with noises exceeding what the algorithm supports.

Indeed, if an oracle indicates whether the computed result is correct or not, it becomes possible to retrieve certain secret key information. In response to this new attack, FHE schemes had to reduce the failure probability to ensure their security. This yields an enlargement of the lattice dimension, leading to a significant slowdown in all exact FHE schemes.

Remark 2.21. All the parameter sets proposed in this manuscript were obtained using the methodologies presented earlier. At the time of their generation, they guaranteed at least 128−bit of security. However, as these parameters have not been regenerated over time, some may no longer meet the same security standards due to advances new attacks.

Notably, prior to the emergence of IND-CPA$^{\mathrm{D}}$ attacks, the failure probability $\mathsf{p}_{\mathsf{fail}}$ was not treated as a critical security parameter. Graphs with a failure probability close to 2^{-40} were considered sufficiently secure, which explains why some of our reported results exhibit a relatively *high* $\mathsf{p}_{\mathsf{fail}}$

Today, to maintain a security level of $\lambda = 2^{128}$, the failure probability $\mathsf{p}_{\mathsf{fail}}$ should ideally be taken into account and set to 2^{-128}.

2.6 Limitations of TFHE

In Chapter 2, we introduced the fundamental principles of TFHE. Although the scheme is highly efficient, it still has several limitations that prevent it from reaching even greater efficiency. Below, we present a list of these limitations:

Limitation 1 (Padding Bit)**.** *As presented in Remark 2.14, to correctly perform a blind rotation, the padding bit needs to be known (and usually set to zero) to obtain the desired result.*

This limitation can naturally be addressed by evaluating only negacyclic functions, but the problem remains that we can only evaluate N distinct values when the message space lies within $[0, 2N-1]$.

Limitation 2 (Parallelization)**.** *The PBS (Algorithm 11)—the most computationally expensive algorithm used throughout any TFHE graph—cannot be easily parallelized due to its core process. Indeed, during the chain of CMux operations, each CMux must wait for the result of the previous one before it can proceed.*

However, parallelizing the bootstrapping would lead to significant performance improvements in any TFHE graph evaluation.

Limitation 3 (Noise Plateau)**.** *This limitation was already presented in Definition 20.*

We recall that the noise plateau is the threshold in the size of a ciphertext beyond which noise can no longer be reduced without compromising security. This noise plateau leads to an unnecessary level of security when working with large dimensions.

Limitation 4 (Precision)**.** *TFHE is very efficient for small precision (smaller than 8 bits). This limitation stems from the bootstrapping procedure. To perform bootstrapping, polynomial multiplications must be carried out with polynomials of degree N, and this degree is directly related to the number of bits that need to be bootstrapped. The more bits we want to bootstrap, the larger the polynomial degree must be, which in turn slows down the bootstrapping process.*

Limitation 5 (Public Material Size)**.** *To perform algorithms such as key switching (Algorithm 5) or bootstrapping (Algorithm 11), one needs access to public material (see Remark 2.15).*

In some cases, the size of the public material can be very large, reaching several gigabytes.

Limitation 6 (GLWE Secret Key Size)**.** *GLWE ciphertexts work with polynomials whose degree is a power of two (in $\mathfrak{R}_{q,N}$ with N a power of two). If larger polynomials are needed, it becomes necessary to double the polynomial degree. When we twice the polynomial size, we cannot reduce the secret key size by choosing an arbitrary number of secret key coefficients. This limitation is closely related to Limitation 3.*

Limitation 7 (Multi inputs)**.** *TFHE bootstrapping operates on a single input ciphertext, which restricts its use to univariate function evaluation. Extending programmable bootstrapping (PBS) to support multiple input ciphertexts would enable the evaluation of multivariate functions.*

Limitation 8 (Multi LUT evaluation)**.** *TFHE bootstrapping operates with a single lookup table as input. Therefore, evaluating multiple functions requires performing a separate bootstrapping for each function.*

Limitation 9 (Efficiency)**.** *Even though TFHE is becoming increasingly faster, it still remains slow compared to the plaintext evaluation of any graph. Reducing the gap between these two worlds is currently one of the biggest challenges in TFHE, and more generally, in FHE.*

Naturally, this list is non-exhaustive, and other limitations may still emerge as the field continues to evolve.

Chapter 3

More Feature in TFHE

The previous chapter introduced the foundation of the TFHE scheme. While these initial operations already support a wide range of use cases, several limitations remain, as discussed in Subsection 2.6. Addressing these limitations is essential for designing a less constrained homomorphic encryption scheme.

In this chapter, we present a set of algorithms from the literature that address specific limitations identified in TFHE. Each algorithm has been studied independently, but many of them can be combined to improve existing techniques.

In the first section, we introduce several well known algorithms that expand the design space for new cryptographic constructions. Alongside these tools, we also present improvements to the bootstrapping procedure that allow the evaluation of larger lookup tables with reduced public material, allowing us to address Limitation 5. In the second section, we present advanced algorithms aimed at overcoming Limitation 2, enabling parallelization of different steps of the bootstrapping process. The third section introduces new techniques to address Limitation 1, allowing bootstrapping to be performed without relying on a fixed padding bit. Finally, the last section presents several methods designed to overcome Limitation 8, enabling the evaluation of multiple lookup tables using only one algorithm.

All algorithms have been studied during this thesis, but the bit extraction (Algorithm 15) and the extended bootstrapping (Algorithm 17 and Algorithm 18) have been deeply studied in our articles [BBB+23] and [BORT25]; therefore, more details are provided in this chapter. For the other algorithms, more details can be found in their respective original publication.

In the following chapters, we illustrate how the proposed algorithms are employed and integrated into larger constructions (Chapters 7 and 8), how we have improved them (Chapter 4), and how our proposed solutions achieve better efficiency (Chapter 6).

3.1 Relevant Algorithms and Improved Bootstrapping

This section introduces a set of new tools that enhance the capabilities of the TFHE scheme. For instance, we present an algorithm, named circuit bootstrapping, which transforms an LWE ciphertext encrypting a message m into a GGSW ciphertext encrypting the same message.

We also describe two methods for evaluating lookup tables, the vertical and the horizontal packing, as well as an algorithm that enables the extraction of all message bits encrypted in an LWE ciphertext.

Finally, we introduce a technique to perform a **PBS** with multiple input ciphertexts, and another method that improves the bootstrapping procedure when working with large polynomials.

3.1.1 Circuit Bootstrapping

Circuit bootstrapping is an algorithm that transforms an LWE ciphertext into a GGSW ciphertext. This technique was first introduced in [CGGI17]. It takes as input an LWE ciphertext, a bootstrapping key (BSK) and several key switching key. Then it outputs a GGSW ciphertexts encrypting the input message m. In addition to create a GGSW ciphertext, the circuit bootstrapping also reduces the noise of the input ciphertext.

The circuit bootstrapping algorithm begins by performing $\ell_{\mathbf{CBS}}$ programmable bootstrapping to create $\ell_{\mathbf{CBS}}$ LWE ciphertexts encrypting the message $m\frac{q}{\mathcal{B}^i}$ for $i \in [1, \ell_{\mathbf{CBS}}]$ under the flatten GLWE secret key $\boldsymbol{s}'$. Then it performs $(k+1)$ private key switch (Remark 2.12) of each of the $\ell_{\mathbf{CBS}}$ ciphertexts to obtain $(k+1)\cdot\ell_{\mathbf{CBS}}$ GLWE ciphertexts encrypting $m\frac{q}{\mathcal{B}^i}S_j$ for $i \in [1, \ell_{\mathbf{CBS}}]$ and $j \in [0, k]$. These ciphertexts are then rearranged to form the final GGSW ciphertext. The complete circuit bootstrapping procedure is detailed in Algorithm 12, and the corresponding output noise bounds can be derived using the proofs of Theorem 2.8 and Theorem 2.15 with further details available in [CGGI17].

Algorithm 12: $\overline{\overline{\mathsf{CT}}}_{\mathsf{out}} \leftarrow \mathbf{CBS}\left(\mathsf{ct}_{\mathsf{in}}, \mathsf{BSK}, \{\mathsf{KSK}_j\}_{j\in[0,k]}\right)$

Context:
- $\boldsymbol{s} = (s_0, \ldots, s_{n-1})$: the LWE secret key
- $\boldsymbol{S'} = (S'_0, \ldots, S'_{k-1})$: the GLWE secret key
- $\boldsymbol{s'}$ the flatten representation of the GLWE secret key, Definition 11
- $\Delta m + e \in \mathbb{Z}_q$: The encoded message of $\mathsf{ct}_{\mathsf{in}}$, Definition 8
- LUT_{f_i} : The Lookup table evaluating the function $m \mapsto m\frac{q}{\mathcal{B}_{\mathbf{CBS}}^i}$, Definition 16
- CT_{f_i} : Trivial encryption of LUT_{f_i}, Remark 2.6
- $(\mathcal{B}_{\mathbf{CBS}}, \ell_{\mathbf{CBS}}) \in (\mathbb{Z}^*)^2, (\mathcal{B}_{\mathbf{PBS}}, \ell_{\mathbf{PBS}}) \in (\mathbb{Z}^*)^2, (\mathcal{B}_{\mathbf{KS}}, \ell_{\mathbf{KS}}) \in (\mathbb{Z}^*)^2$: The (base, level) decomposition for the **CBS**, resp. **PBS**, resp. **KS**.
- $\mathsf{KSK}_{i\in[0,k-1]} = \left\{\mathsf{KSK}_{i,j} \in \mathrm{GLEV}^{\mathcal{B}_{\mathbf{KS}},\ell_{\mathbf{KS}}}_{\boldsymbol{S}_{\mathsf{out}}}(-S'_i \cdot s'_j)\right\}_{j\in[0,n-1]}$
  $\cup \left\{\mathsf{KSK}_{i,n} \in \mathrm{GLEV}^{\mathcal{B}_{\mathbf{KS}},\ell_{\mathbf{KS}}}_{\boldsymbol{S}_{\mathsf{out}}}(S'_i)\right\}$
- $\mathsf{KSK}_k = \left\{\mathsf{KSK}_{k,j} \in \mathrm{GLEV}^{\mathcal{B}_{\mathbf{KS}},\ell_{\mathbf{KS}}}_{\boldsymbol{S}_{\mathsf{out}}}(s'_j)\right\}_{j\in[0,n-1]}$

Input:
- $\mathsf{ct}_{\mathsf{in}} \in \mathrm{LWE}_{\boldsymbol{s}}(m) \subseteq \mathbb{Z}_q^{n+1}$
- $\mathsf{BSK} = \left\{\mathsf{BSK}_i \in \mathrm{GGSW}^{\mathcal{B}_{\mathbf{PBS}},\ell_{\mathbf{PBS}}}_{\boldsymbol{S'}}(s_i)\right\}_{i\in[0,n-1]}$
- $\mathsf{KSK} = \{\mathsf{KSK}_j\}_{j\in[0,k]}$

Output: $\left\{\overline{\overline{\mathsf{CT}}}_{\mathsf{out}} \in \mathrm{GGSW}^{\mathcal{B}_{\mathbf{CBS}},\ell_{\mathbf{CBS}}}_{\boldsymbol{S'}}(m)\right.$

for $i \in [1, \ell_{\mathbf{CBS}}]$ **do**
  $\mathsf{ct}_i \leftarrow \mathbf{PBS}(\mathsf{ct}_{\mathsf{in}}, \mathsf{LUT}_{f_i}, \mathsf{BSK})$; `/* Algorithm 11 */`
for $i \in [1, \ell_{\mathbf{CBS}}]$ **do**
  for $j \in [0, k]$ **do**
    $\mathsf{CT}_{i,j} \leftarrow \mathsf{PrivateKS}(\mathsf{ct}_i, \mathsf{KSK}_j)$; `/* Algorithm 4 with Remark 2.12 */`
return $\overline{\overline{\mathsf{CT}}}_{\mathsf{out}} = \{\mathsf{CT}_{i,j}\}_{i\in[1,\ell_{\mathbf{CBS}}]}^{j\in[0,k]}$

Remark 3.1. The cost of the circuit bootstrapping (Algorithm12) can be approximated by:

$$\mathsf{Cost}_{\mathbf{CBS}}^{\ell_{\mathbf{PBS}},\ell_{\mathbf{CBS}},k,N,n,q} = \ell_{\mathbf{CBS}} \cdot \mathsf{Cost}_{\mathbf{PBS}}^{\ell_{\mathbf{PBS}},k,N,n,q} + \ell_{\mathbf{CBS}}(k+1) \cdot \mathsf{Cost}_{\mathsf{PrivateKS}}^{\ell_{\mathbf{CBS}},k,N}$$

Where $\mathsf{Cost}_{\mathsf{PrivateKS}}^{\ell_{\mathbf{CBS}},k,N}$ is equivalent to the cost of an external product ($\mathsf{Cost}_{\mathsf{ExternalProduct}}^{\ell_{\mathbf{CBS}},k,N}$).

3.1.2 Horizontal and Vertical Packing

In the original TFHE paper [CGGI16a], the authors proposed two algorithms: *vertical packing* and *horizontal packing*, both allowing evaluation of lookup tables without reducing the noise.

First, horizontal packing takes as input a single message m composed of p bits, where the message is encrypted in p GGSW ciphertexts (Definition 14), where each GGSW ciphertext corresponds to a bit of m, along with 2^p lookup tables, $\mathsf{LUT}_0, \dots, \mathsf{LUT}_{2^p-1}$. Then, horizontal packing outputs the m^{th} lookup table $\mathsf{LUT}_m \in \mathfrak{R}_{q,N}$ encrypted in a GLWE ciphertext. The core idea is to construct a binary tree using CMux operations (Algorithm 8) on all the lookup tables to select the one corresponding to the message. At each step of the CMux-tree, the algorithm removes half of the lookup tables. The complete process is detailed in Algorithm 13, and additional explanations are provided in [CGGI16a].

Algorithm 13: $\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{HorizontalPacking}\left(\{\mathsf{LUT}_{f_i}\}_{i\in[0,2^p-1]}, \left\{\overline{\overline{\mathsf{CT}}}_i\right\}_{i\in[0,2^p-1]}\right)$

Context:
- $\boldsymbol{S} \in \mathfrak{R}_{q,N}^k$: The GLWE secret key
- m : The p-bits input message such that,$m = \sum_{i=0}^{p-1} m_i 2^i$
- $\mathsf{LUT}_f = [\mathsf{LUT}_{f_0}, \dots, \mathsf{LUT}_{f_{2^p-1}}]$: The Lookup table evaluating the function $m \mapsto f(m)$, Definition 16 with $\mathsf{LUT}_{f_i} \in \mathfrak{R}_{q,N}^k$
- $\{\mathsf{CT}_{f_i}\}_{i\in[0,2^p-1]}$: Trivial encryption of LUT_{f_i}, Remark 2.6
- $(\mathcal{B}, \ell) \in (\mathbb{Z}^*)^2$: The (base, level) decomposition.

Input:
- $\{\mathsf{LUT}_{f_i}\}_{i\in[0,2^p-1]}$
- $\left\{\overline{\overline{\mathsf{CT}}}_i \in \mathrm{GGSW}_{\boldsymbol{s}}^{\mathcal{B},\ell}(m_i)\right\}_{i\in[0,2^p-1]}$

Output: $\left\{\mathsf{CT}_{\mathsf{out}} \in \mathrm{GLWE}_{\boldsymbol{S}}(\mathsf{LUT}_{f_m})\right.$

```
for i ∈ [0, p − 1] do
    for j ∈ [0, 2^{p−i}] do
        CT_{f_j} ← CMux(CT_{f_{2j}}, CT_{f_{2j+1}}, CT̿_i) ;        /* Algorithm 8 */
return CT_out = CT_{f_0}
```

As with horizontal packing, vertical packing takes as input a single message m composed of p bits, where each of these bits is encrypted into a GGSW ciphertext, and $2^{p-\log(N)}$ lookup tables $\mathsf{LUT}_0 = (l_0, \dots, l_{N-1}), \dots, \mathsf{LUT}_{2^{p-\log(N)}-1} = (l_{2^p-N}, \dots, l_{2^p-1})$. It outputs an LWE ciphertext encrypting the value l_m.

At a high level, the algorithm uses the $p - \log(N)$ most significant bits of m, each encrypted in a GGSW ciphertext, to perform horizontal packing and select the correct lookup table that contains l_m. Then, using the remaining $\log(N)$ bits, it performs a blind rotation to select the appropriate coefficient within the chosen lookup table. This operation can also be seen as a way to evaluate large lookup tables containing more than N entries: the full table is split into smaller lookup tables of size N, horizontal packing selects the correct chunk, and blind rotation extracts the desired value. We note that if $2^p < N$, this operation reduces to a simple blind rotation (Algorithm 9). As with the previous algorithm, the complete process is described in Algorithm 14, and further details are provided in [CGGI16a].

Remark 3.2. The cost of the horizontal packing (Algorithm13) can be approximated by:

$$\mathsf{Cost}^{p,\ell,k,N}_{\mathsf{HorizontalPacking}} = (2^p - 1) \cdot \mathsf{Cost}^{\ell,k,N}_{\mathsf{CMux}}.$$

The cost of the vertical packing (Algorithm14) can be approximated by:

$$\mathsf{Cost}^{p,\ell,k,N}_{\mathsf{VerticalPacking}} = \mathsf{Cost}^{\max(0,p-\log_2(N)),\ell,k,N}_{\mathsf{HorizontalPacking}} + \min(\log_2(N), p) \cdot \mathsf{Cost}^{\ell,k,N}_{\mathsf{CMux}} + \mathsf{Cost}^{k,N}_{\mathsf{SampleExtract}}.$$

Algorithm 14: $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{VerticalPacking}\left(\left\{\overline{\overline{\mathsf{CT}}}_i\right\}_{i\in[0,2^p-1]}, \{\mathsf{LUT}_{f_i}\}_{i\in[0,2^{p-\log_2(N)}-1]}\right)$

Context:
$\begin{cases} \boldsymbol{S} \in \mathfrak{R}_{q,N}^k : \text{The GLWE secret key} \\ m : \text{The } p\text{-bits input message such that, } m = \sum_{i=0}^{p-\log_2(N)-1} m_i 2^i \\ \mathsf{LUT}_f = [\mathsf{LUT}_{f_0}, \ldots, \mathsf{LUT}_{f_{2^{p-\log_2(N)}-1}}] : \text{The Lookup table evaluating} \\ \qquad \text{the function } m \mapsto f(m), \text{ Definition 16 with } \mathsf{LUT}_{f_i} \in \mathfrak{R}_{q,N}^k \\ (\mathfrak{B}, \ell) \in (\mathbb{Z}^*)^2 : \text{The (base, level) decomposition.} \end{cases}$

Input:
$\begin{cases} \{\mathsf{LUT}_{f_i}\}_{i\in[0,2^{p-\log_2(N)}-1]} \\ \left\{\overline{\overline{\mathsf{CT}}}_i \in \mathrm{GGSW}_{\boldsymbol{s}}^{\mathfrak{B},\ell}(m_i)\right\}_{i\in[0,p-1]} \end{cases}$

Output: $\left\{\mathsf{ct}_{\mathsf{out}} \in \mathrm{LWE}_{\boldsymbol{S}}(\mathsf{LUT}_f[m])\right.$

`/* Algorithm 13 */`

$\mathsf{CT} \leftarrow \mathsf{HorizontalPacking}\left(\{\mathsf{LUT}_{f_i}\}_{i\in[0,2^{p-\log_2(N)}-1]}, \left\{\overline{\overline{\mathsf{CT}}}_i\right\}_{i\in[\log_2(N),p-1]}\right)$

for $i \in [0, \log_2(N) - 1]$ **do**

  $\mathsf{CT} \leftarrow \mathsf{CMux}(\mathsf{CT}, \mathsf{CT} \cdot X^{2^i}, \overline{\overline{\mathsf{CT}}}_i)$; `/* Algorithm 8 */`

$\mathsf{ct}_{\mathsf{out}} \leftarrow$ **SampleExtract**(CT); `/* Algorithm 10 */`

return $\mathsf{ct}_{\mathsf{out}}$

3.1.3 Bit Extraction

Bit extraction is an algorithm that enables the recovery of all the bits of a message encrypted in an LWE ciphertext. In what follows, we describe how to extract the bits starting from the least significant bit (LSB) up to the most significant bit (MSB). The idea is to iteratively extract the LSB and subtract it from the input ciphertext. This operation reduces the message so that the next least significant bit becomes the new LSB. The process is repeated until all bits of the message m have been extracted.

This algorithm can also be easily modified to extract only a subset of the bits of the input message. As the bit extraction is performed through the evaluation of a negacyclic function, this algorithm can be executed even when the padding bit is unknown, addressing Limitation 1.

Lemma 3.1 (Bit Extraction (Algorithm 15))**.** *Let $\boldsymbol{s} = (s_0, \ldots, s_{n-1}) \in \mathbb{Z}_q^n$ be a binary LWE secret key and $\boldsymbol{S'} \in \mathfrak{R}_{q,N}^k$ be a GLWE secret key and $\boldsymbol{s'}$ his flatten representation. Let $\mathsf{ct}_{\mathsf{in}} = (a_{\mathsf{in},0}, \ldots, a_{\mathsf{in},n-1}, b_{\mathsf{in}}) \in \mathrm{LWE}_{\boldsymbol{s}}(m) \subseteq \mathbb{Z}_{2N}^{n+1}$ encrypting the message $m = \sum_{i=0}^{\delta-1} m_i 2^i \in \mathbb{Z}_p$ with $\delta = \log p$. Let PUB be the public material (BSK and KSK, Remark 2.15).*

Then Algorithm 15 returns a list of LWE ciphertexts encrypting the bits of m, i.e., $\left\{\mathsf{ct}_i \in \mathrm{LWE}_{\boldsymbol{s}}(m_i) \subseteq \mathbb{Z}_q^{kN+1}\right\}_{i\in[0,\delta)}$. An LWE ciphertext encrypting $f(m + e \mod 2N)$ under the secret key $\boldsymbol{s'}$. The cost of this algorithm is:

$$\mathsf{Cost}_{\mathsf{BitExtract}}^{\ell,k,N,n,q,\delta} = (\delta - 1)\left(\mathsf{Cost}_{\mathbf{PBS}}^{\ell,k,N,n,q} + \mathsf{Cost}_{\mathbf{KS}}^{\ell,n,k,N} + \mathsf{Cost}_{\mathsf{Add}}^{k,N}\right).$$

Proof *(Lemma 3.1). To extract the δ occupied bits of the message in a ciphertext, i.e., $m = \sum_{i=0}^{\delta-1} m_i 2^i$, the goal is to extract each bit from the least significant to the most significant and store each result in an individual ciphertext. Each step consists in extracting the current least significant bit.*

At each step, we shift the remaining least significant bit into the padding bit by multiplying the ciphertext ct by $2^{\delta+\pi-1-i}$. Thus, at the i^{th} step, after the shift, the ciphertext ct_i encrypts a message in $[0, \frac{q}{2})$ if $m_i = 0$, and in $[\frac{q}{2}, q)$ if $m_i = 1$.

Due to the negacyclicity property (see Remark 2.1), we cannot directly perform bootstrapping to extract the value of m_i. In fact, applying a bootstrapping with a lookup table encoding the

Algorithm 15: $\mathsf{ct}_0 \dots \mathsf{ct}_{\delta-1} \leftarrow \mathsf{BitExtract}(\mathsf{ct}, \mathsf{PUB})$

Context:
- $\boldsymbol{s} = (s_0, \dots, s_{n-1}) \in \mathbb{Z}_q^n$: the LWE secret key
- $\boldsymbol{S'} = (S'_0, \dots, S'_{k-1}) \in \mathfrak{R}_{q,N}^k$: the GLWE secret key
- $\boldsymbol{s'}$ the flatten representation of the GLWE secret key, Definition 11
- $\Delta m + e \in \mathbb{Z}_q$: The encoded message of $\mathsf{ct}_{\mathsf{in}}$, Definition 8
- π : Number of padding bits
- δ : bits occupied by message in ciphertext ct
  i.e., $m = \sum_{i=0}^{\delta} m_i 2^i$ with $m \in \mathbb{Z}_p$
- $(\mathcal{B}, \ell) \in (\mathbb{Z}^*)^2$: The (base, level) decomposition for the **PBS**

Input:
- $\mathsf{ct} \in \mathrm{LWE}_{\boldsymbol{s'}}(m)$
- PUB : public keys required for the entire algorithm ; `/* Remark 2.15 */`

Output: $(\mathsf{ct}_0 \dots \mathsf{ct}_{\delta-1})$

1 **for** $i \in [0; \delta - 2]$ **do**
2 $\quad \epsilon = \frac{q}{4}$
3 $\quad$ **if** $i == 0$ **and** $\frac{q}{p} \notin \mathbb{N}$ **then**
4 $\quad\quad \epsilon \leftarrow \left\lfloor \frac{q}{4p} \right\rfloor$
5 $\quad \alpha = \frac{q}{2^{\delta+\pi+1}}$
6 $\quad L = [-\alpha, \dots, -\alpha]$
7 $\quad \mathsf{ct}_i \leftarrow \mathsf{ct} \cdot 2^{\delta+\pi-1-i} + (0, \dots, 0, \epsilon)$
8 $\quad \mathsf{ct}_{\mathbf{KS}_i} \leftarrow \mathbf{KS}(\mathsf{ct}_i, \mathsf{PUB})$; `/* Algorithmn4 */`
9 $\quad \mathsf{ct}'_i \leftarrow \mathbf{PBS}(\mathsf{ct}_{\mathbf{KS}_i}, \mathsf{PUB}, L)$; `/* Algorithmn11 */`
10 $\quad \mathsf{ct}_i \leftarrow \mathsf{ct}'_i + (0, \dots, 0, \alpha)$
$\quad$ `/* Subtract the extracted bit from the original ciphertext */`
11 $\quad \mathsf{ct} \leftarrow \mathsf{Sub}(\mathsf{ct}, \mathsf{ct}_i)$
12 $\mathsf{ct}_{\delta-1} \leftarrow \mathsf{ct}$
13 **return** $\mathsf{ct}_0 \dots \mathsf{ct}_{\delta-1}$

polynomial $P(X) = -\alpha \cdot \sum_{i=0}^{N-1} X^i$ *returns* $-\alpha$ *when the encrypted message of* ct_i *is in* $[0, \frac{q}{2})$*, and* α *when it is in* $[\frac{q}{2}, q)$*, i.e.,*

$$\begin{cases} \alpha & \text{if } \mathsf{Decrypt}(\mathsf{ct}_i \cdot 2^{\delta+\pi-1-i}) \in [\frac{q}{2}, q) \\ -\alpha & \text{otherwise.} \end{cases}$$

However, if the encrypted value is close to the bounds 0 *or* $\frac{q}{2}$*, noise can influence the result. For example, if* ct_i *encrypts* $\frac{q}{2}$ *(i.e.,* $m_i = 1$*) with a negative noise, it may yield the same output as* ct_i *encrypting* 0 *(i.e.,* $m_i = 0$*) with a positive noise. As a result, we cannot reliably extract bit values directly using this lookup table.*

To resolve this, we must be cautious when encoded values are close to the thresholds 0 *or* $\frac{q}{2}$*, as only the noise will determine the* **PBS** *output, potentially causing incorrect results. To mitigate this issue, we introduce a corrective term in Line 7. In order to choose the right correcting term, we need to determine the smaller distance (denoted* $\mathsf{d}(\cdot, \cdot)$*) between the preceding encoded value* v_1 *and* q*, and the preceding encoded value* v_2 *and* $\frac{q}{2}$*, i.e.:*

$$\min\left(\mathsf{d}\left(v_1 \in \left[0; \frac{q}{2}\right), \frac{q}{2}\right), \mathsf{d}\left(v_2 \in \left[\frac{q}{2}; q\right), q\right)\right).$$

We now compute the two distances. At this point, we need to distinct between two cases:

1. p ***is a power of two***, *i.e.,* $p = 2^\delta$. *The message* m *are encoded by* $\frac{q}{p \cdot 2^\pi} \cdot m \mod q$ *with* $m \in [0, 2^\delta)$. *For the shift we compute* $\frac{q}{p \cdot 2^\pi} \cdot i \cdot 2^{\delta+\pi-1} \mod q$, *so the only remaining encoded values are* 0 *and* $\frac{q}{2}$, *so the distance between these two values is* $d = \frac{q}{2}$ *and the correcting term can be bound by half of this distance, i.e.,* $\epsilon = \frac{q}{4}$.

2. p ***is not a power of two***. *As* p *is much smaller than* q, *after the first shift, for all* $j \in [0, p)$ *we obtain the following bound on the encoded values:*

$$\begin{aligned}\left\lfloor \frac{q}{p \cdot 2^\pi} \cdot j \right\rceil \cdot 2^{\delta+\pi-1} \mod q &\leq \left(\frac{q}{p \cdot 2^\pi} \cdot j + \frac{1}{2} \right) \cdot 2^{\delta+\pi-1} \mod q \\ &\leq \frac{q}{p} \cdot j \cdot 2^{\delta-1} + 2^{\delta+\pi-2} \mod q.\end{aligned}$$

The next is step is to compute the minimum of the distances:

$$\min \left(\mathsf{d}\left(v_1 \in \left[0; \frac{q}{2}\right), \frac{q}{2} \right), \mathsf{d}\left(v_2 \in \left[\frac{q}{2}; q\right), q \right) \right) - 2^{\delta+\pi-2}, v_{i\in[1,2]} \in \left\{ \frac{q}{p} \cdot j \mod q \right\}_{j \in [0, p-1]}.$$

First we can bound v_1*:*

$$v_1 \leq \frac{q}{2} - \left\lfloor \frac{q}{2p} \right\rceil.$$

So we have $d_1 = \mathsf{d}\left(v_1 \in [0; \frac{q}{2}), \frac{q}{2}\right) \geq \left\lfloor \frac{q}{2p} \right\rceil$.

Next we can bound v_2*:*

$$v_2 \leq q - \left\lfloor \frac{q}{p} \right\rceil.$$

So we have $d_2 = \mathsf{d}\left(v_2 \in \left[\frac{q}{2}; q\right), q\right) > \left\lfloor \frac{q}{p} \right\rceil \geq \left\lfloor \frac{q}{2p} \right\rceil$. *The distance is then bounded by* $\left\lfloor \frac{q}{2p} \right\rceil - 2^{\delta+\pi-2}$. *The correcting term is finally defined as half of this bound, i.e.,* $\epsilon = \left\lfloor \frac{q}{4p} \right\rceil - 2^{\delta+\pi-3}$.

Since the term $2^{\delta+\pi-2}$ *is very small regarding* q, *it can be neglected. About the noise bound, this term is also negligible, since it is smaller than* 1 *before the shift.*

By adding ϵ *to* $\mathsf{ct}_i \cdot 2^{\delta+\pi-1}$ *we ensure that for any message, an error* e *of size* $|e| < \epsilon$ *will lead to a correct PBS evaluation. This means that before the shift, the noise in* ct_i *should be smaller than* $\epsilon \cdot 2^{-\lfloor \delta+\pi-1 \rfloor}$.

At this point, the less significant bit has been extracted and stored into a new LWE_0. *To extract the next bit, we first subtract* LWE_0 *to the input ciphertext* ct. *With this operation we ensure that the less significant bit is now equal to* 0. *As we want to extract the second less significant bit, we now shift by* $2^{\delta+\pi-2}$. *Finding the corrective term* ϵ *is much easier in this case, as the second bit is equal to* 0 *after the shift. Hence, we can take* $\epsilon = \frac{q}{4}$ *and extract the bit with a* **PBS**. *To extract the remaining bits, we just need to repeat the previous steps (subtraction, shift, add* $\epsilon = \frac{q}{4}$ *and* **PBS***).* □

3.1.4 Multivariate Bootstrapping

In Section 2.6, one of the identified limitations is that PBS takes only a single ciphertext as input (see Limitation 7). Allowing multiple ciphertexts as input to a single PBS would enable the evaluation of multivariate functions.

In this section we will see how to take advantage of the encoding presented in Section 2.4 to perform multivariate using a trick that was already proposed in [CZB+22]. If the degrees of the ciphertexts allow, the idea is to concatenate two messages m_1 and m_2 (or more) respectively

encrypted in ct_1 and ct_2 by re-scaling the first one with constant multiplication to $\mu_2 + 1$ (where μ_2 is the worst possible value that can be reached by the m_2) and add it to ct_2 and finally compute a PBS on the concatenation. Once the two messages are concatenated in a single ciphertext, the bi-variate LUT L can be simply evaluated as a univariate LUT L' on the concatenation of m_1 and m_2. A visual example of Algorithm 16 in the bivariate case is proposed in Figure 3.1;

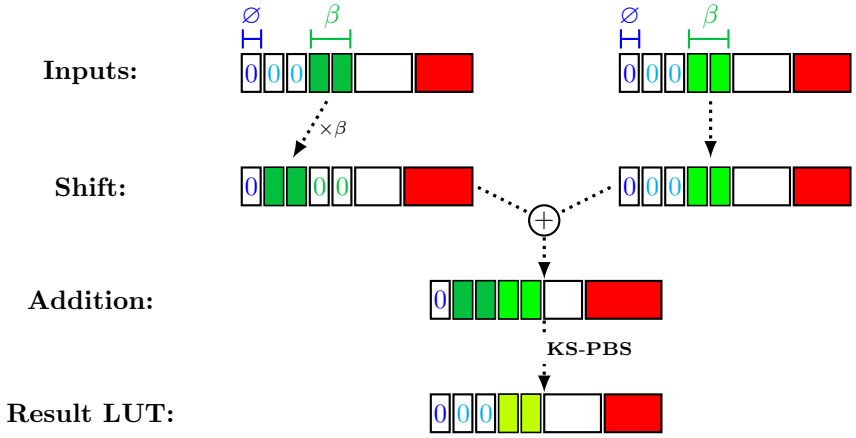

Figure 3.1: Example of a bi-variate LUT evaluation with shift and PBS.

Algorithm 16: $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{Multivariate}-\mathbf{PBS}(\mathsf{ct}_{i i\in[0,\alpha]}, \mathsf{LUT}_f, \mathsf{BSK})$

Context:
- $\boldsymbol{s} = (s_0, \ldots, s_{n-1})$: the LWE secret key
- $\boldsymbol{S'} = (S'_0, \ldots, S'_{k-1})$: the GLWE secret key
- $\boldsymbol{s'}$ the flatten representation of the GLWE secret key, Definition 11
- $\alpha + 1$: The number of input ciphertext
- $\mathsf{ct}_{i\in[0,\alpha]}$: The input ciphertext encoded as define in Section 2.4.3 with an empty carry and with $p = \beta^\alpha$
- LUT_f : The Lookup table evaluating the function $(x_0, \cdots x_\alpha) \mapsto f(x_0, \cdots x_\alpha)$
- CT_f : Trivial encryption of LUT_f, Remark 2.6
- $(\mathcal{B}, \ell) \in (\mathbb{Z}^*)^2$: The (base, level) decomposition.

Input:
- $\mathsf{ct}_i \in \mathrm{LWE}_{\boldsymbol{s}}(m_i)_{i\in[0,\alpha]}$
- $\mathsf{CT}_f \in \mathrm{GLWE}_{\boldsymbol{S}}(\mathsf{LUT}_f) \subseteq \mathfrak{R}_{q,N}^{k+1}$
- $\mathsf{BSK} = \left\{\mathsf{BSK}_i \in \mathrm{GGSW}_{\boldsymbol{S'}}^{\mathcal{B},\ell}(s_i)\right\}_{i\in[0,n-1]}$

Output: $\left\{\mathsf{ct}_{\mathsf{out}} \in \mathrm{LWE}_{\boldsymbol{s'}}\left(f\left(m_0, \ldots, m_\alpha\right)\right) \subseteq \mathbb{Z}_q^{Nk+1}\right.$

$\mathsf{ct} \leftarrow (\mathbf{0}, 0) \in \mathbb{Z}_q^{n+1}$
for $i \in [0, \alpha]$ **do**
  $\mathsf{ct}_i \leftarrow \mathsf{ct}_i \cdot \beta^i$; /* Theorem 2.5 */
  $\mathsf{ct} \leftarrow \mathsf{ct} + \mathsf{ct}_i$; /* Algorithm 3 */
$\mathsf{ct}_{\mathsf{out}} \leftarrow \mathbf{PBS}(\mathsf{ct}, \mathsf{LUT}_f, \mathsf{BSK})$; /* Algorithm 11 */
return $\mathsf{ct}_{\mathsf{out}}$

3.1.5 Extended Bootstrapping

In [LY23], the authors propose a new method to evaluate large lookup tables in $\mathfrak{R}_{q,\eta N}$, where $\eta \in \mathbb{Z}$ is a power of two, by using public materials composed of polynomials in $\mathfrak{R}_{q,N}$ and extending these polynomials to $\mathfrak{R}_{q,\eta N}$. They name this new bootstrapping techniqueExtended Programmable Bootstrapping (**EBS**). Our work [BORT25], in addition to other improvements presented in Chapter 4, extends this result to arbitrary cyclotomic polynomials.

This improvement is particularly efficient when the polynomial size reaches the noise plateau (Limitation 3). Moreover, it enables bootstrapping of higher precision messages, addressing Limitation 4.

First let us define the ι function permitting to extend a polynomial in $\mathfrak{R}_{q,N}$ to $\mathfrak{R}_{q,\eta N}$.

Lemma 3.2. *Let $\eta \in \mathbb{Z}$ be a power of two and let the function ι such that:*

$$\begin{aligned} \iota : \mathfrak{R}_{q,N} &\to \mathfrak{R}_{q,\eta N} \\ P(X) = \sum_{i=0}^{N-1} p_i \cdot X^i &\longmapsto P_{\mathsf{ext}}(X) = \sum_{i=0}^{N-1} p_i \cdot X^{\eta i}. \end{aligned}$$

Then, ι is a ring homomorphism.

Proof *(Lemma 3.2). Let us show that ι is a ring homomorphism. We have $\iota(1) = 1$. Let $P(X) = \sum_{i=0}^{N-1} a_i X^i$ and $Q(X) = \sum_{i=0}^{N-1} b_i X^i$ both in $\mathfrak{R}_{q,N}$. Let $P_{\mathsf{ext}} = \iota(P(X)) = \sum_{i=0}^{N-1} p_i X^{\eta i}$ and $Q_{\mathsf{ext}} = \iota(Q(X)) = \sum_{i=0}^{N-1} q_i X^{\eta i}$ both in $\mathfrak{R}_{q,\eta N}$.*

The additive morphism is verified: $\iota(P(X) + Q(X)) = \iota(\sum_{i=0}^{N-1}(p_i + q_i)X^i) = \sum_{i=0}^{N-1}(p_i + q_i)X^{\eta i} = \iota(P(X)) + \iota(Q(X))$. The multiplication morphism is verified: indeed, as a polynomial is a sum of monomials and the morphism is verified for addition, we only need to study the multiplication of two monomials. So for i and j both in $[0, N)$, we have: $\iota\left(X^i \cdot X^j \mod X^N + 1\right) = \iota\left(X^{i+j} \mod X^N + 1\right) = X^{(i+j)\eta} \mod X^{\eta N} + 1 = X^{i\eta} X^{j\eta} \mod X^{\eta N} + 1 = \iota\left(X^i \mod X^N + 1\right) \iota\left(X^j \mod X^N + 1\right)$. □

By applying the ι function to each polynomial of a GLWE ciphertext in $\mathfrak{R}_{q,N}^{k+1}$ (resp., a GGSW ciphertext in $\mathfrak{R}_{q,N}^{(k+1)\ell \times (k+1)}$) under a secret key $\boldsymbol{S}$, we obtain an extended GLWE ciphertext in $\mathfrak{R}_{q,\eta N}^{k+1}$ (resp., an extended GGSW ciphertext in $\mathfrak{R}_{q,\eta N}^{(k+1)\ell \times (k+1)}$) under an extended secret key $\boldsymbol{S}_{\mathsf{ext}} = \iota(\boldsymbol{S})$.

$$\begin{aligned} \iota\left(\mathrm{GLWE}_{\boldsymbol{S}}(\Delta M)\right) &= \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{ext}}}(\Delta M_{\mathsf{ext}}) \\ \iota\left(\mathrm{GGSW}_{\boldsymbol{S}}^{\mathcal{B},\ell}(M')\right) &= \mathrm{GGSW}_{\boldsymbol{S}_{\mathsf{ext}}}^{\mathcal{B},\ell}(M'_{\mathsf{ext}}). \end{aligned}$$

Then the external product is naturally compliant with the extended ciphertexts:

$$\mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{ext}}}(\Delta M_{\mathsf{ext}}) \boxdot \mathrm{GGSW}_{\boldsymbol{S}_{\mathsf{ext}}}^{\mathcal{B},\ell}(M'_{\mathsf{ext}}) = \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{ext}}}(\Delta M_{\mathsf{ext}} \cdot M'_{\mathsf{ext}}).$$

Lemma 3.3. *Let $\mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{ext}}}(\Delta M) \in \mathfrak{R}_{q,\eta N}^{k+1}$ an* GLWE *ciphertext encrypting the message $M \in \mathfrak{R}_{p,N}$ under the extended secret key $\boldsymbol{S}_{\mathsf{ext}} \leftarrow \iota(\boldsymbol{S})$, with $\boldsymbol{S} \in \mathfrak{R}_{q,\eta N}^{k}$. Let $\mathrm{GGSW}_{\boldsymbol{S}_{\mathsf{ext}}}^{\mathcal{B},\ell}(M'_{\mathsf{ext}}) \in \mathfrak{R}_{q,\eta N}^{(k+1)\ell \times (k+1)}$ be and extended* GGSW *ciphertext encrypting an extended message $M'_{\mathsf{ext}} \in \mathfrak{R}_{p,\eta N}$ under the same extended secret key $\boldsymbol{S}_{\mathsf{ext}}$. Then we have $\mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{ext}}}(\Delta M) \boxdot \mathrm{GGSW}_{\boldsymbol{S}_{\mathsf{ext}}}^{\mathcal{B},\ell}(M'_{\mathsf{ext}}) = \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{ext}}}(\Delta M \cdot M'_{\mathsf{ext}})$.*

Proof *(Lemma 3.3). Let $M = \sum_{i=0}^{\eta N-1} m_i X^i = \sum_{i=0}^{\eta-1} \sum_{j=0}^{N-1} m_{iN+j} X^{iN+j} \in \mathfrak{R}_{q,\eta N}$. We have*

$\sum_{i=0}^{\eta-1} \iota\left(\mathrm{GLWE}_{\boldsymbol{S}}\left(\sum_{j=0}^{N-1} \Delta m_{iN+j} X^j\right)\right) \cdot X^i = \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{ext}}}(\Delta M)$. *Then, we have:*

$$\begin{aligned}
&\sum_{i=0}^{\eta-1} \iota\left(\mathrm{GLWE}_{\boldsymbol{S}}\left(\sum_{j=0}^{N-1} \Delta m_{iN+j} X^j\right)\right) \boxdot \mathrm{GGSW}_{\boldsymbol{S}_{\mathsf{ext}}}^{\mathcal{B},\ell}\left(M'_{\mathsf{ext}}\right) \cdot X^i \\
=&\sum_{i=0}^{\eta-1}\left(\mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{ext}}}\left(\sum_{j=0}^{N-1} \Delta m_{iN+j} X^{\eta j} \cdot M'_{\mathsf{ext}}\right)\right) \cdot X^i \\
=&\sum_{i=0}^{\eta-1}\left(\mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{ext}}}\left(\sum_{j=0}^{N-1} \Delta m_{iN+j} X^{\eta j} \cdot M'_{\mathsf{ext}} \cdot X^i\right)\right) = \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{ext}}}\left(\Delta M \cdot M'_{\mathsf{ext}}\right).
\end{aligned}$$

□

Lemma 3.3 shows that the message in the extended GLWE ciphertext does not need to be extended through the ι function, as long as the secret key of the GLWE ciphertext is $\boldsymbol{S}_{\mathrm{ext}}$. By encrypting a lookup table in $\mathfrak{R}_{q,\eta N}$ into a trivial extended GLWE ciphertext, the **EBS** can be computed as in the original **PBS** algorithm. The **EBS** is described in Algorithm 17.

Algorithm 17: $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathbf{EBS}(\mathsf{ct}_{\mathsf{in}}, \mathsf{LUT}_f, \mathsf{BSK})$

Context:
- $\boldsymbol{s} = [s_0, \ldots, s_{n-1}] \subseteq \mathbb{Z}_q^n$
- $\mathsf{BSK}_i \in \mathrm{GGSW}_{\boldsymbol{S}}^{\mathcal{B},\ell}(s_i) \subseteq \mathfrak{R}_{q,N}^{(k+1)\ell\times(k+1)}$
- $\mathsf{BSK}_{i,\mathsf{ext}} \in \mathrm{GGSW}_{\boldsymbol{S}_{\mathsf{ext}}}^{\mathcal{B},\ell}(s_i) \subseteq \mathfrak{R}_{q,\eta N}^{(k+1)\ell\times(k+1)}$
- $\iota : \mathfrak{R}_{q,N} \to \mathfrak{R}_{q,\eta N}$, as defined in Lemma 3.2.
- $\boldsymbol{S}_{\mathsf{ext}}$ is the extended secret key.$(\boldsymbol{S}_{\mathsf{ext}} \leftarrow \iota(\boldsymbol{S}))$

Input:
- $\mathsf{BSK} = (\mathsf{BSK}_0, \ldots \mathsf{BSK}_{n-1}) \in \left[\mathrm{GGSW}_{\boldsymbol{S}}^{\mathcal{B},\ell}\right]^n \subseteq \left[\mathfrak{R}_{q,N}^{(k+1)\ell\times(k+1)}\right]^n$
- $\mathsf{ct}_m = (a_0, \ldots, a_{n-1}, b) \in \mathrm{LWE}_{\boldsymbol{s}} \subseteq \mathbb{Z}_q^{n+1}$
- $\mathsf{LUT}_f \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{ext}}} \subseteq \mathfrak{R}_{q,\eta N}^{(k+1)\ell\times(k+1)}$

Output: $\mathsf{ct}_{f(m)} = \left(a_0^{\mathsf{out}}, \ldots, a_{kN-1}^{\mathsf{out}}, b\right) \in \mathrm{LWE}_{\boldsymbol{s}_{\mathsf{out}}} \subseteq \mathbb{Z}_q^{kN+1}$

```
ct_MS = (ã_0, ..., ã_{n-1}, b̃) ← MS(ct_m, η·2N) ;          /* Algorithm6 */
CT_BR ← LUT_f · X^{b̃}
for i in [0, n−1] do
    BSK_{i,ext} ← ι(BSK_i)
    CT_BR ← CMux(CT_BR, CT_BR · X^{ã_i}, BSK_{i,ext}) ;     /* Algorithm8 */
return ct_{f(m)} ← SampleExtract(CT_BR) ;                   /* Algorithm10 */
```

Theorem 3.1 (Extended PBS (Algorithm 17))**.** *Performing an* **EBS** *taking as input a ciphertext* $\mathsf{ct}_m \in \mathbb{Z}_q^n$*, a bootstrapping key* $\mathsf{BSK} = (\mathsf{BSK}_0, \ldots \mathsf{BSK}_{n-1}) \in \left[\mathrm{GGSW}_{\boldsymbol{S}}^{\mathcal{B},\ell}\right]^n \subseteq \left[\mathfrak{R}_{q,N}^{(k+1)\ell\times(k+1)}\right]^n$ *and a lookup table* $\mathsf{LUT}_f \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{ext}}} \subseteq \mathfrak{R}_{q,\eta N}^{(k+1)\ell\times(k+1)}$ *output a ciphertext* $\mathsf{ct}_{f(m)} \in \mathrm{LWE}_{\boldsymbol{s}_{\mathsf{out}}} \subseteq \mathbb{Z}_q^{kN+1}$ *with an output noise estimated by the formula to* $\mathsf{Var}(\mathbf{EBS}) = \mathsf{Var}(\mathbf{PBS})$.

The cost of Algorithm 17 is: $\mathsf{Cost}_{\mathbf{EBS}}^{\ell,k,\eta,N,n,q} = \mathsf{Cost}_{\mathbf{PBS}}^{\ell,k,\eta N,n,q}$.

Proof *(Theorem 3.1). The proof of Lemma 3.1 is similar to the proof of the classical bootstrapping Theorem 2.15. Indeed Lemma 3.3 show that the external product can be done in an extended context, then we can perform the blind rotation. All other algorithms remain unchanged between the classical and the extended context.* □

In the case of classical PBS, the size of the public material is directly linked to the size of the lookup table, which is itself determined by the size of the message being evaluated. The larger the message values we want to evaluate, the larger the lookup table and the public material need to be.

With the technique introduced in this section, the size of the public material can be reduced while still allowing the evaluation of large lookup tables. This **EBS** offers another degree of freedom when choosing the parameters and results in better noise propagation. Finally, this technique permits to address Limitation 5.

3.2 Parallelized Bootstrapping

Bootstrapping is not naturally parallelizable due to the chain of CMux operations executed during the blind rotation algorithm (Algorithm 9), as highlighted in Limitation 2. In the literature, several works such as [BMMP18, LLW+24, ZYL+18, JP22, LY23] propose techniques to address this limitation.

We begin by presenting the second improvement introduced in [LY23]. Building on their first contribution, which permits computations over extended polynomials, the second improvement allows these extended polynomials to be split into smaller polynomials, enabling parallelization of operations across them. This technique makes it possible to parallelize the computation of a single CMux by distributing the work over smaller-degree polynomials. This technique is presented in the Subsection 3.2.1

The second technique, described in [BMMP18, LLW+24], is straightforward. It involves expanding the CMux chain such that multiple CMux operations can be executed in parallel, each with the same individual cost. This technique is detailed and applied later in Chapter 4.

Finally, the third technique, proposed in [ZYL+18, JP22], consists of creating a special secret key such that this new secret key permits to obtain the result of several CMuxes with a single external product (Algorithm 7). This third technique is known as Multi Bit bootstrapping and is presented in the Subsection 3.2.2.

The drawbacks of the second and the third techniques are the exponential growth of the secret key size and a reduced effectiveness in noise management.

3.2.1 Parallelized Extended Bootstrapping

After defining the function ι, the authors of [LY23] introduced another function, τ, which allows splitting a polynomial message encrypted in an extended GLWE ciphertext into several smaller messages encrypted in smaller GLWE ciphertexts. This enables performing independent external products on these smaller ciphertexts, thereby enabling the parallelization of the external product (see Limitation 2).

Let us begin by defining the isomorphism τ, which allows splitting a polynomial in $\mathfrak{R}_{q,\eta N}$ into η polynomials in $\mathfrak{R}_{q,N}$.

Lemma 3.4. *Let the function τ be defined as:*

$$\tau : \mathfrak{R}_{q,\eta N} \to [\mathfrak{R}_{q,N}]^{\eta}$$
$$P(X) = \sum_{i=0}^{\eta N-1} p_i \cdot X^i \longmapsto [P_0(X), \ldots, P_{\eta-1}(X)] .$$

With $P_j(X) = \sum_{i=0}^{N-1} p_{i\eta+j} \cdot X^i$ for $j \in [0, \eta)$. Then τ is an isomorphism.

Proof *(Lemma 3.4). Let f be a function such that:*

$$f : [\mathfrak{R}_{q,N}]^{\eta} \to \mathfrak{R}_{q,\eta N}$$
$$[P_0(X), \ldots, P_{\eta-1}(X)] \longmapsto P(X) = \sum_{i=0}^{\eta N-1} p_i \cdot X^i.$$

With $P_j(X) = \sum_{i=0}^{N-1} p_{i\eta+j} \cdot X^i$ for $j \in [0, \eta)$. Then, we directly have $f(\tau(P(X))) = P(X)$ and $\tau\left(f\left([P_0(X), \ldots, P_{\eta-1}(X)]\right)\right) = [P_0(X), \ldots, P_{\eta-1}(X)]$. So f corresponds to τ^{-1}, thus τ is an isomorphism. □

By applying the function τ to each polynomial composing a GLWE ciphertext in $\mathfrak{R}_{q,\eta N}^{k+1}$, encrypted under an extended secret key $\boldsymbol{S}_{\mathsf{ext}} = [S_{\mathsf{ext},0}, \ldots, S_{\mathsf{ext},k-1}]$ (with $S_{\mathsf{ext},i} = \sum_{j=0}^{N-1} s_{j,i} X^{\eta j}$), we then obtain η GLWE ciphertexts in $\mathfrak{R}_{q,N}^{k+1}$ encrypted under a secret key $\boldsymbol{S} = [S_0, \ldots, S_{k-1}]$ (with $S_i = \sum_{j=0}^{N-1} s_{j,i} X^j$). Performing an external product between an extended GGSW ciphertext encrypting a bit b and an GLWE ciphertext encrypted under an extended secret key can be computed on η GLWE by using the function τ. Indeed we have:

$$\begin{aligned} & \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{ext}}}(\Delta M) \boxdot \mathrm{GGSW}_{\boldsymbol{S}_{\mathsf{ext}}}^{\mathcal{B},\ell}(b) \\ = \quad & \tau^{-1}\left[\mathrm{GLWE}_{\boldsymbol{S}}(\Delta M_0) \boxdot \mathrm{GGSW}_{\boldsymbol{S}}^{\mathcal{B},\ell}(b), \ldots, \mathrm{GLWE}_{\boldsymbol{S}}(\Delta M_{\eta-1}) \boxdot \mathrm{GGSW}_{\boldsymbol{S}}^{\mathcal{B},\ell}(b)\right]. \end{aligned}$$

with $\mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{ext}}}(\Delta M) \in \mathfrak{R}_{q,\eta N}^{k+1}$ and $\tau(M) = (M_0, \ldots, M_{\eta-1})$ with $M \in \mathfrak{R}_{p,\eta N}$ and $M_i \in \mathfrak{R}_{p,N}$ for $i \in [0, N)$.

In what follows, we denote the extended lookup table by $\mathsf{LUT} \in \mathfrak{R}_{q,\eta N}^{k+1}$. The smaller lookup table obtained by $\tau(\mathsf{LUT})$ are denoted $\mathsf{lut}_i \in \mathfrak{R}_{q,N}^{k+1}$ for $i \in [0, \eta)$, $\tau(\mathsf{LUT}) = [\mathsf{lut}_0, \ldots, \mathsf{lut}_{\eta-1}]$.

In [LY23], during the blind rotation, at each CMuxes, authors proposed using the function τ to split the lookup table in $\mathfrak{R}_{q,\eta N}^{k+1}$ into η lookup tables in $\mathfrak{R}_{q,\eta N}^{k+1}$. Then, they performed several small CMuxes in parallel and used the function τ^{-1} to reconstruct a lookup table in $\mathfrak{R}_{q,\eta N}^{k+1}$ to perform the rotation of the entire lookup table. The **Parallelized-EBS** is described in Algorithm 18.

Algorithm 18: $\mathsf{ct}_{\mathsf{out}} \leftarrow$ **Parallelized-EBS**$(\mathsf{ct}_{\mathsf{in}}, \mathsf{LUT}_f, \mathsf{BSK})$

Context:
- $\boldsymbol{s} = [s_0, \ldots, s_{n-1}] \subseteq \mathbb{Z}_q^n$
- $\mathsf{BSK}_i \in \mathrm{GGSW}_{\boldsymbol{S}}^{\mathcal{B},\ell}(s_i) \subseteq \mathfrak{R}_{q,N}^{(k+1)\ell \times (k+1)}$
- $\eta = 2^k$: The extended factor
- $\iota : \mathfrak{R}_{q,N} \to \mathfrak{R}_{q,\eta N}$, as defined in Lemma 3.2.
- $\tau : \mathfrak{R}_{q,\eta N} \to [\mathfrak{R}_{q,N}]^{\eta}$, as defined in Lemma 3.4.
- $\boldsymbol{S}_{\mathsf{ext}}$ is the extended secret key.$(\boldsymbol{S}_{\mathsf{ext}} \leftarrow \iota(\boldsymbol{S}))$

Input:
- $\mathsf{BSK} = (\mathsf{BSK}_0, \ldots, \mathsf{BSK}_{n-1}) \in \left[\mathrm{GGSW}_{\boldsymbol{S}}^{\mathcal{B},\ell}\right]^n \subseteq \left[\mathfrak{R}_{q,N}^{(k+1)\ell \times (k+1)}\right]^n$
- $\mathsf{ct}_m = (a_0, \ldots, a_{n-1}, b) \in \mathrm{LWE}_{\boldsymbol{s}} \subseteq \mathbb{Z}_q^{n+1}$
- $\mathsf{LUT}_f \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{ext}}} \subseteq \mathfrak{R}_{q,\eta N}^{(k+1)\ell \times (k+1)}$

Output: $\mathsf{ct}_{f(m)} = \left(a_0^{\mathsf{out}}, \ldots, a_{kN-1}^{\mathsf{out}}, b\right) \in \mathrm{LWE}_{\boldsymbol{s}_{\mathsf{out}}} \subseteq \mathbb{Z}_q^{kN+1}$

$\mathsf{ct}_{\mathbf{MS}} = \left(\tilde{a}_0, \ldots, \tilde{a}_{n-1}, \tilde{b}\right) \leftarrow \mathbf{MS}(\mathsf{ct}_m, \eta \cdot 2N)$; /* Algorithm 6 */

$\mathsf{CT}_{\mathbf{BR}} \leftarrow \mathsf{LUT}_f \cdot X^{\tilde{b}}$

$\left[\mathsf{CT}_{\mathbf{BR}}^0, \ldots, \mathsf{CT}_{\mathbf{BR}}^{\eta-1}\right] \leftarrow \tau\left(\mathsf{CT}_{\mathbf{BR}}\right)$

for $i \in [0, n-1]$ **do**

  $\left[\mathsf{CT}_{\mathbf{BR},\mathsf{tmp}}^0, \ldots, \mathsf{CT}_{\mathbf{BR},\mathsf{tmp}}^{\eta-1}\right] \leftarrow \tau\left(\mathsf{CT}_{\mathbf{BR}} \cdot X^{\tilde{a}_i}\right)$

  for j *in* $[0, \eta-1]$ **do**

    /* Each step of this loop can be done in parallel */

    $\mathsf{CT}_{\mathbf{BR}}^j \leftarrow \mathsf{CMux}(\mathsf{CT}_{\mathbf{BR}}^j, \mathsf{CT}_{\mathbf{BR},\mathsf{tmp}}^j, \mathsf{BSK}_i)$; /* Algorithm 8 */

  $\mathsf{CT}_{\mathbf{BR}} = \tau^{-1}\left(\mathsf{CT}_{\mathbf{BR}}^0, \ldots, \mathsf{CT}_{\mathbf{BR}}^{\eta-1}\right)$

return $\mathsf{ct}_{f(m)} \leftarrow$ **SampleExtract**$(\mathsf{CT}_{\mathbf{BR}}^0)$; /* Algorithm 10 */

Remark 3.3. The noise distribution of this algorithm is similar to the one from the **PBS** described in Theorem 2.15. The difference is about the polynomial degree: for a fixed precision, when $\eta > 1$, a classical **PBS** operates with polynomials of degree $N\eta$, whereas the **Parallelized-EBS** operates with polynomials of degree N. Overall, the noise added during the **Parallelized-EBS** is equivalent

to the noise added by a classical bootstrapping with a polynomial size equal to N, whereas to obtain the same result, a classical bootstrapping needs to work with a polynomial size equal to ηN. The cost of Algorithm 17 is:

$$\mathsf{Cost}^{\ell,k,\eta,N,n,q}_{\textbf{Parallelized-EBS}} = \mathsf{Cost}^{n,q,2\eta N}_{\mathsf{ModulusSwitch}} + \eta \cdot \mathsf{Cost}^{\ell,k,N,n}_{\mathsf{BlindRotation}} + \mathsf{Cost}^{k,N}_{\mathsf{SampleExtract}}.$$

Remark 3.4. In [LY23], authors utilize this method for the parallelized version and perform all the computations in the extended ring for the sequential version. However, this trick can also be applied in a sequential context leading to a better or equal version than the extended one. In the extended bootstrapping with $\mathfrak{R}_{q,\eta N}$, the cost of one polynomial product is approximately $N\eta \log_2(N\eta)$ (i.e., using an FFT-based algorithm). In the parallelized version, to perform the same operation in $\mathfrak{R}_{q,N}$, we need to perform η polynomial products, each with an individual cost of $N \log_2(N)$. We then refer to the **EBS** (Algorithm 18) for either the parallelized or the sequential versions.

In [LY23], they mention that using τ and τ^{-1} at each step is not mandatory but they did not explicit how to perform this rotation. In the following, we explicit how to compute the rotation without using τ and τ^{-1} at each step of the blind rotation, by performing inner rotations on each smaller luts and updating their index accordingly. We make this rotation explicit in the following lemma:

Lemma 3.5. *Let τ the function defined in Lemma 3.4. Let $P(X) = \sum_{i=0}^{\eta N-1} p_i X^i$ be a polynomial in $\mathfrak{R}_{q,\eta N}$ such that $\tau(P(X)) = [P_0(X), \dots, P_{\eta-1}(X)]$ with $P_i(X) = \sum_{j=0}^{N-1} p_{j\eta+i} X^j$ a polynomial in $\mathfrak{R}_{q,N}$ for $i \in [0, \eta)$.*

For any $\kappa \in \mathbb{Z}$ we have $\tau(P(X) \cdot X^\kappa) = [P'_0(X), \dots, P'_{\eta-1}(X)]$ with $P'_j(X) = P_{[(j-\kappa)]_\eta}(X) \cdot X^{\lceil \frac{\kappa-j}{\eta} \rceil}$.

Proof *(Lemma 3.5).* *Let $P(X) = \sum_{i=0}^{\eta N-1} p_i X^i \in \mathfrak{R}_{q,\eta N}$ such that $\tau(P(X)) = [P_0(X), \dots, P_{\eta-1}(X)]$ with $P_i(X) = \sum_{j=0}^{N-1} p_{j\eta+i} X^j \in \mathfrak{R}_{q,N}$. We give a proof by induction over the rotation κ.*

Base case: $\kappa = 1$ We have the following equation:

$$\begin{aligned}
P(X) \cdot X &= \sum_{i=1}^{\eta N-1} p_{i-1} X^i + p_{\eta N-1} X^{\eta N} \\
P(X) \cdot X &= \sum_{j=0}^{\eta-1} \sum_{i=0}^{N-1} p_{i\eta-1+j} X^{i\eta+j} - p_{\eta N-1} \sum_{i=0}^{N-1} X^{\eta i} \mod \mathfrak{R}_{q,\eta N} \text{ (with } p_{-1} = 0) \\
&= -p_{\eta N-1} + \sum_{i=1}^{N-1} p_{i\eta-1} X^{\eta i} + \sum_{j=1}^{\eta-1} \sum_{i=0}^{N-1} p_{i\eta-1+j} X^{i\eta+j} \mod \mathfrak{R}_{q,\eta N}.
\end{aligned}$$

By the previous equation, we have $\tau(P(X) \cdot X) = [P_{\eta-1}(X) \cdot X, P_0(X), \dots, P_{\eta-2}(X)] = [P'_0(X), \dots, P'_{\eta-1}(X)]$ which correspond to $P'_j(X) = P_{[(j-1)]_\eta}(X) \cdot X^{\lceil \frac{1-j}{\eta} \rceil}$

Inductive hypothesis: We assume that for a given κ, the hypothesis is true for the previous step. So we have $\tau(P(X) \cdot X^\kappa) = [P'_0(X), \dots, P'_{\eta-1}(X)]$ with $P'_j(X) = P_{[(j-\kappa)]_\eta}(X) \cdot X^{\lceil \frac{\kappa-j}{\eta} \rceil}$ for $i \in [0, \eta)$.

Inductive step: For $\kappa + 1$, we obtain:

$$\begin{aligned}
\tau(P(X) \cdot X^{\kappa+1}) = \tau(P(X) \cdot X^\kappa \cdot X) &= [P'_{\eta-1}(X) \cdot X, P'_0(X), \dots, P'_{\eta-2}(X)] \\
&= [P''_0(X), \dots, P''_{\eta-1}(X)].
\end{aligned}$$

With

$$\begin{aligned}
P''_0(X) = P'_{[\eta-1]_\eta}(X) \cdot X &= P_{[\eta-(\kappa+1)]_\eta}(X) \cdot X^{\lceil \frac{\kappa-(\eta-1)}{\eta} \rceil} \cdot X \\
&= P_{[\eta-(\kappa+1)]_\eta}(X) \cdot X^{\lceil \frac{\kappa-(\eta-1)+\eta}{\eta} \rceil} = P_{[-(\kappa+1)]_\eta}(X) \cdot X^{\lceil \frac{\kappa+1}{\eta} \rceil}.
\end{aligned}$$

and, for $j \in [1, \eta - 1)$*:*

$$P''_j(X) = P'_{[j-1]_\eta}(X) = P_{[j-1-\kappa]_\eta}(X) \cdot X^{\lceil \frac{\kappa-(j-1)}{\eta} \rceil} = P_{[j-(\kappa+1)]_\eta}(X) \cdot X^{\lceil \frac{\kappa+1-j}{\eta} \rceil}.$$

□

Remark 3.5. A consequence of this lemma is that applying a rotation X^κ, in $\mathsf{LUT} \in \mathfrak{R}_{q,\eta N}$, can be expressed as changing the index of $\mathsf{lut} \in \mathfrak{R}_{q,N}$ along with an inner rotation within each lut. Furthermore, we show that two lookup tables, lut_i and lut_j, with $i \neq j$, will never interact during the rotation.

To perform the **Parallelized-EBS** without using τ and τ^{-1}, we simply need to remove Lines 5 and 8 from Algorithm 18, and replace Line 7 with:

$$\mathsf{CT}^j_{\mathbf{BR}} \leftarrow \mathsf{CMux}\left(\mathsf{CT}^j_{\mathbf{BR}}, \mathsf{CT}^{[j-a_i]_\eta}_{\mathbf{BR}} \cdot X^{\lceil \frac{a_i - j}{\eta} \rceil}, \mathsf{BSK}_i\right).$$

3.2.2 Multi-Bit Bootstrapping

The technique proposed in [ZYL+18, JP22] focuses exclusively on parallelizing the blind rotation (Algorithm 9), which is the most expensive part of the bootstrapping. In this section, we focus solely on the blind rotation. To improve the **PBS** (Algorithm 11), it is sufficient to replace the usual blind rotation with the multi bit blind rotation.

The multi bit blind rotation performs blind rotation by grouping several external products into a single external product, producing the same result as computing multiple CMux operations in the standard blind rotation. The number of grouped elements is referred to as the grouping factor and is denoted by gf. To enable this optimization, a new structure for the bootstrapping key (BSK) is required, called the multi-bit bootstrapping key ($\mathsf{MB\text{-}BSK}$). The size of this key grows exponentially with the grouping factor gf.

Similar to the standard variant, the multi-bit approach allows to blindly rotate the lookup table by $X^{b-\sum a_i \cdot s_i}$. However, instead of applying the rotation sequentially with a CMux at each step, the multi-bit scheme applies a block of gf rotation at once through an alternate update step, as described in Algorithm 19.

For instance, when $\mathsf{gf} = 2$, the multi-bit update step is computed as follows:

$$\mathsf{CT}_{\mathsf{out}} \leftarrow \left(X^0 \cdot \overline{\overline{\mathsf{CT}}}_{i,\emptyset} + X^{a_{2i+0}} \cdot \overline{\overline{\mathsf{CT}}}_{i,\{0\}} + X^{a_{2i+1}} \cdot \overline{\overline{\mathsf{CT}}}_{i,\{1\}} + X^{a_{2i+0}+a_{2i+1}} \cdot \overline{\overline{\mathsf{CT}}}_{i,\{0,1\}}\right) \boxdot \mathsf{CT}_{\mathsf{out}}.$$

where $\overline{\overline{\mathsf{CT}}}_{i,\emptyset} \in \mathrm{GGSW}^{\mathcal{R},\ell}_{\boldsymbol{S}}\big((1-s_{2i+0})\cdot(1-s_{2i+1})\big)$, $\overline{\overline{\mathsf{CT}}}_{i,\{0\}} \in \mathrm{GGSW}^{\mathcal{R},\ell}_{\boldsymbol{S}}\big(s_{2i+0}\cdot(1-s_{2i+1})\big)$, $\overline{\overline{\mathsf{CT}}}_{i,\{1\}} \in \mathrm{GGSW}^{\mathcal{R},\ell}_{\boldsymbol{S}}\big((1 - s_{2i+0}) \cdot s_{2i+1}\big)$ and $\overline{\overline{\mathsf{CT}}}_{i,\{0,1\}} \in \mathrm{GGSW}^{\mathcal{R},\ell}_{\boldsymbol{S}}\big(s_{2i+0} \cdot s_{2i+1}\big)$.

In general, for a fixed i, there is exactly one element in the set $\left\{\overline{\overline{\mathsf{CT}}}_{i,J}\right\}_{J\subseteq\{0,\ldots,\mathsf{gf}-1\}}$ that encrypts 1. We denote by $\bar{J}$ the unique subset such that $\prod_{j\in\bar{J}} s_{i\cdot\mathsf{gf}+j} \prod_{j\notin\bar{J}}(1 - s_{i\cdot\mathsf{gf}+j}) = 1$, i.e., $\bar{J} = \{j \mid s_{i\cdot\mathsf{gf}+j} = 1\}$, while the other ciphertexts encrypt 0. It follows that the new ciphertext $\mathsf{CT}_{\mathsf{out}}$ encrypts the plaintext of the old $\mathsf{CT}_{\mathsf{out}}$ multiplied by $X^{\sum_{j\in\bar{J}} a_{i\cdot\mathsf{gf}+j}} = X^{\sum_{j=0}^{\mathsf{gf}-1} a_{i\cdot\mathsf{gf}+j} s_{i\cdot\mathsf{gf}+j}}$.

Remark 3.6. The cost of the multi bit blind rotation (Algorithm 19) can be approximated by:

$$\mathsf{Cost}^{n,\mathsf{gf},\ell,k,N}_{\mathsf{MB-BR}} = y \cdot \mathsf{Cost}^{\ell,k,N}_{\mathsf{ExternalProduct}}.$$

Where gf is the grouping factor and where $n = \mathsf{gf} \cdot y$ with $y \in \mathbb{N}$.

3.3 Without Padding Bootstrapping

In Section 2.4, we seen that a padding bit is needed to correctly evaluate non-negacyclic function (see Limitation 1). A **WoP-PBS**, i.e., a **PBS** which does not require a bit of padding, is a

Algorithm 19: $\mathsf{ct_{out}} \leftarrow \mathsf{MultiBit\text{-}BlindRotation}(\mathsf{ct_{in}}, \mathsf{MB\text{-}BSK}, \mathsf{LUT}_f)$

Context:
$$\begin{cases} \mathsf{gf} \in \mathbb{N}\text{: grouping factor, typically a small integer} \in \{2,3,4\} \\ \qquad \text{where } n = \mathsf{gf} \cdot y \text{ for some } y \in \mathbb{N} \\ \boldsymbol{s} = (s_0, \ldots, s_{n-1}) \in \mathbb{Z}_q^n\text{: the LWE input secret key with } s_i \hookleftarrow \mathcal{U}(\{0,1\}) \\ \boldsymbol{S} = (S_0, \ldots, S_{k-1}) \in \mathfrak{R}_{q,N}^k\text{: the output GLWE secret key} \\ \mathsf{LUT}_f \in \mathfrak{R}_{q,N}\text{: a lookup table for a function } x \mapsto f(x) \\ \Delta m + e \in \mathbb{Z}_{2N}\text{: The encoded message of } \mathsf{ct_{in}} \end{cases}$$

Input:
$$\begin{cases} \mathsf{ct_{in}} = (a_0, \ldots, a_{n-1}, b) \in \mathrm{LWE}_{\boldsymbol{s}}(m) \subseteq \mathbb{Z}_{2N}^{n+1} \\ \mathsf{MB\text{-}BSK} = \left\{ \left\{ \overline{\overline{\mathsf{CT}}}_{i,J} \in \mathrm{GGSW}_{\boldsymbol{S}}^{\mathcal{B},\ell}(\prod_{j\in J} s_{i\cdot\mathsf{gf}+j} \prod_{j\notin J}(1 - s_{i\cdot\mathsf{gf}+j})) \right\}_{J\subseteq\{0,\ldots,\mathsf{gf}-1\}} \right\}_{i=0}^{y-1}\text{:} \\ \qquad \text{a multi-bit bootstrapping key from } \boldsymbol{s} \text{ to } \boldsymbol{S} \\ \mathsf{CT}_f \in \mathrm{GLWE}_{\boldsymbol{S}}(M_f) \subseteq \mathfrak{R}_{q,N}^{k+1}\text{: an encrypted (possibly trivially) lookup table} \end{cases}$$

Output: $\mathsf{CT_{out}} \in \mathrm{GLWE}_{\boldsymbol{S}}\left(M_f \cdot X^{-\Delta m - e}\right)$, where e is the noise in $\mathsf{ct_{in}}$

$\mathsf{CT_{out}} \leftarrow \mathsf{CT}_f \cdot X^{-b}$
for $0 \le i \le y-1$ **do**
  $\mathsf{CT_{out}} \leftarrow \left(\sum_{J\subseteq\{0,\ldots,\mathsf{gf}-1\}} X^{\sum_{j\in J} a_{i\cdot\mathsf{gf}+j}} \cdot \overline{\overline{\mathsf{CT}}}_{i,J}\right) \boxdot \mathsf{CT_{out}}$
return $\mathsf{CT_{out}}$

method that was introduced for the first time in [CLOT21]. In this section, we first present the first **WoP-PBS** introduced in [CLOT21] followed by the **WoP-PBS** from [KS21]. Then we introduce the **WoP-PBS** presented in [LMP21, YXS+21].

[CLOT21] WoP-PBS. The **WoP-PBS** algorithm introduced in [CLOT21] takes as input an LWE ciphertext encrypting a message m. It consists of first performing a **PBS** to extract the sign (i.e., the most significant bit), followed by another **PBS** to evaluate the target function f. The two results of the **PBS** algorithm are then multiplied. As a result, the **WoP-PBS** procedure outputs an LWE ciphertext encrypting $f(m)$.

Before detailing the [CLOT21] **WoP-PBS**, we first to introduce how we perform the multiplication between two LWE ciphertexts without using the external product algorithm (Algorithm 7). Although this technique is more costly than the external product (see External Product in Subsection 4) and requires a relinearization key, it does not rely on GGSW ciphertexts. This alternative method (detailed in Algorithm 20) consists of performing a tensor product followed by a relinearization, which allows to switch back to the input secret key.

We then present the **WoP-PBS** in Algorithm 21.

Remark 3.7. The cost of the [CLOT21] **WoP-PBS** (Algorithm 21) can be approximated by:

$$\mathsf{Cost}_{\mathsf{WoPPBSCLOT21}}^{\ell,k,N,n,q} = 2 \cdot \mathsf{Cost}_{\mathbf{PBS}}^{\ell,k,N,n,q} + \mathsf{Cost}_{\mathbf{KS}}^{\ell,n,k,N} + \mathsf{Cost}_{\mathsf{LWEMult}}^{\ell,n,k,N}.$$

And cost of the LWE Multiplication (Algorithm 20) can be approximated by:

$$\mathsf{Cost}_{\mathsf{LWEMult}}^{\ell,n,k,N} = 2 \cdot \mathsf{Cost}_{\mathbf{KS}}^{\ell,n,k,N} + \mathsf{Cost}_{\mathsf{TensorProduct}}^{k,N} + \mathsf{Cost}_{\mathsf{Relin}}^{\ell,k,N} + \mathsf{Cost}_{\mathsf{SampleExtract}}^{k,N}.$$

Whit

$$\mathsf{Cost}_{\mathsf{TensorProduct}}^{k,N} = 2(k+1)N\mathsf{Cost}_{\mathsf{FFT}} + k\mathsf{Cost}_{\mathsf{iFFT}} + (k+1)^2 N\mathsf{Cost}_{\mathsf{MltFFT}} + kN\mathsf{Cost}_{\mathsf{addFFT}}.$$

and

$$\begin{aligned} \mathsf{Cost}_{\mathsf{Relin}}^{\ell,k,N} = {} & N\ell k\mathsf{Cost}_{\mathbf{Dec}} + k\ell\mathsf{Cost}_{\mathsf{FFT}} + k\ell(k+1)N\mathsf{Cost}_{\mathsf{multFFT}} \\ & + (k\ell - 1)(k+1)N\mathsf{Cost}_{\mathsf{addFFT}} + (k+1)\mathsf{Cost}_{\mathsf{iFFT}}. \end{aligned}$$

Algorithm 20: $\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{LWEMult}(\mathsf{ct}_1, \mathsf{ct}_2, \mathsf{RLK}, \mathsf{KSK})$

Context:
- $\boldsymbol{s} = (s_0, \ldots, s_{n-1})$: the LWE secret key
- $\boldsymbol{S'} = (S'_0, \ldots, S'_{k-1})$: the GLWE secret key
- $\boldsymbol{s'}$ the flatten representation of the GLWE secret key, Definition 11
- $\mathsf{KSK}_i \in \mathrm{GLEV}^{\mathcal{B},\ell}_{\boldsymbol{S}}(-s_i)$: Definition 13
- $\mathsf{RLK}_{i,j} \in \mathrm{GLEV}^{\mathcal{B},\ell}_{\boldsymbol{S}}(S_i \cdot S_j)$: Definition 13

Input:
- $\mathsf{ct}_1 \in \mathrm{LWE}_{\boldsymbol{s}}(m_1) \in \mathbb{Z}_q^{n+1}$
- $\mathsf{ct}_2 \in \mathrm{LWE}_{\boldsymbol{s}}(m_2) \in \mathbb{Z}_q^{n+1}$
- $\mathsf{KSK} = \{\mathsf{KSK}_i\}_{i \in [0, k_{\mathsf{in}}-1]}$: The key switching key from $\boldsymbol{S}_{\mathsf{in}}$ to $\boldsymbol{S}_{\mathsf{out}}$
- $\mathsf{RLK} = \{\mathsf{RLK}_{i,j}\}^{j \in [0,i]}_{i \in [0,k-1]}$: The relinearization key

Output: $\mathsf{CT}_{\mathsf{out}} \in \mathrm{GLWE}_{\boldsymbol{S}}(M \cdot P) \subseteq \mathfrak{R}^{k+1}_{q,N}$

```
/* Algorithm 4                                                    */
```
$\mathsf{CT}_1 = (A_{1,0}, \ldots, A_{1,k-1}, B_1) \in \mathrm{GLWE}_{\boldsymbol{S'}}(m_1) \leftarrow \mathsf{LWEKeySwitch}(\mathsf{ct}_1, \mathsf{KSK})$

```
/* Algorithm 4                                                    */
```
$\mathsf{CT}_2 = (A_{2,0}, \ldots, A_{2,k-1}, B_2) \in \mathrm{GLWE}_{\boldsymbol{S'}}(m_2) \leftarrow \mathsf{LWEKeySwitch}(\mathsf{ct}_1, \mathsf{KSK})$

```
/* Tensor product                                                 */
```
for $i \in [0, k-1]$ **do**
- $T'_i \leftarrow \left\lceil \left\lfloor \frac{[A_{1,i} \cdot A_{2,i}]_Q}{\Delta} \right\rceil \right\rfloor_q$
- $A'_i \leftarrow \left\lceil \left\lfloor \frac{[A_{1,i} \cdot B_2 + A_{2,i} \cdot B_1]_Q}{\Delta} \right\rceil \right\rfloor_q$
- **for** $j \in [0, i]$ **do**
  - $R'_{i,j} \leftarrow \left\lceil \left\lfloor \frac{[A_{1,i} \cdot A_{2,j} + A_{2,i} \cdot A_{1,j}]_Q}{\Delta} \right\rceil \right\rfloor_q$

$B' \leftarrow \left\lceil \left\lfloor \frac{[B_1 \cdot B_2]_Q}{\Delta} \right\rceil \right\rfloor_q$

```
/* Relinearization                                                */
```
$\mathsf{CT} \leftarrow \left(A'_0, \ldots, A'_{k-1}, B'\right) + \sum_{i=0}^{k-1} \langle \overline{CT}_{i,i}, \mathbf{Dec}^{\mathcal{B},\ell}(T'_i) \rangle + \sum_{j=0}^{i} \langle \overline{CT}_{i,j}, \mathbf{Dec}^{\mathcal{B},\ell}(R'_{i,j}) \rangle$

$\mathsf{ct}_{\mathsf{out}} \leftarrow \mathbf{SampleExtract}(\mathsf{CT})$; `/* Algorithm 10 */`

return $\mathsf{ct}_{\mathsf{out}}$

[KS21] WoP-PBS. This second variant of **WoP-PBS** takes as input an LWE ciphertext encrypting a message m. As in the previous algorithm, the first step consists in extracting the sign bit (i.e., the most significant bit). The extracted sign, initially encrypted in an LWE ciphertext, is then transformed into a GGSW ciphertext using circuit bootstrapping (Algorithm 12).

With the resulting GGSW ciphertext, the algorithm selects between two lookup tables, one corresponding to a positive sign and the other to a negative sign, using a CMux (Algorithm 8). Finally, the selected lookup table is evaluated using a classical **PBS** (Algorithm 11). This algorithm is also known as Full Domain Functional Bootstrapping (FDFB).

Remark 3.8. The cost of the [KS21] **WoP-PBS** (Algorithm 22) can be approximated by:

$$\begin{aligned}&\mathsf{Cost}^{\ell_{\mathbf{PBS}},\ell_{\mathbf{CBS}},k,N,n,q}_{\mathsf{WoPPBSKS21}} \\ &= 2 \cdot \left(\mathsf{Cost}^{\ell_{\mathbf{PBS}},k,N,n,q}_{\mathbf{PBS}} + \cdot \mathsf{Cost}^{\ell_{\mathbf{PBS}},n,k,N}_{\mathbf{KS}} + \mathbb{C}_{add}\right) + \mathsf{Cost}^{\ell_{\mathbf{PBS}},\ell_{\mathbf{CBS}},k,N,n,q}_{\mathbf{CBS}} + \mathsf{Cost}^{\ell_{\mathbf{PBS}},k,N}_{\mathsf{CMux}}.\end{aligned}$$

Where $\mathbb{C}_{add}$ denotes the complexity of adding two elements of $\mathbb{Z}_q$.

Algorithm 21: $\mathsf{ct}_{\mathsf{out}} \leftarrow$ **WoP-PBS**[CLOT21]$(\mathsf{ct}_{\mathsf{in}}, \mathsf{LUT}, \mathsf{PUB})$

Context:
- $\boldsymbol{s} = (s_0, \ldots, s_{n-1})$: the LWE secret key
- $\boldsymbol{S'} = (S'_0, \ldots, S'_{k-1})$: the GLWE secret key
- $\Delta m + e \in \mathbb{Z}_q$: The encoded message of $\mathsf{ct}_{\mathsf{in}}$, Definition 8
- LUT_i : Lookup table evaluating the function $x \mapsto f_i(x)$ for $i \in \{0, 1\}$
- $\mathsf{LUT}_{\mathbb{1}}$: Lookup table evaluating the function $x \mapsto 1$

Input:
- $\mathsf{ct}_{\mathsf{in}} = (a_{\mathsf{in},0}, \ldots, a_{\mathsf{in},n-1}, b_{\mathsf{in}}) \in \mathrm{LWE}_{\boldsymbol{s}}(m) \subseteq \mathbb{Z}_{2N}^{n+1}$
- LUT : Lookup Table if the padding bit equals 1
- PUB : public keys required for the entire algorithm ; `/* Remark 2.15 */`

Output: $\mathsf{ct}_{\mathsf{out}} \in \mathrm{LWE}_{\boldsymbol{s}}\left(f\left(\Delta m + e\right)\right) \subseteq \mathfrak{R}_{q,N}^{k+1}$

$\mathsf{ct}_{\mathsf{in}} \leftarrow \mathbf{KS}(\mathsf{ct}_{\mathsf{in}}, \mathsf{KSK})$; `/* Algorithm 4 */`
$\mathsf{ct}_{\mathsf{sign}} \leftarrow \mathbf{PBS}(\mathsf{ct}_{\mathsf{in}}, \mathsf{LUT}_{\mathbb{1}}, \mathsf{BSK})$; `/* Algorithm 11 */`
$\mathsf{ct}_{f_\pm} \leftarrow \mathbf{PBS}(\mathsf{ct}_{\mathsf{in}}, \mathsf{LUT}, \mathsf{BSK})$; `/* Algorithm 11 */`
$\mathsf{ct}_f \leftarrow \mathsf{LWEMult}(\mathsf{ct}_{f_\pm}, \mathsf{ct}_{\mathsf{sign}})$; `/* Algorithm 20 */`
return $\mathsf{ct}_{\mathsf{out}} = \mathsf{ct}_f$

Algorithm 22: $\mathsf{ct}_{\mathsf{out}} \leftarrow$ **WoP-PBS**[KS21]$(\mathsf{ct}_{\mathsf{in}}, \mathsf{PUB}, \mathsf{LUT}_0, \mathsf{LUT}_1)$

Context:
- $\boldsymbol{s} = (s_0, \ldots, s_{n-1})$: the LWE secret key
- $\boldsymbol{S'} = (S'_0, \ldots, S'_{k-1})$: the GLWE secret key
- $\Delta m + e \in \mathbb{Z}_q$: The encoded message of $\mathsf{ct}_{\mathsf{in}}$, Definition 8
- LUT_i : Lookup table evaluating the function $x \mapsto f_i(x)$ for $i \in \{0, 1\}$
- $\mathsf{LUT}_{\mathbb{1}}$: Lookup table evaluating the function $x \mapsto 1$

Input:
- $\mathsf{ct}_{\mathsf{in}} = (a_{\mathsf{in},0}, \ldots, a_{\mathsf{in},n-1}, b_{\mathsf{in}}) \in \mathrm{LWE}_{\boldsymbol{s}}(m) \subseteq \mathbb{Z}_{2N}^{n+1}$
- LUT_1 : Lookup Table if the padding bit equals 1
- LUT_0 : Lookup Table if the padding bit equals 0
- PUB : public keys required for the entire algorithm ; `/* Remark 2.15 */`

Output: $\mathsf{ct}_{\mathsf{out}} \in \mathrm{LWE}_{\boldsymbol{s}}\left(f_{\mathsf{sign}}\left(\Delta m + e\right)\right) \subseteq \mathfrak{R}_{q,N}^{k+1}$

`/* Generalization of Proof 13 (Algorithm 15) to enable the extraction of the most significant bit for message larger than one bit */`
$\mathsf{ct}_{\mathsf{in}} \leftarrow \mathbf{KS}(\mathsf{ct}_{\mathsf{in}}, \mathsf{PUB})$; `/* Algorithmn4 */`
$\mathsf{ct}_{\mathsf{tmp}} \leftarrow \mathsf{ct}_{\mathsf{in}} + (\mathbf{0}, \Delta/2)$
$\mathsf{ct}_{\mathsf{sign}} \leftarrow \mathbf{PBS}\left(\mathsf{ct}_{\mathsf{tmp}}, \mathsf{LUT}_{\mathbb{1}}, \mathsf{PUB}\right)$; `/* Algorithm 11 */`
$\mathsf{ct}_{\mathsf{sign}} \leftarrow \mathsf{ct}_{\mathsf{sign}} + (\mathbf{0}, \Delta/2)$
$\mathsf{ct}_{\mathsf{sign}} \leftarrow \mathbf{KS}(\mathsf{ct}_{\mathsf{sign}}, \mathsf{PUB})$; `/* Algorithmn4 */`
$\overline{\overline{\mathsf{CT}}}_{\mathsf{sign}} \leftarrow \mathbf{CBS}\left(\mathsf{ct}_{\mathsf{sign}}, \mathsf{PUB}\right)$; `/* Algorithm 12 */`
$\mathsf{LUT} \leftarrow \mathsf{CMux}\left(\mathsf{LUT}_0, \mathsf{LUT}_1, \overline{\overline{\mathsf{CT}}}_{\mathsf{sign}}\right)$; `/* Algorithm 8 */`
$\mathsf{ct}_{\mathsf{out}} \leftarrow \mathbf{PBS}(\mathsf{ct}_{\mathsf{in}}, \mathsf{LUT}, \mathsf{PUB})$; `/* Algorithm 11 */`
return $\mathsf{ct}_{\mathsf{out}}$

[LMP21, YXS+21] WoP-PBS. Finally, the last **WoP-PBS** bootstrapping algorithm starts by enlarging the message space by adding a random bit which act as a new random padding bit. This newly created padding bit is then extracted and subtracted from the enlarged input ciphertext.

As a result, the ciphertext now encrypts the original input message along with a padding bit set to zero, which enables the use of a classical **PBS** to compute the desired result.

Remark 3.9. The cost of the [LMP21] **WoP-PBS** (Algorithm 23) can be approximated by:

$$\mathsf{Cost}^{\ell,k,N,n,q}_{\mathsf{WoPPBSLMP22}} = \mathsf{Cost}^{n,q,2q}_{\mathsf{ModulusSwitch}} + 2 \cdot \mathsf{Cost}^{\ell,k,N,n,q}_{\mathbf{PBS}} + 2 \cdot \mathsf{Cost}^{\ell,n,k,N}_{\mathbf{KS}} + \mathsf{Cost}^{k,N}_{\mathsf{Add}} + 2 \cdot \mathbb{C}_{add}.$$

Where $\mathbb{C}_{add}$ denotes the complexity of adding two elements of $\mathbb{Z}_q$.

Algorithm 23: $\mathsf{ct}_{\mathsf{out}} \leftarrow$ **WoP-PBS**[LMP22]$(\mathsf{ct}_{\mathsf{in}}, \mathsf{BSK}, \mathsf{LUT})$

Context:
- $\boldsymbol{s} = (s_0, \ldots, s_{n-1})$: the LWE secret key
- $\boldsymbol{S'} = (S'_0, \ldots, S'_{k-1})$: the GLWE secret key
- $\Delta m + e \in \mathbb{Z}_q$: The encoded message of $\mathsf{ct}_{\mathsf{in}}$, Definition 8
- $\mathsf{LUT}_{\mathbb{1}}$: Lookup table evaluating the function $x \mapsto 1$

Input:
- $\mathsf{ct}_{\mathsf{in}} = (a_{\mathsf{in},0}, \ldots, a_{\mathsf{in},n-1}, b_{\mathsf{in}}) \in \mathrm{LWE}_{\boldsymbol{s}}(m) \subseteq \mathbb{Z}^{n+1}_{2N}$
- LUT : Lookup table evaluating the function $x \mapsto f(x)$
- PUB : public keys required for the entire algorithm ; `/* Remark 2.15 */`

Output: $\mathsf{ct}_{\mathsf{out}} \in \mathrm{LWE}_{\boldsymbol{s}}\left(f\left(\Delta m + e\right)\right) \subseteq \mathfrak{R}^{k+1}_{q,N}$

```
/* Increase the message/ciphertext space by adding a random padding bit   */
ct ∈ Z^n_{2q} ← ModulusSwitch(ct_in, 2q) ;                    /* Algorithm 6 */
/* Generalization of Proof 13 (Algorithm 15) to enable the extraction of
   the most significant bit for message larger than one bit               */
ct_tmp ← ct + (0, Δ/2)
ct_tmp ← KS(ct_tmp, PUB) ;                                   /* Algorithmn4 */
ct_sign ← PBS(ct_tmp, LUT_𝟙, BSK) ;                          /* Algorithm 11 */
ct_sign ← ct_sign + (0, Δ/2)
ct ← ct − ct_sign
ct ← KS(ct, PUB) ;                                           /* Algorithmn4 */
ct_out ← PBS(ct, LUT, BSK);                                  /* Algorithm 11 */
return ct_out
```

In all the presented algorithm, one of the main step consisting on extracting the sign (i.e., the most significant bit). The proof of correctness of this operation can easily be adapted from the proof of Algorithm 15 (Theorem 3.1).

Later in Chapter 6, we will introduce a new variant of **WoP-PBS** developed and studied during this thesis.

3.4 Multiple LUT evaluation

In Section 2.6, one of the identified limitations is that the **PBS** procedure can evaluate only a single lookup table per operation (see Limitation 8). In this section, we present several solutions proposed in the literature to overcome this constraint.

We begin with the many-LUT **PBS** method introduced in [CLOT21], which enables the evaluation of multiple functions during a single bootstrapping operation. We then discuss the bootstrapping techniques proposed in [CIM19] and [GBA21], which introduce more complex algorithm to overcome Limitation 8.

Bootstrapping Many Lut. The first technique proposed to overcome Limitation 8 is the many-LUT bootstrapping method introduced in [CLOT21]. This PBS technique is similar to the classical version (Algorithm 11); however, compared to classical PBS, it requires some side information about the message. The ability to evaluate multiple functions lies in how the lookup table is constructed using this side information. Then, instead of performing a single sample extraction, several sample extractions are necessary, depending on the side information available.

During PBS, one performs a rotation of a lookup table representing N distinct values. To ensure correctness, a padding bit is required (Remark 2.14), which allows the encoded message to lie in the range $[0, N-1]$. If two padding bits are used, the encoded message lies in $[0, N/2-1]$, and only half of the LUT is used during rotation. Consequently, the remaining part of the lookup table can be repurposed to encode additional functions. The more we know empty bit in the encoded message, the more we can evaluate different functions.

Algorithm 24: $\{\mathsf{ct}_{\mathsf{out},i}\}_{i\in[0,2^{\pi-1}]} \leftarrow \mathsf{ManyLut}(\mathsf{ct}_{\mathsf{in}}, \mathsf{CT}_f, \mathsf{BSK})$

Context:
- $\boldsymbol{s} = (s_0, \dots, s_{n-1})$: the LWE secret key
- $\boldsymbol{S'} = (S'_0, \dots, S'_{k-1})$: the GLWE secret key
- $\boldsymbol{s'}$ the flatten representation of the GLWE secret key, Definition 11
- $\Delta m + e \in \mathbb{Z}_q$: The encoded message of $\mathsf{ct}_{\mathsf{in}}$, Definition 8
- π : The number of padding bits
- LUT_f : The Lookup table evaluating the function
  $x \mapsto f_1(x)$ on the $N/2^{\pi-1}$ first coefficients,
  then the function $x \mapsto f_2(x)$ on the $N/2^{\pi-1}$ next coefficients
  and so on until $x \mapsto f_{2^{\pi-1}}(x)$
- CT_f : Trivial encryption of LUT_f, Remark 2.6
- $(\mathfrak{B}, \ell) \in (\mathbb{Z}^*)^2$: The (base, level) decomposition.

Input:
- $\mathsf{ct}_{\mathsf{in}} = (a_{\mathsf{in},0}, \dots, a_{\mathsf{in},n-1}, b_{\mathsf{in}}) \in \mathrm{LWE}_{\boldsymbol{s}}(m) \subseteq \mathbb{Z}_{2N}^{n+1}$
- $\mathsf{CT}_f \in \mathrm{GLWE}_{\boldsymbol{S}}(\mathsf{LUT}_f) \subseteq \mathfrak{R}_{q,N}^{k+1}$
- $\mathsf{BSK} = \left\{\mathsf{BSK}_i \in \mathrm{GGSW}_{\boldsymbol{S'}}^{\mathfrak{B},\ell}(s_i)\right\}_{i\in[0,n-1]}$

Output: $\left\{\mathsf{ct}_{\mathsf{out}} \in \mathrm{LWE}_{\boldsymbol{s'}}\left(f\left(\Delta m + e\right)\right) \subseteq \mathfrak{R}_{q,N}^{k+1}\right.$

```
ct ← ModulusSwitch(ct_in, 2N);                    /* Algorithm 6 */
CT ← BlindRotation(ct_in, CT_f, BSK);             /* Algorithm 9 */
for i ∈ [1, 2^{π-1}] do
    ct_{out,i} ← SampleExtract(CT);               /* Algorithm 10 */
    CT ← CT · X^{N/2^{π-1}};                      /* Theorem 2.4 */
return {ct_{out,i}}_{i∈[0,2^{π-1}]}
```

Theorem 3.2 (Bootstrapping Many-LUT (Algorithm 24))**.** *Let $\boldsymbol{s} = (s_0, \dots, s_{n-1}) \in \mathbb{Z}_q^n$ be a binary LWE secret key and $\boldsymbol{S'} \in \mathfrak{R}_{q,N}^k$ be a GLWE secret key and $\boldsymbol{s'}$ his flatten representation. Let $\mathsf{ct}_{\mathsf{in}} = (a_{\mathsf{in},0}, \dots, a_{\mathsf{in},n-1}, b_{\mathsf{in}}) \in \mathrm{LWE}_{\boldsymbol{s}}(m) \subseteq \mathbb{Z}_{2N}^{n+1}$ encrypting the encoded message $m\Delta + e \in \mathbb{Z}_q$ with π padding bits fixed to zero. Let BSK be the bootstrapping key as presented in Definition 17 Finally let the lookup table $\mathsf{LUT}_f \in \mathfrak{R}_{q,N}$ such that the $N/2^{\pi-1}$ first coefficients encode the function $f_1 : x \in [0, N/2^{\pi-1}] \to f_1(x)$, then the next $N/2^{\pi-1}$ encode the function $f_2 : x \in [0, N/2^{\pi-1}] \to f_2(x)$ and so on until the function $f_{2^{\pi-1}} : x \in [0, N/2^{\pi-1}] \to f_{2^{\pi-1}}(x)$.*

Then Algorithm 24 returns $2^{\pi-1}$ LWE ciphertexts each encrypting the input message evaluated with one of the function $\left\{\mathsf{ct}_{\mathsf{out},i} \in \mathrm{LWE}_{\boldsymbol{s'}}\left(f_i(\Delta m + e \mod 2N)\right) \subseteq \mathfrak{R}_{q,N}^{k+1}\right\}_{i\in[0,2^{\pi-1}]}$. An LWE ciphertext encrypting $f(m + e \mod 2N)$ under the secret key $\boldsymbol{s'}$

The cost of the Many Lut Bootstrapping (Algorithm 24) is equivalent to that of the standard bootstrapping (Algorithm 11), except that $2^{\pi-1}$ sample extractions must be performed.

$$\mathsf{Cost}^{\ell,k,N,n,q,\pi}_{\mathsf{ManyLut}} = \mathsf{Cost}^{\ell,k,N,n,q}_{\mathbf{PBS}} + (2^{\pi-1}-1)\mathsf{Cost}^{k,N}_{\mathsf{SampleExtract}}.$$

And for a trivially encrypted lookup table (Definition 16), the output noise variance is equal to:

$$\mathsf{Var}(\mathsf{ManyLut}) = \mathsf{Var}(\mathbf{PBS}).$$

Proof *(Theorem 3.2). The proof is similar to that of bootstrapping (Theorem 2.15). The main differences lie in the lookup table generation and the final sample extractions. After the blind rotation, we obtain $\mathsf{CT} = \mathsf{CT}_f \cdot X^{\Delta m+e}$ with $\Delta m + e \in [0, N/2^{\pi-1}]$. So the first sample extract returns $\mathsf{ct}_{\mathsf{out},1} = \mathrm{LWE}_{\boldsymbol{s}'}(f_1(\Delta m + e))$. We then apply the rotation by $N/2^{\pi-1}$, so we have $\mathsf{CT} = \mathsf{CT}_f \cdot X^{\Delta m+e+N/2^{\pi-1}}$. Since each encoded function occupies $N/2^{\pi-1}$ coefficients, the first coefficient on CT now corresponds to the value $f_2(\Delta m + e)$, which is extracted in the next sample extraction. The algorithm continues this way until all function results have been extracted.* □

Bootstrapping [CIM19]. The second technique proposed to address Limitation 8 is the bootstrapping method introduced in [CIM19]. This technique enables the evaluation of multiple functions $\{f_i\}_i$ over a single encrypted value. Each function f_i is first represented as a polynomial P_i. Then, for each of these polynomials, one needs to find a common polynomial Q such that for every i we have $P'_i \cdot Q = P_i$. This process is referred to as $\mathsf{PolyFactor}$ where $(Q, \{P'_i\}_i) \leftarrow \mathsf{PolyFactor}(\{P_i\}_i)$. A programmable bootstrapping is then performed using the lookup table Q and the input ciphertext $\mathsf{ct}_{\mathsf{in}}$. Finally, each polynomial P'_i is multiplied with the result of the **PBS** resulting in GLWE ciphertexts that encrypting $P'_i \cdot Q$ (see polynomial multiplication Theorem 2.6). Each resulting GLWE ciphertext is then sample extracted to obtain an LWE ciphertext encrypting $f_i(\Delta m + e)$.

Remark 3.10. Compared to classical PBS, the bootstrapping method from [CIM19] can take as input multiple lookup tables, enabling the evaluation of several functions on a single ciphertext.

The drawback of this approach is the noise introduced during the final polynomial multiplication by the constant P'_i. Indeed, this multiplication adds a significant amount of noise, depending on the value of P'_i. But compared to the previous technique, no side information is required.

The cost of Algorithm 25 can be approximated by:

$$\mathsf{Cost}^{\ell,k,N,n,q,z}_{\mathsf{CIM19}\mathbf{PBS}} = \mathsf{Cost}^{\ell,k,N,n,q}_{\mathbf{PBS}} + z \cdot \mathsf{Cost}^{k,N}_{\mathsf{SampleExtract}}.$$

Bootstrapping [GBA21]. In [GBA21], the authors propose two bootstrapping methods, namely *tree*-**PBS** and *chain*-**PBS**, which enable bootstrapping over high-precision ciphertexts, solving Limitation 4. Furthermore, by employing specially constructed lookup tables, these **PBS** methods also allow for the evaluation of multivariate functions using multiple input ciphertexts, thus solving Limitation 7.

The goal of this bootstrapping is to enable the evaluation of multivariate function taking as input several ciphertexts. These input ciphertexts can either represent distinct variables—allowing the evaluation of a multivariate function via a multivariate lookup table or they can encode a single high-precision message split across multiple ciphertexts. This encoding for large precision is discussed in greater detail in Chapter 7 Section 7.2.

The case of a high-precision message encoded in two ciphertexts is detailed in Algorithm 26, though the same approach can be adapted to evaluate bivariate lookup tables as well. The main difference between the two cases lies in the generation of the lookup table.

The idea behind the *tree*-**PBS** (Algorithm 26), is to leverage the **PBS** technique from [CIM19] (Algorithm 25) on the first input ciphertext to compute multiple intermediate values. These values are then used to construct a new lookup table via a packing key switch (Theorem 2.9). Finally, this new lookup table is evaluated using a classical **PBS** applied with the second input ciphertext. To extend the procedure to more than two ciphertexts, the idea is to perform several

Algorithm 25: $\{\mathsf{ct}_{\mathsf{out},i}\}_{i\in[0,z]} \leftarrow$ [CIM19] Bootstrapping$(\mathsf{ct}_{\mathsf{in}}, \{P_i\}_{i\in[0,z]}, \mathsf{BSK})$

Context:
- $\boldsymbol{s} = (s_0, \ldots, s_{n-1})$: the LWE secret key
- $\boldsymbol{S'} = (S'_0, \ldots, S'_{k-1})$: the GLWE secret key
- $\boldsymbol{s'}$ the flatten representation of the GLWE secret key, Definition 11
- $\Delta m + e \in \mathbb{Z}_q$: The encoded message of $\mathsf{ct}_{\mathsf{in}}$, Definition 8
- z : The number of function
- P_i : The function f_i encoded as a polynomial
- $(\mathcal{B}, \ell) \in (\mathbb{Z}^*)^2$: The (base, level) decomposition.

Input:
- $\mathsf{ct}_{\mathsf{in}} = (a_{\mathsf{in},0}, \ldots, a_{\mathsf{in},n-1}, b_{\mathsf{in}}) \in \mathrm{LWE}_{\boldsymbol{s}}(m) \subseteq \mathbb{Z}_q^{n+1}$
- $\{P_i\}_{i\in[0,z]}$
- $\mathsf{BSK} = \left\{\mathsf{BSK}_i \in \mathrm{GGSW}^{\mathcal{B},\ell}_{\boldsymbol{S'}}(s_i)\right\}_{i\in[0,n-1]}$

Output: $\left\{\{\mathsf{ct}_{\mathsf{out},i} \in \mathrm{LWE}_{\boldsymbol{s'}}\left(f_i\left(\Delta m + e\right) \subseteq \mathbb{Z}_q^{kN+1}\right)\}_{i\in[0,z]}\right.$

1 $(Q, P'_i) \leftarrow \mathsf{PolyFactor}(\{P_i\}_{i\in[0,z]})$

`/* Algorithm 11, without sample extract (Algorithm 10) */`

2 $\mathsf{CT} \leftarrow \mathbf{PBS}(\mathsf{ct}_{\mathsf{in}}, Q, \mathsf{BSK})$

3 **for** $i \in [0, z]$ **do**

4 $\quad \mathsf{CT}_i \leftarrow \mathsf{CT} \cdot P'_i$

5 $\quad \mathsf{ct}_{\mathsf{out},i} \leftarrow \mathsf{SampleExtract}(\mathsf{CT}_i)$; `/* Algorithm 10 */`

6 **return** $\{\mathsf{ct}_{\mathsf{out},i}\}_{i\in[0,z]}$

successive [CIM19]$-$**PBS**, where at each step, the evaluated lookup table is derived from the result of the previous [CIM19] bootstrapping.

In the second method, i.e., the *chain*-**PBS**, the idea is that the output of a lookup is used to construct the selector for the next lookup, whereas in the tree-based approach, the output is used to construct the next lookup table. This distinction has significant implications for both error propagation and the overall functionality of the algorithm.

Remark 3.11. The cost of the [GBA21] **WoP-PBS** (Algorithm 26) can be approximated by:

$$\mathsf{Cost}^{\ell,k,N,n,q,z}_{\mathsf{GBA21}\mathbf{PBS}} = \mathsf{Cost}^{\ell,k,N,n,q,z}_{\mathsf{CIM19}\mathbf{PBS}} + N \cdot \mathsf{Cost}^{k,N}_{\mathsf{Add}} + \cdot\mathsf{Cost}^{\ell,k,N,n,q}_{\mathbf{PBS}} + z \cdot \mathsf{Cost}^{\ell,n,k,N}_{\mathbf{KS}}.$$

Algorithm 26: $\{\mathsf{ct}_{\mathsf{out}}\} \leftarrow$ [GBA21] Bootstrapping$(\mathsf{ct}_1, \mathsf{ct}_2, \{P_i\}_{i\in[0,p-1]}, \mathsf{PUB})$

Context:
$$\begin{cases} \boldsymbol{s} = (s_0, \ldots, s_{n-1}): \text{ the LWE secret key} \\ \boldsymbol{S'} = (S'_0, \ldots, S'_{k-1}): \text{ the GLWE secret key} \\ \boldsymbol{s'} \text{ the flatten representation of the GLWE secret key, Definition 11} \\ \Delta m_i + e_i \in \mathbb{Z}_q : \text{The encoded message of } \mathsf{ct}_i, \\ \qquad \text{with } m_i \in \mathbb{Z}_p \text{ for } i \in [0,1], \text{ Definition 8} \\ P_i : \text{The function } f_i \text{ encoded as a polynomial} \\ (\mathscr{B}, \ell) \in (\mathbb{Z}^*)^2 : \text{The (base, level) decomposition.} \end{cases}$$

Input:
$$\begin{cases} \mathsf{ct}_1 \in \mathrm{LWE}_{\boldsymbol{s}}(m_1) \subseteq \mathbb{Z}_q^{n+1} \\ \mathsf{ct}_2 \in \mathrm{LWE}_{\boldsymbol{s}}(m_2) \subseteq \mathbb{Z}_q^{n+1} \\ \{P_i\}_{i\in[0,p-1]} \\ \mathsf{BSK} = \left\{\mathsf{BSK}_i \in \mathrm{GGSW}^{\mathscr{B},\ell}_{\boldsymbol{S'}}(s_i)\right\}_{i\in[0,n-1]} \\ \mathsf{KSK} = \left\{\mathsf{KSK}_i \in \mathrm{GLEV}^{\mathscr{B},\ell}_{\boldsymbol{S}_{\mathsf{out}}}(-s_{\mathsf{in},i})\right\}_{i\in[0,n-1]} \end{cases}$$

Output: $\left\{\{\mathsf{ct}_{\mathsf{out},i} \in \mathrm{LWE}_{\boldsymbol{s'}}\left(f_i\left(\Delta m + e\right) \subseteq \mathbb{Z}_q^{kN+1}\right)\}_{i\in[0,z]}\right.$

$\{\mathsf{ct}_i\}_{i\in[0,p-1]} \leftarrow$ [CIM19]$-$Bootstrapping$(\mathsf{ct}_1, \{P_i\}_{i\in[0,p-1]}, \mathsf{BSK})$; /* Algorithm 25 */
$\mathsf{LUT} \leftarrow (\mathbf{0}, 0)$
for $i \in [0, N/p-1]$ **do**
  $\mathsf{CT} \leftarrow \mathsf{LWEKeySwitch}(\mathsf{ct}_i, \mathsf{KSK})$; /* Algorithm 4 */
  for $j \in [0, p-1]$ **do**
    $\mathsf{LUT} \leftarrow \mathsf{LUT} + \mathsf{CT} \cdot X^{i\cdot p+j}$
$\mathsf{ct}_{\mathsf{out}} \leftarrow \mathbf{PBS}(\mathsf{ct}_2, \mathsf{LUT}, \mathsf{BSK})$; /* Algorithm 11 */
return $\{\mathsf{ct}_{\mathsf{out}}\}$

Part II

TFHE High-Performance Primitives

This part of the thesis explores new advancements that enhance the efficiency of TFHE and, more generally, Fully Homomorphic Encryption. While FHE has made significant progress over the past decade, computational efficiency remains a critical challenge for its practical deployment. To address this challenge, we investigate different approaches, such as developing and optimizing new algorithms or introducing new security assumptions that enable the creation of novel cryptographic techniques.

Chapter 4

Accelerating TFHE with Sorted Bootstrapping Techniques

In Chapter 2, we outlined several limitations of the TFHE scheme. Notably, we presented that TFHE bootstrapping becomes progressively less efficient as the precision increases, and impractical for high-precision computations. This chapter focuses on improving the bootstrapping for medium precision making TFHE more efficient for these precisions. In Chapter 3, Sections 3.1.5 and 3.2.1, we detail the Extended **PBS** (**EBS**) and its parallelized version, a bootstrapping technique that is particularly efficient for medium precision (between 5 and 8 bits). We recall, that the **EBS** improves the performance of blind rotation by splitting the polynomials used during the blind rotation into several smaller polynomials. This approach enables faster computation of the external products, directly reducing the execution time of the **PBS**.

This chapter presents new methods and optimizations based on the **EBS** algorithm, leading to significant performance improvements. First, we show a simple but not straightforward technique used to eliminate external products necessary to perform the blind rotation during the **SBS**. Then, we show how the number of removed operations can be improved by modifying the modulus switch, and finally, we present how to increase the parallelization of this new technique.

4.1 Overview of the Construction

The first improvement presented in this chapter is the Sorted Extended PBS (**SBS**). By analyzing the rotation of lookup tables in the **EBS**, we identify patterns of rotation among the different split lookup tables. These patterns reveal that certain external products are unnecessary for the correctness of the final result and then can be removed during the blind rotation. By sorting the order of each rotation during the blind rotation, the number of external products can be minimized, offering a speedup compared to the **EBS**.

More precisely, in the **EBS**, the sample extract only needs to be performed on the first split lookup table lut_0. Thus, in the last step, only one of the η CMuxes operations needs to be computed, the one that returns lut_0. This insight can be pushed further: during the extended blind rotation, only the CMuxes operations that impact the output of the last CMux need to be computed. The idea is to perform a backward tracing of the full blind rotation, to identify which external products are necessary to obtain the correct result. In Section 3.1.5, we introduced the $\tau(\cdot)$ function, which allows splitting a GLWE ciphertext under an extended secret key into multiple GLWE ciphertexts of smaller degree. Then, Lemma 3.5 shows that a rotation r of $\tau(\mathsf{LUT}) = [\mathsf{lut}_0, \ldots, \mathsf{lut}_{\eta-1}]$ can be expressed as a change of indices along with an internal

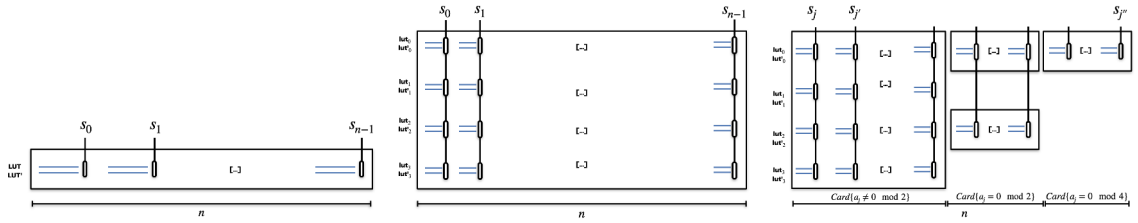

Figure 4.1: Representation of the chain of CMuxes during the blind rotation using the usual TFHE approach [CGGI20] (left side), the Extended bootstrapping one [LY23] (middle) and our Sorted bootstrapping (right side). Lines represent ciphertexts, and rounded rectangles correspond to CMuxes. Larger lines in left side represents larger polynomial degrees.

rotation, such that $\tau(\mathsf{LUT} \cdot X^r) = [\mathsf{lut}'_0, \ldots, \mathsf{lut}'_{\eta-1}]$ where $\mathsf{lut}'_i = \mathsf{lut}_{[(i-r)]_\eta} \cdot X^{\left\lceil \frac{r-i}{\eta} \right\rceil}$, for $i \in [0, \eta)$. By examining the rotation by r, we show that if $r = 0 \mod \eta$, lut'_0 only corresponds to a rotation of lut_0. Similarly, if $r = 0 \mod \eta/2$, lut'_0 corresponds either to a rotation of lut_0 or a rotation of $\mathsf{lut}_{\eta/2}$. Finally, when $r \neq 0 \mod 2$, lut'_0 can corresponds to a rotation of any lut. During bootstrapping, the rotation is performed using the coefficients a_i from the input LWE ciphertext. The goal is now to minimize the number of external products used during the blind rotation. This implies that the first rotations by all the $a_i \neq 0 \mod 2$ require η CMuxes. The subsequent rotations for a_i values such that $a_i \neq 0 \mod 2^2$ need $\eta/2$ CMuxes, and so on. The final rotations require only one CMux. This results in a CMux tree, where at each step, the number of CMuxes operations is divided by 2. In comparison, the usual approach would requires η CMuxes for each rotation. As previously explained, the goal is to sort the coefficients a_i in order to construct a CMux tree that minimizes the number of CMux operations required to perform the blind rotation. This optimization is fully detailed in Section 4.2.

The second improvement is a variant of the modulus switch (Algorithm 6), named Companion Modulus Switch (CMS), designed to increase the number of elements in the blind rotation that fulfill the necessary conditions for reducing the number of external products in the **SBS**. The goal of the CMS is to improve the sorting in the **SBS** by grouping more elements into classes of sorted elements which reduces the number of external products. To achieve this, instead of applying a standard rounding, the CMS computes a larger approximation, ensuring that the resulting values belong to appropriate sorted classes. For instance, when computing a **SBS**, the largest number of external products required in a single step occurs when the rotation satisfies $a_j \neq 0 \mod 2$. In a bootstrapping, the a_j are the results of the usual modulus switch, i.e., a rounding operation, which could equivalently be written as either the ceil or a floor, depending on the input value. Results are generally rounded to the nearest representable value, i.e., using the *rounding to the nearest* mode. In some cases, this mode gives a value a_j such that $a_j \neq 0 \mod 2$, whereas computing another rounding could have led to a more convenient value $a_j = 0 \mod 2$. The larger the value of i for $i \in [0, \log_2(\eta)]$ such that the rounding of a_j equals $0 \mod 2^i$, the greater the reduction in the number of external products. The idea is then to take the adequate rounding mode for certain a_j where $a_j \neq 0 \mod 2$. Practically, this means that if the rounding operation is equivalent to a floor, we apply the ceiling instead, and vice versa. To limit the additional noise introduced compared to the usual modulus switch, it is generally better to modify only a subset of the values rather than all of them. This introduces a new fine-grained parameter that optimizes the number of CMuxes to compute while limiting noise growth. We show that only the first layer of a_j needs to be considered to maximize the efficiency of this approach. This adjustment results in an additional latency reduction for the **PBS** and is detailed in Section 4.3.

Finally, we focus on reducing the latency of both **EBS** and **SBS** by mixing several steps of the blind rotation into a single parallel operation. We demonstrate how techniques from [BMMP18, LLW+24] can be effectively integrated into our sorted approach. The idea stems

from developing the original CMux equation to mix and match the rotations done through the keys, and the sorted bootstrapping. To compute a bootstrapping using the two combined techniques, instead of computing n sequential CMuxes over polynomials of size $\eta \cdot N$, we can compute a bootstrapping in the equivalent of $\frac{n}{\varpi}$ sequential CMuxes by using $\eta \cdot 2^{\varpi}$ threads, where ϖ corresponds to the number of packed rotations. We analyze the balance between the size of the polynomials and the number of parallelized external products to achieve optimal parallelization efficiency. This leads to a highly parallel bootstrapping, particularly well suited for hardware architectures. This is detailed in Section 4.4.

4.2 Sorted Extended Bootstrapping

In this section, propose a new method to compute the **EBS** presented in Section 3.2.1 which in a sequential context provides significant speedups compared to the classical **PBS** or the **EBS**, and in a parallelized context, frees up some threads (i.e., improves the throughput). To enhance the work proposed with [LY23], the idea is to sort the a_i mask coefficients of the input LWE ciphertext of the **PBS**. By performing this sorting, some of the CMuxes can be removed during the computation of the blind rotation. This new **PBS** is called **SBS** for Sorted Extended Bootstrapping.

The idea comes from the following observation: since we work with an extended secret key, i.e., a secret key composed of both random elements and known zeros, the sample extraction (Algorithm 10) performed at the end of bootstrapping only needs to be applied to coefficients that are encrypted under unknown key elements and involved in the encryption of b_0. When we apply the function τ to the lookup table, i.e., $\tau(\mathsf{LUT}) = (\mathsf{lut}_0, \ldots, \mathsf{lut}_{\eta-1})$, the value b_0 is encrypted in the first lookup table, lut_0. Then, we only need to perform sample extraction on lut_0, as the other values do not affect the encryption of b_0. In a standard bootstrapping, at the n^{th} step, we compute: $(\mathsf{LUT} \cdot X^{a_{n-1}} - \mathsf{LUT}) \boxdot \mathrm{GGSW}^{\mathcal{R},\ell}_{\boldsymbol{S}_{\mathsf{ext}}}(s_{n-1}) + \mathsf{LUT}$ (Algorithm 8). To achieve the equivalent computation in the extended bootstrapping(Algorithm 17), we apply the function τ to the ciphertext and compute $(\mathsf{lut}'_j - \mathsf{lut}_j) \boxdot \mathrm{GGSW}^{\mathcal{R},\ell}_{\boldsymbol{S}}(s_{n-1}) + \mathsf{lut}_j$ for $j \in [0, \eta)$ where $\mathsf{lut}'_j = \mathsf{lut}_{[(j-a_{n-1})]_\eta} \cdot X^{\left\lceil \frac{a_{n-1}-j}{\eta} \right\rceil}$ (see Lemma 3.5). However, in the extended bootstrapping case, only the final result of lut_0 matters, then we only need to compute the CMux impacting the result on lut_0 without losing any information required for subsequent steps.

First, we notice that the CMuxes can be computed in any order without impacting the result, as long as the a_i rotation is performed with the corresponding GGSW encrypting s_i. This comes from the linear part of the decryption, i.e., the dot product between $\boldsymbol{s}$ and $\boldsymbol{a}$. So, to perform the blind rotation, we can sort the input pairs $(a_i, \mathrm{GGSW}^{\mathcal{R},\ell}(s_i))$ in any specific order. So we raise the following question: Can we generalize the observation made for the last CMux to the other steps of the CMux during the blind rotation?

Let us introduce some arithmetic results that will be used to prove the correctness of the **SBS**.

Lemma 4.1. *Let $\eta \in \mathbb{Z}$ be a power of two. Let μ be a divisor of η. For $a \in \mathbb{Z}$, for $i \in [0, \eta)$ if $a = 0 \mod \mu$ and $i + a = 0 \mod \eta$ then $i = 0 \mod \mu$.*

Proof *(Lemma 4.1). Let $\eta \in \mathbb{Z}$ be a power of two and $\mu \in \mathbb{Z}$ such that $\mu | \eta$. Let $a \in \mathbb{Z}$, such that $a = 0 \mod \mu$, so it exists $k \in \mathbb{Z}$ such that $a = k\mu$. Let $i + a = 0 \mod \eta$, so it exists a $p \in \mathbb{Z}$ such that $i + a = p\eta$. We have:*

$$i + a = p\eta \Leftrightarrow i + k\mu = p\mu\frac{\eta}{\mu} \Leftrightarrow i = -k\mu + p\frac{\eta}{\mu}\mu = \mu\left(-k + p\frac{\eta}{\mu}\right).$$

□

Lemma 3.5 states that a rotation of a polynomial $P(X)$ by κ can be expressed as $\tau(P(X) \cdot X^{\kappa}) = [P'_0(X), \ldots, P'_{\eta-1}(X)]$ where $P'_j(X) = P_{[(j-\kappa)]_\eta}(X) \cdot X^{\left\lceil \frac{\kappa-j}{\eta} \right\rceil}$. Then, according to Lemma 4.1, after a rotation where $\kappa = 0 \mod \mu$, only the polynomial P_i with index $i = 0 \mod \mu$ can appear as the first polynomial $P'_0(X)$.

This observation can be directly applied to the lookup table during blind rotation: when performing a rotation by $a = 0 \mod \mu$, only the lookup table components with indices $i = 0 \mod \mu$ can impact the first lookup table lut_0. All other lookup table have no direct impact on lut_0 during this step of the rotation. However, since these components may influence the correctness in subsequent steps, they cannot be directly removed from the computation.

Definition 28. *Let $2^\xi = \eta \in \mathbb{Z}$ be the expansion factor. Let an* LWE *ciphertext $(a_0, \ldots, a_{n-1}, b) \subseteq \mathbb{Z}_q^{n+1}$, we denote $\mathfrak{a}_k$ the set containing the a_i, for $i \in [0, n)$, such that $a_i = 0 \mod 2^k$ and $a_i \neq 0 \mod 2^{k+1}$ for $k \in [0, \xi)$, and we denote $\mathfrak{a}_\xi$ the set containing the a_i such that $a_i = 0 \mod \eta$. Note that each a_i belongs to only one of the sets $\mathfrak{a}$.*

4.2.1 Sorted Bootstrapping Algorithm

This section describes how to sort the mask elements of the input LWE ciphertext, a core component of the **SBS** algorithm. Each coefficient is sorted into sets denoted $\mathfrak{a}_k$, and we compute only the necessary CMux operations for each of these sets, starting with $\mathfrak{a}_0$ elements, then $\mathfrak{a}_1$ elements, and continuing up to $\mathfrak{a}_\xi$. These sets are composed of the mask coefficients of the input ciphertext based on their results modulo 2^k for each $k \in [0, \xi]$ with $\xi \in \mathbb{N}$. This sorting strategy maximizes the number of unnecessary CMux operations that can be removed without impacting the correctness. As explained in the introduction, this approach ensures that we only execute the operations that impact the first lookup table, without losing any information necessary for subsequent steps. Specifically, during the blind rotation, when a coefficient a belongs to $\mathfrak{a}_k$, for some $k \in [0, \xi]$, we need to compute only $\frac{\eta}{2^k}$ external products. This is because only $\frac{\eta}{2^k}$ indices $i \in [0, \eta)$ satisfy $i = 0 \mod 2^k$ (Lemma 4.1).

We first introduce the Sorted Extended Programmable Bootstrapping (**SBS**) procedure in Algorithm 27. We then prove the correctness of this method and analyze the average performance gain it offers.

Remark 4.1. The noise distribution of this algorithm is similar to the one from the **PBS** described in Theorem 2.15. The difference is about the polynomial degree: for a fixed precision, when $\eta > 1$, a classical **PBS** operates with polynomials of degree $N\eta - 1$, whereas the **SBS** operates with polynomials of degree $N - 1$. As a result, the noise introduced by the **SBS** is smaller than that of the classical **PBS** for the same target precision. It can be approximated using the same estimation formula: $\mathsf{Var}(\mathbf{SBS}) = \mathsf{Var}(\mathbf{PBS})$ with polynomial size equals N instead of ηN.

Lemma 4.2 (Cost & Correctness of Algorithm 27)**.** *Algorithm 27 takes as input the bootstrapping key $\mathsf{BSK} = (\mathsf{BSK}_0, \cdots \mathsf{BSK}_{n-1}) \in \left[\mathfrak{R}_{q,N}^{(k+1)\ell \times (k+1)}\right]^n$ (Definition 17), the lookup table $\mathsf{LUT}_f \in \mathfrak{R}_{q,\eta N}^{(k+1)\ell \times (k+1)}$ encoding the function f (Definition 16) and the input ciphertext $\mathsf{ct}_m \in \mathrm{LWE}_{\boldsymbol{s}}(m) \subseteq \mathbb{Z}_q^{n+1}$. Then Algorithm 27 outputs the ciphertext $\mathsf{ct}_{f(m)} \in \mathrm{LWE}_{\boldsymbol{s}_{\mathsf{out}}}(f(m)) \subset \mathbb{Z}_q^{kN+1}$ with probability $1 - \mathsf{p}_{\mathsf{fail}}$ (Definition 2). On average, the number of* CMuxes *required in Algorithm 27 is $n \cdot \left(\frac{2\eta^2+1}{3\eta}\right)$ and his cost is:*

$$\mathsf{Cost}_{\mathbf{SBS}}^{\ell,k,N,\nu} = \mathsf{Cost}_{\mathbf{MS}}^{n,2\eta N} + n\left(\frac{2\eta^2+1}{3\eta}\right)\mathsf{Cost}_{\mathsf{CMux}}^{\ell,k,N} + \mathsf{Cost}_{\mathbf{SE}}^{N}.$$

Proof *(Correctness of Algorithm 27). The correctness of the* **EBS***, i.e., the case where all* CMuxes *are computed, has been proven in [LY23] and recalled in Section 3.1.5 in proof of Theorem 3.3 and proof of Theorem 3.4. In this proof, we show that the correctness is preserved after removing the useless* CMuxes*.*

We recall that, at the end of the **SBS***, the sample extract is only done on the first split lookup table (lut_0), so the blind rotation only need to compute the external products which impact this* lut*.*

From Lemma 4.1, we know that only the lookup table lut_i with indices $i = 0 \mod 2^k$ can affect lut_0 when the rotation is $\tilde{a} = 0 \mod 2^k$. The goal is now to show that moving to the next congruence class does not involve any lookup table component that was previously discarded.

Algorithm 27: $\mathsf{ct}_{f(m)} \leftarrow \mathbf{SBS}(\mathsf{ct}_m, \mathsf{LUT}_f, \mathsf{BSK})$

Context:
- $\boldsymbol{s} = [s_0, \ldots, s_{n-1}] \subseteq \mathbb{Z}_q^n$
- $\mathsf{BSK}_i \in \mathrm{GGSW}_{\boldsymbol{S}}^{\mathcal{R},\ell}(s_i) \subseteq \mathfrak{R}_{q,N}^{(k+1)\ell\times(k+1)}$
- $\eta = 2^\xi$: The extended factor
- $\iota : \mathfrak{R}_{q,N} \to \mathfrak{R}_{q,\eta N}$, as defined in Lemma 3.2.
- $\tau : \mathfrak{R}_{q,\eta N} \to [\mathfrak{R}_{q,N}]^\eta$, as defined in Lemma 3.4.
- $\boldsymbol{S}_{\mathsf{ext}}$ is the extended secret key.$(\boldsymbol{S}_{\mathsf{ext}} \leftarrow \iota(\boldsymbol{S}))$

Input:
- $\mathsf{BSK} = (\mathsf{BSK}_0, \ldots, \mathsf{BSK}_{n-1}) \in \left[\mathfrak{R}_{q,N}^{(k+1)\ell\times(k+1)}\right]^n$
- $\mathsf{ct}_m = (a_0, \ldots, a_{n-1}, b) \in \mathrm{LWE}_{\boldsymbol{s}} \subseteq \mathbb{Z}_q^{n+1}$
- $\mathsf{LUT}_f \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{ext}}} \subseteq \mathfrak{R}_{q,\eta N}^{(k+1)\ell\times(k+1)}$

Output: $\mathsf{ct}_{f(m)} = \left(a_0^{\mathsf{out}}, \ldots, a_{kN-1}^{\mathsf{out}}, b\right) \in \mathrm{LWE}_{\boldsymbol{s}_{\mathsf{out}}} \subseteq \mathbb{Z}_q^{kN+1}$

$\mathsf{ct}_{\mathbf{MS}} = \left(\tilde{a}_0, \cdots \tilde{a}_{n-1}, \tilde{b}\right) \leftarrow \mathbf{MS}(\mathsf{ct}_m, \eta \cdot 2N)$
$\mathfrak{a}_0 = \mathfrak{a}_1 = \cdots = \mathfrak{a}_\xi = \varnothing$
$\mathbf{BSK}'_0 = \mathbf{BSK}'_2 = \cdots = \mathbf{BSK}'_\xi = \varnothing$
for $i \in [0, n-1]$ **do**
  $j \leftarrow 1$
  while $\tilde{a}_i = 0 \mod 2^j$ *and* $j < \xi$ **do**
    $j \leftarrow j + 1$
  $\mathfrak{a}_{j-1} \leftarrow \mathsf{append}(\mathfrak{a}_{j-1}, \tilde{a}_i)$
  $\mathbf{BSK}'_{j-1} \leftarrow \mathsf{append}\left(\mathbf{BSK}'_{j-1}, \mathsf{BSK}_i\right)$
$\mathsf{CT}_{\mathbf{BR}} \leftarrow \mathsf{LUT}_f \cdot X^{\tilde{b}}$
$\left[\mathsf{CT}_{\mathbf{BR}}^0, \ldots, \mathsf{CT}_{\mathbf{BR}}^{\eta-1}\right] \leftarrow \tau\left(\mathsf{CT}_{\mathbf{BR}}\right)$
for $i \in [0, \xi]$ **do**
  for $j \in [0, \mathsf{len}(\mathfrak{a}_i))$ **do**
    for $k \in [0, \frac{\eta}{2^i} - 1]$ **do**
      `/* Each step of the loop can be done in parallel */`
      $\mathsf{CT}_{\mathbf{BR}}^{k\cdot 2^i} \leftarrow \mathsf{CMux}\left(\mathsf{CT}_{\mathbf{BR}}^{k\cdot 2^i}, \mathsf{CT}_{\mathbf{BR}}^{[k\cdot 2^i - \mathfrak{a}_i[j]]_\eta} \cdot X^{\left\lceil \frac{\mathfrak{a}_i[j] - k\cdot 2^i}{\eta} \right\rceil}, \mathbf{BSK}'_i[j]\right)$;
      `/* Algorithm 8 */`
$\mathsf{ct}_{f(m)} \leftarrow \mathbf{SampleExtract}(\mathsf{CT}_{\mathbf{BR}}^0)$ **return**

When moving to the next congruence class $\mathfrak{a}_{k+1}$, only the indices $i = 0 \mod 2^{k+1}$ influence lut_0, and it holds that: $\left\{i \in [0,\eta) \mid i = 0 \mod 2^{k+1}\right\} \subset \left\{i \in [0,\eta) \mid i = 0 \mod 2^k\right\}$; Thus, when transitioning from $\mathfrak{a}_k$ to $\mathfrak{a}_{k+1}$, none of the lookup table components removed in earlier steps will affect the subsequent computation.

This guarantees that only the necessary CMux *operations are performed during the blind rotation, without compromising correctness.*

□

Proof *(Cost of Algorithm 27). The a_i are uniformly distributed for $i \in [0, n-1]$ and sorted into sets $\mathfrak{a}_k$, for $k \in [0, \xi]$, as explained in Definition 28. When $\tilde{a}$ is in $\mathfrak{a}_k$ for $k \in [0, \xi]$, only $\eta/2^k$ external products need to be computed. On average, $\left(1 - \frac{1}{2}\right) \cdot n$ elements are in $\mathfrak{a}_0$ (and thus there remains $\frac{n}{2}$ elements such that $a_i = 0 \mod 2$). For each of these elements, we need to compute η* CMuxes. *In the second step, with the $\frac{n}{2}$ remaining a_i, on average, we have $\mathsf{card}(\mathfrak{a}_1) = \left(1 - \frac{1}{2}\right) \cdot \frac{n}{2}$ (and thus there remain $\frac{n}{2^2}$ elements such that $a_i = 0 \mod \mu$). For each coefficients in $\mathfrak{a}_1$, we need to compute $\frac{\eta}{2}$* CMuxes.

We continue this process until the set $\mathfrak{a}_{\xi-1}$. At this step, there remain $\left(1 - \frac{1}{2}\right) \cdot \frac{n}{2^{\xi-1}}$ elements

that are not equal to zero modulus η *and* $\frac{n}{2^\xi}$ *elements such that* $a_i = 0 \mod 2^{\xi-1}$. *For these* a_i, *we need to compute* $\frac{\eta}{2^{\xi-1}} \cdot \mathsf{CMuxes}$. *For each of the* $\frac{n}{\eta}$ *remaining* a_i *in* $\mathfrak{a}_\xi$, *we need to compute one* CMux. *So, the total number of* CMuxes *is equal to:*

$$\frac{n}{2^\xi} + \sum_{i=0}^{\xi-1} \left(1 - \frac{1}{2}\right) \frac{n}{2^i} 2^{\xi-i} = n \left(2^{-\xi} + 2^{-1} \frac{2^\xi - 2^{-\xi}}{1 - 2^{-2}}\right) = n \cdot \left(\frac{2\eta^2 + 1}{3\eta}\right).$$

□

Remark 4.2. For the last a_i of each set $\mathfrak{a}_k$, for $k \in [0, \xi]$, we only need to compute $\eta/2^{k+1}$ CMuxes (compared to $\eta/2^k$ CMuxes for all the other a_i of the set $\mathfrak{a}_k$). Indeed, the lut of index j such that $j = 0 \mod 2^k$ and $j \neq 0 \mod 2^{k+1}$ will never be used in the following computation (the a_i of the set $\mathfrak{a}_{k+1}$ work only on the lut of index j such that $j = 0 \mod 2^{k+1}$). By considering this remark, we can reduce the number of CMuxes by $\frac{\eta}{2} \sum_{i=0}^{\xi-1} \frac{1}{2^i} = \eta - 1$.

4.3 Companion Modulus Switch

In Section 4.2.1, we show that when the mask elements are sorted per $\mathfrak{a}_k$ for $k \in [0, \xi]$, only $\frac{\eta}{2^k}$ external product at each step of the blind rotation are needed. When $\tilde{a}$ is in $\mathfrak{a}_k$, the higher the value k, the more we can reduce the number of external products. An easy way to improve the **SBS** is to reduce the number of mask elements in $\mathfrak{a}_0$ (i.e., when the maximum number of external products needs to be computed at each step of the blind rotation) and maximize the number of mask elements $\tilde{a}$ such that $\tilde{a} \in \mathfrak{a}_\eta$. To achieve this, we propose modifying the modulus switch by intentionally selecting either the ceiling or floor during the rounding operation. By doing so, for a mask element that would typically be in $\mathfrak{a}_0$ with the classical modulus switch, selecting the opposite rounding result (i.e., the floor if the rounding returns the ceiling and vice versa) can place the mask element into another $\mathfrak{a}_k$ for $k \in [1, \xi]$. The goal is to identify mask values where changing the rounding result shifts the value to $\mathfrak{a}_k$ for a k value close to ξ. The drawback of this modification is its impact on the noise. By selecting the ceiling or floor instead of the rounding result, we increase the rounding error added by the modified mask elements.

The modulus switch is already well known to be one of the most noisiest operations, therefore, we cannot arbitrarily change all the elements as desired. To moderate this noise growth, we introduce a new parameter d representing the number of $\tilde{a}_i$ on which this modified modulus switch will be applied. Assuming a fixed failure probability and security level, if the noise increase is too large, cryptographic parameters must also be larger to ensure the previous conditions. This might ruin the performance gain from this optimization. The goal is then to find the best value for d to improve the overall performance. We refer to it as the Companion Modulus Switch and denote it by CMS.

In this section, we first study the noise evolution of the CMS to ensure the correctness of the whole algorithm. Indeed the output noise of the CMS needs to satisfy the noise constraints for a given failure probability to be correctly applied during the **SBS**. We then present the average and the maximum gain offer by this method.

Lemma 4.3 (Noise CMS)**.** *Let* $(a_0, \ldots, a_{n-1}, b) \in \mathrm{LWE}_{\boldsymbol{s}} \subseteq \mathbb{Z}_q^{n+1}$ *be the input ciphertext. Let* $\alpha = \frac{q_{\mathsf{in}}}{q_{\mathsf{out}}}$ *and let* σ_{in}^2 *denote the input noise variance. For a chosen* d, *the* CMS *operation is done by computing* $a'_i = \left[\left\lfloor \frac{q_{\mathsf{out}}}{q_{\mathsf{in}}} a_i \right\rceil\right]_{q_{\mathsf{out}}}$ *for* $i \in [0, n-d)$ *and* $a''_i = \left[\left\lfloor \frac{q_{\mathsf{out}}}{q_{\mathsf{in}}} a_i \right\rfloor\right]_{q_{\mathsf{out}}} + \left[\left\lceil \frac{q_{\mathsf{out}}}{q_{\mathsf{in}}} a_i \right\rceil\right]_{q_{\mathsf{out}}} - \left[\left\lfloor \frac{q_{\mathsf{out}}}{q_{\mathsf{in}}} a_i \right\rceil\right]_{q_{\mathsf{out}}}$ *for* $i \in [n-d, n)$. *Any of the* a_i *for the* d *modified values can be chosen, but to simplify the* d *last values are taken. The variance of the* CMS *is:*

$$\begin{aligned}\mathsf{Var}(e_{\mathsf{CMS}}) = \frac{\sigma_{\mathsf{in}}^2}{\alpha^2} + \frac{1}{12} - \frac{1}{12\alpha^2} + \frac{d}{16\alpha^2 - 48\alpha + 144} \\ + \frac{n-d}{24} + \frac{(n-d)}{48\alpha^2} + \frac{d(7\alpha^4 - 39\alpha^3 + 29\alpha^2 + 90\alpha - 72)}{24\alpha^2(\alpha - 3)^2}.\end{aligned}$$

and

$$\mathbb{E}(e_{\mathsf{CMS}}) = \frac{1}{2\alpha} \cdot \left(\frac{n-d}{2} - 1\right) + \frac{d}{4\alpha - 12}.$$

Proof *(Lemma 4.3). Let note* $a'_i = \left\lfloor \frac{q_{\mathsf{out}}}{q_{\mathsf{in}}} a_i \right\rceil = \frac{q_{\mathsf{out}}}{q_{\mathsf{in}}} a_i + \bar{a}_i$. *Then we have* $a'_i \in \mathfrak{U}\left(\left[\frac{-q_{\mathsf{out}}}{2}, \frac{q_{\mathsf{out}}}{2}\right)\right)$ *and* $\bar{a}_i \in \frac{q_{\mathsf{out}}}{q_{\mathsf{in}}}\left[\frac{-q_{\mathsf{in}}}{2q_{\mathsf{out}}}, \frac{q_{\mathsf{in}}}{2q_{\mathsf{out}}}\right)$. *So* $\mathsf{Var}(a'_i) = \frac{q_{\mathsf{out}}^2 - 1}{12}$ *and* $\mathbb{E}(a'_i) = \frac{-1}{2}$, *and that* $\mathsf{Var}(\bar{a}_i) = \frac{1}{12} - \frac{q_{\mathsf{out}}^2}{12 q_{\mathsf{in}}^2}$ *and* $\mathbb{E}(\bar{a}_i) = -\frac{-q_{\mathsf{out}}}{2q_{\mathsf{in}}}$.

Let note $a''_i = \frac{q_{\mathsf{out}}}{q_{\mathsf{in}}} a_i + \bar{\bar{a}}_i$ *the ceiling or the floor chosen as the opposite of the round result (if* $a'_i = \left\lfloor \frac{q_{\mathsf{out}}}{q_{\mathsf{in}}} a_i \right\rceil = \left\lceil \frac{q_{\mathsf{out}}}{q_{\mathsf{in}}} a_i \right\rceil$ *then* $a''_i = \left\lfloor \frac{q_{\mathsf{out}}}{q_{\mathsf{in}}} a_i \right\rfloor$ *and if* $a'_i = \left\lfloor \frac{q_{\mathsf{out}}}{q_{\mathsf{in}}} a_i \right\rceil = \left\lfloor \frac{q_{\mathsf{out}}}{q_{\mathsf{in}}} a_i \right\rfloor$ *then* $a''_i = \left\lceil \frac{q_{\mathsf{out}}}{q_{\mathsf{in}}} a_i \right\rceil$*). Then we have* $a''_i \in \mathfrak{U}\left(\left[\frac{-q_{\mathsf{out}}}{2}, \frac{q_{\mathsf{out}}}{2}\right)\right)$ *and* $\bar{\bar{a}}_i \in \frac{q_{\mathsf{out}}}{q_{\mathsf{in}}}\mathfrak{U}\left(\left(\frac{-q_{\mathsf{in}}}{q_{\mathsf{out}}}, \frac{-q_{\mathsf{in}}}{2q_{\mathsf{out}}}\right) \cup \left[\frac{q_{\mathsf{in}}}{2q_{\mathsf{out}}}, \frac{q_{\mathsf{in}}}{q_{\mathsf{out}}}\right)\right)$ *Let* $\frac{q_{\mathsf{in}}}{q_{\mathsf{out}}} = \alpha$.

First, let compute the expectation and the variance of $\bar{\bar{a}}_i$.

$$\alpha \mathbb{E}(\bar{\bar{a}}_i) = \frac{1}{\alpha - 3}\left(\sum_{i=-\alpha+1}^{-\frac{\alpha}{2}-1} i + \sum_{i=\frac{\alpha}{2}}^{\alpha-1} i\right) = \frac{1}{\alpha-3}\frac{\alpha}{2} = \frac{1}{2-6\alpha^{-1}}$$
$$\mathbb{E}(\bar{\bar{a}}_i) = \frac{1}{2\alpha - 6}.$$

$$\alpha^2 \mathsf{Var}(\bar{\bar{a}}_i) = \frac{1}{\alpha-3}\left[\sum_{i=-\alpha+1}^{-\frac{\alpha}{2}-1} \underbrace{\left(i - \frac{1}{2-6\alpha^{-1}}\right)^2}_{I} + \sum_{i=\frac{\alpha}{2}}^{\alpha-1} \underbrace{\left(i - \frac{1}{2-6\alpha^{-1}}\right)^2}_{I}\right].$$

First let isolate I *and develop the equation:*

$$\left(i - \frac{1}{2-6\alpha^{-1}}\right)^2 = \underbrace{i^2}_{II} - \underbrace{\frac{2i}{2-6\alpha^{-1}}}_{III} + \underbrace{\left(\frac{1}{2-6\alpha^{-1}}\right)^2}_{IV}.$$

Now let compute II, III *and* IV *with the sums* $\sum_{i=-\alpha+1}^{-\frac{\alpha}{2}-1}$ *and* $\sum_{i=\frac{\alpha}{2}}^{\alpha-1}$

$$\sum_{i=-\alpha+1}^{-\frac{\alpha}{2}-1} \frac{-2i}{2-6\alpha^{-1}} + \sum_{i=\frac{\alpha}{2}}^{\alpha-1} \frac{-2i}{2-6\alpha^{-1}}$$
$$= \frac{-2}{2-6\alpha^{-1}}\left(\sum_{i=-\alpha+1}^{-\frac{\alpha}{2}-1} i + \sum_{i=\frac{\alpha}{2}}^{\alpha-1} i\right) = \frac{-\alpha}{2-6\alpha^{-1}}.$$

$$\sum_{i=-\alpha+1}^{-\frac{\alpha}{2}-1} \left(\frac{1}{2-6\alpha^{-1}}\right)^2 + \sum_{i=\frac{\alpha}{2}}^{\alpha-1} \left(\frac{1}{2-6\alpha^{-1}}\right)^2 = \left(\frac{1}{2-6\alpha^{-1}}\right)^2 (\alpha - 3).$$

$$\sum_{i=-\alpha+1}^{-\frac{\alpha}{2}-1} i^2 + \sum_{i=\frac{\alpha}{2}}^{\alpha-1} i^2 = 2\sum_{i=\frac{\alpha}{2}+1}^{\alpha-1} i^2 + \frac{\alpha^2}{4}$$
$$= \frac{1}{3}\left((\alpha-1)\alpha(2\alpha-1) - \left(\frac{\alpha}{2}+1\right)\left(\frac{\alpha}{2}+2\right)(\alpha+3)\right) + \frac{\alpha^2}{4}$$
$$= \frac{1}{3}\left(\frac{7\alpha^3}{4} - \frac{21\alpha^2}{4} - \frac{11\alpha}{2} + 6\right) + \frac{\alpha^2}{4}.$$

We now compute all together and we obtain the final variance:

$$\begin{aligned}\alpha^2\mathsf{Var}(\bar{\bar{a}}_i) &= \frac{1}{\alpha-3}\left[\frac{-\alpha}{2-6\alpha^{-1}} + \left(\frac{1}{2-6\alpha^{-1}}\right)^2(\alpha-3)\right.\\ &\qquad \left.+\frac{1}{3}\left(\frac{7\alpha^3}{4} - \frac{21\alpha^2}{4} - \frac{11\alpha}{2} + 6\right) + \frac{\alpha^2}{4}\right]\\ &= \frac{7\alpha^4 - 39\alpha^3 + 29\alpha^2 + 90\alpha - 72}{12(\alpha-3)^2}\\ \mathsf{Var}(\bar{\bar{a}}_i) &= \frac{7\alpha^4 - 39\alpha^3 + 29\alpha^2 + 90\alpha - 72}{12\alpha^2(\alpha-3)^2}.\end{aligned}$$

Now, we can compute the expectation and the variance of the **MS** *noise. First we decrypt the output.*

$$\begin{aligned}&\mathsf{Decrypt}(\left(a'_0, \cdots a'_{n-1-d}, a''_{n-d}, \cdots a''_{n-1}, b'\right), \boldsymbol{s}) = b' - \sum_{i=0}^{n-d-1} a'_i s_i - \sum_{i=n-d}^{n-1} a''_i s_i\\ &= \alpha^{-1}b + \bar{b} - \sum_{i=0}^{n-d-1}(\alpha^{-1}a_i + \bar{a}_i)s_i - \sum_{i=n-d}^{n-1}(\alpha^{-1}a_i + \bar{\bar{a}}_i)s_i\\ &= \alpha^{-1}\left(b - \sum_{i=0}^{n-1} a_i s_i\right) + \bar{b} - \sum_{i=0}^{n-1-d}\bar{a}_i \cdot s_i - \sum_{i=n-d}^{n-1}\bar{\bar{a}}_i s_i\\ &= \alpha^{-1}\delta m + \alpha^{-1}e + \bar{b} - \sum_{i=0}^{n-1-d}\bar{a}_i s_i - \sum_{i=n-d}^{n-1}\bar{\bar{a}}_i s_i.\end{aligned}$$

Next, we can study the error in the case of binary keys ($\mathsf{Var}(s_i) = \frac{1}{4}$ and $\mathbb{E}(s_i) = \frac{1}{2}$):

$$\begin{aligned}\mathsf{Var}(E_{\mathsf{CMS}}) &= \mathsf{Var}\left(\alpha^{-1}e + \bar{b} - \sum_{i=0}^{n-1-d}\bar{a}_i \cdot s_i - \sum_{i=n-d}^{n-1}\bar{\bar{a}}_i s_i\right)\\ &= \mathsf{Var}(\alpha^{-1}\sigma_{\mathsf{in}}) + \mathsf{Var}(\bar{b}) + (n-d)\mathsf{Var}(\bar{a}_i)(\mathsf{Var}(s_i) + \mathbb{E}^2(si))\\ &+(n-d)\mathbb{E}^2(\bar{a}_i)\mathsf{Var}(s_i) + d\mathsf{Var}(\bar{\bar{a}}_i)(\mathsf{Var}(s_i) + \mathbb{E}^2(si)) + d\mathbb{E}^2(\bar{\bar{a}}_i)\mathsf{Var}(s_i)\\ &= \frac{\sigma^2_{\mathsf{in}}}{\alpha^2} + \frac{1}{12} - \frac{1}{12\alpha^2} + \frac{n-d}{24} + \frac{(n-d)}{48\alpha^2}\\ &+\frac{d(7\alpha^4 - 39\alpha^3 + 29\alpha^2 + 90\alpha - 72)}{24\alpha^2(\alpha-3)^2} + \frac{d}{16\alpha^2 - 48\alpha + 144}.\end{aligned}$$

and

$$\begin{aligned}\mathbb{E}(E_{\mathsf{CMS}}) &= \mathbb{E}\left(\alpha^{-1}e + \bar{b} - \sum_{i=0}^{n-1-d}\bar{a}_i s_i - \sum_{i=n-d}^{n-1}\bar{\bar{a}}_i s_i\right)\\ &= \cancel{\mathbb{E}\left(\alpha^{-1}e\right)} + \mathbb{E}\left(\bar{b}\right) + \sum_{i=0}^{n-1-d}\mathbb{E}\left(\bar{a}_i s_i\right) + \sum_{i=n-d}^{n-1}\mathbb{E}\left(\bar{\bar{a}}_i s_i\right) = \frac{1}{2\alpha}\cdot\left(\frac{n-d}{2} - 1\right) + \frac{d}{4\alpha - 12}.\end{aligned}$$

□

The CMS can be implemented in Algorithm 27 by substituting the existing modulus switch in line 1 with the new one. From Lemma 4.3, we have the correctness of this new version of Algorithm 27 if the noise added by the CMS is smaller than a given constraint that ensure the correctness of the **SBS** with a given failure probability (Definition 23). The Companion Modulus Switch is denoted as:

$$\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{CMS}(\mathsf{CT}_{\mathsf{in}}, q_{\mathsf{out}}, d)$$

With the CMS, we reduce the number of CMuxes needed to compute the **BR**. Indeed, we can find the $\tilde{a}_i$ values that require η CMuxes and modify them to compute only 2^k CMuxes with $0 \leq k < \xi$. In the best case, we aim for the modified $\tilde{a}_i$ to be congruent to 0 modulus η, requiring only one CMux to be performed. Due to the CMS, we have fewer coefficients in $\mathfrak{a}_0$. The next lemma shows, on average, how many CMuxes are removed during the computation of the **SBS**.

Lemma 4.4. *Let $d \in \mathbb{Z}$. By performing Algorithm 27 with the Companion Modulus Switch $\mathsf{CMS}(\mathsf{CT}_{\mathsf{in}}, q_{\mathsf{out}}, d)$ instead of the classical Modulus Switch $\mathbf{MS}(\mathsf{CT}_{\mathsf{in}}, q_{\mathsf{out}})$, on the average case, we only need to compute $n \cdot \left(\frac{2\eta^2+1}{3\eta}\right) - d\frac{-\eta^2-2}{3\eta}$ CMuxes. In the best case, where all the modified values satisfy $a_i = 0 \mod \eta$, we only need to compute $n \cdot \left(\frac{2\eta^2+1}{3\eta}\right) - d(\eta-1)$ CMuxes. The cost of the CMS is similar to the one of the usual **MS** and we obtain:*

$$\mathsf{Cost}^{d,\ell,k,N,\nu}_{\mathbf{SBS}-\mathsf{CMS}} = \mathsf{Cost}^{n,2\eta N}_{\mathbf{MS}} + \left(n\left(\frac{2\eta^2+1}{3\eta}\right) - d\frac{-\eta^2-2}{3\eta}\right) \cdot \mathsf{Cost}^{\ell,k,N}_{\mathsf{CMux}} + \mathsf{Cost}^{N}_{\mathbf{SE}}; \text{ (average).}$$

$$\mathsf{Cost}^{d,\ell,k,N,\nu}_{\mathbf{SBS}-\mathsf{CMS}} = \mathsf{Cost}^{n,2\eta N}_{\mathbf{MS}} + \left(n\left(\frac{2\eta^2+1}{3\eta}\right) - d(\eta-1)\right) \cdot \mathsf{Cost}^{\ell,k,N}_{\mathsf{CMux}} + \mathsf{Cost}^{N}_{\mathbf{SE}}; \text{ (best).}$$

Proof *(Lemma 4.4). This proof follows the same idea as in Proof 16. In Proof 16, we saw that we need to compute $n \cdot \left(\frac{2\eta^2+1}{3\eta}\right)$ CMuxes to perform a blind rotation. With the CMS, we choose d values a_i in $\mathfrak{a}_0$ and we modify them to obtain a_i in $\mathfrak{a}_k$ for some $k \in [1, \xi]$. As we modify d values, we need to subtract the $d \cdot \eta$ CMuxes computed with the original a_i and add the new CMuxes performed with the d modified values to the total number of CMuxes. For these d new a_i, on average, there are $\left(1 - \frac{1}{2}\right) d$ new a_i such that a_i is in $\mathfrak{a}_1$. For these a_i, we need to compute $\eta/2$ CMuxes. After this step, there remain $\frac{1}{2}d$ values and from these values, we have $\left(1 - \frac{1}{2}\right)\frac{1}{2}d$ values such that $a_i \in \mathfrak{a}_2$. For these a_i, we only need to compute $\eta/2^2$ CMuxes, and so on until $a_i \in \mathfrak{a}_\xi$ where we only need to compute one CMux.*

$$\underbrace{n \cdot \left(\frac{2\eta^2+1}{3\eta}\right)}_{I} - d\eta + \sum_{i=0}^{\xi-1}\left(1-\frac{1}{2}\right)\frac{d}{2^i}2^{\xi-i} = I - d\eta + d2^{\xi-1}\sum_{i=0}^{\xi-1}2^{-2i}$$

$$= I - d\eta + d2^{\xi-1}\frac{1-2^{-2\xi}}{1-2^{-2}} = I - d\frac{-\eta^2-2}{3\eta}.$$

In the best case, all the d modified a_i belong to $\mathfrak{a}_\eta$. So, at each step of the blind rotation, only one CMux is required to perform what previously needed η CMuxes. □

4.4 Parallelism to Scale Performance

In [ZYL+18] or [JP22], the authors give some methods to parallelize multiple sequential CMuxes of the blind rotation. In essence, their method uses a bootstrapping key that encrypts cross-products of consecutive secret-key bits. With this larger bootstrapping key, it becomes possible to precompute rotations for multiple mask elements and apply them with a single external product, rather than one per mask element, thereby reducing the overall cost of the bootstrapping operation. An important difference with the traditional external product is that with these methods, the message encrypted in each of the GGSWs is a polynomial and not just a secret key bit. This explain why these techniques cannot be directly applied to the **SBS**. Indeed, one condition for using τ and performing an external product with smaller polynomials is to have a GGSW ciphertext encrypting an integer and not a polynomial (See Section 3.2.1).

To parallelize the CMuxes, we will follow the methods proposed in [BMMP18, LLW+24]. This methods lies in expanding the computation of several CMuxes such that all the operations required

to perform these CMuxes can be executed in parallel with the same individual cost. First, we show how to apply these on the original extended bootstrapping. Second, we detail the required changes to make these techniques compliant with the **SBS**.

4.4.1 More Parallelism for the EBS

In this section, a method is introduced to parallelize multiple sequential CMuxes, while retaining the capability to parallelize each external product with the previously described techniques used in **SBS** or in **EBS**.

First, as in [BMMP18, LLW+24], we observe that two consecutive CMuxes on a GLWE ciphertext LUT to apply the rotation $\tilde{a}_0 s_0$ and $\tilde{a}_1 s_1$ using respectively BSK_{s_0} and BSK_{s_1} (where $\mathsf{BSK}_x \in \mathrm{GGSW}_{\boldsymbol{S}}^{\mathfrak{B},\ell}(x)$ is the bootstrapping key encrypting the secret x) are computed using the following formula:

$$\begin{aligned}&\big(\big(\mathsf{LUT}\cdot X^{\tilde{a}_0}-\mathsf{LUT}\big)\boxdot\mathsf{BSK}_{s_0}+\mathsf{LUT}\big)\boxdot\mathsf{BSK}_{s_1}\\&\quad+\big(\big(\mathsf{LUT}\cdot X^{\tilde{a}_0}-\mathsf{LUT}\big)\boxdot\mathsf{BSK}_{s_0}+\mathsf{LUT}\big)=\mathsf{LUT}\cdot X^{\tilde{a}_0 s_0+\tilde{a}_1 s_1}.\end{aligned}$$

With $\mathsf{BSK}_{s_i} \in \mathrm{GGSW}_{\boldsymbol{S}}^{\mathfrak{B},\ell}(s_i)$ for $i \in \{0,1\}$. This equation can be rewrite like:

$$\begin{aligned}&\big(\mathsf{LUT}\cdot X^{\tilde{a}_1}\big)\boxdot\mathsf{BSK}_{(1-s_0)(s_1)}+\big(\mathsf{LUT}\cdot X^{\tilde{a}_0+\tilde{a}_1}\big)\boxdot\mathsf{BSK}_{(s_0)(s_1)}\\&+\mathsf{LUT}\boxdot\mathsf{BSK}_{(1-s_0)(1-s_1)}+\big(\mathsf{LUT}\cdot X^{\tilde{a}_0}\big)\boxdot\mathsf{BSK}_{(s_0)(1-s_1)}=\mathsf{LUT}\cdot X^{\tilde{a}_0 s_0+\tilde{a}_1 s_1}.\end{aligned}$$

With $\mathsf{BSK}_{(1-s_0)(1-s_1)} \in \mathrm{GGSW}_{\boldsymbol{S}}^{\mathfrak{B},\ell}((1-s_0)(1-s_1))$, $\mathsf{BSK}_{(s_0)(1-s_1)} \in \mathrm{GGSW}_{\boldsymbol{S}}^{\mathfrak{B},\ell}((s_0)(1-s_1))$, and $\mathsf{BSK}_{(1-s_0)(s_1)} \in \mathrm{GGSW}_{\boldsymbol{S}}^{\mathfrak{B},\ell}((1-s_0)(s_1))$, $\mathsf{BSK}_{(s_0)(s_1)} \in \mathrm{GGSW}_{\boldsymbol{S}}^{\mathfrak{B},\ell}((s_0)(s_1))$.

Compared to two sequential CMuxes, this operation requires twice as many CMuxes as the previous equation. However, compared to sequential CMuxes where two CMuxes must be executed sequentially, these four CMuxes can all be performed in parallel. Moreover, each BSK encrypts an integer, allowing each external product to be computed using an **EBS**. Since in a parallel context the **EBS** is faster than a classical **PBS**, this parallelization becomes even more efficient. By generalizing this idea, we obtain the following lemma:

Lemma 4.5. *Let ϖ the number of* CMuxes *performed in parallel with the secret keys $\{s_0,\dots,s_{\varpi-1}\}$, with $s_i \hookleftarrow \mathcal{U}(0,1)$ and the associated mask elements $(\tilde{a}_0,\dots,\tilde{a}_{\varpi-1})$. Let $\mathfrak{S}$ be the set $\{0,\dots,\varpi-1\}$. For each subset $\mathfrak{s}_\mathfrak{r} \subseteq \mathfrak{S}$, we define the bootstrapping key $\mathsf{BSK}_{\mathfrak{s}_\mathfrak{r}}$ which encrypts the secret value $\Pi_{j\in\mathfrak{s}_\mathfrak{r}} s_j \Pi_{i\in\mathfrak{S}\setminus\mathfrak{s}_\mathfrak{r}}(1-s_i)$ for $\mathfrak{r}\in[0,2^\varpi)$. So to perform ϖ* CMuxes *in parallel, we need to have a bootstrapping key 2^ϖ times larger than a traditional* **PBS**. *A rotation of a* GLWE *by $X^{\sum_{i=0}^{\varpi-1}\tilde{a}_i\cdot s_i}$ can be computed using the following formula:*

$$\sum_{\mathfrak{r}=0}^{2^\varpi-1}\mathrm{GLWE}\cdot X^{\sum_{i\in\mathfrak{s}_\mathfrak{r}}\tilde{a}_i}\boxdot\mathsf{BSK}_{\mathfrak{s}_\mathfrak{r}}=\mathrm{GLWE}\cdot X^{\sum_{i=0}^{\varpi-1}\tilde{a}_i\cdot s_i}.$$

And the noise variance of ϖ parallel CMuxes *is:*

$$\begin{aligned}\mathsf{Var}(e)&=2^\varpi\ell(k+1)N\frac{\mathfrak{B}^2+2}{12}\sigma^2_{\mathrm{GGSW}}+\frac{2^\varpi\sigma^2_{\mathrm{GLWE}}}{2}+\frac{1}{16}(1-kN\mathbb{E}(s_i))^2\\&+\frac{q^2-\mathfrak{B}^{2\ell}}{24\mathfrak{B}^{2\ell}}\left(2^\varpi+kN\left(\mathsf{Var}(s_i)+\mathbb{E}^2(s_i)\right)\right)+\frac{kN}{8}\mathsf{Var}(si)+2^{\varpi-4}.\end{aligned}$$

Proof *(Correctness of Lemma 4.5). The product $\Pi_{j\in\mathfrak{s}_\mathfrak{r}} s_j$ is equal to one only if all the secret keys s_j for $j\in\mathfrak{s}_\mathfrak{r}$ are equal to one. The product $\Pi_{i\in\mathfrak{S}\setminus\mathfrak{s}_\mathfrak{r}}(1-s_i)$ is equal to one only if all the secret keys s_i for $i\in\mathfrak{S}\setminus\mathfrak{s}_\mathfrak{r}$ are equal to zero. There exists only one subset $\mathfrak{s}_\mathfrak{r}$ containing all the indices such that all the secret keys equal to one are represented and no secret keys equal to zero are represented. Thus, the subset $\mathfrak{S}\setminus\mathfrak{s}_\mathfrak{r}$ contains only the indices of the secret keys equals to 0. Let us denote this*

subset $\mathfrak{s}$. With this subset, we have $\Pi_{j\in\mathfrak{s}} s_j \Pi_{i\in\mathfrak{S}\setminus\mathfrak{s}}(1-s_i)$ is equal to one. All the other products with the other subsets will be equal to zero. The product $\Pi_{j\in\mathfrak{s}} s_j \Pi_{i\in\mathfrak{S}\setminus\mathfrak{s}}(1-s_i)$ equals one and is associated to the sum $\sum_{j\in\mathfrak{s}} \tilde{a}_j$, which corresponds to the $\tilde{a}_j$ where the secret keys are equal to one. So $\mathrm{GLWE}\cdot X^{\sum_{i\in\mathfrak{s}} \tilde{a}_i} \boxdot \mathsf{BSK}_{\mathfrak{s}}$ is equal to $\mathrm{GLWE}\cdot X^{\sum_{i=0}^{\varpi-1} \tilde{a}_i\cdot s_i}$ all the other products will return a ciphertext encrypting zero. □

Proof *(Noise analysis of Lemma 4.5). The previous operation is done by performing ϖ operations in parallel between a $\mathrm{GLWE}_{\boldsymbol{S}}$ ciphertext encrypted with a noise variance σ^2_{GLWE} and a bootstrapping key $\mathsf{BSK} \in \mathrm{GGSW}^{\mathcal{B},\ell}_{\boldsymbol{S}}$ encrypting a secret key under a noise variance σ^2_{GGSW}. Next, we sum all the result. We know that only one of the secret keys $\mathsf{BSK}_{\mathfrak{s}_\mathfrak{r}}$, for $\mathfrak{r} \in [0, 2^\varpi)$, is equal to one. So for the bootstrapping key that is equal to one we have the following noise variance (Theorem 2.11):*

$$\mathsf{Var}(e_{EP^1}) = \ell(k+1)N\frac{\mathcal{B}^2+2}{12}\sigma^2_{\mathrm{GGSW}} + \frac{\sigma^2_{\mathrm{GLWE}}}{2} + \frac{1}{16}(1-kN\mathbb{E}(s_i))^2 + \frac{q^2-\mathcal{B}^{2\ell}}{24\mathcal{B}^{2\ell}}\left(1+kN\left(\mathsf{Var}(s_i)+\mathbb{E}^2(s_i)\right)\right) + \frac{kN}{8}\mathsf{Var}(si).$$

Where $\mathsf{Var}(e_{EP^1})$ correspond to the variance added by an external product where the secret key is uniformly random in {0,1}.

And for the other external product, where the encrypted secret keys equal zero, we have the noise variance:

$$\mathsf{Var}(e_{EP^0}) = \ell(k+1)N\frac{\mathcal{B}^2+2}{12}\sigma^2_{\mathrm{GGSW}} + \frac{\sigma^2_{\mathrm{GLWE}}}{2} + \frac{1}{16} + \frac{q^2-\mathcal{B}^{2\ell}}{24\mathcal{B}^{2\ell}}.$$

Where, $\mathsf{Var}(e_{EP^0})$ denotes the variance introduced by an external product where the secret key is equal to zero.

Then the variance equals $\mathsf{Var}(e) = \mathsf{Var}(e_{EP^1}) + (2^\varpi - 1)\mathsf{Var}(e_{EP^0})$ □

4.4.2 More Parallelism for the SBS

In the previous section, we have seen how to parallelize groups of CMuxes in a **EBS**. In Section 4.2.1, we introduced the **SBS**. In this section, we will see how to sort the a_i to parallelize the CMuxes and maximize the advantage offered by the sorting.

Lemma 4.6. *By packing $\varpi \cdot$ CMuxes together as defined in Lemma 4.5, and using a* **SBS**, *the cost of the new algorithm is:*

$$\sum_{i=0}^{\xi}\left(1-\frac{1}{2^\varpi}\right)\left(\frac{1}{2^\varpi}\right)^i n2^\varpi\frac{\eta}{2^i} + \frac{1}{\eta^\varpi}\left(\frac{1}{2^\varpi}\right)^{\xi+1} n2^\varpi = 2^\varpi\left[\left(1-\frac{1}{2^\varpi}\right)\eta\frac{1+2^{-(\varpi+1)(\xi+1)}}{1-2^{-(\varpi+1)}} + \eta^{-\varpi}2^{-\varpi\xi}\right].$$

Proof *(Lemma 4.6). The proof follows the one of Lemma 4.2, except that the condition for removing CMuxes needs to be verified by the ϖ consecutive $\tilde{a}_i$ done in parallel. By taking several $\tilde{a}_i$ from different $\mathfrak{a}^k$ for $k \in [0,\xi] \cup \{\eta\}$, we can reduce the numbers of CMuxes as for the $\tilde{a}_i$ in the $\mathfrak{a}$ with the smallest k. Then, the blind rotation is performed as in Lemma 4.2 by parallelizing $\varpi \cdot$ CMuxes at each step of the blind rotation. By applying the same methodology, we obtain the given equation.* □

Remark 4.3. By using the companion modulus switch presented in Lemma 4.3, we can drastically increase the probability that ϖ a_i are on a set $\mathfrak{a}$, which reduces the numbers of external products needed to perform the **SBS**.

Applying this method to the **SBS** is theoretically not faster than grouping ϖ CMuxes to perform an **EBS**, but during computation, it frees some threads, consequently reducing synchronization times between two CMuxes. This results in a small speed-up in addition to freeing threads. When $\varpi = 1$, this operation corresponds to the **SBS** presented in Section 4.2.1.

4.5 Experimental Results

In this section, we describe the experiments conducted. We determined parameter sets for each experiment for a security level $\lambda = 2^{132}$ and failure probabilities: 2^{-40}, 2^{-64}, 2^{-80}, and 2^{-128}. We show experiments for precision ranging from 4 to 9 bits, where the improvements become apparent. As the proposed techniques allow working with large lookup tables while maintaining small polynomial sizes, the improvements become evident when the noise plateau is reached, as described in Definition 20. All parameter sets were determined following the methodology proposed in Section 2.5. When the parameters are the same as those of the previous method, and consequently no improvements are observed, we denote them as "-".

In the presented experiments, we used the same operations as in the atomic pattern CJP [CJP21], which consist of a key switch followed by a bootstrapping. This model served as a reference for benchmarking all other experiments, i.e., we performed a key switch before any bootstrapping. Our main experiments compared the **EBS**, the **SBS** and the **SBS** with the CMS to the baseline **PBS**. This comparison shows that our method consistently outperforms the one proposed in [LY23], offering a speed-up ranging from 1.75 to 8.28 times. To have a full understanding of the impact of each technique, we perform further comparisons, first by taking the **EBS** as the baseline in Table 4.2. Then, to isolate the contribution of the CMS, we set the **SBS** as the baseline and analyze the relative improvements in Table 4.3. Finally, the last comparison in Table 4.4 shows the impact of the sorted bootstrapping compared to the **EBS** both in a parallelized context (16 threads).

For each experiment, we highlight the gains achieved compared to the baseline. The most significant gains are written in bold. The experiments presented in this chapter were conducted on AWS with an `hpc7.96xlarge` instance equipped with an AMD EPYC 9R14 Processor running at 3.7 GHz, 192 vCPUs, and 768 GiB of memory. The experiments utilized the `TFHE-rs` library [Zam22]. All the parameter sets used can be found in the Appendix: Tables A.2, A.5, A.7, and A.10 for the **EBS** and **SBS** and Tables A.3, A.8, and A.11 for the **SBS** with the Companion Modulus Switch.

p_{fail}	Precision	4	5	6	7	8	9
2^{-40}	**PBS** [CJP21]	13.390 ms	38.159 ms	106.920 ms	233.04 ms	517.97 ms	1480.8 ms
	EBS [LY23]	–	25.055 ms	68.579 ms	137.960 ms	279.08 ms	561.890 ms
	Gain		1.52×	1.56×	1.69×	1.86×	2.63×
	SBS	–	20.193 ms	48.499 ms	97.026 ms	196.440 ms	383.390 ms
	Gain		1.89×	2.20×	2.40×	2.64×	**3.86**×
	SBS + CMS	–	19.900 ms	47.606 ms	87.352 ms	168.38 ms	342.51 ms
	Gain		1.92×	2.25×	2.67×	**3.08**×	**4.32**×
2^{-64}	**PBS** [CJP21]	14.016 ms	51.042 ms	112.660 ms	268.560 ms	759.87 ms	3357.2 ms
	EBS [LY23]	–	38.874 ms	75.020 ms	145.620 ms	290.9 ms	601.330 ms
	Gain		1.31×	1.50×	1.84×	2.61×	**5.58**×
	SBS	–	28.335 ms	54.691 ms	101.89 ms	195.860 ms	405.740 ms
	Gain		1.80×	2.06×	2.63×	**3.88**×	**8.28**×
2^{-80}	**PBS** [CJP21]	17.369 ms	103.510 ms	222.900 ms	502.95 ms	1414.4 ms	3500.3 ms
	EBS [LY23]	–	46.752 ms	136.040 ms	266.680 ms	542.300 ms	1118.8 ms
	Gain		2.21×	1.64×	1.89×	2.61×	**3.13**×
	SBS	–	35.428 ms	93.721 ms	178.590 ms	357.620 ms	755.120 ms
	Gain		2.92×	2.38×	2.82×	**3.96**×	**4.64**×
	SBS + CMS	–	29.944 ms	67.443 ms	128.660 ms	256.320 ms	521.76 ms
	Gain		**3.33**×	**3.25**×	**3.69**×	**5.32**×	**6.71**×
2^{-128}	**PBS** [CJP21]	32.858 ms	108.76 ms	256.90 ms	517.01 ms	1441.0 ms	4082.6 ms
	EBS [LY23]	24.808 ms	51.305 ms	135.86 ms	272.92 ms	553.04 ms	1145.0 ms
	Gain	1.32×	2.12×	1.89×	1.90×	2.60×	**3.56**×
	SBS	21.059 ms	37.334 ms	94.098 ms	185.40 ms	376.48 ms	766.65 ms
	Gain	1.56×	2.91×	2.73×	2.79×	**3.83**×	**5.33**×
	SBS + CMS	18.775 ms	37.011 ms	80.879 ms	153.88 ms	320.84 ms	676.720 ms
	Gain	1.75×	2.94×	**3.18**×	**3.36**×	**4.49**×	**6.03**×

Table 4.1: Comparison of **KS-PBS**, **KS-EBS**, **KS-SBS** and **KS-SBS** with CMS, (Base line **KS-PBS**). All parameter sets are in Appendix A.1.1. Parameters used for **PBS** are in Table A.1, A.4, A.6 and A.9. Parameters used for **EBS** and **SBS** are in Table A.2, A.5, A.7 and A.10. Finally, parameters used for **SBS** with Companion Modulus Switch are in Table A.3, A.8 and A.11. All the comparisons were conducted on single thread.

p_{fail}	Precision	4	5	6	7	8	9
2^{-40}	**EBS** [LY23]	-	25.055 ms	68.579 ms	137.960 ms	279.08 ms	561.890 ms
	SBS	-	20.193 ms	48.499 ms	97.026 ms	196.440 ms	383.390 ms
	Gain		1.24×	1.41×	1.42×	1.42×	1.47×
	SBS + CMS	-	19.900 ms	47.606 ms	87.352 ms	168.38 ms	342.51 ms
	Gain		1.26×	1.44×	**1.58**×	**1.66**×	**1.64**×
2^{-64}	**EBS** [LY23]	-	38.874 ms	75.020 ms	145.620 ms	290.9 ms	601.330 ms
	SBS	-	28.335 ms	54.691 ms	101.89 ms	195.860 ms	405.740 ms
	Gain		1.37×	1.37×	1.43×	1.48×	1.48×
2^{-80}	**EBS** [LY23]	-	46.752 ms	136.040 ms	266.680 ms	542.300 ms	1118.800 ms
	SBS	-	35.428 ms	93.721 ms	178.590 ms	357.620 ms	755.120 ms
	Gain		1.32×	1.45×	1.49×	**1.51**×	1.48×
	SBS + CMS	-	29.944 ms	67.443 ms	128.660 ms	256.320 ms	521.76 ms
	Gain		**1.56**×	**2.02**×	**2.07**×	**2.11**×	**2.14**×
2^{-128}	**EBS** [LY23]	24.808 ms	51.305 ms	135.860 ms	272.920 ms	553.040 ms	1145.000 ms
	SBS	21.059 ms	37.334 ms	94.098 ms	185.40 ms	376.480 ms	766.650 ms
	Gain	1.18×	1.37×	1.44×	1.47×	1.47×	**1.50**×
	SBS + CMS	18.775 ms	37.011 ms	80.879 ms	153.88 ms	320.84 ms	676.720 ms
	Gain	1.32×	1.39×	**1.52**×	**1.53**×	**1.55**×	**1.53**×

Table 4.2: Comparison of **KS-EBS**, **KS-SBS** and **KS-SBS** with CMS, (Base line **KS-EBS**). All parameter sets are in Appendix A.1.1. Parameters used for **EBS** and **SBS** are in Table A.2, A.5, A.7 and A.10. Finally, parameters used for **SBS** with Companion Modulus Switch are in Appendix A.1.1 in Table A.3, A.8 and A.11. All the comparisons were conducted on single thread.

p_{fail}	Precision	4	5	6	7	8	9
2^{-40}	**SBS**	–	20.193 ms	48.499 ms	97.026 ms	196.440 ms	383.390 ms
	SBS + CMS	–	19.900 ms	47.606 ms	87.352 ms	168.38 ms	342.51 ms
	Gain		1.02×	1.02×	1.11×	1.17×	1.12×
2^{-80}	**SBS**	–	35.428 ms	93.721 ms	178.590 ms	357.620 ms	755.120 ms
	SBS + CMS	–	29.944 ms	67.443 ms	128.660 ms	256.320 ms	521.76 ms
	Gain		1.18×	**1.39**×	**1.39**×	**1.40**×	**1.45**×
2^{-128}	**SBS**	21.059 ms	37.334 ms	94.098 ms	185.40 ms	376.480 ms	766.650 ms
	SBS + CMS	18.775 ms	37.011 ms	80.879 ms	153.88 ms	320.84 ms	676.720 ms
	Gain	1.12×	1.01×	1.16×	**1.20**×	1.17×	1.13×

Table 4.3: Comparison of **KS-SBS** with **KS-SBS** with CMS, (Baseline: **KS-SBS**). All parameter sets are in Appendix A.1.1. Parameters used for **SBS** are in Table A.2, A.5, A.7 and A.10. Finally, parameters used for **SBS** with Companion Modulus Switch are in Table A.3, A.8 and A.11. All the comparisons were conducted on single thread.

p_{fail}	Precision	2	3	4	5	6	7	8	9
2^{-40}	**EBS** [LY23]	27.874 ms	33.930 ms	37.006 ms	39.197 ms	41.012 ms	57.405 ms	62.474 ms	120.13 ms
	SBS	21.502 ms	26.990 ms	30.018 ms	30.920 ms	33.174 ms	57.543 ms	54.484 ms	116.94 ms
	Gain	1.30×	1.26×	1.23×	1.27×	1.24×	1.00×	1.15×	1.03×
2^{-64}	**EBS** [LY23]	28.861 ms	34.749 ms	42.066 ms	39.979 ms	44.540 ms	67.070 ms	81.859 ms	152.42 ms
	SBS	22.269 ms	28.012 ms	34.802 ms	36.764 ms	35.854 ms	46.369 ms	59.621 ms	142.10 ms
	Gain	1.30×	1.24×	1.21×	1.09×	1.24×	**1.45**×	1.37×	1.07×
2^{-80}	**EBS** [LY23]	29.555 ms	40.686 ms	37.847 ms	40.542 ms	49.049 ms	69.116 ms	115.97 ms	224.86 ms
	SBS	22.848 ms	28.025 ms	35.331 ms	38.197 ms	41.721 ms	62.723 ms	111.73 ms	210.86 ms
	Gain	1.29×	**1.45**×	1.07×	1.06×	1.17×	1.10×	1.04×	1.07×
2^{-128}	**EBS** [LY23]	31.260 ms	42.073 ms	43.992 ms	40.363 ms	56.615 ms	62.079 ms	161.17 ms	276.29 ms
	SBS	23.992 ms	29.039 ms	30.958 ms	32.065 ms	56.789 ms	53.287 ms	151.28 ms	266.99 ms
	Gain	**1.30**×	**1.45**×	**1.42**×	1.26×	1.00×	1.17×	1.07×	1.03×

Table 4.4: Comparison of the parallelized version of the **EBS** with the parallelized version **SBS**, (Baseline: **EBS**). All parameter sets are in Appendix A.1.1. Parameters used for parallelized **EBS** and parallelized **SBS** are in Table A.12, A.13, A.14 and A.15. All experiments were conducted on 16 threads.

Chapter 5

New Secret Keys for Enhanced Performance in TFHE

In Chapter 2, we introduced all the ciphertext types necessary to define the TFHE scheme. In particular, we showed that, for both security and efficiency reasons, TFHE operates with polynomials in the ring $\mathfrak{R}_{q,N}$. As a result, when selecting parameter sets for GLWE or GGSW ciphertexts, the polynomial degree must be a power of two, and intermediate polynomial sizes cannot be used.

In this chapter, we introduce two new types of secret key distributions. The first allows computations within the ring $\mathfrak{R}_{q,N}$ while using secret keys where the number of unknown coefficients is not necessarily a power of two. The second type enables the sharing of certain coefficients between different keys. These new distributions lead to the development of new algorithms and improved noise propagation, resulting in an overall improvement to the TFHE scheme. In this chapter, we study in detail the security and performance enhancements offered by these new key distributions.

5.1 Introduction

In this chapter, we introduce two novel types of secret keys. We begin by describing each key independently, and then demonstrate how they can be combined together. We call these two new distributions *partial secret keys* and *secret keys with shared randomness* . The first kind, the partial secret keys, consists of allowing a GLWE secret key, traditionally containing kN random elements (sampled from a distribution $\mathfrak{D}$), to contain only ϕ random elements and setting the rest to zeros. Intuitively, this technique enables the use of a smaller secret key of size ϕ, while keeping a large polynomial degree N. This allows bootstrapping of higher precision messages with a better noise management. As a result, the security of the GLWE assumption now relies on the parameter ϕ instead of the dimension N. This new secret key is detailed in Section 5.2

The second kind, the secret keys with shared randomness, consists of reusing the randomness from a larger key (for example the input key of the key switch algorithm) inside a smaller key (its output), instead of generating it independently as done traditionally. For instance, we can consider two integers $1 < n_0 < n_1$ and a secret key $\boldsymbol{s}^{(1)} \in \mathbb{Z}_q^{n_1}$ generated in the traditional manner (either sampled from a uniform binary/ternary, or a small Gaussian). Let us write it as a concatenation of two vectors: $\boldsymbol{s}^{(1)} = \boldsymbol{r}^{(0)}||\boldsymbol{r}^{(1)}$. We can now build a smaller secret key out of $\boldsymbol{s}^{(1)}$ such that the smaller one will be included in the larger one, in its first coefficients: $\boldsymbol{s}^{(0)} = \boldsymbol{r}^{(0)} \in \mathbb{Z}_q^{n_0}$ and $\boldsymbol{s}^{(1)} = \boldsymbol{r}^{(0)}||\boldsymbol{r}^{(1)} \in \mathbb{Z}_q^{n_1}$. This second new secret key is then presented in Section 5.3.

Finally, in Section 5.4, we study the impact of combining secret keys with shared randomness with partial secret keys. All our contributions reduce the size of the secret key, and then have a significant impact on the size of public keys, including key-switching keys and bootstrapping keys. It also has an impact on the operations that use these keys, i.e., key-switching and bootstrapping.

5.2 Partial GLWE Secret Key

As presented in the introduction, the partial GLWE secret key is composed of two parts, the first one contains random secret elements (sampled from a distribution $\mathcal{D}$) and the second part is filled with *zeros at known positions*. As a simple example, we can define the following partial GLWE secret key: $\boldsymbol{S} = (S_0, S_1) \in \mathfrak{R}^2_{q,N}$ with $S_0 = \sum_{j=0}^{N-1} s_{0,j} X^j$ and $S_1 = \sum_{j=0}^{N/2-1} s_{1,j} X^j$ where $s_{0,0}, \cdots, s_{0,N-1}$ and $s_{1,0}, \cdots, s_{1,\frac{N}{2}-1}$ are sampled from $\mathcal{D}_S$, and the other coefficients are publicly known to be set to zero. We recall the two limitations of TFHE (already mentioned in Section 2.6):

1. There is *no fine-grained control over the size of a GLWE secret key*, it is of the form kN with N a power of two;
2. When one increases n (or kN), a plateau in terms of noise variance is reached. Concretely, n_{plateau} is the first value of this plateau i.e., for larger value of n, the minimal standard deviation of the noise is constant. We evaluated its value using the lattice estimator (Subsection 2.1.4) to be 2443 for 128 bits of security and $q = 2^{64}$.

Thanks to these new types of secret keys, these limitations can be overcome. These new keys help reduce noise growth during certain operations (for instance, the external product or the key switch) over RLWE and GLWE ciphertexts, either due to the presence of known zeros in the secret keys or the reduction in computation steps when secret keys are reused. Since bootstrapping is a chain of external products, partial secret keys will also be beneficial to its noise growth. After the sample extract, we can discard the mask elements associated with the positions of the zeros resulting in a smaller LWE dimension compared to traditional secret keys. This smaller LWE dimension will likely improve the cost of the next key switch.

In this section, we first formally define the notion of partial secret key, and then study the hardness of the underlying problem. Finally, we list the different advantages and improvements they offer.

Definition 29 (GLWE Partial Secret Key)**.** *A Partial GLWE secret key is a vector $\boldsymbol{S}^{[\phi]} \in \mathfrak{R}^k_{q,N}$ associated with its filling amount ϕ such that $0 \leq \phi \leq kN$. This key will have ϕ random coefficients sampled from a distribution $\mathcal{D}_S$ and $kN - \phi$ known zeros. Both the locations of the random elements and the zeros are public. By convention, the coefficients start at coefficient $s_{0,0}$, then $s_{0,1}$ and so on. When the first polynomial is entirely filled, the second polynomial starts with $s_{1,0}$ and so on, until ϕ coefficients are determined, up to $s_{k-1,N-1}$.*

We now define the flattened representation of a Partial GLWE Secret Key.

Definition 30 (Flattened Representation of a Partial GLWE Secret Key)**.** *A partial GLWE secret key $\boldsymbol{S}^{[\phi]} = \left(S_0 = \sum_{j=0}^{N-1} s_{0,j} X^j, \cdots, S_{k-1} = \sum_{j=0}^{N-1} s_{k-1,j} X^j\right) \in \mathfrak{R}^k_{q,N}$ (Definition 29) can be viewed as a flattened LWE secret key $\bar{\boldsymbol{s}} = (\bar{s}_0, \cdots, \bar{s}_{\phi-1}) \in \mathbb{Z}^\phi$ in the following manner: $\bar{s}_{iN+j} := s_{i,j}$, for $0 \leq j < N$ and $0 \leq i < k$ with $iN + j < \phi$. This flattened representation contains only ϕ unknown coefficients.*

5.2.1 Hardness of Partial GLWE

The GLWE partial secret key problem $\boldsymbol{S}^{[\phi]} \in \mathfrak{R}^k_{q,N}$ from Definition 29, seems to be at least as hard as a GLWE problem in a ring of dimension ϕ. First, we present the GLWE alternate partial secret key, a key where the secret elements are separated by $2^\nu - 1$ known zeros. We prove the security of a such secret key distribution by proving that the GLWE problem in $\mathfrak{R}^{k+1}_{q,N/2^\nu}$ is equivalent to the GLWE problem in $\mathfrak{R}^{k+1}_{q,N}$ instantiated with alternate partial GLWE secret keys, this problem is denoted P-GLWE$_{\nu,q,k,N,\chi}$.

Definition 31 (Alternate Partial GLWE Secret Key)**.** *An alternate partial* GLWE *secret, is a* GLWE *secret where the key alternates between one unknown element and $2^\nu - 1$ known elements.*

This key has of $\frac{N}{2^\nu}$ random coefficients sampled from a distribution $\mathcal{D}_S$ and $N - \frac{N}{2^\nu}$ known zero coefficients. As for the partial GLWE *secret key (Definition 29), both the locations of the random elements and the known zeros are public. The binary version of partial secret keys in $\mathfrak{R}_{q,N}$ is defined by $S = \sum_{k=0}^{N/2^\nu-1} s_k \cdot X^{k\cdot 2^\nu}$, with $s_i \hookleftarrow \mathcal{U}(\{0,1\})$.*

Theorem 5.1 shows that the alternate partial GLWE problem (Definition 31) on the ring $\mathfrak{R}_{q,N}$ is at least as hard as the GLWE problem on the ring $\mathfrak{R}_{q,N/2^\nu}$.

Theorem 5.1 (Hardness of P-GLWE)**.** *For any $\nu \in \mathbb{Z}$, the* P-GLWE$_{\nu,q,k,N,\chi}$ *problem with samples in $\mathfrak{R}_{q,N}^{k+1}$ is as least as hard than solving the $GLWE_{N/2^\nu,k,\chi}$ problem with 2^ν samples in $\mathfrak{R}_{q,N/2^\nu}^{k+1}$.*

Proof *(Theorem 5.1). The idea of this proof is to pack 2^ν* GLWE$_{N/2^\nu,k,\chi}$ *samples in one* P-GLWE$_{\nu,q,k,N,\chi}$ *sample. To do so, we differentiate the 2^ν samples from* GLWE$_{N/2^\nu,k,\chi}$ *in $\mathfrak{R}_{N/2^\nu,q}^{k+1}$, by noting them* GLWE$^w_{\boldsymbol{S}(X)}$ *with $w \in [0, 2^\nu)$. Observe that all of them are encrypted under the same secret key $\boldsymbol{S} = (S_0, \ldots, S_{k-1}) \in \mathfrak{R}_{q,N/2^\nu}^k$, with $S_i = \sum_{j=0}^{N/2^\nu-1} s_{i,j} X^j$, with $i \in [0, k)$.*

Each one of the k polynomials composing the GLWE$^w_{\boldsymbol{S}(X)}$ *sample is noted with an exponent w: $A_i^w = \sum_{j=0}^{N/2^\nu-1} a_{i,j}^w X^j$, with $i \in [0, k)$. Starting from these 2^ν samples, we define a new sample from* P-GLWE$_{\nu,q,k,N,\chi}$*. First, for each sample* GLWE$^w_{\boldsymbol{S}(X)} \in \mathfrak{R}_{q,N/2^\nu}^{k+1}$*, we need to evaluate each polynomial in X^ν:*

$$\begin{aligned} \mathfrak{R}_{q,N/2^\nu} &\longrightarrow \mathfrak{R}_{q,N}, \\ A_i^w(X) = \sum_{j=0}^{N/2^\nu-1} a_{i,j}^w \cdot X^j &\longmapsto \widetilde{A_i^w(X)} = \sum_{j=0}^{N/2^\nu-1} a_{i,j}^w \cdot X^{j\cdot 2^\nu}. \end{aligned}$$

So, for each sample GLWE$^w_{\boldsymbol{S}(X)}$ *in $\mathfrak{R}_{q,N/2^\nu}^{k+1}$, we obtain a new sample $\widetilde{\text{GLWE}}^w_{\boldsymbol{S}(X^{2^\nu})}$ in $\mathfrak{R}_{q,N/2}^{k+1}$. We notice that for each polynomial, each coefficient is separated from the other by $2^\nu - 1$ zeros. Following the previous definition of* P-GLWE *(Definition 31), the secret key is in the desired shape. But the $\widetilde{A_i^w(X)}$ polynomials are not uniform anymore, only the coefficients of degree multiple of 2^ν are. So we can't already define $\widetilde{\text{GLWE}}^w_{\boldsymbol{S}(X^{2^\nu})}$ as a sample of* P-GLWE$_{\nu,q,k,N,\chi}$*. For each $\widetilde{\text{GLWE}}^w_{\boldsymbol{S}(X^{2^\nu})}$ we now rotate all the A_i^w and the B^w polynomials by X^w: We now sum all of them together to obtain the expected sample from* P-GLWE$_{\nu,q,k,N,\chi} \in \mathfrak{R}_{q,N}^{k+1}$*:*

$$\begin{aligned} &\sum_{w=0}^{2^\nu-1} \left(A_0^w(X^{2^\nu}) X^w, \ldots, A_{k-1}^w(X^{2^\nu}) X^w, B^w(X^{2^\nu}) X^w \right) \\ &\qquad = (A_0, \ldots, A_{k-1}, B) \in \text{P-GLWE}_{\nu,q,k,N,\chi}. \end{aligned}$$

with:

$$\begin{aligned} S_i &= \sum_{j=0}^{N/2^\nu-1} s_{i,j} \cdot X^{j\cdot 2^\nu} = \sum_{j=0}^{N-1} \tilde{s}_{i,j} \cdot X^j \text{ for } i \in [0,k) \\ A_i &= \sum_{w=0}^{2^\nu-1} \widetilde{A_i^w(X)} X^w = \sum_{w=0}^{2^\nu-1} \sum_{j=0}^{N/2^\nu-1} a_{i,j}^w X^{j\cdot 2^\nu + w} = \sum_{j=0}^{N-1} \tilde{a}_{i,j} X^j \text{ for } i \in [0,k) \\ B &= \sum_{w=0}^{2^\nu-1} B^w(X^{2^\nu}) X^w = \sum_{w=0}^{2^\nu-1} \sum_{j=0}^{N/2^\nu-1} b_j^w X^{j\cdot 2^\nu + w} = \sum_{j=0}^{N-1} \tilde{b}_j X^j. \end{aligned}$$

Let us focus on how b_j^w evolve all along the reduction:

$$b_j^w = \sum_{i=0}^{k-1} \left(\sum_{\tau=0}^{j} a_{i,\tau}^w \cdot s_{i,j-\tau} - \sum_{\tau=j+1}^{N/2^\nu-1} a_{i,\tau}^w \cdot s_{i,N+j-\tau} \right) + e_j^w$$
$$= \sum_{i=0}^{k-1} \left(\sum_{\tau=0}^{j2^\nu+w} \tilde{a}_{i,\tau} \cdot \tilde{s}_{i,j2^\nu+w-\tau} - \sum_{\tau=j2^\nu+w+1}^{N-1} \tilde{a}_{i,\tau} \cdot \tilde{s}_{i,2^\nu(N+j)+w-\tau} \right) + \tilde{e}_{j2^\nu+w} = \tilde{b}_{j2^\nu+w}.$$

Each coefficient is correctly decrypted and each $\tilde{b}_{j2^\nu+w}$ is equal to b_j^w. Moreover, the polynomials A_i of the new P-GLWE$_{\nu,q,k,N,\chi}$ *sample follows the same distribution as the polynomials A_i^w (resp. B). To conclude, we have packed several* GLWE$_{N/2^\nu,k,\chi}$ *ciphertext in one* P-GLWE$_{N,k,\chi}$ *ciphertext by increasing the dimension of this new ciphertext without changing the noise distribution χ.* □

Remark 5.1 (Security of Partial Secret Key)**.** The reduction presented in Theorem 5.1 proves that the partial alternate secret keys (Definition 31) problem in $\mathfrak{R}_{q,N}^k$ is at least as hard as a GLWE problem in $\mathfrak{R}_{q,N/2^\nu}^k$, i.e., when $\phi = N/2^\nu$. So adding zeros at specific places in the secret key and increasing the dimension from $N/2^\nu$ to N allows keeping the same security level. We assume this result is generalizable to any $\phi < N$.

Now, if we take two GLWE samples such that the first one is encrypted under an alternate partial key (Definition 31) and the second one is encrypted under a secret partial key (Definition 29) which have the same amount of unknown coefficients, this two samples should be indistinguishable.

We recall that in LWE samples, the security depends on the dimension and the noise (increasing one could allow to reduce the other one). Intuitively, the security of GLWE samples, encrypted under a partial key with ϕ random elements, is linked to the relation of ϕ and the noise σ (instead of N and σ). A larger ϕ will lead to a smaller noise σ. To sum up, to guarantee a given level of security for GLWE samples encrypted under a partial secret key with ϕ random elements, we use the noise parameter given for LWE samples of dimension $n = \phi$ with the same level of security.

Impact of Partial Key on the Noise Distribution. Regarding the security of the partial secret key and the different attacks presented in Section 2.1.3, we can use the lattice estimator to find out the smallest noise variance σ^2 for an LWE $\in \mathbb{Z}_q^{\phi+1}$ guarantying the desired level of security λ.

5.2.2 Algorithm with Partial GLWE Secret Keys

Partial GLWE secret keys enable to reduce the computational cost and/or the noise growth for certain algorithms. For a given failure probability and security level, the parameter sets obtained after optimization will lead to better timings for the functionality (more details in Section 5.5).

In what follows, we describe all the advantages of using partial secret keys by first introducing a variant relying on the GLWE-to-GLWE key switching(Algorithm 30) and a second one relying on the secret product GLWE-to-GLWE key switching (Algorithm 31). Moreover, partial GLWE secret keys can be used to design a new and more efficient LWE-to-LWE key switching that is FFT-based (Algorithm 32). The idea is an adaptation of [CDKS20] but now exploits the use of partial GLWE secret keys. First we cast the input LWE ciphertext into a GLWE ciphertext (Algorithm 29) so we can apply a GLWE-to-GLWE key switching to go to a partial GLWE secret key. This leverages the speed-up coming from the FFT. Finally, we compute a sample extract (Algorithm 28). We then detail each step of the Algorithm 32, which is studied more in details in Section 5.2.2.5.

5.2.2.1 Sample Extract with Partial GLWE Secret Keys

In Section 2.4, we introduced the sample extract algorithm (Algorithm 10). In the case of partial GLWE secret keys, the algorithm only needs to extract the mask coefficient corresponding to unknown secret key coefficients. The complete algorithm is described in 28, and can trivially be

adapted in the context of partial GLWE secret keys. They are generalizations of the same algorithm used for "traditional" secret keys. Indeed, a traditional secret key is captured when $\phi = k \times N$.

Algorithm 28: $\mathsf{ct_{out}} \leftarrow \mathsf{ConstantSampleExtract}(\mathsf{CT_{in}})$

Context:
$$\begin{cases} \boldsymbol{S}^{[\phi]} \in \mathfrak{R}_{q,N}^{k} : \text{a partial secret key (Definition 29)} \\ (k-1)N+1 \leq \phi \leq kN : \text{filling amount of the partial secret key} \\ \bar{\boldsymbol{s}} \in \mathbb{Z}^{\phi} : \text{the flattened version of } \boldsymbol{S}^{[\phi]} \text{ (Definition 30)} \\ M = \sum_{i=0}^{N-1} m_i X^i \in \mathfrak{R}_{p,N} \\ \mathsf{CT_{in}} = \left(\sum_{i=0}^{N-1} a_{0,i} X^i, \ldots, \sum_{i=0}^{N-1} a_{k-1,i} X^i, \sum_{i=0}^{N-1} b_i X^i\right) \subseteq \mathfrak{R}_{q,N}^{k+1} \end{cases}$$

Input: $\mathsf{CT_{in}} \in \mathrm{GLWE}_{\boldsymbol{S}^{[\phi]}}(M)$: a GLWE encryption of the plaintext M

Output: $\mathsf{ct_{out}} \in \mathrm{LWE}_{\bar{\boldsymbol{s}}}(m_0)$: an LWE encryption of the plaintext m_0

for $i \in [0; \phi - 1]$ **do**
  set $\alpha := \left\lfloor \frac{i}{N} \right\rfloor$, $\beta := (N - i) \mod N$ and $\gamma := 1 - (\beta == 0)$
  set $a_{\mathsf{out},i} := (-1)^{\gamma} \cdot a_{\alpha,\beta}$
return $\mathsf{ct_{out}} := (a_{\mathsf{out},0}, \ldots, a_{\mathsf{out},\phi-1}, b_0) \subseteq \mathbb{Z}_q^{\phi+1}$

Remark 5.2. The correctness proof of Algorithm 28 is similar to the proof of Theorem 2.14, of the sample extraction algorithm (Algorithm 10).

Noise and Cost of Sample Extract A sample extract, whether it includes a partial secret key or not, does not add any noise to the plaintext. The cost is also roughly the same and remains negligible.

Inverse Constant Sample Extract An LWE ciphertext of size $n+1$ can trivially be cast into a GLWE ciphertext of size $k+1$ and with polynomials of size N. For completeness, the process is detailed in Alg. 29.

We obviously need $n \leq kN$. If $n = kN$, the output is a GLWE ciphertext under a traditional secret key, otherwise it is a GLWE ciphertext under a partial GLWE secret key. Note that the constant term of the output GLWE plaintext is exactly the plaintext of the input LWE ciphertext, however the rest of the coefficients of the output GLWE ciphertext are filled with uniformly random values. We have the property that for all $m \in \mathbb{Z}_p$, for all $\boldsymbol{s} \in \mathbb{Z}_q^n$, for all $\mathsf{ct} \in \mathrm{LWE}_{\boldsymbol{s}}(p) \subseteq \mathbb{Z}_q^{n+1}$ and for all $(k, N) \in \mathbb{N}^2$ s.t. $n \leq kN$:

$$\mathsf{ct} = \mathsf{ConstantSampleExtract}\left(\mathsf{ConstantSampleExtract}^{-1}(\mathsf{ct}, k, N)\right).$$

5.2.2.2 Key Switch with Partial GLWE Secret Key

A GLWE-to-GLWE key switching with $N > 1$, as described in Algorithm 30 takes as input a GLWE ciphertext $\mathsf{CT_{in}} \in \mathfrak{R}_{q,N}^{k_{\mathsf{in}}+1}$ encrypting the plaintext $M \in \mathfrak{R}_{p,N}$ under the secret key $\boldsymbol{S}^{[\phi_{\mathsf{in}}]} \in \mathfrak{R}_{q,N}^{k_{\mathsf{in}}}$, and outputs $\mathsf{CT_{out}} \in \mathfrak{R}_{q,N}^{k_{\mathsf{out}}+1}$ encrypting the plaintext $\Delta M + E_{\mathsf{KS}} \in \mathfrak{R}_{q,N}$ under the secret key $\boldsymbol{S}^{[\phi_{\mathsf{out}}]} \in \mathfrak{R}_{q,N}^{k_{\mathsf{out}}}$. The noise E_{KS} added during this procedure, is composed of a rounding error plus a linear combination of the noise from the key switching key ciphertexts. The larger ϕ_{in}, the more significant the rounding error.

Theorem 5.2 (Noise of GLWE Key Switch)**.** *Let* $\mathsf{CT_{in}} \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]}}(M) \subseteq \mathfrak{R}_{q,N}^{k_{\mathsf{in}}+1}$ *be a GLWE ciphertext encrypting the message* $M \in \mathfrak{R}_{p,N}$ *under the partial GLWE secret key* $\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} = (S_{\mathsf{in},0}, \ldots, S_{\mathsf{in},k_{\mathsf{in}}-1}) \in \mathfrak{R}_{q,N}^{k_{\mathsf{in}}}$*. Let* $\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]} \in \mathfrak{R}_{q,N}^{k_{\mathsf{out}}}$ *be an output partial GLWE secret key. Let* $\mathfrak{B} \in \mathbb{Z}^*$ *be the base decomposition and* $\ell \in \mathbb{Z}^*$ *the level decomposition. Let*

Algorithm 29: $\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{ConstantSampleExtract}^{-1}(\mathsf{ct}_{\mathsf{in}}, k, N)$

Context:
$\begin{cases} \boldsymbol{s} \in \mathbb{Z}_q^n : \text{ the input LWE secret key} \\ \boldsymbol{S}^{[n]} \in \mathfrak{R}_{q,N}^k : \text{a partial secret key (Definition 29)} \\ \qquad \text{such that its flattened version is } \boldsymbol{s} \text{ (Definition 30)} \\ R := \sum_{i=1}^{N-1} r_i \cdot X^i \in \mathfrak{R}_{q,N}, \text{ where } r_i \text{ are uniformly random} \\ \mathsf{ct}_{\mathsf{in}} = (a_0, \ldots, a_{n-1}, b) \subseteq \mathbb{Z}_q^{n+1} \\ p \in \mathbb{Z}_q \end{cases}$

Input:
$\begin{cases} \mathsf{ct}_{\mathsf{in}} \in \mathrm{LWE}_{\boldsymbol{s}}(m) : \text{an LWE encryption of the plaintext } m \\ k \in \mathbb{N} : \text{ the output GLWE dimension} \\ N \in \mathbb{N} : \text{ the output polynomial size} \end{cases}$

Output: $\mathsf{CT}_{\mathsf{out}} \in \mathrm{GLWE}_{\boldsymbol{S}^{[n]}}(m + R)$: a GLWE encryption

`/* put the` b `part in a polynomial */`

1 set $B' := b \in \mathfrak{R}_{q,N}$

`/* put the rest in polynomials */`

2 **for** $i \in [0; k \cdot N]$ **do**

3 set $\alpha := \left\lfloor \frac{i}{N} \right\rfloor$, $\beta := (N - i) \mod N$ and $\gamma := 1 - (\beta == 0)$

4 **if** $i \leq \phi - 1$ **then**

5 set $a'_{\alpha,\beta} := (-1)^\gamma \cdot a_i$

6 **else**

7 set $a'_{\alpha,\beta} := 0$

8 **return** $\mathsf{CT}_{\mathsf{out}} := \left(A'_0 := \sum_{j=0}^{N-1} a'_{0,j} X^j, \quad \cdots \quad, A'_{k-1} := \sum_{j=0}^{N-1} a'_{k-1,j} X^j, \; B'\right) \subseteq \mathfrak{R}_{q,N}^{k+1}$

Algorithm 30: $\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{GLWEKeySwitch}(\mathsf{CT}_{\mathsf{in}}, \mathsf{KSK})$

Context:
$\begin{cases} \boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} \in \mathfrak{R}_{q,N}^{k_{\mathsf{in}}} : \text{the input partial secret key (Definition 29)} \\ \boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} = (S_{\mathsf{in},0}, \ldots, S_{\mathsf{in},k_{\mathsf{in}}-1}) \\ \boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]} \in \mathfrak{R}_{q,N}^{k_{\mathsf{out}}} : \text{the output partial secret key (Definition 29)} \\ (k_{\mathsf{in}} - 1)N < \phi_{\mathsf{in}} \leq k_{\mathsf{in}} N \text{ and } (k_{\mathsf{out}} - 1)N < \phi_{\mathsf{out}} \leq k_{\mathsf{out}} N \\ \ell \in \mathbb{N} : \text{ the number of levels in the decomposition} \\ \mathfrak{B} \in \mathbb{N} : \text{ the base in the decomposition} \end{cases}$

Input:
$\begin{cases} \mathsf{CT}_{\mathsf{in}} \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]}}(M) \subseteq \mathfrak{R}_{q,N}^{k_{\mathsf{in}}+1}, \text{ with } M \in \mathfrak{R}_{p,N} \\ \mathsf{KSK} = \{(\mathsf{KSK}_0, \ldots, \mathsf{KSK}_{k_{\mathsf{in}}-1})\}\text{; } \texttt{/* Definition 15 */} \\ \text{With } \mathsf{KSK}_i \in \mathrm{GLEV}_{\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}}^{\mathfrak{B},\ell}(S_{\mathsf{in},i}^{[\phi_{\mathsf{in}}]}) \text{ for } i \in [0, k_{\mathsf{in}} - 1] \end{cases}$

Output: $\mathsf{CT}_{\mathsf{out}} \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}}(M) \subseteq \mathfrak{R}_{q,N}^{k_{\mathsf{out}}+1}$

1 Set $\mathsf{CT}_{\mathsf{out}} := (0, \ldots, 0, B) \in \mathfrak{R}_{q,N}^{k_{\mathsf{out}}+1}$

2 **for** $i \in [0; k_{\mathsf{in}} - 1]$ **do**

`/* Decompose the mask */`

3 Update $\mathsf{CT}_{\mathsf{out}} = \mathsf{CT}_{\mathsf{out}} - \left\langle \mathsf{KSK}_i, \mathbf{Dec}^{(\mathfrak{B},\ell)}(A_i) \right\rangle$

4 **return** $\mathsf{CT}_{\mathsf{out}}$

$\mathsf{KSK} = \left\{\mathsf{KSK}_i \in \mathrm{GLEV}^{\mathcal{B},\ell}_{\boldsymbol{S}^{[\phi_{\mathsf{out}}]}_{\mathsf{out}}}(S_{\mathsf{in},i})\right\}_{i\in[0,k_{\mathsf{in}}-1]} \in \mathfrak{R}^{(k_{\mathsf{out}}+1)\cdot\ell\cdot k_{\mathsf{in}}}_{q,N}$ *be the key switching key as presented in Definition 15.*

Then Algorithm 4 outputs $\mathsf{CT}_{\mathsf{out}} \in \mathrm{GLWE}_{\boldsymbol{S}^{[\phi_{\mathsf{out}}]}_{\mathsf{out}}}(m) \subseteq \mathfrak{R}^{k_{\mathsf{out}}+1}_{q,N}$ *a GLWE ciphertext encrypting the input message* M *under the secret key* $\boldsymbol{S}^{[\phi_{\mathsf{out}}]}_{\mathsf{out}}$*. The variance of the noise of each coefficient of the output can be estimated by:*

$$\begin{aligned}\mathsf{Var}\left(\mathsf{CT}_{\mathsf{out}}\right) = \sigma^2_{\mathsf{in}} + \phi_{\mathsf{in}}\left(\frac{q^2-\mathcal{B}^{2\ell}}{12\mathcal{B}^{2\ell}}\right)\left(\mathsf{Var}\left(\boldsymbol{S}^{[\phi_{\mathsf{in}}]}_{\mathsf{in}}\right) + \mathbb{E}^2\left(\boldsymbol{S}^{[\phi_{\mathsf{in}}]}_{\mathsf{in}}\right)\right)\\ + \frac{\phi_{\mathsf{in}}}{4}\mathsf{Var}\left(\boldsymbol{S}^{[\phi_{\mathsf{in}}]}_{\mathsf{in}}\right) + \ell k_{\mathsf{in}} N\sigma^2_{\mathsf{ksk}}\frac{\mathcal{B}^2+2}{12}\,.\end{aligned}$$

Proof *(Theorem 5.2). The inputs of a* GLWE-*to*-GLWE *key switching (Algorithm 30) are:*

- *The input GLWE ciphertext:* $\mathsf{CT}_{\mathsf{in}} = (\boldsymbol{A}_{\mathsf{in}}, B_{\mathsf{in}}) \in \mathrm{GLWE}_{\boldsymbol{S}^{[\phi_{\mathsf{in}}]}_{\mathsf{in}}}(\Delta\cdot M) \subseteq \mathfrak{R}^{k_{\mathsf{in}}+1}_{q,N}$*, where* $B_{\mathsf{in}} = \sum_{i=0}^{k_{\mathsf{in}}-1} A_{\mathsf{in},i}\cdot S_{\mathsf{in},i} + \Delta\cdot M + E_{\mathsf{in}}$*,* $A_{\mathsf{in},i} = \sum_{j=0}^{N-1} a_{i,j}\cdot X^j \hookleftarrow \mathcal{U}(\mathfrak{R}_{q,N})$ *for all* $i \in [0,k)$ *and* $E_{\mathsf{in}} = \sum_{j=0}^{N-1} e_j\cdot X^j$*, and* $e_j \hookleftarrow \mathcal{N}_{\sigma^2_{\mathsf{in}}}$ *for all* $j\in[0,N-1)$*.*
- *The key switch key:* $\mathsf{KSK} = (\mathsf{KSK}_0,\ldots,\mathsf{KSK}_{k_{\mathsf{in}}-1})$*, where* $\mathsf{KSK}_i \in \mathrm{GLEV}_{\boldsymbol{S}^{[\phi_{\mathsf{out}}]}_{\mathsf{out}}}(S_{\mathsf{in},i}) = \left(\mathrm{GLWE}_{\boldsymbol{S}^{[\phi_{\mathsf{out}}]}_{\mathsf{out}}}\left(\frac{q}{\mathcal{B}}S_{\mathsf{in},i}\right),\ldots,\mathrm{GLWE}_{\boldsymbol{S}^{[\phi_{\mathsf{out}}]}_{\mathsf{out}}}\left(\frac{q}{\mathcal{B}^\ell}S_{\mathsf{in},i}\right)\right)$ *for all* $0\le i<k_{\mathsf{in}}$*. We note by* $\mathsf{KSK}_{i,j} = (\boldsymbol{A_{i,j}}, B_{i,j}) \in \mathrm{GLWE}_{\boldsymbol{S}^{[\phi_{\mathsf{out}}]}_{\mathsf{out}}}\left(\frac{q}{\mathcal{B}^{j+1}}S_{\mathsf{in},i}\right)$*, for all* $0\le i<k_{\mathsf{in}}$ *and for all* $0\le j<\ell$*, where* $B_{i,j} = \sum_{\tau=0}^{k_{\mathsf{out}}-1} A_{i,j,\tau}\cdot S^{[\phi_{\mathsf{out}}]}_{\mathsf{out},\tau} + \frac{q}{\mathcal{B}^{j+1}}S_{\mathsf{in},i} + E_{\mathsf{ksk},i,j}$*, and* $E_{\mathsf{ksk},i,j} = \sum_{\tau=0}^{N-1} e_{\mathsf{ksk},i,j,\tau}\cdot X^\tau$ *and* $e_{\mathsf{ksk},i,j,\tau} \hookleftarrow \mathcal{N}_{\sigma^2_{\mathsf{ksk}}}$*.*

The output of this algorithm is: $\mathsf{CT}_{\mathsf{out}} = (\boldsymbol{A}_{\mathsf{out}}, B_{\mathsf{out}}) \in \mathrm{GLWE}_{\boldsymbol{S}^{[\phi_{\mathsf{out}}]}_{\mathsf{out}}}(\Delta\cdot M) \subseteq \mathfrak{R}^{k_{\mathsf{out}}+1}_{q,N}$*. By definition, in the decomposition described in Definition 5, we have that* $\mathbf{Dec}^{(\mathcal{B},\ell)}(A_{\mathsf{in},i}) = \left(\widetilde{A}_{\mathsf{in},i,0},\ldots,\widetilde{A}_{\mathsf{in},i,\ell-1}\right)$ *such that* $\widetilde{A}_{\mathsf{in},i} = \sum_{j=0}^{\ell-1}\frac{q}{\mathcal{B}^{j+1}}\widetilde{A}_{\mathsf{in},i,j}$*, for all* $0\le i<k_{\mathsf{in}}$*.*

Let define $\bar{A}_{\mathsf{in},i} = A_{\mathsf{in},i} - \widetilde{A}_{\mathsf{in},i}$*,* $|\bar{a}_{i,\tau}| = |a_{i,\tau} - \widetilde{a}_{i,\tau}| < \frac{q}{2\mathcal{B}^\ell}$*,* $\bar{a}_{i,\tau} \in \left[\frac{-q}{2\mathcal{B}^\ell}, \frac{q}{2\mathcal{B}^\ell}\right)$ *for all* $0\le\tau<N$*. So we have that their expectations and variances are respectively* $\mathbb{E}(\bar{a}_{i,\tau}) = -\frac{1}{2}$*,* $\mathsf{Var}(\bar{a}_{i,\tau}) = \frac{q^2}{12\mathcal{B}^{2\ell}} - \frac{1}{12}$*,* $\mathbb{E}(\widetilde{a}_{i,\tau}) = -\frac{1}{2}$ *and* $\mathsf{Var}(\widetilde{a}_{i,\tau}) = \frac{\mathcal{B}^2-1}{12}$*.*

Now, we can decrypt:

$$\begin{aligned}B_{\mathsf{out}} - \left\langle \boldsymbol{A}_{\mathsf{out}}, \boldsymbol{S}^{[\phi_{\mathsf{out}}]}_{\mathsf{out}}\right\rangle &= \left\langle (\boldsymbol{A}_{\mathsf{out}}, B_{\mathsf{out}}), \left(-\boldsymbol{S}^{[\phi_{\mathsf{out}}]}_{\mathsf{out}}, 1\right)\right\rangle\\ &= \left\langle (\mathbf{0}, B_{\mathsf{in}}) - \sum_{i=0}^{k_{\mathsf{in}}-1}\mathbf{Dec}^{(\mathcal{B},\ell)}(A_{\mathsf{in},i})\cdot\mathsf{KSK}_i, \left(-\boldsymbol{S}^{[\phi_{\mathsf{out}}]}_{\mathsf{out}}, 1\right)\right\rangle\\ &= B_{\mathsf{in}} - \sum_{i=0}^{k_{\mathsf{in}}-1}\sum_{j=0}^{\ell-1}\widetilde{A}_{\mathsf{in},i,j}\left\langle \mathsf{KSK}_{i,j}, \left(-\boldsymbol{S}^{[\phi_{\mathsf{out}}]}_{\mathsf{out}}, 1\right)\right\rangle\\ &= B_{\mathsf{in}} - \sum_{i=0}^{k_{\mathsf{in}}-1}\sum_{j=0}^{\ell-1}\widetilde{A}_{\mathsf{in},i,j}\left(\frac{q}{\mathcal{B}^{j+1}}S_{\mathsf{in},i} + E_{\mathsf{ksk},i,j}\right)\\ &= \underbrace{B_{\mathsf{in}} - \sum_{i=0}^{k_{\mathsf{in}}-1}\widetilde{A}_{\mathsf{in},i}S_{\mathsf{in},i}}_{(I)} - \underbrace{\sum_{i=0}^{k_{\mathsf{in}}-1}\sum_{j=0}^{\ell-1}\widetilde{A}_{\mathsf{in},i,j}\cdot E_{\mathsf{ksk},i,j}}_{(II)}.\end{aligned}$$

Now let's focus on the w^{th} coefficient of part (I):

$$\begin{aligned}
&b_{\mathsf{in},w} - \sum_{i=0}^{k_{\mathsf{in}}-1}\left(\sum_{\tau=0}^{w}\widetilde{a}_{\mathsf{in},i,w-\tau}\cdot s_{\mathsf{in},i,\tau} - \sum_{\tau=w+1}^{N-1}\widetilde{a}_{\mathsf{in},i,N+w-\tau}\cdot s_{\mathsf{in},i,\tau}\right)\\
&= b_{\mathsf{in},w} - \sum_{i=0}^{k_{\mathsf{in}}-1}\left(\sum_{\tau=0}^{w}\left(a_{\mathsf{in},i,w-\tau} - \bar{a}_{\mathsf{in},i,w-\tau}\right)\cdot s_{\mathsf{in},i,\tau}\right.\\
&\qquad\qquad\left. - \sum_{\tau=w+1}^{N-1}\left(a_{\mathsf{in},i,N+w-\tau} - \bar{a}_{\mathsf{in},i,N+w-\tau}\right)\cdot s_{\mathsf{in},i,\tau}\right)\\
&= \Delta m_w + e_w + \sum_{i=0}^{k_{\mathsf{in}}-1}\left(\sum_{\tau=0}^{w}\bar{a}_{\mathsf{in},i,w-\tau}\cdot s_{\mathsf{in},i,\tau} - \sum_{\tau=w+1}^{N-1}\bar{a}_{\mathsf{in},i,N+w-\tau}\cdot s_{\mathsf{in},i,\tau}\right).
\end{aligned}$$

Now let's focus on the w^{th} coefficient of part (II):

$$\sum_{i=0}^{k_{\mathsf{in}}-1}\sum_{j=0}^{\ell-1}\left(\sum_{\tau=0}^{w}\widetilde{a}_{\mathsf{in},i,j,w-\tau}\cdot e_{\mathsf{ksk},i,j,\tau} - \sum_{\tau=w+1}^{N-1}\widetilde{a}_{\mathsf{in},i,j,N+w-\tau}\cdot e_{\mathsf{ksk},i,j,\tau}\right).$$

We can now isolate the output error for the w^{th} coefficient and remove the message coefficient. We obtain that the output error is:

$$\begin{aligned}
e'_w = e_w + \underbrace{\sum_{i=0}^{k_{\mathsf{in}}-1}\left(\sum_{\tau=0}^{w}\bar{a}_{\mathsf{in},i,w-\tau}\cdot s_{\mathsf{in},i,\tau} - \sum_{\tau=w+1}^{N-1}\bar{a}_{\mathsf{in},i,N+w-\tau}\cdot s_{\mathsf{in},i,\tau}\right)}_{(*)}\\
+ \sum_{i=0}^{k_{\mathsf{in}}-1}\sum_{j=0}^{\ell-1}\left(\sum_{\tau=0}^{w}\widetilde{a}_{\mathsf{in},i,j,w-\tau}\cdot e_{\mathsf{ksk},i,j,\tau} - \sum_{\tau=w+1}^{N-1}\widetilde{a}_{\mathsf{in},i,j,N+w-\tau}\cdot e_{\mathsf{ksk},i,j,\tau}\right).
\end{aligned}$$

Observe that in the term $(*)$ there are $k_{\mathsf{in}}N - \phi_{\mathsf{in}}$ terms of type $\bar{a}_{\mathsf{in},i,\cdot}\cdot s_{\mathsf{in},i,\cdot}$ that are equal to 0. So we have:

$$\begin{aligned}
\mathsf{Var}(e'_w) &= \mathsf{Var}(e_w) + \phi_{\mathsf{in}}\cdot\mathsf{Var}\left(\bar{a}_{\mathsf{in},i,\cdot}\cdot s_{\mathsf{in},i,\cdot}\right) + k_{\mathsf{in}}\cdot\ell\cdot N\cdot\mathsf{Var}\left(\widetilde{a}_{\mathsf{in},i,j,\cdot}\cdot e_{\mathsf{ksk},i,j,\cdot}\right)\\
&= \sigma^2_{\mathsf{in}} + \phi_{\mathsf{in}}(\mathsf{Var}\left(\bar{a}_{\mathsf{in},i,\cdot}\right)\mathsf{Var}\left(s_{\mathsf{in},i,\cdot}\right) + \mathsf{Var}\left(\bar{a}_{\mathsf{in},i,\cdot}\right)\mathbb{E}^2\left(s_{\mathsf{in},i,\cdot}\right)\\
&\quad + \mathbb{E}^2(\bar{a}_{\mathsf{in},i,\cdot})\mathsf{Var}\left(s_{\mathsf{in},i,\cdot}\right))\\
&\quad + \ell k_{\mathsf{in}}N\left(\mathsf{Var}\left(\widetilde{a}_{\mathsf{in},i,j,\cdot}\right)\mathsf{Var}\left(e_{\mathsf{ksk},i,j,\cdot}\right) + \mathbb{E}^2\left(\widetilde{a}_{\mathsf{in},i,j,\cdot}\right)\mathsf{Var}\left(e_{\mathsf{ksk},i,j,\cdot}\right)\right.\\
&\quad \left. + \mathsf{Var}\left(\widetilde{a}_{\mathsf{in},i,j,\cdot}\right)\mathbb{E}^2\left(e_{\mathsf{ksk},i,j,\cdot}\right)\right)\\
&= \sigma^2_{\mathsf{in}} + \phi_{\mathsf{in}}\left(\frac{q^2-\mathcal{B}^{2\ell}}{12\mathcal{B}^{2\ell}}\right)\left(\mathsf{Var}\left(\boldsymbol{S}^{[\phi_{\mathsf{in}}]}_{\mathsf{in}}\right) + \mathbb{E}^2\left(\boldsymbol{S}^{[\phi_{\mathsf{in}}]}_{\mathsf{in}}\right)\right)\\
&\quad + \frac{\phi_{\mathsf{in}}}{4}\mathsf{Var}\left(\boldsymbol{S}^{[\phi_{\mathsf{in}}]}_{\mathsf{in}}\right) + \ell k_{\mathsf{in}}N\frac{\mathcal{B}^2+2}{12}\sigma^2_{ksk}.
\end{aligned}$$

Note that when $\phi_{\mathsf{in}} = k_{\mathsf{in}}\cdot N$ we end up with the same formula than the classical GLWE to GLWE key Switch. □

Remark 5.3 (Cost of a GLWE Key Switch)**.** We recall that the cost of a GLWE-to-GLWE key switching, which remains the same whether it involves partial secret keys or not, is

$$\begin{aligned}
\mathsf{Cost}^{k_{\mathsf{in}},k_{\mathsf{out}},N,\ell}_{\mathsf{FftLweKeySwitch}} &= k_{\mathsf{in}}\cdot\ell\cdot\mathsf{Cost}^N_{\mathbf{FFT}} + (k_{\mathsf{out}}+1)\cdot\mathsf{Cost}^N_{\mathbf{iFFT}}\\
&\quad + Nk_{\mathsf{in}}\ell\cdot(k_{\mathsf{out}}+1)\cdot\mathsf{Cost}^N_{\times_{\mathbb{C}}} + N\cdot(k_{\mathsf{in}}\ell-1)\cdot(k_{\mathsf{out}}+1)\cdot\mathsf{Cost}^N_{+_{\mathbb{C}}}.
\end{aligned}$$

where $+_{\mathbb{C}}$ and $\times_{\mathbb{C}}$ represent a double-complex addition and multiplication (in the FFT domain) respectively, and $\mathbf{FFT}_N$ (resp. $\mathbf{iFFT}_N$) the Fast Fourier Transform (resp., inverse FFT).

5.2.2.3 Secret Product GLWE Key Switch with Partial Secret Key

A GLWE-to-GLWE key switch, also computing a product with a secret polynomial, as described in Algorithm 31, follows the exact same definition than above, except that the output ciphertext encrypts $P \cdot M + E_{\mathsf{KS}}$ where $P \in \mathfrak{R}_{q,N}$ is the secret polynomial hidden in the key switching key. The added noise E_{KS} also depends on the input secret key $\boldsymbol{S}^{[\phi_{\mathsf{in}}]}$ and its filling amount ϕ_{in}. Indeed, this term is the product between the rounding term (dependent on ϕ_{in}) and the polynomial P.

Algorithm 31: $\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{SecretProductFftGLWEKeySwitch}(\mathsf{CT}_{\mathsf{in}}, \mathsf{KSK})$

Context:
$\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} \in \mathfrak{R}_{q,N}^{k_{\mathsf{in}}}$: the input partial secret key (Definition 29)
$\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} = (S_{\mathsf{in},0}, \ldots, S_{\mathsf{in},k_{\mathsf{in}}-1})$
$\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]} \in \mathfrak{R}_{q,N}^{k_{\mathsf{out}}}$: the output partial secret key (Definition 29)
$(k_{\mathsf{in}} - 1)N < \phi_{\mathsf{in}} \leq k_{\mathsf{in}}N$ and $(k_{\mathsf{out}} - 1)N < \phi_{\mathsf{out}} \leq k_{\mathsf{out}}N$
$P = \sum_{i=0}^{N-1} p_i X^i \in \mathfrak{R}_{q,N}$
$\ell \in \mathbb{N}$: the number of levels in the decomposition
$\mathfrak{B} \in \mathbb{N}$: the base in the decomposition

Input:
$\mathsf{CT}_{\mathsf{in}} = (A_0, \ldots, A_{k_{\mathsf{in}}-1}, B) \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]}}(M) \subseteq \mathfrak{R}_{q,N}^{k_{\mathsf{in}}+1}$, with $M \in \mathfrak{R}_{p,N}$
$\mathsf{KSK} = \left\{(\mathsf{KSK}_0, \ldots, \mathsf{KSK}_{k_{\mathsf{in}}})\,;\ \texttt{/* Definition 15 */}\right\}$
With $\mathsf{KSK}_i \in \mathrm{GLEV}^{\mathfrak{B},\ell}_{\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}}(S_{\mathsf{in},i} \cdot P)$ for $i \in [0, k_{\mathsf{in}} - 1]$
And $\mathsf{KSK}_k \in \mathrm{GLEV}^{\mathfrak{B},\ell}_{\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}}(P)$

Output: $\mathsf{CT}_{\mathsf{out}} \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}}(P \cdot M) \subseteq \mathfrak{R}_{q,N}^{k_{\mathsf{out}}+1}$

1 Set $\mathsf{CT}_{\mathsf{out}} := \left\langle \mathsf{KSK}_{k_{\mathsf{in}}}, \mathbf{Dec}^{(\mathfrak{B},\ell)}(B) \right\rangle$
2 **for** $i \in [0; k_{\mathsf{in}} - 1]$ **do**
  `/* Decompose the mask */`
3 Update $\mathsf{CT}_{\mathsf{out}} = \mathsf{CT}_{\mathsf{out}} - \left\langle \mathsf{KSK}_i, \mathbf{Dec}^{(\mathfrak{B},\ell)}(A_i) \right\rangle$
4 **return** $\mathsf{CT}_{\mathsf{out}}$

Theorem 5.3 (Noise of Secret-Product GLWE Key Switch)**.** *After performing a Secret-Product key switching (Algorithm 31), taking as input a* GLWE *ciphertext* $\mathsf{CT}_{\mathsf{in}} \in \mathfrak{R}_{q,N}^{k_{\mathsf{in}}+1}$ *under the secret key* $\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} \in \mathfrak{R}_{q,N}^{k_{\mathsf{in}}}$ *and a key switching key with noise variance* σ^2_{KSK} *encrypting a secret message* M_2*, and outputting a* GLWE *ciphertext* $\mathsf{CT}_{\mathsf{out}} \in \mathfrak{R}_{q,N}^{k_{\mathsf{out}}+1}$ *under the secret key* $\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]} \in \mathfrak{R}_{q,N}^{k_{\mathsf{out}}}$*, the noise variance of each coefficient of the output can be estimated by*

$$\begin{aligned}\mathsf{Var}\left(\mathsf{CT}_{\mathsf{out}}\right) &= \ell(k_{\mathsf{in}}+1)N\sigma^2_{\mathsf{KSK}}\frac{\mathfrak{B}^2+2}{12} \\ &+ ||M_2||_2^2 \cdot \left(\sigma^2_{\mathsf{in}} + \left(\frac{q^2 - \mathfrak{B}^{2\ell}}{12\mathfrak{B}^{2\ell}}\right)\left(1 + \phi_{\mathsf{in}}\left(\mathsf{Var}\left(\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]}\right) + \mathbb{E}^2\left(\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]}\right)\right)\right) + \frac{\phi_{\mathsf{in}}}{4}\mathsf{Var}\left(\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]}\right)\right).\end{aligned}$$

Proof *(Theorem 5.3) The proof can trivially be adapted from the proof of Theorem 5.2.* □

5.2.2.4 External Product with Partial Secret Key

An external product is a special case of a secret-product GLWE-to-GLWE key switch where the input secret key and the output secret key are the same. It is pretty easy to compute the noise this procedure will add. The cost to compute a GLWE external product whether it includes a partial secret key or not, is the same.

Theorem 5.4 (Noise of GLWE External Product)**.** *The external product algorithm is the same as the algorithm of secret-product* GLWE *key switch (Algorithm 32). The only difference is that the external product uses the same key* $\boldsymbol{S}^{[\phi]} \in \mathfrak{R}_{q,N}^{k}$ *as input and as output, and the key switching key is now seen as a* GGSW *ciphertext of message* M_2 *encrypted with noise variance* σ_2^2*. For each coefficient of the output* $\mathsf{CT}_{\mathsf{out}}$*, the noise variance can be estimated by*

$$\begin{aligned}\mathsf{Var}\left(\mathsf{CT}_{\mathsf{out}}\right) &= \ell(k+1)N\sigma_2^2\frac{\mathcal{B}^2+2}{12} \\ &+ ||M_2||_2^2 \cdot \left(\sigma_{\mathsf{in}}^2 + \left(\frac{q^2-\mathcal{B}^{2\ell}}{12\mathcal{B}^{2\ell}}\right)\left(1+\phi\left(\mathsf{Var}\left(\boldsymbol{S}^{[\phi]}\right)+\mathbb{E}^2\left(\boldsymbol{S}^{[\phi]}\right)\right)\right)+\frac{\phi}{4}\mathsf{Var}\left(\boldsymbol{S}^{[\phi]}\right)\right) .\end{aligned}$$

Proof *(Theorem 5.4) The proof can trivially be adapted from the proof of Theorem 5.3 with* $k = k_{\mathsf{in}} = k_{\mathsf{out}}$ *and* $\boldsymbol{S}^{[\phi]} = \boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} = \boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}$*.* □

Noise Advantage with TFHE's PBS. Using a partial GLWE secret key to encrypt a bootstrapping key for TFHE's programmable bootstrapping enables two convenient features: first to have a smaller output LWE ciphertext with less than $k \cdot N + 1$ coefficients, and second to reduce the noise growth in each external product (see Theorem 5.4).

External product is the main operation used in the CMuxes of the blind rotation, as explained above. The direct consequence of having smaller output ciphertexts is the fact that we can perform smaller LWE-to-LWE key switchings before the next PBS. Furthermore, when $k \cdot N$ is large enough to reach the noise plateau (as explained in Limitation 2), partial secret keys enable to avoid adding unnecessary noise to the bootstrapping.

5.2.2.5 LWE-to-LWE Key Switch

Finally, we study the complete algorithm to compute and LWE-to-LWE key switch. We assume using the GLWE-to-GLWE key switch, but the formulae can easily be adapted to the private product one.

Theorem 5.5 (Noise & Cost of FFT-Based LWE Key Switch)**.** *We consider the new LWE-to-LWE key switch as described in Algorithm 32. Its cost is the same as the cost of a GLWE-to-GLWE key switch as introduced in Remark 5.3 i.e.,* $\mathscr{C}\left(\mathsf{FftLweKeySwitch}\right) = \mathscr{C}\left(\mathsf{GlweKeySwitch}\right)$*.*

The output noise can be expressed from the noise formula of the GLWE-to-GLWE key switch (Theorem 5.2). To sum up, the output noise is:

$$\mathsf{Var}\left(\mathsf{FftLweKeySwitch}\right) = \mathsf{FftError}_{k_{\max},N,\mathcal{B},\ell} + \mathsf{Var}\left(\mathsf{GlweKeySwitch}\right)$$

with $\phi_{\mathsf{in}} = n_{\mathsf{in}}$*,* $\phi_{\mathsf{out}} = n_{\mathsf{out}}$*,* $k_{\max} = \max\left(k_{\mathsf{in}}, k_{\mathsf{out}}\right)$ *and* $\mathsf{FftError}_{k_{\max},N,\mathcal{B},\ell}$ *being the error added by the FFT conversions.*

Proof *(Theorem 5.5). Expressing the cost is quite straight forward, since we can neglect the complexity of the sample extraction and its inverse. The estimation of the variance of the error is immediate as well. We use the corrective formula introduced in Subsection 2.5.3 to estimate an upper bound on the FFT error. Indeed, it is easy to see that the FFT-based LWE key switch with* k_{in} *and* k_{out} *is a special case of an external product with* $k_{\max} = \max\left(k_{\mathsf{in}}, k_{\mathsf{out}}\right)$ *where some of the ciphertexts composing the* GGSW *are trivial encryptions of 0 or 1 (no noise, all mask elements set to zero and the plaintext put in the b/B part).* □

Practical Improvement. The use of partial secret keys provides a significant practical improvement in homomorphic computations. Indeed, it introduces an additional degree of freedom when selecting parameters, allowing for the choice of more efficient parameter sets that result in reduced latency.

Table 5.1 presents a comparison of our techniques and the state-of-the-art [CJP21]. More details on the experiments are reported in Section 5.5.1.

Algorithm 32: $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{FftLweKeySwitch}(\mathsf{ct}_{\mathsf{in}}, \mathsf{KSK})$

Context:
$\begin{cases} \boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} \in \mathfrak{R}_{q,N}^{k_{\mathsf{in}}} : \text{the input partial secret key (Definition 29)} \\ \boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} = (S_{\mathsf{in},0}, \ldots, S_{\mathsf{in},k_{\mathsf{in}}-1}) \\ \boldsymbol{s}_{\mathsf{in}} : \text{the flatten version of } \boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} \text{(Definition 11)} \\ \boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]} \in \mathfrak{R}_{q,N}^{k_{\mathsf{out}}} : \text{the output partial secret key (Definition 29)} \\ (k_{\mathsf{in}}-1)N < \phi_{\mathsf{in}} \leq k_{\mathsf{in}}N \text{ and } (k_{\mathsf{out}}-1)N < \phi_{\mathsf{out}} \leq k_{\mathsf{out}}N \\ \ell \in \mathbb{N} : \text{ the number of levels in the decomposition} \\ \mathfrak{B} \in \mathbb{N} : \text{ the base in the decomposition} \end{cases}$

Input:
$\begin{cases} \mathsf{ct}_{\mathsf{in}} \in \mathrm{LWE}_{\boldsymbol{s}_{\mathsf{in}}}(m) \subseteq \mathbb{Z}_q^n, \text{ with } m \in \mathbb{Z}_p \\ \mathsf{KSK} = \left\{ \left(\mathsf{KSK}_0, \ldots, \mathsf{KSK}_{k_{\mathsf{in}}-1} \right) \right\}; \texttt{ /* Definition 15 */} \\ \text{With } \mathsf{KSK}_i \in \mathrm{GLEV}_{\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}}^{\mathfrak{B},\ell}(S_{\mathsf{in},i}) \text{ for } i \in [0, k_{\mathsf{in}}-1] \end{cases}$

Output: $\mathsf{ct}_{\mathsf{out}} \in \mathrm{LWE}_{\boldsymbol{s}_{\mathsf{out}}}(m) \subseteq \mathbb{Z}_q^{\phi_{\mathsf{out}}+1}$

`/* Inverse of a constant sample extraction (Algorithm 29) */`

1 Set $\mathsf{CT} = (A_0, \ldots, A_{k_{\mathsf{in}}-1}, B) \leftarrow \mathsf{ConstantSampleExtraction}^{-1}(\mathsf{ct}_{\mathsf{in}}, k_{\mathsf{in}}, N) \in \mathfrak{R}_{q,N}^{k_{\mathsf{in}}+1}$

`/* GLWE Key Switch with Partial Secret Keys (Algorithm 30) */`

2 Set $\mathsf{CT}' \leftarrow \mathsf{GlweKeySwitch}(\mathsf{CT}, \mathsf{KSK}) \in \mathfrak{R}_{q,N}^{k_{\mathsf{out}}+1}$

`/* Constant sample extraction (Algorithm 28) */`

3 Set $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{ConstantSampleExtract}\left(\mathsf{CT}'\right) \in \mathbb{Z}_q^{\phi_{\mathsf{out}}+1}$

4 **return** $\mathsf{ct}_{\mathsf{out}}$

5.3 Secret Keys with Shared Randomness

To use FHE schemes, one needs to generate several secret keys of different sizes (Remark 2.15). Our main observation is that instead of sampling those keys independently, we can generate a list of α nested GLWE keys with the same level of security λ.

As a simple example we consider three integers $1 < n_0 < n_1 < n_2$ and a secret key $\boldsymbol{s}^{(2)} = \boldsymbol{r}^{(0)} || \boldsymbol{r}^{(1)} || \boldsymbol{r}^{(2)} \in \mathbb{Z}_q^{n_2}$ generated in the traditional manner. We can now build two smaller secret keys out of $\boldsymbol{s}^{(2)}$ such that for all pair of keys, the smaller one will be included in the bigger one, in its first coefficients: $\boldsymbol{s}^{(0)} = \boldsymbol{r}^{(0)} \in \mathbb{Z}_q^{n_0}$ and $\boldsymbol{s}^{(1)} = \boldsymbol{r}^{(0)} || \boldsymbol{r}^{(1)} \in \mathbb{Z}_q^{n_1}$, as represented in Figure 5.1. With this new secret keys, the cost and the noise of a key switch between $\boldsymbol{s}^{(1)}$ and $\boldsymbol{s}^{(0)}$ will no longer depend on n_0 and n_1 but on $n_1 - n_0$ and n_0. If we want to key switch from $\boldsymbol{s}^{(0)}$ to $\boldsymbol{s}^{(1)}$, the key switch will come for free: it will add no noise and will have no cost. Note that each of those secret keys use a different variance for the noise added during encryption: the smaller the secret key, the larger the required noise variance, so they can all guarantee the same level of security λ.

In this section, we first define the secret key with shared randomness. We then study the impact of these keys on the security of the underlying LWE problems. Finally, we list the different advantages and improvements which they offer.

Definition 32 (GLWE Secret Keys with Shared Randomness)**.** *Two GLWE secret keys $\boldsymbol{S} \in \mathfrak{R}_{q,N}^{k}$ and $\boldsymbol{S}' \in \mathfrak{R}_{q,N'}^{k'}$, with $kN \leq k'N'$, are said to share randomness if we have that for all $0 \leq i < kN$, $\bar{s}_i = \bar{s}'_i$, where $\bar{s}_i$ and $\bar{s}'_i$ respectively come from the flattened view (Definition 11) of $\boldsymbol{S}$ and $\boldsymbol{S}'$. We note by $\boldsymbol{S} \prec \boldsymbol{S}'$ this relationship between secret keys.*

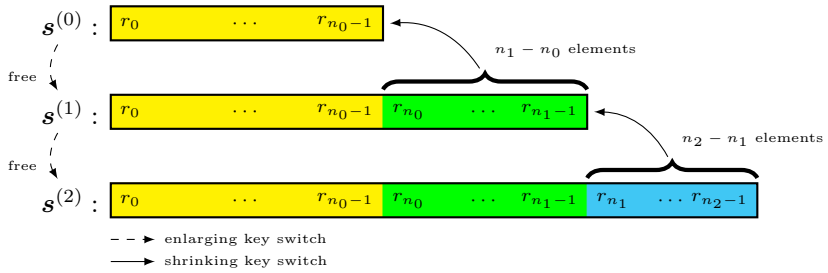

Figure 5.1: Illustration of simplified key switch procedures between three LWE secret keys with shared randomness.

5.3.1 Hardness of Secret Keys with Shared Randomness

Let us consider different samples of GLWE with shared randomness, i.e., samples under the LWE secret key $\boldsymbol{s}^0 = (s_0, \dots s_{n_0-1})$ and other samples under the secret key $\boldsymbol{s}^1 = (s_0, \dots s_{n_0-1}, s_{n_0}, \dots s_{n_1})$. By only considering the samples under the secret key $\boldsymbol{s}^0$, all these samples are secure and have a level of security λ. The same holds for samples under the secret key $\boldsymbol{s}^1$. We now study the level of security of several samples of GLWE considered together with shared secret keys.
First, we present the decisional LWE problem with shared randomness and prove that, under certain conditions, this problem can be reduced to a LWE problem. Next we show that the new operations offered by the secret key with shared randomness can not impact the security.

Definition 33 (Secret Key with Shared Randomness Decisional Problem)**.** *Let $n_1 > n_0$. Given a secret key $\boldsymbol{s}^{(0)} \in \mathbb{Z}_q^{n_0}$ following a given distribution $\mathcal{D}$, a secret $\boldsymbol{r} \in \mathbb{Z}_q^{n_1-n_0}$ following the same distribution $\mathcal{D}$, and two errors distribution χ_0 and χ_1, we define the LWE with secret key with shared randomness samples and we note* sh-LWE$_{n_0,\chi_0,n_1,\chi_1}$ *the pairs $\left((\boldsymbol{a_0}, b_0 = \left\langle \boldsymbol{a_0}, \boldsymbol{s}^{(0)} \right\rangle + e_0), (\boldsymbol{a_1}, b_1 = \left\langle \boldsymbol{a_1}, \boldsymbol{s}^{(1)} \right\rangle + e_1)\right) \in \mathbb{Z}_q^{n_0+1} \times \mathbb{Z}_q^{n_1+1}$, where $\boldsymbol{s}^{(1)} = \boldsymbol{s}^{(0)} || \boldsymbol{r}$, $\boldsymbol{a_0} \hookleftarrow \mathcal{U}\left(\mathbb{Z}_q\right)^{n_0}$, $\boldsymbol{a_1} \hookleftarrow \mathcal{U}\left(\mathbb{Z}_q\right)^{n_1}$, $e_0 \hookleftarrow \chi_0$ and $e_1 \hookleftarrow \chi_1$.*

The decision sh-LWE$_{n_0,\chi_0,n_1,\chi_1}$ *problem consist of distinguishing m independent samples from $\mathcal{U}\left(\mathbb{Z}_q^{n_0+1} \times \mathbb{Z}_q^{n_1+1}\right)$ from m independent samples $((\boldsymbol{a_0}, b_0), (\boldsymbol{a_1}, b_1)) \in$ LWE$_{n_0,\chi_0} \times$ LWE$_{n_1,\chi_1} \subseteq \mathbb{Z}_q^{n_0+1} \times \mathbb{Z}_q^{n_1+1}$ as defined above.*

Theorem 5.6 (Hardness of sh-LWE)**.** *If we have three random distributions χ_0, χ_1 and χ' such that, if we sample $e_1 \hookleftarrow \chi_1$ and $e' \hookleftarrow \chi'$, $e_1 + e'$ follows the distribution χ_0. Then* sh-LWE$_{n_0,\chi_0,n_1,\chi_1}$ *with m samples is at least as hard than* LWE$_{n_0,\chi_1}$ *with $2m$ samples.*

Remark 5.4 (Error Distribution χ)**.** In GLWE-based FHE schemes, χ usually follows a discrete normal distribution. The condition for Theorem 5.6 is then always verified. In the following, the goal is to use a noise variance σ_0 for χ_0 and a noise variance σ_1 for χ_1 such that $n_0 < n_1$ and $\sigma_0 > \sigma_1$.

Proof *(Theorem 5.6). We define an instance of* LWE$_{n_0,\chi_1}$ *where the samples are encrypted under a secret key $\boldsymbol{s}^{(0)} \in \mathbb{Z}_q^{n_0}$ which follows a given distribution $\mathcal{D}$, and where for the given distribution χ_1 it exists a distribution χ' such that, for any $e_1 \hookleftarrow \chi_1$ and for any $e' \hookleftarrow \chi'$, we have that $e_1 + e'$ follows a distribution χ_0.*

We now prove that solving the problem sh-LWE$_{n_0,\chi_0,n_1,\chi_1}$ *is at least as hard than solving the problem* LWE$_{n_0,\chi_1}$*. To do so, we consider an oracle that can solve the decision* sh-LWE$_{n_0,\chi_0,n_1,\chi_1}$ *problem and show that a such oracle can solve the decisional* LWE$_{n_0,\chi_1}$ *instance.*

Observe that, starting from an LWE$_{n_0,\chi_1}$ *sample, we can easily create either an* LWE$_{n_0,\chi_0}$ *sample or an* LWE$_{n_1,\chi_1}$ *sample. To create an* LWE$_{n_0,\chi_0}$ *sample $(\boldsymbol{a}_0, b_0 = \left\langle \boldsymbol{a}_0, \boldsymbol{s}^{(0)} \right\rangle + e_0) \in \mathbb{Z}_q^{n_0+1}$, with e_0 coming from a distribution χ_0, from an* LWE$_{n_0,\chi_1}$ *sample $(\boldsymbol{a}_1, b_1 = \left\langle \boldsymbol{a}_1, \boldsymbol{s}^{(0)} \right\rangle + e_1) \in \mathbb{Z}_q^{n_0+1}$, with e_1 coming from a distribution χ_1, we only need to take $\boldsymbol{a}_0 = \boldsymbol{a}_1$ and modify the noise. Following the above condition, it is sufficient to sample $e' \hookleftarrow \chi'$ and then take $b_0 = b_1 + e'$, to make the noise in b_0 be equal to $e_1 + e'$. This now follows the distribution χ_0.*

To create an $\mathrm{LWE}_{n_1,\chi_1}$ *sample* $(\boldsymbol{a}_1, b_1 = \langle \boldsymbol{a}_1, \boldsymbol{s}^{(1)} \rangle + e_1) \in \mathbb{Z}_q^{n_1+1}$ *from a* $\mathrm{LWE}_{n_0,\chi_1}$ *sample* $(\boldsymbol{a}_0, b_0 = \langle \boldsymbol{a}_0, \boldsymbol{s}^{(0)} \rangle + e_1) \in \mathbb{Z}_q^{n_0+1}$*, we start by generating a random key* $r \in \mathbb{Z}_q^{n_1-n_0}$*, which follows the same distribution* $\mathcal{D}_s$ *than* $s^{(0)}$*, as well as a new vector* $\boldsymbol{a}' \in \mathcal{U}\left(\mathbb{Z}_q^{n_1-n_0}\right)$*. Then, we take* $\boldsymbol{a}_1 = \boldsymbol{a}_0 || \boldsymbol{a}'$ *and* $b_1 = b_0 + \langle \boldsymbol{a}', \boldsymbol{r}' \rangle$*, which give us an* $\mathrm{LWE}_{n_1,\chi_1}$ *sample as expected.*

Following what just described, we observe that given $2m$ $\mathrm{LWE}_{n_0,\chi_1}$ *samples, we can generate* m $\mathrm{LWE}_{n_0,\chi_0}$ *samples and* m $\mathrm{LWE}_{n_1,\chi_1}$ *samples. Now we can provide all the valid samples of* $\mathrm{LWE}_{n_0,\chi_0} \times \mathrm{LWE}_{n_1,\chi_1}$ *to the oracle. Otherwise, when the decisional* $\mathrm{LWE}_{n_0,\chi_1}$ *problem send uniform samples in* $\mathbb{Z}_q^{n_0}$*, the two transformations proposed before also return uniform samples in* $\mathbb{Z}_q^{n_0}$ *or in* $\mathbb{Z}_q^{n_1}$*. As the oracle can solve the decision* sh-$\mathrm{LWE}_{n_0,\chi_0,n_1,\chi_1}$ *problem, we can solve the decision* $\mathrm{LWE}_{n_0,\chi_1}$ *problem.* □

Remark 5.5 (Security with more than two shared Keys)**.** The Proof 5.3.1, can easily be adapted to more than only two shared keys.

Operations Under Shared Randomness Any known homomorphic operation (that we know) that makes two or more ciphertexts interact (encrypted under the same key or different keys) will have as a result a ciphertext with a level of security at least as high as the input with the lowest security level. In light of the common existing attacks, the level of security of a set of GLWE samples encrypted under secret keys with shared randomness is then lower bounded by the level of security of the GLWE having the smallest level of security.

As for the partial secret keys, this new type of keys may lead to new unknown attacks and the level of security could be impacted. But at the current state-of-the-art, no attacks seem to have an impact on secret key with shared randomness. However, if one of the key sets is compromised, the other key sets will be impacted consequently.

5.3.2 Advantages of Secret Keys with Shared Randomness

Using secret keys with shared randomness enables us to speed up homomorphic computations and to reduce the amount of noise added by these operations. This is particularly useful for LWE-to-LWE key switch procedures.

5.3.2.1 Advantages with LWE-to-LWE Key Switch

The first operation that benefits from secret keys with shared randomness is key switching. Figure 5.1 illustrates key switching processes between three LWE secret keys with shared randomness. A key switch to a bigger key is represented with dotted arrows and is called *enlarging key switch*. A key switch to a smaller key is represented with solid arrows and is called *shrinking key switch*.

Enlarging Key Switch. When we consider a ciphertext $\mathsf{ct}_{\mathsf{in}} = (a_0, \ldots, a_{n_1-1}, b) \in \mathrm{LWE}_{\boldsymbol{s}^{(1)}}(m) \subseteq \mathbb{Z}_q^{n_1+1}$ under the secret key $\boldsymbol{s}^{(1)} \in \mathbb{Z}_q^{n_1}$ and want to key switch it to the secret key $\boldsymbol{s}^{(2)} \in \mathbb{Z}_q^{n_2}$, where $\boldsymbol{s}^{(1)} \prec \boldsymbol{s}^{(2)}$, The algorithm reduces to simply appending zeros to the end of the ciphertext:

$$\mathsf{ct}_{\mathsf{out}} := (a_0, \ldots, a_{n_1-1}, 0, \ldots, 0, b) \in \mathrm{LWE}_{\boldsymbol{s}^{(2)}}(m) \subseteq \mathbb{Z}_q^{n_2+1}.$$

Algorithm 33, describes this procedure in detail. We note that we only use this algorithm with LWE ciphertexts, but it can trivially be extended to GLWE ciphertexts.

To sum up, the enlarging key switching are basically zero-cost operations and do not require the use of a public key. They also add no noise, instead of adding a linear combination of freshly encrypted ciphertexts under $\boldsymbol{s}^{(2)}$. The proof of next theorem is trivial.

Theorem 5.7 (Cost & Noise of Enlarging Key Switching)**.** *When working with secret keys with shared randomness, the cost of an enlarging key switching (Algorithm 33) is reduced to zero, and the noise in the output is the same as the one in the input (no noise is added).*

Algorithm 33: $\mathsf{ct_{out}} \leftarrow \mathsf{EnlargingKeySwitch}(\mathsf{ct_{in}})$

Context:
- $\boldsymbol{s}_{\mathsf{in}} \in \mathbb{Z}_q^{n_{\mathsf{in}}}$: the input secret key
- $\boldsymbol{s}_{\mathsf{out}} \in \mathbb{Z}_q^{n_{\mathsf{out}}}$: the output secret key
- $\boldsymbol{s}_{\mathsf{in}} \prec \boldsymbol{s}_{\mathsf{out}}$: secret keys with shared randomness (Definition 32)
- $m \in \mathbb{Z}_p$
- $\mathsf{ct_{in}} = (a_0, \ldots, a_{n_{\mathsf{in}}-1}, b) \in \mathbb{Z}_q^{n_{\mathsf{in}}+1}$

Input: $\mathsf{ct_{in}} \in \mathrm{LWE}_{\boldsymbol{s}_{\mathsf{in}}}(m)$
Output: $\mathsf{ct_{out}} \in \mathrm{LWE}_{\boldsymbol{s}_{\mathsf{out}}}(m)$

`/* Pad with zeros between the mask and the` b `part */`

Set $\mathsf{ct_{out}} := (a_0, \ldots, a_{n-1}, 0, \ldots, 0, b) \in \mathbb{Z}_q^{n_{\mathsf{out}}+1}$

return $\mathsf{ct_{out}}$

Shrinking Key Switch. When we consider a ciphertext $\mathsf{ct_{in}} = (a_0, \ldots, a_{n_2-1}, b) \in \mathbb{Z}_q^{n_2+1}$ under the secret key $\boldsymbol{s}^{(2)} \in \mathbb{Z}_q^{n_2}$ and we want to key switch it to the secret key $\boldsymbol{s}^{(1)} \in \mathbb{Z}_q^{n_1}$, where $\boldsymbol{s}^{(1)} \prec \boldsymbol{s}^{(2)}$ and $\boldsymbol{s}^{(2)} = \boldsymbol{s}^{(1)} || \boldsymbol{r}^{(2)}$, the algorithm is simplified precisely because of the shared randomness:

1. the parts $(a_0, \ldots, a_{n_1-1})$ and b do not need to be processed but simply reorganized into a temporary ciphertext: $\mathsf{ct} = (a_0, \ldots, a_{n_1-1}, b) \in \mathbb{Z}_q^{n_1+1}$,
2. the part $(a_{n_1}, \ldots, a_{n_2-1})$ has to be key switched, which can be viewed as a traditional key switching algorithm: i.e., key switching the ciphertext $(a_{n_1}, \ldots, a_{n_2-1}, 0) \in \mathbb{Z}_q^{n_2-n_1+1}$ with a key switching key going from the secret key $\boldsymbol{r}^{(2)}$ to $\boldsymbol{s}^{(1)}$, and at the end, adding it to ct and returning the result.

Algorithm 34, describes this procedure in detail. We only use this algorithm with LWE ciphertexts, but it can be also trivially extended to GLWE ciphertexts.

Algorithm 34: $\mathsf{ct_{out}} \leftarrow \mathsf{ShrinkingKeySwitch}(\mathsf{ct_{in}}, \mathsf{KSK})$

Context:
- $\boldsymbol{s}_{\mathsf{in}} = (s_0, \ldots, s_{n_{\mathsf{in}}-1}) \in \mathbb{Z}_q^{n_{\mathsf{in}}}$: the input secret key
- $\boldsymbol{s}_{\mathsf{out}} \in \mathbb{Z}_q^{n_{\mathsf{out}}}$: the output secret key
- $n_{\mathsf{out}} < n_{\mathsf{in}}$
- $\boldsymbol{s}_{\mathsf{out}} \prec \boldsymbol{s}_{\mathsf{in}}$: secret keys with shared randomness (Definition 32)
- $m \in \mathbb{Z}_p$
- $\mathsf{ct_{in}} = (a_0, \ldots, a_{n_{\mathsf{in}}-1}, b) \in \mathbb{Z}_q^{n_{\mathsf{in}}+1}$
- $\ell \in \mathbb{N}$: the number of levels in the decomposition
- $\mathcal{B} \in \mathbb{N}$: the base in the decomposition

Input:
- $\mathsf{ct_{in}} \in \mathrm{LWE}_{\boldsymbol{s}_{\mathsf{in}}}(m)$
- $\mathsf{KSK} = \{\mathsf{KSK}_i\}_{n_{\mathsf{out}} \leq i < n_{\mathsf{in}}}$: a key switching key
- With $\mathsf{KSK}_i \in \mathrm{LEV}_{\boldsymbol{s}_{\mathsf{out}}}(s_{\mathsf{in},i})$

Output: $\mathsf{ct_{out}} \in \mathrm{LWE}_{\boldsymbol{s}_{\mathsf{out}}}(m)$

`/* Keep the beginning of the mask and the` B `part */`

Set $\mathsf{ct_{out}} := (a_0, \ldots, a_{n_{\mathsf{out}}-1}, b) \in \mathbb{Z}_q^{n_{\mathsf{out}}+1}$

for $i \in [n_{\mathsf{out}}; n_{\mathsf{in}} - 1]$ **do**

  `/* Decompose the rest of the mask */`

  Update $\mathsf{ct_{out}} = \mathsf{ct_{out}} - \left\langle \mathsf{KSK}_i, \mathbf{Dec}^{(\mathcal{B},\ell)}(a_i) \right\rangle$

return $\mathsf{ct_{out}}$

To sum up, the shrinking key switching requires smaller key switching keys: their size becomes proportional to $n_2 - n_1$ instead of n_2. As a consequence, the computation is faster, equivalent to key switch a ciphertext of size $n_2 - n_1 + 1$ instead of $n_2 + 1$. Finally, the noise in the output is also smaller because the algorithm involves a smaller linear combination of freshly encrypted ciphertexts under $\boldsymbol{s}^{(1)}$.

Theorem 5.8 (Cost & Noise of Shrinking Key Switching)**.** *Consider two secret keys with shared randomness $\boldsymbol{s}^{(0)} \prec \boldsymbol{s}^{(1)}$ with $\boldsymbol{s}^{(0)} \in \mathbb{Z}_q^{n_0}$, $\boldsymbol{s}^{(1)} \in \mathbb{Z}_q^{n_1}$ and $1 < n_0 < n_1$. Let $\mathcal{B} \in \mathbb{N}^*$ and $\ell \in \mathbb{N}^*$ be the decomposition base and level used in key switching. The cost of our shrinking key switching (Algorithm 34) is $\ell\,(n_1 - n_0)\,(n_0 + 1)$ integer multiplications and $(\ell\,(n_1 - n_0) - 1)\,(n_0 + 1)$ integer additions. The noise added by the procedure satisfies*

$$\begin{aligned}\mathsf{Var}(\mathsf{ShrinkingKeySwitch}) = (n_1 - n_0)\left(\frac{q^2 - \mathcal{B}^{2\ell}}{12\mathcal{B}^{2\ell}}\right)\left(\mathsf{Var}\left(\boldsymbol{s}_{\mathsf{in}}\right) + \mathbb{E}^2\left(\boldsymbol{s}_{\mathsf{in}}\right)\right) \\ + \frac{(n_1 - n_0)}{4}\mathsf{Var}\left(\boldsymbol{s}_{\mathsf{in}}\right) + \ell \cdot (n_1 - n_0) \cdot \frac{\mathcal{B}^2 + 2}{12}\sigma^2_{\mathsf{KSK}}\,.\end{aligned}$$

Proof *(Theorem 5.8). The proof is similar to that of Algorithm 4 (Theorem 2.8), except that we only need to key switch the unshared elements.* □

5.3.2.2 Stair Key Switch

In Section 5.3.2.1, we showed that using secret keys with shared randomness can reduce the cost of one of the central algorithms in TFHE, resulting in an overall speedup. However, this concept can also be used locally inside a key switch procedure to explore a cost/noise trade-off.

For simplicity, let us consider an FHE use case where there are only two LWE secret keys, and only a key switch from the large one to the small one. We start by setting the two secret keys with shared randomness. The idea here is to add one or several secret keys with shared randomness, only during the key switch procedure.

For example, let us assume a fixed decomposition base $\mathcal{B}$, a fixed number of levels ℓ and let $\boldsymbol{s}^{(2)}$ be our large secret key and $\boldsymbol{s}^{(0)}$ be our small (as defined in Section 5.3.2.1). To key switch from $\boldsymbol{s}^{(2)}$ to $\boldsymbol{s}^{(0)}$, we will add one intermediate secret key with shared randomness $\boldsymbol{s}^{(1)}$ and compute first a key switch from $\boldsymbol{s}^{(2)}$ to $\boldsymbol{s}^{(1)}$ and then another from $\boldsymbol{s}^{(1)}$ to $\boldsymbol{s}^{(0)}$. This algorithm will be more costly, because its first part will be a linear combination of $(n_2 - n_1)$ ciphertexts of size $n_1 + 1$, and its second part a linear combination of $(n_1 - n_0)$ ciphertexts of smaller size $n_0 + 1$, instead of having a single linear combination of $n_2 - n_0$ ciphertexts of size $n_0 + 1$: so the total number of ciphertexts in the linear combination and in the key switching key has not changed ($n_2 - n_1 + n_1 - n_0 = n_2 - n_0$ as in the key switching from $\boldsymbol{s}^{(2)}$ to $\boldsymbol{s}^{(0)}$), but the linear combinations are slightly more costly and the ciphertexts composing the key switching keys slightly larger. However, this algorithm produces less noise: indeed its first part has ciphertexts with lower noise because they are encrypted under a larger secret key.

Here is the trade-off we want to study. The extreme is to go from $\boldsymbol{s}^{(\mathsf{nb})}$ to $\boldsymbol{s}^{(0)}$ by key switching one element of the key in each key switching, meaning that we will have a total number of $\mathsf{nb} = n_{\mathsf{nb}} - n_0$ shrinking key switching (Algorithm 34) to perform. So nb corresponds to the steps in the stair. This means considering a total number of shared keys equals to $\mathsf{nb} + 1$, including the secret $\boldsymbol{s}^{(\mathsf{nb})}$ and $\boldsymbol{s}^{(0)}$ which are the end points of the stair. We call the added keys between $\boldsymbol{s}^{(\mathsf{nb})}$ and $\boldsymbol{s}^{(0)}$ intermediate secret keys, so we have a total of $\mathsf{nb} - 1$ intermediate secret keys. In practice, we start with coefficient $a_{n_{\mathsf{nb}}-1}$ and key switch it to the secret key with $n_{\mathsf{nb}} - 1$ elements, add it to the rest, and do the same with the next last element, and so on until we reach the desired secret key, one coefficient at a time. The other extreme case is when we key switch directly from $\boldsymbol{s}^{(1)}$ and $\boldsymbol{s}^{(0)}$ without intermediary key switchings, so $\mathsf{nb} = 1$.

Algorithm 35 gives details about this procedure. It is important to point out that there are now nb couples of decomposition parameters $(\mathcal{B}_\alpha, \ell_\alpha)$ for $0 \leq \alpha \leq \mathsf{nb} - 1$, one for each step of the stairs.

Algorithm 35: $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{StairKeySwitch}\left(\mathsf{ct}_{\mathsf{in}}, \{\mathsf{KSK}_\alpha\}_{0\leq\alpha\leq\mathsf{nb}-1}\right)$

Context:
$$\begin{cases} \mathsf{nb} \in \mathbb{N} : \text{ the number of steps in the algorithm} \\ n_0 < n_1 < \cdots < n_{\mathsf{nb}} \\ \boldsymbol{s}^{(\mathsf{nb})} \in \mathbb{Z}_q^{n_{\mathsf{nb}}} : \text{the input secret key} \\ \boldsymbol{s}^{(0)} \in \mathbb{Z}_q^{n_0} : \text{the output secret key} \\ \boldsymbol{s}^{(\alpha)} \in \mathbb{Z}_q^{n_\alpha}, \forall 1 \leq \alpha \leq \mathsf{nb}-1 : \text{ intermediate secret keys} \\ \boldsymbol{s}^{(0)} \prec \boldsymbol{s}^{(1)} \prec \cdots \prec \boldsymbol{s}^{(\mathsf{nb})} : \text{ secret keys shared randomness (Definition 32)} \end{cases}$$

Input:
$$\begin{cases} \mathsf{ct}_{\mathsf{in}} \in \mathrm{LWE}_{\boldsymbol{s}^{(\mathsf{nb})}}(m) \subseteq \mathbb{Z}_q^{n_{\mathsf{nb}}+1}, \text{ with } m \in \mathbb{Z}_p \\ \{\mathsf{KSK}_\alpha\}_{0\leq\alpha\leq\mathsf{nb}-1} : \text{ intermediate key switching key as in Algorithm 34} \\ \qquad\qquad\qquad\qquad \text{where } \mathsf{KSK}_\alpha \text{ switches from } \boldsymbol{s}^{(\alpha+1)} \text{ to } \boldsymbol{s}^{(\alpha)} \end{cases}$$

Output: $\mathsf{ct}_{\mathsf{out}} \in \mathrm{LWE}_{\boldsymbol{s}^{(0)}}(m) \subseteq \mathbb{Z}_q^{n_0+1}$

`/* Set the counter to go from nb − 1 to 0 */`

1 Set $\alpha := \mathsf{nb} - 1$

`/* Set the initial ciphertext */`

2 Set $\mathsf{ct} := \mathsf{ct}_{\mathsf{in}}$

3 **while** $\alpha >= 0$ **do**

`/* Call to Algorithm 34 */`

4 Update $\mathsf{ct} \leftarrow \mathsf{ShrinkingKeySwitch}(\mathsf{ct}, \mathsf{KSK}_\alpha) \in \mathrm{LWE}_{\boldsymbol{s}^{(\alpha)}}(m) \subseteq \mathbb{Z}_q^{n_\alpha+1}$

5 $\alpha := \alpha - 1$

6 **return** $\mathsf{ct}_{\mathsf{out}} := \mathsf{ct}$

Theorem 5.9 (Cost & Noise of Stair Shrinking Key Switching)**.** *Consider the stair key switch as detailed in Algorithm 35. Its cost is* $\sum_{\alpha=0}^{\mathsf{nb}-1} \ell_\alpha (n_{\alpha+1} - n_\alpha)(n_\alpha + 1)$ *integer multiplications and* $\sum_{\alpha=0}^{\mathsf{nb}-1} (\ell_\alpha (n_{\alpha+1} - n_\alpha) - 1)(n_\alpha + 1)$ *integer additions. The noise added by the procedure satisfies*

$$\begin{aligned} \mathsf{Var}(\mathsf{StairShrinkKS}) = &\sum_{\alpha=0}^{\mathsf{nb}-1} (n_{\alpha+1} - n_\alpha) \left(\frac{q^2 - \mathfrak{B}_\alpha^{2\ell_\alpha}}{12\mathfrak{B}_\alpha^{2\ell_\alpha}}\right) \left(\mathsf{Var}\left(\boldsymbol{s}^{(\alpha+1)}\right) + \mathbb{E}^2\left(\boldsymbol{s}^{(\alpha+1)}\right)\right) \\ &+ \frac{(n_{\alpha+1} - n_\alpha)}{4} \mathsf{Var}\left(\boldsymbol{s}^{(\alpha+1)}\right) + \ell_\alpha \cdot (n_{\alpha+1} - n_\alpha) \cdot \frac{\mathfrak{B}_\alpha^2 + 2}{12} \sigma^2_{\mathsf{KSK}_\alpha}. \end{aligned}$$

Proof *(Theorem 5.9). The cost and noise of the stair shirnking key switching can be trivially deduced from the Theorem 5.8. Indeed, at step* α *of the loop in Algorithm 35, the cost of the shrinking key switching is* $\ell_\alpha (n_{\alpha+1} - n_\alpha)(n_\alpha + 1)$ *integer multiplications and* $(\ell_\alpha (n_{\alpha+1} - n_\alpha) - 1)(n_\alpha + 1)$ *integer additions.*

The variance of the noise added at the step α *is:*

$$\begin{aligned} \mathsf{Var}(\mathsf{ShrinkKS}_\alpha) = &(n_{\alpha+1} - n_\alpha) \left(\frac{q^2 - \mathfrak{B}_\alpha^{2\ell_\alpha}}{12\mathfrak{B}_\alpha^{2\ell_\alpha}}\right) \left(\mathsf{Var}\left(\boldsymbol{s}^{(\alpha+1)}\right) + \mathbb{E}^2\left(\boldsymbol{s}^{(\alpha+1)}\right)\right) \\ &+ \frac{(n_{\alpha+1} - n_\alpha)}{4} \mathsf{Var}\left(\boldsymbol{s}^{(\alpha+1)}\right) + \ell_\alpha \cdot (n_{\alpha+1} - n_\alpha) \cdot \frac{\mathfrak{B}_\alpha^2 + 2}{12} \sigma^2_{\mathsf{KSK}_\alpha}. \end{aligned}$$

To obtain the total cost of the algorithm and the total variance of the noise added, we simply iterate from $\alpha = 0, \ldots, \mathsf{nb} - 1$. □

Remark 5.6 (Stairs in the Blind Rotation.)**.** A similar process can be introduced in the blind rotation algorithm. The idea would be, during the blind rotation, to progressively use GLWE partial secret keys (Definition 29) with a smaller filling amount ϕ which will reduce the output

noise of the blind rotate. As with the stair shrinking key switch, we could use different bases and levels in the external products thus offering potentially an overall speed-up. We leave this problem as a topic for future work.

Practical Improvement. The use of shared secret keys brings a practical significant improvement to homomorphic computations: Table 5.1, presents a comparison of our techniques to the state-of-the-art [CJP21]. More detailed experiments are reported in Section 5.5.2.

5.4 Combining Both Techniques

In this section, we provide details on FHE algorithms that benefit from having secret keys that are both partial and with shared randomness.

Partial GLWE secret keys with shared randomness are simply a list of partial GLWE secret keys with some public knowledge about shared coefficients. This type of keys is a combination of shared randomness and partial secret keys, offering advantages of both types.

It is possible to design a faster shrinking key switch (Algorithm 30) which uses partial secret keys (Definition 29). This means that for this faster algorithm, we use both partial secret keys and secret keys with shared randomness. Details about this new procedure is given in Algorithm 36.

Algorithm 36: $\mathsf{ct_{out}} \leftarrow \mathsf{FftShrinkingKeySwitch}(\mathsf{ct_{in}}, \mathsf{KSK})$

Context:
$$\begin{cases} n_{\mathsf{out}} < n_{\mathsf{in}}, \quad n_{\mathsf{in}} - n_{\mathsf{out}} \leq k_{\mathsf{KSK,in}} \cdot N_{\mathsf{KSK}} \text{ and } n_{\mathsf{out}} \leq k_{\mathsf{KSK,out}} \cdot N_{\mathsf{KSK}} \\ \boldsymbol{s}_{\mathsf{out}} \prec \boldsymbol{s}_{\mathsf{in}} : \text{ secret keys shared randomness (Definition 32)} \\ \boldsymbol{s}_{\mathsf{out}} \in \mathbb{Z}_q^{n_{\mathsf{out}}} : \text{the output LWE secret key} \\ \boldsymbol{s} = (s_{n_{\mathsf{out}}}, \ldots, s_{n_{\mathsf{in}}-1}) \in \mathbb{Z}_q^{n_{\mathsf{in}}-n_{\mathsf{out}}} \\ \boldsymbol{s}_{\mathsf{in}} = \boldsymbol{s}_{\mathsf{out}} || \boldsymbol{s} \in \mathbb{Z}_q^{n_{\mathsf{in}}} : \text{the input LWE secret key} \end{cases}$$

Input:
$$\begin{cases} \mathsf{ct_{in}} = (a_0, \ldots, a_{n_{\mathsf{in}}-1}, b) \in \mathrm{LWE}_{\boldsymbol{s}_{\mathsf{in}}}(m) \subseteq \mathbb{Z}_q^{n_{\mathsf{in}}+1}, \text{ where } p \in \mathbb{Z}_q \\ \mathsf{KSK} = \left\{ (\mathsf{KSK}_0, \ldots, \mathsf{KSK}_{k_{\mathsf{in}}-1}) \right\}; \texttt{ /* Definition 15 */} \\ \text{With } \mathsf{KSK}_i \in \mathrm{GLEV}^{\mathcal{B},\ell}_{\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}}(S_{\mathsf{in},i}) \text{ for } i \in [0, k_{\mathsf{in}} - 1] \end{cases}$$

Output: $\mathsf{ct_{out}} \in \mathrm{LWE}_{\boldsymbol{s}_{\mathsf{out}}}(m)$

`/* Split the input LWE ciphertext into two parts: one related to` $\boldsymbol{s}_{\mathsf{out}}$`, and the rest */`

1 Set $\mathsf{ct}_0 := (a_0, \ldots, a_{n_{\mathsf{out}}-1}, b) \in \mathbb{Z}_q^{n_{\mathsf{out}}+1}$

2 Set $\mathsf{ct}_1 := (a_{n_{\mathsf{out}}}, \ldots, a_{n_{\mathsf{in}}-1}, 0) \in \mathbb{Z}_q^{n_{\mathsf{in}}-n_{\mathsf{out}}+1}$

3 Set $\mathsf{ct}'_1 \leftarrow \mathsf{FftLweKeySwitch}(\mathsf{ct}_1, \mathsf{KSK}) \in \mathbb{Z}_q^{n_{\mathsf{out}}+1}$ `/* Call Algorithm 32 */`

4 **return** $\mathsf{ct_{out}} = \mathsf{ct}_0 + \mathsf{ct}'_1$

Theorem 5.10 (Noise & Cost of the FFT-Based Shrinking Key Switch)**.** *We consider the FFT-based LWE shrinking key switching as detailed in Algorithm 36. Its cost can be expressed from the cost of a GLWE-to-GLWE key switch (Remark 5.3) since we neglect the costs of sample extraction and its inverse. The cost is then* $\mathscr{C}(\mathsf{FftShrinkingKeySwitch}) = \mathscr{C}(\mathsf{GlweKeySwitch})$*. Note that* k_{in} *is smaller thanks to the shared randomness property of the secret keys, which leads to a faster procedure. The added noise can be expressed from the noise formula of the GLWE-to-GLWE key switch (Theorem 5.2) which gives* $\mathsf{Var}(\mathsf{FftShrinkingKeySwitch}) = \mathsf{FftError}_{k_{\max},N,\mathcal{B},\ell} + \mathsf{Var}(\mathsf{GlweKeySwitch})$ *with* $\phi_{\mathsf{in}} = n_{\mathsf{out}} - n_{\mathsf{in}}$ *and* $k_{\max} = \max(k_{\mathsf{in}}, k_{\mathsf{out}})$*.*

Proof *(Theorem 5.10). The estimation of the variance of the error is immediate. For the FFT error, we refer to Subsection 2.5.3 and proof of Theorem 5.5.* □

Algorithm 37 summarizes the process to compute a key switch when both approaches are mixed.

Theorem 5.11. *(Noise of GLWE Key Switching With Partial & Shared Randomness Keys) Perform a key switching (Algorithm 37) from* $\mathsf{CT}_{\mathsf{in}} \in \mathfrak{R}_{q,N}^{k_{\mathsf{in}}+1}$ *under the secret key* $\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} \in \mathfrak{R}_{q,N}^{k_{\mathsf{in}}}$*, to* $\mathsf{CT}_{\mathsf{out}} \in \mathfrak{R}_{q,N}^{k_{\mathsf{out}}+1}$ *under the secret key* $\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]} \in \mathfrak{R}_{q,N}^{k_{\mathsf{out}}}$*, where the key are shared and partial, i.e.,* $\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]} \prec \boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]}$*. Each coefficient of the output has added noise estimated as*

$$\begin{aligned}\mathsf{Var}(\mathsf{GlweKeySwitch}') = (\phi_{\mathsf{in}} - \phi_{\mathsf{out}}) \left(\frac{q^2 - \mathcal{B}^{2\ell}}{12\mathcal{B}^{2\ell}}\right) \left(\mathsf{Var}\left(\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]}\right) + \mathbb{E}^2\left(\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]}\right)\right) \\ + \frac{\phi_{\mathsf{in}} - \phi_{\mathsf{out}}}{4} \mathsf{Var}\left(\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]}\right) + \ell(k_{\mathsf{in}} - k_{\mathsf{out}}) N \sigma_{\mathsf{ksk}}^2 \frac{\mathcal{B}^2 + 2}{12}\,.\end{aligned}$$

Algorithm 37: $\mathsf{CT}_{\mathsf{out}} \leftarrow \mathsf{GlweKeySwitch}'(\mathsf{CT}_{\mathsf{in}}, \mathsf{KSK})$

Context:
- $\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} \in \mathfrak{R}_{q,N}^{k_{\mathsf{in}}}$: the input partial secret key (Definition 29)
- $\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} = (S_{\mathsf{in},0}, \ldots, S_{\mathsf{in},k_{\mathsf{in}}-1})$
- $\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]} \in \mathfrak{R}_{q,N}^{k_{\mathsf{out}}}$: the output partial secret key (Definition 29)
- $\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]} = (S_{\mathsf{out},0}, \ldots, S_{\mathsf{out},k_{\mathsf{out}}-1})$
- $(k_{\mathsf{in}} - 1)N < \phi_{\mathsf{in}} \leq k_{\mathsf{in}} N$ and $(k_{\mathsf{out}} - 1)N < \phi_{\mathsf{out}} \leq k_{\mathsf{out}} N$
- $\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]} \prec \boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]}$: secret keys with shared randomness (Definition 32)
- $\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} \neq \boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}$ and $k_{\mathsf{out}} \leq k_{\mathsf{in}}$
- $k \in \{k_{\mathsf{out}} - 1, k_{\mathsf{out}}\}$ such that $\forall 0 \leq i < k,\ S_{\mathsf{in},i} = S_{\mathsf{out},i}$
- $\mathsf{CT}_{i,j} \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}}\left(\frac{q}{\mathcal{B}^j} \cdot S_{\mathsf{in},i}\right)$, for $k_{\mathsf{out}} \leq i < k_{\mathsf{in}}$ & $0 \leq j < \ell$
- if $k = k_{\mathsf{out}} - 1$:
  - $\mathsf{CT}_{k,j} \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}}\left(\frac{q}{\mathcal{B}^j} \cdot (S_{\mathsf{in},k} - S_{\mathsf{out},k})\right)$, for $0 \leq j < \ell$
- $\ell \in \mathbb{N}$: the number of levels in the decomposition
- $\mathcal{B} \in \mathbb{N}$: the base in the decomposition

Input:
- $\mathsf{CT}_{\mathsf{in}} = (A_0, \ldots, A_{k_{\mathsf{in}}-1}, B) \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]}}(M)$
- $\mathsf{KSK} = \{\boldsymbol{K}_i = (\mathsf{CT}_{i,0}, \ldots, \mathsf{CT}_{i,\ell-1})\}_{k \leq i < k_{\mathsf{in}}}$

Output: $\mathsf{CT}_{\mathsf{out}} \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}}(M)$

```
/* Keep the B part and the first part of the mask                    */
```
Set $\mathsf{CT}_{\mathsf{out}} := (A_0, \ldots, A_{k_{\mathsf{out}}-1}, B) \in \mathfrak{R}_{q,N}^{k_{\mathsf{out}}+1}$

```
/* Different public material for this potential partial-shared secret key
   polynomial                                                        */
```
if $k = k_{\mathsf{out}} - 1$ **then**
  Update $\mathsf{CT}_{\mathsf{out}} = \mathsf{CT}_{\mathsf{out}} - \left\langle \boldsymbol{K}_k, \mathbf{Dec}^{(\mathcal{B},\ell)}(A_k) \right\rangle$

for $i \in [k_{\mathsf{out}}; k_{\mathsf{in}} - 1]$ **do**
  ```
  /* Decompose the mask                                              */
  ```
  Update $\mathsf{CT}_{\mathsf{out}} = \mathsf{CT}_{\mathsf{out}} - \left\langle \boldsymbol{K}_i, \mathbf{Dec}^{(\mathcal{B},\ell)}(A_i) \right\rangle$

return $\mathsf{CT}_{\mathsf{out}}$

Proof *(Theorem 5.11). Lets consider two shared and partial secret keys such that* $\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]} \prec \boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]}$*. We have* $\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]} = (S_{\mathsf{out},0}, \ldots, S_{\mathsf{out},k_{\mathsf{out}}-1})$*, where* $S_{\mathsf{out},k_{\mathsf{out}}-1} = \sum_{i=0}^{\phi_{\mathsf{out}}-(k_{\mathsf{out}}-1)N-1} s_{\mathsf{out},k_{\mathsf{out}}-1,i} X^i$ *we call* $S_{\mathsf{out},k_{\mathsf{out}}-1}$*:* $\underline{S}$*.*

We have $\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} = (S_{\mathsf{in},0}, \ldots, S_{\mathsf{in},k_{\mathsf{in}}-1})$ *such that for all* $j \in [0, k_{\mathsf{out}} - 1)$*,* $S_{\mathsf{out},j} = S_{\mathsf{in},j}$ *and* $S_{\mathsf{in},k_{\mathsf{out}}-1} = \underline{S} + \bar{S}$ *where* $\bar{S} = \sum_{j=\phi_{\mathsf{out}}-(k_{\mathsf{out}}-1)N}^{N-1} s_{\mathsf{in},k_{\mathsf{out}}-1,j} X^j$*.*

The inputs of a GLWE key switching with partial & shared randomness keys (Algorithm 37)

are:

- *The input GLWE ciphertext:* $\mathsf{CT}_{\mathsf{in}} = (\boldsymbol{A}_{\mathsf{in}}, B_{\mathsf{in}}) \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]}}(\Delta \cdot M) \subseteq \mathfrak{R}_{q,N}^{k_{\mathsf{in}}+1}$, *where* $B_{\mathsf{in}} = \sum_{i=0}^{k_{\mathsf{in}}-1} A_{\mathsf{in},i} \cdot S_{\mathsf{in},i} + \Delta \cdot M + E_{\mathsf{in}}$, $A_{\mathsf{in},i} = \sum_{j=0}^{k-1} a_{i,j} \cdot X^j \hookleftarrow \mathcal{U}(\mathfrak{R}_{q,N})$ *for all* $i \in [0,k)$ *and* $E_{\mathsf{in}} = \sum_{j=0}^{k-1} e_j \cdot X^j$, *and* $e_j \hookleftarrow \mathcal{N}_{\sigma_{\mathsf{in}}^2}$ *for all* $j \in [0, N-1)$.
- *The key switch key:* $\mathsf{KSK} = (\mathsf{KSK}_{k_{\mathsf{out}}-1}, \mathsf{KSK}_{k_{\mathsf{out}}} \cdots, \mathsf{KSK}_{k_{\mathsf{in}}-1})$, *where* $\mathsf{KSK}_i \in \mathrm{GLEV}_{\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}}(S_{\mathsf{in},i}) = \left(\mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}}\left(\frac{q}{\mathcal{B}} S_{\mathsf{in},i}\right), \ldots, \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}}\left(\frac{q}{\mathcal{B}^\ell} S_{\mathsf{in},i}\right)\right)$ *for all* $k_{\mathsf{out}} \leq i < k_{\mathsf{in}}$, *and* $\mathsf{KSK}_{k_{\mathsf{out}}-1} \in \mathrm{GLEV}_{\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}}(\bar{S}) = \left(\mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}}\left(\frac{q}{\mathcal{B}} \bar{S}\right), \ldots, \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}}\left(\frac{q}{\mathcal{B}^\ell} \bar{S}\right)\right)$
  We note by $\mathsf{KSK}_{i,j} = (\boldsymbol{A}_{i,j}, B_{i,j}) \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}}\left(\frac{q}{\mathcal{B}^{j+1}} S_{\mathsf{in},i}\right)$, *for all* $k_{\mathsf{out}} \leq i < k_{\mathsf{in}}$ *for all* $0 \leq j < \ell$, *where* $B_{i,j} = \sum_{\tau=0}^{k_{\mathsf{out}}-1} A_{i,j,\tau} \cdot S_{\mathsf{out},\tau} + \frac{q}{\mathcal{B}^{j+1}} S_{\mathsf{in},i} + E_{\mathsf{ksk},i,j}$, *and* $E_{\mathsf{ksk},i,j} = \sum_{\tau=0}^{N-1} e_{\mathsf{ksk},i,j,\tau} \cdot X^\tau$ *and* $e_{\mathsf{ksk},i,j,m} \hookleftarrow \mathcal{N}_{\sigma_{\mathsf{ksk}}^2}$.
  We note $\mathsf{KSK}_{k_{\mathsf{out}}-1,j} = (\boldsymbol{A}_{k_{\mathsf{out}}-1,j}, B_{k_{\mathsf{out}}-1,j}) \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}}\left(\frac{q}{\mathcal{B}^{j+1}} \bar{S}\right)$ *for all* $0 \leq j < \ell$, *where* $B_{k_{\mathsf{out}}-1,j} = \sum_{\tau=0}^{k_{\mathsf{out}}-1} A_{k_{\mathsf{out}}-1,j,\tau} \cdot S_{\mathsf{out},\tau} + \frac{q}{\mathcal{B}^{j+1}} \bar{S} + E_{\mathsf{ksk},k_{\mathsf{out}}-1,j}$, *and* $E_{\mathsf{ksk},k_{\mathsf{out}}-1,j} = \sum_{\tau=0}^{N-1} e_{\mathsf{ksk},k_{\mathsf{out}}-1,j,\tau} \cdot X^\tau$ *and* $e_{\mathsf{ksk},k_{\mathsf{out}}-1,j,m} \hookleftarrow \mathcal{N}_{\sigma_{\mathsf{ksk}}^2}$.

The output of this algorithm is: $\mathsf{CT}_{\mathsf{out}} = (\boldsymbol{A}_{\mathsf{out}}, B_{\mathsf{out}}) \in \mathrm{GLWE}_{\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}}(\Delta \cdot M) \subseteq \mathfrak{R}_{q,N}^{k_{\mathsf{out}}+1}$.
By definition, for any polynomial $A_{\mathsf{in},i}$, *we have the decomposition (described in Definition 5),* $\mathbf{Dec}^{(\mathcal{B},\ell)}(A_{\mathsf{in},i}) = \left(\tilde{A}_{\mathsf{in},i,1}, \ldots, \tilde{A}_{\mathsf{in},i,\ell}\right)$ *such that* $\tilde{A}_{\mathsf{in},i} = \sum_{j=0}^{\ell-1} \frac{q}{\mathcal{B}^{j+1}} \tilde{A}_{\mathsf{in},i,j}$. *Now, we can decrypt:*

$$
\begin{aligned}
B_{\mathsf{out}} - \left\langle \boldsymbol{A}_{\mathsf{out}}, \boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]} \right\rangle &= \left\langle (\boldsymbol{A}_{\mathsf{out}}, B_{\mathsf{out}}), \left(-\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}, 1\right) \right\rangle \\
&= \Big\langle (A_{\mathsf{in},0}, \ldots, A_{\mathsf{in},k_{\mathsf{out}}-1}, 0 \cdots, 0, B_{\mathsf{in}}) - \mathbf{Dec}^{(\mathcal{B},\ell)}(A_{\mathsf{in},k_{\mathsf{out}}-1})\, \mathsf{KSK}_{k_{\mathsf{out}}-1} \\
&\quad - \sum_{i=k_{\mathsf{out}}}^{k_{\mathsf{in}}-1} \mathbf{Dec}^{(\mathcal{B},\ell)}(A_{\mathsf{in},i})\, \mathsf{KSK}_i, \left(-\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}, 1\right) \Big\rangle \\
&= B_{\mathsf{in}} - \sum_{i=0}^{k_{\mathsf{out}}-1} A_{\mathsf{in},i} S_{\mathsf{out},i} - \sum_{j=0}^{\ell-1} \tilde{A}_{\mathsf{in},k_{\mathsf{out}}-1,j} \left\langle \mathsf{KSK}_{k_{\mathsf{out}}-1,j}, \left(-\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}, 1\right) \right\rangle \\
&\quad - \sum_{i=k_{\mathsf{out}}}^{k_{\mathsf{in}}-1} \sum_{j=0}^{\ell-1} \tilde{A}_{\mathsf{in},i,j} \left\langle \mathsf{KSK}_{i,j}, \left(-\boldsymbol{S}_{\mathsf{out}}^{[\phi_{\mathsf{out}}]}, 1\right) \right\rangle \\
&= B_{\mathsf{in}} - \sum_{i=0}^{k_{\mathsf{out}}-2} A_{\mathsf{in},i} S_{\mathsf{in},i} - A_{\mathsf{in},k_{\mathsf{out}}-1} \underline{S} - \sum_{j=0}^{\ell-1} \tilde{A}_{\mathsf{in},k_{\mathsf{out}}-1,j} \left(\frac{q}{\mathcal{B}^{j+1}} \bar{S} + E_{\mathsf{ksk},k_{\mathsf{out}}-1,j}\right) \\
&\quad - \sum_{i=k_{\mathsf{out}}}^{k_{\mathsf{in}}-1} \sum_{j=0}^{\ell-1} \tilde{A}_{\mathsf{in},i,j} \left(\frac{q}{\mathcal{B}^{j+1}} S_{\mathsf{in},i} + E_{\mathsf{ksk},i,j}\right) \\
&= \underbrace{B_{\mathsf{in}} - \sum_{i=0}^{k_{\mathsf{out}}-1} A_{\mathsf{in},i} S_{\mathsf{in},i} - A_{\mathsf{in},k_{\mathsf{out}}-1} \underline{S}}_{(I)} \underbrace{- \tilde{A}_{\mathsf{in},k_{\mathsf{out}}-1} \bar{S} - \sum_{j=0}^{\ell-1} \tilde{A}_{\mathsf{in},k_{\mathsf{out}}-1,j} \cdot E_{\mathsf{ksk},k_{\mathsf{out}}-1,j}}_{(II)} \\
&\quad \underbrace{- \sum_{i=k_{\mathsf{out}}}^{k_{\mathsf{in}}-1} \tilde{A}_{\mathsf{in},i} S_{\mathsf{in},i} - \sum_{i=k_{\mathsf{out}}}^{k_{\mathsf{in}}-1} \sum_{j=0}^{\ell-1} \tilde{A}_{\mathsf{in},i,j} \cdot E_{\mathsf{ksk},i,j}}_{(III)}.
\end{aligned}
$$

After decrypting, we can split the previous result in three distinct part and analyze the noise provide by each of them. The first part of the result (term (I)*) is only composed of the noise present in the* B_{in}.
The second part of the result (term (II)*) can be seen as a key switching with partial key (Algorithm 30) from* $\bar{S}$ *to* S_{out}. *The proof of noise add by this part follows the proof of Theorem 5.2.*
As for the second part of the result, the third part of the result (term (III)*) can be seen as a key switching with partial key (Algorithm 30) from* $(S_{\mathsf{in},k_{\mathsf{out}}}, \ldots, S_{\mathsf{in},k_{\mathsf{in}}-1})$ *to* S_{out}. *The proof of noise add by this part follows as well the proof of Theorem 5.2.*
By adding this different noises, we will obtain $\mathsf{Var}(e_{\mathsf{out}}) = \mathsf{Var}(I) + \mathsf{Var}(II) + \mathsf{Var}(III)$ *where:*

$$
\begin{aligned}
\mathsf{Var}(I) &= \sigma_{\mathsf{in}}^2 \\
\mathsf{Var}(II) &= (Nk_{\mathsf{out}} - \phi_{\mathsf{out}}) \left(\frac{q^2 - \mathcal{B}^{2\ell}}{12\mathcal{B}^{2\ell}} \right) \left(\mathsf{Var}\left(\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} \right) + \mathbb{E}^2 \left(\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} \right) \right) \\
&\quad + \frac{Nk_{\mathsf{out}} - \phi_{\mathsf{out}}}{4} \mathsf{Var}\left(\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} \right) + \ell N \sigma_{\mathsf{ksk}}^2 \frac{\mathcal{B}^2 + 2}{12} \\
\mathsf{Var}(III) &= (\phi_{\mathsf{in}} - Nk_{\mathsf{out}}) \left(\frac{q^2 - \mathcal{B}^{2\ell}}{12\mathcal{B}^{2\ell}} \right) \left(\mathsf{Var}\left(\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} \right) + \mathbb{E}^2 \left(\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} \right) \right) \\
&\quad + \frac{\phi_{\mathsf{in}} - Nk_{\mathsf{out}}}{4} \mathsf{Var}\left(\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} \right) + \ell(k_{\mathsf{in}} - k_{\mathsf{out}} - 1) N \sigma_{\mathsf{ksk}}^2 \frac{\mathcal{B}^2 + 2}{12}.
\end{aligned}
$$

To conclude we have:

$$
\begin{aligned}
\mathsf{Var}(e_{\mathsf{out}}) &= \sigma_{\mathsf{in}}^2 + (\phi_{\mathsf{in}} - \phi_{\mathsf{out}}) \left(\frac{q^2 - \mathcal{B}^{2\ell}}{12\mathcal{B}^{2\ell}} \right) \left(\mathsf{Var}\left(\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} \right) + \mathbb{E}^2 \left(\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} \right) \right) \\
&\quad + \frac{\phi_{\mathsf{in}} - \phi_{\mathsf{out}}}{4} \mathsf{Var}\left(\boldsymbol{S}_{\mathsf{in}}^{[\phi_{\mathsf{in}}]} \right) + \ell(k_{\mathsf{in}} - k_{\mathsf{out}}) N \sigma_{\mathsf{ksk}}^2 \frac{\mathcal{B}^2 + 2}{12}.
\end{aligned}
$$

□

5.5 Parameters & Benchmarks

In this section, we describe how to generate FHE parameters for all our experiments. We use the procedure introduced in Chapter 2, Section 2.5 to compare the different approaches. To demonstrate the impact of partial and/or secret keys with shared randomness, we use the Atomic Pattern (AP) (Definition 26) called CJP (the name coming from the paper [CJP21]). After recalling its definition, we explain how to optimize parameters for the different experiments and show the different improvements (both in computational time and size of public material) brought forward by each of the new procedures introduced.

Real life applications use additions and multiplications by public integers (i.e., a dot product) between two consecutive bootstrappings. Formally, given a list of ciphertexts $\{\mathsf{ct}_i\}_{i\in[1,\alpha]} \in (\mathrm{LWE}_{\boldsymbol{s}_{\mathsf{in}}})^\alpha$ (with independent noise values) and a list of integers $\{\omega_i\}_{i\in[1,\alpha]} \in \mathbb{Z}^\alpha$, one computes $\sum_{i=1}^{\alpha} \omega_i \cdot \mathsf{ct}_i$. In that case, we have $\nu^2 = \sum_{i=1}^{\alpha} \omega_i^2$ and ν is used to fully describe the noise growth during a dot product (Theorem 2.7). We set $\nu = 2^p$ where p is the precision of the message. For every experiment below, the probability of failure is set to $p_{\mathsf{fail}} \leq 2^{-13.9}$. Note that with the FHE parameter generation process, any other probability can be chosen. In what follows, we use the CJP atomic pattern which denotes the chaining of a dot product, a key switch and a PBS.

All of the experiments presented have been carried out on AWS with a m6i.metal instance Intel Xeon 8375C (Ice Lake) at 3.5 GHz, with 128 vCPUs and 512.0 GiB of memory using the TFHE-rs library [Zam22][1]. In Appendix A.2 (Tables A.16,A.17,A.18 and A.19), we give the parameter sets used for the experiments reported in Table 5.2 along with benchmarks and public material sizes.

5.5.1 Partial GLWE Secret Key

We conduct three experiments with partial GLWE secret keys (Definition 29) that are displayed in Table 5.1. This shows the cost estimated by the optimizer (divided by 10^6) in function of the precision.

Our baseline is CJP. The first experiment focuses on the CJP atomic pattern where the GLWE secret key could be partial with a filling amount ϕ. During optimization, we set ϕ to the minimum between $k \cdot N$ and the value n_{plateau} discussed in limitation 2. As expected, this is mostly better with larger precisions, starting at $p = 6$ where the plateau is reached.

The second experiment considers the CJP atomic pattern where the traditional LWE-to-LWE key switch is replaced with the FFT-base LWE key switch introduced in Algorithm 32. During the

[1] `https://github.com/zama-ai/tfhe-rs/tree/artifact_ccs_2024`

optimization, we had to introduce new FHE parameters for this particular key switch: an input GLWE dimension k_{in}, an output GLWE dimension k_{out} and a polynomial size N_{KS}. We observe a significant improvement for all precisions when using this key switch, but it is more visible with smaller precisions, between 1 and 6.

The third and last experiment is the combination of the two first ones: we allow the GLWE secret key to be partial (when the plateau is reached) and use the FFT-based LWE key switch (Algorithm 32). As expected, this last experiment outperforms the other two. We can see a significant improvement for all precisions.

Note that there is no way to build an LWE-to-LWE key switch based on the FFT without partial secret keys, so no comparison with our results can be done.

5.5.2 Secret Keys with Shared Randomness

We conduct two experiments with secret keys with shared randomness (Definition 32), and we display the results predicted by an optimizer in Table 5.1.

The first experiment is the CJP atomic pattern where we allow the secret keys to share their randomness using the shrinking LWE key switch described in Algorithm 34. We observe a significant improvement with small precisions, up to $p = 6$.

The second and last experiment is the CJP atomic pattern where we allow the secret keys to share their randomness, so we can use the 2-step stair LWE key switch from Algorithm 35. We see a significant improvement at all precisions. Note that if one tries to trivially have a 2-step stair key switch without any shared randomness, the computational cost is basically the same as in CJP.

Precision & 2−norm	1		2		3		4		5		6		8		10	
	Cost	Gain	Cost	Gain	Cost	Gain	Cost	Gain	Cost	Gain	Cost	Gain	Cost	Gain	Cost	Gain
CJP	31	–	42	–	63	–	78	–	118	–	347	–	2351	–	20813	–
Partial: BSK	31	−0%	42	−0%	63	−0%	78	−0%	118	−0%	318	−8%	1934	−18%	16449	**−21%**
Partial: FFT−KS	25	−18%	33	−21%	41	**−35%**	62	−21%	92	−22%	298	−14%	2053	−12%	18841	−9%
Partial: BSK + FFT−KS	25	−18%	33	−21%	41	**−35%**	62	−21%	92	−22%	285	−17%	1879	−20%	16426	−21%
Shared: Shrinking−KS	29	−9%	39	−8%	49	**−23%**	72	−8%	105	−11%	336	−3%	2331	−0.8%	20785	−0.1%
Shared Stair−KS	27	−15%	35	−17%	42	**−32%**	66	−17%	94	−20%	316	−9%	2057	−12%	16624	−20%

Table 5.1: Comparison in terms of estimated execution time, between traditional CJP, our baseline, two variants of CJP based on secret keys with shared randomness and three variants based on partial secret keys.

5.5.3 Combining Both

We conduct two experiments with both partial (Definition 29) and secret keys with shared randomness (Definition 32). As previously, the blue dashed curve with the • symbol shows the CJP baseline.

The first experiment is the CJP atomic pattern where we allow the secret keys to be partial and to share their randomness. We use the 2-step stair LWE key switch from Algorithm 35 and we allow the GLWE secret key to be partial when the plateau is reached. This is the red solid curve with the + symbol. We see a definite improvement at all precisions.

The second and last experiment also focuses on the CJP atomic pattern where we allow secret keys to be partial and to share their randomness. We allow the GLWE secret key to be partial (when the plateau is reached), and use the FFT-based LWE key switch (Algorithm 36) since our secret keys also share randomness. On the figure, it is the green dotted curve with the ▼ symbol. We see a similar improvement at all precisions.

We plot the timings obtained with benchmarks in Figures 5.2 and 5.3 to validate our predictions. Both the stair key switch curve and the FFT shrinking key switch curve are below our baseline as predicted, and we have even better results with the FFT shrinking key switch than expected. Note that at precision $p = 3$ we have a 2.4 speed-up factor compared to the baseline (Figure 5.2).

Our new secret key generation also has the advantage to reduce the key sizes. For those experiments, we plot the size of the public material needed in Figure 5.5.3, to demonstrate their

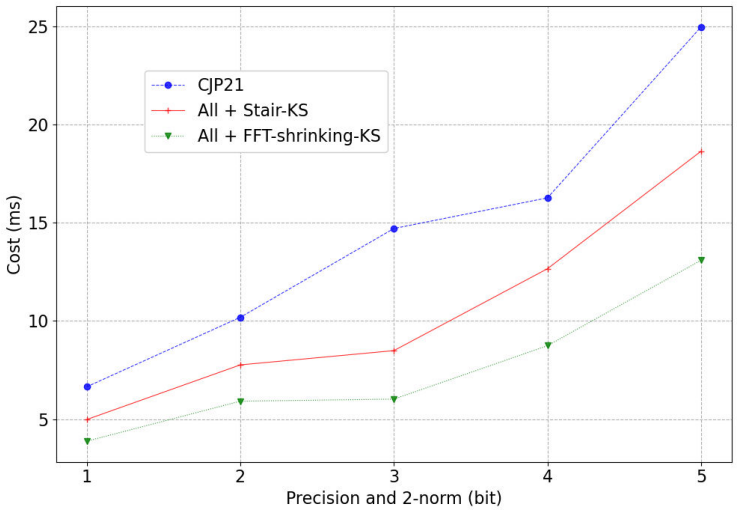

Figure 5.2: Comparison in terms of time of computation of the traditional CJP, our baseline, with two variants of CJP based on both partial secret keys and secret keys with shared randomness. Details can be found in Section 5.5.3 and exact plotted values can be found in Tables A.16, A.17, A.18 and A.19 in Appendix.

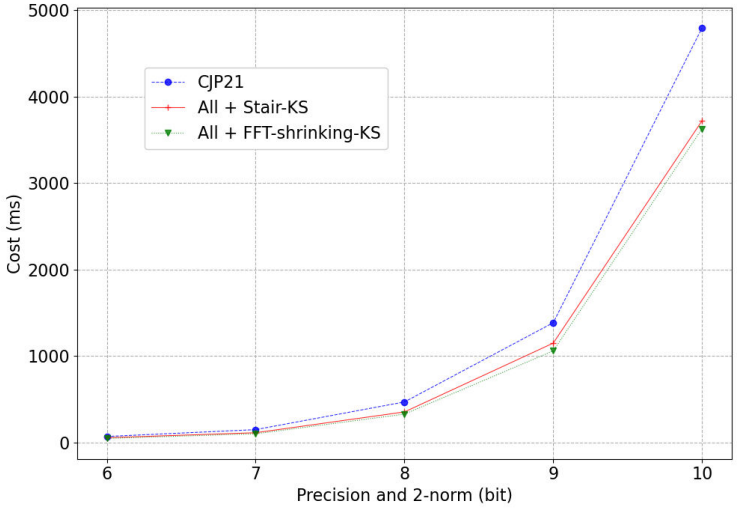

Figure 5.3: Comparison in terms of time of computation, between traditional CJP, our baseline, and two variants of CJP based on both partial secret keys and secret keys with shared randomness. Details can be found in Section 5.5.3 and exact plotted values can be found in Tables A.16, A.17, A.18 and A.19 in Appendix.

benefit in this matter. For instance, the storage needed for the public material when $p = 3$ is going from approximately 105 MB with the CJP method, to 50 MB with the FFT-based approach.

5.6 Some Higher Level Applications

Through Sections 5.2.2, 5.3.2 and 5.4, we discussed the many advantages of using partial and/or secret keys with shared randomness. We now discuss the advantages at a somewhat high level.

Key Switching Key Compression. When one deploys an FHE instance using the shared

Precision & 2-norm	1		2		3		4		5		6		8		10	
	Time	Gain	Time	Gain	Time	Gain	Time	Gain	Time	Gain	Time	Gain	Time	Gain	Time	Gain
CJP	5.43	–	8.75	–	12.2	–	12.6	–	20.0	–	55.6	–	415	–	4710	–
All + Stair−KS	3.78	−30%	6.28	−28%	6.22	**−49%**	9.35	−25%	13.8	−31%	44.3	−20%	323	−22%	3620	−23%
All + FFT shrinking−KS	3.27	−39%	5.32	−39%	5.12	**−58%**	7.38	**−41%**	11.0	**−45%**	41.1	−26%	306	−26%	3603	-23%

Table 5.2: Comparison in terms of computational time (in ms) of the traditional CJP, our baseline, with two variants of CJP based on both partial secret keys and secret keys with shared randomness.

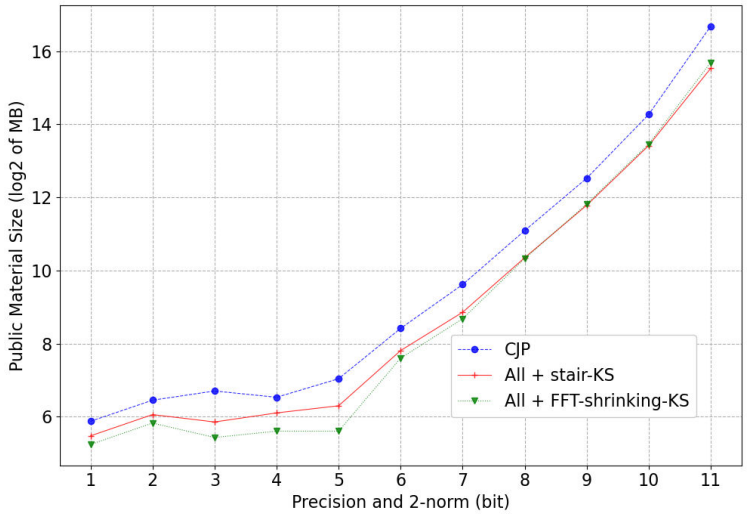

Figure 5.4: Comparison between traditional CJP and two variants of CJP based on both partial secret keys and secret keys with shared randomness. More details in Section 5.5.3 and exact plotted values in Tables A.16, A.17, A.18 and A.19 in Appendix.

randomness property, the total amount of public material for key switching is smaller than usual. Indeed, this only requires to generate all the shrinking key switching keys (Algorithm 34), from the largest key to the smallest. All of these shrinking key switching keys are way smaller than the sum of all the traditional key switching keys that are usually needed. Note that it is possible to provide more levels in some of the key switching keys, and only use the ones that are needed at a moment for a given noise constraint.

Compressed Bootstrapping Key. Similarly, with secret keys with shared randomness, the amount of public material for bootstrapping keys can be reduced. A bootstrapping key is a list of GGSW ciphertexts, each one encrypting a secret key coefficient of the input LWE secret key. Then, giving the GGSW ciphertexts for the largest LWE secret key of the instance is enough. Whenever bootstrapping an LWE ciphertext with a smaller dimension is required, one will only use the first part of the bootstrapping key. In the same spirit, additional levels can be added, and only used when strictly needed.

Easier Parameter Set Conversion. Later, Chapter 7, Section 7.3.4, considers use-cases where there are a couple of coexisting parameter sets, and it is necessary to move from one to the other. Using shared (and partial) secret keys helps converting more efficiently ciphertexts between two (or more) parameter sets. This is due to the removing of some key switchings and limiting the noise growth.

Multikey Compatibility. Both the partial and shared randomness properties are preserved in the MK-FHE (such as [KKL+22, KMS22]) and in threshold-FHE approaches. Indeed, summing two partial secret keys results in another partial secret key, and summing two pairs of secret keys with shared randomness together results in a new pair of secret keys with shared randomness. Those new secret keys could improve the performance of MK-FHE and threshold-FHE, which are in general less efficient that the ones of (single key) FHE, as well as reduce the total size of the

public material.

Other FHE Schemes. Partial and secret keys with shared randomness could be used in other FHE schemes such as FHEW [DM15] or NTRU-based schemes (such as [BIP+22]). This types of keys could also be used in BFV [Bra12, FV12] or CKKS [CKKS17] when larger polynomials are required for the same modulus q, for instance.

Combination With Fixed Hamming Weight. Both partial and secret keys with shared randomness could be instantiated with a fixed Hamming weight We do not explore this topic any further here.

LWE Encryption Public Key With GLWE Material. If one wants to take advantage of the FFT to encrypt fresh LWE ciphertexts with a secret key $\boldsymbol{s} \in \mathbb{Z}_q^n$, and/or shrink the size of ciphertexts with partial GLWE secret key, it is possible to provide a GLWE encryption public key for a partial GLWE secret key $\boldsymbol{S}^{[\phi=n]} \in \mathfrak{R}_{q,N}^k$ such that its flattened version is actually $\boldsymbol{s}$. In this case, one uses GLWE encryption and applies a sample extract right after that to obtain the desired LWE ciphertext.

Chapter 6

Removing the Padding bit

Section 3.3 presented several state-of-the-art algorithms designed to address the limitations introduced in Section 2.6. These contributions include algorithm that either avoid the requirement for padding bits (Limitation 1), facilitate computations at higher precision levels (Limitation 4), or support multi-input ciphertext evaluations (Limitation 7).

In this chapter, we introduce a new algorithm that simultaneously resolves all the three limitations: Limitation 1, 4, and 7. We start by providing a detailed description of this new algorithm and present an in-depth comparison with state-of-the-art techniques, demonstrating a significant improvement for high precision. Overcoming these limitations allows for more efficient deployment of novel FHE algorithms and is essential for advanced constructions, such as those described in Chapters 7 and 8.

6.1 Introduction

As discussed in Section 2.4, the **PBS** takes a single LWE ciphertext as input and outputs one LWE ciphertext corresponding to the LUT evaluation of the encrypted input message. However, when multiple messages are encoded in multiple LWE ciphertexts, or when we need to evaluate a function with multiple input values, a single **PBS** is not enough and can not easily perform multi-input evaluation. The Tree-PBS method proposed in 2021 by Guimarães, Borin and Aranha [GBA21] (Algorithm 26), enables us to evaluate a large lookup table over multiple input ciphertexts. For completeness, we provide details about how to use this technique for large homomorphic integers in 7.3.3. The Tree-PBS is a valid solution for the evaluation of generic LUT for multiple input ciphertexts, but its complexity increases exponentially with the number of ciphertexts. Additionally, the Tree-PBS technique uses the classical **PBS** [CGGI20, CJP21], which has the constraints on the bit of padding and on the small precision of the messages.

This chapter focuses on solving Limitation 1, 4 and 7 while improving the previous solutions presented in Sections 3.3. First, we will compare the current solution proposed in the state-of-the-art in Section 6.2. Then, in Section 6.3, we will present our new **WoP-PBS** algorithm. This new technique consists of using the bit extract (Algorithm 15) followed by circuit bootstrapping (Algorithm 12) to obtain GGSW ciphertexts encrypting the bit decomposition of each message. With these GGSW ciphertexts, we can evaluate generic LUTs on large integers. This approach scales more efficiently than Tree-PBS and removes constraints on padding bits for high-precision inputs.

6.2 Comparison Between $\mathscr{A}^{(\mathbf{CJP}21)}$ and $\mathscr{A}^{(\mathbf{GBA}21)}$

In Chapter 2, we introduced various **PBS** techniques. In particular, Section 3.4 presented several bootstrapping methods that allow the evaluation of multiple lookup tables, including the boot-

strapping proposed in [GBA21] (Algorithm 26). This bootstrapping method is particularly efficient for evaluating multiple ciphertexts representing high precision message. In this section, we compare this approach with the classical **PBS**.

To begin, we define an additional atomic pattern type, denoted `apGBA`, in Definition 26. This pattern is composed of a dot product, a key switch, and the Tree-PBS procedure introduced in [GBA21]. The only way to compare the PBS of [CGGI20] in apCJP and the tree-PBS in apGBA is by solving equation 2.3 for the two types of atomic patterns with a range of 2-norms and a range of message precision, and finally plot the results.

In the Figure 6.1, we display the comparison between apCJP and apGBA for 4 distinct 2-norms and for message precision in $\{2^1, \ldots, 2^{24}\}$. The padding bit is not included in the message precision.

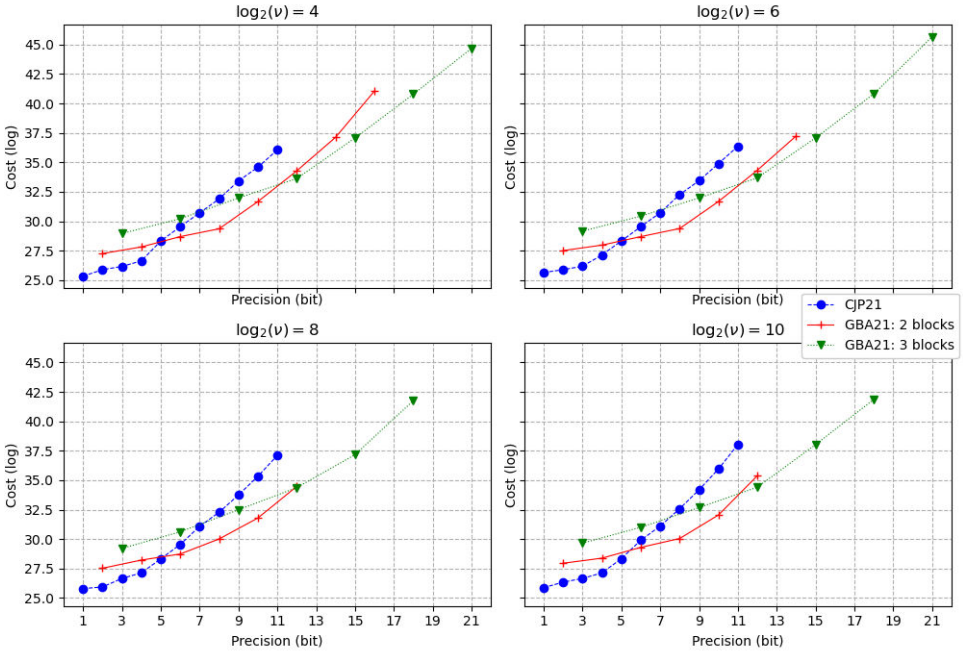

Figure 6.1: In this figure, we compare the cost of AP type apCJP and AP of type apGBA with 2 and 3 blocks.

In this experiment, we choose $\mathcal{P}(N) = \{2^1, \ldots, 2^{18}\}$, the search space of the polynomial size N. We set $q = 2^{64}$ and we used a probability of failure $p_{\mathsf{fail}} \approx 2^{-35}$ and one bit of padding (i.e. $\pi = 1$).

Remark 6.1 (Noise Bound)**.** For apCJP, the noise bound (definition 23) is defined as $t(p,1) = \frac{q}{2^{1+1} \cdot p \cdot z^*(p_{\mathsf{fail}})}$.

For apGBA, the noise bound needs to be computed differently because this AP with 2 blocks (respectively 3 blocks) involves η_2 (respectively η_3) PBS, all sources of potential failures.

$$\eta_i = i \cdot \frac{p^{i-1} - 1}{p - 1} + 1, \text{ with } i \text{ the number of blocks.}$$

To guarantee a global failure probability for one apGBA, the noise bound needs to be computed from the number η_i of PBS. We start by computing the failure probability needed for one PBS defined as $p'_i = 1 - (1 - p_{\mathsf{fail}})^{\frac{1}{\eta_i}}$ and from it we can finally compute the noise bound for each PBS $t(p,1) = \frac{q}{2^{1+1} \cdot p \cdot z^*(p'_i)}$.

The first takeaway is that TFHE bootstrapping (in atomic pattern apCJP, blue/• curve) can only handle messages up to 11 bits of precision. By using these parameters set, the cost of this

atomic pattern with regards to the precision is an exponential function in two parts. For precisions above 4 to 5 bits (padding bit not included), adding a bit of precision more than doubles the cost, indeed the polynomial size doubles for every additional bit of precision. TFHE PBS does not scale well with the precision, to maximize efficiency, it should not be used when the messages have more than 5 bits of precision.

For apGBA, we used on the first layer the multi-value PBS introduced in [CIM19] and we used PBS over encrypted lookup tables [CGGI20] on the other layers. The tree-PBS of [GBA21] takes as input a vector of ciphertexts each containing part of the message. The red/+ curve (respectively green/▼ curve) represents the cost to compute a tree-PBS over 2 ciphertexts (respectively 3 ciphertexts) each one containing a chunk of the message. Using this, we can reach precisions that are not feasible with the bootstrapping from [CGGI20]. Above 11 bits, we cannot find parameters that will guarantee the correctness of apCJP. Regarding the tree-PBS with 2 blocks, it becomes interesting in term of cost with 6 bits of precision or more, and offers parameters up to 16 bits of precision. For higher precision, no feasible solution could be found. The tree-PBS with 3 blocks provides a way to go above that and we found solutions for precision up to 21 bits. It is more efficient than the other two starting at 10 bits of precision. It is important to notice that even if solutions exist, computing apGBA over message of 21 bits costs more than 2^{20} times the cost of [CGGI20] PBS over Boolean messages.

To conclude this comparison, [CGGI20]'s bootstrapping used as in [CJP21] (i.e., with a KS before and not after) is the best way to apply a function over message of small precision (1 to 5 bits). For precision above 11 bits, we have to use the tree-PBS in [GBA21]. But as we can see in the figures, we need an algorithm more efficient than [GBA21] when it becomes too expensive, i.e., above 9 bits, especially if one wants to build efficient operations over larger homomorphic integers with TFHE and still being able to compute LUTs on them.

6.3 Multi-Input Lookup Table Evaluation

In this section, we present a new without padding **PBS** (**WoP-PBS**) that is able to take as input not only one LWE ciphertext but several, it is able to round (or truncate or more) each of the input messages to a given precision, and it can be used to compute several LUT on the same set of inputs at the cost of (about) a single LUT.

Our method is based on two building blocks: the circuit bootstrapping (Algorithm 12) and the mixed (or vertical or horizontal) packing from [CGGI20] (Algorithm 14) presented in Chapter 3. In practice, the algorithm executes the following steps:

- It starts by using generalized **PBS** [CLOT21], evaluating a scaled sign function (negacyclic), and homomorphic subtraction to extract all the bits of each encrypted message. Each bit is output as a LWE ciphertext (see Algorithm 15).
- It converts each of the LWE ciphertexts extracted by previous step into GGSW ciphertexts, by using circuit bootstrapping [CGGI20] (see Algorithm 12).
- It uses the GGSW ciphertexts from previous step to evaluate the LUT as a mixed (or vertical or horizontal) packing [CGGI20] (see Algorithm 14): it consists in practice in a CMux tree, followed by a blind rotation and one (or several) sample extraction.

The cleartext representation of the new **WoP-PBS** is presented in Figure 6.2.

In general, the circuit bootstrapping is the most expensive part of the algorithm (each circuit bootstrapping requires several **PBS**s, each followed by several functional key switchings). Since the number of circuit bootstrapping corresponds to the number of bits composing the input message, the technique generally scales linearly in the size of the input message. However, after a certain input size, the mixed packing stops being negligible and becomes as costly (or even more) than the circuit bootstrapping part: roughly speaking, this happens when the number of CMuxes in the mixed packing part becomes as big as the number of CMuxes in the **PBS**s computed inside the circuit bootstrappings (e.g., for the parameter sets that we use in our experiments, this happens when the input size is about 28 bits).

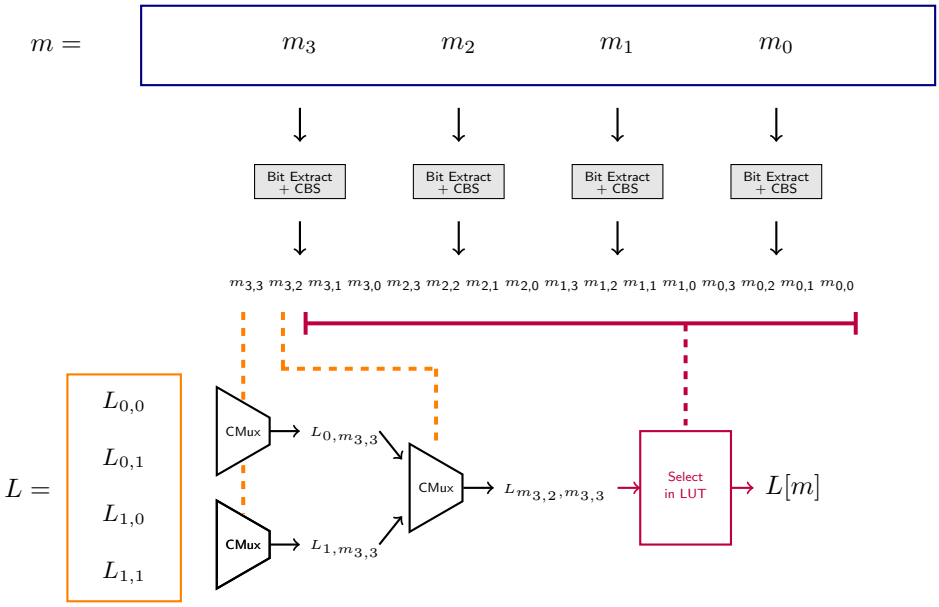

Figure 6.2: Cleartext evaluation of the new **WoP-PBS** (toy example). The values $m_{i,j}$ (for $i,j \in \{0,1,2,3\}$) are bits. We split the LUT L into 4 smaller LUTs $(L_{0,0}, L_{0,1}, L_{1,0}, L_{1,1})$ to be evaluated in the CMux tree. The output LUT of this tree is given in input to the operation selecting the right output of a LUT (corresponding to the blind rotation). The output $L[m]$ is the element in the LUT L corresponding to the input message m. The Bit Extract blocks correspond to the line 2 in Algorithm 38 and the CMux tree followed by a blind rotation corresponds to the vertical packing (VerticalPacking, line 6).

We provide the details of the technique (using vertical packing in this case) in Algorithm 38. To evaluate several LUT, we just need to repeat the vertical packing for each LUT evaluation.

Lemma 6.1 (Correctness of Algorithm 38)**.** *Let κ ciphertexts $\mathsf{ct}_i \in \mathrm{LWE}_{\boldsymbol{s}}(m_i)$ with $m_i \in \mathbb{Z}_{p_i}$ with $\log_2(p_i) = \delta_i$, such that $m_i = \sum_{j=0}^{\delta_i - 1} m_{i,j} 2^j$, for $i \in [0, \kappa - 1]$ and let $m = \sum_{i=0}^{\kappa-1} m_i 2^{\sum_{j=0}^{i-1} \delta_j}$ be the message formed by th concatenation of each messages. Let PUB be the public material, i.e., the BSK, the KSK and the PKSK as detailed in Remark 2.15. Let ν be the number of functions evaluated. Let L_j the set of lookup table representing the function f_j for $j \in [0, \nu - 1]$, each composed of $\log_2\left(\left\lceil \frac{\Omega}{N} \right\rceil\right)$ lookup tables such that $L_j = \{\mathsf{LUT}_{j,0}, \ldots, \mathsf{LUT}_{j,\log_2\left(\left\lceil \frac{\Omega}{N} \right\rceil\right)-1}\}$, such that $\mathsf{LUT}_j \in \mathfrak{R}_{q,N}^{k+1}$ with $\Omega = \max\left(N, \prod_{i=0}^{\kappa} 2^{\delta_i}\right)$. Then Algorithm 38 takes as input $\{\mathsf{ct}_i\}_{i \in [0,\kappa-1]}$, PUB and $\{L_j\}_{j \in [0,\nu-1]}$ and outputs $\{\mathsf{ct}_j \in \mathrm{LWE}_{\boldsymbol{s}}(f_j(m))\}_{j \in [0,\nu-1]}$.*

Proof *(Lemma 6.1). In what follows, we prove that the output of Algorithm 38 is: $(\mathrm{LWE}_s(f_0(m)), \ldots, \mathrm{LWE}_s(f_{\nu-1}(m)))$, with $m = \sum_{i=0}^{\kappa-1} m_i 2^{\sum_{j=0}^{i-1} \delta_j}$ the concatenation of each messages. First, Algorithm 15 extracts the occupied bit of each ciphertext. For each extracted bit, Algorithm 12 transforms the corresponding* LWE *ciphertext into a* GGSW *ciphertext encrypting a single bit. Then, taking as input all the* GGSW *ciphertexts, the vertical packing outputs, for each lookup table, a ciphertext encrypting the index of the lookup table corresponding to the binary decomposition of the concatenated bits from all messages $m_i = \sum_{j=0}^{\delta_i - 1} m_{i,j} 2^j$, where $i \in [0, \kappa - 1]$. We refer to Section 3 for the correctness of the bit extraction algorithm (Algorithm 15), the circuit bootstrapping algorithm (Algorithm 12), and the vertical packing algorithm (Algorithm 14).* □

Algorithm 38: $\{\mathsf{ct}_{\mathsf{out}_i}\}_{i=0}^{\nu-1} \leftarrow \mathsf{WoP\text{-}PBS}((\mathsf{ct}_0,\ldots,\mathsf{ct}_{\kappa-1}),\mathsf{PUB},\{L_0,\ldots,L_{\nu-1}\})$

Context:
Δ_i : scaling factor for the ciphertext ct_i
δ_i : bits occupied by message in ciphertext ct_i starting from Δ_i such that $m_i = \sum_{j=0}^{\delta_i-1} m_{i,j}2^j$
We note $m = \sum_{i=0}^{\kappa-1} m_i 2^{\sum_{j=0}^{i-1}\delta_j}$
$\Omega = \max\left(N, 2^{\sum_{i=0}^{\kappa-1}\delta_i}\right)$
$(\mathcal{B}_{\mathbf{CBS}}, \ell_{\mathbf{CBS}})$: the base and level of the output GGSW ciphertexts to the circuit bootstrapping
$L_j = \left\{\mathsf{LUT}_{j,0},\ldots,\mathsf{LUT}_{j,\log_2\left(\left\lceil\frac{\Omega}{N}\right\rceil\right)-1}\right\}_{j\in[0,\kappa-1]}$

Input:
$(\mathsf{ct}_0,\ldots,\mathsf{ct}_{\kappa-1})$ with for $0 \leq i < \kappa$, $\mathsf{Decode}\left(\mathsf{Decrypt}\left(\mathsf{ct}_i\right)\right) = m_i$
PUB : public keys required for the whole algorithm ; `/* Remark 2.15 */`
ν Lookup Tables : $L = \{L_0,\ldots,L_{\nu-1}\}$,

Output: $\{\mathsf{ct}_{\mathsf{out}_i} \in \mathrm{LWE}_{\boldsymbol{s}}\left(l_{i,m}\right)\}_{i=0}^{\nu-1}$

1 **for** $i \in [0;\kappa-1]$ **do**
  `/* Extract all bits from the LSB of the message to the non empty MSB */`
2 $[\mathsf{ct}'_{i,0},\ldots,\mathsf{ct}'_{i,\delta_i-1}] \leftarrow \mathsf{BitExtract}(\mathsf{ct}_i)$; `/* Algorithm 15 */`
3 **for** $j \in [0;\delta_i-1]$ **do**
    `/* Circuit bootstrap [CGGI20] the extracted bit into a GGSW */`
4 $\overline{\overline{C}}_{i,j} \leftarrow \mathsf{CircuitBootstrap}(\mathsf{ct}'_{i,j},\mathsf{PUB})$; `/* Algorithm 12 */`
5 **for** $k \in [0;\nu-1]$ **do**
  `/* Vertical Packing LUT evaluation [CGGI20] */`
6 $\mathsf{ct}_{\mathsf{out}_k} \leftarrow \mathsf{VerticalPacking}\left(\left\{\overline{\overline{C}}_{i,j}\right\}_{i\in[0;\kappa-1]}^{j\in\lceil 0;\delta_i-1\rceil}, L_k\right)$; `/* Algorithm 14 */`
7 **return** $\{\mathsf{ct}_{\mathsf{out}_i}\}_{i=0}^{\nu-1}$

Lemma 6.2. *The noise of the output of Algorithm 38 is:*

$$\mathsf{Var}(E_{\mathbf{WoP\text{-}PBS}}) = \mathsf{Var}(E_{\mathbf{CBS}}) + \underbrace{\left(\sum_{i=0}^{\kappa-1}\delta_i\right)\ell_{\mathbf{CBS}}(k+1)N\frac{\mathcal{B}_{\mathbf{CBS}}^2+2}{12}\mathsf{Var}(E_{\mathbf{CBS}}) + \left(\sum_{i=0}^{\kappa-1}\delta_i\right)\frac{kN}{32} + }_{mixed\ packing}$$
$$\underbrace{+\left(\sum_{i=0}^{\kappa-1}\delta_i\right)\frac{q^2-\mathcal{B}_{\mathbf{CBS}}^{2\ell_{\mathbf{CBS}}}}{24\mathcal{B}_{\mathbf{CBS}}^{2\ell_{\mathbf{CBS}}}}\left(1+\frac{kN}{2}\right) + \frac{\left(\sum_{i=0}^{\kappa-1}\delta_i\right)}{16}\left(1-\frac{kN}{2}\right)^2}_{mixed\ packing}.$$

With

$$\mathsf{Var}(E_{\mathbf{CBS}}) = \underbrace{n\ell_{\mathbf{BR}}(k+1)N\frac{\mathcal{B}_{\mathbf{BR}}^2+2}{12}\mathsf{Var}(\mathsf{BSK}) + n\frac{q^2-\mathcal{B}_{\mathbf{BR}}^{2\ell_{\mathbf{BR}}}}{24\mathcal{B}_{\mathbf{BR}}^{2\ell_{\mathbf{BR}}}}\left(1+\frac{kN}{2}\right)+}_{\mathbf{PBS}}$$
$$\underbrace{+\frac{nkN}{32}+\frac{n}{16}\left(1-\frac{kN}{2}\right)^2}_{\mathbf{PBS}} + \underbrace{\ell_{\mathbf{BR}}(n+1)\frac{\mathcal{B}_{\mathbf{BR}}^2+2}{12}\mathsf{Var}(\mathsf{KSK})+}_{\textit{private functional }\mathbf{KS}}$$
$$\underbrace{+\frac{q^2-\mathcal{B}_{\mathbf{BR}}^{2\ell_{\mathbf{BR}}}}{24\mathcal{B}_{\mathbf{BR}}^{2\ell_{\mathbf{BR}}}}\left(1+\frac{n}{2}\right)+\frac{n}{32}+\frac{1}{16}\left(1-\frac{n}{2}\right)^2}_{\textit{private functional }\mathbf{KS}}.$$

Proof *(Lemma 6.1). The noise of the output of Algorithm 38 corresponds to the noise of a circuit bootstrapping (a* **PBS** *(proof of Theorem 2.15), followed by a private functional* **KS** *(i.e., an external product (proof of Theorem 2.11))) followed by* $\sum_{i=0}^{\kappa-1}\delta_i$ *CMuxes (all the keys are uniformly binary). The formula can be obtained from the noise formulae presented in Chapter 2 and Chapter 3. More detailed of the noise formulae can be found in [CLOT21].* □

Remark 6.2. The cost of Algorithm 38 can be approximated by:

$$\mathsf{Cost}_{\mathbf{WoP\text{-}PBS}}^{\ell_{\mathbf{PBS}},\ell_{\mathbf{CBS}},k,N,n,q,\nu,\Omega,\delta} = \kappa\cdot\delta\cdot\mathsf{Cost}_{\mathbf{CBS}}^{\ell_{\mathbf{PBS}},\ell_{\mathbf{CBS}},k,N,n,q} + \nu\cdot\mathsf{Cost}_{\mathsf{VerticalPacking}}^{\Omega,\ell_{\mathbf{PBS}},k,N} + \kappa\cdot\mathsf{Cost}_{\mathsf{BitExtract}}^{\ell_{\mathbf{PBS}},k,N,n,q,\delta}.$$

Where each ciphertext is composed of messages of δ bits.

Observe that the base and level used in the **PBS** for bit extraction and in the **PBS** for circuit bootstrapping might be chosen differently. Several optimizations are possible in Algorithm 38. We did not include them directly in Algorithm 38 to simplify the explanation:

- The **PBS**s in the first step of the algorithm can either be computed independently, or sequentially, from LSB to MSB, by removing an extracted bit from the input ciphertext before extracting the next one.
- The second step of the circuit bootstrapping, which is a series of several packing functional key switchings, can be improved by following a similar footstep as a technique proposed in [CCR19]. We perform an initial LWE-to-GLWE **KS** (not functional) to each of the outputs of the **PBS**, and then, as already done in [CCR19], we perform an external product times the GGSW encryption of the GLWE secret key to obtain the remaining GLWE ciphertexts. This allows us to reduce the size of public evaluation keys at the cost of a slightly larger noise in the output.
- The **KS**-**PBS** performed in the BitExtract algorithm is a Generalized **PBS**, as described in [CLOT21], so the modulus switching directly reads the next bit to be extracted. The sign function is evaluated in order to re-scale the bit at the right scaling factor. The circuit bootstrappings used in Line 4 are also instantiated with a Generalized **PBS**. If we choose parameters that allow to have more padding bits, we could improve the circuit bootstrappings with a PBSmanyLUT, as described in [CLOT21], i.e., perform all the **PBS** in a circuit bootstrapping at the cost of a single **PBS**. Using this technique imposes an additional constraint on the noise in input of the circuit bootstrapping.
- We can observe that one of the **PBS** of the circuit bootstrappings used in Line 4 could be avoided by slightly modifying the Bit Extract algorithm (Algorithm 15) to provide the extracted bit at the right re-scaling factor.

Remark 6.3. In general, the number of circuit bootstrappings performed in Algorithm 38 corresponds to the number of bits of the input message. However, this number might be slightly larger in some special cases, such as the case where the carry buffers have not been emptied beforehand, or the case of non power of two encoding. In these cases, we might need to extract more bits of information, and so perform more **PBS**s during bit extraction and more circuit bootstrappings.

Furthermore, different possible inputs might encode the same value, hence the LUT L needs to contain some kind of redundancy. If the goal is to compute the discrete function f, one needs to compute the L as $L[(m_0, \ldots, m_{\kappa-1})] = \mathsf{Encode}\,(f\,(\mathsf{Decode}\,(m_0, \ldots, m_{\kappa-1})))$.

Remark 6.4 (Faster Algorithm 38 for Special LUTs)**.** Observe that the new **WoP-PBS** approach can be also adapted, and be very convenient, for particular LUTs such as the ReLU or the sign function in the radix mode, as instance. Indeed, for these functions we are only interested in the MSB part of the message, so the mixed packing is greatly simplified, and the cost of the **WoP-PBS** becomes linear in the number of blocks.

6.4 Comparison Between $\mathcal{A}^{(\text{this work})}$, $\mathcal{A}^{(\text{CJP21})}$ and $\mathcal{A}^{(\text{GBA21})}$

In Section 6.3, we introduced a new **WoP-PBS** in Algorithm 38. We can now resume our comparison, started in Section 6.2, to find out which algorithm is the best (depending on some parameters) to compute over ciphertexts with large precision. To do so, we consider a new atomic pattern type $\mathcal{A}^{(\text{this work})}$ composed of a dot product (**DP**, Theorem 2.7) and the **WoP-PBS** (Algorithm 38). As this algorithm can work on a single ciphertext or on several ciphertexts containing chunks of the message, we present three variants: 1, 2 and 4 blocks. We display a comparison between $\mathcal{A}^{(\text{CJP21})}$, $\mathcal{A}^{(\text{GBA21})}$ and $\mathcal{A}^{(\text{this work})}$ on figure 6.3. We used the exact same context as in Figure 6.1 for this experiment, so the failure probability is for the three of them $p_{\mathsf{fail}} \approx 2^{-35}$.

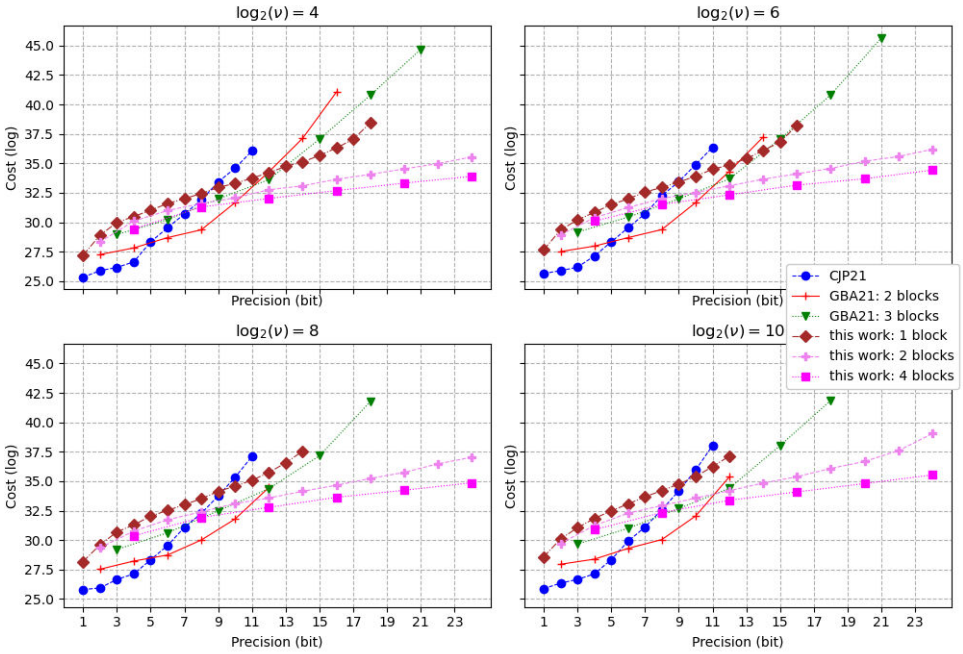

Figure 6.3: In this figure, we evaluate a LUT over a few encrypted inputs. We compare $\mathcal{A}$ type $\mathcal{A}^{(\text{this work})}$, corresponding to the **WoP-PBS** introduced in this chapter (1, 2 and 4 blocks), and $\mathcal{A}$ type $\mathcal{A}^{(\text{GBA21})}$, corresponding to the Tree-PBS [GBA21] (2 and 3 blocks). As a baseline, $\mathcal{A}$ of type $\mathcal{A}^{(\text{CJP21})}$ is also plotted.

Remark 6.5 (Noise Bound)**.** For $\mathcal{A}^{(\text{CJP21})}$ and $\mathcal{A}^{(\text{GBA21})}$, please refer to Remark 6.1. For $\mathcal{A}^{(\text{this work})}$, we have a certain number of sequential bit extractions per input LWE ciphertext / block. In theory, we want to take into account all those potential PBS (one per bit extraction), but we noticed that the first one dominates all the others regarding noise. In fact, their impact

on the total failure probability is negligible compared with the first bit extraction. Our experiments showed that for 2-norms $\nu \geq 4$, and for failure probability below 2^{-25} this assumption holds. We leave as future works the exploration of this topic. With this assumption, we start by computing the failure probability needed for one PBS defined as $p'_i = 1 - (1 - p_{\mathsf{fail}})^{\frac{1}{\kappa}}$, since there are κ input LWE ciphertexts. From it, we can finally compute the noise bound for each PBS $t\left(p, 0\right) = \frac{q}{2^1 \cdot p \cdot z^*\left(p'_i\right)}$.

The brown/⧫ curve represents the cost of the best parameter set for an atomic pattern $\mathscr{A}^{(\text{this work})}$ working over one block. We can immediately notice that, between 1 and 9 bits of precision, $\mathscr{A}^{(\text{CJP21})}$ is more interesting than the new bootstrapping (Algorithm 38). However, with precisions from 10 bits and above, $\mathscr{A}^{(\text{this work})}$ has solutions that are more efficient than the $\mathscr{A}^{(\text{CJP21})}$'s existing ones, and finds solutions when $\mathscr{A}^{(\text{CJP21})}$ cannot. For small ν, it offers solutions that are slightly better than the ones from $\mathscr{A}^{(\text{GBA21})}$.

The pink/✚ curve (respectively the pink/■ curve) represents the atomic pattern $\mathscr{A}^{(\text{this work})}$ for two blocks (respectively four blocks) of message. On those curves, we see that it scales much better than the other atomic pattern types. With Algorithm 38, we manage to find solution up to 24 bits of precision. Those solutions are costly but far less than the ones for $\mathscr{A}^{(\text{GBA21})}$, and for comparison, it is only 2^{10} times more costly to compute a LUT over a message with 24-bits of precision with $\mathscr{A}^{(\text{this work})}$ than compute a LUT with 1-bit of precision with $\mathscr{A}^{(\text{CJP21})}$. Also for 18 bits of precision, the new WoP-PBS with two blocks is approximately 2^7 times faster than the tree-PBS in $\mathscr{A}^{(\text{GBA21})}$ with three blocks.

To sum up, for small precisions (up to 5 bits), TFHE **PBS** is the best option among the three considered. Above 10/11 bits of precision, the algorithm we introduced in this chapter (Algorithm 38) becomes the best alternative and improves the state-of-the-art by a non-negligible factor.

Remark 6.6 (LUT Evaluation for Even More Precision)**.** It is important to observe that evaluating a LUT on integers larger than e.g., 30 bits, even in clear, becomes too expensive in terms of memory (e.g., a LUT for 30-bit input and output integers contains $2^{30} \cdot 64$ bits $= 8$ GB of information). So both techniques, the Tree-PBS and our new **WoP-PBS**, are anyway not practical anymore.

Remark 6.7 (Small Public Material for Algorithm 38)**.** An important observation to make about Algorithm 38 is that the size of the needed public material scales way better than a tree-PBS as in [GBA21]. As an example, for a total of 18 bits of precision we have a key of 1.65 GB for $\mathscr{A}^{(\text{GBA21})}$ and a size of 0.926 GB for $\mathscr{A}^{(\text{this work})}$.

6.5 Comparison Between $\mathscr{A}^{(\textbf{this work})}$ and $\mathscr{A}^{(\textbf{LMP21})}$

A few **WoP-PBS** constructions have been proposed in the literature. Some works [KS21, LMP21] already compare them, but our optimization framework enables us to truly do it by comparing them at the best of their efficiency. This can be done by putting each of them in a different atomic pattern type and finding optimal parameters for different 2-norms and precisions. To do so, we create one additional atomic pattern type called $\mathscr{A}^{(\text{LMP21})}$ composed of a **DP**, a **KS** and the **WoP-PBS** from [LMP21]. We used the exact same context as in Figure 6.1 for this experiment, so the failure probability is for the both of them $p_{\mathsf{fail}} \approx 2^{-35}$. We display in figure 6.4 the comparison between our new **WoP-PBS** (Algorithm 38, blue/• curve) in $\mathscr{A}^{(\text{this work})}$ and the **WoP-PBS** from [LMP21] in $\mathscr{A}^{(\text{LMP21})}$ (red/+ curve).

Remark 6.8 (Noise Bound)**.** For $\mathscr{A}^{(\text{this work})}$, please refer to Remark 6.5. For $\mathscr{A}^{(\text{LMP21})}$, we consider the two sequential PBS involved in the algorithms. They almost have the same amount of input noise and thus we assume that they both contribute equally to the overall failure probability. We experimented the two possible scenarios, (i) taking the first PBS's input noise for the computation or (ii) taking the second one. We did not observe any difference between the two approaches for the considered failure probabilities and 2-norms. We start by computing the failure probability

needed for one PBS defined as $p'_i = 1 - (1 - p_{\mathsf{fail}})^{\frac{1}{2}}$ and from it we can finally compute the noise bound for each PBS $t\,(p, 0) = \frac{q}{2^{1+1} \cdot p \cdot z^*(p'_i)}$.

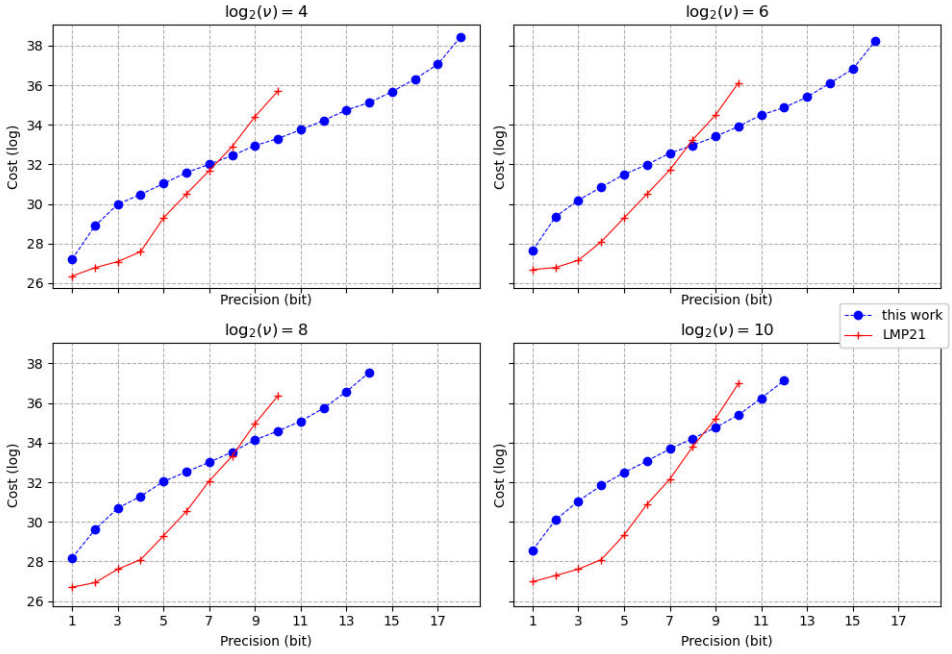

Figure 6.4: In this figure, we compare the cost of $\mathscr{A}$ type $\mathscr{A}^{(\text{this work})}$ and type $\mathscr{A}^{(\text{LMP21})}$. The first one corresponds to **DP-KS** followed by our new **WoP-PBS** (Section 6.3), and the second one to **DP-KS** followed by the **WoP-PBS** from [LMP21].

The first thing that we learn on the **WoP-PBS** of [LMP21] is that it does not scale well with big precisions, which is not surprising as the algorithm uses as subroutine two **PBS** from [CGGI20] to compute the **WoP-PBS**. Thus, as for $\mathscr{A}^{(\text{CJP21})}$, for precisions above 10, we do not find any feasible solutions. We can also identify as before two parts on the curve, the first one for small precisions (1 to 8 bits) and a second one for higher precisions: the reason behind this sudden growth in cost is also due to the increase of the polynomial size to manage bigger messages.

Thanks to the new **WoP-PBS** (Algorithm 38), we are able to compute a **WoP-PBS** over large messages. To conclude, this new algorithm scales better than existing algorithms to compute LUTs over large message and we do not need a padding bit which is a known constraint of TFHE bootstrapping.

Part III

Representation for Homomorphic Integer and Floating-Point

This part of the thesis explores new constructions for representing larger numerical values in TFHE, overcoming the limitation of small message spaces (less than 10 bits). Expanding the range of representable values is essential for building new applications with real-world use cases.

To address these challenges, we first introduce efficient encodings and algorithms for representing large integers within the constraints of TFHE. Then, building on these new constructions, we develop more complex structures to efficiently represent floating-point numbers and design dedicated arithmetics. By addressing these challenges, we extend the practical usability of TFHE beyond simple Boolean and small integer computations.

Chapter 7

Homomorphic Large Integers

In Chapter 2, we discussed how TFHE was initially designed as a Boolean scheme and later extended to support small integers (smaller than 8 bits). However, modern architectures rely on 32- or 64-bit integers, and most software is built around these native precisions. As seen in the introduction, an objective of FHE is to replicate the clear world, allowing any cleartext operation to be executed homomorphically. Alas, TFHE is limited to small integer precision, which restricts its use cases as an FHE scheme (Limitation 4). Indeed, many use cases require higher precision and cannot be directly executed with TFHE. Solutions must be found either to work with smaller representations, such as in machine learning, where circuits can be quantized to smaller integers (at the cost of precision), or to represent high precision using specific encodings for TFHE, as will be presented in this chapter.

Even with the advancements detailed in Part II, particularly with the methods proposed in Chapter 4 and Chapter 6, achieving high precision using a single ciphertext remains a significant challenge. In this chapter, we take on this challenge and present different encodings and constructions based on several ciphertexts that enable us to efficiently reach higher precision.

7.1 Introduction

In this chapter, we propose to study how to represent large integers by using several LWE ciphertext. In the state-of-the-art, several approaches using many ciphertexts to represent a single message are proposed. We can summarize these approaches in two main categories: the radix and the CRT (Chinese Reminder Theorem) representations.

The radix representation consists in decomposing a message into several chunks according to a decomposition base. It is very similar to the representation in base 10 we use in our daily lives, where to represent a large number we use several digits. Then the idea is to put each of the elements of the decomposition into a separate ciphertext and to define the new encryption of the large message as the list of these ciphertexts.

The CRT approach consists in representing a number x modulo a large integer $\Omega = \prod_{i=0}^{\kappa} \omega_i$, where the ω_i are all co-primes, as the list of its residues $x_i = x \mod \omega_i$. Each of the reduced elements is then encrypted into a different ciphertext and, as for the radix approach, the new encryption of the large message modulo Ω is the list of these ciphertexts. Observe that the CRT approach in the plaintext space is different from the well known SIMD style [GHS12].

In order to use these two approaches in TFHE, the elements of the decomposition (for the radix approach) and the residues (for the CRT approach) need to be quite small (generally less than 8 bits).

The approach of splitting a message into multiple ciphertexts has already been proposed for binary radix decomposition in FHEW [DM15] and TFHE [CGGI20], and for other representations in [BST20], [GBA21], [KO22], [CZB+22], [LMP21] and [CLOT21]. However, none of them takes

advantage of carry buffers, introduced in Definition 8, to make the computations more efficient between multi-ciphertext encrypted integers by avoiding bootstrapping as much as possible.

The idea of using the CRT approach is mentioned in [KS21] but authors do not change the traditional TFHE encoding to fit the CRT representation. In the following sections, we provide detailed algorithms to describe the use of the CRT in the plaintext space with two different approaches (with or without carry buffers, along with their respective encoding). Both of the radix and the CRT approaches are detailed in Section 7.2.

These techniques are the first step towards larger precision. Indeed, they have some limitations. In particular, the radix approach does not allow any modulo to be represented, but only multiples of a certain base (or bases), and the CRT is limited on the maximal number that could be represented, because there exists a very limited amount of primes or co-primes smaller than 8 bits. In Section 7.3, we demonstrate how to overcome these limitations.

7.2 Modular Arithmetic with Several LWE ciphertexts

In TFHE, a single ciphertext can efficiently encrypt up to 8-bits of information. Larger messages should be encrypted in a different way: a possibility is to use many ciphertexts to encrypt a single large precision message in LWE. In that case, there are two options that are already used in the literature: the *radix representation* or the CRT *representation*. They are both valid approaches but have some limitations in their actual state.

We begin by introducing radix-based large integer using the carry/message encoding detailed in Subsection 2.4.3. Next, we demonstrate how to perform basic operations, such as addition and multiplication, using this representation. Lastly, we present how to achieve high precision using the CRT representation with TFHE.

7.2.1 Radix-Based Large Integers

The radix based approach consists in encrypting a large integer modulo $\Omega = \prod_{i=0}^{\kappa-1} \beta_i$ as a list of $\kappa \in \mathbb{N}$ LWE ciphertexts. Each of the κ ciphertexts is defined according to a pair $(\beta_i, p_i) \in \mathbb{N}^2$ of parameters, such that $2 \leq \beta_i \leq p_i < q$, which respectively corresponds to the message subspace and the carry-message subspace involved with the modular arithmetic, as described in Definition 8 Figure 7.1 gives a visual representation out of a toy example.

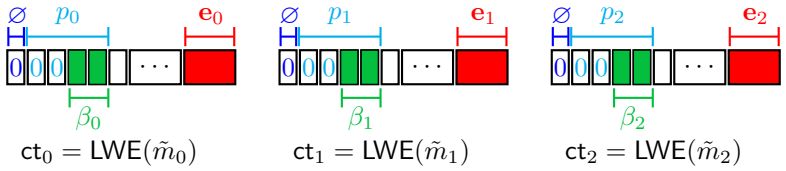

Figure 7.1: Plaintext representation of a fresh radix-based modular integer of length $\kappa = 3$ working modulo $\Omega = (2^2)^3$ with $\mathsf{msg} = m_0 + m_1 \cdot \beta_0 + m_2 \cdot \beta_0 \cdot \beta_1$. The symbol $\varnothing$ represents the padding bit needed for the **PBS**. For each block we have $\widetilde{m_i} = \mathsf{Encode}(m_i, p_i, q)$. For all $0 \leq i < \kappa$ we have $\beta_i = 4$, $p_i = 16$, $\kappa = 3$ and $\Omega = 4^3$.

In practice, the restriction for Ω is that it has to be a product of small basis. Indeed, TFHE-like schemes do no scale well when one is increasing the precision, so the good practice is to keep $p_i \leq 2^8$.

To *encode* a message $\mathsf{msg} \in \mathbb{Z}_\Omega$, one needs to decompose it into a list of $\{m_i\}_{i=0}^{\kappa-1}$ such that $\mathsf{msg} = m_0 + \sum_{i=1}^{\kappa-1} m_i \cdot \left(\prod_{j=0}^{i-1} \beta_j\right)$. Then we can independently call the Encode function (definition 8) on each m_i so we have $\widetilde{m}_i = \mathsf{Encode}(m_i, 2^\pi \cdot p_i, q)$ with π the number of bits of padding. Finally we can encrypt each $\widetilde{m}_i$ into an LWE ciphertext such that we have

$\mathbf{ct} = [\mathsf{ct}_\kappa, \ldots, \mathsf{ct}_0] \in [\mathrm{LWE}(\widetilde{m}_{\kappa-1}), \ldots, \mathrm{LWE}(\widetilde{m}_0)]$. To *decode*, we simply recompose the integer from the m_i values.

In terms of *operations* between radix-based large integers, it is important to recall that two messages can interact if they are encoded and encrypted with the same parameters. The majority of the arithmetic operations can be computed by using a schoolbook approach (homomorphically mixing linear operations and **PBS**) and by keeping an eye on the degree of fullness in each block (Definition 18). When carries are full, they need to be propagated to next block: this is done by extracting the carry and the message and adding carry to the next block (last carry can be thrown away).

7.2.1.1 Radix-Based Large Integer Operations

In this section, we present the main algorithms for efficiently working with large integers (we note that several additional algorithms, including more optimized variants, are presented in [BdBB+25]). For efficiency reasons, we focus on the radix representation, where the carry and the message spaces represent the same amount of information. For the κ ciphertexts composing a large integer, we have $p_i = \beta_i^2 = \sqrt[\kappa]{\Omega}$ for $i \in [0, \kappa - 1]$. This representation will be intensively used in the next chapter (Chapter 8).

Carry Propagation During arithmetic operations, the carry buffer in each block accumulates values, and at some point, it becomes necessary to free the carry space through carry propagation. This is required either when the carry spaces reach their capacity or when the next algorithm needs to operate with a fresh carry space.

The carry propagate is a sequential algorithm that starts from the least significant block extract both the carry and the message using two separate **PBS**. The extracted carry is then added to the next blocks, and this process continue until the most significant block, where we only need to extract the message. Indeed the carry value in the most significant block represents value higher than the modulo Ω. This procedure is detailed in Algorithm 39.

Algorithm 39: $\mathbf{ct}_{\mathsf{out}} \leftarrow \mathsf{CarryPropagate}\ (\mathbf{ct}, \mathsf{PUB})$

Context:
- $\mathsf{LUT}_{\mathsf{Carry}}$: LUT to return the carry of the ciphertext (return $\left\lfloor \frac{m}{\beta} \right\rfloor$).
- $\mathsf{LUT}_{\mathsf{Msg}}$: LUT to return the message of the ciphertext (return $m \mod \beta$).
- p : the carry-message modulus
- β : the message modulus

Input:
- $\mathbf{ct} = [\mathsf{ct}_{\kappa-1}, \ldots, \mathsf{ct}_0] \in \mathbb{Z}_q^{(n+1)\cdot\kappa}$
- PUB : Public materials for **KS** and **PBS**; `/* Remark 2.15 */`

Output:
- $\mathbf{ct_{out}} = [\mathsf{ct}_{\mathsf{out},\kappa-1}, \ldots, \mathsf{ct}_{\mathsf{out},0}] \in \mathbb{Z}_q^{(n+1)\cdot\kappa}$

```
1 for i ∈ [0; κ − 1] do
2     if i ≠ κ − 1 then
          /* Extract the carry                                          */
3         ct_carry ← KS-PBS(ct_i, PUB, LUT_Carry) ;   /* Algorithm 4 and 11  */
4         ct_{i+1} ← ct_{i+1} + ct_carry
      /* Extract the message                                            */
5     ct_{out,i} ← KS-PBS(ct_i, PUB, LUT_Msg) ;       /* Algorithm 4 and 11  */
6 return (ct_out = [ct_{out,κ−1}, ··· ct_{out,0}])
```

Theorem 7.1 (Correctness of Algorithm 39)**.** *Let* $\mathbf{ct} = [\mathsf{ct}_{\kappa-1}, \ldots, \mathsf{ct}_0]$ *be ciphertexts encrypting* $\mathsf{msg} = \sum_{i=0}^{\kappa-1} m_i \cdot \beta^i$ *such that* ct_i *encrypts* m_i *with* $m_i < p - \beta$*. Then Algorithm 39 returns* $\mathbf{ct_{out}} = [\mathsf{ct}_{\mathsf{out},\kappa-1}, \ldots, \mathsf{ct}_{\mathsf{out},0}]$ *encrypting* $\mathsf{msg} \mod \Omega$ *such that* $\mathsf{ct}_{\mathsf{out},i}$ *encrypting* $m_{\mathsf{out},i}$ *with* $m_i < \beta$*.*

The cost of Algorithm 39 is: $\mathsf{Cost}^{\ell,k,N,n,q,\kappa}_{\mathsf{CarryPropagate}} = (2\kappa-1)\left(\mathsf{Cost}^{\ell,k,N,n,q}_{\mathbf{PBS}} + \mathsf{Cost}^{\ell,n,k,N}_{\mathbf{KS}}\right)$.

Proof *(Theorem 7.1). In this Algorithm, each block* ct_i *encrypts* $m_i = c_i \cdot \beta + \widetilde{m_i} < p - \beta$. *The message extraction returns* $\widetilde{m_i} < \beta$. *The carry extraction returns the carry* $c_i < \beta$ *from block* i *and adds this carry to the following block* $i+1$. *As* $m_{i+1} = c_{i+1} \cdot \beta + \widetilde{m_{i+1}} < p - \beta$, *we have* $m_{i+1} = c_{i+1} \cdot \beta + \widetilde{m_{i+1}} + c_i < p$ *so we can perform the addition without compromising the padding bits and without losing any information. After Algorithm 39, each ciphertext of* $\mathbf{ct}$ *encrypts* $\widetilde{m}_i < \beta$ *such that* $\sum_{i=0}^{\kappa-1} m_i \cdot \beta^i \mod \Omega = \sum_{i=0}^{\kappa-1} \widetilde{m}_i$.

At each step, one needs to compute two **PBS***: one to extract the message and another one to extract the carry, except for the most significant block, where only one* **PBS** *is required to extract the message. We note that more efficient algorithms have been studied in [BdBB+25].* □

Radix Addition. The addition between two large integers is presented in Algorithm 40.

Algorithm 40: $\mathbf{ct}_{\mathsf{out}} \leftarrow \mathsf{RadixAdd}\ (\mathbf{ct}_0, \mathbf{ct}_1, \mathsf{PUB})$

Context:
- β : the message modulus
- $p = \beta^2$: the carry-message modulus

Input:
- $\mathbf{ct}_0 = [\mathsf{ct}_{0,\kappa-1}, \ldots, \mathsf{ct}_{0,0}] \in \mathbb{Z}_q^{(n+1)\cdot\kappa}$
- $\mathbf{ct}_1 = [\mathsf{ct}_{1,\kappa-1}, \ldots, \mathsf{ct}_{1,0}] \in \mathbb{Z}_q^{(n+1)\cdot\kappa}$
- PUB : Public materials for **KS** and **PBS**; `/* Remark 2.15 */`

Output:
- $\mathbf{ct}_{\mathsf{out}} = [\mathsf{ct}_{\mathsf{out},\kappa-1}, \ldots, \mathsf{ct}_{\mathsf{out},0}] \in \mathbb{Z}_q^{(n+1)\cdot\kappa}$

```
for i ∈ [0; κ − 1] do
    ct_out,i ← Add(ct_0,i, ct_1,i) ;                                  /* Algorithm 3 */
[ct_out,κ−1, ··· ct_out,0] ← CarryPropagate([ct_out,κ−1, ··· ct_out,0], PUB) ;   /* Algorithm 39 */
return (ct_out = [ct_out,κ−1, ··· ct_out,0])
```

Theorem 7.2 (Correctness of Algorithm 40)**.** *Let* $\mathbf{ct}_i = [\mathsf{ct}_{i,\kappa-1}, \ldots, \mathsf{ct}_{i,0}]$ *be ciphertexts encrypting* $\mathsf{msg_i} = \sum_{j=0}^{\kappa-1} m_{i,j} \cdot \beta^j$ *such that* $\mathsf{ct}_{i,j}$ *encrypts* $m_{i,j}$ *for* $i \in \{0,1\}$ *such that* $m_{0,j} + m_{1,j} < p$. *Then Algorithm 40 returns* $\mathbf{ct}_{\mathsf{out}} = [\mathsf{ct}_{\mathsf{out},\kappa-1}, \ldots, \mathsf{ct}_{\mathsf{out},0}]$ *encrypting* $\mathsf{msg}_0 + \mathsf{msg}_1 \mod \Omega$.

The cost of Algorithm 40 is: $\mathsf{Cost}^{\ell,k,N,n,q,\kappa}_{\mathsf{RadixAdd}} = \kappa \cdot \mathsf{Cost}^{n,1}_{\mathsf{add}} + \mathsf{Cost}^{\ell,k,N,n,q,\kappa}_{\mathsf{CarryPropagate}}$.

Proof *(Theorem 7.2). As for each message* $m_{0,j}$, $m_{1,j}$ *we have* $m_{0,j} + m_{1,j} < p - \beta$, *for each ciphertext we can do the addition of* $\mathsf{ct}_{0,j}$ *and* $\mathsf{ct}_{1,j}$ *encrypting the sum of* $m_{0,j}$ *and* $m_{1,j}$ *in the ciphertext* $\mathsf{ct}_{\mathsf{out},j}$ *(Algorithm 3). Finally, we have* $\mathbf{ct}_{\mathsf{out}}$ *which encrypts* $\sum_{j=0}^{\kappa-1} m_{0,j}\beta^j + \sum_{j=0}^{\kappa-1} m_{1,j}\beta^j = \sum_{j=0}^{\kappa-1}(m_{0,j} + m_{1,j})\beta^j \mod \Omega$.

We observe that the most computationally expensive part of the algorithm is the carry propagation. Depending on the parameters and the degree of fullness, when chaining multiple additions, it is not necessary to perform a carry propagation after each addition.

Moreover, if the carry space is nearly full, i.e., if for two messages $m_{0,j}$ *and* $m_{1,j}$ *we have* $m_{0,j} + m_{1,j} \geq p - \beta$, *we need to perform a carry propagation (Algorithm 39) before performing the addition.* □

Radix Multiplication. The multiplication of two large integers is presented in Algorithm 41. Before performing this operation, the carry buffers of both ciphertexts must be empty. If they are not, a carry propagation (Algorithm 39) must be executed before.

Theorem 7.3 (Correctness of Algorithm 41)**.** *Let* $\mathbf{ct}_i = [\mathsf{ct}_{i,\kappa-1}, \ldots, \mathsf{ct}_{i,0}]$ *be ciphertexts encrypting* $\mathsf{msg_i} = \sum_{j=0}^{\kappa-1} m_{i,j} \cdot \beta^j$ *such that* $\mathsf{ct}_{i,j}$ *encrypts* $m_{i,j}$ *for* $i \in \{0,1\}$ *with* $m_{i,j} < \beta$. *Then Algorithm 41 returns* $\mathbf{ct}_{\mathsf{out}} = [\mathsf{ct}_{\mathsf{out},\kappa-1}, \ldots, \mathsf{ct}_{\mathsf{out},0}]$ *encrypting* $\mathsf{msg}_0 \cdot \mathsf{msg}_1 \mod \Omega$.

Algorithm 41: $\mathbf{ct}_{\mathsf{out}} \leftarrow \mathsf{RadixMul}\ (\mathbf{ct}_0, \mathbf{ct}_1, \mathsf{PUB})$

Context:
- β : the message modulus
- $p = \beta^2$: the carry-message modulus
- $\mathsf{LUT}_{\mathsf{MSB}}$: LUT to return $\left(\left\lceil \frac{m}{\beta} \right\rceil \cdot (m \mod \beta)\right) \mod \beta$.
- $\mathsf{LUT}_{\mathsf{LSB}}$: LUT to return $\frac{\left(\left\lfloor \frac{m}{\beta} \right\rfloor \cdot (m \mod \beta)\right)}{\beta} \mod \beta$.

Input:
- $\mathbf{ct}_0 = [\mathsf{ct}_{0,\kappa-1}, \ldots, \mathsf{ct}_{0,0}] \in \mathbb{Z}_q^{(n+1)\cdot\kappa}$
- $\mathbf{ct}_1 = [\mathsf{ct}_{1,\kappa-1}, \ldots, \mathsf{ct}_{1,0}] \in \mathbb{Z}_q^{(n+1)\cdot\kappa}$
- PUB : Public materials for **KS** and **PBS**; `/* Remark 2.15 */`

Output:
- $\mathbf{ct}_{\mathsf{out}} = [\mathsf{ct}_{\mathsf{out},\kappa-1}, \ldots, \mathsf{ct}_{\mathsf{out},0}] \in \mathbb{Z}_q^{(n+1)\cdot\kappa}$

1 **for** $i \in [0; \kappa - 1]$ **do**
2 $\quad \mathsf{ct}_{\mathsf{tmp}} \leftarrow \mathsf{ct}_{0,i} \cdot \beta$
3 $\quad \mathbf{ct}_{\mathsf{LSB},i} \leftarrow \{\}$
4 $\quad \mathbf{ct}_{\mathsf{MSB},i} \leftarrow \{\}$
$\quad$ `/* Trivial encryption of zero, Remark 2.6 */`
5 $\quad \mathsf{tmp}_{\mathsf{LSB}} \leftarrow \{\mathsf{ct}(0)\}^{i}$
6 $\quad \mathsf{tmp}_{\mathsf{MSB}} \leftarrow \{\mathsf{ct}(0)\}^{i+1}$
7 $\quad$ **for** $j \in [\kappa - 1 - i; 0]$ **do**
8 $\quad\quad \mathsf{ct}_{\mathsf{tmp}} = \mathsf{add}(\mathsf{ct}_{\mathsf{tmp}}, \mathsf{ct}_{1,j})$; `/* Algorithm 3 */`
9 $\quad\quad \mathbf{ct}_{\mathsf{LSB},i} \leftarrow \mathbf{ct}_{\mathsf{LSB},i}.\mathsf{append}\,(\mathbf{KS\text{-}PBS}(\mathsf{ct}_{\mathsf{tmp}}, \mathsf{PUB}, \mathsf{LUT}_{\mathsf{LSB}}))$
10 $\quad\quad$ **if** $i + j \neq \kappa - 1$ **then**
11 $\quad\quad\quad \mathbf{ct}_{\mathsf{MSB},i} \leftarrow \mathbf{ct}_{\mathsf{MSB},i}.\mathsf{append}\,(\mathbf{KS\text{-}PBS}(\mathsf{ct}_{\mathsf{tmp}}, \mathsf{PUB}, \mathsf{LUT}_{\mathsf{MSB}}))$
12 $\quad \mathbf{ct}_{\mathsf{LSB},i} \leftarrow \mathbf{ct}_{\mathsf{LSB},i}.\mathsf{append}\,(\mathsf{tmp}_{\mathsf{LSB}})$
13 $\quad \mathbf{ct}_{\mathsf{MSB},i} \leftarrow \mathbf{ct}_{\mathsf{MSB},i}.\mathsf{append}\,(\mathsf{tmp}_{\mathsf{MSB}})$
14 $\mathbf{ct}_{\mathsf{out}} \leftarrow \{\mathsf{ct}(0)\}^{\kappa}$
15 **for** $i \in [0; \kappa - 1]$ **do**
16 $\quad \mathbf{ct}_{\mathsf{out}} \leftarrow \mathsf{RadixAdd}(\mathbf{ct}_{\mathsf{out}}, \mathbf{ct}_{\mathsf{LSB},i}, \mathsf{PUB})$; `/* Algorithm 40 */`
17 $\quad \mathbf{ct}_{\mathsf{out}} \leftarrow \mathsf{RadixAdd}(\mathbf{ct}_{\mathsf{out}}, \mathbf{ct}_{\mathsf{MSB},i}, \mathsf{PUB})$; `/* Algorithm 40 */`
18 **return** $(\mathbf{ct}_{\mathsf{out}} = [\mathsf{ct}_{\mathsf{out},\kappa-1}, \cdots \mathsf{ct}_{\mathsf{out},0}])$

The cost of Algorithm 41 can be bounded by: $\mathsf{Cost}^{n,\kappa,\ell,k,N,q}_{\mathsf{RadixMul}} = 2\kappa \cdot \mathsf{Cost}^{\ell,k,N,n,q,\kappa}_{\mathsf{RadixAdd}} + \kappa^2/2 \cdot \mathsf{Cost}^{n,1}_{\mathsf{add}} + \kappa(\kappa-1) \cdot \left(\mathsf{Cost}^{\ell,k,N,n,q}_{\mathbf{PBS}} + \mathsf{Cost}^{\ell,n,k,N}_{\mathbf{KS}}\right)$*. This cost estimation does not take into account several potential improvements, such as the one described in Theorem 7.2.*

Proof *(Theorem 7.3). Considering the correctness of each algorithm used in Algorithm 41, during each iteration of the loop at line 1, we compute the equivalent of the clear result* $m_{0,i} \cdot \beta^i \cdot \sum_{j=0}^{\kappa-1-i} m_{1,j}\beta^j = \sum_{j=0}^{\kappa-1-i} m_{0,i} \cdot m_{1,j}\beta^{j+i}$*. Since* $m_{0,i} < \beta$ *and* $m_{1,j} < \beta$*, we have* $m_{0,i} \cdot m_{1,j} = c_{i,j} \cdot \beta + \widetilde{m}_{i,j} < \beta^2 = p$*. Thus, for each block multiplication, we have* $m_{0,i}\beta^i \cdot m_{1,j}\beta^j = c_{i,j} \cdot \beta^{i+j+1} + \widetilde{m}_{i,j}\beta^{i+j}$*. We can rewrite* $m_{0,i} \cdot \beta^i \cdot \sum_{j=0}^{\kappa-1-i} m_{1,j}\beta^j$ *as* $\sum_{j=0}^{\kappa-1-i} \widetilde{m}_{i,j}\beta^{j+i} + \sum_{j=0}^{\kappa-2-i} c_{i,j}\beta^{j+i+1}$*. We note that* $\sum_{j=0}^{\kappa-1-i} \widetilde{m}_{i,j}\beta^{j+i}$ *is encrypted in* $\mathbf{ct}_{\mathsf{LSB},i}$ *and* $\sum_{j=0}^{\kappa-2-i} c_{i,j}\beta^{j+i+1}$ *is encrypted in* $\mathbf{ct}_{\mathsf{MSB},i}$

Then, the rest of the proof is straightforward and follows the standard schoolbook multiplication method. □

7.2.2 CRT-Based Large Integers

The CRT based approach consists in encrypting a large integer modulo $\Omega = \prod_{i=0}^{\kappa-1} \beta_i$ as a list of κ LWE ciphertexts, such that each pair β_i and $\beta_{j\neq i}$ of bases are co-primes. Each of the κ ciphertexts is defined according to a pair $\{\beta_i, p_i\}_{0\leq i<\kappa}$ such that $2 \leq \beta_i < p_i < q$.

In order to *encode* a message $\mathsf{msg} \in \mathbb{Z}_\Omega$, one needs to compute $\{m_i\}_{i=0}^{\kappa-1}$ such that $\mathsf{msg} = m_i \mod \beta_i$ for all $0 \leq i < \kappa$. Then we can independently encode and encrypt each m_i into an LWE ciphertext. To *decode*, we simply need to compute the modular reduction in base β_i and compute the inverse of the CRT.

With this CRT encoding, we have to empty the carry buffers when they are (almost) full. Indeed, the quantity overlapping the base β_i is not needed to maintain correctness but when using TFHE **PBS**, the bit of padding needs to be preserved. We need to only call the message extraction algorithm, described in Remark 2.17 when needed.

All the arithmetic operations can be performed independently on the blocks by using the operators described in Section 2.3. Concerning the evaluation of LUT, the only known way in the literature to compute them on CRT-based large integers, is the technique proposed by [KS21], that can be used only when the LUT to evaluate is CRT friendly. By CRT friendly, we intend a LUT L that can be independently evaluated in each component, i.e., L such that $\mathsf{Encode}_{\mathsf{CRT}}(L(\mathsf{msg})) = (L_0(m_0), \ldots, L_{\kappa-1}(m_{\kappa-1}))$ where $\mathsf{Encode}_{\mathsf{CRT}} = (m_0, \ldots, m_{\kappa-1})$. For generic LUT evaluations, we can used either the **WoP-PBS** (Algorithm 38) presented in Chapter 6 or the technique introduced by [GBA21] (Algorithm 26).

Native CRT. In TFHE, we can also encode CRT integers by using no padding bit and no carry buffer (so no degree of fullness either), and by encoding the message m_i as $\left\lfloor \frac{q}{\beta_i} \cdot m_i \right\rceil$. By doing so, additions and scalar multiplications become native and do not require any PBS, except for noise reduction. To compute additions one can use the LWE addition on each residue, and to compute a scalar multiplication by α, one can decompose α with the CRT basis into smaller integers, and compute scalar multiplications with them. Without the bit of padding, the **PBS** can be evaluated only with a **WoP-PBS** algorithm such as the ones presented in Section 3.3 and Algorithm 38 introduced in Chapter 6.

7.2.3 Limitations

The radix and CRT approaches discussed in this section are a first step towards solving the precision problem in TFHE-like schemes. However, they come with limitations:

- The radix approach is limited to the modulo Ω that can be expressed as a product of bases. But if the modulo is as instance a large prime, no solution is known.
- The CRT approach suffers from the CRT requirements, i.e., co-prime bases, and the precision limitation we have in practice with TFHE. Indeed, there are a limited number of primes between 2 and 128. It means that this approach is good when Ω is composed of small enough co-prime factors but for the rest of the possible Ω we need other solutions.

In the next section, we provide solutions to overcome all these limitations.

7.3 TFHE-based Large Integers

In this section, we propose two improvements. First, we generalize the radix approach to support any large modulus Ω. Then, we introduce a hybrid approach that combine the advantage of both the radix and the CRT approaches and allows us to work efficiently with any moduli. In practice, without the first improvement, the number of possible CRT residues is limited by the number of small prime integers, significantly restricting the range of available general moduli Ω offered by the hybrid approach.

7.3.1 Generalization of radix to any large modulus

By using the radix representation, homomorphic modular integers are defined modulus Ω, that is equal to the product of the bases $\beta_i \in \mathbb{N}, i \in [0, \kappa-1]$, i.e., $\Omega = \prod_{i=0}^{\kappa-1} \beta_i$. Here, we propose to remove this restriction by generalizing the previous arithmetic to any modulus Ω s.t. $\prod_{j=0}^{\kappa-2} \beta_j < \Omega < \prod_{j=0}^{\kappa-1} \beta_j$. The only difference with the previous approaches lies in the computation of the modular reduction. In what follows, we propose two complementary methods to perform this modular reduction, whose efficiency depends on Ω and the product of the selected basis.

First method for modular reduction. The first method consists in performing multiple LUT evaluations in the most significant block to reduce it modulo Ω. Indeed, the modular reduction is applied on the κ^{th} block (i.e., $\mathsf{ct}_{\kappa-1}$) which represents $m_{\kappa-1} \cdot \prod_{i=0}^{\kappa-2} \beta_i$ with $m_{\kappa-1} < p_{\kappa-1}$, and which might be larger than Ω. The complete process is detailed in Algorithm 42. The modular reduction is performed as a series of κ **PBS** (with **KS**, Line 2) and the result is a radix-based integer with a base $(\beta_0, \ldots, \beta_{\kappa-1})$ decomposition. The final step is to add the first $\kappa-1$ blocks of the result of the modular reduction to the first $\kappa-1$ blocks of the input (Line 4) and to replace the last block in the result by the $(\kappa-1)$-th block obtained in the modular reduction (Line 5).

Algorithm 42: $(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1}) \leftarrow \mathsf{ModReduction}_1((\mathsf{ct}_0, \ldots, \mathsf{ct}_{\kappa-1}), \mathsf{PUB})$

Context:
$$\begin{cases} L_j: \text{ r-redundant LUT for } \begin{cases} \mathbb{Z}_{p_{\kappa-1}} \to \mathbb{Z}_{\mathcal{B}_j} \\ x \mapsto x'_j = \mathsf{Decomp}_j\left(x \cdot \prod_{h=0}^{\kappa-2} \mathcal{B}_h \mod \Omega\right) \end{cases} \\ x'_j \text{ is the j-th element in the decomposition in base } (\mathcal{B}_0, \ldots, \mathcal{B}_{\kappa-1}) \\ \text{s.t. } x \cdot \prod_{h=0}^{\kappa-2} \mathcal{B}_h \mod \Omega = x'_0 + \sum_{i=1}^{\kappa-1} x'_i \cdot \left(\prod_{j=0}^{i-1} \mathcal{B}_j\right) \end{cases}$$

Input:
$$\begin{cases} (\mathsf{ct}_0, \ldots, \mathsf{ct}_{\kappa-1}), \text{ encrypting } \mathsf{msg} = m_0 + \sum_{i=1}^{\kappa-1} m_i \cdot \left(\prod_{j=0}^{i-1} \mathcal{B}_i\right) \\ \text{s.t. } \mathsf{ct}_i \text{ encrypts message } m_i \text{ with parameters } (\mathcal{B}_i, p_i) \\ \mathsf{PUB}\text{: public material for } \textbf{KS} \text{ and } \textbf{PBS}\text{; } \texttt{/* Remark 2.15 */} \end{cases}$$

Output: $(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1})$, encrypting $\mathsf{msg} = m_0 + \sum_{i=1}^{\kappa-1} m_i \cdot \left(\prod_{j=0}^{i-1} \mathcal{B}_i\right) \mod \Omega$

`/* Decompose message in block` $\kappa-1$ `with respect to base` $(\mathcal{B}_0, \ldots, \mathcal{B}_{\kappa-1})$ `*/`

1 **for** $j \in [0; \kappa-1]$ **do**

2 $\quad c_j \leftarrow \textbf{KS-PBS}(\mathsf{ct}_{\kappa-1}, \mathsf{PUB}, L_j)$

`/* Add (as in Algorithm 3) decomposition to all the blocks up to` $\kappa-2$ `*/`

3 **for** $j \in [0; \kappa-2]$ **do**

4 $\quad \mathsf{ct}'_j \leftarrow \mathsf{Add}(\mathsf{ct}_j, c_j)$

`/* Replace` $\kappa-1$ `block with` $\kappa-1$ `element in decomposition */`

5 $\mathsf{ct}'_{\kappa-1} \leftarrow c_{\kappa-1}$

6 **return** $(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1})$

Observe that the κ **KS-PBS** in Line 2 of Algorithm 42 could be replaced by optimized procedures evaluating several different LUT on the same input ciphertext. A few constructions have been proposed in the literature, such as the **PBS**manyLUT [CLOT21] (see Algorithm 24) or the multi-value bootstrapping [CIM19] (see Algorithm 25).

Theorem 7.4. *(Correctness of Algorithm 42) Let κ ciphertexts ct_i for $i \in [0, \kappa-1]$ representing a large integer modulo Ω as defined in Section 7.2.1. By applying Algorithm 42, we correctly clear the carry space without losing any information from the input ciphertexts.*

Proof *(Theorem 7.4). By construction, we have that $\prod_{j=0}^{\kappa-2} \beta_j < \Omega < \prod_{j=0}^{\kappa-1} \beta_j$. Then, reducing the $(\kappa-1)$-th block encrypting the message $m_{\kappa-1} < p_{\kappa-1}$, rescaled by the product $\prod_{i=0}^{\kappa-2} \beta_i$ modulus*

Ω *is enough to correctly clear its carry space without loosing information. This is homomorphically done by evaluating the* κ *functions* $x \in \mathbb{Z}_{p_{\kappa-1}} \mapsto \mathsf{Decomp}_j\left(x \cdot \prod_{h=0}^{\kappa-2} \beta_h \mod \Omega\right)$ *with* $j \in [0, \kappa-1]$. *Then, for all* $i \in [0; \kappa-1]$, $\mathsf{Decrypt}(c_i) = r_i$, *giving* $r = r_0 + \sum_{i=1}^{\kappa-1} r_i \cdot \left(\prod_{j=0}^{i-1} \beta_j\right)$ *with* $r_i < \beta_i$ *and* $0 \leq r < \Omega$. *The last step is to compute the addition between each* ct_i *but the* $(\kappa-1)$*-th with the* c_i. *The final output is given by* $\mathsf{ct}' = (\mathsf{ct}'_0, \ldots \mathsf{ct}'_{\kappa-1})$. *Then, for all* $i \in [0; \kappa-2]$, $\mathsf{Decrypt}(\mathsf{ct}'_i) = m_i + r_i$, *such that* $\mathsf{Decrypt}(\mathsf{ct}') = m_0 + r_0 + \sum_{i=1}^{\kappa-2}(m_i + r_i) \cdot \left(\prod_{j=0}^{i-1} \beta_j\right) + r_{k-1} \prod_{j=0}^{\kappa-1} \beta_j$. □

Second method for modular reduction. The second method idea is based on the shape of $-\prod_{h=0}^{\kappa-2} \beta_h$ (i.e., the negation of the scaling factor of the message in the $\kappa-1$ block) reduced modulo Ω. The radix decomposition is:

$$\prod_{h=0}^{\kappa-2} \beta_h \mod \Omega = \nu_0 + \nu_1 \cdot \beta_0 + \nu_2 \cdot \beta_0\beta_1 + \cdots + \nu_{\kappa-1} \cdot \prod_{j=0}^{\kappa-2} \beta_j.$$

If $\nu_{\kappa-1} = 0$ and the other elements of the decomposition, i.e., $\nu_0, \nu_1, \ldots, \nu_{\kappa-2}$, are small integers (ideally many of them set to 0), then this method is more efficient. Indeed, when these conditions are respected, the idea is to replace the MSB block by multiplying it by the non-zero constants ν_j and subtracting the results from the j-th input block, for $j \in [0, \kappa-2]$. Some multiplications with positive constants are needed and might require some carry propagation prior to them depending on the degrees of fullness. This method is detailed in Algorithm 44. In the general case where the bases for each block are different, the algorithm performs a homomorphic decomposition into the right base, corresponding to a series of **PBS**s, as detailed in Algorithm 43. This step could be skipped if the bases are compatible. The padding algorithm that follows is simply a padding with zero ciphertexts for the addition and subtraction to work.

Algorithm 43: $(\mathsf{ct}_j)_{j\in[0,\gamma]} \leftarrow \mathsf{Decomp}\,(\mathsf{ct}_{\mathsf{in}}, \beta, \boldsymbol{p}, \mathsf{PUB})$

Context:
- $(q, p, \deg)$: parameters of $\mathsf{ct}_{\mathsf{in}}$
- $\mu := \deg \cdot (p-1)$
- $q_{i,\beta}(x) = \begin{cases} x, \text{ if } i = -1 \\ \left\lfloor \frac{q_{i-1,\beta}(x)}{\mathfrak{B}_i} \right\rfloor, \text{ if } i \geq 0 \end{cases}$
- $r_{i,\beta}(x) = q_{i-1,\beta}(x) - q_{i,\beta}(x) \cdot \mathfrak{B}_i, \ i \geq 0$
- $\gamma_\beta(x) = \begin{cases} \min(i \in \Omega), \ \Omega = \{i < |\beta|, \ q_{i,\beta}(x) = 0\} \\ |\beta| \text{ if } \Omega = \varnothing. \end{cases}$
- $\gamma := \gamma_\beta(\mu)$
- $\boldsymbol{s} \in \mathbb{Z}^n$: the secret key
- $L_{i,\beta}$: a LUT for $x \to r_{i,\beta}(x) \cdot \frac{q}{2 \cdot p_i}, i \in [0, \kappa-1]$

Input:
- $\mathsf{ct}_{\mathsf{in}}$: LWE encryption of a message m
- $(\boldsymbol{p}, \beta) \in \mathbb{N}^{\kappa^2}$
- PUB: public material for **KS** and **PBS**; `/* Remark 2.15 */`

Output: $(\mathsf{ct}_j)_{j\in[0,\gamma]}$ encrypting the message m

```
for j ∈ [0, γ] do
    ct_j ← KS-PBS(ct_in, PUB, L_i)
    with ct_j LWE encryption with parameters (q, B_j, p_j, deg = min((B_j − 1)/(p_j − 1), (q_{j−1,β}(μ) − 1)/(p_j − 1)))
end
return (ct_j)_{j∈[0,γ]}
```

Example of Algorithm 44. Let us develop the algorithm for a 3-blocks integer:

$$m = m_0 + m_1\beta_0 + m_2\beta_0\beta_1 \text{ and } \beta_0\beta_1 = \nu_0 + \nu_1\beta_0 + \nu_2\beta_0\beta_1 \bmod \Omega.$$

therefore,

$$\begin{aligned} m &= m_0 + m_1\beta_0 + m_2\nu_0 + m_2\nu_1\beta_0 + m_2\nu_2\beta_0\beta_1 \\ &= (m_0 + m_2\nu_0) + (m_1 + m_2\nu_1)\beta_0 + (m_2\nu_2)\beta_0\beta_1 \mod \Omega. \end{aligned}$$

If $\nu_2 = 0$, then we will have emptied the last block. Let us try this method on an example: $\Omega = 1055$, $\kappa = 3$, $\boldsymbol{\beta} = (\beta, \beta, \beta)$ with $\beta = 2^5$ and $\boldsymbol{p} = (p, p, p)$ with $p = 2^7$. Observe that $(2^5)^2 \bmod 1055 = -31 = -1 \cdot 2^5 + 1$ (so $\nu_0 = 1$, $\nu_1 = -1$ and $\nu_2 = 0$). Then, the new reduced ciphertext would be composed by:

- in the block 0: the addition between the previous block 0 and the previous block 2 multiplied times 1;
- in the block 1: the addition between the previous block 1 and the previous block 2 multiplied times -1;
- in the block 2: an encryption of 0.

Algorithm 44: $\left(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1}\right) \leftarrow \mathsf{ModReduction}_2((\mathsf{ct}_0, \ldots, \mathsf{ct}_{\kappa-1}), \mathsf{PUB})$

Context: $\begin{cases} \boldsymbol{\nu} = (\nu_0, \nu_1, \ldots, \nu_{\kappa-1}) \text{ be a convenient decomposition s.t.} \\ \prod_{h=0}^{\kappa-2} \beta_h \mod \Omega = \nu_0 + \nu_1\beta_0 + \nu_2\beta_0\beta_1 + \cdots + \nu_{\kappa-2}\prod_{j=0}^{\kappa-3}\beta_j \end{cases}$

Input: $\begin{cases} (\mathsf{ct}_0, \ldots, \mathsf{ct}_{\kappa-1}), \text{ encrypting } \mathsf{msg} = m_0 + \sum_{i=1}^{\kappa-1} m_i \left(\prod_{j=0}^{i-1}\beta_i\right) \\ \text{s.t. } \mathsf{ct}_i \text{ encrypts message } m_i \text{ with parameters } (\beta_i, p_i) \end{cases}$

Output: $\left(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1}\right)$ encrypting $\mathsf{msg} = m_0 + \sum_{i=1}^{\kappa-1} m_i \left(\prod_{j=0}^{i-1}\beta_i\right) \mod \Omega$

```
/* Copy input and set the κ − 1 block to zero (trivial encryption,
   Remark 2.6)                                                    */
```
1 $\left(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1}\right) \leftarrow (\mathsf{ct}_0, \ldots, \mathsf{ct}_{\kappa-2}, \mathbf{0})$

2 **for** $j \in [0; \kappa-2]$ **do**

```
    /* Multiply block κ − 1 times ν_j, Multiplication with a Positive Constant
       as in Theoreme 2.5.                                          */
```
3 **if** $\nu_j < 0$ **then**

4 $c_j \leftarrow \mathsf{ScalarMul}(\mathsf{ct}_{\kappa-1}, -\nu_j)$

5 **else**

6 $c_j \leftarrow \mathsf{ScalarMul}(\mathsf{ct}_{\kappa-1}, \nu_j)$

```
    /* Decompose (Algorithm 43) c_j block starting from the β_j     */
```
7 $(c_{j,0}, \ldots, c_{j,\kappa-j-1}) \leftarrow \mathsf{Decomp}\left(c_j, (\beta_i)_{i\in[j,\kappa-1]}, (p_i)_{i\in[j,\kappa-1]}, \mathsf{PUB}\right)$

```
    /* Update the output                                            */
```
8 **if** $\nu_j < 0$ **then**

9 $\left(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1}\right) \leftarrow \mathsf{Add}\left(\left(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1}\right), \left(c'_{j,0}, \ldots, c'_{j,k-1}, \mathbf{0}, \ldots, \mathbf{0}\right)\right)$

10 **else**

11 $\left(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1}\right) \leftarrow \mathsf{Sub}\left(\left(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1}\right), \left(c'_{j,0}, \ldots, c'_{j,k-1}, \mathbf{0}, \ldots, \mathbf{0}\right)\right)$

12 **return** $\left(\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{\kappa-1}\right)$

Observe that in Algorithm 44 the subtraction algorithm follows the regular schoolbook subtraction modulo an integer Ω.

Theorem 7.5. *(Correctness of Algorithm 44) Let κ ciphertexts ct_i for $i \in [0, \kappa-1]$ representing a large integer modulo Ω as defined in Section 7.2.1. If the degree of fullness of the input $(\kappa-1)$-th block is small enough to be able to perform a constant multiplication times the largest of the constants $\nu_0, \ldots, \nu_{\kappa-2}$ (as defined in Paragraph 6), followed by a homomorphic addition, Algorithm 44, correctly output a modular reduction without losing any information from the input ciphertexts.*

Proof *(Theorem 7.5). If the degree of fullness of the input $(\kappa-1)$-th block is small enough to be able to perform a constant multiplication times the largest of the constants $\nu_0, \ldots, \nu_{\kappa-2}$, followed by a homomorphic addition, the algorithm can start. In fact, the algorithm consists in multiplying the non-zero constants ν_j times the block $\mathsf{ct}_{\kappa-1}$ and then to subtract the result to the input ct_j block, for $j \in [0, \kappa-2]$. The result of this operation, by definition of the constants $\nu_0, \ldots, \nu_{\kappa-1}$, is a new radix-based encryption of msg reduced modulo Ω. In case the bases in the blocks are not the same, a homomorphic decomposition step (as described in Algorithm 43) needs to be performed before addition.*

As for Algorithm 42, this new ciphertext is not a "fresh" ciphertext, in the sense that the carries in the blocks are not all empty (because of the homomorphic addition). A carry propagation step can be applied if necessary and it can be used to continue the computations. □

7.3.2 Larger Integer using Hybrid Representation

As we explained above, the CRT-only approach has some limitations. To overcome them, we create a new homomorphic hybrid representation that mixes the CRT-based approach with the radix-based approach, in order to take advantage of the best of both worlds. The idea is to use the CRT approach as the top layer in the structure, and to represent the CRT residues by using radix-based modular integers when needed: with this approach we do not have any more restrictions on Ω.

Encode. Let $(\Omega_0, \ldots, \Omega_{\kappa-1})$ be integers co-primes to each other, i.e., (Ω_i, Ω_j) co-primes for all $i \neq j$, and let $\Omega = \prod_{i=0}^{\kappa-1} \Omega_i$. To encode a message $\mathsf{msg} \in \mathbb{Z}_\Omega$, as in the CRT-only approach, the message is split into a list of $\{\mathsf{msg}_i\}_{i=0}^{\kappa-1}$ such that $\mathsf{msg}_i = \mathsf{msg} \mod \Omega_i$ for all $0 \leq i < \kappa$. At this point, for each message msg_i for $i \in [0, \kappa-1]$, the encoding used for radix-based modular integers is used (Encode from Definition 8). Then, any CRT residues Ω_i have its own list of radix bases: $(\beta_{i,\kappa_i-1}, \ldots, \beta_{i,0})$ and more generally its parameters $\{(\beta_{i,j}, p_{i,j})\}_{0 \leq j < \kappa_i} \in \mathbb{N}^{2\kappa_i}$. The formal encoding is described in the Figure 7.2.

$$
\mathsf{msg} \mod \Omega \mapsto \begin{cases}
\mathsf{msg}_0 = \mathsf{msg} \mod \Omega_0 \mapsto \begin{cases} \{m_{0,j}\}_{j=0}^{\kappa_0-1} \text{ s.t.} \\ \mathsf{msg}_0 = m_{0,0} + \sum_{j=1}^{\kappa_0-1} m_{0,j} \cdot \left(\prod_{k=0}^{j-1} \beta_{0,k}\right) \\ \text{and } \tilde{m}_{0,j} = \mathsf{Encode}\left(m_{0,j}, p_{0,j}, q\right) \\ \forall\ 0 \leq\ j < \kappa_0 \end{cases} \\
\vdots \\
\mathsf{msg}_{\kappa-1} = \mathsf{msg} \mod \Omega_{\kappa-1} \mapsto \begin{cases} \{m_{\kappa-1,j}\}_{j=0}^{\kappa_{\kappa-1}-1} \text{ s.t.} \\ \mathsf{msg}_{\kappa-1} = m_{\kappa-1,0} + \sum_{j=1}^{\kappa_{\kappa-1}-1} m_{0,j} \cdot \left(\prod_{k=0}^{j-1} \beta_{\kappa-1,k}\right) \\ \text{and } \tilde{m}_{\kappa-1,j} = \mathsf{Encode}\left(m_{\kappa-1,j}, p_{\kappa-1,j}, q\right) \\ \forall\ 0 \leq\ j < \kappa_{\kappa-1} \end{cases}
\end{cases}
$$

Figure 7.2: Hybrid approach visualization combining CRT representation on the top level and radix representation below.

Decode. The decoding is done in two steps: first, each independent radix-based modular integer is decoded to obtain the independent residues modulo $\Omega_0, \ldots, \Omega_{\kappa-1}$, and then the CRT is inverted to retrieve the message modulo Ω.

Arithmetic operations. To perform any homomorphic operation, it is enough to perform the computation on each radix component independently, as shown for the CRT-only approach. Then, depending on the Ω_i values, the modular reduction algorithms (i.e., Alg. 42 or Alg. 44) can be used.

The hybrid approach can be seen as a generalization of both the CRT-only approach (if $\kappa_i = 1$ for all $0 \leq i < \kappa$) and the pure radix-based modular integer approach (if $\kappa = 1$). It also covers the mixed cases where some of the κ_i are equal to 1 and the others are greater.

7.3.3 Generic Lookup Table Evaluation

One of the strengths of TFHE is its ability to evaluate any univariate function using bootstrapping. However, as seen in Chapter 3, the native bootstrapping of TFHE is inefficient for evaluating multivariate functions. To overcome this limitation, the only known efficient solutions are the Tree-PBS [GBA21] and the **WoP-PBS** technique presented in Chapter 6. We now recall how these two bootstrapping techniques can be used for high-precision computations.

Tree PBS approach on Radix-Based Modular Integers. In [GBA21] the plaintext integers are all encrypted under the same basis β: we offer here the possibility to evaluate a large lookup table with integers set in different basis $(\beta_0, \ldots, \beta_{\kappa-1})$.

Let $\Omega = \prod_{i=0}^{\kappa-1} \beta_i$, and let $L = [l_0, l_1, \ldots, l_{\Omega-1}]$ be a LUT with Ω elements. We want to evaluate this LUT on a radix-based modular integer encrypting a message $\mathsf{msg} = m_0 + \sum_{i=1}^{\kappa-1} m_i \prod_{j=0}^{i-1} \beta_j$.

Then, to evaluate the new multi-radix tree-PBS we performs the following steps:

1. We note as $B = \{\beta_i | i \in [0, \kappa - 1]\}$ and as $\vartheta(\beta_i)$ the component m_i of msg associated to β_i.
2. We define $\beta_{\max} = \max(\beta \in B)$.
3. We split the LUT L into $\nu = \frac{\prod_{\beta_i \in B} \beta_i}{\beta_{\max}}$ smaller LUTs $(L_0, \ldots, L_{\nu-1})$ that each contain $\beta_{\max}$ different elements of L.
4. We compute a **PBS** on each of the ν LUTs using the ciphertext encrypting $\vartheta(\beta_{\max})$ as a selector.
5. We build a new large lookup table L by packing, with a key switching, the results of the ν iterations of the **PBS** in previous step.
6. We remove $\beta_{\max}$ from B: $B = B - \beta_{\max}$.
7. We repeat the steps from 2 to 6 until B is empty.

The generalized multi-radix tree-PBS takes as input a radix-based modular integer ciphertext, a large lookup table L and the public material required for the **PBS** and key switching and returns a LWE ciphertext. The signature is: $\mathsf{ct_{out}} \leftarrow \mathsf{Tree\text{-}PBS}((\mathsf{ct}_0, \ldots, \mathsf{ct}_{\kappa-1}), \mathsf{PUB}, L)$.

WoP-PBS approach on Radix-Based Modular Integers. In Chapter 6, we present how to perform the **WoP-PBS** over several ciphertexts with messages encoded in different bases $(\beta_0, \ldots, \beta_{\kappa-1})$.

Let $\lceil \log_2(\beta_i) \rceil = \delta_i$, assuming that $\prod_{i=0}^{\kappa-1} 2^{\delta_i} > N$, we denote $\Omega = \prod_{i=0}^{\kappa-1} 2^{\delta_i}$. Let $L = \left\{\mathsf{LUT}_0, \ldots, \mathsf{LUT}_{\lceil \log_2\left(\frac{\Omega}{N}\right) \rceil - 1}\right\}$ be a LUT with Ω elements where $\{\mathsf{LUT}_i\}_{i \in [0, \lceil \log_2\left(\frac{\Omega}{N}\right) \rceil - 1]}$ is a small lookup table composed of N elements. We want to evaluate this LUT on a radix-based modular integer encrypting a message $\mathsf{msg} = m_0 + \sum_{i=1}^{\kappa-1} m_i \prod_{j=0}^{i-1} 2^{\delta_j}$.

Then, to evaluate the new multi-radix encoding with the **WoP-PBS** we performs the following steps:

1. We first extract the δ_i bits of the i^{th} ciphertext.
2. Each LWE ciphertext encrypting a bit is transformed into GGSW ciphertext using a circuit bootstrapping (Algorithm 12).
3. We then evaluate a vertical packing (Algorithm 14) using the GGSW representing the $\log_2\left(\frac{\Omega}{N}\right)$ most significant bits to select the lookup table $\mathsf{LUT_{res}}$ with $\mathsf{res} = \sum_{i=N}^{\kappa-1} m_i \prod_{j=0}^{i-1} 2^{\delta_j}$.
4. Next, using the N remaining GGSW ciphertexts, we execute a blind rotation corresponding to a rotation by $\sum_{i=0}^{N-1} m_i \prod_{j=0}^{i-1} 2^{\delta_j}$ elements.

5. Finally we sample extract the constant terms which corresponds to the encrypted value $f(m_0, \ldots, \kappa - 1)$.

More details on this algorithm are provided in Chapter 6, in Algorithm 38.

For all the representations, i.e., the CRT, the native CRT, and the hybrid approaches, the multi-radix Tree-PBS and the **WoP-PBS** operate in the same manner. Only the lookup table generation needs to be changed. As studied in Chapter 6, the **WoP-PBS** is more efficient for large precision. In the following, we focus on a 16-bit representation where the **WoP-PBS** is more suitable. Consequently, our benchmark evaluation focuses exclusively on this approach.

7.3.4 Benchmarks

In this section, we provide a few practical benchmarks for integers of sizes 16 and 32 bits. All the cryptographic parameters are detailed in Appendix A.3.1. The specifications of the machine are: Intel(R) Xeon(R) Platinum 8375C@2.90GHz with 504GB of RAM. Note that such an amount of RAM is not needed: all benchmarks can be run on a basic laptop. All implementations are done using TFHE-rs[1].

Compatibility Between $\mathcal{A}^{(\text{this work})}$ and $\mathcal{A}^{(\text{CJP21})}$. We generated couples of parameter sets that are compatible, one for $\mathcal{A}^{(\text{CJP21})}$ and the other for $\mathcal{A}^{(\text{this work})}$. By compatible, we mean that one can go from one to the other freely and smoothly. From $\mathcal{A}^{(\text{CJP21})}$ to $\mathcal{A}^{(\text{this work})}$, one needs to remove the bit of padding in the usual LUT of $\mathcal{A}^{(\text{CJP21})}$'s **PBS**. From $\mathcal{A}^{(\text{this work})}$ to $\mathcal{A}^{(\text{CJP21})}$, one needs to add a bit of padding in the LUT of the usual $\mathcal{A}^{(\text{this work})}$'s WoP-PBS. But we also need other guarantees to be able to freely compose atomic patterns $\mathcal{A}^{(\text{CJP21})}$ and $\mathcal{A}^{(\text{this work})}$. In particular, we need to guarantee that (i) each atomic pattern can absorb/deal with input noise either coming from $\mathcal{A}^{(\text{CJP21})}$ or $\mathcal{A}^{(\text{this work})}$ and (ii) the input LWE dimensions of each atomic pattern are compatible i.e., the product of the GLWE dimension k by the polynomial size N must be equal in both atomic patterns. We could remove constraint (ii) by adding two key switching keys, one to go from $\mathcal{A}^{(\text{CJP21})}$ to $\mathcal{A}^{(\text{this work})}$ and one to go from $\mathcal{A}^{(\text{this work})}$ to $\mathcal{A}^{(\text{CJP21})}$: we leave it as future work.

To satisfy those two conditions, we decided to first solve the optimization problem on $\mathcal{A}^{(\text{this work})}$ and later on $\mathcal{A}^{(\text{CJP21})}$ with more constraints. The first optimization gives us the product $k \cdot N$ and the output variance of $\mathcal{A}^{(\text{this work})}$. Then we solve the optimization problem for $\mathcal{A}^{(\text{CJP21})}$ with an additional constraint for the polynomial size N and the GLWE dimension k to satisfy (i) and using the maximum between the output noise of $\mathcal{A}^{(\text{CJP21})}$ and $\mathcal{A}^{(\text{this work})}$ as the input noise of $\mathcal{A}^{(\text{CJP21})}$ which satisfies (ii). This approach works well for parameter couples (#1,#8) and (#2,#9). But for the last parameter set couple (#3,#10), there is no solution for $\mathcal{A}^{(\text{CJP21})}$ with the aforementioned constraints. For this special case, we reverse the order of the optimization and first solve the optimization problem for $\mathcal{A}^{(\text{CJP21})}$ and then for $\mathcal{A}^{(\text{this work})}$ with the additional constraints mentioned above.

7.3.4.1 Experimental results

The tables presented in this section contain timings related to 16 and 32-bit integer operations using the radix approach (Table 7.1), the CRT approach (Table 7.2) and the native CRT approach (Table 7.3). The benchmarks measure timings to compute homomorphic additions, multiplications, carry cleanings (apart from the native CRT approach) and LUT evaluations (only for 16-bits integers). As explained in Remark 6.6, it is not doable to evaluate LUT on 32-bit integers.

Radix Approach. In Table 7.1, dedicated to the radix approach, we display two instances of 16-bit integers and three instances of 32-bit integers. The number of additions is bounded by the room available in the carry buffer, and once it is full, a carry cleaning is needed.

[1] `https://github.com/zama-ai/tfhe-rs`

For the 16-bit integers, it is possible to use both the $\mathscr{A}^{(\text{CJP21})}$ and the $\mathscr{A}^{(\text{this work})}$ operators. This means that for 16-bits integer, classical arithmetic uses the usual **PBS** ($\mathscr{A}^{(\text{CJP21})}$), and LUT evaluation is done with the **WoP-PBS** ($\mathscr{A}^{(\text{this work})}$). We assume that the **WoP-PBS** is done over integers with free carry buffers (i.e., after a carry cleaning). The parameters have been generated as described in Paragraph 7.3.4. Note that the addition does not require any PBS to be computed (this is denoted with a star $+^*$), but is done accordingly to the parameters generated for the bootstrapping.

For 32-bit integers, only arithmetic operations are possible. So, cryptographic parameters are optimized following $\mathscr{A}^{(\text{CJP21})}$ only. Hence, some operations are executed faster on 32-bit integers than on 16-bit ones.

integer parameters				**PBS based operations**				**WoP-PBS based operations**	
Ω	p	carry modulus	κ	param ID	$+^*$	$\times$	carry cleaning	param ID	LUT evaluation
2^{16}	2^1	2^1	16	#1	12.8 µs	29.0 s	932 ms	#8	823 ms
2^{16}	2^2	2^2	8	#2	6.67 µs	5.73 s	657 ms	#9	1.80 s
2^{32}	2^1	2^1	32	#4	19.1 µs	43.8 s	685 ms	∅	
2^{32}	2^2	2^2	16	#5	12.3 µs	9.60 s	514 ms		
2^{32}	2^4	2^4	8	#6	137 µs	25.0 s	6320 ms		

Table 7.1: Sequential benchmarks for 16-bit and 32-bit homomorphic integers based on the radix approach. The star (*) means that a PBS is not required to compute the operation.

Remark 7.1 (Multiplication Failure Probability)**.** When 32-bit integers are represented with 32 blocks (i.e., $\kappa = 32$), the number of AP of type $\mathscr{A}^{(\text{CJP21})}$ required to compute a multiplication is quadratic in the number of blocks. Because the error probability p_{fail} of this AP is bounded by $2^{-13.9}$ in our experiments, the error probability at the level of the multiplication will be increased greatly. Timings are clearly not in favor of this representation, and the probability of having an error is small enough for the other representations (with a smaller number of blocks). One solution is to keep the same value of p_{fail} and consider a small enough κ, resulting in a better trade-off between running time and failure probability at the multiplication level (e.g., the one associated with the parameter ID#5). Another way of solving this problem would be to have another parameter set dedicated to the multiplication algorithm, with a smaller failure probability p_{fail} but we leave that as an future work.

CRT Approach. Table 7.2 is dedicated to the CRT approach. In this representation, each block have a dedicated basis, and are independent by construction. We display one instance for 16-bit integers and another one for 32-bit integers. For both of them we show the total time needed to compute the operations, as well as the amortized time when the implementation is multi-threaded. As for Table 7.1, the number of additions is bounded by the room available in the carry buffer, and once it is full, a carry cleaning in needed. Note that in the case of homomorphic evaluation of polynomial functions, using the CRT representation offers better timings, since it is sufficient to compute a **PBS** on each CRT residue. The timings are then the same as the ones of the carry cleaning when there is one block per residue, otherwise it means that we are considering the hybrid approach, and in that case, it is the cost of a LUT evaluation separately on each block.

For the 16-bit integers, the basis is given by $\Omega = 2^3 \cdot 3^2 \cdot 7 \cdot 11 \cdot 13 \approx 2^{16}$. As for 16-bit in radix representation, it is possible to use both the $\mathscr{A}^{(\text{CJP21})}$ and the $\mathscr{A}^{(\text{this work})}$ operators. However, the major difference here is about the parameter optimization: in this case, the atomic pattern $\mathscr{A}^{(\text{CJP21})}$ has been privileged. Thus, the timings for the evaluating a LUT using a **WoP-PBS** are way slower. By removing the constraint of compatibility, the performance should be closer to the one of Table 7.3. The **WoP-PBS** is parallelized by extracting bits for each block independently. Then, each LUT evaluation outputting one block (and taking all bits as input) is computed in parallel: note that this approach could also be applied in the case of the radix decomposition.

We consider the basis defined by $\Omega = 2^5 \cdot 3^5 \cdot 5^4 \cdot 7^4 \approx 2^{32}$ to represent 32-bit integers using the hybrid representation. For instance, to represent integers under the modulus 7^4, we use radix-based

integers with 4 blocks and a message modulus equals to 7. Thanks to the CRT representation, by using this basis, multiplications can be computed with the fast bi-variate **PBS** approach described in Algorithm 16.

Ω	type of execution	PBS based operations				WoP-PBS based operations	
		param ID	$+^*$	$\times$	carry cleaning	param ID	LUT evaluation
$\approx 2^{16}$	sequential	#3	8.36 µs	401 ms	251 ms	#10	23.1 s
	5 threads		1.67 µs	80.3 ms	50.2 ms		4.61 s
$\approx 2^{32}$	sequential	#7	27.6 µs	5.17 s	2400 ms	∅	
	4 threads		8.78 µs	1.82 s	729 ms		

Table 7.2: Benchmarks for 16-bit homomorphic integers based on the CRT approach and 32-bit integers are computed with a hybrid approach. We use the following CRT basis: $\Omega = 7{\cdot}8{\cdot}9{\cdot}11{\cdot}13 \approx 2^{16}$ and $\Omega = 2^5 \cdot 3^5 \cdot 5^4 \cdot 7^4 \approx 2^{32}$.

Native CRT Approach. In Table 7.3, dedicated to the native CRT approach (detailed in Paragraph 7.2.2), we display one instance of 16-bit integers and another of 32-bit integers. We consider $\Omega = 7 \cdot 8 \cdot 9 \cdot 11 \cdot 13 \approx 2^{16}$ and respectively $\Omega = 3 \cdot 11 \cdot 13 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 32 \approx 2^{32}$ Since there is no carry buffer in this representation, there is no need for a carry cleaning. However, to avoid incorrect computation, the number of additions is bounded for these parameter sets by the value ν. Once this bound is reached, a **WoP-PBS** is required to reduce the noise.

We observe a slower latency for the multiplication with 32-bit integers, 36.8 seconds, which leads us to think that for the precision around 32 bits, a hybrid approach is more efficient. Indeed, the native CRT approach requires to have in a single LWE ciphertext a small enough noise (after the bootstrapping) to preserve the message (with a size equal to the co-prime modulo) and the room for the 2-norm ν needed to compute multiplications with known integers or additions between ciphertexts. So when one tries to build a big Ω, since small prime numbers are not infinite, they end up with big co-prime residues and as a consequence needs big 2-norm which means very slow parameter sets.

Ω	type of execution	WoP-PBS based operations				
		param ID	$+^*$		$\times$	LUT evaluation
			ν	time		
$\approx 2^{16}$	sequential	#11	5	4.32 µs	7.42 s	3.81 s
	5 threads			0.862 µs	1.65 s	0.761 s
$\approx 2^{32}$	sequential	#12	2^5	6.98 µs	36.8 s	∅
	8 threads			0.873 µs	5.31 s	

Table 7.3: Benchmarks for 16-bit and 32-bit homomorphic integers based on the native CRT approach. We use the following CRT basis: $\Omega = 7{\cdot}8{\cdot}9{\cdot}11{\cdot}13 \approx 2^{16}$ and $\Omega = 3{\cdot}11{\cdot}13{\cdot}19{\cdot}23{\cdot}29{\cdot}31{\cdot}32 \approx 2^{32}$.

Chapter 8

From Integers to Floating-points

In Chapter 7, we presented different methods to efficiently represent large integers and mimic the precision of modern architectures. In addition to 64- and 32-bit integers, another commonly used representation is floating-point numbers. This chapter focuses on representing homomorphic floating-point numbers, which completes the classical representation framework, allowing us to mimic all conventional number formats used in the clear domain.

In this chapter, we propose two new methods to represent floating-point numbers. The first is based on the **WoP-PBS** (Algorithm 38) presented in Chapter 6, and suits well for floating-point numbers with low precision, but is impractical for larger precision (see Remark 6.6). The second method relies on the radix representation introduced in Chapter 7. This representation can be efficiently used for precisions not covered by the first technique. With these two techniques, we can now efficiently reach a wide range of floating-point precisions.

8.1 Introduction

In the current landscape, FHE schemes, and especially TFHE, are not designed to seamlessly integrate with established floating-point standards, which are tailored to match hardware specifications, such as bit sizes and the frequent use of conditional instructions. This chapter introduces novel homomorphic operators that can serve as the building blocks for constructing homomorphic floating-point arithmetic with arbitrary precision within a given FHE framework.

To construct floating-point arithmetic, we must represent mantissas $\mathfrak{m}$ and exponents $\mathfrak{e}$, which can be viewed as large integers as presented in Chapter 7 and the sign $\mathfrak{s}$, which can be viewed as a boolean ciphertext. We introduce novel algorithms that automatically retain only the most significant bits and discard some of the least significant bits, maintaining the same representation throughout. This mimics a well-known round mode of floats, called the rounding towards zero mode. We refer to [MBDD$^+$18, Section 2.2.1] or to [Hwa24] for more details. Our first method, based on Chapter 6, ensures the correct encoding after each operation. This method is effective for small mantissas and exponents but does not scale well. The second method, based on Chapter 7, leverages circuit bootstrapping to obtain the carry value, allowing the ciphertext to be updated accordingly. This approach is faster than the first method for large mantissas or exponents.

Building on large integers, we introduce two different methods to build efficient floating-point arithmetic, each with its own pros and cons. The first approach, detailed in Section 8.3.1, heavily utilizes the alternative **PBS** proposed in Chapter 6 (referred to as **WoP-PBS**) to perform operations on floating-point numbers. This method allows for the efficient evaluation of functions on ciphertexts that encrypt large integers. Our approach, while straightforward, distinguishes itself

from other homomorphic floating-point methods by representing a floating-point number within a single ciphertext (or more). This technique is especially effective when working with low precision floating-point numbers, typically up to 12 bits.

The second approach, which is the core of this contribution, constructs floating-point arithmetic based on TFHE that can follow standards such as [Ins08] and is not constrained by precision. Our new algorithms make extensive use of an extended version of CMux, a homomorphic operator that selects between two inputs based on an encrypted decision bit. This method is key to developing faster homomorphic operations that effectively combine the sign, mantissa, and exponent of one or more homomorphic floats. During homomorphic floating-point operations, to compute the resulting mantissa, an extra LWE ciphertext is used to make the operation both faster and more precise. Indeed, since we work in the worst case scenario to avoid losing any information, it is necessary to preserve the message contained in the last carry of the mantissa. This approach, soberly titled homomorphic floating-point (HFP). As it reaches execution times close to the second, compared to other implementations that are slower by several minutes, it is the first that can be considered suitable for deployment in real world use cases. Beyond the description of all arithmetic operations, including the division, we also provide their noise analysis and the hypotheses done to generate the cryptographic parameters and their associated failure probability. We also include algorithms to efficiently compute the ReLU and the sigmoid, two of the most used functions in machine learning. We show how to easily extend our approach to take into account some floating-point subtleties, like the special values ($\pm\infty$ or NaN). As a simple application and showcase of the versatility of floating-points, we briefly detail how to compute the approximation of any functions. In the end, our method outperforms the state-of-the-art (as shown in the next section), which was more about showing that floating-points might be doable with TFHE, rather than giving a practical solution as we do here.

8.1.1 Prior Approaches

Efforts to develop efficient FHE computation methods for real numbers can be categorized into two approaches: the fixed-point arithmetic approach and the floating-point arithmetic approach. Most of the first attempts [CSVW16, Lai17, AN16] focus on the BFV scheme [Bra12, FV12]. In [CSVW16, Lai17], the authors chose the fixed-point approach. Roughly, their idea is to decompose a real number into two integers, one representing the value before the point and the second representing the value after the point. The binary decomposition of the two integers is encrypted in one RLWE ciphertext such that the integer part is encrypted over the coefficient of small degree and the fractional part is encrypted on the coefficient of high degree. This method encounters two significant limitations: first, after several operations, accuracy is compromised due to the mixing of the fractional and the integer parts of the number. Second, the computation must remain within a certain modulus limit; exceeding this threshold also results in a loss of correctness. Thus, using fixed-point arithmetic is particularly suitable for FHE schemes where the depth of the circuit is somewhat bounded, since they share a similar constraint. In fact, an encrypted floating-point number is often represented by one ciphertext for the sign, one or several ciphertexts for the mantissa and one or several ciphertexts for the exponent. Their approach is also based on FV, whose bootstrapping is neither efficient nor programmable. This results in unpractical methods that cannot be adapted to TFHE.

In the TFHE context, to the best of our knowledge, only two techniques have been studied [ML20] and [LS22]. The former [ML20] uses the traditional floating-point representation, where each LWE ciphertext contains one boolean value. Then, they rely only on boolean gate operations to perform the floating-point computation. Beyond the lack of space efficiency, the major problem with this solution is its time complexity. In the latter [LS22], the authors take advantage of RLWE ciphertexts to represent floats. They have a floating-point in three parts: an RLWE for the sign, one for the mantissa and another one for the exponent. In this representation, the sign and the exponent are each represented on the first coefficient of their RLWE. The mantissa is represented by several coefficients depending on the base of the decomposition. In this work, they propose a method to detect overflow based on an encrypted witness. As previously, the main drawback lies

in the time complexity, which suffers from an exponential factor related to the size of the exponent and the use of tensor product and relinearization (Algorithm 20).

In our work, we use the standard representation proposed before (several LWE ciphertexts for the mantissa, several LWE ciphertexts for the exponent and one LWE ciphertext for the sign). The main change in our algorithm is the use of circuit bootstrapping (CBS) which is costlier than a **PBS** ($\mathsf{Cost}_{\mathbf{CBS}} \approx \ell_{\mathbf{CBS}} \cdot \mathsf{Cost}_{\mathbf{PBS}}$ (See Remark 3.1)) but gives the possibility to perform CMuxes and reduce the cost of the TFHE algorithm used in the floating-point arithmetic. Note that our work could benefit from recent optimizations done in [WWL+24].

The code for [ML20] is not publicly available, and despite significant effort, we were unable to successfully run the code provided with [LS22]. As a result, the comparison below is based on the timings reported in the respective papers. In [ML20], the experiments were conducted on an Intel i7−6700@3.40 GHz (up to 4 GHz) with 8 threads. Since the computational model (i.e., sequential or parallel) is not specified, we assume these experiments were run sequentially. In the latter paper [LS22], experiments were run on an Intel Xeon Silver 4210@2.40 GHz, with 40 threads.

As shown in Table 8.1, our approach significantly outperforms existing methods, achieving at least an 8-fold improvement (for 32-bit floating-point multiplication) and up to 100-fold improvement (for 64-bit floating-point addition). Using a m6i.metal instance Intel Xeon 8375C (Ice Lake) at 3.5 GHz, with 128 vCPUs and 512.0 GiB of memory using the TFHE-rs library [Zam22], the sequential timings are:

Paper (Precision)	**Add**	**Mul**
[ML20] (32-bits)	490s	162s
[LS22] (32-bits)	530s	443s
Ours (32-bits)	7s	20s
[ML20] (64-bits)	1200s	686s
[LS22] (64-bits)	858s	808s
Ours (64-bits)	12s	82s

Table 8.1: Comparison of addition and multiplication times with the state-of-the-art.

As explained above, each technique was evaluated on different machines, with some assumptions regarding the nature of the computation (sequential or parallel). To provide a more complete comparison, Table 8.2 presents a complexity comparison in terms of the number of PBS operations required for the main operations (addition and multiplication) between our work and the state-of-the-art. This comparison highlights the removal of the exponential factor in the complexity formulas, further emphasizing the efficiency of our approach.

In this context, the terms $\rho_{\mathfrak{m}} \cdot \ell_{\mathfrak{m}}$ and $\rho_{\mathfrak{e}} \cdot \ell_{\mathfrak{e}}$ represent the number of bits in the mantissa and exponent, respectively. In our method, $\rho_{\mathfrak{m}}$ (and $\rho_{\mathfrak{e}}$) corresponds to the message space in each LWE ciphertext for the mantissa (and exponent), while $\ell_{\mathfrak{m}}$ (and $\ell_{\mathfrak{e}}$) indicates the number of ciphertexts representing the mantissa (and exponent). Considering the complexity of the two previous works based on TFHE, we achieve a significant gain by eliminating the exponential factor for addition (present in both previous techniques) and for multiplication specifically in [LS22].

	Addition Complexity	Multiplication Complexity
[ML20]	$\approx 2^{\rho_{\mathfrak{e}}\ell_{\mathfrak{e}}}(\ell_{\mathfrak{m}}\rho_{\mathfrak{m}} + \ell_{\mathfrak{e}}\rho_{\mathfrak{e}})\log(\ell_{\mathfrak{m}}\rho_{\mathfrak{m}} + \ell_{\mathfrak{e}}\rho_{\mathfrak{e}})\ \mathsf{Cost}_{\mathbf{PBS}}$	$\approx (\ell_{\mathfrak{m}}\rho_{\mathfrak{m}})^2 + \ell_{\mathfrak{e}}\rho_{\mathfrak{e}}\log(\ell_{\mathfrak{e}}\rho_{\mathfrak{e}})\ \mathsf{Cost}_{\mathbf{PBS}}$
[LS22]	$> 2^{\rho_{\mathfrak{e}}\ell_{\mathfrak{e}}+1}\ \mathsf{Cost}_{\mathbf{PBS}}$	$> 2^{\rho_{\mathfrak{e}}\ell_{\mathfrak{e}}+1}\ \mathsf{Cost}_{\mathbf{PBS}}$
Ours	$(3 \cdot \ell_{\mathfrak{m}} + 6 \cdot \ell_{\mathfrak{e}} + 3)\mathsf{Cost}_{\mathbf{PBS}} + (\ell_{\mathfrak{m}} + \ell_{\mathfrak{e}} \cdot \rho_{\mathfrak{e}} + 3)\mathsf{Cost}_{\mathbf{CBS}}$	$\left(\frac{{\ell_{\mathfrak{m}}}^2}{2}\left(1 + \frac{1}{\rho_{\mathfrak{e}}}\right) + \ell_{\mathfrak{m}}\left(2 + \frac{1}{\rho_{\mathfrak{e}}}\right) + 4\right)\mathsf{Cost}_{\mathbf{PBS}} + \mathsf{Cost}_{\mathbf{CBS}}$

Table 8.2: Our Method vs. Existing Works.

8.2 Preliminaries

The constructions proposed in this chapter mainly rely on the radix representation introduced in Chapter 7. In particular, we focus on the case where the carry space and the message space are equal, i.e., $p = \beta^2$ (see Section 7.2.1). To distinguish the different parts of a floating-point number, we use the notation $\mathfrak{m}$ for the mantissa, $\mathfrak{e}$ for the exponent, and $\mathfrak{s}$ for the sign. When an algorithm applies equally to either the mantissa or the sign, we use the generic notation $\mathfrak{z}$.

8.2.1 New Integer Algorithms

Extended Carry Propagate. By construction, after a given number of operations, the carry space is full and we need to call the CarryPropagate algorithm to homomorphically propagate the carries (See Theorem 7.1). The goal of this algorithm is to clear the carry space of each input ciphertext without losing information, except for the most significant carry, which is lost. This loss occurs because, in Chapter 7, the integer representation works modulo $\rho_{\mathfrak{z}}^{\ell_{\mathfrak{z}}}$.

When dealing with floating-point numbers, however, losing this information is not acceptable, especially when the mantissa is full. By retaining this information, we can grow the exponent if necessary. To address this limitation, we modified the CarryPropagate Algorithm 39 and named it CarryPropagateExtended, which is detailed in Algorithm 45. Note that to recover the original CarryPropagate algorithm, it is sufficient to execute the loop from 0 to $\ell_{\mathfrak{z}} - 2$.

Algorithm 45: $\mathbf{ct}_{\mathfrak{z}_{\text{out}}} \leftarrow$ CarryPropagateExtended $(\mathbf{ct}_{\mathfrak{z}_{\text{in}}}, \mathsf{PUB})$

Context:
- $\Delta_{\mathfrak{z}}$: scaling factor
- $\mathsf{LUT}_{\mathsf{Carry}}$: LUT to return the carry of the ciphertext (return $\left\lfloor \frac{m}{\beta_{\mathfrak{z}}} \right\rfloor$).
- $\mathsf{LUT}_{\mathsf{Msg}}$: LUT to return the message of the ciphertext (return $m \mod p_{\mathfrak{z}}$).
- $p_{\mathfrak{z}}$: the carry-message modulus
- $\beta_{\mathfrak{z}}$: the message modulus

Input:
- $\mathbf{ct}_{\mathfrak{z}_{\text{in}}} = [\mathsf{ct}_{\mathfrak{z}_{\text{in}},\ell_{\mathfrak{z}}-1}, \ldots, \mathsf{ct}_{\mathfrak{z}_{\text{in}},0}] \in \mathbb{Z}_q^{(n+1)\cdot\ell_{\mathfrak{z}}}$
- PUB : Public materials for **KS** and **PBS**; `/* Remark 2.15 */`

Output:
- $\mathbf{ct}_{\mathfrak{z}_{\text{out}}} = [\mathsf{ct}_{\text{tmp}}, \mathsf{ct}_{\mathfrak{z}_{\text{out}},\ell_{\mathfrak{z}}-1}, \cdots \mathsf{ct}_{\mathfrak{z}_{\text{out}},0}] \in \mathbb{Z}_q^{(n+1)\cdot(\ell_{\mathfrak{z}}+1)}$

```
1 for i ∈ [0..ℓ_z − 1] do
      /* Extract the carry                                              */
2     ct_tmp ← KS-PBS(ct_{z_in,i}, PUB, LUT_Carry) ;   /* Algorithm 4 and 11 */
      /* Extract the message                                            */
3     ct_{z_out,i} ← KS-PBS(ct_{z_in,i}, PUB, LUT_Msg) ;   /* Algorithm 4 and 11 */
4     if i ≠ ℓ_z−1 then
5         ct_{z_in,i+1} ← ct_{z_in,i+1} + ct_tmp
6 return (ct_{z_out} = [ct_tmp, ct_{z_out,ℓ_z−1}, ··· ct_{z_out,0}])
```

Integer Subtraction. In the following floating-point algorithms, we require an operation that takes as input two vectors of ciphertexts and returns the sign of the subtraction along with a vector of ciphertexts representing the absolute value of the subtraction. This method is detailed in Algorithm 46. Intuitively, in the initial steps, an offset is added to ensure that the messages in each ciphertext of the first input are larger than those of the second input. Next, we perform the subtraction between the adjusted first input and the second input, followed by a carry propagation as described in Algorithm 45. We then extract the most significant bit (MSB) of the top block from the resulting ciphertext. Finally, we traverse all the blocks, returning the value if the MSB

is set to 1, or the opposite if it is not. The sign is encoded in the padding bit, with an offset such that the most significant bit is 0 for positive values and 1 for negative values.

Lemma 8.1 (IntSub* (Algorithm 46))**.** *Let* $\mathbf{ct}_{\mathfrak{z}i} = [\mathsf{ct}_{\mathfrak{z}i,\ell_\mathfrak{z}-1}, \ldots, \mathsf{ct}_{\mathfrak{z}i,0}] \in \left[\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{z}_{i,\ell_\mathfrak{z}-1}), \cdots \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{z}_{i,0})\right] \subseteq \mathbb{Z}_q^{(n+1)\cdot\ell_\mathfrak{z}}$ *with* $i \in \{0,1\}$ *be two ciphertexts encrypting* $\mathfrak{z}_i \in \mathbb{N}$.

Then, Algorithm 46 returns $(\mathbf{ct}_{\mathfrak{z}_{\mathsf{sub}}}, \mathsf{ct}_\mathfrak{s})$ *with* $\mathsf{Decrypt}_{\boldsymbol{s}}(\mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}}}) = |\mathfrak{z}_1 - \mathfrak{z}_2|$ *and* $\mathsf{Decrypt}_{\boldsymbol{s}}(\mathsf{ct}_\mathfrak{s}) = \mathsf{Sign}(\mathfrak{z}_1 - \mathfrak{z}_2)$.

The complexity of Algorithm 46 can be defined as $\mathsf{Cost}^{\ell_\mathfrak{z},\ell,k,N,n,q}_{\mathsf{RadixSub}^*} = (3 \cdot \ell_\mathfrak{z} + 1) \cdot \left(\mathsf{Cost}^{\ell,k,N,n,q}_{\mathbf{PBS}} + \mathsf{Cost}^{\ell,n,k,N}_{\mathbf{KS}}\right)$.

Algorithm 46: $(\mathbf{ct}_{\mathfrak{z}_{sub}}, \mathsf{ct}_{\mathsf{sign}}) \leftarrow \mathsf{RadixSub}^*\ (\mathbf{ct}_{\mathfrak{z}1}, \mathbf{ct}_{\mathfrak{z}2}, \mathsf{PUB})$

Context:
- $\Delta_\mathfrak{z}$: scaling factor
- $\mathsf{LUT}_{\mathsf{Extract}}$: LUT to extract the MSB (i.e., the $(\log_2(q)-2)^{th}$bit)
- LUT_f : LUT to return $\mathfrak{z} - q/4$ if the MSB equals 1; $q/4 - \mathfrak{z}$ otherwise
- $p_\mathfrak{z}$: the carry-message modulus
- $\beta_\mathfrak{z}$: the message modulus

Input:
- $\mathbf{ct}_{\mathfrak{z}1} = [\mathsf{ct}_{\mathfrak{z}1,\ell_\mathfrak{z}-1}, \ldots, \mathsf{ct}_{\mathfrak{z}1,0}] \in \mathbb{Z}_q^{(n+1)\cdot\ell_\mathfrak{z}}$
- $\mathbf{ct}_{\mathfrak{z}2} = [\mathsf{ct}_{\mathfrak{z}2,\ell_\mathfrak{z}-1}, \ldots, \mathsf{ct}_{\mathfrak{z}2,0}] \in \mathbb{Z}_q^{(n+1)\cdot\ell_\mathfrak{z}}$
- PUB : Public materials for **KS**, **PBS** and **CBS**; `/* Remark 2.15 */`

Output:
- $\mathbf{ct}_{\mathfrak{z}_{\mathsf{sub}}} = [\mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}},\ell_\mathfrak{z}-1}, \cdots \mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}},0}] \in \mathbb{Z}_q^{(n+1)\cdot\ell_\mathfrak{z}}$
- $\mathsf{ct}_\mathfrak{s} \in \mathbb{Z}_q^{n+1}$

1 **for** $i \in [1..\ell_\mathfrak{z} - 1]$ **do**
  `/* Ensure that` $\mathsf{ct}_{\mathfrak{z}1,i}$ `is larger than` $\mathsf{ct}_{\mathfrak{z}2,i}$ `*/`
2 $\mathsf{ct}_{\mathfrak{z}1,i} \leftarrow \mathsf{ct}_{\mathfrak{z}1,i} + \mathsf{TrivialEncrypt}(2^{2\cdot\rho_\mathfrak{z}-1}, 1)$; `/* Remark 2.6 */`
3 $\mathsf{ct}_{\mathfrak{z}1,i} \leftarrow \mathsf{ct}_{\mathfrak{z}1,i} - \mathsf{TrivialEncrypt}(2^{\rho_\mathfrak{z}-1}, 1)$; `/* Remark 2.6 */`
4 $\mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}},i} \leftarrow \mathsf{ct}_{\mathfrak{z}1,i} - \mathsf{ct}_{\mathfrak{z}2,i}$
5 $\mathsf{ct}_{\mathfrak{z}1,0} \leftarrow \mathsf{ct}_{\mathfrak{z}1,0} + \mathsf{TrivialEncrypt}(2^{2\cdot\rho_\mathfrak{z}-1}, 1)$; `/* Remark 2.6 */`
6 $\mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}},0} \leftarrow \mathsf{ct}_{\mathfrak{z}1,0} - \mathsf{ct}_{\mathfrak{z}2,0}$
7 $\mathbf{ct}_{\mathfrak{z}_{sub}} \leftarrow \mathsf{CarryPropagate}(\mathbf{ct}_{\mathfrak{z}_{sub}} = [\mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}},\ell_\mathfrak{z}-1}, \ldots, \mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}},0}], \mathsf{PUB})$; `/* Algorithm 39 */`
  `/* Extract the msb to get the sign */`
8 $\mathsf{ct}_\mathfrak{s} \leftarrow$ **KS-PBS**$(\mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}},\ell_\mathfrak{z}-1}, \mathsf{PUB}, \mathsf{LUT}_{\mathsf{Extract}})$; `/* Algorithm 4 and 11 */`
  `/* Return the value if MSB==1, the opposite otherwise */`
9 **for** $i \in [0..\ell_\mathfrak{z} - 1)$ **do**
10 $\mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}},i} \leftarrow \mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}},i} + \mathsf{ct}_\mathfrak{s}$
11 $\mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}},i} \leftarrow$ **KS-PBS**$(\mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}},i}, \mathsf{PUB}, \mathsf{LUT}_\mathsf{f})$; `/* Algorithm 4 and 11 */`
12 $\mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}},\ell_\mathfrak{z}-1} \leftarrow$ **KS-PBS**$(\mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}},\ell_\mathfrak{z}-1}, \mathsf{PUB}, \mathsf{LUT}_\mathsf{f})$; `/* Algorithm 4 and 11 */`
  `/* Put the sign on the padding bit plus flip the bit to keep the representation 0 is positive and 1 is negative */`
13 $\mathsf{ct}_\mathfrak{s} \leftarrow \mathsf{ct}_\mathfrak{s} \cdot 2 + \mathsf{TrivialEncrypt}(q/2, 1)$; `/* Remark 2.6 */`
14 **return** $(\mathsf{ct}_\mathfrak{s}, \mathbf{ct}_{\mathfrak{z}_{sub}} = [\mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}},\ell_\mathfrak{z}-1}, \ldots, \mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}},0}])$

Proof *(Correctness of Algorithm 46). In this algorithm,* $\mathsf{CT}_\mathfrak{z}$ *can represent the mantissa or the exponent. The goal of this algorithm is to return* $|\mathfrak{z}_1 - \mathfrak{z}_2|$ *and* $\mathsf{Sign}(\mathfrak{z}_1 - \mathfrak{z}_2)$*. The first step of this algorithm is to ensure that the message* $\mathfrak{z}_1$ *is bigger than* $\mathfrak{z}_2$*. To do that, we add* $2^{(\ell_\mathfrak{z}+1)\cdot\rho_\mathfrak{z}-1}$ *to* $\mathfrak{z}_1$*, which automatically guarantees that all the* LWE *that compose* $\mathfrak{z}_1$ *such that all the* LWE *of* $\mathfrak{z}_1$ *are bigger than the ones of* $\mathfrak{z}_2$*. Now we can perform the subtraction term by term. After a carry propagate, if* $\mathfrak{z}_1 > \mathfrak{z}_2$*, we retrieve on the most significant* LWE *the added value at the beginning of the algorithm. Otherwise, this added value is used during the subtraction. This value corresponds to* $\mathsf{Sign}(\mathfrak{z}_1 - \mathfrak{z}_2)$*. By extracting this sign, and adding it to each* LWE*, with a* **PBS***, we can see whether*

$\mathfrak{z}_1 > \mathfrak{z}_2$ *or not and choose between the ciphertext obtain after the subtraction or its opposite and get* $|\mathfrak{z}_1 - \mathfrak{z}_2|$.

□

Lemma 8.2 (Noise Constraints of Algorithm 46)**.** *The output noise variances of ciphertexts of Algorithm 46,* $\mathsf{ct}_{\mathfrak{s}}$ *and* $\mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}}}$*, are respectively* $4 \cdot \sigma_{\mathsf{BR}}^2$ *and* σ_{BR}^2.

To guarantee correctness of this operation, we need to find parameters that verify the following inequality:

$$2 \cdot \sigma_{\mathsf{in}}^2 + \sigma_{\mathsf{BR}}^2 + \sigma_{\mathsf{KS}}^2 + \sigma_{\mathsf{MS}}^2 \leq t^2.$$

with σ_{BR} *the noise added by the blind rotation,* σ_{KS} *the noise added by the key switch,* σ_{cmux} *the noise added by the* CMux *and finally* σ_{MS}*, the noise added by the modulus switch and where* t *is a noise bound such that* $t = \frac{\Delta}{2 \cdot z^*(p_{\mathsf{fail}})}$ *with the standard score* $z^*(p_{\mathsf{fail}}) = \sqrt{2} \cdot \mathsf{erf}^{-1}(1 - p_{\mathsf{fail}})$ *and the scaling factor* Δ *introduced in Subsection 2.2.1.*

Proof *(Lemma 8.2). In this proof, we use the same techniques as those introduced in Chapter 2, in Section 2.5. In particular, we use the noise bound (Definition 23), a quantity representing the maximum noise variance that still guarantees the correctness up to some failure probability. We also use a simpler version of Theorem 27 which consists of removing redundant inequalities and dominated constraints. Simply put, if we have two inequalities* $f(x) + g(x) \leq t$ *and* $f(x) \leq t$ *with* f *and* g *two positive functions, we can focus on the first one as the second one will be automatically satisfied if the first is. In this proof and the next ones, when this situation arises, we will say that the second inequality is dominated by the first.*

Let us look at the noise propagation in Algorithm 46. We assume that each input ciphertext contains a noise following a centered Gaussian distribution with a variance σ_{in}^2*. These noises are also assumed to be statistically independent.*

If we find parameters that guarantee the inequality above, the bootstrapping will be successful with probability $1 - p_{\mathsf{fail}}$*. At the end of line 6, the variance of the noise in* $\mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}},i}$ *is* $2 \cdot \sigma_{\mathsf{in}}^2$ *for* $0 \leq i < l_{\mathfrak{z}}$*. The worst operation in terms of noise in the carry propagation on line 7 consists of adding a freshly bootstrapped ciphertext with one of the* $\mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}},i}$ *and applying to it a key switch and a bootstrapping. It means that to have correctness we must verify the following inequality:*

$$2 \cdot \sigma_{\mathsf{in}}^2 + \sigma_{\mathsf{BR}}^2 + \sigma_{\mathsf{KS}}^2 + \sigma_{\mathsf{MS}}^2 \leq t^2.$$

with $\sigma_{\mathsf{BR}}^2, \sigma_{\mathsf{KS}}^2, \sigma_{\mathsf{MS}}^2$*, the noise after respectively a bootstrapping, a key switch and a modulus switch and where* t *is a noise bound such that* $t = \frac{\Delta}{2 \cdot z^*(p_{\mathsf{fail}})}$ *with the standard score* $z^*(p_{\mathsf{fail}}) = \sqrt{2} \cdot \mathsf{erf}^{-1}(1 - p_{\mathsf{fail}})$ *and the scaling factor* Δ *introduced in Section 2.2.1.*

On line 8, another noise constraint appears.

$$\sigma_{\mathsf{BR}}^2 + \sigma_{\mathsf{KS}}^2 + \sigma_{\mathsf{MS}}^2 \leq t^2.$$

As the left hand term is smaller than the one of the previous inequality, we can discard this constraint. On lines 11 and 12, we have the following constraints

$$2 \cdot \sigma_{\mathsf{BR}}^2 + \sigma_{\mathsf{KS}}^2 + \sigma_{\mathsf{MS}}^2 \leq t^2 \ \ \& \ \ \sigma_{\mathsf{BR}}^2 + \sigma_{\mathsf{KS}}^2 + \sigma_{\mathsf{MS}}^2 \leq t^2.$$

Using the fact that the last constraint is dominated by the others, we can remove it from the set of constraints. In the end of Algorithm 46, the sign $\mathsf{ct}_{\mathfrak{s}}$ *has a noise variance of* $4 \cdot \sigma_{\mathsf{BR}}^2$ *and the each ciphertext in the vector* $\mathsf{ct}_{\mathfrak{z}_{\mathsf{sub}}}$ *has a noise variance* σ_{BR}^2. □

8.2.2 Traditional Floating-Point Representation

Floating-points have become the standard to represent real numbers in computer science, as described in [Ins08]. Their main advantage lies in having a variable precision all along computations, giving more flexibility and accuracy. Usually a floating-point is represented by three values: the sign, the mantissa and the exponent. The most common floating-point encodings on CPU are

the single precision, represented on 32 bits (with 1 bit of sign, 8 bits of exponent and 23 bits of mantissa) and the double precision over 64 bits (1 bit of sign, 11 bits of exponent and 52 bits of mantissa). Less common but still useful encodings are the half-precision which represents 16-bit floats, or some alternative called `Bfloat` [WK19]. Without getting into details, these encodings mainly differ in the distribution of the bit number associated to the mantissa and the exponent. Finally, another family called `MiniFloat` is dedicated to floats whose the size is 8 bits. We refer to [MBDD+18] for more information about floats.

Definition 34 (Floating-Point)**.** *Let* $\mathfrak{b} \in \mathbb{N}$ *such that* $\mathfrak{b} \geq 2$*. Let* $\max_{\mathfrak{e}} \in \mathbb{N}^*$ *and a fixed* $\mathsf{bias} \in [0, \max_{\mathfrak{e}}]$*. A floating-point number* $x \in \mathbb{R}$ *is partially characterized by three values* $(\mathfrak{s}, \mathfrak{m}, \mathfrak{e}) \in \{0;1\} \times \mathbb{N} \times [0, \max_{\mathfrak{e}}]$*, such that:* $x = (-1)^{\mathfrak{s}} \cdot \mathfrak{m} \cdot \mathfrak{b}^{\mathfrak{e}-\mathsf{bias}}$*. With this definition, a floating-point number may have several representations. To obtain a unique representation, the mantissa* $\mathfrak{m}$ *must be in the interval* $[1, \mathfrak{b})$ *except for the value zero where* $\mathfrak{m} = 0$*.*

8.3 Homomorphic Floating-Points (HFP)

In this section, we present two approaches for working with floating-point numbers in TFHE. The first introduces a promising yet precision-limited approach that leverages the **WoP-PBS** (Algorithm 38) to efficiently compute operations on floating-point values. The second approach focuses on translating the traditional floating-point format into a TFHE compatible representation suitable for higher precisions. We then describe the initial building blocks required to perform more advanced operations on these homomorphic floating-point numbers.

8.3.1 MiniFloats: WoP-PBS Based Floats

A powerful approach to define homomorphic floats for TFHE-like schemes relies on the **WoP-PBS**. The method is somewhat similar to the gate bootstrapping approach defined in [CGGI20]: each operation is performed using a **WoP-PBS**.

Minifloat Encoding. Let ρ be the number of bits of precision for a message in an LWE ciphertext, and let $\rho_{\mathfrak{m}}$ (resp., $\rho_{\mathfrak{e}}$) be the number of bits of the mantissa (resp., the exponent). In this first attempt at building TFHE-minifloats, we do not need to have distinct ciphertexts for the mantissa, the sign and the exponent. For instance, we can define an 8-bit floating-point with $\rho = 4$, $\rho_{\mathfrak{m}} = 3$ and $\rho_{\mathfrak{e}} = 4$ using only two LWEs, $\mathsf{ct}_1 \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}||\mathfrak{m}_2||\mathfrak{m}_1||\mathfrak{m}_0)$ and $\mathsf{ct}_2 \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_3||\mathfrak{e}_2||\mathfrak{e}_1||\mathfrak{e}_0)$ where $\mathfrak{m}_i$ (resp., $\mathfrak{e}_i$) corresponds to the i^{th} bit of the mantissa (resp., of the exponent). Each element can be dispatched in any order, but the order must be publicly known to correctly generate the LUT. We call this encoding the *minifloat encoding* and we write it as follows:

$$\text{TFHE-Minifloat}^{\rho}(\rho_{\mathfrak{m}}, \rho_{\mathfrak{e}}, \mathsf{bias}) = \mathsf{Encode}^{\rho}\left(\mathfrak{s}||\mathfrak{m}||\mathfrak{e}, p, q\right).$$

Minifloat Operations. Defining operations over the minifloat encoding is easy: each one of them is computed with a **WoP-PBS** where the LUT associated with the operation is given. The **WoP-PBS** can easily be extended to take many ciphertexts as input in order to compute multivariate operations: the bit extraction step can be done for every input and then a single CMux tree using the bits of every input. Let Op be an operation (e.g., an addition), and $\mathsf{LUT}_{\mathsf{Op}}$ its associated LUT (for more details on how to properly generate the LUT, we refer to Chapter 2, Definition 16). For some $k \in \mathbb{N}$, let $\mathbf{ct}_{\mathfrak{f}_i}$ for $i \in [0, k-1]$ be the input ciphertexts. Then, the output is given by: $\mathbf{ct}_{\mathfrak{f}_{\mathsf{out}}} \leftarrow \textbf{WoP-PBS}\left(\{\mathbf{ct}_{\mathfrak{f}_i}\}_{i=0}^{k-1}, \mathsf{PUB}, \mathsf{LUT}_{\mathsf{Op}}\right)$, see Algorithm 38.

The main advantage is about the complexity of this method, which is not dependent on the functions that have to be computed, i.e., any univariate functions will take the same time (e.g., cosine, logarithm, ...).

Remark 8.1. Some operations do not require a complete **WoP-PBS**. For example, to perform a ReLU, we only need to extract the sign and perform a CMux between input value and a trivial LWE (defined in 4) that encrypts zero.

We provide benchmarks for this method in Section 8.7. This method is very efficient but it is limited in terms of precision. In fact, this method does not work when the combined bit size of all inputs of a **WoP-PBS** exceeds approximately twenty bits, because the number of values that need to be represented is too large and the LUT quickly becomes too big (See Remark 6.6). Next section explores another encoding that can efficiently support large floating-point numbers.

8.3.2 Homomorphic Floating-Point Encoding

As in the traditional representation of floating-point numbers, the homomorphic floating-point representation is divided into three parts: the sign, the mantissa and the exponent.

The **sign** ($\mathfrak{s}$) is encoded by one LWE ciphertext. This ciphertext encrypts the value 0 if the sign is positive or 1 if the sign is negative.

The **mantissa** ($\mathfrak{m}$) is encoded by several LWE ciphertexts (at least 2). Each ciphertext associated with the mantissa encodes the same amount of message bits (denoted $\rho_{\mathfrak{m}}$ in the following). For a mantissa represented by $\ell_{\mathfrak{m}}$ LWE ciphertexts, we can represent integers in $\left[0; 2^{\ell_{\mathfrak{m}} \cdot \rho_{\mathfrak{m}}}\right)$. The ciphertext encrypting the most significant bits (respectively, the least significant bits) of the mantissa is called the most significant (respectively, the least significant) ciphertext. With this representation, we can ensure that the precision of the mantissa is at least $((\ell_{\mathfrak{m}} - 1) \cdot \rho_{\mathfrak{m}} + 1)$-bits. Indeed, the least significant ciphertext will be discarded after some operations, so the information in this block must not be seen as additional precision: when the carry space of the most significant LWE ciphertext is full, a new LWE ciphertext is added as the new most significant block. The less significant block is then removed and the exponent value is increased. In floating point arithmetic, this approach is generally called *rounding towards zero*. This way, the exponent can be smaller and represent a larger range of values. In our representation, to keep a unique encoding for any floating-point, the most significant block should always be different from zero (except for the special value zero where all the blocks are equal to zero). So for any non-zero values, the mantissa is an integer in $\left[2^{\rho_{\mathfrak{m}} \cdot (\ell_{\mathfrak{m}} - 1)}; 2^{\rho_{\mathfrak{m}} \cdot \ell_{\mathfrak{m}}}\right)$.

The **exponent** ($\mathfrak{e}$) is encoded by one or more LWE ciphertexts. Each LWE ciphertext encrypts the same amount of bits $\rho_{\mathfrak{e}}$. The value represented by the exponent is in base $2^{\rho_{\mathfrak{m}}}$ (as already mentioned in the mantissa). So an exponent encrypted in $\ell_{\mathfrak{e}}$ LWE ciphertexts represents values in $\left[0; (2^{\rho_{\mathfrak{m}}})^{2^{\ell_{\mathfrak{e}} \cdot \rho_{\mathfrak{e}}}}\right)$. To represent an exponent that can be negative or positive, the positive value encoded in these $\ell_{\mathfrak{e}}$ LWE ciphertexts needs to be subtracted by a value named bias. When we decode, we obtain, $e \in \left[(2^{\rho_{\mathfrak{m}}})^{-\mathsf{bias}}; (2^{\rho_{\mathfrak{m}}})^{2^{\ell \cdot \rho_{\mathfrak{e}}} - \mathsf{bias}}\right)$. The **encoding of the TFHE floating-point** is illustrated in Figure 8.1, and we refer to it as follows:

$$\mathsf{TFHE\_Fp}(\ell_{\mathfrak{m}}, \rho_{\mathfrak{m}}, \ell_{\mathfrak{e}}, \rho_{\mathfrak{e}}, \mathsf{bias}).$$

Bias The value bias can be set to any value, but to represent a large range of values, in the traditional floating-point, this value is often set to be half of the range of the exponent. With our representation, this value should correspond to $2^{\ell_{\mathfrak{e}} \cdot \rho_{\mathfrak{e}} - 1}$, but in our algorithm we choose to use $\mathsf{bias} = 2^{\ell_{\mathfrak{e}} \cdot \rho_{\mathfrak{e}} - 1} + \ell_{\mathfrak{m}} - 1$. Through this specific value, we gain a speed-up in the homomorphic floating multiplication proposed in Algorithm 54.

Special Values. In the floating-point arithmetic, the subnormal values are the closest values to zero: they are represented by an exponent equal to zero and the leading significant digits equal to zero. In our implementation we choose to not represent these values to have better performance, but our algorithm can be easily modified to take these values into account. In what follows, the leading significant block is always strictly positive except for the zero value. This is the only value

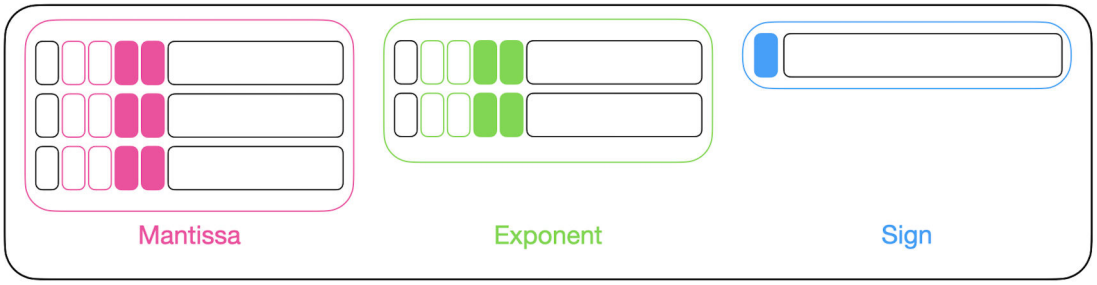

Figure 8.1: The figure illustrates the encoding of a homomorphic floating-point. The Mantissa (in pink) and the Exponent (in green) are split in several ciphertexts which each ciphertext encrypting 4 bits. The fully colored boxes in the figure represent the bits of messages space and the empty colored boxes correspond to the bits of carry space. The sign (in blue) is encoded in only one ciphertext where the information is encrypted on the most significant bit.

represented by each mantissa and exponent blocks equals to zero. Thus, if an operation yields an encrypted float which has its most significant LWE equal to zero, the result will encrypt the zero value (i.e., all the mantissa and exponent LWE will be equal to zero). To keep the algorithms easier to read, other special values like infinities, or NaN are voluntary excluded. Note that the process to support these is detailed in Section 8.6.1.

Encoding and Encrypting. We propose an algorithm to encode and decode real numbers for TFHE with the representation $\mathsf{TFHE\_Fp}(\ell_{\mathfrak{m}}, \rho_{\mathfrak{m}}, \ell_{\mathfrak{e}}, \rho_{\mathfrak{e}}, \mathsf{bias})$. Let $\mathfrak{f}$ be a real number. First, we need to find $\mathfrak{m} \in [0, \ell_{\mathfrak{m}} \cdot \rho_{\mathfrak{m}})$ and $\mathfrak{e} \in [0, \ell_{\mathfrak{e}} \cdot \rho_{\mathfrak{e}})$ such that:

$$\mathfrak{f} = (-1)^{\mathfrak{s}} \cdot \mathfrak{m} \cdot (2^{\rho_{\mathfrak{m}}})^{\mathfrak{e}-\mathsf{bias}}.$$

To obtain a unique representation of these floating-point representations, we impose that the most significant block of the mantissa must be strictly positive (except for the value zero). From this first encoding, we split the mantissa and the exponent according to the $2^{\rho_{\mathfrak{m}}}$ and $2^{\rho_{\mathfrak{e}}}$-radix decompositions, i.e., $\mathbf{m} \leftarrow \mathsf{Encoding}^{2^{\rho_{\mathfrak{m}}}}(\mathfrak{m})$ and $\mathbf{e} \leftarrow \mathsf{Encoding}^{2^{\rho_{\mathfrak{e}}}}(\mathfrak{e})$. The final encoding is given by:

$$\mathsf{Encoding}_{\mathfrak{f}} = \left(\mathfrak{s}, \mathbf{m} = (\mathfrak{m}_{\ell_{\mathfrak{m}}-1}, \ldots, \mathfrak{m}_0)_{\rho_{\mathfrak{m}}}, \mathbf{e} = (\mathfrak{e}_{\ell_{\mathfrak{e}}-1}, \ldots, \mathfrak{e}_0)_{\rho_{\mathfrak{e}}}\right).$$

In absolute value, the maximum value represented by this encoding is $\max = (2^{\ell_{\mathfrak{m}} \cdot \rho_{\mathfrak{m}}} - 1) \cdot (2^{\rho_{\mathfrak{m}}})^{2^{\ell_{\mathfrak{e}} \cdot \rho_{\mathfrak{e}}} - \mathsf{bias} - 1}$. Since the subnormal values are not taken into account, the minimum positive value reached by this encoding (without zero) is $\min = (2^{\ell_{\mathfrak{m}} \cdot \rho_{\mathfrak{m}} - 1}) \cdot (2^{\rho_{\mathfrak{m}}})^{-\mathsf{bias}}$. In Algorithm 47 (resp., Algorithm 48), the method to encrypt (resp., decrypt) a floating-point number is described. The following lemma states the correctness and the notations used to represent homomorphic floats.

Algorithm 47: $\mathbf{ct}_{\mathfrak{f}} \leftarrow \mathsf{EncryptFloat}(\mathfrak{s}, \mathfrak{m}, \mathfrak{e})$

Input: $\mathsf{EncodeFloat}(x) = (\mathfrak{s}, \mathfrak{m}, \mathfrak{e})$
Output: $\mathbf{ct}_{\mathfrak{f}} = [\mathsf{ct}_{\mathfrak{s}}, \mathbf{ct}_{\mathfrak{m}}, \mathbf{ct}_{\mathfrak{e}}]$

$\mathsf{ct}_{\mathfrak{s}} \leftarrow \mathsf{Encrypt}_{\boldsymbol{s}}(\frac{q}{2} \cdot \mathfrak{s}) \in \mathrm{LWE}(\mathfrak{s})$
$\mathbf{ct}_{\mathfrak{m}} = (\mathsf{ct}_{\mathfrak{m},\ell_{\mathfrak{m}}-1}, \ldots, \mathsf{ct}_{\mathfrak{m},0}) \leftarrow \mathsf{Encrypt}_{\boldsymbol{s}}(\mathfrak{m})$
$\mathbf{ct}_{\mathfrak{e}} = (\mathsf{ct}_{\mathfrak{e},\ell_{\mathfrak{e}}-1}, \ldots, \mathsf{ct}_{\mathfrak{e},0}) \leftarrow \mathsf{Encrypt}_{\boldsymbol{s}}(\mathfrak{e})$
return $\mathbf{ct}_{\mathfrak{f}} = [\mathsf{ct}_{\mathfrak{s}}, \mathbf{ct}_{\mathfrak{m}}, \mathbf{ct}_{\mathfrak{e}}]$

Lemma 8.3 (Correctness of DecryptFloat (48))**.** *Let* $\mathfrak{f} = (\mathfrak{s}, \mathfrak{m}, \mathfrak{e}) \in \{0,1\} \times [0, \ell_{\mathfrak{m}} \cdot \rho_{\mathfrak{m}}) \times [0, \ell_{\mathfrak{e}} \cdot \rho_{\mathfrak{e}})$. *Let* $\mathsf{ct}_{\mathfrak{f}} \leftarrow \mathsf{EncryptFloat}(\mathfrak{f})$ *such that* $\mathsf{ct}_{\mathfrak{f}} = [\mathsf{ct}_{\mathfrak{s}}, \mathbf{ct}_{\mathfrak{e}}, \mathbf{ct}_{\mathfrak{m}}]$, *with* $\mathsf{ct}_{\mathfrak{s}} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s})$,

Algorithm 48: $\mathfrak{f} \leftarrow \mathsf{DecryptFloat}(\mathsf{ct}_{\mathfrak{f}})$

Input: $\mathbf{ct}_{\mathfrak{f}} = [\mathsf{ct}_{\mathfrak{s}}, \mathbf{ct}_{\mathfrak{m}}, \mathbf{ct}_{\mathfrak{e}}]$
Output: $\mathfrak{f} = (-1)^{\mathfrak{s}} \cdot \mathfrak{m} \cdot (2^{\rho_{\mathfrak{m}}})^{\mathfrak{e}-\mathsf{bias}} \in \mathbb{R}$

1 **If** $\mathsf{Decrypt}_{\boldsymbol{s}}(\mathsf{ct}_{\mathfrak{s}}) = 0$ **then** $\mathfrak{s} = 1$, **Else** $\mathfrak{s} = -1$
2 $\mathfrak{m} \leftarrow \mathsf{Decrypt}_{\boldsymbol{s}}(\mathsf{ct}_{\mathfrak{m}})$
3 $\mathfrak{e} \leftarrow \mathsf{Decrypt}_{\boldsymbol{s}}(\mathsf{ct}_{\mathfrak{e}})$
4 **return** $\mathfrak{f} \leftarrow (-1)^{\mathfrak{s}} \cdot \mathfrak{m} \cdot (2^{\rho_{\mathfrak{m}}})^{\mathfrak{e}-\mathsf{bias}}$

$\mathbf{ct}_{\mathfrak{e}} = [\mathsf{ct}_{\mathfrak{e},\ell_{\mathfrak{e}}-1}, \ldots, \mathsf{ct}_{\mathfrak{e},0}] \in \left[\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_{1_{\ell_{\mathfrak{e}}-1}}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_{1_0}\right)\right]$ *and* $\mathbf{ct}_{\mathfrak{m}} = [\mathsf{ct}_{\mathfrak{m},\ell_{\mathfrak{m}}-1}, \ldots, \mathsf{ct}_{\mathfrak{m},0}] \in \left[\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{1_{\ell_{\mathfrak{m}}-1}}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{1_0}\right)\right]$. *Then,* $\mathsf{DecryptFloat}_{\boldsymbol{s}}(\mathsf{ct}_{\mathfrak{f}}) = \mathfrak{f}$.

Trivial Encrypt. A trivial encryption is an LWE ciphertext where all the a_i are equal zero (see Remark 2.6). This is trivially extendable to the floats. This is denoted $\mathsf{TrivialEncryptFloat}(\mathfrak{f})$. Sometimes, we only need to trivially encrypt a part of a floating-point: which is suggested by the notation $\mathsf{TrivialEncrypt}(\mathsf{Value}, \mathsf{NumberOfBlocks})$. For instance, to encrypt the value $\mathfrak{v}$ as an exponent, we note $\mathsf{TrivialEncrypt}(\mathfrak{v}, \ell_{\mathfrak{e}})$.

Definition 35 (Maximum error after operation)**.** *In the context of floating-point, due to the encoding, after each operation a small error could be introduced (this error is not tied to the TFHE noise). This added error is denoted ϵ. To find the errors added after each operation by our encoding, we look at the maximal error added in the mantissa and we multiply this error by the exponent. For a floating-point $\mathfrak{f} = (-1)^{\mathfrak{s}} \cdot \mathfrak{m} \cdot (2^{\rho_{\mathfrak{m}}})^{\mathfrak{e}-\mathsf{bias}}$, the error, after an operation, can be bounded by $\mathsf{error}_{\mathfrak{m}} \cdot (2^{\rho_{\mathfrak{m}}})^{\mathfrak{e}-\mathsf{bias}}$. As we do not represent the subnormal values, if the $\mathfrak{e}$ is equal to 0, the error is bounded by $2^{\rho_{\mathfrak{m}} \cdot (\ell_{\mathfrak{m}}-1)} \cdot (2^{\rho_{\mathfrak{m}}})^{-\mathsf{bias}}$.*

Example: Encoding a 64 bits Floating-Point. In the [Ins08] standard, a floating-point is composed of 1 bit of sign, 11 bits of exponent and 52 bits plus one hidden bit of mantissa. So as mentioned in the beginning of Section 8.3, to ensure a precision of at least 53 bits, we need to have $(\ell_{\mathfrak{m}} - 1) \cdot \rho_{\mathfrak{m}} + 1 \geq 53$, i.e., one additional block to perform operations without losing the precision of 53 bits. For the mantissa, we choose $\ell_{\mathfrak{m}} = 27$ with $\rho_{\mathfrak{m}} = 2$. In [Ins08], the exponent value $\mathfrak{e}$ belongs to $[-1023, 1024)$. To simplify the implementation, we prefer to have $\rho_{\mathfrak{e}} = \rho_{\mathfrak{m}}$, and a bias equal to $2^{\ell_{\mathfrak{e}} \cdot \rho_{\mathfrak{e}} - 1} + \ell_{\mathfrak{m}} - 1$ with $\ell_{\mathfrak{e}} = 5$. Thus, this yields in a floating-point exponent in $\left[-\mathsf{bias} \cdot \rho_{\mathfrak{m}}, (2^{\ell_{\mathfrak{e}} \cdot \rho_{\mathfrak{e}}} - \mathsf{bias} - 1) \cdot \rho_{\mathfrak{m}}\right] = [-1076, 970]$ with $\mathsf{bias} = 538$. This allows representing as many values as the standard one, but with a different scale: the upper bound is lower, but more precision is given near values close to zero. These parameters correspond to the representation:

$$\mathsf{TFHE\_Fp_{64b}}(\ell_{\mathfrak{m}} = 27, \rho_{\mathfrak{m}} = 2, \ell_{\mathfrak{e}} = 5, \rho_{\mathfrak{e}} = 2, \mathsf{bias} = 538).$$

More encoding examples for 32-bits, 16-bits and 8-bits are given in Table 8.4 (Section 8.7).

8.3.3 Choosing Between Two Ciphertexts

In what follows, we extensively use Algorithm 49 to homomorphically make a choice between two LWE ciphertext lists depending on an encrypted bit. This algorithm is an extended version of the CMux described in [CGGI20, Lemma 3.16]. The selector is a GGSW ciphertext, and the choice is done between two lists of LWE ciphertexts.

Lemma 8.4. *Let* $\mathbf{ct}_i = [\mathsf{ct}_{\mathfrak{z}i,\ell_{\mathfrak{z}}-1}, \ldots, \mathsf{ct}_{\mathfrak{z}i,0}] \in \left[\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{z}_{i,\ell_{\mathfrak{z}}-1}), \ldots \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{z}_{i,0})\right] \subseteq \mathbb{Z}_q^{(n+1)\cdot\ell_{\mathfrak{z}}}$ *with* $i \in \{0, 1\}$ *be two ciphertexts encrypting* $\mathfrak{z}_i \in \mathbb{N}$. *Let* $\overline{\overline{\mathsf{CT}}}_{\mathsf{Sel}} \in \mathrm{GGSW}_{\boldsymbol{S}}^{\mathcal{B},\ell}(b)$ *(with* $b \in \{0, 1\}$*).*

Then Algorithm 49 returns $\mathbf{ct}_{\mathsf{res}}$ *with* $\mathsf{Decrypt}_{\boldsymbol{s}}(\mathbf{ct}_{\mathsf{res}}) = \mathfrak{z}_{\mathfrak{b}}$.

Algorithm 49: $\mathbf{ct}_{\mathsf{res}} \leftarrow \mathsf{ExtendedCMux}(\mathbf{ct}_0, \mathbf{ct}_1, \overline{\overline{\mathsf{CT}}}_{\mathsf{Sel}})$

Input:
- $\mathbf{ct}_0 = [\mathsf{ct}_{0,\ell_\mathfrak{z}-1}, \ldots, \mathsf{ct}_{0,0}] \in \mathbb{Z}_q^{(n+1)\cdot\ell_\mathfrak{z}}$ with $\mathsf{ct}_{i,j} \in \mathrm{LWE}_{\boldsymbol{s}}(m_{i,j})$
- $\mathbf{ct}_1 = [\mathsf{ct}_{1,\ell_\mathfrak{z}-1}, \ldots, \mathsf{ct}_{1,0}] \in \mathbb{Z}_q^{(n+1)\cdot\ell_\mathfrak{z}}$
- $\overline{\overline{\mathsf{CT}}}_{\mathsf{Sel}} \in \mathrm{GGSW}_{\boldsymbol{S}}^{\mathcal{R},\ell}(b)$ (with $b \in \{0,1\}$)

Output:
- $\mathbf{ct}_{\mathsf{res}} = [\mathsf{ct}_{b,\ell_\mathfrak{z}-1}, \ldots, \mathsf{ct}_{b,0}] \in \mathbb{Z}_q^{(n+1)\cdot\ell_\mathfrak{z}}$

1 **for** $i \in [0..\ell_\mathfrak{z})$ **do**
2 $\quad \mathsf{CT}_{0,i} \leftarrow \mathsf{ConstantSampleExtract}^{-1}(\mathsf{ct}_{0,i})$; /* Algorithm 29 */
3 $\quad \mathsf{CT}_{1,i} \leftarrow \mathsf{ConstantSampleExtract}^{-1}(\mathsf{ct}_{1,i})$; /* Algorithm 29 */
4 $\quad \mathsf{CT}_{\mathsf{res},i} \leftarrow \mathsf{CMux}(\mathsf{CT}_{0,i}, \mathsf{CT}_{1,i}, \overline{\overline{\mathsf{CT}}}_{\mathsf{Sel}})$; /* Algorithm 8 */
5 $\quad \mathsf{ct}_{\mathsf{res},i} \leftarrow \mathsf{Sample\_extract}(\mathsf{CT}_{\mathsf{res},i})$; /* Algorithm 28 */
6 **return** $\mathbf{ct}_{\mathsf{res}} = [\mathsf{ct}_{\mathsf{res},\ell_\mathfrak{z}-1}, \ldots, \mathsf{ct}_{\mathsf{res},0}]$

Proof *(Correctness of Algorithm 49). Let us execute each instruction of Algorithm 49. In Lines 2 and 3, both* LWE *ciphertexts are transformed into* GLWE *ciphertexts with the message on the first coefficient and random messages on all the other coefficients, i.e.,* $\mathsf{CT}(M_{j,i}) \in \mathrm{GLWE}_{\boldsymbol{S}}(M_{j,i})$ *with* $M_{j,i} = m_{j,i} + \sum_{\alpha=1}^{N-1} r_{j,i,\alpha} X^\alpha$*, for some* $r_{i,j,\alpha} \in \mathbb{Z}_q$ *with* $j \in \{0,1\}$*. Next, in Line 4:* $\mathsf{CT}_{res,i}$ *is the result of the* CMux*, i.e.,* $\mathsf{CT}_{res,i}(M_{b,i}) \leftarrow \mathsf{CMux}\left(\mathsf{CT}(M_{1,i}), \mathsf{CT}(M_{0,i}), \overline{\overline{\mathsf{CT}}}_{\mathsf{Sel}}(b)\right)$*, (as defined in Algorithm 8), with* $b \in \{0,1\}$*. At Line 5, we extract the first coefficient of* $\mathsf{CT}_{m_{res},i}$*. The result is then* $\mathsf{ct}(m_{res,i}) \in \mathrm{LWE}_{\boldsymbol{s}}(m_{b,i})$*.* □

Proof *(*CMux *Noise Analysis). We adapt proof of Theorem 2.11 for an external product using a binary secret in the GGSW ciphertext. The only difference is that the GGSW ciphertext is obtained through a* **CBS** *rather than a freshly encrypted ciphertext, so we use the same noise formula. However, the noise of the bootstrapping key is defined as the output noise of circuit bootstrapping instead of fresh encryption.* □

8.3.4 Propagating the Carries

Since HFP are based on radix-based homomorphic integers, the need to propagate the carry must be considered to ensure correctness. Indeed, in each block, carry might accumulate all along computation up to point where the carry space is full. Differently from modular integer computation, where the modulus is generally a power of two, we cannot simply remove the carry from the most significant block. In our case, when the mantissa has a carry that has been propagated to the most significant block, a new one must be created and the last one can be removed. In Algorithm 50, we describe the process to perform this homomorphically. It takes as input a ciphertext encrypting a floating-point, and returns another ciphertext where the carries have been propagated.

Lemma 8.5. *Let* $\mathbf{ct}_\mathfrak{f} = [\mathsf{ct}_\mathfrak{s}, \mathbf{ct}_\mathfrak{m}, \mathbf{ct}_\mathfrak{e}]$ *encrypting* $\mathfrak{f} = (-1)^\mathfrak{s} \cdot \mathfrak{m} \cdot 2^{\rho_\mathfrak{m}^{\mathfrak{e}-\mathsf{bias}}}$*, with* $\mathsf{ct}_\mathfrak{s} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}), \mathbf{ct}_\mathfrak{e} = [\mathsf{ct}_{\mathfrak{e},\ell_\mathfrak{e}-1}, \ldots, \mathsf{ct}_{\mathfrak{e},0}] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{\ell_\mathfrak{e}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_0)]$ *and* $\mathbf{ct}_\mathfrak{m} = [\mathsf{ct}_{\mathfrak{m},\ell_\mathfrak{m}-1}, \ldots, \mathsf{ct}_{\mathfrak{m},0}] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{\ell_\mathfrak{m}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_0)]$ *such that some* $\mathsf{ct}_{\mathfrak{m},i}$ *(resp.,* $\mathsf{ct}_{\mathfrak{e},i}$*) encrypting some* $\mathfrak{m}_i > 2^{\rho_\mathfrak{m}}$ *(resp.,* $\mathfrak{e}_i > 2^{\rho_\mathfrak{e}}$*).*

Then, Algorithm 50 outputs $\mathbf{ct}_{\mathfrak{f}_{\mathsf{out}}}$ *such that* $\mathsf{DecryptFloat}_{\boldsymbol{s}}(\mathbf{ct}_{\mathfrak{f}_{\mathsf{out}}}) = (-1)^{\mathfrak{s}_{\mathsf{out}}} \cdot \mathfrak{m}_{\mathsf{out}} \cdot 2^{\rho_\mathfrak{m}^{\mathfrak{e}_{\mathsf{out}}-\mathsf{bias}}}$ *with* $\mathfrak{m}_{\mathsf{out}} \in \left[2^{\rho_\mathfrak{m}\cdot(\ell_\mathfrak{m}-1)}, 2^{\rho_\mathfrak{m}\cdot\ell_\mathfrak{m}}\right]$ *if* $\mathfrak{m} \in \left[2^{\rho_\mathfrak{m}\cdot(\ell_\mathfrak{m}-1)}, 2^{\rho_\mathfrak{m}\cdot\ell_\mathfrak{m}}\right]$*, otherwise* $\mathfrak{m}_{\mathsf{out}} \in \left[2^{\rho_\mathfrak{m}\cdot(\ell_\mathfrak{m}-1)}, 2^{\rho_\mathfrak{m}\cdot\ell_\mathfrak{m}}\right]$ *and* $\mathfrak{e}_{\mathsf{out}} = \mathfrak{e} + 1$*. Moreover, for all* $i \in [0, \ell_\mathfrak{m} - 1]$ *(resp.,* $i \in [0, \ell_\mathfrak{e} - 1]$*),* $\mathfrak{m}_{\mathsf{out},i} \in [0, 2^{\rho_\mathfrak{m}} - 1]$ *(resp.,* $\mathfrak{e}_{\mathsf{out},i} \in [0, 2^{\rho_\mathfrak{e}} - 1]$*).*

Proof *(Correctness of* CarryPropagateFloat *(Algorithm 50)). In Line 2, we get* $\mathbf{ct}_{\mathfrak{e}'} = [\mathsf{ct}_{\mathfrak{e}',\ell_\mathfrak{e}-1}, \ldots, \mathsf{ct}_{\mathfrak{e}',0}]$ *such that* $\forall i \in [0, \ell_\mathfrak{e} - 1], \mathsf{Decrypt}(\mathsf{ct}_{\mathfrak{e}',i}) = \mathfrak{e}'_i < 2^{\rho_\mathfrak{e}}$ *after the carry propagation. Likewise, in Line 3, we do an extended carry propagation of the mantissa, so that* $\mathbf{ct}_{\mathfrak{m}'} = [\mathsf{ct}_{\mathfrak{m}',\ell_\mathfrak{m}}, \ldots, \mathsf{ct}_{\mathfrak{m}',0}]$ *such that,* $\forall i \in [0, \ell_\mathfrak{m}], \mathsf{Decrypt}(\mathsf{ct}_{\mathfrak{m}',i}) = \mathfrak{m}'_i < 2^{\rho_\mathfrak{m}}$*. Note that the carry on the most significant block is not lost and creates a new* LWE *ciphertext* $\mathsf{ct}_{\mathfrak{m}',\ell_\mathfrak{m}}$ *encrypting the propagated carry of* $\mathsf{ct}_{\mathfrak{m},\ell_\mathfrak{m}-1}$*.*

Algorithm 50: $\mathbf{ct}_{\mathfrak{f}_{\mathsf{out}}} \leftarrow \mathsf{CarryPropagateFloat}\left(\mathbf{ct}_{\mathfrak{f}}\right)$

Context: $\mathsf{LUT}_{\mathsf{id}}$: Lookup Table associated to the id function.

Input: $\begin{cases} \mathbf{ct}_{\mathfrak{f}} = \left[\begin{array}{l} \mathsf{ct}_{\mathfrak{s}} \in \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{s}\right) \\ \mathbf{ct}_{\mathfrak{e}} = [\mathsf{ct}_{\mathfrak{e},\ell_{\mathfrak{e}}-1}, \ldots, \mathsf{ct}_{\mathfrak{e},0}] \in [\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_{\ell_{\mathfrak{e}}-1}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_0\right)] \\ \mathbf{ct}_{\mathfrak{m}} = [\mathsf{ct}_{\mathfrak{m},\ell_{\mathfrak{m}}-1}, \ldots, \mathsf{ct}_{\mathfrak{m},0}] \in [\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{\ell_{\mathfrak{m}}-1}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_0\right)] \end{array}\right. \\ \mathsf{PUB} : \text{Public keys for } \mathbf{KS}, \mathbf{PBS} \text{ and } \mathbf{CBS}; \texttt{/* Remark 2.15 */} \end{cases}$

Output: $\begin{cases} \mathbf{ct}_{\mathfrak{f}_{\mathsf{out}}} = \left[\begin{array}{l} \mathsf{ct}_{\mathfrak{s}_{\mathsf{out}}} \in \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{s}\right) \\ \mathbf{ct}_{\mathfrak{e}_{\mathsf{out}}} = [\mathsf{ct}_{\mathfrak{e}_{\mathsf{out}},\ell_{\mathfrak{e}}-1}, \ldots, \mathsf{ct}_{\mathfrak{e}_{\mathsf{out}},0}] \in [\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_{\mathsf{out},\ell_{\mathfrak{e}}-1,}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_{\mathsf{out},0}\right)] \\ \mathbf{ct}_{\mathfrak{m}_{\mathsf{out}}} = [\mathsf{ct}_{\mathfrak{m}_{\mathsf{out}},\ell_{\mathfrak{m}}-1}, \ldots, \mathsf{ct}_{\mathfrak{m}_{\mathsf{out}},0}] \in [\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{\mathsf{out},\ell_{\mathfrak{m}}-1}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{\mathsf{out},0}\right)] \end{array}\right. \end{cases}$

1 $\mathsf{ct}_{\mathfrak{s}_{\mathsf{out}}} \leftarrow \mathbf{KS}\text{-}\mathbf{PBS}\left(\mathsf{ct}_{\mathfrak{s}}, \mathsf{LUT}_{\mathsf{Id}}, \mathsf{PUB}\right)$; `/* Algorithm 4 and 11 */`

2 $\mathbf{ct}_{\mathfrak{e}'} \leftarrow \mathsf{CarryPropagate}\left(\mathbf{ct}_{\mathfrak{e}}, \mathsf{PUB}\right)$; `/* Algorithm 39 */`

3 $\mathbf{ct}_{\mathfrak{m}'} = [\mathsf{ct}_{\mathfrak{m}',\ell_{\mathfrak{m}}}, \ldots, \mathsf{ct}_{\mathfrak{m}',0}] \leftarrow \mathsf{CarryPropagateExtend}\left(\mathbf{ct}_{\mathfrak{m}}, \mathsf{PUB}\right)$; `/* Algorithm 45 */`

4 $\mathbf{ct}_{\mathfrak{m}_+} = [\mathsf{ct}_{\mathfrak{m}',\ell_{\mathfrak{m}}}, \ldots, \mathsf{ct}_{\mathfrak{m}',1}]$, $\mathbf{ct}_{\mathfrak{m}_-} = [\mathsf{ct}_{\mathfrak{m}',\ell_{\mathfrak{m}}-1}, \ldots, \mathsf{ct}_{\mathfrak{m}',0}]$

5 $\overline{\overline{\mathsf{CT}}} \leftarrow \mathbf{CBS}\left(\mathsf{ct}_{\mathfrak{m}'_{\ell_{\mathfrak{m}}}}, \mathsf{PUB}\right)$; `/* Algorithm 12 */`

6 $\mathbf{ct}_{\mathfrak{m}_{\mathsf{out}}} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}_{\mathfrak{m}_-}, \mathbf{ct}_{\mathfrak{m}_+}, \overline{\overline{\mathsf{CT}}}\right)$; `/* Algorithm 49 */`

7 $\mathbf{ct}_{\mathfrak{e}_{\mathsf{out}}} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}_{\mathfrak{e}'}, \mathbf{ct}_{\mathfrak{e}'} + \mathsf{TrivialEncrypt}(1, \ell_{\mathfrak{e}}), \overline{\overline{\mathsf{CT}}}\right)$; `/* Algorithm 49 */`

8 **return** $\mathbf{ct}_{\mathfrak{f}_{\mathsf{out}}} = (\mathsf{ct}_{\mathfrak{s}_{\mathsf{out}}}; \mathbf{ct}_{\mathfrak{e}_{\mathsf{out}}}; \mathbf{ct}_{\mathfrak{m}_{\mathsf{out}}})$

In the next steps, the idea is to decide if we need to keep this block $\mathsf{ct}_{\mathfrak{m}',\ell_{\mathfrak{m}}}$ *and remove the least significant block (i.e., to return* $\mathbf{ct}_{\mathfrak{m}_+}$*) or if we can discard it (i.e., to return* $\mathbf{ct}_{\mathfrak{m}_-}$*). This allows us to output a result which has the same number of blocks* $\ell_{\mathfrak{m}}$ *than the input. To do so, in Line 6 we perform a circuit bootstrapping returning a* GGSW *ciphertext:* $\overline{\overline{\mathsf{CT}}} \in \mathrm{GGSW}_{\boldsymbol{S}}^{\mathcal{B},\ell}(0)$ *if the new* $\mathsf{ct}_{m',\ell_{\mathfrak{m}}}$ *is in* $\mathrm{LWE}_{\boldsymbol{s}}(0)$*, otherwise,* $\overline{\overline{\mathsf{CT}}}$ *is in* $\mathrm{GGSW}_{\boldsymbol{S}}^{\mathcal{B},\ell}(1)$*. Next in Line 7, Algorithm 49 returns* $\mathbf{ct}_{\mathfrak{m}_+}$ *if* $\overline{\overline{\mathsf{CT}}}$ *is in* $\mathrm{GGSW}_{\boldsymbol{S}}^{\mathcal{B},\ell}(1)$*, or* $\mathbf{ct}_{\mathfrak{m}_-}$ *otherwise. Likewise, in the case where* $\mathsf{ct}_{\mathfrak{m}',\ell_{\mathfrak{m}}}$ *does not encrypt* 0*, the exponent should be updated. The condition is then the same as previously, so that we can use the same selector* $\overline{\overline{\mathsf{CT}}}$ *to choose between the initial value of the exponent* $\mathbf{ct}_{\mathfrak{e}'}$ *or the one which has been increased by one* $\mathbf{ct}_{\mathfrak{e}'} + \mathsf{TrivialEncrypt}(1, \ell_{\mathfrak{e}})$*.* □

Remark 8.2 (Carry Propagation & Refresh)**.** After most operations, we will apply Algorithm 50 to properly propagate the carries and refresh the noise. However, after operations like the ReLU (Algorithm 56) or the approximated Sigmoid (Algorithm 57) that do not fill the carry block, we only need to perform a **PBS** on each ciphertext to obtain fresh noise.

Lemma 8.6 (Noise Constraints of Algorithm 50)**.** *The output ciphertexts of Algorithm 50,* $\mathbf{ct}_{\mathfrak{m}_{\mathsf{out}}}$ *has a noise variance* $\sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{CMux}}$*,* $\mathbf{ct}_{\mathfrak{e}_{\mathsf{out}}}$ *has a noise variance* $\sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{CMux}}$ *and* $\mathsf{ct}_{\mathfrak{s}_{\mathsf{out}}}$ *has a noise variance* σ^2_{BR}*.*

To guarantee correctness of Algorithm 50, we need to find parameters that verify the following inequalities:

$$\sigma^2_{\mathsf{in},\mathfrak{e}} + \sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{KS}} + \sigma^2_{\mathsf{MS}} \leq t^2 \;\; \& \;\; \sigma^2_{\mathsf{in},\mathfrak{m}} + \sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{KS}} + \sigma^2_{\mathsf{MS}} \leq t^2.$$

With $\sigma_{\mathsf{in},\mathfrak{e}}$ *the noise variance of the input exponent ciphertexts,* $\sigma_{\mathsf{in},\mathfrak{m}}$ *the noise variance of the input mantissa ciphertexts and with* σ_{BR} *the noise added by the blind rotation,* σ_{KS} *the noise added by the key switch,* σ_{CMux} *the noise added by the* CMux *and finally* σ_{MS}*, the noise added by the modulus switch and with* t^2*, the noise bound as defined in the proof of the noise constraints of Algorithm 46.*

Proof *(Lemma 8.6). Let us assume that the input ciphertexts* $\mathsf{ct}_{\mathfrak{s}}, \mathbf{ct}_{\mathfrak{e}}$ *and* $\mathbf{ct}_{\mathfrak{m}}$ *have respectively the following noise variances* $\sigma^2_{\mathsf{in},\mathfrak{s}}, \sigma^2_{\mathsf{in},\mathfrak{e}}$ *and* $\sigma^2_{\mathsf{in},\mathfrak{m}}$*. The first line of the algorithm consists in a key switch and a bootstrapping. We have the following noise constraint:* $\sigma^2_{\mathsf{in},\mathfrak{s}} + \sigma^2_{\mathsf{KS}} + \sigma^2_{\mathsf{MS}} \leq t^2$*, with* σ^2_{KS} *and* σ^2_{MS} *the noise added respectively by the key switch and the modulus switch.* t *is a noise*

bound such that $t = \frac{\Delta}{2 \cdot z^*(p_{\mathsf{fail}})}$ *with the standard score* $z^*(p_{\mathsf{fail}}) = \sqrt{2} \cdot \mathsf{erf}^{-1}(1 - p_{\mathsf{fail}})$ *and the scaling factor* Δ *introduced in Section 2.2.1. If we find parameters that guarantee the inequality above, the bootstrapping will be successful with probability* $1 - p_{\mathsf{fail}}$.

Then, we have a carry propagate and an extended carry propagate. We refer to the analysis for Algorithm 46 for the explanation about the constraints in these algorithms:

$$\sigma^2_{\mathsf{in},\mathfrak{e}} + \sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{KS}} + \sigma^2_{\mathsf{MS}} \leq t^2 \ \ \& \ \ \sigma^2_{\mathsf{in},\mathfrak{m}} + \sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{KS}} + \sigma^2_{\mathsf{MS}} \leq t^2.$$

Finally, we have a circuit bootstrapping that also creates a noise constraint $\sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{KS}} + \sigma^2_{\mathsf{MS}} \leq t^2$. *Notice that the left-hand side here is smaller than in both inequalities above, so we can remove this last inequality from the set of constraints. Then, the output ciphertexts* $\mathbf{ct}_{\mathfrak{m}_{\mathsf{out}}}, \mathbf{ct}_{\mathfrak{e}_{\mathsf{out}}}$ *and* $\mathsf{ct}_{\mathfrak{s}_{\mathsf{out}}}$ *have respectively a noise variance* $\sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{cmux}}, \sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{cmux}}$ *and* σ^2_{BR} *with* σ^2_{cmux} *the variance added by an extended* CMux *using a* GGSW *coming from a circuit bootstrapping.* □

8.4 Addition and Subtraction of HFP

In this section, we detail the algorithms used to perform addition and subtraction operations with our floating-point representation. Initially, we describe the operations that manage the mantissa: the first aligns the two mantissas, and the second carries out the subtraction between them, followed by a realignment of the resulting value. Ultimately, the application of these algorithms enables us to efficiently implement homomorphic floating-point (HFP) addition and subtraction operations.

8.4.1 Managing Mantissas and Exponents

To add two floating-point numbers, we can not directly add their mantissas. First, we need their exponents to be equal and the mantissas to be aligned properly. In what follows, we describe the algorithms to homomorphically perform these operations.

Aligned Mantissa Algorithm. Algorithm 51 takes as input ciphertexts encrypting two mantissas and their corresponding exponents, and returns the largest exponent $\mathfrak{e}_{\max}$ along with both aligned mantissas. The first step of this operation is to perform a subtraction between the two exponents to obtain $d = |\mathfrak{e}_1 - \mathfrak{e}_2|$ and the sign of this difference. The sign then allows us to select the largest exponent and the mantissa that needs to be aligned. Finally, a tree of CMux, using the bits of d, aligns the selected mantissa by removing the d least significant ciphertexts from the mantissa associated with $\mathfrak{e}_{\min}$. All the steps of this operation are illustrated in Figure 8.2.

Lemma 8.7 (Aligned mantissa (Algorithm 51))**.** *Let* $\mathbf{ct}_{\mathfrak{m}_i}$ *and* $\mathbf{ct}_{\mathfrak{e}_i}$ *such that* $\mathbf{ct}_{\mathfrak{e}_i} = \left[\mathsf{ct}_{\mathfrak{e}_{i,\ell_\mathfrak{e}-1}}, \ldots, \mathsf{ct}_{\mathfrak{e}_{i,0}}\right] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{i,\ell_\mathfrak{e}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{i,0})]$ *and* $\mathbf{ct}_{\mathfrak{m}_i} = \left[\mathsf{ct}_{\mathfrak{m}_{i,\ell_\mathfrak{m}-1}}, \ldots, \mathsf{ct}_{\mathfrak{m}_{i,0}}\right] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{i,\ell_\mathfrak{m}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{i,0})]$ *with* $i \in \{1, 2\}$ *be two ciphertexts encrypting* $\mathfrak{m}_i \cdot (2^{\rho_\mathfrak{m}})^{\mathfrak{e}_i - \mathsf{bias}}$.

Then, Algorithm 51 returns $(\mathbf{ct}_{\mathfrak{m}'_{1_{\mathsf{res}}}}, \mathbf{ct}_{\mathfrak{m}'_{2_{\mathsf{res}}}}, \mathbf{ct}_{\mathfrak{e}_{\max}})$ *with,* $\mathfrak{e}_{\max} = \max(\mathfrak{e}_1, \mathfrak{e}_2)$. *If* $\mathfrak{e}_1 > \mathfrak{e}_2$, $\mathfrak{m}'_{1_{\mathsf{res}}} = \mathfrak{m}_1$, *then* $\mathfrak{m}'_{2_{\mathsf{res}}} = \left\lfloor \mathfrak{m}_2 / 2^{\rho_\mathfrak{m} \cdot d} \right\rfloor$ *with* $d = \mathfrak{e}_1 - \mathfrak{e}_2$. *Else if* $\mathfrak{e}_1 < \mathfrak{e}_2$, $\mathfrak{m}'_{2_{\mathsf{res}}} = \mathfrak{m}_2$, *then* $\mathfrak{m}'_{1_{\mathsf{res}}} = \left\lfloor \mathfrak{m}_1 / 2^{\rho_\mathfrak{m} \cdot d} \right\rfloor$ *with* $d = \mathfrak{e}_2 - \mathfrak{e}_1$. *Else if* $\mathfrak{e}_1 = \mathfrak{e}_2$, *then* $\mathfrak{m}'_{1_{\mathsf{res}}} = \mathfrak{m}_1$ *and* $\mathfrak{m}'_{2_{\mathsf{res}}} = \mathfrak{m}_2$.
The complexity of the algorithm is:

$$\mathsf{Cost}^{\ell_{\mathbf{PBS}}, \ell_{\mathbf{CBS}}, k, N, n, q, \ell_\mathfrak{e}, \rho_\mathfrak{e}}_{\mathsf{AlignMantissa}} = (\ell_\mathfrak{e} \cdot \rho_\mathfrak{e} + 1) \cdot \mathsf{Cost}^{\ell_{\mathbf{PBS}}, \ell_{\mathbf{CBS}}, k, N, n, q}_{\mathbf{CBS}} + \mathsf{Cost}^{\ell_\mathfrak{e}, \ell_{\mathbf{PBS}}, k, N, n, q}_{\mathsf{RadixSub}^*}$$

Proof *(Correctness of* AlignMantissa *(Algorithm 51)). In what follows, we use the index of a ciphertext to refer to the plaintext value it encrypts, i.e.,* $\mathfrak{z} = \mathsf{Decrypt}_s(\mathsf{ct}_\mathfrak{z})$. *From Algorithm 46, we have that* $d = |\mathfrak{e}_1 - \mathfrak{e}_2|$ *and* $\mathfrak{s} = 0$ *if* $\mathfrak{e}_1 \geq \mathfrak{e}_2$, 1 *otherwise. Each bits of* $\mathsf{ct}_\mathfrak{s}$ *and* $\mathbf{ct}_d$ *are converted to* GGSW *via a* **CBS**, *so that after Line 5, we have:* $\left\{\overline{\overline{\mathsf{CT}}}_{d_i} \in \mathrm{GGSW}_{\boldsymbol{S}}(d_i)\right\}_{i \in [0, \ell_\mathfrak{e} \cdot \rho_\mathfrak{e} - 1]}$ *(such that* $d = \sum_{i=0}^{\ell_\mathfrak{e} \cdot \rho_\mathfrak{e}} d_i 2^i$*) and* $\overline{\overline{\mathsf{CT}}}_\mathfrak{s} \in \mathrm{GGSW}_{\boldsymbol{S}}(d_\mathfrak{s})$. *From Algorithm 49, after Line 6, we get* $\mathfrak{m}^{\langle 0 \rangle}_{\mathsf{out}} = \mathfrak{m}_2$ *if* $d_\mathfrak{s} = 0$, $\mathfrak{m}_1$ *otherwise. This gives which of the mantissa needs to be aligned, i.e., if the ciphertext* $\overline{\overline{\mathsf{CT}}}_{d_\mathfrak{s}}$ *is in* $\mathrm{GGSW}^{\mathcal{B},\ell}_{\boldsymbol{S}}(0)$, $\mathfrak{e}_1 \geq \mathfrak{e}_2$ *so we need to align* $\mathfrak{m}_2$, *otherwise* $\mathfrak{e}_1 < \mathfrak{e}_2$, *and thus* $\mathfrak{m}_1$ *needs*

Algorithm 51: $\left(\mathbf{ct}_{m_{1'}}, \mathbf{ct}_{m_{2'}}, \mathbf{ct}_e\right) \leftarrow \mathsf{AlignMantissa}(\mathbf{ct}_{\mathfrak{e}_1}, \mathbf{ct}_{\mathfrak{m}_1}, \mathbf{ct}_{\mathfrak{e}_2}, \mathbf{ct}_{\mathfrak{m}_2}, \mathsf{PUB})$

Input:
- $\mathbf{ct}_{\mathfrak{e}_1} = [\mathsf{ct}_{\mathfrak{e}_1,\ell_\mathfrak{e}-1}, \ldots, \mathsf{ct}_{\mathfrak{e}_1,0}] \in \left[\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_{1_{\ell_\mathfrak{e}-1}}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_{1_0}\right)\right]$
- $\mathbf{ct}_{\mathfrak{m}_1} = [\mathsf{ct}_{\mathfrak{m}_1,\ell_\mathfrak{m}-1}, \ldots, \mathsf{ct}_{\mathfrak{m}_1,0}] \in \left[\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{1_{\ell_\mathfrak{m}-1}}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{1_0}\right)\right]$
- $\mathbf{ct}_{\mathfrak{e}_2} = [\mathsf{ct}_{\mathfrak{e}_2,\ell_\mathfrak{e}-1}, \ldots, \mathsf{ct}_{\mathfrak{e}_2,0}] \in \left[\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_{2_{\ell_\mathfrak{e}-1}}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_{2_0}\right)\right]$
- $\mathbf{ct}_{\mathfrak{m}_2} = [\mathsf{ct}_{\mathfrak{m}_2,\ell_\mathfrak{m}-1}, \ldots, \mathsf{ct}_{\mathfrak{m}_2,0}] \in \left[\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{2_{\ell_\mathfrak{m}-1}}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{2_0}\right)\right]$
- PUB : Public materials for **KS**, **PBS** and **CBS**; `/* Remark 2.15 */`

Output:
- $\mathbf{ct}_{\mathfrak{m}'_1} = \left[\mathsf{ct}_{\mathfrak{m}'_1,\ell_\mathfrak{m}-1}, \ldots, \mathsf{ct}_{\mathfrak{m}'_1,0}\right] \in \left[\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}'_{1_{\ell_\mathfrak{m}-1}}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}'_{1_0}\right)\right]$
- $\mathbf{ct}_{\mathfrak{m}'_2} = \left[\mathsf{ct}_{\mathfrak{m}'_2,\ell_\mathfrak{m}-1}, \ldots, \mathsf{ct}_{\mathfrak{m}'_2,0}\right] \in \left[\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}'_{2_{\ell_\mathfrak{m}-1}}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}'_{2_0}\right)\right]$
- $\mathbf{ct}_\mathfrak{e} = [\mathsf{ct}_{\mathfrak{e},\ell_\mathfrak{e}-1}, \ldots, \mathsf{ct}_{\mathfrak{e},0}] \in [\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_{\ell_\mathfrak{e}-1}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_0\right)]$

`/* Subtraction between the two exponents follows by the bit extraction */`

$(\mathsf{ct}_\mathfrak{s}, \mathbf{ct}_d = [\mathsf{ct}_{d,\ell_\mathfrak{e}-1}, \ldots, \mathsf{ct}_{d,0}]) \leftarrow \mathsf{RadixSub}^*\left(\mathbf{ct}_{\mathfrak{e}_1}, \mathbf{ct}_{\mathfrak{e}_2}, \mathsf{PUB}\right)$; `/* Algorithm 46 */`

for $i \in [0..\ell_\mathfrak{e} - 1]$ **do**

  for $j \in [0..\rho_\mathfrak{e} - 1]$ **do**

    `/* Extract each bit of` $\mathbf{ct}_d$ `*/`

    $\overline{\overline{\mathsf{CT}}}_{d_{(i\cdot\rho_\mathfrak{e}+j)}} \leftarrow \mathbf{CBS}\left(\mathsf{ct}_{d,i}, \mathsf{PUB}\right)$; `/* Algorithm 12 */`

$\overline{\overline{\mathsf{CT}}}_{d_\mathfrak{s}} \leftarrow \mathbf{CBS}\left(\mathsf{ct}_\mathfrak{s}, \mathsf{PUB}\right)$; `/* Algorithm 12 */`

`/* selects the CT we need to align */`

$\mathbf{ct}^{\langle 0\rangle}_{\mathfrak{m}_{\mathsf{out}}} = [\mathsf{ct}_{\mathfrak{m}_{\mathsf{out}},\ell_\mathfrak{m}-1}, \ldots, \mathsf{ct}_{\mathfrak{m}_{\mathsf{out}},0}] \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}_{\mathfrak{m}_2}, \mathbf{ct}_{\mathfrak{m}_1}, \overline{\overline{\mathsf{CT}}}_{d_\mathfrak{s}}\right)$; `/* Algorithm 49 */`

for $i \in [1..\ell_\mathfrak{m}]$ **do**

  `/* Remove the` i^{th} `less significant blocks and add` i `trivial Zero LWEs on the most significant blocks */`

  $\mathbf{ct}^{\langle i\rangle}_{\mathfrak{m}_{\mathsf{out}}} \leftarrow [\mathsf{TrivialEncrypt}\left(0, i\right), \mathsf{ct}_{\mathfrak{m}_{\mathsf{out}},\ell_\mathfrak{m}-1}, \ldots, \mathsf{ct}_{\mathfrak{m}_{\mathsf{out}},i}]$

for $i \in [0..\ell_\mathfrak{e} \cdot \rho_\mathfrak{e})$ **do**

  if $\left\lfloor \ell_\mathfrak{m}/2^{i+1} \right\rfloor = 0$ **then**

    $\mathbf{ct}^{\langle 0\rangle}_{\mathfrak{m}_{\mathsf{out}}} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}^{\langle 0\rangle}_{\mathfrak{m}_{\mathsf{out}}}, \mathsf{TrivialEncrypt}\left(0, \ell_\mathfrak{m}\right), \overline{\overline{\mathsf{CT}}}_{d_i}\right)$; `/* Algorithm 49 */`

  else

    for $j \in \left[0..\left\lfloor \ell_\mathfrak{m}/2^{i+1} \right\rfloor\right]$ **do**

      `/* If` $\mathbf{ct}^{\langle 2\cdot j+1\rangle}_{\mathfrak{m}_{\mathsf{out}}}$ `is not defined, then it is equal to` $\mathsf{TrivialEncrypt}\left(0, \ell_\mathfrak{m}\right)$ `*/`

      $\mathbf{ct}^{\langle i\rangle}_{\mathfrak{m}_{\mathsf{out}}} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}^{\langle 2\cdot j\rangle}_{\mathfrak{m}_{\mathsf{out}}}, \mathbf{ct}^{\langle 2\cdot j+1\rangle}_{\mathfrak{m}_{\mathsf{out}}}, \overline{\overline{\mathsf{CT}}}_{d_i}\right)$; `/* Algorithm 49 */`

$\mathbf{ct}_{\mathfrak{m}'_1} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}_{\mathfrak{m}_1}, \mathbf{ct}^{\langle 0\rangle}_{\mathfrak{m}_{\mathsf{out}}}, \overline{\overline{\mathsf{CT}}}_{d_\mathfrak{s}}\right)$; `/* Algorithm 49 */`

$\mathbf{ct}_{m'_2} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}^{\langle 0\rangle}_{\mathfrak{m}_{\mathsf{out}}}, \mathbf{ct}_{\mathfrak{m}_2}, \overline{\overline{\mathsf{CT}}}_{d_\mathfrak{s}}\right)$; `/* Algorithm 49 */`

$\mathbf{ct}_\mathfrak{e} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}_{\mathfrak{e}_1}, \mathbf{ct}_{\mathfrak{e}_2}, \overline{\overline{\mathsf{CT}}}_{d_\mathfrak{s}}\right)$; `/* Algorithm 49 */`

return $\left(\mathbf{ct}_{\mathfrak{m}'_1}, \mathbf{ct}_{\mathfrak{m}'_2}, \mathbf{ct}_\mathfrak{e}\right)$

to be aligned. In next loop, encryption of all possible mantissa shifts are created, i.e., $M^{\langle 0\rangle} = \left\{\mathbf{ct}^{\langle i\rangle}_{\mathfrak{m}_{\mathsf{out}}} = [\mathbf{0}, \ldots, \mathbf{0}, \mathsf{ct}_{\mathfrak{m}_{\mathsf{out}},\ell_\mathfrak{m}-1}, \ldots, \mathsf{ct}_{\mathfrak{m}_{\mathsf{out}},i}]\right\}_{i\in[1,\ell_\mathfrak{m}-1]}$ *with* $\mathbf{0} \in \mathrm{LWE}_{\boldsymbol{s}}(0)$*, s.t.* $\mathfrak{m}^{\langle i\rangle}_{\mathsf{out}} = \left\lfloor \mathfrak{m}^{\langle 0\rangle}_{\mathsf{out}}/2^{\rho_\mathfrak{m}\cdot i} \right\rceil$*. The next steps consists in choosing the right mantissa from this set, depending on the value of* d*. Informally,* d *is the number of blocks by which the mantissa should be shifted. At each step* i *of the loop, the set* $M^{\langle i\rangle}$ *is updated with respect of the binary value of* d*, to contains the encryption of each value in* $\left\{\mathbf{ct}^{\langle \alpha\rangle}_{\mathfrak{m}_{\mathsf{out}}} = \left\lfloor \mathfrak{m}^{\langle 0\rangle}_{\mathsf{out}}/2^{\rho_\mathfrak{m}\cdot\alpha} \right\rceil \text{ s.t. } \alpha = \sum_{j=0}^{i-1} d_j 2^j \mod 2^{i+1}\right\}$*. Note that in the case where* $\mathfrak{e}_1 = \mathfrak{e}_2$*, the algorithm will return the selected mantissa without any change. In the end, the set*

is reduced to a singleton containing $\left\lfloor \mathfrak{m}_{\mathsf{out}}^{\langle 0 \rangle} / 2^{\rho_{\mathfrak{m}} \cdot d} \right\rfloor$. *Finally, the last three* CMux *gates replace the mantissa value with the aligned one and select the larger exponent.* □

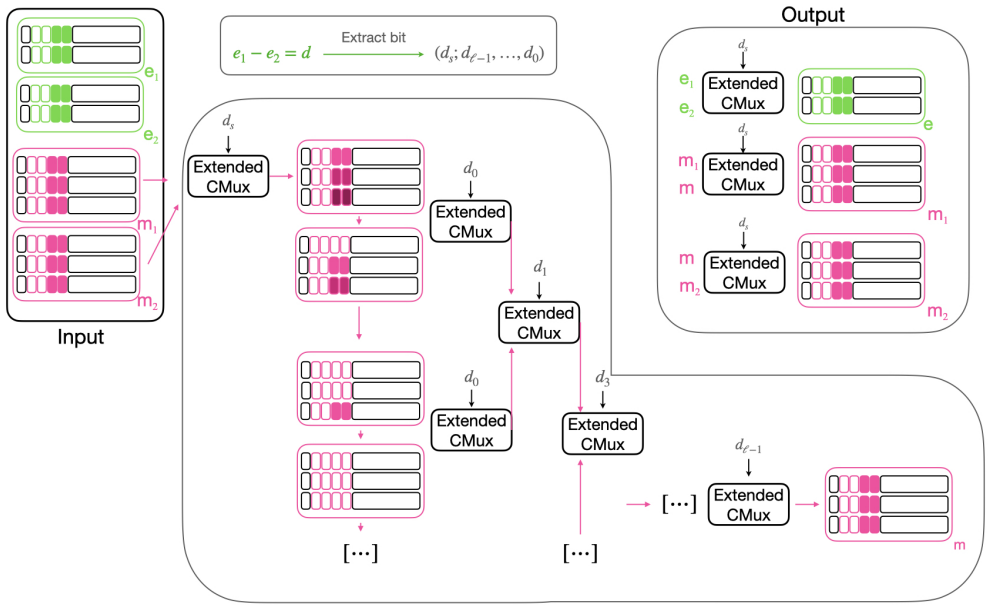

Figure 8.2: This figure illustrates the main steps of the Homomorphic AlignMantissa operation (see Algorithm 51). The goal is to align the mantissas based on their exponents. At a high level, we first use Algorithm 46 on the two exponents (in green) to compute the difference between the two exponents, along with the sign. Using the sign, we can determine which mantissa (in pink) is smaller. Then, with the difference and a tree of Extended CMux operations, we can decide how many ciphertexts are needed to increase the smaller mantissa and align the exponents. The final step involves correctly ordering the two mantissas and selecting the larger exponent.

Lemma 8.8 (Noise Constraints of Algorithm 51)**.** *The output noise variances of the ciphertexts of Algorithm 51,* $\mathbf{ct}_{m'_i}$ *and* $\mathbf{ct}_{\mathfrak{e}}$*, are respectively* $\sigma^2_{\mathsf{in},m} + \frac{(\rho_e \cdot \ell_e + 3)}{2} \cdot \sigma^2_{\mathsf{cmux}}$ *and* $\sigma^2_{\mathsf{in},e} + \sigma^2_{\mathsf{cmux}}$.

To guarantee correctness of Algorithm 51, we need to find parameters that verify the following inequalities:

$$4 \cdot \sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{KS}} + \sigma^2_{\mathsf{MS}} \leq t^2.$$

With $\sigma_{\mathsf{in},\mathfrak{e}}$ *the noise variance of the input exponent ciphertexts,* $\sigma_{\mathsf{in},\mathfrak{m}}$ *the noise variance of the input mantissa ciphertexts and with* σ_{BR} *the noise added by the blind rotation,* σ_{KS} *the noise added by the key switch,* σ_{CMux} *the noise added by the* CMux *and finally* σ_{MS}*, the noise added by the modulus switch. with* t^2*, the noise bound as defined in the proof of the noise constraints of Algorithm 46.*

Proof *(Lemma 8.8). Let us assume that the input ciphertexts* $\mathbf{ct}_{\mathfrak{e}_i}$ *and* $\mathbf{ct}_{\mathfrak{m}_i}$ *have respectively a noise variance* $\sigma^2_{\mathsf{in},\mathfrak{e}}$ *and* $\sigma^2_{\mathsf{in},\mathfrak{m}}$*. The first line calls Algorithm 46 (see Lemma 8.2 for more details). In particular, we compute the output noise of Algorithm 46. The noise variances of* $\mathsf{ct}_{\mathfrak{s}}$ *and* $\mathbf{ct}_{\boldsymbol{d}}$ *are respectively* $4 \cdot \sigma^2_{\mathsf{BR}}$ *and* σ^2_{BR} *with* σ^2_{BR}*, the noise variance of a freshly bootstrapped ciphertext. In the next lines, the algorithm heavily relies on circuit bootstrapping which gives us the following noise constraints:* $\sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{KS}} + \sigma^2_{\mathsf{MS}} \leq t^2$ & $4 \cdot \sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{KS}} + \sigma^2_{\mathsf{MS}} \leq t^2$*, with* σ^2_{KS} *and* σ^2_{MS} *respectively the noise variance added by a key switch and by a modulus switch.* t^2 *represents the noise bound as previously defined in the proof of Lemma 8.2. As the first constraint is dominated by the second, we can remove it from the set of constraints. Then, we apply an extended* CMux *and we create* $\ell_{\mathfrak{m}}$ *vectors of ciphertexts composed of outputs of the previous extended* CMux *and trivial*

ciphertexts. At this stage, we assume that every ciphertext in these vectors has the same noise variance $\sigma^2_{\mathsf{in},m} + \sigma^2_{\mathsf{cmux}}$ *with* σ^2_{cmux} *the noise added by a* CMux *using a GGSW coming from a circuit bootstrap with a noise variance* $\sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{PPKS}}$*. Then, we apply an extended* CMux *tree of depth* $\rho_e \cdot \ell_e$*. Therefore, at the end of the tree, the ciphertexts have a noise variance* $\sigma^2_{\mathsf{in},m} + (\rho_e \cdot \ell_e + 1) \cdot \sigma^2_{\mathsf{cmux}}$*. At the end of the algorithm, assuming that* $\mathbf{ct}_{\mathfrak{s}}$ *encrypt* 0 *or* 1 *with probability* $\frac{1}{2}$*, the noise variance of* $\mathbf{ct}_{m'_i}$ *is* $\sigma^2_{\mathsf{in},m} + \frac{(\rho_e \cdot \ell_e + 3)}{2} \cdot \sigma^2_{\mathsf{cmux}}$*. The noise variance of* $\mathbf{ct}_{\mathfrak{e}}$ *is* $\sigma^2_{\mathsf{in},e} + \sigma^2_{\mathsf{cmux}}$*.*

□

SubMantissa. SubMantissa performs the subtraction of two mantissas and shifts the result such that the most significant block is not empty (unless the result is zero or too small to be represented). It changes the value of the exponent and the sign consequently. To perform this operation, the two mantissas must to be aligned. Algorithm 52 takes as input the encryption of two aligned mantissas, the exponent and the sign, and returns the encryption of the absolute value of the difference of the mantissas, the exponent and the sign of this subtraction.

Lemma 8.9 (SubMantissa (Algorithm 52))**.** *Let* $\mathbf{ct}_{\mathfrak{m}_i} = [\mathsf{ct}_{\mathfrak{m}_i,\ell_{\mathfrak{m}}-1}, \dots \mathsf{ct}_{\mathfrak{m}_i,0}] \in \left[\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{i_{\ell_{\mathfrak{m}}-1}}\right), \dots \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{i_0}\right)\right]$ *with* $i \in \{0,1\}$ *be ciphertexts encrypting* $\mathfrak{m}_i < 2^{\rho_{\mathfrak{m}}}$*. Let* $\mathbf{ct}_{\mathfrak{e}} = [\mathsf{ct}_{\mathfrak{e},\ell_{\mathfrak{e}}-1}, \dots \mathsf{ct}_{\mathfrak{e},0}] \in [\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_{\ell_{\mathfrak{e}}-1}\right), \dots \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_0\right)]$ *a ciphertexts encrypting* $\mathfrak{e} < 2^{\rho_{\mathfrak{e}}}$*. Let* $\mathsf{ct}_{\mathfrak{s}_1} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}_1)$ *and* $\mathsf{ct}_{\mathfrak{s}_2} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}_2)$ *with* $\mathfrak{s}_1 = 1 - \mathfrak{s}_2$ *such that* $\mathfrak{f}_1 = (-1)^{\mathfrak{s}_1} \cdot \mathfrak{m}_1 \cdot (2^{\rho_{\mathfrak{m}}})^{\mathfrak{e}-\mathsf{bias}}$ *and* $\mathfrak{f}_2 = (-1)^{\mathfrak{s}_2} \cdot \mathfrak{m}_2 \cdot (2^{\rho_{\mathfrak{m}}})^{\mathfrak{e}-\mathsf{bias}}$*.*

Then, Algorithm 52 outputs $\mathbf{ct}_{\mathfrak{f}_{\mathsf{sub}}} = (\mathbf{ct}_{\mathfrak{m}_{\mathsf{sub}}}, \mathbf{ct}_{\mathfrak{e}_{\mathsf{sub}}}, \mathsf{ct}_{\mathfrak{s}_{\mathsf{sub}}})$ *such that* $\mathsf{DecryptFloat}_{\boldsymbol{s}}(\mathbf{ct}_{\mathfrak{f}_{\mathsf{sub}}}) = \mathfrak{f}_1 - \mathfrak{f}_2 = (-1)^{\mathfrak{s}_{\mathsf{sub}}} \cdot \mathfrak{m}_{\mathsf{sub}} \cdot (2^{\rho_{\mathfrak{m}}})^{\mathfrak{e}_{\mathsf{sub}}-\mathsf{bias}}$ *with* $\mathfrak{s}_{\mathsf{sub}} = \mathfrak{s}_1$ *if* $\mathfrak{m}_1 \geq \mathfrak{m}_2$*, or* $\mathfrak{s}_2$ *if* $\mathfrak{m}_1 < \mathfrak{m}_2$*.*

Assuming $\mathfrak{m}_1 \neq \mathfrak{m}_2$*, we note* α *be the index of the first non zero block of* $\mathfrak{m} = |\mathfrak{m}_1 - \mathfrak{m}_2|$ *i.e.,* $\alpha = \min_{i \in [0, \ell_{\mathfrak{m}}-1]} \{\ell_{\mathfrak{m}} - 1 - i \text{ s.t. } \mathfrak{m}_i \neq 0\}$*. Then if* $\mathfrak{e} \geq \alpha$*,* $\mathfrak{e}_{\mathsf{sub}} = \mathfrak{e} - \alpha$ *and* $\mathfrak{m}_{\mathsf{sub}} = |\mathfrak{m}_1 - \mathfrak{m}_2| \cdot 2^{\rho_{\mathfrak{m}} \cdot \alpha}$*. Else if* $\mathfrak{m}_1 = \mathfrak{m}_2$ *or if* $\mathfrak{e} - \alpha < 0$*, then,* $\mathfrak{m}_{\mathsf{sub}} = 0$*,* $\mathfrak{e}_{\mathsf{sub}} = 0$*.*
The complexity of the algorithm is:

$$\mathsf{Cost}^{\ell_{\mathbf{PBS}},\ell_{\mathbf{CBS}},k,N,n,q,\ell_{\mathfrak{e}},\rho_{\mathfrak{e}},\ell_{\mathfrak{m}}}_{\mathsf{SubMantissa}} = \mathsf{Cost}^{\ell_{\mathfrak{m}},\ell_{\mathbf{PBS}},k,N,n,q}_{\mathsf{RadixSub}^*} + \mathsf{Cost}^{\ell_{\mathfrak{e}},\ell_{\mathbf{PBS}},k,N,n,q}_{\mathsf{RadixSub}^*} + (\ell_{\mathfrak{m}} + 1) \cdot \mathsf{Cost}^{\ell_{\mathbf{PBS}},\ell_{\mathbf{CBS}},k,N,n,q}_{\mathbf{CBS}}.$$

Proof *(Correctness of* SubMantissa *(Algorithm 52)). The first step of the algorithm is to subtract the two mantissas. We obtain* $\mathbf{ct}_{\mathfrak{m}_0}$ *which is equals to* $|\mathbf{ct}_{\mathfrak{m}_{\mathsf{in},1}} - \mathbf{ct}_{\mathfrak{m}_{\mathsf{in},2}}|$ *and* $\mathsf{ct}_{\mathfrak{s}}$ *the sign of this subtraction. As the two mantissas are aligned, we have* $\mathfrak{m}_0$ *in* $[0, 2^{\ell_{\mathfrak{m}} \cdot \rho_{\mathfrak{m}}})$*.*

At each step i *of the loop, we take the previously computed ciphertext* $\mathsf{ct}_{\mathfrak{m}_{i-1}}$ *encrypting a message* $\mathfrak{m}_{i-1}$ *and build another ciphertext* $\mathsf{ct}_{\mathfrak{m}_i}$ *encrypting the message* $\mathfrak{m}_i = \mathfrak{m}_{i-1} \cdot 2^{\rho_{\mathfrak{m}}}$*. On line 7, we create a* GGSW *ciphertext* $\overline{\overline{\mathsf{CT}}}$ *encrypting* 0 *if the most significant block of the mantissa* $\mathfrak{m}_{i-1,\ell_{\mathfrak{m}}-1}$ *is equal to* 0 *and* 1 *if it contains some non-zero integer. Remember, our goal is to realign the mantissa to stay in the classical representation (i.e.,* $\mathfrak{m}_{\mathsf{sub}} \in \left[2^{(\ell_{\mathfrak{m}}-1) \cdot \rho_{\mathfrak{m}}}, 2^{\ell_{\mathfrak{m}} \cdot \rho_{\mathfrak{m}}}\right)$ *or* $\mathfrak{m}_{\mathsf{sub}} = 0$ *if* $\mathfrak{m}_{\mathsf{in},1} = \mathfrak{m}_{\mathsf{in},2}$*). Therefore, we use a* CMux *to select* $\mathsf{ct}_{\mathfrak{m}_{i-1}}$ *if* $\mathfrak{m}_{i-1,\ell_{\mathfrak{m}}-1} \neq 0$ *and* $\mathsf{ct}_{\mathfrak{m}_i}$ *if* $\mathfrak{m}_{i-1,\ell_{\mathfrak{m}}-1} = 0$*. In the same way, we use the* CMux *to update the value of the exponent. To do so, we take the previously computed* $\mathsf{ct}_{\mathfrak{e}_{i-1}}$ *and create a new ciphertext* $\mathsf{ct}_{\mathfrak{e}_i}$*, a trivial encryption of* i*. As we want to select* $\mathsf{ct}_{\mathfrak{e}_{i-1}}$ *(respectively* $\mathsf{ct}_{\mathfrak{e}_i}$*) if we selected* $\mathsf{ct}_{\mathfrak{m}_{i-1}}$ *(resp.* $\mathsf{ct}_{\mathfrak{m}_i}$*) in the previous step, we can perform a* CMux *with the same GGSW ciphertext* $\overline{\overline{\mathsf{CT}}}$*. Assuming that we have* $\mathfrak{m}_0 \neq 0$*, at the end of the for-loop,* $\mathsf{ct}_{\mathfrak{e}_i}$ *is encrypting a value* α *such that* $\mathfrak{m}_0 \cdot 2^{\rho_{\mathfrak{m}} \cdot \alpha} \in \left[2^{(\ell_{\mathfrak{m}}-1) \cdot \rho_{\mathfrak{m}}}, 2^{\ell_{\mathfrak{m}} \cdot \rho_{\mathfrak{m}}}\right)$ *and* $\mathsf{ct}_{\mathfrak{m}_i}$ *is encrypting* $\mathfrak{m}_0 \cdot 2^{\rho_{\mathfrak{m}} \cdot \alpha}$*. If we have* $\mathfrak{m}_0 = 0$*,* $\mathsf{ct}_{\mathfrak{m}i}$ *will still be equal to* 0*.*

The next step is to update the exponent. In line 13*, we subtract the value* α *encrypted in* $\mathsf{ct}_{\mathfrak{e}_{\ell_{\mathfrak{m}}}}$ *to* $\mathsf{ct}_{\mathfrak{e}}$*. The sign of this subtraction is in* $\mathrm{LWE}_{\boldsymbol{s}}(0)$ *if we can do the subtraction (*$\mathfrak{e} > \alpha$*) otherwise, the result is in* $\mathrm{LWE}_{\boldsymbol{s}}(1)$*, the value of the subtraction is too small to be represented with our encoding. With this sign we create a new* GGSW *ciphertext* $\overline{\overline{\mathsf{CT}}}$*. Finally, the last* CMux *returns the ciphertexts encrypting mantissa* $\mathfrak{m}_{\mathsf{sub}} = \mathfrak{m}_0 \cdot 2^{\rho_{\mathfrak{m}} \cdot \alpha}$ *and the ciphertexts encrypting the exponent* $\mathfrak{e}_{\mathsf{sub}} = \mathfrak{e} - \alpha$ *if the subtraction can be done, otherwise it returns the ciphertexts encrypting 0.* □

Algorithm 52: $(\mathbf{ct}_{\mathsf{m}_{\mathsf{sub}}}, \mathbf{ct}_{\mathfrak{e}_{\mathsf{sub}}}, \mathsf{ct}_{\mathfrak{s}_{\mathsf{sub}}}) \leftarrow \mathsf{SubMantissa}(\mathbf{ct}_{\mathsf{m}_{\mathsf{in},1}}, \mathbf{ct}_{\mathsf{m}_{\mathsf{in},2}}, \mathbf{ct}_{\mathfrak{e}}, \mathsf{ct}_{\mathfrak{s}_1}, \mathsf{PUB})$

Input:
- $\mathbf{ct}_{\mathsf{m}_{\mathsf{in},1}} = \left[\mathsf{ct}_{\mathsf{m}_{\mathsf{in},1},\ell_\mathsf{m}-1}, \ldots, \mathsf{ct}_{\mathsf{m}_{\mathsf{in},1},0}\right] \in \left[\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{\mathsf{in},1_{\ell_\mathsf{m}-1}}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{\mathsf{in},1_0}\right)\right]$
- $\mathbf{ct}_{\mathsf{m}_{\mathsf{in},2}} = \left[\mathsf{ct}_{\mathsf{m}_{\mathsf{in},2},\ell_\mathsf{m}-1}, \ldots, \mathsf{ct}_{\mathsf{m}_{\mathsf{in},2},0}\right] \in \left[\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{\mathsf{in},2_{\ell_\mathsf{m}-1}}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{\mathsf{in},2_0}\right)\right]$
- $\mathbf{ct}_{\mathfrak{e}} = [\mathsf{ct}_{\mathfrak{e},\ell_\mathfrak{e}-1}, \ldots, \mathsf{ct}_{\mathfrak{e},0}] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{\ell_\mathfrak{e}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_0)]$
- $\mathsf{ct}_{\mathfrak{s}_1} \in \mathrm{LWE}_{\boldsymbol{s}}(sign_1)$
- PUB : Public keys for **KS**, **PBS** and **CBS**; `/* Remark 2.15 */`

Output:
- $\mathbf{ct}_{\mathsf{m}_{\mathsf{sub}}} = [\mathsf{ct}_{\mathsf{m}_{\mathsf{sub}},\ell_\mathsf{m}-1}, \ldots, \mathsf{ct}_{\mathsf{m}_{\mathsf{sub}},0}] \in \left[\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{\mathsf{sub}_{\ell_\mathsf{m}-1}}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{\mathsf{sub}_0}\right)\right]$
- $\mathbf{ct}_{\mathfrak{e}_{\mathsf{sub}}} = [\mathsf{ct}_{\mathfrak{e}_{\mathsf{sub}},\ell_\mathfrak{e}-1}, \ldots, \mathsf{ct}_{\mathfrak{e}_{\mathsf{sub}},0}] \in \left[\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_{\mathsf{sub}_{\ell_\mathfrak{e}-1}}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_{\mathsf{sub}_0}\right)\right]$
- $\mathsf{ct}_{\mathfrak{s}_{\mathsf{sub}}} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}_{\mathsf{sub}})$

$(\mathsf{ct}_\mathfrak{s}, \mathbf{ct}_{\mathfrak{m}_0} = [\mathsf{ct}_{\mathfrak{m}_0,\ell_\mathsf{m}-1}, \ldots, \mathsf{ct}_{\mathfrak{m}_0,0}]) \leftarrow \mathsf{RadixSub}^*\left(\mathbf{ct}_{\mathsf{m}_{\mathsf{in},1}}, \mathbf{ct}_{\mathsf{m}_{\mathsf{in},2}}, \mathsf{PUB}\right)$; `/* Algorithm 46 */`

$\mathbf{ct}_{\mathfrak{e}_0} \leftarrow \mathsf{TrivialEncrypt}(0, \ell_\mathfrak{e})$

for $i \in [1..\ell_\mathsf{m}]$ **do**

- $\mathbf{ct}'_{\mathfrak{e}_i} \leftarrow \mathsf{TrivialEncrypt}(i, \ell_\mathfrak{e})$
- $\mathsf{ct}_0 \leftarrow \mathsf{TrivialEncrypt}(0, 1)$
- $\mathbf{ct}_{\mathfrak{m}_i} \leftarrow \left[\mathsf{ct}_{\mathfrak{m}_{i-1},\ell_\mathsf{m}-2}, \ldots, \mathsf{ct}_{\mathfrak{m}_{i-1},0}, \mathsf{ct}_0\right]$

  `/*` $\overline{\overline{\mathsf{CT}}} \in \mathrm{GGSW}^{\mathcal{B},\ell}_{\boldsymbol{S}}(0)$ `if` $\mathsf{ct}_{m_{i-1},\ell_\mathsf{m}-1} \in \mathrm{LWE}_{\boldsymbol{s}}(0)$, $\overline{\overline{\mathsf{CT}}} \in \mathrm{GGSW}^{\mathcal{B},\ell}_{\boldsymbol{S}}(1)$ `otherwise` `*/`
- $\overline{\overline{\mathsf{CT}}} \leftarrow \mathbf{CBS}\left(\mathsf{ct}_{\mathfrak{m}_{i-1},\ell_\mathsf{m}-1}, \mathsf{PUB}\right)$; `/* Algorithm 12 */`
- $\mathbf{ct}_{\mathfrak{m}_i} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}_{\mathfrak{m}_i}, \mathbf{ct}_{\mathfrak{m}_{i-1}}, \overline{\overline{\mathsf{CT}}}\right)$; `/* Algorithm 49 */`
- **if** $i \neq \ell_\mathsf{m}$ **then** $\mathbf{ct}'_{\mathfrak{e}_i} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}'_{\mathfrak{e}_i}, \mathbf{ct}'_{\mathfrak{e}_{i-1}}, \overline{\overline{\mathsf{CT}}}\right)$; `/* Algorithm 49 */`
- **else** $\mathbf{ct}'_{\mathfrak{e}_i} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}_{\mathfrak{e}}, \mathbf{ct}'_{\mathfrak{e}_{i-1}}, \overline{\overline{\mathsf{CT}}}\right)$; `/* Algorithm 49 */`
- ;

$(\mathsf{ct}_{\mathfrak{s}_\mathfrak{e}}, \mathbf{ct}_{\mathfrak{e}_{res}}) \leftarrow \mathsf{RadixSub}^*\left(\mathbf{ct}_{\mathfrak{e}}, \mathbf{ct}'_{\mathfrak{e}_{\ell_\mathsf{m}}}, \mathsf{PUB}\right)$; `/* Algorithm 46 */`

`/*` $\overline{\overline{\mathsf{CT}}}_{s_\mathfrak{e}} \in \mathrm{GGSW}^{\mathcal{B},\ell}_{\boldsymbol{S}}(0)$ `if` $\mathsf{ct}_{\mathfrak{s}_\mathfrak{e}} \in \mathrm{LWE}_{\boldsymbol{s}}(0)$, $\overline{\overline{\mathsf{CT}}}_{\mathfrak{s}_\mathfrak{e}} \in \mathrm{GGSW}^{\mathcal{B},\ell}_{\boldsymbol{S}}(1)$ `otherwise` `*/`

$\overline{\overline{\mathsf{CT}}}_{\mathfrak{s}_\mathfrak{e}} \leftarrow \mathbf{CBS}(\mathsf{ct}_{\mathfrak{s}_\mathfrak{e}}, \mathsf{PUB})$; `/* Algorithm 12 */`

$\mathbf{ct}_{\mathfrak{m}_{\mathsf{sub}}} \leftarrow \mathsf{ExtendedCMux}\left(\mathsf{TrivialEncrypt}(0, \ell_\mathsf{m}), \mathbf{ct}_{\mathfrak{m}_{\ell_\mathsf{m}}}, \overline{\overline{\mathsf{CT}}}_{\mathfrak{s}_\mathfrak{e}}\right)$; `/* Algorithm 49 */`

$\mathbf{ct}_{\mathfrak{e}_{\mathsf{sub}}} \leftarrow \mathsf{ExtendedCMux}\left(\mathsf{TrivialEncrypt}(0, \ell_\mathfrak{e}), \mathbf{ct}_{\mathfrak{e}_{res}}, \overline{\overline{\mathsf{CT}}}_{\mathfrak{s}_\mathfrak{e}}\right)$; `/* Algorithm 49 */`

$\mathsf{ct}_{\mathfrak{s}_{\mathsf{sub}}} \leftarrow \mathsf{ct}_{\mathfrak{s}_1} + \mathsf{ct}_\mathfrak{s}$

return $(\mathbf{ct}_{\mathfrak{e}_{\mathsf{sub}}}; \mathbf{ct}_{\mathfrak{m}_{\mathsf{sub}}}; \mathsf{ct}_{\mathfrak{s}_{\mathsf{sub}}})$

Lemma 8.10 (Noise Constraints of Algorithm 52)**.** *The output noise variances of ciphertexts of Algorithm 52,* $\mathbf{ct}_{\mathfrak{m}_{\mathsf{sub}}}, \mathbf{ct}_{\mathfrak{e}_{\mathsf{sub}}}$ *and* $\mathsf{ct}_{\mathfrak{s}_{\mathsf{sub}}}$*, are respectively* $\sigma^2_{\mathsf{BR}} + (\ell_\mathsf{m}+1)\cdot\sigma^2_{\mathsf{cmux}}$*,* $\sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{cmux}}$ *and* $\sigma^2_{\mathsf{in},s} + 4\sigma^2_{\mathsf{BR}}$*.*

To guarantee correctness of Algorithm 52, we need to find parameters that verify the following inequalities:

$$\sigma^2_{\mathsf{BR}} + (i-1)\cdot\sigma^2_{\mathsf{cmux}} + \sigma^2_{\mathsf{KS}} + \sigma^2_{\mathsf{MS}} \leq t^2 \;\;\&\;\; 4\sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{KS}} + \sigma^2_{\mathsf{MS}} \leq t^2.$$

With $\sigma_{\mathsf{in},\mathfrak{s}}$ *the noise variance of the input sign ciphertext and with* σ_{BR} *the noise added by the blind rotation,* σ_{KS} *the noise added by the key switch,* σ_{CMux} *the noise added by the* CMux *and finally* σ_{MS}*, the noise added by the modulus switch and* t^2*, the noise bound as defined in the proof of the noise constraints of Algorithm 46.*

Proof *(Lemma 8.10). Let us assume that the input ciphertexts* $\mathbf{ct}_\mathfrak{e}$*,* $\mathbf{ct}_{\mathfrak{m}_i}$ *and* $\mathsf{ct}_{\mathfrak{s}_1}$ *have respectively a noise variance* $\sigma^2_{\mathsf{in},\mathsf{e}}$*,* $\sigma^2_{\mathsf{in},\mathsf{m}}$ *and* $\sigma^2_{\mathsf{in},\mathsf{s}}$*.*

Using the noise analysis of Algorithm 46 presented in Lemma 8.2, we know that the noise variance of $\mathbf{ct}_{\boldsymbol{m_0}}$ *and* $\mathsf{ct}_\mathfrak{s}$ *are respectively* σ^2_{BR} *and* $4\cdot\sigma^2_{\mathsf{BR}}$*. In the for-loop, for each index* $1 \leq$

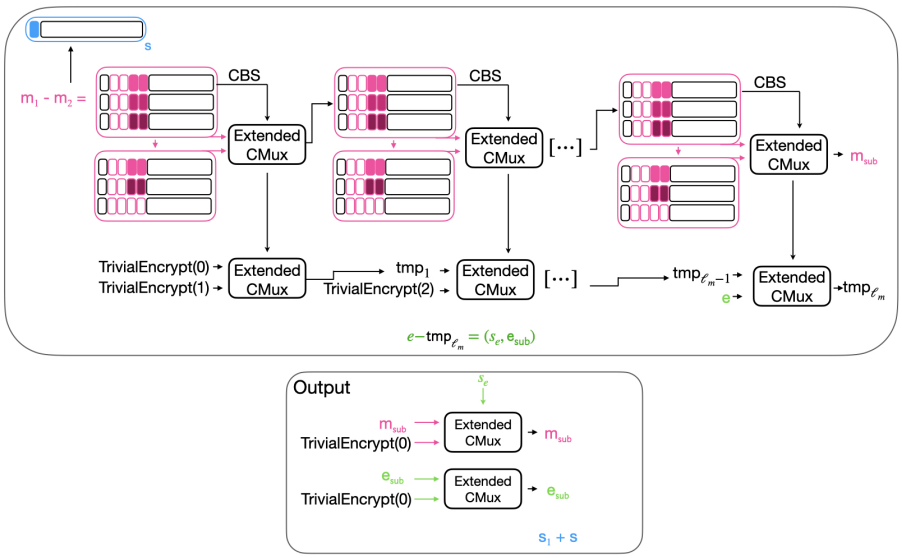

Figure 8.3: This figure represents the main steps of Algorithm 52. At a high level, we first subtract the two mantissas using Algorithm 46, obtaining the absolute value of the subtraction along with the sign. Next, we perform a loop where, at each step, the first ciphertext of the new mantissa is transformed into a GGSW ciphertext using a **CBS**. Using this GGSW ciphertext, we remove the most significant ciphertext if it is empty otherwise, we keep the mantissa. At the same time, at each step, we count the number of ciphertexts removed. When the loop finishes, we subtract the number of removed ciphertexts from the exponent. Finally, we use an extended CMux to return 0 if the exponent is negative or if the mantissa is empty.

$i \leq \ell_{\mathfrak{m}}$, *we can compute the noise constraint* $\sigma^2_{\mathsf{BR}} + (i-1)\cdot\sigma^2_{\mathsf{cmux}} + \sigma^2_{\mathsf{KS}} + \sigma^2_{\mathsf{MS}} \leq t^2$ *and the noise variance of* $\mathbf{ct}_{\mathfrak{m}_i}$ *is* $\sigma^2_{\mathsf{BR}} + i\cdot\sigma^2_{\mathsf{cmux}}$, *the noise variance of* $\mathbf{ct}_{\mathfrak{e}_i}$ *is* $i\cdot\sigma^2_{\mathsf{cmux}}$ *for* $i \neq \ell_{\mathfrak{m}}$ *and* $\max\left((\ell_{\mathfrak{m}}-1)\cdot\sigma^2_{\mathsf{cmux}}, \sigma^2_{\mathsf{in},e}\right) + \sigma^2_{\mathsf{cmux}}$ *for* $i = \ell_{\mathfrak{m}}$. *As in the previous proofs, we only need to retain the noise constraint for* $i = \ell_{\mathfrak{m}}$, *as it dominates the other constraints. Then, we have a call to Algorithm 46 which gives us ciphertexts of variances respectively* $4\sigma^2_{\mathsf{BR}}$ *and* σ^2_{BR}. *Next, we perform a circuit bootstrapping which gives us the following constraint* $4\sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{KS}} + \sigma^2_{\mathsf{MS}} \leq t^2$. *Finally, the algorithm outputs* $\mathbf{ct}_{\mathfrak{m}_{\mathsf{sub}}}, \mathbf{ct}_{\mathfrak{e}_{\mathsf{sub}}}$ *and* $\mathsf{ct}_{\mathfrak{s}_{\mathsf{sub}}}$ *of respective variances* $\sigma^2_{\mathsf{BR}} + (\ell_{\mathfrak{m}}+1)\cdot\sigma^2_{\mathsf{cmux}}$, $\sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{cmux}}$ *and* $\sigma^2_{\mathsf{in},s} + 4\sigma^2_{\mathsf{BR}}$. □

8.4.2 Addition and Subtraction

This operation performs the addition of two homomorphic floating-point numbers. To perform a subtraction, we only need to change the input sign of the second ciphertext. This operation is straightforward, as the sign is on a padding bit, adding the clear integer $q/2$ to the sign ciphertext change the sign of the floating-point.

This operation is based on the previous algorithms. We first need to align the mantissas (Algorithm 51). Next, we perform both the addition and the subtraction (Algorithm 52) of the mantissas and then we choose which of the two results to output based on the signs. All the steps of this operation are illustrated in Figure 8.4.

After this operation, we need to perform the operation CarryPropagateFloat (Algorithm 50) to retrieve a proper homomorphic floating-point representation.

Lemma 8.11 (Addition (Algorithm 53))**.** *Let* $\mathbf{ct}_{\mathfrak{f}_i}$ *such that* $\mathsf{ct}_{\mathfrak{s}_i} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}_i)$,$\mathbf{ct}_{\mathfrak{e}_i} = \left[\mathsf{ct}_{\mathfrak{e}_{i,\ell_{\mathfrak{e}}-1}}, \ldots, \mathsf{ct}_{\mathfrak{e}_{i,0}}\right] \in \left[\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{i,\ell_{\mathfrak{e}}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{i,0})\right]$ *encrypting* $\mathfrak{e}_i < 2^{\rho_{\mathfrak{e}}}$ *and* $\mathbf{ct}_{\mathfrak{m}_i} =$

$\left[\mathsf{ct}_{\mathfrak{m}_{i,\ell_\mathfrak{m}-1}},\ldots,\mathsf{ct}_{\mathfrak{m}_{i,0}}\right] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{i,\ell_\mathfrak{m}-1}),\ldots,\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{i,0})]$ *encrypting* $\mathfrak{m}_i < 2^{\rho_\mathfrak{m}}$ *with* $i \in \{1,2\}$ *be two ciphertexts encrypting* $\mathfrak{f}_i = (-1)^{\mathfrak{s}_i} \cdot \mathfrak{m}_i \cdot (2^{\rho_\mathfrak{m}})^{\mathfrak{e}_i - \mathsf{bias}}$.

Then, Algorithm 53 returns $(\mathbf{ct}_{\mathfrak{m}_{\mathsf{res}}}, \mathbf{ct}_{\mathfrak{e}_{\mathsf{res}}}, \mathsf{ct}_{\mathfrak{s}_{\mathsf{res}}}) = \mathbf{ct}_{\mathfrak{f}_{\mathsf{res}}}$ *such that* $\mathsf{DecryptFloat}(\mathbf{ct}_{\mathfrak{f}_{\mathsf{res}}}) = (-1)^{\mathfrak{s}_{\mathsf{res}}} \cdot \mathfrak{m}_{\mathsf{res}} \cdot (2^{\rho_\mathfrak{m}})^{\mathfrak{e}_{\mathsf{res}}-\mathsf{bias}} = \mathfrak{f}_{\mathsf{res}} = \mathfrak{f}_1 + \mathfrak{f}_2 + \epsilon$ *with,* $\mathfrak{m}'_1 = \mathfrak{m}_1$ *and* $\mathfrak{m}'_2 = \lfloor \mathfrak{m}_2 / 2^{\rho_\mathfrak{m} \cdot \gamma} \rfloor$ *for* $\gamma = \mathfrak{e}_1 - \mathfrak{e}_2$, *if* $\mathfrak{e}_1 \geq \mathfrak{e}_2$. *Otherwise* $\mathfrak{m}'_2 = \mathfrak{m}_2$ *and* $\mathfrak{m}'_1 = \lfloor \mathfrak{m}_1 / 2^{\rho_\mathfrak{m} \cdot \gamma} \rfloor$ *for* $\gamma = \mathfrak{e}_2 - \mathfrak{e}_1$. *For* ϵ, *we refer to Definition 35. If* $\mathfrak{s}_1 = \mathfrak{s}_2$, *then* $\mathfrak{s}_{\mathsf{res}} = \mathfrak{s}_1$, $\mathfrak{e}_{\mathsf{res}} = \max(\mathfrak{e}_1, \mathfrak{e}_2)$ *and* $\mathfrak{m}_{res} = \mathfrak{m}'_1 + \mathfrak{m}'_2$.

Else if $\mathfrak{s}_1 \neq \mathfrak{s}_2$ *and if* $\mathfrak{m}'_1 \geq \mathfrak{m}'_2$, *then* $\mathfrak{s}_{\mathsf{res}} = \mathfrak{s}_1$; *or if* $\mathfrak{m}'_1 < \mathfrak{m}'_2$, *then* $\mathfrak{s}_{\mathsf{res}} = \mathfrak{s}_2$. *Assuming* $\mathfrak{m}'_1 \neq \mathfrak{m}'_2$ *and* $\mathfrak{s}_1 \neq \mathfrak{s}_2$, *let* α *be the index of the first non zero block of* $\mathfrak{m} = |\mathfrak{m}'_1 - \mathfrak{m}'_2|$, *if* $\max(\mathfrak{e}_1, \mathfrak{e}_2) \geq \alpha$, *then* $\mathfrak{e}_{\mathsf{res}} = \max(\mathfrak{e}_1, \mathfrak{e}_2) - \alpha$ *and* $\mathfrak{m}_{\mathsf{res}} = |\mathfrak{m}'_1 - \mathfrak{m}'_2| \cdot 2^{\ell_\mathfrak{m} \cdot \alpha}$. *Else if* $\max(\mathfrak{e}_1, \mathfrak{e}_2) < \alpha$, *then* $\mathfrak{e}_{\mathsf{res}} = 0$, $\mathfrak{m}_{\mathsf{res}} = 0$ *and* $\mathfrak{e}_{\mathsf{res}} = 0$.

The complexity of the algorithm is:

$$\mathsf{Cost}^{\ell_{\mathbf{PBS}},\ell_{\mathbf{CBS}},k,N,n,q,\ell_\mathfrak{e},\rho_\mathfrak{e},\ell_\mathfrak{m}}_{\mathsf{Addition}} = \mathsf{Cost}^{\ell_{\mathbf{PBS}},\ell_{\mathbf{CBS}},k,N,n,q,\ell_\mathfrak{e},\rho_\mathfrak{e},\ell_\mathfrak{m}}_{\mathsf{SubMantissa}} + \mathsf{Cost}^{\ell_{\mathbf{PBS}},\ell_{\mathbf{CBS}},k,N,n,q,\ell_\mathfrak{e},\rho_\mathfrak{e}}_{\mathsf{AlignMantissa}} + \mathsf{Cost}^{\ell_{\mathbf{PBS}},\ell_{\mathbf{CBS}},k,N,n,q}_{\mathbf{CBS}}.$$

Algorithm 53: $\mathbf{ct}_{\mathfrak{f}_{\mathsf{res}}} \leftarrow \mathsf{Addition}(\mathbf{ct}_{\mathfrak{f}_1}, \mathbf{ct}_{\mathfrak{f}_2}, \mathsf{PUB})$

Input:
$\mathbf{ct}_{\mathfrak{f}_1} = \begin{cases} \mathsf{ct}_{\mathfrak{s}_1} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}_1) \\ \mathbf{ct}_{\mathfrak{e}_1} = \left[\mathsf{ct}_{\mathfrak{e}_{1,\ell_\mathfrak{e}-1}},\ldots,\mathsf{ct}_{\mathfrak{e}_{1,0}}\right] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{1,\ell_\mathfrak{e}-1}),\ldots,\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{1,0})] \\ \mathbf{ct}_{\mathfrak{m}_1} = \left[\mathsf{ct}_{\mathfrak{m}_{1,\ell_\mathfrak{m}-1}},\ldots,\mathsf{ct}_{\mathfrak{m}_{1,0}}\right] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{1,\ell_\mathfrak{m}-1}),\ldots,\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{1,0})] \end{cases}$
$\mathbf{ct}_{\mathfrak{f}_2} = \begin{cases} \mathsf{ct}_{\mathfrak{s}_2} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}_2) \\ \mathbf{ct}_{\mathfrak{e}_2} = \left[\mathsf{ct}_{\mathfrak{e}_{2,\ell_\mathfrak{e}-1}},\ldots,\mathsf{ct}_{\mathfrak{e}_{2,0}}\right] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{2,\ell_\mathfrak{e}-1}),\ldots,\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{2,0})] \\ \mathbf{ct}_{\mathfrak{m}_2} = \left[\mathsf{ct}_{\mathfrak{m}_{2,\ell_\mathfrak{m}-1}},\ldots,\mathsf{ct}_{\mathfrak{m}_{2,0}}\right] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{2,\ell_\mathfrak{m}-1}),\ldots,\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{2,0})] \end{cases}$
PUB : Public materials for **PBS**, **KS** and **CBS**; `/* Remark 2.15 */`

Output:
$\mathbf{ct}_{\mathfrak{f}_{\mathsf{res}}} = \begin{cases} \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}_{\mathsf{res}}) \\ \mathbf{ct}_{\mathfrak{e}_{\mathsf{res}}} = [ct_{\mathfrak{e}_{\mathsf{res}},\ell_\mathfrak{e}-1},\ldots,ct_{\mathfrak{e}_0}] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{\mathsf{res},\ell_\mathfrak{e}-1}),\ldots,\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{\mathsf{res},0})] \\ \mathbf{ct}_{\mathfrak{m}_{\mathsf{res}}} = [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{\mathsf{res},\ell_\mathfrak{m}-1}),\ldots,\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{\mathsf{res},0})] \end{cases}$

1 $\left(\mathbf{ct}_{\mathfrak{m}'_1}, \mathbf{ct}_{\mathfrak{m}'_2}, \mathbf{ct}_\mathfrak{e}\right) \leftarrow \mathsf{AlignMantissa}(\mathbf{ct}_{\mathfrak{e}_1}, \mathbf{ct}_{\mathfrak{m}_1}, \mathbf{ct}_{\mathfrak{e}_2}, \mathbf{ct}_{\mathfrak{m}_2}, \mathsf{PUB})$; `/* Algorithm 51 */`
2 $\mathbf{ct}_{\mathfrak{m}_{\mathsf{add}}} \leftarrow \mathsf{Add}(\mathsf{ct}_{\mathfrak{m}'_1}, \mathsf{ct}_{\mathfrak{m}'_2})$; `/* Algorithm 40 */`
3 $(\mathbf{ct}_{\mathfrak{m}_{\mathsf{sub}}}, \mathbf{ct}_{\mathfrak{e}_{\mathsf{sub}}}, \mathsf{ct}_{\mathfrak{s}_{\mathsf{sub}}}) \leftarrow \mathsf{SubMantissa}(\mathbf{ct}_{\mathfrak{m}'_1}, \mathbf{ct}_{\mathfrak{m}'_2}, \mathbf{ct}_\mathfrak{e}, \mathsf{ct}_{\mathfrak{s}_1}, \mathsf{PUB})$; `/* Algorithm 52 */`
`/*` $\overline{\overline{\mathsf{CT}_\mathfrak{s}}} \in \mathrm{GGSW}^{\mathscr{B},\ell}_{\boldsymbol{S}}(0)$ `if` $\mathsf{ct}_{\mathfrak{s}_1} + \mathsf{ct}_{\mathfrak{s}_2} \in \mathrm{LWE}_{\boldsymbol{s}}(0)$; $\overline{\overline{\mathsf{CT}_\mathfrak{s}}} \in \mathrm{GGSW}^{\mathscr{B},\ell}_{\boldsymbol{S}}(1)$ `otherwise. */`
4 $\overline{\overline{\mathsf{CT}_\mathfrak{s}}} = \mathbf{CBS}(\mathsf{ct}_{\mathfrak{s}_1} + \mathsf{ct}_{\mathfrak{s}_2}, \mathsf{PUB})$; `/* Algorithm 12 */`
5 $\mathbf{ct}_{\mathfrak{e}_{\mathsf{res}}} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}_\mathfrak{e}, \mathbf{ct}_{\mathfrak{e}_{\mathsf{sub}}}, \overline{\overline{\mathsf{CT}_\mathfrak{s}}}\right)$; `/* Algorithm 49 */`
6 $\mathbf{ct}_{\mathfrak{m}_{\mathsf{res}}} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}_{\mathfrak{m}_{\mathsf{add}}}, \mathbf{ct}_{\mathfrak{m}_{\mathsf{sub}}}, \overline{\overline{\mathsf{CT}_\mathfrak{s}}}\right)$; `/* Algorithm 49 */`
7 $\mathsf{ct}_{\mathfrak{s}_{\mathsf{res}}} \leftarrow \mathsf{ExtendedCMux}\left(\mathsf{ct}_{\mathfrak{s}_1}, \mathsf{ct}_{\mathfrak{s}_{\mathsf{sub}}}, \overline{\overline{\mathsf{CT}_\mathfrak{s}}}\right)$; `/* Algorithm 49 */`
8 **return** $\mathbf{ct}_{\mathfrak{f}_{\mathsf{res}}} = (\mathsf{ct}_{\mathfrak{s}_{\mathsf{res}}}, \mathbf{ct}_{e_{\mathsf{res}}}, \mathbf{ct}_{m_{\mathsf{res}}})$

Proof *(Correctness of Addition (Algorithm 53)). As defined in AlignMantissa (Algorithm 51), line 1 returns* $\mathfrak{e} = \max(\mathfrak{e}_1, \mathfrak{e}_2)$ *and* $\mathsf{ct}_{\mathfrak{m}'_1}$ *and* $\mathsf{ct}_{\mathfrak{m}'_2}$ *aligned (such that if* $\mathfrak{e}_1 \geq \mathfrak{e}_2$, $\mathfrak{m}'_1 = \mathfrak{m}_1$ *and* $\mathfrak{m}'_2 = \lfloor \mathfrak{m}_2 / 2^{\rho_\mathfrak{m} \cdot \gamma} \rfloor$ *with* $\gamma = \mathfrak{e}_1 - \mathfrak{e}_2$. *And if* $\mathfrak{e}_1 \leq \mathfrak{e}_2$, $\mathfrak{m}'_2 = \mathfrak{m}_2$ *and* $\mathfrak{m}'_1 = \lfloor \mathfrak{m}_1 / 2^{\rho_\mathfrak{m} \cdot \gamma} \rfloor$ *with* $\gamma = \mathfrak{e}_2 - \mathfrak{e}_1$*.).*

Line 2 adds the two aligned mantissas $\mathfrak{m}'_1$ *and* $\mathfrak{m}'_2$ *by adding together each* LWE *ciphertext. As the carry block is empty, these operations can be done directly.*

As defined in Algorithm 52, line 3 returns $\mathsf{ct}_{\mathfrak{m}_{\mathsf{sub}}} = |\mathfrak{m}'_1 - \mathfrak{m}'_2| \cdot 2^{\ell_\mathfrak{m} \cdot \alpha}$, $\mathfrak{s}_{\mathsf{sub}} = \mathfrak{s}_1$ *if* $\mathfrak{m}'_1 > \mathfrak{m}'_2$, *or* $\mathfrak{s}_{\mathsf{sub}} = \mathfrak{s}_2$ *if* $\mathfrak{m}'_1 < \mathfrak{m}'_2$. *Assuming* $\mathfrak{m}'_1 \neq \mathfrak{m}'_2$, *let* α *be the index of the first non zero block of* $\mathfrak{m}_{\mathsf{res}} = |\mathfrak{m}'_1 - \mathfrak{m}'_2| \cdot 2^{\ell_\mathfrak{m} \cdot \alpha}$, *then if* $\mathfrak{e} > \alpha$ *the algorithm returns* $\mathfrak{e}_{\mathsf{sub}} = \mathfrak{e} - \alpha$. *If* $\mathfrak{m}'_1 = \mathfrak{m}'_2$ *or* $\mathfrak{e} < \alpha$, $\mathfrak{m}_{sub} = 0$, $\mathfrak{e}_{\mathsf{sub}} = 0$ *and* $\mathfrak{s}_{\mathsf{sub}} = \mathfrak{s}_1$.

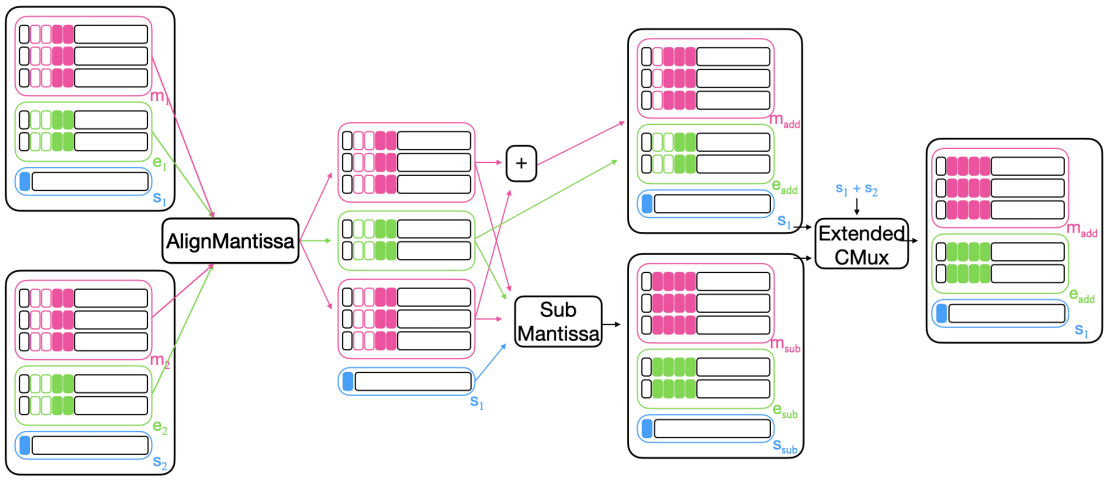

Figure 8.4: This figure gives an overview of each step needed to perform the homomorphic floating-point addition.

By adding the sign on line 4 and performing a **CBS** *over the result, we obtain a* GGSW *ciphertext which encrypt* $\mathfrak{s}_1 + \mathfrak{s}_2$. *So* $\overline{\overline{\mathsf{CT}}} \in \mathrm{GGSW}^{\mathbb{B},\ell}_{\boldsymbol{S}}(0)$ *if the two signs are equal or* $\overline{\overline{\mathsf{CT}}} \in \mathrm{GGSW}^{\mathbb{B},\ell}_{\boldsymbol{S}}(1)$ *if the signs are different.*

So with this GGSW *ciphertext, we will return the mantissa, the exponent and the sign corresponding to the addition if the signs are equal and if they are different, we return the mantissa, the exponent and the sign corresponding to the subtraction as proposed on the lemma 8.11.* □

Lemma 8.12. *(Noise Constraints of Algorithm 53) The output noise variances of ciphertexts of Algorithm 53,* $\mathbf{ct}_{\mathfrak{m}_{\mathsf{res}}}, \mathbf{ct}_{\mathfrak{e}_{\mathsf{res}}}$ *and* $\mathbf{ct}_{\mathfrak{s}_{\mathsf{res}}}$, *are respectively* $\max\left(2\sigma^2_{\mathsf{BR}} + (\rho_e \ell_e + 5)\cdot\sigma^2_{\mathsf{cmux}}, \sigma^2_{\mathsf{BR}} + (\ell_m + 1)\cdot\sigma^2_{\mathsf{cmux}}\right) + \sigma^2_{\mathsf{cmux}}, \sigma^2_{\mathsf{BR}} + 2\sigma^2_{\mathsf{cmux}}$ *and* $5\sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{cmux}}$.

To guarantee correctness of this operation, we need to find parameters that verify the following inequalities:

$$2\max\left((\ell_m - 1)\cdot\sigma^2_{\mathsf{cmux}}, \sigma^2_{\mathsf{BR}} + 2\cdot\sigma^2_{\mathsf{cmux}}\right) + \sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{KS}} + \sigma^2_{\mathsf{MS}} \leq t^2$$
$$\max\left(2\sigma^2_{\mathsf{BR}} + (\rho_e \ell_e + 5)\,\sigma^2_{\mathsf{cmux}}, \sigma^2_{\mathsf{BR}} + (\ell_m + 1)\,\sigma^2_{\mathsf{cmux}}\right) + \sigma^2_{\mathsf{cmux}} + \sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{KS}} + \sigma^2_{\mathsf{MS}} \leq t^2.$$

With $\sigma_{\mathsf{in},\mathfrak{e}}$ *the noise variance of the input exponent ciphertexts,* $\sigma_{\mathsf{in},\mathfrak{m}}$ *the noise variance of the input mantissa ciphertexts and with* σ_{BR} *the noise added by the blind rotation,* σ_{KS} *the noise added by the key switch,* σ_{cmux} *the noise added by the* CMux *and finally* σ_{MS}, *the noise added by the modulus switch and* t^2, *the noise bound as defined in the proof of the noise constraints of Algorithm 46.*

Proof *(Lemma 8.12). Let us assume that the inputs of this algorithm are the outputs of Algorithm 50. It means that the variances of* $\mathbf{ct}_{\mathfrak{s}_i}, \mathbf{ct}_{\mathfrak{e}_i}$ *and* $\mathbf{ct}_{\mathfrak{m}_i}$ *are respectively* $\sigma^2_{\mathsf{BR}}, \sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{cmux}}$ *and* $\sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{cmux}}$ *(see Lemma 8.6).*

The first line calls Algorithm 51, we use Lemma 8.8 to estimate the noise variances of $\mathbf{ct}_{\mathfrak{m}'_1}, \mathbf{ct}_{\mathfrak{m}'_2}$ *which are equal to* $\sigma^2_{\mathsf{BR}} + \left(\frac{\rho_e \ell_e + 5}{2}\right)\cdot\sigma^2_{\mathsf{cmux}}$ *and the noise variance of* $\mathbf{ct}_{\mathfrak{e}'}$ *which is equal to* $\sigma^2_{\mathsf{BR}} + 2\cdot\sigma^2_{\mathsf{cmux}}$. *Then,* $\mathbf{ct}_{\mathfrak{m}'_1}$ *and* $\mathbf{ct}_{\mathfrak{m}'_2}$ *are added which doubles the variance.*

Next, we call Algorithm 52 and using Lemma 8.10, we deduce that the noise variances of $\mathbf{ct}_{\mathfrak{m}_{\mathsf{sub}}}, \mathbf{ct}_{\mathfrak{e}_{\mathsf{sub}}}$ *and* $\mathbf{ct}_{\mathfrak{s}_{\mathsf{sub}}}$ *are respectively* $\sigma^2_{\mathsf{BR}} + (\ell_m + 1)\cdot\sigma^2_{\mathsf{cmux}}, \sigma^2_{\mathsf{BR}} + \sigma^2_{\mathsf{cmux}}$ *and* $5\sigma^2_{\mathsf{BR}}$.

Then, we have a circuit bootstrap which must satisfies the following constraint $2\sigma_{\mathsf{BR}}^2 + \sigma_{\mathsf{KS}}^2 + \sigma_{\mathsf{MS}}^2 \leq t^2$. *Finally, we have an extended* CMux *for the mantissa, the exponent and the sign. The noise variances of* $\mathbf{ct}_{\mathfrak{m}_{\mathsf{res}}}, \mathbf{ct}_{\mathfrak{e}_{\mathsf{res}}}$ *and* $\mathbf{ct}_{\mathfrak{s}_{\mathsf{res}}}$ *are respectively* $\max\left(2\sigma_{\mathsf{BR}}^2 + (\rho_e \ell_e + 5)\cdot\sigma_{\mathsf{cmux}}^2, \sigma_{\mathsf{BR}}^2 + (\ell_m+1)\cdot\sigma_{\mathsf{cmux}}^2\right) + \sigma_{\mathsf{cmux}}^2, \sigma_{\mathsf{BR}}^2 + 2\sigma_{\mathsf{cmux}}^2$ *and* $5\sigma_{\mathsf{BR}}^2 + \sigma_{\mathsf{cmux}}^2$.

Using Lemmas 8.6, 8.2, 8.8 and 8.10 and by noticing that some of the inequalities are dominated by others, we find the complete set of constraints. As we want to be able to chain several additions, we will assume that after the addition, we apply Algorithm 50. The non-dominated set of constraints is the following:

$$2\max\left((\ell_m-1)\cdot\sigma_{\mathsf{cmux}}^2, \sigma_{\mathsf{BR}}^2 + 2\cdot\sigma_{\mathsf{cmux}}^2\right) + \sigma_{\mathsf{BR}}^2 + \sigma_{\mathsf{KS}}^2 + \sigma_{\mathsf{MS}}^2 \leq t^2$$
$$\max\left(2\sigma_{\mathsf{BR}}^2 + (\rho_e\ell_e+5)\,\sigma_{\mathsf{cmux}}^2, \sigma_{\mathsf{BR}}^2 + (\ell_m+1)\,\sigma_{\mathsf{cmux}}^2\right) + \sigma_{\mathsf{cmux}}^2 + \sigma_{\mathsf{BR}}^2 + \sigma_{\mathsf{KS}}^2 + \sigma_{\mathsf{MS}}^2 \leq t^2.$$

We need to find parameters that verify these inequalities to guarantee correctness. □

8.5 Multiplication and Division

In this section, we introduce a very efficient floating-point multiplication with the HFP representation. Then, we detail a second algorithm to perform division. Finally, we briefly describe how to perform the ReLU and an approximated Sigmoid.

8.5.1 Multiplication

This operation computes the product of two HFPs. Following this procedure, it is necessary to apply the CarryPropagateFloat algorithm (Algorithm 50). This step ensures that the homomorphic floating-point number is returned to its standard representation, aligning it with the conventional HFP formalism. At a high level, the goal is to multiply the two mantissas without losing precision. Next, the exponent is updated by computing the sum of the two exponents and subtracting the bias. We note that we selected this bias to allow for efficient computation of this step. Due to the representation, we know that if the most significant ciphertext of the mantissa equals zero, or if the exponent is negative, the result of the operation is zero since we do not work with subnormal values. Otherwise, we return the result of the multiplication along with the updated exponent.

Lemma 8.13 (Multiplication (Algorithm 54))**.** *Let* $\mathbf{ct}_{\mathfrak{f}_i}$ *such that* $\mathsf{ct}_{\mathfrak{s}_i} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}_i)$, $\mathbf{ct}_{\mathfrak{e}_i} = \left[\mathsf{ct}_{\mathfrak{e}_{i,\ell_\mathfrak{e}-1}}, \ldots, \mathsf{ct}_{\mathfrak{e}_{i,0}}\right] \in \left[\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{i,\ell_\mathfrak{e}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{i,0})\right]$ *encrypting* $\mathfrak{e}_i < 2^{\rho_\mathfrak{e}}$ *and* $\mathbf{ct}_{\mathfrak{m}_i} = \left[\mathsf{ct}_{\mathfrak{m}_{i,\ell_\mathfrak{m}-1}}, \ldots, \mathsf{ct}_{\mathfrak{m}_{i,0}}\right] \in \left[\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{i,\ell_\mathfrak{m}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{i,0})\right]$ *encrypting* $\mathfrak{m}_i < 2^{\rho_\mathfrak{m}}$ *with* $i \in \{1,2\}$ *be two ciphertexts encrypting* $\mathfrak{f}_i = (-1)^{\mathfrak{s}_i}\cdot\mathfrak{m}_i\cdot(2^{\rho_\mathfrak{m}})^{\mathfrak{e}_i-\mathsf{bias}}$.

Then Algorithm 54 returns $(\mathbf{ct}_{\mathfrak{m}_{\mathsf{res}}}, \mathbf{ct}_{\mathfrak{e}_{\mathsf{res}}}, \mathsf{ct}_{\mathfrak{s}_{\mathsf{res}}}) = \mathbf{ct}_{\mathfrak{f}_{\mathsf{res}}}$ *such that* $\mathsf{DecryptFloat}(\mathbf{ct}_{\mathfrak{f}_{\mathsf{res}}}) = \mathfrak{f}_{\mathsf{res}} = (-1)^{\mathfrak{s}_{\mathsf{res}}}\cdot\mathfrak{m}_{\mathsf{res}}\cdot(2^{\rho_\mathfrak{m}})^{\mathfrak{e}_{\mathsf{res}}-\mathsf{bias}} = \mathfrak{f}_1\cdot\mathfrak{f}_2 + \epsilon$ *with* $\mathsf{ct}_{\mathfrak{s}_{\mathsf{res}}} = \mathsf{ct}_{\mathfrak{s}_1} + \mathsf{ct}_{\mathfrak{s}_2}$ *and* ϵ *is the maximum error added by the operation as express in Definition 35.*

If $\mathfrak{m}_1 \neq 0$ *and* $\mathfrak{m}_2 \neq 0$, *if* $\mathfrak{e}_1 + \mathfrak{e}_2 \geq \mathsf{bias} - \ell_\mathfrak{m} + 1$ *then* $\mathfrak{m}_{\mathsf{res}} = \left\lfloor \mathfrak{m}_1\cdot\mathfrak{m}_2/2^{(\ell_\mathfrak{m}-1)\cdot\rho_\mathfrak{m}}\right\rfloor$ *and* $\mathfrak{e}_{\mathsf{res}} = \mathfrak{e}_1 + \mathfrak{e}_2 - \mathsf{bias} + \ell_\mathfrak{m} - 1$ *with* $|\epsilon| < (2^{\rho_\mathfrak{m}})^{\mathfrak{e}_{\mathsf{res}}-\mathsf{bias}}$. *If* $\mathfrak{e}_1 + \mathfrak{e}_2 < \mathsf{bias} - \ell_\mathfrak{m} + 1$ *then* $\mathfrak{m}_{\mathsf{res}} = 0$ *and* $\mathfrak{e}_{\mathsf{res}} = 0$ *with* $|\epsilon| < 2^{\rho_\mathfrak{m}\cdot(\ell_\mathfrak{m}-1)}(2^{\rho_\mathfrak{m}})^{-\mathsf{bias}}$. *If* $\mathfrak{m}_1 = 0$ *or* $\mathfrak{m}_2 = 0$ *then* $\mathfrak{m}_{\mathsf{res}} = 0$ *and* $\mathfrak{e}_{\mathsf{res}} = 0$ *with* $|\epsilon| < 2^{\rho_\mathfrak{m}\cdot(\ell_\mathfrak{m}-1)}(2^{\rho_\mathfrak{m}})^{-\mathsf{bias}}$.

The complexity of the algorithm is:

$$\mathsf{Cost}_{\mathsf{Multiplication}}^{\ell_{\mathbf{PBS}},\ell_{\mathbf{CBS}},k,N,n,q,\ell_\mathfrak{m},} = cost_{\mathsf{RadixMul}}^{n,\ell_\mathfrak{m},\ell_{\mathbf{PBS}},k,N,q} + 2\cdot\left(\mathsf{Cost}_{\mathbf{PBS}}^{\ell,k,N,n,q} + \mathsf{Cost}_{\mathbf{KS}}^{\ell,n,k,N}\right) + \mathsf{Cost}_{\mathbf{CBS}}^{\ell_{\mathbf{PBS}},\ell_{\mathbf{CBS}},k,N,n,q}.$$

Proof *(Correctness of Multiplication (Algorithm 54)). In the first step of the operation, we compute the sign of the multiplication by summing the two signs* $\mathfrak{s}_1$ *and* $\mathfrak{s}_2$. *As the sign is on the padding bit, the addition is done modulo 2. If the signs are equal, after the operation, the sign is positive (*$\mathsf{ct}_{\mathfrak{s}_1} + \mathsf{ct}_{\mathfrak{s}_2} \in \mathrm{LWE}_{\boldsymbol{s}}(0)$*), otherwise the sign is negative (*$\mathsf{ct}_{\mathfrak{s}_1} + \mathsf{ct}_{\mathfrak{s}_2} \in \mathrm{LWE}_{\boldsymbol{s}}(1)$*).*

Algorithm 54: $\mathsf{CT}_{\mathfrak{f}_{\text{res}}} \leftarrow \mathsf{Multiplication}(\mathbf{ct}_{\mathfrak{f}_1}, \mathbf{ct}_{\mathfrak{f}_2}, \mathsf{PUB})$

Context: $\begin{cases} \mathsf{LUT}_{\mathfrak{m}}: & \text{LUT to return 1 if the value equals 0; 0 otherwise} \\ \mathsf{LUT}_{\mathfrak{e}}: & \text{LUT to return 0 if } \left(x \geq 2^{\rho_{\mathfrak{e}}-1}\right); \text{ 1 otherwise} \end{cases}$

Input:

$$\begin{cases} \mathbf{ct}_{\mathfrak{f}_1} = \begin{cases} \mathsf{ct}_{\mathfrak{s}_1} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}_1) \\ \mathbf{ct}_{\mathfrak{e}_1} = [\mathsf{ct}_{\mathfrak{e}_1,\ell_{\mathfrak{e}}-1}, \ldots, \mathsf{ct}_{\mathfrak{e}_1,0}] \in \left[\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_{1_{\ell_{\mathfrak{e}}-1}}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_{1_0}\right)\right] \\ \mathbf{ct}_{\mathfrak{m}_1} = [\mathsf{ct}_{\mathfrak{m}_1,\ell_{\mathfrak{m}}-1}, \ldots, \mathsf{ct}_{\mathfrak{m}_1,0}] \in \left[\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{1_{\ell_{\mathfrak{m}}-1}}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{1_0}\right)\right] \end{cases} \\ \mathbf{ct}_{\mathfrak{f}_2} = \begin{cases} \mathsf{ct}_{\mathfrak{s}_2} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}_2) \\ \mathbf{ct}_{\mathfrak{e}_2} = [\mathsf{ct}_{\mathfrak{e}_2,\ell_{\mathfrak{e}}-1}, \ldots, \mathsf{ct}_{\mathfrak{e}_2,0}] \in \left[\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_{2_{\ell_{\mathfrak{e}}-1}}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_{2_0}\right)\right] \\ \mathbf{ct}_{\mathfrak{m}_2} = [\mathsf{ct}_{\mathfrak{m}_2,\ell_{\mathfrak{m}}-1}, \ldots, \mathsf{ct}_{\mathfrak{m}_2,0}] \in \left[\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{2_{\ell_{\mathfrak{m}}-1}}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{2_0}\right)\right] \end{cases} \\ \mathsf{PUB}: \text{Public materials for } \mathbf{KS}, \mathbf{PBS} \text{ and } \mathbf{CBS}; \texttt{ /* Remark 2.15 */} \end{cases}$$

Output:

$$\begin{cases} \mathbf{ct}_{\mathfrak{f}_{\text{res}}} = \begin{cases} \mathsf{ct}_{\mathfrak{s}_{\text{res}}} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}_{\text{res}}) \\ \mathbf{ct}_{\mathfrak{e}_{\text{res}}} = [\mathsf{ct}_{\mathfrak{e}_{\text{res}},\ell_{\mathfrak{e}}-1}, \ldots, \mathsf{ct}_{\mathfrak{e}_{\text{res}},0}] \in \left[\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_{\text{res}_{\ell_{\mathfrak{e}}-1}}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{e}_{\text{res}_0}\right)\right] \\ \mathbf{ct}_{\mathfrak{m}_{\text{res}}} = [\mathsf{ct}_{\mathfrak{m}_{\text{res}},\ell_{\mathfrak{m}}-1}, \ldots, \mathsf{ct}_{\mathfrak{m}_{\text{res}},0}] \in \left[\mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{\text{res}_{\ell_{\mathfrak{m}}-1}}\right), \ldots, \mathrm{LWE}_{\boldsymbol{s}}\left(\mathfrak{m}_{\text{res}_0}\right)\right] \end{cases} \end{cases}$$

1 $\mathsf{ct}_{\mathfrak{s}_{\text{res}}} \leftarrow \mathsf{ct}_{\mathfrak{s}_1} + \mathsf{ct}_{\mathfrak{s}_2}$

2 $\mathbf{ct}_{\mathfrak{m}_{\text{mul}}} = [\mathsf{ct}_{\mathfrak{m}_{\text{mul}},2\ell_{\mathfrak{m}}-1}, \ldots \mathsf{ct}_{\mathfrak{m}_{\text{mul}},0}] \leftarrow \mathsf{RadixMul}(\mathbf{ct}_{\mathfrak{m}_1}, \mathbf{ct}_{\mathfrak{m}_2}, \mathsf{PUB})$; /* Algorithm 41 (modified) */

3 $\mathsf{ct}_{\mathfrak{m}_{\text{mul}},2\ell_{\mathfrak{m}}-2} \leftarrow \mathsf{ct}_{\mathfrak{m}_{\text{mul}},2\ell_{\mathfrak{m}}-1} \cdot 2^{\rho_{\mathfrak{m}}} + \mathsf{ct}_{\mathfrak{m}_{\text{mul}},2\ell_{\mathfrak{m}}-2}$

4 $\mathbf{ct}_{\mathfrak{m}_{\text{res}}} = [\mathsf{ct}_{\mathfrak{m}_{\text{res}},\ell_{\mathfrak{m}}-1}, \ldots, \mathsf{ct}_{\mathfrak{m}_{\text{res}},0}] \leftarrow [\mathsf{ct}_{\mathfrak{m}_{\text{mul}},2\ell_{\mathfrak{m}}-2}, \ldots \mathsf{ct}_{\mathfrak{m}_{\text{mul}},\ell_{\mathfrak{m}}-1}]$
/* $\mathsf{ct}_{\mathsf{tmp}_{\mathfrak{m}}} \in \mathrm{LWE}_{\boldsymbol{s}}(1)$ if $\mathsf{ct}_{\mathfrak{m}_{\text{res}},\ell_{\mathfrak{m}}-1} \in \mathrm{LWE}_{\boldsymbol{s}}(0)$; $\mathsf{ct}_{\mathsf{tmp}_{\mathfrak{e}}} \in \mathrm{LWE}_{\boldsymbol{s}}(0)$ otherwise */

5 $\mathsf{ct}_{\mathsf{tmp}_{\mathfrak{m}}} \leftarrow \mathbf{KS\text{-}PBS}(\mathsf{ct}_{\mathfrak{m}_{\text{res}},\ell_{\mathfrak{m}}-1}, \mathsf{LUT}_{\mathfrak{m}}, \mathsf{PUB})$; /* Algorithm 4 and 11 */

6 $\mathbf{ct}_{\mathfrak{e}_{\text{res}}} = [\mathsf{ct}_{\mathfrak{e}_{\text{res}},\ell_{\mathfrak{e}}-1}, \ldots \mathsf{ct}_{\mathfrak{e}_{\text{res}},0}] \leftarrow \mathsf{RadixAdd}(\mathbf{ct}_{\mathfrak{e}_1}, \mathbf{ct}_{\mathfrak{e}_2}, \mathsf{PUB})$; /* Algorithm 40 */

/* $\mathsf{ct}_{\mathsf{tmp}_{\mathfrak{e}}} \in \mathrm{LWE}_{\boldsymbol{s}}(0)$ if $\mathsf{ct}_{e_{\text{res}},\ell_{\mathfrak{e}}-1} \in \mathrm{LWE}_{\boldsymbol{s}}(x)$ with $x \geq 2^{\rho_{\mathfrak{e}}-1}$; $\mathsf{ct}_{\mathsf{tmp}_{\mathfrak{e}}} \in \mathrm{LWE}_{\boldsymbol{s}}(1)$ otherwise */

7 $\mathsf{ct}_{\mathsf{tmp}_{\mathfrak{e}}} \leftarrow \mathbf{KS\text{-}PBS}(\mathsf{ct}_{e_{\text{res}},\ell_{\mathfrak{e}}-1}, \mathsf{LUT}_{\mathfrak{e}}, \mathsf{PUB})$; /* Algorithm 4 and 11 */

/* bias have been choose such that bias $-\ell_{\mathfrak{m}}+1$ is equal to $2^{\ell_{\mathfrak{e}}\cdot\rho_{\mathfrak{e}}-1}$ so this subtraction can be done only on the most significant block */

8 $\mathsf{ct}_{\mathfrak{e}_{\text{res}},\ell_{\mathfrak{e}}-1} \leftarrow \mathsf{ct}_{\mathfrak{e}_{\text{res}},\ell_{\mathfrak{e}}-1} - \mathsf{TrivialEncrypt}(2^{\rho_{\mathfrak{e}}-1}, 1)$
/* $\mathsf{ct}_{\mathsf{tmp}} \in \mathrm{LWE}_{\boldsymbol{s}}(0)$ if $\mathfrak{e}$ is big enough and the most significant mantissa LWE is not null */

9 $\mathsf{ct}_{\mathsf{tmp}} \leftarrow \mathsf{ct}_{\mathsf{tmp}_{\mathfrak{m}}} + \mathsf{ct}_{\mathsf{tmp}_{\mathfrak{e}}}$
/* encrypt 0 if $\mathsf{ct}_{\mathsf{tmp}} \in \mathrm{LWE}_{\boldsymbol{s}}(0)$, 1 otherwise */

10 $\overline{\overline{\mathsf{CT}}} \leftarrow \mathbf{CBS}(\mathsf{ct}_{\mathsf{tmp}}, \mathsf{PUB})$; /* Algorithm 12 */

11 $\mathbf{ct}_{\mathfrak{e}_{\text{res}}} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}_{\mathfrak{e}_{\text{res}}}, \mathsf{TrivialEncrypt}(0, \ell_{\mathfrak{e}}), \overline{\overline{\mathsf{CT}}}\right)$; /* Algorithm 49 */

12 $\mathbf{ct}_{\mathfrak{m}_{\text{res}}} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}_{\mathfrak{m}_{\text{res}}}, \mathsf{TrivialEncrypt}(0, \ell_{\mathfrak{m}}), \overline{\overline{\mathsf{CT}}}\right)$; /* Algorithm 49 */

13 **return** $\mathbf{ct}_{\text{res}} = (\mathsf{ct}_{\mathfrak{s}_{\text{res}}}; \mathbf{ct}_{\mathfrak{e}_{\text{res}}}; \mathsf{CT}_{\mathfrak{m}_{\text{res}}})$

Next, we compute the multiplication of the two mantissas. RadixMul *return the product of two integers. If* $\mathfrak{m}_1 \neq 0$ *and* $\mathfrak{m}_2 \neq 0$*, we have* $\mathfrak{m}_i \in \left[2^{(\ell_\mathfrak{m}-1)\cdot\rho_\mathfrak{m}}, 2^{\ell_\mathfrak{m}\cdot\rho_\mathfrak{m}}-1\right]$ *for* $i \in \{0,1\}$*, so* $\mathfrak{m}_1 \cdot \mathfrak{m}_2 \in \left[2^{2\cdot(\ell_\mathfrak{m}-1)\cdot\rho_\mathfrak{m}}, 2^{2\cdot\ell_\mathfrak{m}\cdot\rho_\mathfrak{m}} - 2^{\ell_\mathfrak{m}\cdot\rho_\mathfrak{m}+1}+1\right]$ *(note that the value is stored in* $2\cdot\ell_\mathfrak{m}$ *blocks). As we want to keep the classical representation of the mantissa (*$\ell_\mathfrak{m}$ *blocks, where the most significant block is non-zero except when the result equals zero), we will remove the least significant blocks, ensuring that only the* $\ell_\mathfrak{m}$ *most significant blocks remain.*

To do so, we distinguish two cases after the multiplication. The case where $\mathfrak{m}_1 \cdot \mathfrak{m}_2 \in \left[2^{2\cdot(\ell_\mathfrak{m}-1)\cdot\rho_\mathfrak{m}}, 2^{(2\cdot\ell_\mathfrak{m}-1)\cdot\rho_\mathfrak{m}}\right)$ *(the most significant block after the multiplication contains zero) and the case where* $\mathfrak{m}_1 \cdot \mathfrak{m}_2 \in \left[2^{(2\cdot\ell_\mathfrak{m}-1)\cdot\rho_\mathfrak{m}}, 2^{2\cdot\ell_\mathfrak{m}\cdot\rho_\mathfrak{m}} - 2^{\ell_\mathfrak{m}\cdot\rho_\mathfrak{m}+1}+1\right]$*. During the operation* RadixMul*, the carry buffer of each ciphertext in* $\mathbf{ct}_{\mathfrak{m}_{\mathsf{mul}}}$ *is emptied. As the carry buffer of* $\mathsf{ct}_{\mathfrak{m}_{\mathsf{mul}},2\ell_\mathfrak{m}-1}$ *and* $\mathsf{ct}_{\mathfrak{m}_{\mathsf{mul}},2\ell_\mathfrak{m}-2}$ *are empty, by multiplying* $\mathsf{ct}_{\mathfrak{m}_{\mathsf{mul}},2\ell_\mathfrak{m}-1}$ *by* $2^{\rho_\mathfrak{m}}$*, we have* $\mathsf{ct}_{\mathfrak{m}_{\mathsf{mul}},2\ell_\mathfrak{m}-1}\cdot 2^{\rho_\mathfrak{m}} \in \{0\} \cup \left[2^{\rho_\mathfrak{m}}, 2^{2\cdot\rho_\mathfrak{m}}\right)$*. As* $\mathsf{ct}_{\mathfrak{m}_{\mathsf{mul}},2\ell_\mathfrak{m}-2} \in [0, 2^{\rho_\mathfrak{m}})$*, we can sum these two values such that* $\mathsf{ct}_{\mathfrak{m}_{\mathsf{mul}},2\ell_\mathfrak{m}-1} = \mathsf{ct}_{\mathfrak{m}_{\mathsf{mul}},2\ell_\mathfrak{m}-1}\cdot 2^{\rho_\mathfrak{m}} + \mathsf{ct}_{\mathfrak{m}_{\mathsf{mul}},2\ell_\mathfrak{m}-2}$ *(Line 3). We have now* $\mathfrak{m}_1 \cdot \mathfrak{m}_2 \in \left[2^{2\cdot(\ell_\mathfrak{m}-1)\cdot\rho_\mathfrak{m}}, 2^{2\cdot\ell_\mathfrak{m}\cdot\rho_\mathfrak{m}} - 2^{\ell_\mathfrak{m}\cdot\rho_\mathfrak{m}+1}+1\right]$ *stored in* $2\cdot\ell_\mathfrak{m}-1$ *blocks. Now, we remove the* $\ell_\mathfrak{m}-1$ *less significant block of the multiplication which represents too small values for the mantissa precision and the most significant block which have its information already stored in the second most significant block* $\mathsf{ct}_{\mathfrak{m}_{\mathsf{mul}},2\ell_\mathfrak{m}-1}$*. We obtain* $\mathfrak{m}_{\mathsf{res}} = \left\lfloor \mathfrak{m}_1\cdot\mathfrak{m}_2/2^{(\ell_\mathfrak{m}-1)\cdot\rho_\mathfrak{m}}\right\rfloor \in \left[2^{(\ell_\mathfrak{m}-1)\cdot\rho_\mathfrak{m}}, 2^{(\ell_\mathfrak{m}+1)\cdot\rho_\mathfrak{m}}\right)$ *(Line 4). (In practice, we have modified the algorithm* RadixMul *such that the carry propagation of the most significant block is not done and such that the useless parts of the multiplication are not computed (Proof 13)). In the special case where* $\mathfrak{m}_1 = 0$ *or* $\mathfrak{m}_2 = 0$*, the previous step has no impact and* RadixMul *will return the value* 0 *on each block. To distinguish the two cases, line 5, a* **PBS** *checks if the most significant block of the mantissa is equal to zero, such that* $\mathsf{ct}_{\mathsf{tmp}_\mathfrak{m}}$ *is in* $\mathrm{LWE}_{\boldsymbol{s}}(1)$ *if* $\mathsf{ct}_{\mathfrak{m}_{\mathsf{res}},\ell_\mathfrak{m}-1}$ *is in* $\mathrm{LWE}_{\boldsymbol{s}}(0)$*, otherwise it returns* $\mathsf{ct}_{\mathsf{tmp}_\mathfrak{m}}$ *in* $\mathrm{LWE}_{\boldsymbol{s}}(0)$*.*

Now, we need to update the exponent $\mathfrak{e}$*. First we will sum the two exponents (Line 6). Next, as the exponent has the shape* $\mathfrak{e}'_i + \mathsf{bias}$ *after the sum we obtain* $\mathfrak{e}_1 + \mathfrak{e}_2 = \mathfrak{e}_{\mathsf{res}} = \mathfrak{e}'_{\mathsf{res}} + 2\cdot\mathsf{bias}$*. So to keep the same representation, we need to remove one* bias*. Moreover, in the previous step, we have removed the* $\ell_\mathfrak{m}-1$ *blocks of the mantissa, so we need to add* $\ell_\mathfrak{m}-1$ *to the exponent. In Section 8.2.2, we chose the bias such that* $\mathsf{bias}-\ell_\mathfrak{m}+1 = 2^{\ell_\mathfrak{e}\cdot\rho_\mathfrak{e}-1}$ *so we only need to subtract* $2^{\ell_\mathfrak{e}\cdot\rho_\mathfrak{e}-1}$ *from the sum of the exponents. Line 4, a* **PBS** *checks if the exponent is big enough and return an* LWE *ciphertext such that* $\mathsf{ct}_{\mathsf{tmp}_\mathfrak{e}} \in \mathrm{LWE}_{\boldsymbol{s}}(0)$ *if* $\mathsf{ct}_{\mathfrak{e}_{\mathsf{res}},\ell_\mathfrak{e}-1}$ *is in* $\mathrm{LWE}_{\boldsymbol{s}}(x)$ *with* $x \geq 2^{\rho_\mathfrak{e}-1}$*, otherwise it returns* $\mathsf{ct}_{\mathsf{tmp}_\mathfrak{e}} \in \mathrm{LWE}_{\boldsymbol{s}}(1)$*. We can now subtract* $2^{\ell_\mathfrak{e}\cdot\rho_\mathfrak{e}-1}$ *from the sum of the exponent (Line 8). This operation impacts only the most significant block of the exponent and can be performed directly.*

Now, looking at the two previous control LWE *ciphertexts (*$\mathsf{ct}_{\mathsf{tmp}_\mathfrak{m}}$ *and* $\mathsf{ct}_{\mathsf{tmp}_\mathfrak{e}}$*), by summing these two values, we obtain* $\mathsf{ct}_{\mathsf{tmp}_\mathfrak{m}} + \mathsf{ct}_{\mathsf{tmp}_\mathfrak{e}} \in \mathrm{LWE}_{\boldsymbol{s}}(0)$ *if* $\mathsf{ct}_{\mathfrak{m}_{\mathsf{res}},\ell_\mathfrak{m}-1} \notin \mathrm{LWE}_{\boldsymbol{s}}(0)$ *and* $\mathsf{ct}_{\mathfrak{e}_{\mathsf{res}},\ell_\mathfrak{e}-1} \in \mathrm{LWE}_{\boldsymbol{s}}(x)$ *with* $x \geq 2^{\rho_\mathfrak{e}-1}$*. Otherwise, one of the two conditions to perform the multiplication is unmet and the multiplication is not feasible. By using a circuit bootstrapping, we obtain a* GGSW *ciphertext* $\overline{\overline{\mathsf{CT}}}$ *such that* $\overline{\overline{\mathsf{CT}}} \in \mathrm{GGSW}^{\mathcal{B},\ell}_{\boldsymbol{S}}(0)$ *if we can perform the multiplication and* $\overline{\overline{\mathsf{CT}}} \in \mathrm{GGSW}^{\mathcal{B},\ell}_{\boldsymbol{S}}(1)$ *otherwise. With the last 2 lines, if* $\overline{\overline{\mathsf{CT}}}$ *is in* $\mathrm{GGSW}^{\mathcal{B},\ell}_{\boldsymbol{S}}(0)$*, the algorithm returns the result of the multiplication, otherwise the multiplication is not doable and it returns zero.* □

Lemma 8.14 ($\mathsf{Cost}^{n,\ell_\mathfrak{m},\ell_{\mathbf{PBS}},k,N,q}_{\mathsf{RadixMul}}$)**.** *To simplify the algorithm, the operation* RadixMul *proposed in the algorithm, have a complexity which can be bound by* $\left(2\cdot{\ell_\mathfrak{m}}^2 + {\ell_\mathfrak{m}}^2/\rho_\mathfrak{m}\right)\cdot\left(\mathsf{Cost}^{\ell,k,N,n,q}_{\mathbf{PBS}} + \mathsf{Cost}^{\ell,n,k,N}_{\mathbf{KS}}\right)$*. However, in our implementation, we use a slight modification of this algorithm to remove unnecessary computations (the part of the multiplication which does not appear in the final mantissa). In practice, the complexity is bounded by* $\left(2\cdot(\ell_\mathfrak{m}/2+1)^2 + \ell_\mathfrak{m}\cdot(\ell_\mathfrak{m}/2+1)/\rho_\mathfrak{m}\right)\cdot\left(\mathsf{Cost}^{\ell,k,N,n,q}_{\mathbf{PBS}} + \mathsf{Cost}^{\ell,n,k,N}_{\mathbf{KS}}\right)$*.*

Proof *(Lemma 8.14). The mantissa* $\mathfrak{m}$ *represents a value in* $\left[2^{\rho_\mathfrak{m}\cdot(\ell_\mathfrak{m}-1)}, 2^{\rho_\mathfrak{m}\cdot\ell_\mathfrak{m}}-1\right]$*. So* $\mathfrak{m}^2$ *is in* $\left[2^{2\cdot\rho_\mathfrak{m}\cdot(\ell_\mathfrak{m}-1)}, 2^{2\rho_\mathfrak{m}\cdot\ell_\mathfrak{m}} - 2^{\rho_\mathfrak{m}\cdot\ell_\mathfrak{m}+1}+1\right]$*. After a multiplication, the smallest reachable value is* $2^{2\cdot\rho_\mathfrak{m}\cdot(\ell_\mathfrak{m}-1)}$ *and as we keep only the* $\ell_\mathfrak{m}$ *most significant blocks, all the values smaller than* $2^{2\cdot\rho_\mathfrak{m}\cdot(\ell_\mathfrak{m}-1)-\ell_\mathfrak{m}\cdot\rho_\mathfrak{m}} = 2^{\rho_\mathfrak{m}\cdot\ell_\mathfrak{m}-2\rho_\mathfrak{m}}$ *are lost so we don't need to compute them. These values correspond to the part of the mantissa* $\mathfrak{m}'$ *such that* $\mathfrak{m}'^2 < 2^{\rho_\mathfrak{m}\cdot\ell_\mathfrak{m}-2\rho_\mathfrak{m}}$ *so the part* $\mathfrak{m}'$ *such that*

$\mathfrak{m}' < 2^{\rho_\mathfrak{m} \cdot (\ell_\mathfrak{m} - 2)/2}$. *The part of the mantissa* $\mathfrak{m}'$ *corresponds to the* $\ell_\mathfrak{m}/2 - 1$ *least significant blocks of the mantissa* $\mathfrak{m}$. *So we only need to compute the multiplication on the* $\ell_\mathfrak{m}/2 + 1$ *most significant blocks.* □

8.5.2 Division

This operation computes the division of two HFPs. Following this procedure, it is necessary to apply the CarryPropagateFloat algorithm (Algorithm 50). This step will ensure that the homomorphic floating-point number is returned to its standard representation, aligning it with the conventional HFP formalism.

Lemma 8.15 (Division (Algorithm 55))**.** *Let* $\mathbf{ct}_{\mathfrak{f}_i}$ *such that* $\mathsf{ct}_{\mathfrak{s}_i} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}_i)$, $\mathbf{ct}_{\mathfrak{e}_i} = \left[\mathsf{ct}_{\mathfrak{e}_{i,\ell_\mathfrak{e}-1}}, \ldots, \mathsf{ct}_{\mathfrak{e}_{i,0}}\right] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{i,\ell_\mathfrak{e}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{i,0})]$ *encrypting* $\mathfrak{e} < 2^{\rho_\mathfrak{e}}$ *and* $\mathbf{ct}_{\mathfrak{m}_i} = \left[\mathsf{ct}_{\mathfrak{m}_{i,\ell_\mathfrak{m}-1}}, \ldots, \mathsf{ct}_{\mathfrak{m}_{i,0}}\right] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{i,\ell_\mathfrak{m}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{i,0})]$ *encrypting* $\mathfrak{m} < 2^{\rho_\mathfrak{m}}$ *with* $i \in \{1, 2\}$ *be two ciphertexts encrypting* $\mathfrak{f}_i = (-1)^{\mathfrak{s}_i} \cdot \mathfrak{m}_i \cdot (2^{\rho_\mathfrak{m}})^{\mathfrak{e}_i - \mathsf{bias}}$.

Then Algorithm 55 returns $(\mathbf{ct}_{\mathfrak{m}_\mathsf{res}}, \mathbf{ct}_{\mathfrak{e}_\mathsf{res}}, \mathsf{ct}_{\mathfrak{s}_\mathsf{res}}) = \mathbf{ct}_{\mathfrak{f}_\mathsf{res}}$ *such that* $\mathsf{DecryptFloat}(\mathbf{ct}_{\mathfrak{f}_\mathsf{res}}) = \mathfrak{f}_\mathsf{res} = (-1)^{\mathfrak{s}_\mathsf{res}} \cdot \mathfrak{m}_\mathsf{res} \cdot (2^{\rho_\mathfrak{m}})^{\mathfrak{e}_\mathsf{res} - \mathsf{bias}} = \mathfrak{f}_1/\mathfrak{f}_2 + \epsilon$ *with* $|\epsilon| < (2^{\rho_\mathfrak{m}})^{\mathfrak{e}_\mathsf{res} - \mathsf{bias}}$ *s.t.* $\mathsf{ct}_{\mathfrak{s}_\mathsf{res}} = \mathsf{ct}_{\mathfrak{s}_1} + \mathsf{ct}_{\mathfrak{s}_2}$. *If* $\mathsf{ct}_{\mathfrak{e}_2} < \mathsf{ct}_{\mathfrak{e}_1} + \mathsf{bias} + \ell_\mathfrak{m} - 1$, *then* $\mathsf{ct}_{\mathfrak{m}_\mathsf{res}} = \left\lfloor \mathsf{ct}_{\mathfrak{m}_1} \cdot 2^{\ell_\mathfrak{m} - 1} / \mathsf{ct}_{\mathfrak{m}_2} \right\rfloor$ *and* $\mathsf{ct}_{\mathfrak{e}_\mathsf{res}} = \mathsf{ct}_{\mathfrak{e}_1} + \mathsf{bias} + \ell_\mathfrak{m} - 1 - \mathsf{ct}_{\mathfrak{e}_2}$. *Else,* $\mathsf{ct}_{\mathfrak{e}_\mathsf{res}} = 0$ *and* $\mathsf{ct}_{\mathfrak{m}_\mathsf{res}} = 0$.
The complexity of the algorithm is:

$$\mathsf{Cost}^{\ell_{\mathbf{PBS}}, \ell_{\mathbf{CBS}}, k, N, n, q, \ell_\mathfrak{e},}_{\mathsf{Division}} = \mathsf{Cost}^{n, \ell_\mathfrak{e}, \ell_{\mathbf{PBS}}, k, N, q}_{\mathsf{RadixMul}} + \mathsf{Cost}^{2 \cdot \ell_\mathfrak{m}}_{\mathsf{IntDiv}} + \mathsf{Cost}^{\ell_{\mathbf{PBS}}, \ell_{\mathbf{CBS}}, k, N, n, q}_{\mathbf{CBS}}.$$

Algorithm 55: $\mathbf{ct}_\mathfrak{f} \leftarrow \mathsf{Division}(\mathbf{ct}_{\mathfrak{f}_1}, \mathbf{ct}_{\mathfrak{f}_2}, \mathsf{PUB})$

Input:
$\mathbf{ct}_{\mathfrak{f}_1} = \begin{cases} \mathsf{ct}_{\mathfrak{s}_1} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}_1) \\ \mathbf{ct}_{\mathfrak{e}_1} = \left[\mathsf{ct}_{\mathfrak{e}_{1,\ell_\mathfrak{e}-1}}, \ldots, \mathsf{ct}_{\mathfrak{e}_{1,0}}\right] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{1,\ell_\mathfrak{e}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{1,0})] \\ \mathbf{ct}_{\mathfrak{m}_1} = \left[\mathsf{ct}_{\mathfrak{m}_{1,\ell_\mathfrak{m}-1}}, \ldots, \mathsf{ct}_{\mathfrak{m}_{1,0}}\right] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{1,\ell_\mathfrak{m}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{1,0})] \end{cases}$
$\mathbf{ct}_{\mathfrak{f}_2} = \begin{cases} \mathsf{ct}_{\mathfrak{s}_2} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}_2) \\ \mathbf{ct}_{\mathfrak{e}_2} = \left[\mathsf{ct}_{\mathfrak{e}_{2,\ell_\mathfrak{e}-1}}, \ldots, \mathsf{ct}_{\mathfrak{e}_{2,0}}\right] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{2,\ell_\mathfrak{e}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{2,0})] \\ \mathbf{ct}_{\mathfrak{m}_2} = \left[\mathsf{ct}_{\mathfrak{m}_{2,\ell_\mathfrak{m}-1}}, \ldots, \mathsf{ct}_{\mathfrak{m}_{2,0}}\right] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{2,\ell_\mathfrak{m}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{2,0})] \end{cases}$
PUB : Public materials for **KS**, **PBS** and **CBS**; `/* Remark 2.15 */`

Output:
$\mathbf{ct}_{\mathfrak{f}_\mathsf{res}} = \begin{cases} \mathsf{ct}_{\mathfrak{s}_\mathsf{res}} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}_\mathsf{res}) \\ \mathbf{ct}_{\mathfrak{e}_\mathsf{res}} = [\mathsf{ct}_{\mathfrak{e}_\mathsf{res}, \ell_\mathfrak{e}-1}, \ldots, \mathsf{ct}_{\mathfrak{e}_\mathsf{res}, 0}] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{\mathsf{res}_{\ell_\mathfrak{e}-1}}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{\mathsf{res}_0})] \\ \mathbf{ct}_{\mathfrak{m}_\mathsf{res}} = [\mathsf{ct}_{\mathfrak{m}_\mathsf{res}, \ell_\mathfrak{m}-1}, \ldots, \mathsf{ct}_{\mathfrak{m}_\mathsf{res}, 0}] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{\mathsf{res}_{\ell_\mathfrak{m}-1}}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{\mathsf{res}_0})] \end{cases}$

1 $\mathsf{ct}_{\mathfrak{s}_\mathsf{res}} \leftarrow \mathsf{ct}_{\mathfrak{s}_1} + \mathsf{ct}_{\mathfrak{s}_2}$
2 $\mathsf{ct}_{\mathfrak{e}_1} \leftarrow \mathsf{Add}(\mathsf{ct}_{\mathfrak{e}_1}, \mathsf{TrivialEncrypt}(\mathsf{bias} + \ell_\mathfrak{m} - 1, \ell_\mathfrak{e}))$; `/* Algorithm 40 */`
3 $(\mathsf{ct}_{\mathfrak{e}_\mathfrak{s}}, \mathbf{ct}_\mathfrak{e} = [\mathsf{ct}_{\mathfrak{e}, \ell_\mathfrak{e}}, \ldots, \mathsf{ct}_{\mathfrak{e},0}]) \leftarrow \mathsf{RadixSub}^*(\mathbf{ct}_{\mathfrak{e}_1}, \mathbf{ct}_{\mathfrak{e}_2}, \mathsf{PUB})$; `/* Algorithm 46 */`
4 $\mathsf{ct}_{\mathfrak{m}_1} = [\mathsf{ct}_{\mathfrak{m}_1, \ell_\mathfrak{m}-1}, \ldots, \mathsf{ct}_{\mathfrak{m}_1,0}, \mathsf{ct}_0, \ldots, \mathsf{ct}_0] \leftarrow \mathsf{ct}_{\mathfrak{m}_1} || \mathsf{TrivialEncrypt}(0, \ell_\mathfrak{m} - 1)$
5 $\mathsf{ct}_{\mathfrak{m}_2} = [\mathsf{ct}_0, \ldots, \mathsf{ct}_0, \mathsf{ct}_{\mathfrak{m}_2, \ell_\mathfrak{m}-1}, \ldots, \mathsf{ct}_{\mathfrak{m}_2,0}] \leftarrow \mathsf{TrivialEncrypt}(0, \ell_\mathfrak{m} - 1) || \mathsf{ct}_{\mathfrak{m}_2}$
6 $\mathsf{ct}_{\mathfrak{m}_\mathsf{div}} = [\mathsf{ct}_{\mathfrak{m}_\mathsf{div}, 2 \ldots \ell_\mathfrak{m}-1}, \ldots, \mathsf{ct}_{\mathfrak{m}_\mathsf{div},0}] \leftarrow \mathsf{RadixDiv}(\mathsf{ct}_{\mathfrak{m}_1}, \mathsf{ct}_{\mathfrak{m}_2}, \mathsf{PUB})$
7 $\mathsf{ct}_{\mathfrak{m}_\mathsf{div}, 2\ell_\mathfrak{m}-2} \leftarrow \mathsf{ct}_{\mathfrak{m}_\mathsf{div}, 2\ell_\mathfrak{m}-1} \ldots 2^{\rho_\mathfrak{m}} + \mathsf{ct}_{\mathfrak{m}_\mathsf{div}, 2\ell_\mathfrak{m}-2}$
8 $\mathbf{ct}_{\mathfrak{m}_\mathsf{res}} = [\mathsf{ct}_{\mathfrak{m}_\mathsf{res}, \ell_\mathfrak{m}-1}, \ldots \mathsf{ct}_{\mathfrak{m}_\mathsf{res}, 0}] \leftarrow [\mathsf{ct}_{\mathfrak{m}_\mathsf{div}, 2\ell_\mathfrak{m}-2}, \ldots \mathsf{ct}_{\mathfrak{m}_\mathsf{div}, \ell_\mathfrak{m}-1}]$
9 $\overline{\overline{\mathsf{CT}}}_{\mathfrak{e}_\mathfrak{s}} \leftarrow \mathbf{CBS}(\mathsf{ct}_{\mathfrak{e}_\mathfrak{s}}, \mathsf{PUB})$; `/* Algorithm 12 */`
10 $\mathbf{ct}_{\mathfrak{e}_\mathsf{res}} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}_\mathfrak{e}, \mathsf{TrivialEncrypt}(0, \ell_\mathfrak{e}), \overline{\overline{\mathsf{CT}}}_{\mathfrak{e}_\mathfrak{s}}\right)$; `/* Algorithm 49 */`
11 $\mathbf{ct}_{\mathfrak{m}_\mathsf{res}} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}_{\mathfrak{m}_\mathsf{res}}, \mathsf{TrivialEncrypt}(0, \ell_\mathfrak{m}), \overline{\overline{\mathsf{CT}}}_{\mathfrak{e}_\mathfrak{s}}\right)$; `/* Algorithm 49 */`
12 **return** $\mathbf{ct}_\mathfrak{f} = (\mathsf{ct}_{\mathfrak{s}_\mathsf{res}}, \mathbf{ct}_{\mathfrak{e}_\mathsf{res}}, \mathbf{ct}_{\mathfrak{m}_\mathsf{res}})$

Proof *(Correctness of Division (Algorithm 55)). On the first line, we compute the output sign. If the signs are equal, the sign after the operation is positive (*$\mathsf{ct}_{\mathfrak{s}_1} + \mathsf{ct}_{\mathfrak{s}_2} \in \mathrm{LWE}_{\boldsymbol{s}}(0)$*), otherwise the sign is negative (*$\mathsf{ct}_{\mathfrak{s}_1} + \mathsf{ct}_{\mathfrak{s}_2} \in \mathrm{LWE}_{\boldsymbol{s}}(1)$*).*

The algorithm RadixDiv *takes two vectors of ciphertexts that represent two integers and returns the quotient of the division. In our context, we can not divide directly the mantissas. In fact, the mantissas are in the same interval and very close, if we were to divide them directly, the quotient would only be a value of a few digits. In our case, we want a value in the interval* $\left[2^{\rho_\mathfrak{m}\cdot(\ell_\mathfrak{m}-1)}, 2^{\rho_\mathfrak{m}\cdot\ell_\mathfrak{m}}\right)$*. To get a result in the right interval, we add some blocks encrypting zeros after the blocks of the first mantissa* $\mathfrak{m}_1$ *(Line 4). Adding the zeros to* $\mathbf{ct}_{\mathfrak{m}_1}$ *corresponds to compute* $\mathfrak{m}_1 \cdot 2^{\ell_\mathfrak{m}\cdot\rho_\mathfrak{m}} \in \left[2^{\rho_\mathfrak{m}\cdot(2\ell_\mathfrak{m}-1)}, 2^{\rho_\mathfrak{m}\cdot 2\ell_\mathfrak{m}}\right)$*. Now if we divide* $\mathfrak{m}_1 \cdot 2^{\ell_\mathfrak{m}\cdot\rho_\mathfrak{m}}$ *by* $\mathfrak{m}_2$*, we will obtain a result in* $\left[2^{(\ell_\mathfrak{m}-1)\cdot\rho_\mathfrak{m}}, 2^{(\ell_\mathfrak{m}+1)\cdot\rho_\mathfrak{m}}\right)$*. As explained in Proof 13, this value can be stored in* $\ell_\mathfrak{m}$ *blocks if we use the carry buffer of the most significant block. The carry buffer will be later cleaned during the call to CarryPropagateFloat (Algorithm 50). So after the shift of the mantissa (Line 4) and the division (Line 6), we obtain a new mantissa in the interval* $\left[2^{(\ell_\mathfrak{m}-1)\cdot\rho_\mathfrak{m}}, 2^{(\ell_\mathfrak{m}+1)\cdot\rho_\mathfrak{m}}\right)$*.*

After the division of the mantissa, we need to update the exponent. To do so, we need to subtract (Line 3) the two exponents, then add the bias and finally add the number of trivial ciphertexts added in Line 4. If the subtraction of the exponent returns a negative result (Line 3, $\mathsf{ct}_{\mathfrak{e}_s} \in \mathrm{LWE}_\mathbf{s}(0)$*), the division can not be done. In this case, Algorithm 55 returns the value zero (Line 10 and 11), otherwise it returns the result of the division of the two floating-point numbers.* □

8.6 More Features over HFP

This section extends our approach to cover a wider range of practical applications by adding the support of special values and efficient approximate functions.

8.6.1 Managing Special Values

In the classical floating-point arithmetic, when a value overflows the highest bound of the exponent, this value can not be represented anymore. In this case, the floating-point reaches the infinity. As previously described, our algorithms do not manage the values plus/minus infinity and Not a Number (NaN), but as we show now, they can easily be extended to do so. The idea is to add two encrypted Booleans to represent $+\infty$ and $-\infty$ such that: if only one is set, then it means that we have reached plus (resp., minus) infinity; if both are set, the value is interpreted as NaN. During an operation, the infinity value is reached as soon as an overflow occurs on the exponent, i.e., if the carry of the most significant block of the exponent is not empty. This check is done by computing a simple **PBS** on $\mathsf{ct}_{\mathsf{res},\ell_\mathfrak{e}-1}$, which returns a flag encrypting 0 if the carry is empty, or 1 otherwise. This flag is then given as an additional input to the CarryPropagateFloat (Alg. 50). Then, by computing a **CBS** on the sign, this will return the correct sign of the flag, i.e., plus or minus infinity (or 0). This process ends with the computation of a simple CMux tree which will properly update Booleans ciphertext depending on the flag value. Regarding the support of special values, the overhead is linear in the number of blocks composing the HFP. Thus, in comparison with the numbers of **PBS** or **CBS** needed to perform operations without any special values, the overhead should be negligible.

8.6.2 Computing Function Approximations

Beyond the arithmetic operations, floating-point numbers are particularly convenient to compute approximations of complex functions, via the Taylor series. A Taylor series of a real function $f(x)$ that is infinitely differentiable at a real number a is the power series $\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$, where $n!$ denotes the factorial of n and $f^{(n)}(a)$ denotes the n^{th} derivative of f evaluated at the point a. When $a = 0$, this is called a Maclaurin series and takes the form $\sum_{n=0}^{\infty} \frac{f^{(n)}(0)}{n!}x^n$. The Maclaurin method is advantageous when working with homomorphic floating-point numbers, given that the value of $f(a)$ is not known. Computing such a series is a direct application of our method, as each of its term can be computed using the previously defined arithmetic operators. Another advantage is that the needed precision and the computational time can be adjusted to fit the use case, i.e., by changing the

value of n. For instance, we have practically computed $\sin(x) \approx \sum_{n=0}^{2} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!}$ and $\cos(x) \approx \sum_{n=0}^{2} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!}$, which gives good results for values of $x \in [-1, 1]$. In Table 8.3, we present numerical values obtained from our approximations of the cosine and sine functions using the Maclaurin series.

	$\mathsf{Cos}(0.9636989235877991)$	$\mathsf{Sin}(0.41880202293395996)$
Exact value (64 bits)	0.5704859425112639	0.4066663011129846
Approximate value (64 bits)	**0.5715802311897278**	**0.4066667483866177**
Approximate value (32 bits)	**0.57158023**	**0.4066667**6
This work (32bits)	**0.5715802311897278**	**0.4066667**5567626953

Table 8.3: Result obtain for the cosines and sinus with Maclaurin series. The bold digits are the one which are equal to the digits of the approximate result of the clear double precision value.

8.6.3 More Operations

With the homomorphic floating-point representation, we can efficiently support usual functions used in machine learning. To evaluate the ReLU function, we can apply a circuit bootstrapping on the sign $\mathfrak{s}$ and return either the input floating-point or an encryption of zero using a CMux. The complete algorithm is detailed in Algorithm 56 in Section 8.6.3.

In the same manner, we can evaluate an approximate sigmoid function that returns the identity for values in the interval $[-1, 1]$, and returns the constant value 1 or -1 otherwise. More details are provided in Algorithm 57 in Section 8.6.3. To be closer to the classical sigmoid, we can combine this approximate sigmoid with the Maclaurin series introduced in Section 8.6.2. Other classical operations like the minimum, the maximum or the equality between two values can be easily performed on homomorphic floating-point numbers.

ReLU. A Rectified Linear Unit (ReLU) is an activation function widely used in neural networks. It computes the function $f(x) = \max(0, x)$ for $x \in \mathbb{R}$.

Lemma 8.16 (ReLU (Algorithm 56))**.** *Let* $\mathbf{ct}_{\mathfrak{f}}$ *such that* $\mathsf{ct}_{\mathfrak{s}} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s})$,$\mathbf{ct}_{\mathfrak{e}} = \left[\mathsf{ct}_{\mathfrak{e}_{\ell_{\mathfrak{e}}-1}}, \ldots, \mathsf{ct}_{\mathfrak{e}_0}\right] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{\ell_{\mathfrak{e}}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_0)]$ *encrypting* $\mathfrak{e} < 2^{\rho_{\mathfrak{e}}}$ *and* $\mathbf{ct}_{\mathfrak{m}} = \left[\mathsf{ct}_{\mathfrak{m}_{\ell_{\mathfrak{m}}-1}}, \ldots, \mathsf{ct}_{\mathfrak{m}_0}\right] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{\ell_{\mathfrak{m}}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_0)]$ *encrypting* $\mathfrak{m} < 2^{\rho_{\mathfrak{m}}}$ *be the ciphertext encrypting* $\mathfrak{f} = (-1)^{\mathfrak{s}} \cdot \mathfrak{m} \cdot (2^{\rho_{\mathfrak{m}}})^{\mathfrak{e}-\mathsf{bias}}$.

Then Algorithm 56 returns $(\mathbf{ct}_{\mathfrak{m}_{\mathsf{res}}}, \mathbf{ct}_{\mathfrak{e}_{\mathsf{res}}}, \mathsf{ct}_{\mathfrak{s}_{\mathsf{res}}}) = \mathbf{ct}_{\mathfrak{f}_{\mathsf{res}}}$ *such that* $\mathsf{DecryptFloat}(\mathbf{ct}_{\mathfrak{f}_{\mathsf{res}}}) = \mathfrak{f}_{\mathsf{res}}$ *with* $\mathfrak{f}_{\mathsf{res}} = (-1)^{\mathfrak{s}_{\mathsf{res}}} \cdot \mathfrak{m}_{\mathsf{res}} \cdot (2^{\rho_{\mathfrak{m}}})^{\mathfrak{e}_{\mathsf{res}}-\mathsf{bias}}$. *If* $\mathsf{ct}_{\mathfrak{s}} \in \mathrm{LWE}_{\boldsymbol{s}}(0)$, $\mathsf{ct}_{\mathfrak{s}_{\mathsf{res}}} = \mathsf{ct}_{\mathfrak{s}}$, $\mathsf{ct}_{\mathfrak{e}_{\mathsf{res}}} = \mathsf{ct}_{\mathfrak{e}}$ *and* $\mathsf{ct}_{\mathfrak{m}_{\mathsf{res}}} = \mathsf{ct}_{\mathfrak{m}}$. *Else* $\mathbf{ct}_{\mathfrak{f}_{\mathsf{res}}}$ *encrypt zero.*
The complexity of the algorithm is: $\mathsf{Cost}_{\mathsf{ReLU}}^{\ell_{\mathbf{PBS}},\ell_{\mathbf{CBS}},k,N,n,q} = \mathsf{Cost}_{\mathbf{CBS}}^{\ell_{\mathbf{PBS}},\ell_{\mathbf{CBS}},k,N,n,q}$.

Proof *(ReLU (Algorithm 56)). First we use a* **CBS** *on the sign to obtain a* GGSW *ciphertext such that* $\overline{\overline{\mathsf{CT}}} \in \mathrm{GGSW}_{\boldsymbol{S}}^{\mathbb{B},\ell}(0)$ *if* $\mathsf{ct}_{\mathfrak{s}}$ *is in* $\mathrm{LWE}_{\boldsymbol{s}}(0)$*, otherwise,* $\overline{\overline{\mathsf{CT}}}$ *is in* $\mathrm{GGSW}_{\boldsymbol{S}}^{\mathbb{B},\ell}(1)$

Next with the GGSW *ciphertext, we return the input ciphertext if the sign is positive otherwise we return zero.*

□

Approximate Sigmoid. An efficient algorithm to compute an approximation of the sigmoid function compatible with the HFP representation is presented in Algorithm 57.

8.7 Experimental Results

In this section, we demonstrate the practicability of our results by providing all cryptographic parameters, encodings, and both sequential and parallel timings.

Algorithm 56: $\mathbf{ct}_{\mathfrak{f}} \leftarrow \mathsf{ReLU}(\mathbf{ct}_{\mathfrak{f}}, \mathsf{PUB})$

Input:
$$\begin{cases} \mathbf{ct}_{\mathfrak{f}} = \begin{cases} \mathsf{ct}_{\mathfrak{s}} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}) \\ \mathbf{ct}_{\mathfrak{e}} = [\mathsf{ct}_{\mathfrak{e},\ell_{\mathfrak{e}}-1}, \ldots, \mathsf{ct}_{\mathfrak{e},0}] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{\ell_{\mathfrak{e}}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_0)] \\ \mathbf{ct}_{\mathfrak{m}} = [\mathsf{ct}_{\mathfrak{m},\ell_{\mathfrak{m}}-1}, \ldots, \mathsf{ct}_{\mathfrak{m},0}] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}\ell_{\mathfrak{m}}-1), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_0)] \end{cases} \\ \mathsf{PUB}: \text{Public materials for } \mathbf{KS}, \mathbf{PBS} \text{ and } \mathbf{CBS}; \texttt{ /* Remark 2.15 */} \end{cases}$$

Output:
$$\begin{cases} \mathbf{ct}_{\mathfrak{f}} = \begin{cases} \mathsf{ct}_{\mathfrak{s}} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}_{\mathsf{res}}) \\ \mathbf{ct}_{\mathfrak{e}_{\mathsf{res}}} = [\mathsf{ct}_{\mathfrak{e}_{\mathsf{res}},\ell_{\mathfrak{e}}-1}, \ldots, \mathsf{ct}_{\mathfrak{e}_{\mathsf{res}},0}] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{\mathsf{res},\ell_{\mathfrak{e}}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{\mathsf{res},0})] \\ \mathbf{ct}_{\mathfrak{m}_{\mathsf{res}}} = \left[\mathsf{ct}_{\mathfrak{m}_{\mathsf{res}},\ell_{\mathfrak{m}}-1}, \ldots, \mathsf{ct}_{\mathfrak{m},\mathsf{res},0}\right] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}\mathsf{res}, \ell_{\mathfrak{m}}-1), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_{\mathsf{res},0})] \end{cases} \end{cases}$$

`/* encrypt 0 if` $sign == 0$`, 1 otherwise */`

1 $\overline{\overline{\mathsf{CT}}} \leftarrow \mathbf{CBS}(\mathsf{ct}_{\mathfrak{s}}, \mathsf{PUB})$; `/* Algorithm 12 */`

2 $\mathbf{ct}_{\mathfrak{e}_{\mathsf{res}}} \leftarrow \mathsf{ExtendedCMux}(\mathbf{ct}_{\mathfrak{e}}, \mathsf{TrivialEncrypt}(0, \ell_{\mathfrak{e}}), \overline{\overline{\mathsf{CT}}})$; `/* Algorithm 49 */`

3 $\mathbf{ct}_{\mathfrak{m}_{\mathsf{res}}} \leftarrow \mathsf{ExtendedCMux}(\mathbf{ct}_{\mathfrak{m}}, \mathsf{TrivialEncrypt}(0, \ell_{\mathfrak{m}}), \overline{\overline{\mathsf{CT}}})$; `/* Algorithm 49 */`

4 **return** $\mathbf{ct}_{\mathfrak{f}} = [\mathsf{ct}_{\mathfrak{s}}, \mathbf{ct}_{\mathfrak{m}_{\mathsf{res}}}, \mathbf{ct}_{\mathfrak{e}_{\mathsf{res}}}]$

Encodings. In Table 8.4, we describe the different encodings used to represent 64, 32, 16 and 8 bits floating-point numbers (Sec. 8.3.2) in the homomorphic world. Chapter 6 and 7 indicates that a 4-bit precision message leads to the best precision-cost ratio; therefore, we focus on representations with $\rho_{\mathfrak{m}} = \rho_{\mathfrak{e}} = 2$. However, variations with $\rho_{\mathfrak{m}} \neq \rho_{\mathfrak{e}}$ may yield better timings depending on the use case. Additionally, in Table 8.4, we give the encoding for the TFHE-minifloats encoded over 8 bits as detailed in Sec. 8.3.1. For the TFHE-Minifloats, the value of the bias does not impact the timings and can be freely chosen.

Parameter Selection. In Lemma 8.12, we found two noise constraints that the parameters must satisfy in order to guarantee the correctness of Algorithm 53. We applied the same reasoning to Algorithm 54 and found that all the additional noise constraints are dominated by the constraints introduced in Lemma 8.12. It means that parameters that satisfy the constraints of Algorithm 53 will also satisfy the constraints of Algorithm 54.

As explained in Lemmas 8.11 and 8.13, the number of **PBS** in each algorithm is different and this has an impact on the failure probability of each algorithm. We followed the methodology presented in Chapter 2, Section 2.5, to compute the individual **PBS** failure probability using the number of dominant **PBS** in each algorithm. Using the parameters presented in Table 8.5, the maximal failure probability for the homomorphic addition and for the homomorphic multiplication are respectively $2^{-13.9}$ and $2^{-12.8}$ (note that the failure probability of one **KS-PBS** is smaller than 2^{-40}). We tested these parameters on a chain of a hundred operations on random inputs with random operations without detecting any errors due to the noise, only errors due to floating-point approximations.

Timings. All of our experiments have been carried out on AWS with a m6i.metal instance Intel Xeon 8375C (Ice Lake) at 3.5 GHz, with 128 vCPUs and 512.0 GiB of memory using the TFHE-rs library [Zam22]. Our code is available here[1]. In Table 8.6, we give the timings in seconds for all the arithmetic operations (i.e., add, sub, mul, div) and the ReLU and Sigmoid functions. Both sequential and parallel timings are given when possible (e.g., the division over integers is only implemented in parallel in TFHE-rs). Note that all arithmetic operations are followed by a carry propagation, which is obviously taken into account in the timings.

Finally, in Table 8.7, we present the timings for the **WoP-PBS** based approach. Although the multiplication timings are quite similar between HFP and minifloats, the addition operations perform significantly better using the **WoP-PBS**. This means that when computing with 8-bit

[1] `https://github.com/zama-ai/tfhe-rs/tree/artifact_tches_2025`

Algorithm 57: $\mathbf{ct}_{\mathfrak{f}} \leftarrow \mathsf{ApproxSigmoid}(\mathbf{ct}_{\mathfrak{f}}, \mathsf{PUB})$

Context: $\begin{cases} \mathsf{LUT}_{\mathfrak{m}}: & \text{LUT to return 1 if the value equals 1, 1 otherwise} \\ \mathsf{LUT}_{\mathfrak{e}}: & \text{LUT to return 0 if } \left(x >= 2^{\rho_{\mathfrak{e}}-1}\right); \text{ 1 otherwise} \end{cases}$

Input: $\begin{cases} \mathbf{ct}_{\mathfrak{f}} = \begin{cases} \mathsf{ct}_{\mathfrak{s}} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}) \\ \mathbf{ct}_{\mathfrak{e}} = [\mathsf{ct}_{\mathfrak{e},\ell_{\mathfrak{e}}-1}, \ldots, \mathsf{ct}_{\mathfrak{e},0}] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{\ell_{\mathfrak{e}}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_0)] \\ \mathbf{ct}_{\mathfrak{m}} = [\mathsf{ct}_{\mathfrak{m},\ell_{\mathfrak{m}}-1}, \ldots, \mathsf{ct}_{\mathfrak{m},0}] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}\ell_{\mathfrak{m}}-1), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_0)] \end{cases} \\ \mathsf{PUB}: \text{Public materials for } \mathbf{KS}, \mathbf{PBS} \text{ and } \mathbf{CBS}; \texttt{/* Remark 2.15 */} \end{cases}$

Output: $\begin{cases} \mathbf{ct}_{\mathfrak{f}} = \begin{cases} \mathsf{ct}_{\mathfrak{s}} \in \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{s}) \\ \mathbf{ct}_{\mathfrak{e}} = [\mathsf{ct}_{\mathfrak{e},\ell_{\mathfrak{e}}-1}, \ldots, \mathsf{ct}_{\mathfrak{e},0}] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_{\ell_{\mathfrak{e}}-1}), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{e}_0)] \\ \mathbf{ct}_{\mathfrak{m}} = [\mathsf{ct}_{\mathfrak{m},\ell_{\mathfrak{m}}-1}, \ldots, \mathsf{ct}_{\mathfrak{m},0}] \in [\mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}\ell_{\mathfrak{m}}-1), \ldots, \mathrm{LWE}_{\boldsymbol{s}}(\mathfrak{m}_0)] \end{cases} \end{cases}$

$\mathsf{ct}_1 = [\mathbf{ct}_{\mathfrak{e}_1}, \mathbf{ct}_{\mathfrak{m}_1}, \mathsf{ct}_{\mathfrak{s}_1}] \leftarrow \mathsf{TrivialEncryptFloat}(1)$

$\mathsf{ct}_{-1} = [\mathbf{ct}_{\mathfrak{e}_{-1}}, \mathbf{ct}_{\mathfrak{m}_{-1}}, \mathsf{ct}_{\mathfrak{s}_{-1}}] \leftarrow \mathsf{TrivialEncryptFloat}(-1)$

$\overline{\overline{\mathsf{CT}}}_{\mathfrak{s}} \leftarrow \mathbf{CBS}(\mathsf{ct}_{\mathfrak{s}}, \mathsf{PUB})$; `/* encrypt 0 if` $sign == 0$`, 1 otherwise */`

$\mathsf{ct}_{\mathfrak{s}_{\mathsf{tmp}}} \leftarrow \mathsf{ExtendedCMux}\left(\mathsf{ct}_{\mathfrak{s}_1}, \mathsf{ct}_{\mathfrak{s}_{-1}}, \overline{\overline{\mathsf{CT}}}_{\mathfrak{s}}\right)$; `/* Algorithm 49 */`

$\mathbf{ct}_{\mathfrak{e}_{\mathsf{tmp}}} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}_{\mathfrak{e}_1}, \mathbf{ct}_{\mathfrak{e}_{-1}}, \overline{\overline{\mathsf{CT}}}_{\mathfrak{s}}\right)$; `/* Algorithm 49 */`

$\mathbf{ct}_{\mathfrak{m}_{\mathsf{tmp}}} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}_{\mathfrak{m}_1}, \mathbf{ct}_{\mathfrak{m}_{-1}}, \overline{\overline{\mathsf{CT}}}_{\mathfrak{s}}\right)$; `/* Algorithm 49 */`

`/*` $\mathsf{ct}_{\mathfrak{e}_{\ell_{\mathfrak{e}}-1}} \in \mathrm{LWE}_{\boldsymbol{s}}(x)$ `with` $x < 2^{\rho_{\mathfrak{e}}-1}$ `then` $\mathrm{LWE}_{\boldsymbol{s}}(1)$ `else` $\mathrm{LWE}_{\boldsymbol{s}}(0)$ `*/`

$\mathsf{ct}_{\mathsf{tmp}_{\mathfrak{e}}} \leftarrow \mathbf{KS\text{-}PBS}\left(\mathsf{ct}_{\mathfrak{e}_{\ell_{\mathfrak{e}}-1}}, \mathsf{PUB}, \mathsf{LUT}_{\mathfrak{e}}\right)$; `/* Algorithm 4 and 11 */`

`/*` $\mathsf{ct}_{\mathfrak{m}_{\ell_{\mathfrak{m}}-1}} \in \mathrm{LWE}_{\boldsymbol{s}}(x)$ `with` $x < 2^{\rho_{\mathfrak{m}}-1}$ `then` $\mathrm{LWE}_{\boldsymbol{s}}(1)$ `else` $\mathrm{LWE}_{\boldsymbol{s}}(0)$ `*/`

$\mathsf{ct}_{\mathsf{tmp}_{\mathfrak{m}}} \leftarrow \mathbf{KS\text{-}PBS}\left(\mathsf{ct}_{\mathfrak{m}_{\ell_{\mathfrak{m}}-1}}, \mathsf{PUB}, \mathsf{LUT}_{\mathfrak{m}}\right)$; `/* Algorithm 4 and 11 */`

$\mathsf{ct}_{\mathsf{tmp}} \leftarrow \mathsf{ct}_{\mathsf{tmp}_{\mathfrak{m}}} + \mathsf{ct}_{\mathsf{tmp}_{\mathfrak{e}}}$; `/* tmp equals zero only if` $|\mathbf{Dec}(\mathbf{ct}_{\mathfrak{f}})| > 1$ `*/`

`/* encrypt 0 if tmp == 0, 1 otherwise */`

$\overline{\overline{\mathsf{CT}}}_{\mathsf{tmp}} \leftarrow \mathbf{CBS}(\mathsf{ct}_{\mathsf{tmp}}, \mathsf{PUB})$; `/* Algorithm 12 */`

$\mathsf{ct}_{\mathfrak{s}} \leftarrow \mathsf{ExtendedCMux}\left(\mathsf{ct}_{\mathfrak{s}}, \mathsf{ct}_{\mathfrak{s}_{\mathsf{tmp}}}, \overline{\overline{\mathsf{CT}}}_{\mathsf{tmp}}\right)$; `/* Algorithm 49 */`

$\mathbf{ct}_{\mathfrak{e}} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}_{\mathfrak{e}}, \mathbf{ct}_{\mathfrak{e}_{\mathsf{tmp}}}, \overline{\overline{\mathsf{CT}}}_{\mathsf{tmp}}\right)$; `/* Algorithm 49 */`

$\mathbf{ct}_{\mathfrak{m}} \leftarrow \mathsf{ExtendedCMux}\left(\mathbf{ct}_{\mathfrak{m}}, \mathbf{ct}_{\mathfrak{m}_{\mathsf{tmp}}}, \overline{\overline{\mathsf{CT}}}_{\mathsf{tmp}}\right)$; `/* Algorithm 49 */`

return $\mathbf{ct}_{\mathfrak{f}} = [\mathsf{ct}_{\mathfrak{s}}, \mathbf{ct}_{\mathfrak{m}}, \mathbf{ct}_{\mathfrak{e}}]$

floats, using the minifloats is generally better. Any larger precision requires to run the HFP method, whose timings show that many circuits based on floats can be practically evaluated for the first time. Only the division operation cannot be considered as practical. Note that the division is suffering from the slowness of the division over the integers, and not really from the approach described in here.

In the previous table, we present benchmarks obtained with a failure probability around 2^{-14}. To better evaluate our new algorithms, we also include benchmarks of the addition and the multiplication with a failure probability of around 2^{-40} (see Table 8.8). We observe that reducing the failure probability has only a minor impact on execution time which proves that our contribution scales well with small failure probabilities.

	$\ell_{\mathfrak{m}}$	$\rho_{\mathfrak{m}}$	$\ell_{\mathfrak{e}}$	$\rho_{\mathfrak{e}}$	bias
TFHE_FP$_{64b}$	27	2	5	2	539
TFHE_FP$_{32b}$	13	2	4	2	140
TFHE_FP$_{16b}$	6	2	3	2	37
TFHE_FP$_{8b}$	3	2	2	2	10
TFHE-Minifloat$^{\rho=4}$	3	∅	4	∅	8
TFHE-Minifloat$^{\rho=2}$	3	∅	4	∅	8

Table 8.4: Encodings for HFP and Minifloats.

	LWE parameters		GLWE parameters			PBS parameters		CBS parameters		LWE-to-LWE KS		Packing KS	
	n	$\log_2(\sigma_n)$	N	$\log_2(\sigma_N)$	k	Base	Level	Base	Level	Base	Level	Base	Level
TFHE_FP$_{64b}$	736	−16.59	1024	−51.49	2	12	3	8	2	1	14	17	2
TFHE_FP$_{32b}$	720	−16.17	1024	−51.49	2	12	3	8	2	1	15	17	2
TFHE_FP$_{16b}$	728	−16.38	1024	−51.49	2	15	2	6	3	1	14	17	2
TFHE_FP$_{8b}$	720	−16.17	1024	−51.49	2	15	2	6	3	1	14	13	2
TFHE-Minifloat^4	592	−12.77	1024	−51.49	2	9	4	14	1	2	5	17	2
TFHE-Minifloat^2	564	−12.02	1024	−51.49	2	12	3	13	1	2	5	17	2

Table 8.5: Homomorphic floating-point parameters used for the 8 bits, 16 bits, 32 bits and 64 bits equivalent representation (See section 8.3.2).

		Addition (Alg.53 & 50)	Multiplication (Alg.54 & 50)	Division (Alg.55 & 50)	Sigmoid (Alg.57)	ReLU (Alg.56)
TFHE_FP$_{64b}$	Sequential	12.32 s	87.15 s	∅	0.342 s	0.122 s
	Parallel	3.98 s	2.26 s	39.75 s	∅	∅
TFHE_FP$_{32b}$	Sequential	7.10 s	20.57 s	∅	0.342 s	0.120 s
	Parallel	2.50 s	1.03 s	15.18 s	∅	∅
TFHE_FP$_{16b}$	Sequential	3.89 s	3.83 s	∅	0.361 s	0.155 s
	Parallel	1.52 s	0.558 s	4.34 s	∅	∅
TFHE_FP$_{8b}$	Sequential	2.21 s	1.19 s	∅	0.388 s	0.153 s
	Parallel	1.13 s	0.444 s	1.76 s	∅	∅

Table 8.6: Timings of the HFP depending on the precision.

	TFHE-Minifloat$^{\rho=4}$	TFHE-Minifloat$^{\rho=2}$
Bivariate Operation (e.g., add, mul, ...)	1.2819 s	0.9957 s

Table 8.7: Timings for the 8 bit representations of the TFHE-Minifloat.

		Add	**Mul**
TFHE_FP$_{32b}$	Sequential	7.49s	23.2s
TFHE_FP$_{32b}$	Parallelized	2.83s	1.24s
TFHE_FP$_{64b}$	Sequential	13.3s	98.2s
TFHE_FP$_{64b}$	Parallelized	4.28s	2.75s

Table 8.8: Performance for Addition and Multiplication using pfail $\leq 2^{-40}$.

Chapter 9

Conclusion

In the previous chapters, we explored several topics aimed at improving multiple aspects of TFHE by proposing solutions to the limitations introduced in Section 2.6, with the goal of addressing the question raised in the introduction:

How can TFHE be leveraged to homomorphically compute primitive data types, bridging the gap between encrypted and plaintext computation?

To answer this question and address the identified limitations, we have investigated improvements to existing algorithms, new data representations, and novel algorithms throughout this manuscript. In TFHE, the PBS is among the most computationally expensive algorithms, despite its efficiency for small precisions. Nevertheless, it remains a central operation, as nearly every circuit relies on it. For efficiency reasons, the PBS is generally not executed alone; instead, TFHE employs the atomic pattern introduced in [CJP21]. This pattern comprises dot products, key switching, and bootstrapping (Figure 2.3), and improving any of these steps results in a global performance gain for the TFHE scheme.

After presenting TFHE, its limitations, and the state of the art in the first part of the manuscript, the second part introduces several methods for enhancing TFHE. We focus on improving core operations, including the PBS (Chapter 4), as well as key-switching and parameter selection (Chapter 5). In the final chapter of this part, Chapter 6, we present a novel algorithm that addresses multiple limitations identified in Section 2.6 and is employed extensively in the third part of the manuscript. Chapter 4 introduces a novel methodology for optimizing bootstrapping by minimizing the number of necessary CMux operations to execute the algorithm. This new technique relies primarily on the bootstrapping method proposed in [LY23], which consists in evaluating each step of the bootstrapping over several small polynomials, resulting in multiple CMux operations per step instead of a single CMux operation over large polynomials. Our technique involves sorting the mask elements to reveal patterns during the blind rotation. These patterns help identify CMux operations that do not affect the final result, allowing us to safely eliminate them. This method enhances bootstrapping efficiency and offers speed-ups, particularly for high-precision computations. Based on this first improvement, we also investigated a modification of the modulus switching algorithm, aimed at increasing the number of operations that can be safely skipped during bootstrapping without affecting the final result. Using this new technique, we achieve a speedup of up to 2.4× compared to the PBS from [LY23], and up to 8.2× compared to the classical PBS. Finally, we presented a method to highly parallelize the bootstrapping proposed in [LY23] as well as our technique. This technique has not been intensively explored yet, as it requires dedicated hardware such as FPGAs or GPUs and will be studied later as part of future work.

Then, Chapter 5 presented another angle of research by introducing two new distributions for secret keys: partial secret keys and secret keys with shared randomness. The first one, the partial secret key, is used in GLWE ciphertexts, especially when the polynomial size becomes too

large. This new secret key, with known zero coefficients, allows us to introduce new algorithms that take advantage of this representation. In addition to these new algorithms, reducing the number of unknown coefficients in the secret key helps to reduce noise propagation. The second key distribution, the secret key with shared randomness, as the name suggests, allows coefficients to be shared between different secret keys. With this representation, during a key switch, only the key elements that are not shared need to be switched. Moreover, this structure enables performing the key switch in multiple steps, allowing a trade-off between efficiency and noise growth. As with the partial secret key, this structure and the resulting algorithms provide improvements in both noise growth and execution time. Finally, we evaluated the combined use of both key types and conducted a detailed analysis of the noise propagation, leading to a reduction in the execution time of the CJP atomic pattern by up to 58%.

Chapter 6 present a new without padding PBS. This WoP-PBS is based on several known building blocks: the bootstrapping, the circuit bootstrapping, and the Vertical Packing. This technique consists in first extracting all the bits of all ciphertexts, then converting all the resulting LWE ciphertexts encrypting bits into GGSW ciphertexts, and finally executing a CMux tree and a blind rotation using these GGSW ciphertexts. This new method allows the efficient evaluation of multi-input operations and enables the use of large lookup tables, making it possible to perform computations with high-precision representations as presented in the next chapter. This technique permits to efficiently addresses multiple TFHE limitations within a single operation and, through a detailed study, we demonstrate that this technique outperforms previously proposed solutions for large precisions. In the following two chapters, this algorithm is intensively used to evaluate multivariate operations over multiple input ciphertexts.

Chapters 7 and 8 constitute the third part of the manuscript. In this part, we focus on representing primitive data types, such as 32-bit or 64-bit integers and floating-points. In, Chapter 7 presents different techniques to efficiently represent large homomorphic integers. To work with TFHE and represent large integers, one solution consists of splitting a large integer into several smaller ones using a radix decomposition, a CRT decomposition, or a hybrid approach combining both. These smaller integers are then encoded in multiple ciphertexts, taking advantage of the message/carry encoding. This chapter then studies how to efficiently perform operations on these ciphertexts, while using the carry to perform leveled operations before bootstrapping or to perform multivariate operations. This chapter presents one of the first practical applications of the WoP-PBS algorithm, introduced in the previous chapter. It demonstrates how this technique can be applied to operate directly on CRT or hybrid based representations, enabling bootstrapping that simultaneously processes all ciphertexts involved in the representation.

Based on previous chapters, Chapter 8 proposes two efficient techniques to represent floating point numbers, each with its own pros and cons.

The first technique is based on the WoP-PBS described in Chapter 6. In this approach, the three components of a floating point are encoded across one or more ciphertexts, with all operations performed using the WoP-PBS technique. This solution is highly efficient for small precision but becomes inefficient as the precision increases. The performances of this method are directly tied to the in-depth study of the algorithm.

The second technique is based on the radix representation introduced in Chapter 7. First, each part of a floating-point (sign, mantissa, and exponent) is split into three distinct integers, and each integer is then decomposed using radix representation, according to the methodology presented in the previous chapter. The objective is to design new algorithms capable of efficiently working with this representation. In particular, we need algorithms that interact effectively across the different parts of an encoded floating-point. We also studied the noise propagation and parameter selection to ensure efficiency, security, and a relatively low failure probability.

Each solution to the limitations proposed in these different chapters has been studied independently, but many of them can be easily combined, resulting in a global improvement of the TFHE

scheme in terms of efficiency, latency, and message precision. All these contributions make the scheme significantly more practical, helping to bridge the gap between homomorphic encryption and conventional plaintext computation.

In addition to this research work, all the solutions were implemented and tested in the TFHE-rs library [Zam22]. Many of the solutions introduced in this manuscript are now public and used every day by the users of the library.

Future Works. As presented in Chapter 2 and 3, bootstrapping is a well known operation and many studies have been conducted to improve this algorithm. Even though a lot of research has already been done, recent works such as the bootstrapping proposed in [LMK+23], the one proposed by [LY23], and the follow-up enhancements presented in Chapter 4, show that there is still room for major advancements. As studied in this manuscript, solutions can arise from new methods of parallelization, improved techniques for performing blind rotation, or rethinking the bootstrapping process itself, such as with the WoP-PBS presented in Chapter 6. Even though the PBS is the slowest algorithm, improving other algorithms, such as the key switch, can also contribute to overall performance gains. Another topic explored in Chapter 5 is the size of the public material. Over time, and especially with the growing need to reduce the failure probability of algorithms, the size of the public material has increased, often reaching several gigabits. Reducing this size while maintaining both the security and performance of the algorithms is a highly challenging yet promising research direction.

Our constructions, presented in Chapters 7 and 8, rely on standard algorithms such as bootstrapping (Algorithm 11) and circuit bootstrapping (Algorithm 12). Employing more refined homomorphic techniques, like those introduced in Chapter 4, or in recent works such as [LMK+23] for bootstrapping and [WWL+24] for circuit bootstrapping, could yield significant performance improvements. Additional gains may also be achieved by optimizing the underlying arithmetic of these homomorphic operations. All these techniques need to be studied more in details to make this new constructions even faster.

This manuscript focuses solely on TFHE, but we have seen that many FHE schemes are lattice-based cryptosystems and all based on the LWE assumption and its derivatives. Thus, it is interesting to study recent improvements from other schemes to determine if they can be adapted to TFHE. For instance, it could be interesting to bootstrap several ciphertexts with a better amortized cost, such as the PBS proposed with CKKS. We note that already several work have been done in this sens such as [MS18, GP25, LW23]. Similarly, the techniques proposed for TFHE could potentially be transferred to other FHE schemes, leading to broader advancements across the field.

Finally, practical fully homomorphic encryption is a relatively young field, and many breakthrough improvements may emerge in the coming decades, helping to bridge the gap between the clear world and the encrypted world, leading to massive adoption of FHE. These breakthroughs can arise from various sources such as new algorithms, new lattice-based schemes, novel assumptions, or dedicated hardware. Currently, even modest amelioration can spark new lines of thinking and eventually lead to major breakthroughs. All the work presented in this manuscript has already paved the way for new research directions and has been presented at conferences and published in journals. I hope that the advancements achieved during this thesis, along with my future contributions, will continue to inspire further research and contribute to greater advancements. For this reason, I intend to continue working on FHE, making new enhancements, and contributing to the collective effort to make it as practical as possible.

Bibliography

[ABW10] Benny Applebaum, Boaz Barak, and Avi Wigderson. Public-key cryptography from different assumptions. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 171–180, 2010.

[ACPS09] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In *Advances in Cryptology-CRYPTO 2009: 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, pages 595–618. Springer, 2009.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange-a new hope. In *USENIX security symposium*, volume 2016, 2016.

[AGVW17] Martin R Albrecht, Florian Göpfert, Fernando Virdia, and Thomas Wunderer. Revisiting the expected cost of solving usvp and applications to lwe. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, pages 297–322. Springer, 2017.

[Ajt96] Miklós Ajtai. Generating hard instances of lattice problems. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 99–108, 1996.

[Alb17] Martin R Albrecht. On dual lattice attacks against small-secret lwe and parameter choices in helib and seal. In *Advances in Cryptology–EUROCRYPT 2017: 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30–May 4, 2017, Proceedings, Part II*, pages 103–129. Springer, 2017.

[AN16] Seiko Arita and Shota Nakasato. Fully homomorphic encryption for point numbers. In *International Conference on Information Security and Cryptology*, pages 253–270. Springer, 2016.

[APS15] Martin R Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[ASP14] Jacob Alperin-Sheriff and Chris Peikert. Faster bootstrapping with polynomial error. In *Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I 34*, pages 297–314. Springer, 2014.

[BBB+22] Loris Bergerat, Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Parameter optimization & larger precision for (T)FHE. *IACR Cryptol. ePrint Arch.*, page 704, 2022.

[BBB+23] Loris Bergerat, Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Parameter optimization and larger precision for (t) fhe. *Journal of Cryptology*, 36(3):28, 2023.

[BBC+25] Loris Bergerat, Charlotte Bonte, Benjamin R Curtis, Jean-Baptiste Orfila, Pascal Paillier, and Samuel Tap. Sharing the mask: Tfhe bootstrapping on packed messages. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(4):925–971, 2025.

[BCL+23] Loris Bergerat, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, Adeline Roux-Langlois, and Samuel Tap. Faster secret keys for (t) fhe. *Cryptology ePrint Archive*, 2023.

[BCL+25] Loris Bergerat, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Tfhe gets real: an efficient and flexible homomorphic floating-point arithmetic. *Cryptology ePrint Archive*, 2025.

[BdBB+25] Mathieu Ballandras, Mayeul de Bellabre, Loris Bergerat, Charlotte Bonte, Carl Bootland, Benjamin R. Curtis, Jad Khatib, Jakub Klemsa, Arthur Meyre, Thomas Montaigu, Jean-Baptiste Orfila, Nicolas Sarlin, Samuel Tap, and David Testé. *TFHE-rs: A (Practical) Handbook, First Edition*. Zama, https://github.com/zama-ai/tfhe-rs-handbook/blob/main/tfhe-rs-handbook.pdf, February 2025.

[BDK+18] Joppe 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)*, pages 353–367. IEEE, 2018.

[BDPR98] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology—CRYPTO'98: 18th Annual International Cryptology Conference Santa Barbara, California, USA August 23–27, 1998 Proceedings 18*, pages 26–45. Springer, 1998.

[BGGJ20] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. Chimera: Combining ring-lwe-based fully homomorphic encryption schemes. *Journal of Mathematical Cryptology*, 14(1):316–338, 2020.

[BGN05] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-dnf formulas on ciphertexts. In *Theory of Cryptography: Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005. Proceedings 2*, pages 325–341. Springer, 2005.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 309–325, 2012.

[BIP+22] Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder V. L. Pereira, and Nigel P. Smart. FINAL: faster FHE instantiated with NTRU and LWE. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part II*, volume 13792 of *Lecture Notes in Computer Science*, pages 188–215. Springer, 2022.

[BJRLW23] Katharina Boudgoust, Corentin Jeudy, Adeline Roux-Langlois, and Weiqiang Wen. On the hardness of module learning with errors with short distributions. *Journal of Cryptology*, 36(1):1, 2023.

[BKW03] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *Journal of the ACM (JACM)*, 50(4):506–519, 2003.

[BLNRL23] Olivier Bernard, Andrea Lesavourey, Tuong-Huy Nguyen, and Adeline Roux-Langlois. Log-s-unit lattices using explicit stickelberger generators to solve approx ideal-svp. In *Advances in Cryptology–ASIACRYPT 2022: 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5–9, 2022, Proceedings, Part III*, pages 677–708. Springer, 2023.

[BMMP18] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part III 38*, pages 483–512. Springer, 2018.

[BORT25] Loris Bergerat, Jean-Baptiste Orfila, Adeline Roux-Langlois, and Samuel Tap. Accelerating tfhe with sorted bootstrapping techniques. *Under Submission*, 2025.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. *IACR Cryptology ePrint Archive*, 2012:78, 2012.

[BRL20] Olivier Bernard and Adeline Roux-Langlois. Twisted-phs: Using the product formula to solve approx-svp in ideal lattices. In *Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part II 26*, pages 349–380. Springer, 2020.

[BST20] Florian Bourse, Olivier Sanders, and Jacques Traoré. Improved secure integer comparison via homomorphic encryption. In *Cryptographers' Track at the RSA Conference*, pages 391–416. Springer, 2020.

[BV14] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on computing*, 43(2):831–871, 2014.

[CCP+24] Jung Hee Cheon, Hyeongmin Choe, Alain Passelègue, Damien Stehlé, and Elias Suvanto. Attacks against the indcpa-d security of exact fhe schemes. *Cryptology ePrint Archive*, 2024.

[CCR19] Hao Chen, Ilaria Chillotti, and Ling Ren. Onion ring ORAM: efficient constant bandwidth oblivious RAM from (leveled) TFHE. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 345–360. ACM, 2019.

[CDKS20] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient homomorphic conversion between (ring) LWE ciphertexts. *IACR Cryptol. ePrint Arch.*, 2020:15, 2020.

[CDW17] Ronald Cramer, Léo Ducas, and Benjamin Wesolowski. Short stickelberger class relations and application to ideal-svp. In *Advances in Cryptology–EUROCRYPT 2017: 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30–May 4, 2017, Proceedings, Part I*, pages 324–348. Springer, 2017.

[CGGI16a] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, pages 3–33, 2016.

[CGGI16b] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption library, August 2016. https://tfhe.github.io/tfhe/.

[CGGI17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, pages 377–408, 2017.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[CHK+18] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full rns variant of approximate homomorphic encryption. In *International Conference on Selected Areas in Cryptography*, pages 347–368. Springer, 2018.

[CIM19] Sergiu Carpov, Malika Izabachène, and Victor Mollimard. New techniques for multi-value input homomorphic evaluation and applications. In *Cryptographers' Track at the RSA Conference*, pages 106–126. Springer, 2019.

[CJP21] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In *CSCML 2021*. Springer, 2021.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, pages 409–437, 2017.

[CLOT21] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for tfhe. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 670–699. Springer, 2021.

[Con15] Keith Conrad. Cyclotomic extensions. *Preprint*, 2015.

[CSBB24] Marina Checri, Renaud Sirdey, Aymen Boudguiga, and Jean-Paul Bultel. On the practical cpad security of "exact" and threshold fhe schemes and libraries. *Cryptology ePrint Archive*, 2024.

[CSVW16] Anamaria Costache, Nigel P Smart, Srinivas Vivek, and Adrian Waller. Fixed-point arithmetic in she schemes. In *International Conference on Selected Areas in Cryptography*, pages 401–422. Springer, 2016.

[CZB+22] Pierre-Emmanuel Clet, Martin Zuber, Aymen Boudguiga, Renaud Sirdey, and Cédric Gouy-Pailler. Putting up the swiss army knife of homomorphic calculations by means of tfhe functional bootstrapping. Cryptology ePrint Archive, Report 2022/149, 2022. `https://ia.cr/2022/149`.

[DH76] WHITFIELD DIFFIE and MARTIN E HELLMAN. New directions in cryptography. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 22(6), 1976.

[DKL+18] Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 238–268, 2018.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, pages 617–640, 2015.

[DM17] Léo Ducas and Daniele Micciancio. FHEW: a fully homomorphic encryption library, 2017.

[DSDGR20] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. Lwe with side information: attacks and concrete security estimation. In *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part II*, pages 329–358. Springer, 2020.

[DSGKS18] Dana Dachman-Soled, Huijing Gong, Mukul Kulkarni, and Aria Shahverdi. Partial key exposure in ring-lwe-based cryptosystems: Attacks and resilience. Cryptology ePrint Archive, Paper 2018/1068, 2018. `https://eprint.iacr.org/2018/1068`.

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[GBA21] Antonio Guimarães, Edson Borin, and Diego F Aranha. Revisiting the functional bootstrap in tfhe. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 229–253, 2021.

[GE21] Craig Gidney and Martin Ekerå. How to factor 2048 bit rsa integers in 8 hours using 20 million noisy qubits. *Quantum*, 5:433, 2021.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178, 2009.

[GH11] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 129–148. Springer, 2011.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the aes circuit. In *Annual Cryptology Conference*, pages 850–867. Springer, 2012.

[GINX16] Nicolas Gama, Malika Izabachene, Phong Q Nguyen, and Xiang Xie. Structural lattice reduction: generalized worst-case to average-case reductions and homomorphic cryptosystems. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*, pages 528–558. Springer, 2016.

[GJS15] Qian Guo, Thomas Johansson, and Paul Stankovski. Coded-bkw: Solving lwe using lattice codes. In *Advances in Cryptology–CRYPTO 2015: 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, pages 23–42. Springer, 2015.

[GP25] Antonio Guimarães and Hilder VL Pereira. Fast amortized bootstrapping with small keys and polynomial noise overhead. *Cryptology ePrint Archive*, 2025.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. *IACR Cryptology ePrint Archive*, 2013:340, 2013.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. Ntru: A ring-based public key cryptosystem. In *Algorithmic Number Theory (ANTS III)*, volume 1423 of *LNCS*, pages 267–288. Springer, 1998.

[Hwa24] Vincent Hwang. Formal verification of emulated floating-point arithmetic in falcon. In *International Workshop on Security*, pages 125–141. Springer, 2024.

[Ins08] Institute of Electrical and Electronics Engineers. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[ISO] ISO. International organization for standardization. `https://www.iso.org/standard/87638.html`.

[Joy21] Marc Joye. Balanced non-adjacent forms. In *Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part III 27*, pages 553–576. Springer, 2021.

[JP22] Marc Joye and Pascal Paillier. Blind rotation in fully homomorphic encryption with extended keys. In *International Symposium on Cyber Security, Cryptology, and Machine Learning*, pages 1–18. Springer, 2022.

[JW22] Marc Joye and Michael Walter. Liberating tfhe: Programmable bootstrapping with general quotient polynomials. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 1–11, 2022.

[KDE+23] Andrey Kim, Maxim Deryabin, Jieun Eom, Rakyong Choi, Yongwoo Lee, Whan Ghang, and Donghoon Yoo. General bootstrapping approach for rlwe-based homomorphic encryption. *IEEE Transactions on Computers*, 73(1):86–96, 2023.

[KF15] Paul Kirchner and Pierre-Alain Fouque. An improved bkw algorithm for lwe with applications to cryptography and lattices. In *Advances in Cryptology–CRYPTO 2015: 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I 35*, pages 43–62. Springer, 2015.

[KKL+22] Taechan Kim, Hyesun Kwak, Dongwon Lee, Jinyeong Seo, and Yongsoo Song. Asymptotically faster multi-key homomorphic encryption from homomorphic gadget decomposition. *IACR Cryptol. ePrint Arch.*, page 347, 2022.

[KMS22] Hyesun Kwak, Seonhong Min, and Yongsoo Song. Towards practical multi-key TFHE: parallelizable, key-compatible, quasi-linear complexity. *IACR Cryptol. ePrint Arch.*, page 1460, 2022.

[KO22] Jakub Klemsa and Melek Onen. Parallel operations over tfhe-encrypted multi-digit integers. Cryptology ePrint Archive, Report 2022/067, 2022.

[KS21] Kamil Kluczniak and Leonard Schild. FDFB: full domain functional bootstrapping towards practical fully homomorphic encryption. *CoRR*, 2021.

[Lai17] Kim Laine. Simple encrypted arithmetic library 2.3. 1. *Microsoft Research https://www. microsoft. com/en-us/research/uploads/prod/2017/11/sealmanual-2-3-1. pdf*, 2017.

[LATV12] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 1219–1234, 2012.

[LLW+24] Zhihao Li, Xianhui Lu, Zhiwei Wang, Ruida Wang, Ying Liu, Yinhang Zheng, Lutan Zhao, Kunpeng Wang, and Rui Hou. Faster ntru-based bootstrapping in less than 4 ms. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(3):418–451, 2024.

[LM21] Baiyu Li and Daniele Micciancio. On the security of homomorphic encryption on approximate numbers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 648–677. Springer, 2021.

[LMK+23] Yongwoo Lee, Daniele Micciancio, Andrey Kim, Rakyong Choi, Maxim Deryabin, Jieun Eom, and Donghoon Yoo. Efficient fhew bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 227–256. Springer, 2023.

[LMP21] Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov. Large-precision homomorphic sign evaluation using fhew/tfhe bootstrapping. Cryptology ePrint Archive, Report 2021/1337, 2021. `https://ia.cr/2021/1337`.

[LMP22] Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov. Large-precision homomorphic sign evaluation using fhew/tfhe bootstrapping. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 130–160. Springer, 2022.

[LMSS23] Changmin Lee, Seonhong Min, Jinyeong Seo, and Yongsoo Song. Faster tfhe bootstrapping with block binary keys. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, pages 2–13, 2023.

[LNP22] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Maxime Plançon. Lattice-based zero-knowledge proofs and applications: shorter, simpler, and more general. In *Annual International Cryptology Conference*, pages 71–101. Springer, 2022.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT 2010*. Springer, 2010.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015.

[LS18] Vadim Lyubashevsky and Gregor Seiler. Short, invertible elements in partially splitting cyclotomic rings and applications to lattice-based zero-knowledge proofs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 204–224. Springer, 2018.

[LS22] Seunghwan Lee and Dong-Joon Shin. Overflow-detectable floating-point fully homomorphic encryption. *Cryptology ePrint Archive*, 2022.

[LW23] Zeyu Liu and Yunhao Wang. Amortized functional bootstrapping in less than 7 ms, with o˜(1) polynomial multiplications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 101–132. Springer, 2023.

[LY23] Kang Hoon Lee and Ji Won Yoon. Discretization error reduction for high precision torus fully homomorphic encryption. In *IACR International Conference on Public-Key Cryptography*, pages 33–62. Springer, 2023.

[MBDD+18] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, Serge Torres, et al. *Handbook of floating-point arithmetic*. Springer, 2018.

[Mic01] Daniele Micciancio. The shortest vector in a lattice is hard to approximate to within some constant. *SIAM journal on Computing*, 30(6):2008–2035, 2001.

[ML20] Subin Moon and Younho Lee. An efficient encrypted floating-point representation using heaan and tfhe. *Security and Communication Networks*, 2020, 2020.

[MR07] Daniele Micciancio and Oded Regev. Worst-case to average-case reductions based on gaussian measures. *SIA journal on computing*, 37(1):267–302, 2007.

[MR09] Daniele Micciancio and Oded Regev. Lattice-based cryptography. In *Post-quantum cryptography*, pages 147–191. Springer, 2009.

[MS18] Daniele Micciancio and Jessica Sorrell. Ring packing and amortized fhew bootstrapping. *Cryptology ePrint Archive*, 2018.

[NIS] NIST. Post-quantum cryptography standardization. `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization`.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques*, pages 223–238. Springer, 1999.

[Pei09] Chris Peikert. Public-key cryptosystems from the worst-case shortest vector problem. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 333–342, 2009.

[PFH+20] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon. *Post-Quantum Cryptography Project of NIST*, 2020.

[PMHS19] Alice Pellet-Mary, Guillaume Hanrot, and Damien Stehlé. Approx-svp in ideal lattices with pre-processing. In *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part II 38*, pages 685–716. Springer, 2019.

[PV21] Eamonn W Postlethwaite and Fernando Virdia. On the success probability of solving unique svp via bkz. In *Public-Key Cryptography–PKC 2021: 24th IACR International Conference on Practice and Theory of Public Key Cryptography, Virtual Event, May 10–13, 2021, Proceedings, Part I*, pages 68–98. Springer, 2021.

[RAD+78] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *STOC 2005*. ACM, 2005.

[RSA78] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[SE94] Claus-Peter Schnorr and Martin Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical programming*, 66:181–199, 1994.

[Sho94] Peter W Shor. Polynomial time algorithms for discrete logarithms and factoring on a quantum computer. In *Algorithmic Number Theory: First International Symposium, ANTS-I Ithaca, NY, USA, May 6–9, 1994 Proceedings 1*, pages 289–289. Springer, 1994.

[SSTX09] Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. Efficient public key encryption based on ideal lattices. In *ASIACRYPT 2009*. Springer, 2009.

[VDGHV10] Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29*, pages 24–43. Springer, 2010.

[WK19] Shibo Wang and Pankaj Kanwar. Bfloat16: The secret to high performance on cloud tpus. *Google Cloud Blog*, 4, 2019.

[WWL+24] Ruida Wang, Yundi Wen, Zhihao Li, Xianhui Lu, Benqiang Wei, Kun Liu, and Kunpeng Wang. Circuit bootstrapping: faster and smaller. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 342–372. Springer, 2024.

[YAZ+19] Rupeng Yang, Man Ho Au, Zhenfei Zhang, Qiuliang Xu, Zuoxia Yu, and William Whyte. Efficient lattice-based zero-knowledge arguments with standard soundness: construction and applications. In *Annual International Cryptology Conference*, pages 147–175. Springer, 2019.

[YXS+21] Zhaomin Yang, Xiang Xie, Huajie Shen, Shiying Chen, and Jun Zhou. Tota: fully homomorphic encryption with smaller parameters and stronger security. *Cryptology ePrint Archive*, 2021.

[Zam22] Zama. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data, 2022. `https://github.com/zama-ai/tfhe-rs`.

[ZYL+18] Tanping Zhou, Xiaoyuan Yang, Longfei Liu, Wei Zhang, and Ningbo Li. Faster bootstrapping with multiple addends. *IEEE Access*, 6:49868–49876, 2018.

Chapter A

Appendix

A.1 Appendix Chapter 4

A.1.1 Parameters

Precision	LWE Dimension (n)	LWE Noise (σ_n)	GLWE Dimension (k)	Polynomial Size (N)	GLWE Noise (σ_N)	PBS Base Log ($\mathfrak{B}_{\mathbf{PBS}}$)	PBS Level ($\ell_{\mathbf{PBS}}$)	KS Base Log($\mathfrak{B}_{\mathbf{KS}}$)	KS Level($\ell_{\mathbf{KS}}$)
2	750	1.5×10^{-5}	3	512	1.9×10^{-11}	17	1	4	3
3	797	6.7×10^{-6}	2	1024	2.8×10^{-15}	23	1	4	3
4	796	6.8×10^{-6}	1	2048	2.8×10^{-15}	23	1	3	5
5	891	1.3×10^{-6}	1	4096	2.1×10^{-19}	22	1	2	9
6	925	7.3×10^{-7}	1	8192	2.1×10^{-19}	15	2	3	6
7	997	2.1×10^{-7}	1	16384	2.1×10^{-19}	15	2	3	6
8	1069	6.1×10^{-8}	1	32768	2.1×10^{-19}	15	2	3	7
9	1136	1.9×10^{-8}	1	65536	2.1×10^{-19}	11	3	3	7

Table A.1: Summary of **PBS** Parameters for $\mathsf{p}_{\mathsf{fail}} = 2^{-40}$.

Precision	LWE Dimension (n)	LWE Noise (σ_n)	GLWE Dimension (k)	Polynomial Size (N)	GLWE Noise (σ_N)	PBS Base Log ($\mathscr{B}_{\mathbf{PBS}}$)	PBS Level ($\ell_{\mathbf{PBS}}$)	KS Base Log ($\mathscr{B}_{\mathbf{KS}}$)	KS Level($\ell_{\mathbf{KS}}$)	Extended Factor($\log_2(\eta)$)
5	850	2.6×10^{-6}	1	2048	2.8×10^{-15}	23	1	3	5	1
6	885	1.4×10^{-6}	2	1024	2.8×10^{-15}	23	1	2	8	3
7	898	1.1×10^{-6}	1	2048	2.8×10^{-15}	15	2	2	9	3
8	943	5.4×10^{-7}	1	2048	2.8×10^{-15}	15	2	2	9	4
9	973	3.2×10^{-7}	1	2048	2.8×10^{-15}	15	2	1	20	5

Table A.2: Summary of **EBS** and **SBS** Parameters for $\mathsf{p_{fail}} = 2^{-40}$.

Precision	LWE Dimension (n)	LWE Noise (σ_n)	GLWE Dimension (k)	Polynomial Size (N)	GLWE Noise (σ_N)	PBS Base Log ($\mathscr{B}_{\mathbf{PBS}}$)	PBS Level ($\ell_{\mathbf{PBS}}$)	KS Base Log ($\mathscr{B}_{\mathbf{KS}}$)	KS Level($\ell_{\mathbf{KS}}$)	Extended Factor ($\log_2(\eta)$)	Modified MS Values (d)
5	856	2.4×10^{-6}	1	2048	2.8×10^{-15}	23	1	3	5	1	20
6	885	1.4×10^{-6}	2	1024	2.8×10^{-15}	23	1	2	8	3	1
7	924	7.5×10^{-7}	1	2048	2.8×10^{-15}	15	2	2	9	3	94
8	967	3.5×10^{-7}	1	2048	2.8×10^{-15}	15	2	2	10	4	90
9	996	2.1×10^{-7}	1	2048	2.8×10^{-15}	15	2	1	20	5	76

Table A.3: Summary of **SBS** with CMS Parameters for $\mathsf{p_{fail}} = 2^{-40}$.

Precision	LWE Dimension (n)	LWE Noise (σ_n)	GLWE Dimension (k)	Polynomial Size (N)	GLWE Noise (σ_N)	PBS Base Log ($\mathscr{B}_{\mathbf{PBS}}$)	PBS Level ($\ell_{\mathbf{PBS}}$)	KS Base Log ($\mathscr{B}_{\mathbf{KS}}$)	KS Level($\ell_{\mathbf{KS}}$)
2	781	8.8×10^{-6}	4	512	2.8×10^{-15}	23	1	4	3
3	858	2.3×10^{-6}	2	1024	2.8×10^{-15}	23	1	5	3
4	834	3.5×10^{-6}	1	2048	2.8×10^{-15}	23	1	3	5
5	902	1.0×10^{-6}	1	4096	2.1×10^{-19}	15	2	3	6
6	977	3.0×10^{-7}	1	8192	2.1×10^{-19}	16	2	3	6
7	1061	7.0×10^{-8}	1	16384	2.1×10^{-19}	15	2	3	7
8	1112	2.9×10^{-8}	1	32768	2.1×10^{-19}	11	3	2	11
9	1163	1.2×10^{-8}	1	131072	2.1×10^{-19}	11	3	3	8

Table A.4: Summary of **PBS** Parameters for $\mathsf{p}_{\mathsf{fail}} = 2^{-64}$.

Precision	LWE Dimension (n)	LWE Noise (σ_n)	GLWE Dimension (k)	Polynomial Size (N)	GLWE Noise (σ_N)	PBS Base Log ($\mathscr{B}_{\mathbf{PBS}}$)	PBS Level ($\ell_{\mathbf{PBS}}$)	KS Base Log ($\mathscr{B}_{\mathbf{KS}}$)	KS Level($\ell_{\mathbf{KS}}$)	Extended Factor ($\log_2(\eta)$)
5	888	1.4×10^{-6}	2	1024	2.8×10^{-15}	23	1	2	8	2
6	896	1.2×10^{-6}	1	2048	2.8×10^{-15}	15	2	2	9	2
7	946	5.1×10^{-7}	1	2048	2.8×10^{-15}	15	2	2	9	3
8	993	2.2×10^{-7}	1	2048	2.8×10^{-15}	15	2	2	10	4
9	1045	9.3×10^{-8}	1	2048	2.8×10^{-15}	15	2	1	21	5

Table A.5: Summary of **EBS** and **SBS** Parameters for $\mathsf{p}_{\mathsf{fail}} = 2^{-64}$.

Precision	LWE Dimension (n)	LWE Noise (σ_n)	GLWE Dimension (k)	Polynomial Size (N)	GLWE Noise (σ_N)	PBS Base Log ($\mathfrak{B}_{\mathbf{PBS}}$)	PBS Level ($\ell_{\mathbf{PBS}}$)	KS Base Log ($\mathfrak{B}_{\mathbf{KS}}$)	KS Level($\ell_{\mathbf{KS}}$)
2	772	1.0×10^{-5}	4	512	2.8×10^{-15}	22	1	3	5
3	829	3.8×10^{-6}	2	1024	2.8×10^{-15}	23	1	3	5
4	876	1.7×10^{-6}	1	2048	2.8×10^{-15}	23	1	2	9
5	928	7.0×10^{-7}	1	8192	2.1×10^{-19}	15	2	4	4
6	958	4.1×10^{-7}	1	16384	2.1×10^{-19}	14	2	3	6
7	1056	7.7×10^{-8}	1	32768	2.1×10^{-19}	15	2	4	5
8	1085	4.6×10^{-8}	1	65536	2.1×10^{-19}	11	3	3	7
9	1204	6.0×10^{-9}	1	131072	2.1×10^{-19}	11	3	3	8

Table A.6: Summary of **PBS** Parameters for $\mathsf{p_{fail}} = 2^{-80}$.

Precision	LWE Dimension (n)	LWE Noise (σ_n)	GLWE Dimension (k)	Polynomial Size (N)	GLWE Noise (σ_N)	PBS Base Log ($\mathfrak{B}_{\mathbf{PBS}}$)	PBS Level ($\ell_{\mathbf{PBS}}$)	KS Base Log ($\mathfrak{B}_{\mathbf{KS}}$)	KS Level($\ell_{\mathbf{KS}}$)	Extended Factor ($\lfloor \log_2(\eta) \rfloor$)
5	873	1.8×10^{-6}	1	2048	2.8×10^{-15}	23	1	3	5	2
6	922	7.7×10^{-7}	2	1024	2.8×10^{-15}	23	1	2	9	4
7	910	9.5×10^{-7}	1	2048	2.8×10^{-15}	15	2	2	9	4
8	940	5.7×10^{-7}	1	2048	2.8×10^{-15}	15	2	1	19	5
9	984	2.6×10^{-7}	1	2048	2.8×10^{-15}	15	2	1	20	6

Table A.7: Summary of **EBS** and **SBS** Parameters for $\mathsf{p_{fail}} = 2^{-80}$.

Precision	LWE Dimension (n)	LWE Noise (σ_n)	GLWE Dimension (k)	Polynomial Size (N)	GLWE Noise (σ_N)	PBS Base Log ($\mathcal{B}_{\text{PBS}}$)	PBS Level (ℓ_{PBS})	KS Base Log ($\mathcal{B}_{\text{KS}}$)	KS Level(ℓ_{KS})	Extended Factor ($\lfloor \log_2(\eta) \rfloor$)	Modified MS Values (d)
5	873	1.8×10^{-6}	1	2048	2.8×10^{-15}	23	1	2	8	2	151
6	891	1.3×10^{-6}	1	2048	2.8×10^{-15}	14	2	2	9	3	255
7	935	6.2×10^{-7}	1	2048	2.8×10^{-15}	15	2	2	9	4	256
8	966	3.6×10^{-7}	1	2048	2.8×10^{-15}	15	2	1	19	5	256
9	1013	1.6×10^{-7}	1	2048	2.8×10^{-15}	15	2	1	21	6	255

Table A.8: Summary of **SBS** with CMS Parameters for $\mathsf{p_{fail}} = 2^{-80}$.

Precision	LWE Dimension (n)	LWE Noise (σ_n)	GLWE Dimension (k)	Polynomial Size (N)	GLWE Noise (σ_N)	PBS Base Log ($\mathcal{B}_{\text{PBS}}$)	PBS Level (ℓ_{PBS})	KS Base Log ($\mathcal{B}_{\text{KS}}$)	KS Level(ℓ_{KS})
2	783	8.5×10^{-6}	2	1024	2.8×10^{-15}	23	1	4	3
3	788	7.8×10^{-6}	1	2048	2.8×10^{-15}	23	1	3	5
4	860	2.2×10^{-6}	1	4096	2.1×10^{-19}	22	1	3	5
5	916	8.6×10^{-7}	1	8192	2.1×10^{-19}	15	2	3	6
6	983	2.7×10^{-7}	1	16384	2.1×10^{-19}	15	2	3	6
7	1089	4.3×10^{-8}	1	32768	2.1×10^{-19}	15	2	4	5
8	1113	2.8×10^{-8}	1	65536	2.1×10^{-19}	11	3	3	7
9	1176	9.7×10^{-9}	1	131072	2.1×10^{-19}	9	4	3	8

Table A.9: Summary of **PBS** Parameters for $\mathsf{p_{fail}} = 2^{-128}$.

Precision	LWE Dimension (n)	LWE Noise (σ_n)	GLWE Dimension (k)	Polynomial Size (N)	GLWE Noise (σ_N)	PBS Base Log ($\mathscr{B}_{\mathbf{PBS}}$)	PBS Level ($\ell_{\mathbf{PBS}}$)	KS Base Log ($\mathscr{B}_{\mathbf{KS}}$)	KS Level($\ell_{\mathbf{KS}}$)	Extended Factor ($\log_2(\eta)$)
4	832	3.6×10^{-6}	1	2048	2.8×10^{-15}	22	1	3	5	1
5	905	1.0×10^{-6}	1	2048	2.8×10^{-15}	23	1	2	9	2
6	889	1.3×10^{-6}	1	2048	2.8×10^{-15}	16	2	2	9	3
7	932	6.5×10^{-7}	1	2048	2.8×10^{-15}	15	2	2	9	4
8	963	3.8×10^{-7}	1	2048	2.8×10^{-15}	15	2	1	19	5
9	1009	1.7×10^{-7}	1	2048	2.8×10^{-15}	15	2	1	21	6

Table A.10: Summary of **EBS** and **SBS** Parameters for $\mathsf{p_{fail}} = 2^{-128}$.

Precision	LWE Dimension (n)	LWE Noise (σ_n)	GLWE Dimension (k)	Polynomial Size (N)	GLWE Noise (σ_N)	PBS Base Log ($\mathscr{B}_{\mathbf{PBS}}$)	PBS Level ($\ell_{\mathbf{PBS}}$)	KS Base Log ($\mathscr{B}_{\mathbf{KS}}$)	KS Level($\ell_{\mathbf{KS}}$)	Extended Factor ($\log_2(\eta)$)	Modified MS Values (d)
4	859	2.309×10^{-6}	1	2048	2.845×10^{-15}	23	1	3	5	1	123
5	905	1.044×10^{-6}	1	2048	2.845×10^{-15}	23	1	2	9	2	0
6	922	7.786×10^{-7}	1	2048	2.845×10^{-15}	15	2	2	9	3	150
7	966	3.644×10^{-7}	1	2048	2.845×10^{-15}	15	2	2	10	4	148
8	994	2.248×10^{-7}	1	2048	2.845×10^{-15}	15	2	1	20	5	137
9	1033	1.147×10^{-7}	1	2048	2.845×10^{-15}	15	2	1	21	6	96

Table A.11: Summary of **SBS** with CMS Parameters for $\mathsf{p_{fail}} = 2^{-128}$.

Precision	LWE Dimension	LWE Noise	GLWE Dimension	Polynomial Size	GLWE Noise	PBS Base Log	PBS Level	KS Base Log	KS Level	Extended Factor
2	726	2.3×10^{-5}	6	256	2.0×10^{-11}	17	1	4	3	4
3	773	1.0×10^{-5}	4	512	2.8×10^{-15}	23	1	4	3	4
4	845	2.9×10^{-6}	4	512	2.8×10^{-15}	23	1	5	3	4
5	893	1.3×10^{-6}	4	512	2.8×10^{-15}	23	1	5	3	4
6	937	6.0×10^{-7}	4	512	2.8×10^{-15}	23	1	4	4	4
7	976	3.1×10^{-7}	2	1024	2.8×10^{-15}	15	2	4	4	4
8	970	3.4×10^{-7}	1	2048	2.8×10^{-15}	15	2	3	6	4
9	1070	6.1×10^{-8}	1	4096	2.2×10^{-19}	15	2	4	5	4

Table A.12: Summary of parallel **EBS** and parallel **SBS** Parameters for $\mathsf{p}_{\mathsf{fail}} = 2^{-40}$.

Precision	LWE Dimension	LWE Noise	GLWE Dimension	Polynomial Size	GLWE Noise	PBS Base Log	PBS Level	KS Base Log	KS Level	Extended Factor
2	750	1.5×10^{-5}	6	256	2.0×10^{-11}	17	1	4	3	4
3	796	6.8×10^{-6}	4	512	2.8×10^{-15}	23	1	4	3	4
4	861	2.2×10^{-6}	4	512	2.8×10^{-15}	23	1	5	3	4
5	914	8.9×10^{-7}	4	512	2.8×10^{-15}	23	1	5	3	4
6	958	4.2×10^{-7}	2	1024	2.8×10^{-15}	23	1	4	4	4
7	977	3.0×10^{-7}	2	1024	2.8×10^{-15}	15	2	3	6	4
8	1067	6.4×10^{-8}	1	2048	2.8×10^{-15}	15	2	4	5	4
9	1119	2.6×10^{-8}	1	4096	2.2×10^{-19}	11	3	3	8	4

Table A.13: Summary of parallel **EBS** and parallel **SBS** Parameters for $\mathsf{p}_{\mathsf{fail}} = 2^{-64}$.

Precision	LWE Dimension	LWE Noise	GLWE Dimension	Polynomial Size	GLWE Noise	PBS Base Log	PBS Level	KS Base Log	KS Level	Extended Factor
2	767	1.1×10^{-5}	6	256	2.0×10^{-11}	17	1	4	3	4
3	809	5.5×10^{-6}	4	512	2.8×10^{-15}	23	1	4	3	4
4	868	2.0×10^{-6}	4	512	2.8×10^{-15}	23	1	5	3	4
5	927	7.1×10^{-7}	4	512	2.8×10^{-15}	23	1	5	3	4
6	958	4.2×10^{-7}	2	1024	2.8×10^{-15}	23	1	3	6	4
7	936	6.1×10^{-7}	1	2048	2.8×10^{-15}	15	2	3	6	4
8	1031	1.2×10^{-7}	1	4096	2.2×10^{-19}	15	2	4	5	4
9	1091	4.2×10^{-8}	1	8192	2.2×10^{-19}	15	2	3	7	4

Table A.14: Summary of parallel **EBS** and parallel **SBS** Parameters for $\mathsf{p_{fail}} = 2^{-80}$.

Precision	LWE Dimension	LWE Noise	GLWE Dimension	Polynomial Size	GLWE Noise	PBS Base Log	PBS Level	KS Base Log	KS Level	Extended Factor
2	801	6.3×10^{-6}	6	256	2.0×10^{-11}	17	1	3	5	4
3	840	3.2×10^{-6}	4	512	2.8×10^{-15}	23	1	5	3	4
4	884	1.5×10^{-6}	4	512	2.8×10^{-15}	23	1	5	3	4
5	922	7.8×10^{-7}	4	512	2.8×10^{-15}	23	1	4	4	4
6	958	4.2×10^{-7}	2	1024	2.8×10^{-15}	15	2	4	4	4
7	959	4.1×10^{-7}	1	2048	2.8×10^{-15}	15	2	3	6	4
8	1055	7.8×10^{-8}	1	4096	2.2×10^{-19}	15	2	4	5	4
9	1088	4.4×10^{-8}	1	8192	2.2×10^{-19}	11	3	3	7	4

Table A.15: Summary of parallel **EBS** and parallel **SBS** Parameters for $\mathsf{p_{fail}} = 2^{-128}$.

A.2 Appendix Chapter 5

p	Partial Shared Keys	LWE-KS Algorithm	GLWE Parameters		PBS Parameters		LWE-KS Parameters		Metrics	
			Parameter	Value	Parameter	Value	Parameter	Value	Name	Value
1	✗	traditional LWE-to-LWE	n	588	$\log_2(\mathcal{B}_{\mathsf{PBS}})$	15	$\log_2(\mathcal{B}_{\mathsf{KS}})$	3	time	5.43
			$\log_2(\sigma_n)$	-12.66						
			k	5	ℓ_{PBS}	1	ℓ_{KS}	3	size	58.6
			$\log_2(N)$	8						
			$\log_2(\sigma_{k\cdot N})$	-31.07						
1	✓	2 steps *(Alg. 35)*	n	532	$\log_2(\mathcal{B}_{\mathsf{PBS}})$	15	n_{KS}	782	time	3.78
			$\log_2(\sigma_n)$	-11.17			$\log_2(\sigma_{n_{\mathsf{KS}}})$	-17.82		
			k	5	ℓ_{PBS}	1	$\log_2(\mathcal{B}_{\mathsf{KS}_1})$	9	size	44.45
			$\log_2(N)$	8			ℓ_{KS_1}	1		
			ϕ	1280			$\log_2(\mathcal{B}_{\mathsf{KS}_2})$	2		
			$\log_2(\sigma_\phi)$	-31.07			ℓ_{KS_2}	4		
1	✓	FFT-based *(Alg. 36)*	n	534	$\log_2(\mathcal{B}_{\mathsf{PBS}})$	15	k_{in}	3	time	3.27
			$\log_2(\sigma_n)$	-11.22			k_{out}	3		
			k	5	ℓ_{PBS}	1	$\log_2(N_{\mathsf{KS}})$	8	size	37.76
			$\log_2(N)$	8			$\log_2(\mathcal{B}_{\mathsf{KS}})$	1		
			ϕ	1280			ℓ_{KS}	9		
			$\log_2(\sigma_\phi)$	-31.07						
2	✗	traditional LWE-to-LWE	n	668	$\log_2(\mathcal{B}_{\mathsf{PBS}})$	18	$\log_2(\mathcal{B}_{\mathsf{KS}})$	4	time	8.75
			$\log_2(\sigma_n)$	-14.79						
			k	6	ℓ_{PBS}	1	ℓ_{KS}	3	size	87.45
			$\log_2(N)$	8						
			$\log_2(\sigma_{k\cdot N})$	-37.88						
2	✓	2 steps *(Alg. 35)*	n	576	$\log_2(\mathcal{B}_{\mathsf{PBS}})$	18	n_{KS}	896	time	6.28
			$\log_2(\sigma_n)$	-12.34			$\log_2(\sigma_{n_{\mathsf{KS}}})$	-20.85		
			k	6	ℓ_{PBS}	1	$\log_2(\mathcal{B}_{\mathsf{KS}_1})$	10	size	66.55
			$\log_2(N)$	8			ℓ_{KS_1}	1		
			ϕ	1536			$\log_2(\mathcal{B}_{\mathsf{KS}_2})$	2		
			$\log_2(\sigma_\phi)$	-37.88			ℓ_{KS_2}	5		
2	✓	FFT-based *(Alg. 36)*	n	590	$\log_2(\mathcal{B}_{\mathsf{PBS}})$	18	k_{in}	1	time	5.32
			$\log_2(\sigma_n)$	-12.71			k_{out}	1		
			k	6	ℓ_{PBS}	1	$\log_2(N_{\mathsf{KS}})$	10	size	56.64
			$\log_2(N)$	8			$\log_2(\mathcal{B}_{\mathsf{KS}})$	1		
			ϕ	1536			ℓ_{KS}	11		
			$\log_2(\sigma_\phi)$	-37.88						

Table A.16: Parameter sets, benchmarks for PBS+LWE-KS and sizes of public material for CJP and two variants based on both partial and secret keys with shared randomness. Note that we use $\log_2(\nu) = p$. Sizes are given in MB and times in milliseconds. The parameter sets are for a failure probability of $p_{\mathsf{fail}} \leq 2^{-13.9}$.

p	Partial Shared Keys	LWE-KS Algorithm	GLWE Parameters		PBS Parameters		LWE-KS Parameters		Metrics	
			Parameter	Value	Parameter	Value	Parameter	Value	Name	Value
3	✗	traditional LWE-to-LWE	n	720	$\log_2(\mathfrak{B}_{\mathsf{PBS}})$	21	$\log_2(\mathfrak{B}_{\mathsf{KS}})$	4	time	12.2
			$\log_2(\sigma_n)$	-16.17						
			k	4	ℓ_{PBS}	1	ℓ_{KS}	3	size	104.1
			$\log_2(N)$	9						
			$\log_2(\sigma_{k\cdot N})$	-51.49						
3	✓	2 steps *(Alg. 35)*	n	648	$\log_2(\mathfrak{B}_{\mathsf{PBS}})$	18	n_{KS}	944	time	6.22
			$\log_2(\sigma_n)$	-14.25			$\log_2(\sigma_{n_{\mathsf{KS}}})$	-22.13		
			k	3	ℓ_{PBS}	1	$\log_2(\mathfrak{B}_{\mathsf{KS}_1})$	7		
			$\log_2(N)$	9			ℓ_{KS_1}	2	size	57.83
			ϕ	1536			$\log_2(\mathfrak{B}_{\mathsf{KS}_2})$	2		
			$\log_2(\sigma_\phi)$	-37.88			ℓ_{KS_2}	6		
3	✓	FFT-based *(Alg. 36)*	n	686	$\log_2(\mathfrak{B}_{\mathsf{PBS}})$	18	k_{in}	1	time	5.12
			$\log_2(\sigma_n)$	-15.27			k_{out}	1		
			k	3	ℓ_{PBS}	1	$\log_2(N_{\mathsf{KS}})$	10		
			$\log_2(N)$	9			$\log_2(\mathfrak{B}_{\mathsf{KS}})$	1	size	43.08
			ϕ	1536			ℓ_{KS}	13		
			$\log_2(\sigma_\phi)$	-37.88						
4	✗	traditional LWE-to-LWE	n	788	$\log_2(\mathfrak{B}_{\mathsf{PBS}})$	23	$\log_2(\mathfrak{B}_{\mathsf{KS}})$	4	time	12.6
			$\log_2(\sigma_n)$	-17.98						
			k	2	ℓ_{PBS}	1	ℓ_{KS}	3	size	92.39
			$\log_2(N)$	10						
			$\log_2(\sigma_{k\cdot N})$	-51.49						
4	✓	2 steps *(Alg. 35)*	n	664	$\log_2(\mathfrak{B}_{\mathsf{PBS}})$	22	n_{KS}	1126	time	9.35
			$\log_2(\sigma_n)$	-14.68			$\log_2(\sigma_{n_{\mathsf{KS}}})$	-26.97		
			k	2	ℓ_{PBS}	1	$\log_2(\mathfrak{B}_{\mathsf{KS}_1})$	13		
			$\log_2(N)$	10			ℓ_{KS_1}	1	size	68.68
			ϕ	2048			$\log_2(\mathfrak{B}_{\mathsf{KS}_2})$	2		
			$\log_2(\sigma_\phi)$	-51.49			ℓ_{KS_2}	6		
4	✓	FFT-based *(Alg. 36)*	n	682	$\log_2(\mathfrak{B}_{\mathsf{PBS}})$	23	k_{in}	3	time	7.38
			$\log_2(\sigma_n)$	-15.16			k_{out}	3		
			k	2	ℓ_{PBS}	1	$\log_2(N_{\mathsf{KS}})$	9		
			$\log_2(N)$	10			$\log_2(\mathfrak{B}_{\mathsf{KS}})$	1	size	48.61
			ϕ	2048			ℓ_{KS}	14		
			$\log_2(\sigma_\phi)$	-51.49						
5	✗	traditional LWE-to-LWE	n	840	$\log_2(\mathfrak{B}_{\mathsf{PBS}})$	23	$\log_2(\mathfrak{B}_{\mathsf{KS}})$	3	time	20.0
			$\log_2(\sigma_n)$	-19.36						
			k	1	ℓ_{PBS}	1	ℓ_{KS}	6	size	131.3
			$\log_2(N)$	11						
			$\log_2(\sigma_{k\cdot N})$	-51.49						
5	✓	2 steps *(Alg. 35)*	n	732	$\log_2(\mathfrak{B}_{\mathsf{PBS}})$	23	n_{KS}	1171	time	13.8
			$\log_2(\sigma_n)$	-16.49			$\log_2(\sigma_{n_{\mathsf{KS}}})$	-28.17		
			k	1	ℓ_{PBS}	1	$\log_2(\mathfrak{B}_{\mathsf{KS}_1})$	9		
			$\log_2(N)$	11			ℓ_{KS_1}	2	size	78.62
			ϕ	2048			$\log_2(\mathfrak{B}_{\mathsf{KS}_2})$	2		
			$\log_2(\sigma_\phi)$	-51.49			ℓ_{KS_2}	7		
5	✓	FFT-based *(Alg. 36)*	n	766	$\log_2(\mathfrak{B}_{\mathsf{PBS}})$	23	k_{in}	3	time	11.0
			$\log_2(\sigma_n)$	-17.39			k_{out}	3		
			k	1	ℓ_{PBS}	1	$\log_2(N_{\mathsf{KS}})$	9		
			$\log_2(N)$	11			$\log_2(\mathfrak{B}_{\mathsf{KS}})$	1	size	48.58
			ϕ	2048			ℓ_{KS}	15		
			$\log_2(\sigma_\phi)$	-51.49						

Table A.17: Parameter sets, benchmarks for PBS+LWE-KS and sizes of public material for CJP and two variants based on both partial and secret keys with shared randomness. Note that we use $\log_2(\nu) = p$. Sizes are given in MB and times in milliseconds. The parameter sets are for a failure probability of $p_{\mathsf{fail}} \leq 2^{-13.9}$.

p	Partial Shared Keys	LWE-KS Algorithm	GLWE Parameters		PBS Parameters		LWE-KS Parameters		Metrics	
			Parameter	Value	Parameter	Value	Parameter	Value	Name	Value
6	✗	traditional LWE-to-LWE	n $\log_2(\sigma_n)$	840 −19.36	$\log_2(\mathfrak{B}_{\mathsf{PBS}})$	14	$\log_2(\mathfrak{B}_{\mathsf{KS}})$	3	time	55.6
			k $\log_2(N)$ $\log_2(\sigma_{k\cdot N})$	1 12 −62.00	ℓ_{PBS}	2	ℓ_{KS}	5	size	341.4
6	✓	2 steps *(Alg. 35)*	n $\log_2(\sigma_n)$	748 −16.91	$\log_2(\mathfrak{B}_{\mathsf{PBS}})$	14	n_{KS} $\log_2(\sigma_{n_{\mathsf{KS}}})$ $\log_2(\mathfrak{B}_{\mathsf{KS}_1})$ ℓ_{KS_1} $\log_2(\mathfrak{B}_{\mathsf{KS}_2})$ ℓ_{KS_2}	1313 −31.94 16 1 2 8	time	44.3
			k $\log_2(N)$ ϕ $\log_2(\sigma_\phi)$	1 12 2443 −62.00	ℓ_{PBS}	2			size	224.2
6	✓	FFT-based *(Alg. 36)*	n $\log_2(\sigma_n)$	774 −17.61	$\log_2(\mathfrak{B}_{\mathsf{PBS}})$	14	k_{in} k_{out} $\log_2(N_{\mathsf{KS}})$ $\log_2(\mathfrak{B}_{\mathsf{KS}})$ ℓ_{KS}	1 1 11 1 15	time	41.1
			k $\log_2(N)$ ϕ $\log_2(\sigma_\phi)$	1 12 2443 −62.00	ℓ_{PBS}	2			size	194.0
7	✗	traditional LWE-to-LWE	n $\log_2(\sigma_n)$	896 −20.85	$\log_2(\mathfrak{B}_{\mathsf{PBS}})$	15	$\log_2(\mathfrak{B}_{\mathsf{KS}})$	3	time	129.0
			k $\log_2(N)$ $\log_2(\sigma_{k\cdot N})$	1 13 −62.00	ℓ_{PBS}	2	ℓ_{KS}	6	size	784.4
7	✓	2 steps *(Alg. 35)*	n $\log_2(\sigma_n)$	776 −17.66	$\log_2(\mathfrak{B}_{\mathsf{PBS}})$	15	n_{KS} $\log_2(\sigma_{n_{\mathsf{KS}}})$ $\log_2(\mathfrak{B}_{\mathsf{KS}_1})$ ℓ_{KS_1} $\log_2(\mathfrak{B}_{\mathsf{KS}_2})$ ℓ_{KS_2}	1332 −32.45 10 2 1 16	time	101.0
			k $\log_2(N)$ ϕ $\log_2(\sigma_\phi)$	1 13 2443 −62.00	ℓ_{PBS}	2			size	463.3
7	✓	FFT-based *(Alg. 36)*	n $\log_2(\sigma_n)$	818 −18.78	$\log_2(\mathfrak{B}_{\mathsf{PBS}})$	14	k_{in} k_{out} $\log_2(N_{\mathsf{KS}})$ $\log_2(\mathfrak{B}_{\mathsf{KS}})$ ℓ_{KS}	1 1 11 1 16	time	90.3
			k $\log_2(N)$ ϕ $\log_2(\sigma_\phi)$	1 13 2443 −62.00	ℓ_{PBS}	2			size	409.5
8	✗	traditional LWE-to-LWE	n $\log_2(\sigma_n)$	968 −22.77	$\log_2(\mathfrak{B}_{\mathsf{PBS}})$	11	$\log_2(\mathfrak{B}_{\mathsf{KS}})$	3	time	415
			k $\log_2(N)$ $\log_2(\sigma_{k\cdot N})$	1 14 −62.00	ℓ_{PBS}	3	ℓ_{KS}	6	size	2179
8	✓	2 steps *(Alg. 35)*	n $\log_2(\sigma_n)$	816 −18.72	$\log_2(\mathfrak{B}_{\mathsf{PBS}})$	11	n_{KS} $\log_2(\sigma_{n_{\mathsf{KS}}})$ $\log_2(\mathfrak{B}_{\mathsf{KS}_1})$ ℓ_{KS_1} $\log_2(\mathfrak{B}_{\mathsf{KS}_2})$ ℓ_{KS_2}	1359 −33.17 9 2 1 17	time	323
			k $\log_2(N)$ ϕ $\log_2(\sigma_\phi)$	1 14 2443 −62.00	ℓ_{PBS}	3			size	1304
8	✓	FFT-based *(Alg. 36)*	n $\log_2(\sigma_n)$	854 −19.73	$\log_2(\mathfrak{B}_{\mathsf{PBS}})$	11	k_{in} k_{out} $\log_2(N_{\mathsf{KS}})$ $\log_2(\mathfrak{B}_{\mathsf{KS}})$ ℓ_{KS}	1 1 11 1 18	time	306
			k $\log_2(N)$ ϕ $\log_2(\sigma_\phi)$	1 14 2443 −62.00	ℓ_{PBS}	3			size	1282

Table A.18: Parameter sets, benchmarks for PBS+LWE-KS and sizes of public material for CJP and two variants based on both partial and secret keys with shared randomness. Note that we use $\log_2(\nu) = p$. Sizes are given in MB and times in milliseconds. The parameter sets are for a failure probability of $p_{\mathsf{fail}} \leq 2^{-13.9}$.

p	Partial Shared Keys	LWE-KS Algorithm	GLWE Parameters		PBS Parameters		LWE-KS Parameters		Metrics	
			Parameter	Value	Parameter	Value	Parameter	Value	Name	Value
9	✗	traditional LWE-to-LWE	n	1024	$\log_2(\mathcal{B}_{\mathsf{PBS}})$	9	$\log_2(\mathcal{B}_{\mathsf{KS}})$	3	time	1340
			$\log_2(\sigma_n)$	-24.26						
			k	1	ℓ_{PBS}	4	ℓ_{KS}	7	size	5890
			$\log_2(N)$	15						
			$\log_2(\sigma_{k\cdot N})$	-62.00						
9	✓	2 steps *(Alg. 35)*	n	860	$\log_2(\mathcal{B}_{\mathsf{PBS}})$	8	n_{KS}	1388	time	1010
			$\log_2(\sigma_n)$	-19.89			$\log_2(\sigma_{n_{\mathsf{KS}}})$	-33.94		
			k	1	ℓ_{PBS}	4	$\log_2(\mathcal{B}_{\mathsf{KS}_1})$	10	size	3525
			$\log_2(N)$	15			ℓ_{KS_1}	2		
			ϕ	2443			$\log_2(\mathcal{B}_{\mathsf{KS}_2})$	1		
			$\log_2(\sigma_\phi)$	-62.00			ℓ_{KS_2}	18		
9	✓	FFT-based *(Alg. 36)*	n	902	$\log_2(\mathcal{B}_{\mathsf{PBS}})$	8	k_{in}	1	time	1003
			$\log_2(\sigma_n)$	-21.01			k_{out}	1		
			k	1	ℓ_{PBS}	4	$\log_2(N_{\mathsf{KS}})$	11	size	3609
			$\log_2(N)$	15			$\log_2(\mathcal{B}_{\mathsf{KS}})$	1		
			ϕ	2443			ℓ_{KS}	18		
			$\log_2(\sigma_\phi)$	-62.00						
10	✗	traditional LWE-to-LWE	n	1096	$\log_2(\mathcal{B}_{\mathsf{PBS}})$	6	$\log_2(\mathcal{B}_{\mathsf{KS}})$	2	time	4710
			$\log_2(\sigma_n)$	-26.17						
			k	1	ℓ_{PBS}	6	ℓ_{KS}	12	size	19730
			$\log_2(N)$	16						
			$\log_2(\sigma_{k\cdot N})$	-62.00						
10	✓	2 steps *(Alg. 35)*	n	904	$\log_2(\mathcal{B}_{\mathsf{PBS}})$	6	n_{KS}	1417	time	3620
			$\log_2(\sigma_n)$	-21.06			$\log_2(\sigma_{n_{\mathsf{KS}}})$	-34.71		
			k	1	ℓ_{PBS}	6	$\log_2(\mathcal{B}_{\mathsf{KS}_1})$	11	size	10940
			$\log_2(N)$	16			ℓ_{KS_1}	2		
			ϕ	2443			$\log_2(\mathcal{B}_{\mathsf{KS}_2})$	1		
			$\log_2(\sigma_\phi)$	-62.00			ℓ_{KS_2}	19		
10	✓	FFT-based *(Alg. 36)*	n	938	$\log_2(\mathcal{B}_{\mathsf{PBS}})$	6	k_{in}	3	time	3603
			$\log_2(\sigma_n)$	-21.97			k_{out}	3		
			k	1	ℓ_{PBS}	6	$\log_2(N_{\mathsf{KS}})$	9	size	11260
			$\log_2(N)$	16			$\log_2(\mathcal{B}_{\mathsf{KS}})$	1		
			ϕ	2443			ℓ_{KS}	20		
			$\log_2(\sigma_\phi)$	-62.00						
11	✗	traditional LWE-to-LWE	n	1132	$\log_2(\mathcal{B}_{\mathsf{PBS}})$	2	$\log_2(\mathcal{B}_{\mathsf{KS}})$	2	time	43900
			$\log_2(\sigma_n)$	-27.13						
			k	1	ℓ_{PBS}	20	ℓ_{KS}	13	size	105300
			$\log_2(N)$	17						
			$\log_2(\sigma_{k\cdot N})$	-62.00						
11	✓	2 steps *(Alg. 35)*	n	984	$\log_2(\mathcal{B}_{\mathsf{PBS}})$	3	n_{KS}	1471	time	18000
			$\log_2(\sigma_n)$	-23.19			$\log_2(\sigma_{n_{\mathsf{KS}}})$	-36.15		
			k	1	ℓ_{PBS}	12	$\log_2(\mathcal{B}_{\mathsf{KS}_1})$	11	size	47330
			$\log_2(N)$	17			ℓ_{KS_1}	2		
			ϕ	2443			$\log_2(\mathcal{B}_{\mathsf{KS}_2})$	1		
			$\log_2(\sigma_\phi)$	-62.00			ℓ_{KS_2}	21		
11	✓	FFT-based *(Alg. 36)*	n	1018	$\log_2(\mathcal{B}_{\mathsf{PBS}})$	3	k_{in}	3	time	19450
			$\log_2(\sigma_n)$	-24.10			k_{out}	3		
			k	1	ℓ_{PBS}	13	$\log_2(N_{\mathsf{KS}})$	9	size	52940
			$\log_2(N)$	17			$\log_2(\mathcal{B}_{\mathsf{KS}})$	1		
			ϕ	2443			ℓ_{KS}	22		
			$\log_2(\sigma_\phi)$	-62.00						

Table A.19: Parameter sets, benchmarks for PBS+LWE-KS and sizes of public material for CJP and two variants based on both partial and secret keys with shared randomness. Note that we use $\log_2(\nu) = p$. Sizes are given in MB and times in milliseconds. The parameter sets are for a failure probability of $p_{\mathsf{fail}} \leq 2^{-13.9}$.

A.3 Appendix Chapter 7

A.3.1 Parameters

In Tables A.20 and A.21, we report the cryptographic parameters that we use to compute our benchmarks. All of them have been obtained with the optimization framework. In those tables, the notation $\mathfrak{B}$ (resp. ℓ) refers to the basis (resp. the number of levels) parameter used for a given FHE algorithm such as a key switch or a [CGGI20]'s **PBS**. By default, the cryptographic parameters ensure 128 bits of security, a failure probability $p_{\mathsf{fail}}\left(\mathcal{A}^{(\text{CJP21})}\right), p_{\mathsf{fail}}\left(\mathcal{A}^{(\text{this work})}\right) \leq 2^{-13.9}$ i.e. a standard score (Definition 2) of 4 which is pretty easy to experiment with.

Remark A.1 (Biggest 2-Norm)**.** For a given message modulo $\mathcal{B}$ and carry-message modulo p one can find the worst 2-norm that they could encounter in the modular arithmetic defined in Section 2.3. Indeed, a fresh encoding is at worst $\mathcal{B}-1$, and the biggest message one can consider before needing to empty the carry buffer is $p-1$, so the biggest integer one can multiply a ciphertext with is $\left\lfloor \frac{p-1}{\mathcal{B}-1} \right\rfloor$ which is the biggest 2-norm.

param ID	AP parameters		LWE		GLWE			LWE-to-LWE key switch		PBS		WoP-PBS compatible
	p	ν	n	$\log_2(\sigma)$	k	$\log_2(N)$	$\log_2(\sigma)$	$\log_2(\mathfrak{B})$	ℓ	$\log_2(\mathfrak{B})$	ℓ	
1	2^2	3	615	-13.38	4	9	-51.49	2	5	12	3	#8
2	2^4	5	702	-15.69	2	10	-51.49	2	7	9	4	#9
3	2^6	5	872	-20.21	1	12	-62.00	4	4	22	1	#10
4	2^2	3	667	-14.76	6	8	-37.88	4	3	18	1	∅
5	2^4	5	784	-17.87	2	10	-51.49	4	3	23	1	∅
6	2^8	17	983	-23.17	1	14	-62.00	4	5	15	2	∅
7	2^6	9	838	-19.30	1	12	-62.00	3	5	15	2	∅

Table A.20: Optimized parameters for $\mathcal{A}$ of type $\mathcal{A}^{(\text{CJP21})}$.

In Table A.20, we provide seven parameter sets for $\mathcal{A}^{(\text{CJP21})}$, each one with a bit of padding, a specific message modulus p and specific 2-norm ν. In Table A.21, we provide five parameters sets for $\mathcal{A}^{(\text{this work})}$, each one with a specific (carry-) message modulo p, a specific number of bits to extract per LWE ciphertext during the **WoP-PBS**, a specific number κ of input LWE ciphertext to the **WoP-PBS** and a specific 2-norm ν. They do not have a bit of padding. In parameter IDs #11 and #12, the message modulus specifies the CRT base used and the corresponding number of bits to extract for each base.

param ID	AP parameters				LWE		GLWE			micro parameters		
	p	bit(s) to extract	κ	ν	n	$\log_2(\sigma)$	k	$\log_2(N)$	$\log_2(\sigma)$	operator	$\log_2(\mathfrak{B})$	ℓ
8 *compatible with CJP#1*	2^2	1	16	3	549	-11.62	2	10	-51.49	LWE-to-LWE key switch	2	5
										PBS	12	3
										packing key switch	17	2
										circuit bootstrapping	13	1
9 *compatible with CJP#2*	2^4	2	8	5	534	-11.22	2	10	-51.49	LWE-to-LWE key switch	2	5
										PBS	12	3
										packing key switch	17	2
										circuit bootstrapping	9	2
10 *compatible with CJP#3*	2^6	4	5	5	538	-11.33	4	10	-62.00	LWE-to-LWE key switch	1	10
										PBS	4	11
										packing key switch	20	2
										circuit bootstrapping	7	4
11	$\begin{pmatrix}7\\8\\9\\11\\13\end{pmatrix}$	$\begin{pmatrix}3\\3\\4\\4\\4\end{pmatrix}$	5	5	696	-15.53	2	10	-51.49	LWE-to-LWE key switch	2	7
										PBS	9	4
										packing key switch	17	2
										circuit bootstrapping	7	3
12	$\begin{pmatrix}3\\11\\13\\19\\23\\29\\31\\32\end{pmatrix}$	$\begin{pmatrix}2\\4\\4\\5\\5\\5\\5\\5\end{pmatrix}$	8	2^5	781	-17.79	1	11	-51.49	LWE-to-LWE key switch	1	16
										PBS	5	8
										packing key switch	13	3
										circuit bootstrapping	6	4

Table A.21: Optimized parameters for $\mathcal{A}$ of type $\mathcal{A}^{(\text{this work})}$.

Titre : Vers une Arithmétique Homomorphe Pratique et Efficace.

Mots Clefs : Cryptologie, Chiffrement Totalement Homomorphe, TFHE.

Résumé : Le chiffrement totalement homomorphe (Fully Homomorphic Encryption, FHE) est une famille de schémas de chiffrement permettant d'effectuer des opérations directement sur des données chiffrées. Grâce à cette propriété, les schémas FHE permettent l'évaluation de circuits tout en préservant la confidentialité des utilisateurs. En conséquence, le FHE trouve des applications dans de nombreux domaines tels que l'apprentissage automatique, la blockchain, et bien d'autres. Au cours des dernières décennies, le domaine est passé de schémas nécessitant un temps impraticable pour évaluer de petits circuits à des schémas capables d'évaluer des circuits complexes dans un temps raisonnable, ouvrant la voie à l'adoption du FHE à l'échelle industrielle.

Cette thèse s'inscrit dans ce contexte, avec un accent particulier sur le schéma TFHE. TFHE est un schéma particulièrement efficace pour effectuer des opérations sur des messages de faible précision, principalement grâce à une opération, le *bootstrapping*, utilisée tout au long du circuit pour garantir le résultat des opérations. Malgré son efficacité, TFHE présente encore certaines limitations que nous cherchons à surmonter dans ce manuscrit. Comme mentionné précédemment, TFHE dispose d'algorithmes très performants pour traiter des messages de petite précision, mais de nombreux circuits reposent sur des entiers 32 ou 64 bits, ou encore les nombres à virgule flottante. Concernant cette première limitation, nous avons étudié comment représenter efficacement ces types de données avec TFHE en utilisant des encodages et des algorithmes dédiés. De plus, bien que TFHE soit l'un des schémas FHE les plus efficaces, il reste lent comparé aux opérations sur des données en clair. Afin de réduire ces différences, nous avons étudié des algorithmes bas niveau et de nouvelles primitives visant à diminuer le coût global des opérations FHE. L'ensemble de ces améliorations permet la création d'une arithmétique homomorphe efficace et pratique, réduisant ainsi l'écart entre le monde en clair et le monde homomorphe.

Title: Towards Efficient and Practical Homomorphic Arithmetics.

Key Words : Cryptology, Fully Homomorphic Encryption, TFHE.

Abstract: Fully Homomorphic Encryption (FHE) is a family of encryption schemes permitting operations over encrypted data. Thanks to this property, FHE schemes allow the evaluation of circuits while preserving user privacy. As a result, FHE has applications in diverse domains such as machine learning, blockchain, and so on. Over the last decades, the field has evolved from schemes taking an impracticable amount of time to evaluate small circuits to schemes able to evaluate complex circuits in a reasonable amount of time, leading to the beginning of the adoption of FHE at the industrial level.

This thesis begins in this context, with a focus on the TFHE scheme. TFHE is an FHE scheme that is highly efficient at performing operations over small message precision, mainly due to a core operation called bootstrapping, which is used all along the circuit to ensure correctness. However, despite its efficiency, TFHE still has some limitations that we aim to overcome in this manuscript. As mentioned before, TFHE has very efficient algorithms to work with small message precision, but many circuits work with primitive data types such as 32- or 64-bit integers or floating-point numbers. Regarding this first limitation, we study how to efficiently represent these data types with TFHE by using dedicated encodings and algorithms. Although TFHE is one of the fastest FHE schemes, it remains slow compared to plaintext operations. To reduce the timing differences, we study low-level algorithms and new primitives to reduce the global cost of FHE operations. All these improvements permit the creation of efficient and practical homomorphic arithmetic, bridging the gap between the clear world and the homomorphic world.