THÈSE de DOCTORAT DE L'ÉCOLE CENTRALE DE LYON

membre de l'**Université de Lyon**

École Doctorale N° 160

Électronique, Électrotechnique et Automatique (EEA)

Spécialité de doctorat : Électronique, Nanotechnologie, Optique et Laser

Soutenue publiquement le 10 Novembre 2026, par :

Diamante Simone Crescenzo

New Hardware Implementations of Post-Quantum Cryptography for Sustainable Electronic Systems

Devant le jury composé de :

Nele MENTENS	Professeure	KU Leuven	Rapporteure
Paolo MAISTRI	Chargé de recherche HDR	CNRS / TIMA	Rapporteur
Giorgio DI NATALE	Directeur de recherche	CNRS / TIMA	Examinateur
Emanuele VALEA	Ingénieur de recherche	CEA-List	Co-encadrant
Alberto BOSIO	Professeur	ECL / INL	Directeur de thèse

Thèse préparée au CEA-List (Laboratoire LFIM) et financée dans le cadre du projet Carnot AxPQC.

Acknowledgements

Questo lavoro è frutto di impegno e sacrifici svolti durante i tre anni del mio dottorato di ricerca, e, giunto alla fine, desidero dedicare un pensiero a coloro che mi hanno accompagnato in questo percorso.

Primi fra tutti desidero ringraziare mamma e papà, che hanno dovuto far fronte al peso di avermi lontano per tutto questo tempo, e credo che questo sia stato difficile almeno quanto lo stesso completamento di tale percorso. Grazie per tutte le parole di incoraggiamento e conforto, e per tutto il sostegno che mi avete dimostrato senza bisogno di chiedere.

Un ringraziamento particolare a mia sorella Chiara, che ha sopportato la mia lontananza accontentandosi di qualche momento insieme ogni tanto, senza mai tirarsi indietro nel dimostrarmi il suo smisurato affetto anche quando avrebbe avuto bisogno di una mia presenza più stabile. Sei un esempio.

Un immenso grazie a Federica, mia compagna di vita, per esserti fatta carico di una relazione che per questi anni si è potuta vivere appieno solo qualche giorno a settimana. Grazie per non aver dubitato mai nemmeno un secondo della nostra forza, e di avermi supportato senza voler sentire storie anche quando io sono stato il primo a non credere nelle mie capacità. Grazie per i tuoi colori, i sorrisoni che mi regali, i tanti viaggi insieme e i ricordi che mi riempiono il cuore e che hanno saputo tenermi compagnia in quella cameretta d'oltralpe. Grazie per esserti presa cura di me, per essere venuta fin qui innumerevoli volte, e per non aver mai perso l'occasione di sognare insieme. Sono grato alla vita per averti riportata sul mio cammino, che adesso desidero continuare accanto a te. Ti amo dal profondo del mio cuore, e non vedo l'ora di poterti vivere ogni giorno.

Un profondo ringraziamento a nonna, a zii e cugini, e a tutta la famiglia per non perdere mai l'occasione di farmi arrivare il loro affetto nonostante la distanza e con il desiderio di accompagnarmi in questo percorso e nella vita.

Una dedica in particolare per la mia famiglia americana, Kellie, Kade, Cole, Carson e Brad, con immenso affetto e gratitudine. Quest'anno, ancora una volta, mi hanno accolto come se non fosse passato un giorno dalla mia partenza. Ogni momento insieme è un dono unico, è stato un regalo doppio potervi anche rivedere in Italia, e spero che le nostre strade si incrocino ad ogni occasione possibile.

Ai miei boys, che non hanno mai perso un'occasione per spronarmi, strapparmi un sorriso, farmi vivere le mie passioni, e tutto facendo il conto alla rovesca insieme a me per il mio rientro. È un privilegio sapere di poter sempre contare su di voi e pennellare insieme tutte le curve che decideremo di condividere.

Ad Andrea, grazie per avermi spinto ed incoraggiato ad andare avanti in questi anni, con l'augurio di poter condividere il nostro sogno a due ruote sempre di più da adesso in poi.

Ai miei amici di sempre, grazie per la pazienza e perseveranza nel cercare di condividere del tempo insieme.

A Giorgia, Gianluca, Tiziana e Michele, grazie di cuore per avermi fatto sentire parte della famiglia dal primo momento e per l'affetto immenso che sapete regalare in ogni piccolo gesto.

Ad Antonio, Salima, Andrea, Anna, Mario, Rebecca, Momo, Giancarlo e Clémence, per essere stati la mia famiglia grenoblese sin dai primi giorni di questa avventura. Grazie per le innumerevoli risate, avventure e momenti che abbiamo condiviso. Vi aspetto ad Avigliana!

Grazie ai mitici colleghi del lab, Stefano, Andrea, Ivan, Riccardo e Giorgio, per i confronti tecnici ma anche e soprattutto per gli 'Urgent Meetings', l'ambiente sereno e le risate che hanno accompagnato il nostro lavoro.

Un sentito ringraziamento ad Emanuele e Alberto, per aver creduto che io fossi la persona giusta per costruire una parte di questo progetto. Siete stati un esempio in questo campo, grazie per i consigli, i discorsi di supporto quando le cose sembravano non andare nella direzione sperata. È stato un vero piacere e un privilegio lavorare insieme, sono fiero del lavoro che abbiamo portato a compimento.

Infine, grazie alla mia MiTo, umile e fedele destriero, per aver affrontato con me le condizioni più avverse di questa fetta di Alpi per permettermi di inseguire i miei sogni, raggiungere i miei affetti, e per avermi sempre riportato a casa.

This work is the result of commitment and sacrifices made during the three years of my PhD, and, having reached the end, I wish to dedicate a thought to those who have accompanied me along this journey.

First and foremost, I wish to thank Mom and Dad, who had to bear the weight of having me far away for all this time, and I believe this was at least as difficult as completing this journey itself. Thank you for all the words of encouragement and comfort, and for all the support you have shown me without having to ask.

A special thank you to my sister Chiara, who endured my absence by settling for a few moments together every now and then, without ever holding back in showing me her boundless affection even when she would have needed a more stable presence from me. You are an example.

An immense thank you to Federica, my life partner, for taking on the weight of a relationship that over these years could only be lived to the fullest a few days a week. Thank you for never doubting our strength for even a second, and for supporting me without accepting any excuses even when I was the first not to believe in my abilities. Thank you for your colors, the big smiles you give me, the many trips together, and the memories that fill my heart and kept me company in that little room across the Alps. Thank you for taking care of me, for coming all the way here countless times, and for never missing the chance to dream together. I am grateful to life for bringing you back onto my path, which I now wish to continue beside you. I love you from the bottom of my heart, and I cannot wait to experience every day with you.

A deep thank you to Grandma, to my aunts, uncles, and cousins, and to the whole family for never missing an opportunity to send me their affection despite the distance, with the desire to accompany me along this journey and in life.

A special dedication to my American family, Kellie, Kade, Cole, Carson, and Brad, with immense affection and gratitude. This year, once again, they welcomed me as if not a single day had passed since my departure. Every moment together is a unique gift; it was a double gift to also be able to see you again in Italy, and I hope our paths cross at every possible opportunity.

To my boys, who have never missed a chance to spur me on, bring a smile to my face, and let me live out my passions, all while counting down with me until my return. It is a privilege to know I can always count on you and drive together all the turns we decide to share.

To Andrea, thank you for pushing and encouraging me to keep going over these years, with the wish that we can share our two-wheeled dream more and more from now on.

To my lifelong friends, thank you for your patience and perseverance in trying to share time together.

To Giorgia, Gianluca, Tiziana, and Michele, thank you from the bottom of my heart for making me feel like part of the family from the very first moment and for the immense affection you know how to give in every little gesture.

To Antonio, Salima, Andrea, Anna, Mario, Rebecca, Momo, Giancarlo, and Clémence, for being my Grenoble family since the very first days of this adventure. Thank you for the countless laughs, adventures, and moments we have shared. I'll be waiting for you in Avigliana!

Thank you to my legendary lab colleagues, Stefano, Andrea, Ivan, Riccardo, and Giorgio, for the technical discussions, but also and above all for the 'Urgent Meetings', the relaxed atmosphere, and the laughs that accompanied our work.

A heartfelt thank you to Emanuele and Alberto, for believing that I was the right person to build a part of this project. You have been an example in this field. Thank you for the advice and the supportive talks when things did not seem to be going in the hoped-for direction. It has been a true pleasure and a privilege working together, and I am proud of the work we have accomplished.

Finally, thank you to my MiTo, humble and faithful steed, for braving with me the harshest conditions of this slice of the Alps to allow me to chase my dreams, reach my loved ones, and for always bringing me home.

Abstract

Post-Quantum Cryptography (PQC) is essential for securing digital communications against the emerging threat of quantum computing. Among PQC algorithms, lattice-based constructions are particularly promising due to their strong security guarantees. Many leading schemes rely on the Learning With Errors (LWE) problem, which provides the computational hardness necessary for cryptographic security. Fundamentally, LWE involves computing a matrix-vector product over a finite field and perturbing the result by adding a small error vector. In practical hardware implementations, executing these schemes introduces severe bottlenecks, driven heavily by resource-intensive pseudo-randomness generation for seed expansion and complex, dedicated sampling circuits required for explicit error generation. To mitigate these architectural overheads, the paradigm of Approximate Computing (AxC) presents a promising alternative, yet it is traditionally employed in error-tolerant domains where strict mathematical exactness is non-essential. Consequently, its application to cryptography is largely unexplored in literature, as non-exact computations are conventionally perceived as an inherent security vulnerability.

This thesis addresses the cross-domain overlay between AxC and LWE-based cryptographic schemes, introducing the Approximate LWE (AxLWE) paradigm. This implementation methodology replaces the traditional explicit error generation chain by leveraging the intrinsic errors introduced by approximate operators. Unlike previous research that uses approximate arithmetic merely to simplify matrix arithmetic units, or methods relying on physical overscaling, AxLWE utilizes the inherent inaccuracies of state-of-the-art digital approximate operators. To ensure cryptographic integrity, the thesis proposes a systematic selection methodology for approximate operators based on error distribution parameters and security bounds. Based on this foundation, a global architectural analysis is established to evaluate the impact of the AxLWE paradigm compared to conventional, exact LWE hardware implementations.

Résumé

La cryptographie post-quantique (PQC) est essentielle pour sécuriser les communications numériques face au développement de l'informatique quantique. Parmi les algorithmes de PQC, les constructions basées sur les réseaux euclidiens sont particulièrement prometteuses en raison de leurs fortes garanties de sécurité. De nombreux schémas de premier plan reposent sur le problème de l'apprentissage avec erreurs, ou Learning With Errors (LWE), qui fournit la dureté calculatoire nécessaire à la sécurité cryptographique. Fondamentalement, LWE implique de calculer un produit matrice-vecteur sur un corps fini et de perturber le résultat en y ajoutant un petit vecteur d'erreur. Dans les implémentations matérielles pratiques, l'exécution de ces schémas introduit d'importants goulots d'étranglement, principalement induits par la génération de pseudo-aléatoire, très gourmande en ressources, pour l'expansion de graine, ainsi que par les circuits d'échantillonnage complexes et dédiés requis pour la génération explicite d'erreurs. Pour atténuer ces surcoûts architecturaux, le paradigme du calcul approché, ou Approximate Computing (AxC), présente une alternative prometteuse, bien qu'il soit traditionnellement employé dans des domaines tolérants aux erreurs où l'exactitude mathématique stricte n'est pas essentielle. Par conséquent, son application à la cryptographie reste largement inexplorée dans la littérature, car les calculs non exacts sont conventionnellement perçus comme une vulnérabilité de sécurité.

Cette thèse aborde le chevauchement inter-domaines entre l'AxC et les schémas cryptographiques basés sur LWE, en introduisant le paradigme LWE Approché, ou Approximate LWE (AxLWE). Cette méthodologie d'implémentation remplace la chaîne traditionnelle de génération explicite d'erreurs en tirant parti des erreurs intrinsèques introduites par les opérateurs approchés. Contrairement aux recherches précédentes qui utilisent l'arithmétique approchée uniquement pour simplifier les unités matricielles, ou aux méthodes reposant sur une forte mise à l'échelle physique (overscaling), AxLWE utilise les inexactitudes inhérentes des opérateurs numériques approchés de l'état de l'art. Afin de garantir l'intégrité cryptographique, la thèse apporte une méthodologie de sélection systématique pour les opérateurs approchés, basée sur les paramètres de distribution d'erreurs et les bornes de sécurité. Sur cette base, une analyse architecturale globale est établie pour évaluer l'impact du paradigme AxLWE par rapport aux implémentations matérielles LWE exactes et conventionnelles.

Résumé étendue

La cryptographie constitue la pierre angulaire indispensable de la sécurité de l'information moderne, sous-tendant la confiance, la vie privée et l'infrastructure de notre société numérique mondiale. Chaque jour, des milliards d'équipements interconnectés s'appuient sur des protocoles cryptographiques pour sécuriser un large éventail d'activités quotidiennes, allant des transactions financières courantes et de la protection des dossiers médicaux sensibles à la sécurisation des infrastructures industrielles critiques, des communications militaires et des mises à jour logicielles à distance dans les systèmes autonomes. Fondamentalement, ces protocoles sont conçus pour faire respecter quatre piliers de la sécurité de l'information : la confidentialité, l'intégrité, l'authenticité et la non-répudiation. Les architectures modernes de sécurité numérique atteignent ces propriétés en combinant diverses primitives cryptographiques. Les schémas de chiffrement symétrique préservent la confidentialité à grande échelle, les codes d'authentification de message assurent l'intégrité des données et la vérification de l'origine, et l'infrastructure à clés publiques fournit les signatures numériques et les protocoles d'échange de clés. Ensemble, ils établissent les bases de confiance nécessaires pour protéger l'économie numérique mondiale. L'intégrité de l'ensemble de cet écosystème numérique est aujourd'hui confrontée à une menace émergente majeure due aux progrès rapides du matériel d'informatique quantique. Propulsée par des investissements académiques et industriels considérables de la part de leaders technologiques tels qu'IBM, Google et Intel, l'ingénierie quantique est passée des modèles théoriques à des processeurs physiques de plus en plus denses. Contrairement aux ordinateurs classiques, qui traitent des bits déterministes restreints à un état de 0 ou 1 de manière séquentielle, les machines quantiques fonctionnent à l'aide de qubits. En exploitant le principe mécanique quantique de superposition, N qubits interconnectés peuvent représenter et manipuler 2^N états distincts simultanément, permettant un paradigme massif de parallélisme quantique. Bien que l'avantage quantique pratique reste contraint par de courts temps de cohérence et des taux d'erreur de porte élevés, la menace à long terme pour la cryptographie demeure un scénario futur réaliste. Spécifiquement, l'algorithme de Shor [Sho94] démontre une accélération superpolynomiale capable de résoudre la factorisation des entiers et les logarithmes discrets en temps polynomial. Ces deux problèmes mathématiques constituent le socle de presque toutes les normes cryptographiques asymétriques largement déployées, notamment RSA [RSA78], DSA [Nat13] et Elliptic Curve Cryptography (ECC) [Mil85, Kob87]. En conséquence, un ordinateur quantique scientifiquement pertinent compromettrait totalement les systèmes à clé publique actuels, rendant les signatures numériques falsifiables et exposant les canaux chiffrés.

Alors que les primitives symétriques comme AES [Dwo01] et les fonctions de hachage cryptographiques appartenant à la famille Secure Hashing Algorithm (SHA) ne sont que légèrement affectées, l'algorithme de Grover [Gro96] ne fournissant qu'une accélération quadratique pouvant être adéquatement compensée en doublant la taille des clés ou des empreintes, la cryptographie asymétrique ne dispose d'aucun recours mathématique similaire, ses hypothèses fondamentales étant fondamentalement brisées par les algorithmes quantiques. Pour faire face à cette vulnérabilité potentielle, la communauté cryptographique migre actuellement vers la Post-Quantum Cryptography (PQC). La PQC englobe une nouvelle classe d'algorithmes à clé publique conçus pour résister aux attaques des ordinateurs classiques et quantiques à grande échelle. De manière cruciale, plutôt que de reposer sur la physique quantique, ces algorithmes s'exécutent sur du matériel standard. Ils tirent leur sécurité de problèmes mathématiques multidimensionnels alternatifs, tels que des constructions basées sur les réseaux, les codes correcteurs d'erreurs et le hachage, qui restent informatiquement intraitables tant pour les machines traditionnelles que quantiques. Cette transition est nécessaire dès aujourd'hui car la mise à jour des systèmes mondiaux prend du temps, et les données chiffrées actuelles sont déjà vulnérables à l'interception et au stockage en vue d'un déchiffrement quantique futur.

La mobilisation mondiale en faveur d'architectures résistantes aux attaques quantiques a pris une structure formelle en 2016, lorsque le National Institute of Standards and Technology (NIST) a lancé une initiative internationale systématique pour identifier des remplaçants aux normes à clé publique existantes vulnérables à l'algorithme de Shor [Nat16]. Ce processus visait directement les spécifications établies pour l'établissement de clés et les signatures numériques. Cette orientation stratégique est étroitement reflétée à l'échelle internationale, notamment à travers des recommandations conjointes publiées par des autorités européennes telles que l'French National Agency for the Security of Information Systems (ANSSI) et le German Federal Office for Information Security (BSI), accentuant toutes deux sur une migration synchronisée et planifiée pour protéger les infrastructures numériques à long terme [NF24]. Contrairement aux compétitions de cryptographie symétrique précédentes qui ont désigné un seul gagnant (comme celles pour AES et SHA-3), le NIST a structuré le processus PQC de manière à sélectionner un portefeuille diversifié d'algorithmes. Pour évaluer les soumissions, qui dépassaient initialement 80 candidats, le NIST a établi cinq niveaux de sécurité distincts étalonnés par rapport à la résistance en bits classique et quantique des primitives symétriques existantes, allant d'AES-128 et SHA-256 jusqu'à AES-256. Les candidats ont été rigoureusement sélectionnés sur la base de leurs marges de sécurité mathématique, de leurs performances calculatoires (incluant la latence d'exécution, l'empreinte mémoire et les surcoûts de communication tels que la taille des clés et des textes chiffrés), et de leur adaptabilité dans divers environnements logiciels et matériels. Tout au long de cette évaluation en plusieurs étapes, la diversité mathématique est apparue comme une exigence critique pour atténuer le risque d'une unique percée cryptanalytique catastrophique. Cela a conduit à des sous-catégories distinctes au sein des familles mathématiques dominantes. Par exemple, au sein de la cryptographie basée sur les réseaux, une distinction claire existe entre les réseaux structurés et les réseaux non structurés, plus conservateurs. Un exemple éminent

est FrodoKEM [A+20]. Bien qu'il ait finalement été écarté des derniers tours du processus du NIST en raison de ses surcoûts importants en termes de performances et de taille de clés, il reste explicitement recommandé par des agences comme l'ANSSI pour les applications hautement sensibles car sa sécurité dérive directement d'hypothèses sous-jacentes du problème LWE non structuré. Inversement, pour la standardisation principale, le NIST a priorisé les variantes de réseaux structurés en raison de leur efficacité supérieure. Cela a culminé en 2022 avec la sélection de CRYSTALS-Dilithium et Falcon pour les signatures numériques, l'alternative basée sur le hachage à état SPHINCS+, et ML-KEM (développé à l'origine sous le nom de CRYSTALS-Kyber) comme norme définitive pour l'encapsulation de clés sous le FIPS 203 [NIS24b]. En tant que mécanisme basé sur les réseaux modulaires, ML-KEM atteint un compromis optimal entre solidité cryptographique et débit matériel. De manière cruciale, son architecture interne est profondément dépendante des primitives cryptographiques symétriques. Son cœur opérationnel utilise les fonctions SHA-3 et SHAKE (dérivées de la permutation Keccak) pour exécuter des opérations essentielles telles que la génération de matrices publiques, l'extension de graines pseudo-aléatoires et la génération d'erreurs. Par conséquent, l'efficacité opérationnelle et la consommation de ressources d'une implémentation matérielle de ML-KEM sont fondamentalement liées aux performances de son infrastructure SHA-3 sous-jacente. Enfin, pour maintenir la diversité algorithmique face aux vulnérabilités potentielles des hypothèses sur les réseaux, le NIST a étendu son évaluation à d'autres familles mathématiques. Alors que les candidats basés sur les isogénies comme SIKE ont été disqualifiés à la suite de vulnérabilités classiques en temps polynomial découvertes au cours du processus d'évaluation, les schémas basés sur les codes tels que Classic McEliece [BCL+17], BIKE [ABB+22] et HQC [MAB+18] ont été conservés pour une analyse approfondie. Cela a culminé au début de l'année 2025 avec la sélection officielle du protocole Hamming Quasi-Cyclic (HQC) pour servir de principale alternative normalisée basée sur les codes à ML-KEM, établissant avec succès un socle cryptographique structurellement diversifié pour l'ère post-quantique.

Parmi les différents cadres mathématiques stimulant cette transition, un problème fondateur principal pour nombre de ces schémas post-quantiques est le problème de l'apprentissage avec erreurs (LWE), initialement formulé par Regev [Reg09], qui établit un lien mathématique entre primitives cryptographiques à clé publique et la dureté dans le pire des cas des réseaux géométriques en grande dimension. Algébriquement, le problème LWE standard nécessite de calculer un produit matrice-vecteur sur un corps fini et de perturber ensuite le résultat en ajoutant un petit vecteur d'erreur extrait d'une distribution de probabilité spécifique. La traduction de ce cadre théorique en cryptosystèmes concrets et pratiques, tels que le mécanisme d'encapsulation de clés conservateur FrodoKEM, introduit des défis d'implémentation majeurs. Pour minimiser la bande passante de communication, les matrices publiques massives requises pour les opérations algébriques ne peuvent être stockées ou transmises directement. À la place, elles doivent être générées de manière déterministe à la volée à partir de courtes graines aléatoires provenant d'un True Random Number Generator (TRNG). Dans les implémentations conformes aux efforts de standardisation modernes, cette génération de matrice s'appuie fortement sur des primitives

cryptographiques symétriques dérivées de la permutation Keccak. La complexité structurelle sous-jacente de ces primitives est définie par la norme FIPS 202, qui spécifie la construction éponge consistant en des phases itératives d'absorption et d'essorage d'état. Au sein de ce cadre, une distinction opérationnelle critique existe entre les fonctions de hachage cryptographiques traditionnelles, qui appliquent des entrées de longueur arbitraire sur des sorties de longueur fixe, et les eXtendable-Output Functions (XOFs) telles que SHAKE. Ces architectures XOF peuvent produire un flux non borné et arbitraire de bits pseudo-aléatoires à partir d'une seule chaîne d'entrée, une caractéristique opérationnelle essentielle à la fois pour l'extension de matrice et pour les procédures d'échantillonnage d'erreur. Au-delà de la génération de matrice publique, le goulot d'étranglement calculatoire principal dans l'exécution des cryptosystèmes basés sur LWE réside dans la chaîne de génération explicite d'erreurs. Ce pipeline multi-étapes commence par l'absorption d'une valeur de graine véritablement aléatoire, se poursuit par l'extension XOF pour produire un flux de bits pseudo-aléatoires, et culmine dans un module d'échantillonnage dédié. Ce module transforme le flux de bits pseudo-aléatoires en valeurs discrètes conformes à une distribution de probabilité cible stricte, plus particulièrement la distribution gaussienne discrète, qui est largement adoptée pour garantir des marges de sécurité optimales [N+19]. La complexité architecturale élevée de la permutation éponge Keccak combinée aux exigences de précision strictes de la logique d'échantillonnage d'erreur introduit de sévères goulots d'étranglement au niveau des ressources. Pour cette raison, l'accélération matérielle du standard SHA-3 est largement explorée dans la littérature, se révélant cruciale pour les systèmes à contraintes matérielles, avec de nouvelles variantes architecturales encore proposées dans la littérature récente [CRA+24].

Ces goulots d'étranglement calculatoires se traduisent directement par des surcoûts physiques considérables lors de l'exécution de primitives post-quantiques sur des plateformes matérielles dédiées, impactant particulièrement la surface de silicium, la consommation d'énergie et l'efficacité énergétique globale. Pour faire face à ces limitations au niveau matériel, le calcul approximatif (Approximate Computing (AxC)) présente une alternative stratégique prometteuse. Généralement établi comme une philosophie de conception pour le développement de systèmes efficaces en énergie [HO13], ce paradigme détend délibérément l'exigence d'exactitude mathématique stricte, introduisant des erreurs fonctionnelles ou structurelles directement dans les calculs arithmétiques pour obtenir des réductions significatives de la complexité des circuits, de la dissipation de puissance et de la latence de traitement. Dans la littérature existante, l'application du calcul approximatif (AxC) est presque exclusivement restreinte aux applications intrinsèquement tolérantes aux erreurs, telles que le traitement numérique du signal, le traitement de flux multimédia et les architectures d'apprentissage automatique. À l'inverse, son application en ingénierie cryptographique reste rare, l'arithmétique non exacte étant généralement évitée en raison de craintes concernant les risques potentiels pour la sécurité ou les vulnérabilités par canaux cachés. Néanmoins, les architectures cryptographiques basées sur LWE présentent une intersection conceptuelle, car une erreur stochastique, mathématiquement bornée, constitue une exigence algorithmique explicite plutôt qu'un défaut. Par conséquent, exploiter des approxima-

tions matérielles contrôlées pour introduire nativement cette erreur requise offre une alternative de conception viable, bien que largement inexplorée, qui transforme une vulnérabilité de sécurité traditionnelle en une source d'optimisation architecturale.

La littérature existante examinant l'intersection entre le calcul approximatif et la cryptographie basée sur les réseaux se divise généralement en deux axes de recherche distincts. Le premier axe se concentre sur l'intégration d'unités arithmétiques approximatives pour simplifier les calculs intensifs de produits matrice-vecteur ou polynomiaux, principalement au sein de configurations Ring-LWE. Par exemple, des travaux initiaux ont exploré l'utilisation d'un multiplicateur impartial à dynamique variable (DRUM) pour optimiser la fonction de déchiffrement [BHS18], tandis que des conceptions ultérieures ont introduit des multiplicateurs modulaires approximatifs (AxMM) tronquant les valeurs gaussiennes discrètes signées de cinq bits à quatre bits [KBK$^+$20]. Des approximations similaires de multiplicateurs multiniveaux via la truncation d'échantillons d'erreur ont été proposées et analysées sous l'angle de la sécurité [KBW$^+$21, O$^+$22]. Bien que ces variations méthodologiques réussissent à optimiser les performances et l'empreinte matérielle du chemin de données algébrique, les méthodes basées sur la truncation ne caractérisent pas la distribution finale de l'erreur introduite et s'appuient toujours in fine sur des échantillonneurs gaussiens conventionnels pour générer explicitement l'erreur requise. Abordant le problème sous une perspective différente, un second axe de recherche explore la génération implicite de l'erreur cryptographique requise à travers les inexactitudes au sein des opérations arithmétiques sous-jacentes. Ce concept s'appuie sur le problème classique de l'apprentissage avec bruit (Learning Parity with Noise (LPN)), qui est une instance légère du problème LWE définie sur un corps binaire où une erreur de un bit est traditionnellement générée via un générateur matériel de nombres aléatoires et un échantillonneur de Bernoulli. Kamel et al. ont formalisé le remplacement de cette chaîne d'échantillonnage externe en introduisant le concept de Learning Parity with Physical Noise (LPPN), démontrant que les inexactitudes calculatoires intrinsèques au sein d'un produit scalaire inexact pouvaient servir directement de terme d'erreur requis [K$^+$20]. La réalisation pratique du principe LPPN a été ultérieurement validée à travers un prototype FPGA démontrant que des erreurs physiques dépendantes des données d'entrée ne compromettent pas significativement la sécurité [B$^+$21], un concept qui a ensuite été généralisé au cadre LWE standard sous le nom de Learning With Physical Errors (LWPE) [B$^+$22]. Dans ces premières implémentations d'erreur physique, les inexactitudes calculatoires nécessaires étaient induites par des techniques de surcadençage en tension (Voltage-Frequency Over-Scaling (VOS)) dépendantes de l'analogique. Cependant, faire fonctionner des circuits au-delà de leurs limites de sécurité introduit des contraintes pratiques majeures, car cela nécessite des procédures de calibrage complexes, des lignes de retard d'horloge variable précises, et des composants analogiques difficiles à protéger contre les manipulations physiques. Pour traiter la sécurité théorique de ces comportements physiques dépendants des données, ces formulations individuelles ont été récemment unifiées au sein d'un cadre analytique global nommé Learning With Errors with Output Dependencies (LWE-OD). Ce cadre modélise de manière abstraite une large classe de problèmes d'apprentissage où la distribution d'erreur sous-jacente dépend explicitement

de la valeur du produit scalaire calculé et, par conséquent, de la clé secrète. En utilisant un problème intermédiaire (noté qLWE) dans une approche en deux étapes basée sur un simulateur, cette méthodologie établit des conditions de vérification explicites garantissant qu'une instance LWE-OD demeure au moins aussi difficile sur le plan calculatoire que son équivalent LWE standard [HMRS26]. En définitive, la portée principale de ces cadres physiques et théoriques fondateurs réside dans leur démonstration de la faisabilité fondamentale de cette philosophie de conception, prouvant que des calculs approximatifs peuvent remplacer entièrement les échantillonneurs gaussiens traditionnels. En validant ce paradigme, ces travaux pionniers ouvrent la voie à des recherches ultérieures vers des stratégies d'approximation entièrement numériques et plus robustes, capables d'exploiter en toute sécurité les avantages de la génération d'erreur implicite.

Contributions de la thèse

Dans cette thèse, nous étudions le chevauchement interdomaine entre le paradigme du calcul approximatif (AxC) et les architectures cryptographiques basées sur les réseaux, en introduisant le paradigme Approximate Learning With Errors (AxLWE) pour redéfinir la génération d'erreur cryptographique. Dans le cadre des conventions traditionnelles de l'ingénierie cryptographique, l'approximation est systématiquement évitée parce que l'arithmétique non exacte est typiquement perçue comme une vulnérabilité de sécurité susceptible d'introduire des défauts déterministes ou de faciliter l'exploitation par canaux cachés. Ce travail remet en question cette vision établie en démontrant que dans le contexte spécifique du problème de l'apprentissage avec erreurs (LWE), une erreur stochastique, mathématiquement bornée, constitue une exigence algorithmique explicite plutôt qu'un défaut du système.

En alignant l'erreur cryptographique nécessaire à la sécurité basée sur les réseaux avec les inexactitudes fonctionnelles nativement produites par des composants matériels approximatifs, cette thèse présente une méthodologie inédite pour affranchir les systèmes des chaînes de génération d'erreur coûteuses, en exploitant des opérateurs approximatifs numériques de pointe pour introduire intrinsèquement l'erreur requise.

Comme étape préalable pour soutenir ce paradigme, la phase initiale de cette recherche se concentre sur le développement et l'analyse de l'étape de génération de pseudo-aléatoire sous-jacente à la chaîne de génération d'erreur. Ce travail préliminaire examine la complexité architecturale de la construction éponge Keccak définie par la norme FIPS 202, conduisant à la conception d'un accélérateur matériel conforme au standard, prenant en charge toutes les fonctions de hachage SHA-3 et les XOFs telles que SHAKE. Pour traiter les vulnérabilités physiques potentielles sur les plateformes embarquées, cette phase inclut une implémentation ciblée d'une variante masquée de premier ordre du cœur Keccak utilisant la technique Domain-Oriented Masking (DOM). Dans ce cadre, nous analysons les coûts de déploiement et les surcoûts en aléa frais typiquement associés aux architectures DOM, en mettant en œuvre un répartiteur d'aléa de support basé sur les chiffrements Trivium et Bivium et en évaluant l'implémentation

via la synthèse ASIC et une analyse Test Vector Leakage Assessment (TVLA) sur FPGA. Servant de brique matérielle préalable pour l'optimisation de la chaîne de génération d'erreur, ce développement établit un socle sécurisé et conforme à la norme pour la phase initiale d'extension de graines.

En nous appuyant sur ce socle cryptographique sécurisé, la thèse s'oriente vers l'analyse de la façon dont les implémentations conventionnelles basées sur les réseaux utilisent les flux de bits pseudo-aléatoires étendus pour exécuter une génération d'erreur explicite. En évaluant les stratégies d'implémentation courantes des échantillonneurs d'erreur dédiés, plus particulièrement les circuits gaussiens discrets, nous cartographions les pénalités de latence substantielles, les surcoûts de stockage et les complexités matérielles introduites par la conversion de bits pseudo-aléatoires en distributions de probabilité cibles précises. Pour répondre conceptuellement à ces limitations, ce travail s'appuie sur et étend le paradigme fondateur de l'apprentissage avec erreurs physiques (LWPE) [B$^+$22], qui a établi le premier la viabilité théorique du remplacement des échantillonneurs explicites par des inexactitudes calculatoires implicites. En étendant ce cadre sous-jacent vers un espace de conception entièrement numérique, nous proposons le paradigme AxLWE comme alternative aux pipelines traditionnels de génération d'erreur explicite.

Les premières implémentations d'erreur physique au sein du cadre LWPE reposaient sur des techniques de surcadençage en tension (VOS) dépendantes de l'analogique, nécessitant des procédures de calibrage complexes devant être exécutées au cas par cas sur chaque composant pour tenir compte de la variabilité des procédés de fabrication. De plus, les caractéristiques de l'erreur introduite de manière implicite sont intrinsèquement dépendantes de la technologie, ce qui limite la portabilité de telles architectures sur différentes structures matérielles. Pour surmonter ces limitations, la stratégie AxLWE proposée introduit des erreurs fonctionnelles et déterministes directement dans la multiplication matrice-vecteur en exploitant des composants numériques issus d'EvoApproxLib [M$^+$17], une bibliothèque open-source d'opérateurs approximatifs de pointe. En s'appuyant sur une conception entièrement numérique, cette approche atteint l'indépendance vis-à-vis de la technologie, améliorant considérablement la portabilité et la facilité de déploiement de l'architecture cryptographique à travers diverses technologies silicium.

Pour garantir que cette simplification au niveau arithmétique ne meurtrisse pas l'intégrité cryptographique, nous contribuons à une méthodologie de caractérisation systématique des opérateurs approximatifs numériques, en évaluant les candidats issus de bibliothèques de conception avancées sur la base de leurs paramètres exacts de distribution d'erreur et de leurs bornes statistiques. Cette méthodologie garantit que les erreurs introduites par les opérateurs approximatifs sont statistiquement alignées sur la distribution d'erreur spécifique requise par les schémas basés sur LWE afin de maintenir la sécurité cryptographique.

Enfin, cette thèse fait le pont entre validation théorique et application pratique en ciblant un cas d'usage post-quantique du monde réel, à savoir le protocole conservateur basé sur des réseaux non structurés FrodoKEM [A$^+$20]. L'implémentation du paradigme AxLWE au sein de

ce schéma normalisé nous permet de vérifier que notre méthodologie ne dépasse pas les limites strictes du taux d'échec de déchiffrement (DFR) désignées par la spécification de FrodoKEM, tout en démontrant que l'architecture approximative se conforme avec succès aux vecteurs de test connus (Known Answer Tests (KATs)) du NIST. En concevant, implémentant et synthétisant des accélérateurs matériels dédiés LWE et AxLWE complets, nous menons une analyse architecturale globale comparant des configurations fonctionnant avec et sans échantillonneurs d'erreur explicites traditionnels. Cette synthèse systémique fournit une évaluation globale du paradigme, démontrant que la technique AxLWE préserve la correction algorithmique lors des opérations cryptographiques tout en générant des bénéfices physiques majeurs par la réduction simultanée de la complexité matérielle structurelle, des empreintes de surface silicium, de l'absorption de puissance et de la consommation d'énergie globale.

Contents

1 Introduction **1**

2 Background **9**

2.1 Lattice-based Cryptography 14

2.2 The Learning With Errors (LWE) Problem 18

2.2.1 Problem Definitions 18

2.2.2 Problem Structure 18

2.2.3 Role of Randomness 19

2.2.4 Role of the Error in LWE 20

2.2.5 Hardness of LWE 20

2.2.6 Variants of LWE-based Problems 21

2.2.7 LWE-based Public Key Encryption 23

2.2.8 Decryption Failure Rate 24

2.2.9 Summary 25

3 LWE Implementations **27**

3.1 Pseudo-Randomness Generation 28

3.1.1 Introduction to True Random Number Generators (TRNG) 28

3.1.2 Introduction to Pseudorandom Number Generators (PRNG) 29

3.2 Error Sampling in LWE 33

3.2.1 Methodologies for Discrete Gaussian Sampling 35

3.2.2 LWE-Based Key Encapsulation Mechanisms 39

3.3 Summary 51

4 FIPS 202 Masked Hardware Accelerator 53

4.1 Side-Channel Attacks and Masking Countermeasures 54

4.1.1 Side-Channel Analysis 55

4.1.2 The Masking Countermeasure 56

4.1.3 Introduction to Test Vector Leakage Assessment (TVLA) 59

4.1.4 Masked ML-KEM Implementation 61

4.2 Existing Works 63

4.2.1 Masked Keccak Hardware Architectures 63

4.2.2 Randomness Generation for Masked Implementations 65

4.3 Architecture Description 66

4.3.1 Modes of Operation 67

4.3.2 Randomness Dispatcher 68

4.3.3 Integration in a 32-bit RISC-V System-on-Chip 70

4.4 Results and Comparison 71

4.4.1 Implementation Results on FPGA and ASIC 71

4.4.2 Security Analysis with Test Vector Leakage Assessment 73

4.4.3 Masked ML-KEM performance with masked SHA-3 Accelerator on a RISC-V SoC 74

4.4.4 Comparison with the State of the Art 76

4.5 Summary 77

5 Approximate Computing as Controlled Error Source for LWE-based Schemes 81

5.1 Introduction to Approximate Computing 82

5.1.1 Customizing Number Representation and Precision 83

5.1.2 Hardware-Level Approximations 83

5.1.3 Software-Level Approximations 84

5.1.4 Application-Level Approximations 84

5.2 Related Work on Approximate Lattice Cryptography 85

5.3 Replacing Gaussian Error Generation via VOS 86

5.4 From Explicit Error Sampling to Embedded Errors with AxLWE 89

5.5 Choosing a Suitable Approximate Operator 91

5.5.1 Approximate Adder Characterization Methodology 93
5.6 Approximate FrodoKEM Case Study 95
5.7 Approximate LWE Architecture 106
5.7.1 Baseline LWE Hardware Accelerator 107
5.7.2 AxLWE Hardware Accelerator 110
5.8 Results 113
5.9 Summary 115

6 Conclusion **119**

List of Figures **127**

List of Tables **131**

References **133**

Appendix A Source Code for Approximate Adders Characterization **147**

Appendix B Source Code Samples for Approximate FrodoKEM Implementation **153**

Chapter 1

Introduction

Cryptography serves as the indispensable cornerstone of modern information security, underpinning the trust, privacy, and infrastructure of our global digital society. Every day, billions of interconnected devices rely on cryptographic protocols to secure a vast spectrum of daily activities, ranging from routine financial transactions and the protection of sensitive medical records to securing critical industrial infrastructures, military communications, and over-the-air software updates in autonomous systems. Fundamentally, these protocols are designed to enforce four pillars of information security: confidentiality, integrity, authenticity, and non-repudiation. Modern digital security architectures achieve these properties by combining diverse cryptographic primitives. Symmetric encryption schemes preserve confidentiality at scale, message-authentication codes ensure data integrity and origin verification, and public-key infrastructure provides digital signatures and key exchange protocols. Cooperatively, they establish the trust baselines necessary to protect the global digital economy. The integrity of this entire digital ecosystem is now facing a critical emerging threat due to rapid advancements in quantum computing hardware. Propelled by substantial academic and industrial investments from technology leaders such as IBM, Google, and Intel, quantum engineering has transitioned from theoretical models to increasingly dense physical processors. Unlike classical computers, which process deterministic bits restricted to a state of either 0 or 1 sequentially, quantum machines operate using qubits. By exploiting the quantum mechanical principle of superposition, N interconnected qubits can represent and manipulate 2^N distinct states simultaneously, enabling a massive paradigm of quantum parallelism. While practical quantum advantage remains constrained by short coherence times and high gate error rates, the long-term threat to cryptography remains a realistic future scenario. Specifically, Shor's algorithm [Sho94] demonstrates a superpolynomial speedup capable of solving integer factorization and discrete logarithms in polynomial time. These two mathematical problems form the bedrock of almost all widely deployed asymmetric cryptographic standards, including Rivest–Shamir–Adleman (RSA) [RSA78], Digital Signature Standard (DSS) [Nat13], and Elliptic Curve Cryptography (ECC) [Mil85, Kob87]. Consequently, a cryptographically relevant quantum computer would fully compromise current public-key systems, leaving digital signatures forgeable and encrypted channels exposed. While symmetric primitives like Ad-

vanced Encryption Standard (AES) [Dwo01] and cryptographic hash functions belonging to the Secure Hashing Algorithm (SHA) family are only mildly affected, as Grover's algorithm [Gro96] provides merely a quadratic speedup that can be adequately countered by doubling key or output sizes, asymmetric cryptography possesses no such mathematical recourse, as its foundational assumptions are fundamentally broken by quantum algorithms. To address this potential vulnerability, the cryptographic community is currently transitioning to Post-Quantum Cryptography (PQC). PQC encompasses a new class of public-key algorithms designed to resist attacks from both classical and large-scale quantum computers. Crucially, rather than relying on quantum physics, these algorithms run on standard hardware. They derive their security from alternative, multidimensional mathematical problems, such as constructions based on lattices, error-correcting codes, and hashes, that remain computationally intractable for both traditional and quantum machines. This transition is necessary today because updating global systems takes time, and current encrypted data is already vulnerable to being intercepted and stored for future quantum decryption.

The global mobilization toward quantum-resistant architectures gained formal structure in 2016, when National Institute of Standards and Technology (NIST) launched a systematic international initiative to identify replacements for legacy public-key standards vulnerable to Shor's algorithm [Nat16]. This process directly targeted established specifications for key establishment and digital signatures. This strategic direction is closely mirrored internationally, notably through joint guidance issued by European authorities such as the French National Agency for the Security of Information Systems (ANSSI) and the German Federal Office for Information Security (BSI), both emphasizing a synchronized, planned migration to protect long-term digital infrastructure [NF24]. Unlike previous symmetric cryptography competitions that yielded a single winner (such as those for AES and SHA-3), NIST structured the PQC process to select a diverse portfolio of algorithms. To evaluate the submissions, which initially exceeded 80 candidate schemes, NIST established five distinct security levels benchmarked against the classical and quantum bit-strength of existing symmetric primitives, ranging from AES-128 and SHA-256 up to AES-256. Candidates were rigorously screened based on their mathematical security margins, their computational performance (including execution latency, memory footprint, and communication overheads like key and ciphertext sizes), and their adaptability across varied software and hardware environments. Throughout this multi-stage evaluation, mathematical diversity emerged as a critical requirement to mitigate the risk of a single catastrophic cryptanalytic breakthrough. This led to distinct sub-categories within the dominant mathematical families. For instance, within lattice-based cryptography, a clear distinction exists between structured lattices and more conservative, unstructured lattices. A prominent example is FrodoKEM [A$^+$20]. Although it was eventually omitted from the later rounds of the NIST process due to its substantial performance and key-size overheads, it remains explicitly recommended by agencies like ANSSI for highly sensitive applications because its security derives directly from foundational, unstructured Learning With Errors (LWE) problem assumptions. Conversely, for primary standardization, NIST prioritized structured lattice variants

due to their superior efficiency. This culminated in 2022 with the selection of CRYSTALS-Dilithium and Falcon for digital signatures, the stateful hash-based alternative SPHINCS+, and ML-KEM (originally developed as CRYSTALS-Kyber) as the definitive standard for key encapsulation under FIPS 203 [NIS24b]. As a module-lattice-based mechanism, ML-KEM achieves an optimal trade-off between cryptographic strength and implementation performance. Crucially, its internal architecture is deeply reliant on symmetric cryptographic primitives. Its core design utilizes SHA-3 and SHAKE functions (derived from the Keccak permutation) to perform essential operations such as public matrix generation, pseudorandom seed expansion, and error generation. Consequently, the operational efficiency and resource consumption of an ML-KEM hardware implementation are fundamentally bound to the performance of its underlying SHA-3 implementation. Finally, to maintain algorithmic diversity against potential vulnerabilities in lattice assumptions, NIST extended its evaluation to alternative mathematical families. While isogeny-based candidates like SIKE were disqualified following classical polynomial-time vulnerabilities discovered during the review process, code-based schemes such as Classic McEliece [BCL$^+$17], BIKE [ABB$^+$22], and HQC [MAB$^+$18] were retained for further analysis. This culminated in early 2025 with the formal selection of the Hamming Quasi-Cyclic (HQC) scheme to serve as the primary standardized code-based alternative to ML-KEM, successfully establishing a structurally diverse cryptographic baseline for the post-quantum era.

Among the various mathematical frameworks driving this transition, a primary foundational problem for many of these post-quantum schemes is the Learning With Errors (LWE) problem, originally formulated by Regev [Reg09], which establishes a mathematical link between public-key cryptographic primitives and the worst-case hardness of high-dimensional geometric lattices. Algebraically, the standard LWE problem requires computing a matrix-vector product over a finite field and subsequently perturbing the result by adding a small error vector drawn from a specific probability distribution. Translating this theoretical framework into concrete, practical cryptosystems, such as the conservative key encapsulation mechanism FrodoKEM, introduces major implementation challenges. To minimize communication bandwidth, the massive public matrices required for the algebraic operations cannot be stored or transmitted directly. Instead, they must be deterministically generated on-the-fly from short, random seeds sourced from a TRNG. In implementations compliant with modern standardization efforts, this matrix generation relies heavily on symmetric cryptographic primitives derived from the Keccak permutation. The underlying structural complexity of these primitives is defined by the FIPS 202 standard, which specifies the sponge construction consisting of iterative state absorption and squeezing phases. Within this framework, a critical operational distinction exists between traditional cryptographic hash functions, which map arbitrary inputs to fixed-length outputs, and XOFs such as SHAKE. These XOF architectures can produce an arbitrary, unbound stream of pseudo-random bits from a single input string, an operational characteristic that is essential for both matrix expansion and error sampling procedures. Beyond public matrix generation, the primary computational bottleneck in executing LWE-based cryptosystems resides within the explicit error generation chain. This multi-stage pipeline begins with the absorption of a truly random seed value, proceeds

through XOF expansion to yield a pseudo-random bitstream, and culminates in a dedicated sampling module. This module transforms the pseudo-random bitstream into discrete values that conform to a strict target probability distribution, most notably the discrete Gaussian distribution, which is widely adopted to ensure optimal security margins [N+19]. The high architectural complexity of the Keccak sponge permutation combined with the strict precision requirements of the error sampling logic introduces severe resource bottlenecks. For this reason, the hardware acceleration of the SHA-3 standard is widely explored in the literature, proving crucial for hardware-constrained systems, with new architectural variants still being proposed in recent research [CRA+24].

These computational bottlenecks translate directly into substantial physical overheads when executing post-quantum primitives on dedicated hardware platforms, particularly impacting silicon area, power consumption, and overall energy efficiency. To address these hardware-level limitations, AxC presents a promising alternative strategy. Generally established as a design philosophy for energy-efficient system development [HO13], this paradigm deliberately relaxes the requirement of strict mathematical exactness, introducing functional or structural errors directly into the arithmetic computations to achieve significant reductions in circuit complexity, power dissipation, and processing latency. Within the existing literature, the application of AxC is almost exclusively restricted to inherently error-tolerant applications, such as digital signal processing, multimedia streaming, and machine learning architectures [CCRR13, Mit16, VCRR15]. Conversely, its application in cryptographic engineering remains rare, as non-exact arithmetic is typically avoided due to concerns over potential security risks or side-channel vulnerabilities [CRPS26]. Nevertheless, LWE-based cryptographic architectures present a conceptual intersection, because a mathematically bounded, stochastic error is an explicit algorithmic requirement rather than a flaw. Consequently, leveraging controlled hardware approximations to natively introduce this required error offers a viable, yet largely unexplored, design alternative that turns a traditional security vulnerability into a source of architectural optimization.

Existing literature examining the intersection of approximate computing and lattice-based cryptography is generally divided into two distinct research streams. The first stream focuses on integrating approximate arithmetic units to simplify the intensive matrix-vector or polynomial computations, primarily within Ring-LWE configurations. For instance, initial efforts explored the usage of a Dynamic Range Unbiased Multiplier to optimize the decryption function [BHS18], while subsequent designs introduced Approximate Modular Multipliers (AxMM) that truncate signed discrete Gaussian values from five bits to four bits [KBK+20]. Similar multi-level multiplier approximations via error sample truncation have been proposed and analyzed from a security perspective [KBW+21, O+22]. Although these methodology variations succeed in optimizing the performance and footprint overheads of the algebraic data path, truncation-based methods do not characterize the final distribution of the introduced error and still ultimately rely on conventional Gaussian samplers to explicitly generate the required error. Approaching the problem from a different perspective, a separate research stream explores generating the

required cryptographic error implicitly through inaccuracies within the underlying arithmetic operations. This concept builds upon the classic LPN problem, which is a lightweight instance of the LWE problem defined over a binary field where a one-bit error is traditionally generated via a hardware random number generator and a Bernoulli sampler. Kamel et al. [K+20] formalized the replacement of this external sampling chain by introducing LPPN, demonstrating that intrinsic computational inaccuracies within an inexact inner product could serve directly as the required error term. The practical realization of the LPPN principle was subsequently validated through an FPGA prototype demonstrating that input-dependent physical errors do not significantly compromise security [B+21], a concept that was later generalized to the standard LWE framework as LWPE [B+22]. In these initial physical error implementations, the necessary computational inaccuracies were induced through analog-dependent VOS techniques. However, operating circuits beyond their safe limits introduces significant practical constraints, as it requires complex calibration procedures, precise variable clock delay lines, and analog components that are difficult to protect against physical tampering. To address the theoretical security of these data-dependent physical behaviors, these individual formulations were recently unified under a comprehensive analytical framework known as LWE-OD. This framework abstractly models a broad class of learning problems where the underlying error distribution depends explicitly on the computed inner product value and consequently on the secret key. By utilizing an intermediate problem (denoted as qLWE) within a two-step, simulator-based approach, this methodology establishes explicit verification conditions ensuring that an LWE-OD instance remains at least as computationally hard as its standard LWE counterpart [HMRS26]. Ultimately, the primary significance of these foundational physical and theoretical frameworks lies in their demonstration of the fundamental feasibility of this design philosophy, proving that approximate computations can entirely replace traditional Gaussian samplers. By validating this paradigm, these pioneering works pave the way for further research into more robust, fully-digital approximation strategies that can securely exploit the benefits of implicit error generation.

Thesis Contribution

In this thesis, we investigate the cross-domain overlay between the paradigm of Approximate Computing (AxC) and lattice-based cryptographic architectures, introducing the AxLWE paradigm to redefine the generation of cryptographic error. Within the traditional conventions of cryptographic engineering, approximation is systematically avoided because non-exact arithmetic is typically perceived as a security vulnerability that could introduce deterministic flaws or facilitate side-channel exploitation. This work challenges that established view by demonstrating that within the specific context of the Learning With Errors (LWE) problem, a mathematically bounded, stochastic error is an explicit algorithmic requirement rather than a system defect. By aligning the cryptographic error necessary for lattice-based security with the functional inaccuracies natively produced by approximate hardware components, this thesis presents a

novel methodology to overcome expensive error generation chains, utilizing state-of-the-art digital approximate operators to inherently introduce the required error.

As a preliminary step to support this paradigm, the initial phase of this research focuses on the development and analysis of the pseudo-randomness generation stage underlying the error generation chain. This preliminary work examines the architectural complexity of the Keccak sponge construction defined by the FIPS 202 standard, leading to the design of a standard-compliant hardware accelerator that supports all SHA-3 hash functions and eXtendable-Output Functions (XOFs) such as SHAKE. To address potential physical vulnerabilities on embedded platforms, this phase includes a targeted implementation of a first-order masked variant of the Keccak core using DOM. Within this scope, we analyze the deployment costs and fresh randomness overheads typically associated with DOM architectures, implementing a supportive randomness dispatcher based on Trivium and Bivium ciphers and evaluating the implementation via ASIC synthesis and FPGA-based TVLA. Serving as a preliminary building block for the optimization of the error generation chain, this development establishes a secure, standard-compliant baseline for the initial seed expansion phase.

Building upon this secure cryptographic baseline, the thesis transitions to analyzing how conventional lattice-based implementations utilize expanded pseudo-random bitstreams to execute explicit error generation. By evaluating the common implementation strategies of dedicated error samplers, most notably discrete Gaussian circuits, we map out the substantial latency penalties, storage overheads, and hardware complexities introduced by converting pseudo-random bits into precise target probability distributions. To address these limitations conceptually, this work builds upon and extends the foundational paradigm of Learning With Physical Errors (LWPE) [B$^+$22], which first established the theoretical viability of replacing explicit samplers with implicit computational inaccuracies. By expanding this underlying framework into a fully digital design space, we propose the AxLWE paradigm as an alternative to traditional explicit error generation pipelines. Early physical error implementations within the LWPE framework relied on analog-dependent VOS techniques, which require complex calibration procedures that must be executed on a per-device basis to account for manufacturing process variability. Moreover, the characteristics of the implicitly introduced error are inherently technology-dependent, limiting the portability of such designs across different hardware fabrics. To overcome these limitations, the proposed AxLWE strategy introduces functional, deterministic errors directly into the matrix-vector multiplication by leveraging digital components from EvoApproxLib [M$^+$17], an open-source library of state-of-the-art approximate operators. By relying on a fully-digital design, this approach achieves technology independence, substantially improving the portability and ease of deployment of the cryptographic architecture across diverse silicon technologies.

To ensure that this arithmetic-level simplification does not compromise cryptographic integrity, we contribute a systematic characterization methodology for digital approximate operators, evaluating candidates from advanced design libraries based on their exact error distribution parameters and statistical bounds. This methodology ensures that the errors introduced by

the approximate operators are statistically aligned to the specific error distribution required by LWE-based schemes to maintain cryptographic security.

Finally, this thesis bridges theoretical validation and practical application by targeting a real-world post-quantum use case, specifically the conservative, unstructured lattice-based protocol FrodoKEM [A+20]. Implementing the AxLWE paradigm within this standardized scheme allows us to verify that our methodology does not exceed the strict Decryption Failure Rate (DFR) boundaries designated by the FrodoKEM specification, while also demonstrating that the approximate architecture successfully complies with the NIST KATs. By designing, implementing, and synthesizing complete LWE and AxLWE dedicated hardware accelerators, we conduct a global architectural analysis comparing configurations operating with and without traditional explicit error samplers. This systemic synthesis provides a comprehensive evaluation of the paradigm, demonstrating that the AxLWE technique preserves algorithmic correctness during cryptographic operations while yielding major physical benefits by simultaneously reducing structural hardware complexity, silicon area footprints, power absorption, and overall energy consumption.

Thesis Outline

Chapter 2 establishes the necessary theoretical foundations of post-quantum cryptography by presenting the fundamental properties, structural configurations, and cryptographic variants of the Learning With Errors (LWE) problem. It details the core computational formulation $\mathbf{As} + \mathbf{e} = \mathbf{b} \pmod{q}$, highlighting the strict parametric error boundaries required to prevent decoding threshold breaches, decryption failures, and key-reconstructing attacks. Finally, it reviews modern algebraic adaptations, including Ring-LWE (RLWE), Module-LWE (MLWE), Learning with Rounding (LWR), and Learning With Errors with Output Dependencies (LWE-OD), mapping out the mandatory theoretical framework for analyzing approximate hardware acceleration paradigms.

Chapter 3 details the operational blocks and structural sequences required to transition abstract lattice-based primitives into practical implementations. It begins with pseudo-randomness generation, analyzing true hardware entropy sources alongside deterministic seed-expansion primitives, including shift-register stream ciphers and Keccak sponge constructions. Next, it examines the multi-stage error sampling chain, focusing on discrete Gaussian distribution shaping methodologies like cumulative distribution tables and the Knuth–Yao methodology. Finally, after establishing all necessary building blocks, the chapter integrates these elements into full key-encapsulation workflows, providing practical implementation details about the unstructured FrodoKEM and module-structured ML-KEM protocols.

Chapter 4 details the system-level validation and physical security assessment of a first-order masked SHA-3 hardware accelerator optimized for post-quantum cryptography and embedded

within a 32-bit RISC-V System-on-Chip (SoC). The architecture features a one round-per-cycle Keccak datapath based on DOM and integrates a Randomness Dispatcher to protect intermediate states against glitch-induced leakage. Evaluated using a standardized `ML-KEM.Decaps` routine running on the RISC-V-based SoC, the hardware delivers an overall system-level speedup factor of $2\times$. The chapter provides a transparent account of real area-security trade-offs via synthesis on an Artix-7 FPGA and a standard-cell 22nm FD-SOI library, fully validated through electromagnetic Test Vector Leakage Assessment (TVLA).

Chapter 5 introduces the approximate Learning With Errors (AxLWE) paradigm, demonstrating how digital hardware approximation can be harnessed as an implicit source of controlled errors to optimize PQC hardware implementations. By embedding an approximate adder like `add16u_0GN` directly into the internal accumulation step of the $\mathbf{A} \cdot \mathbf{s}$ inner product, the required cryptographic error is introduced directly during computation, completely removing explicit Gaussian sampler logic. This approach is validated through a software-emulated FrodoKEM-640 case study (proving a decryption failure rate consistent with the original implementation) and through a protocol-independent hardware prototype to evaluate the benefits in silicon implementations. Digital synthesis results demonstrate quantitative optimization metrics, yielding substantial savings across total silicon area, execution latency, power dissipation, and cumulative energy efficiency.

Chapter 6 concludes the thesis by summarizing the central contributions presented in the previous chapters and discussing potential directions for future research.

Chapter 2

Background

Historically, *cryptography* (derived from the Greek term *kryptos*, meaning hidden) is considered a relatively young formal science whose rigorous academic study began approximately one hundred years ago. Nevertheless, the physical evidence of altering information for security purposes is ancient, with early traces dating back to around 1900 BC found within the inscriptions of ancient Egyptian tombs. In more recent times, cryptography has evolved into a comprehensive framework of studies, techniques, and practices engineered to guarantee secure communications between two legitimate entities in the explicit presence of malicious third parties, also commonly referred to as adversaries. A communication channel is defined as secure when the underlying messages or data shared between the two communicating parties cannot be accessed or understood by an unauthorized observer. The primary objective of an adversary is to act as a malicious entity whose aim is to prevent the users of a cryptosystem from achieving their goals, primarily protecting the confidentiality, integrity, and availability of data, though an adversary can also counterfeit the explicit identities of the participating entities in alternative scenarios.

The primary application of cryptography is data encryption, which is structurally defined by a coordinated set of elements. The original, readable data structure is referred to as the *plaintext*, which represents any readable set of data such as a message, a document, or a file. Applying an encryption algorithm, also called a *cipher*, to this plaintext in combination with a specific key transforms the readable data into an unreadable, obfuscated format known as the *ciphertext*. The underlying cryptographic algorithm defines the precise mathematical rules used to perform this transformation via the process of *encryption*, which encodes data to transform plaintext into ciphertext, as well as the reverse process of *decryption*, which decodes data to retrieve the original plaintext from a given ciphertext. The cryptographic *key* serves as the essential variable input to the cryptographic algorithm, uniquely determining the exact nature of the forward transformation of plaintext into ciphertext and its reverse operation.

To realize secure communications, a cryptosystem must be carefully engineered to satisfy several essential concepts. The system must enforce *confidentiality*, which means preserving authorized restrictions on access and disclosure, including explicit means for protecting personal

privacy and proprietary information, ensuring that unauthorized actors are completely unable to access the real value or meaning of a certain set of data. Alongside confidentiality, the system must maintain *integrity*, which is the process of guarding against improper information modification or destruction such that it can be guaranteed that the received message is exactly the one originally sent by the sender, assuming that data remains in its intended state and can only be edited by authorized parties. Furthermore, the infrastructure must handle *authentication* to formally verify the explicit identity of a user, a process, or a device, often as a strict prerequisite for allowing access to resources in an information system. This is accompanied by *authorization*, which is the process of giving an entity explicit permission to access a specific resource or function, possibly granting modification rights to some extent. Finally, the system must provide *non-repudiation*, which ensures that an entity cannot deny having originated, sent, or approved a specific message or document, typically guaranteed through digital signatures.

The discussion surrounding cryptographic algorithms can be exceptionally complex, though this work focuses specifically on lattice-based primitives grounded in the LWE problem, particularly the internal Keccak-based pseudo-random expansion and discrete error sampling mechanisms that drive its execution. From a high-level perspective, cryptographic algorithms can be divided into three main categories consisting of *hash functions*, *symmetric cryptography*, and *asymmetric cryptography*. Another useful procedure frequently deployed alongside these categories is a Key Encapsulation Mechanism (KEM), which is a primitive employed to secure symmetric cryptography elements for transmission by means of asymmetric cryptography, a hybrid technique that is discussed thoroughly in the following sections. A hash function (Figure 2.1) is a versatile, one-way cryptographic algorithm that maps an input string of any size, called a message, to a unique output of a fixed length of bits, commonly known as a hash or a digest. Hash functions must be strictly deterministic, meaning that whenever identical input values are supplied, the algorithm will always create the exact same output digest. To be secure, they must be pre-image resistant, meaning that it is computationally infeasible to reverse a hash value to recover the original plaintext, and collision resistant, a property that guarantees a different unique digest for each distinct input. Hashing is highly useful for ensuring the authenticity of a piece of data and verifying that it has not been tampered with, because even a small change in the input message will create an entirely different output hash, a characteristic feature also known as the avalanche effect. Consequently, hash functions are one of the basic tools of modern cryptography used in information security to authenticate transactions, messages, and digital signatures. While some commonly used historical hash functions like Message Digest 5 (MD5) and SHA-1 are now considered completely insecure due to discovered collision vulnerabilities [WY05, SBK$^+$17], modern cryptosystems rely on standardized suites such as SHA-2 and SHA-3 [Nat15a, Nat15b], high-throughput alternatives like BLAKE3 [OANWO20], or specialized algorithms, notably Argon2, which is very resilient to brute-force attacks.

According to NIST definitions, an eXtendable-Output Function (XOF) (Figure 2.2) represents an advanced variant of the classic cryptographic hash where the output can be dynamically extended to any desired length. It can be viewed as an extension of the cryptographic hash that

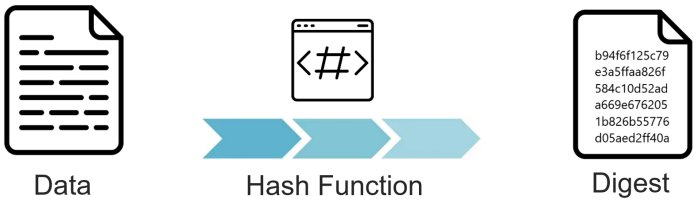


Figure 2.1 Hash Function

allows its output stream to be arbitrarily long. In particular, the sponge construction, introduced in Section 3.1.2, makes any sponge-based hash a natural XOF. Consequently, standard XOFs such as SHAKE128 and SHAKE256 can either generate arbitrarily long pseudorandom streams or produce fixed-length digests on demand.

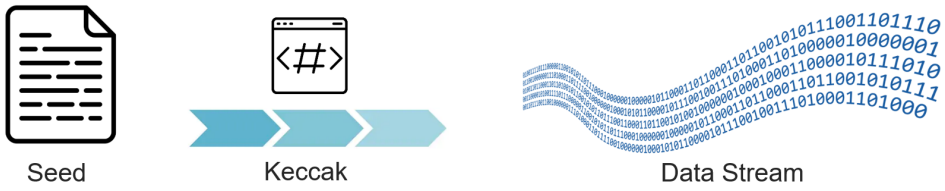


Figure 2.2 eXtendable-Output Function

Symmetric-key algorithms, also known simply as ciphers, are commonly used to protect the confidentiality of data, such as within storage devices or in any application where the secure exchange of the secret key is not an immediate concern. While symmetric ciphers can be used for transmitting encrypted data, it is critically important to not send the key alongside the ciphertext, otherwise an attacker can intercept the stream and easily decode the message. The term symmetric comes from the fact that the exact same key is used for both encryption and decryption operations, establishing a direct and fast method that is widely employed across modern digital systems. Symmetric-key encryption can take advantage of two main computational solutions, namely *stream ciphers* and *block ciphers*. A stream cipher is a symmetric cryptographic primitive that encrypts plaintext bit by bit (or digit by digit) by combining it with a synchronized pseudorandom keystream. While computationally efficient, stream ciphers are inherently malleable and susceptible to bit-flipping attacks unless coupled with dedicated message authentication mechanisms. A block cipher instead processes fixed-size blocks of data, meaning that if a message is larger than one block, the encryption is split over multiple blocks and repeated, utilizing padding techniques involving bits or characters that fill up unused portions of a data structure whenever the message length is not an exact multiple of the block size. Historically, the Data Encryption Standard (DES) established early protocols, though its original implementation is considered too weak and has been substituted by 2DES and 3DES. A more modern cipher intensively adopted over the last decades is the Advanced Encryption Standard (AES), which has served as the definitive standard since 2001.

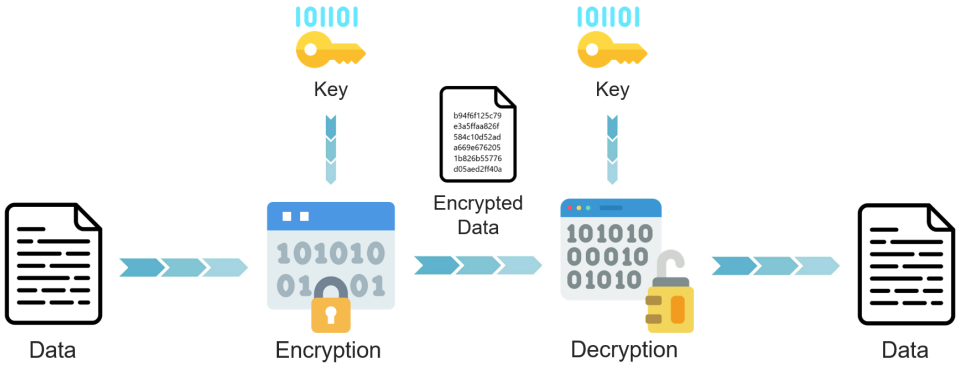


Figure 2.3 Symmetric-Key Encryption [Fre]

Asymmetric cryptography (Figure 2.4), also known as public-key cryptography, overcomes the core key-distribution constraints of symmetric systems by utilizing a mathematically linked pair of keys to encrypt and decrypt data. This pair of keys includes a *public key*, which can be shared openly with anyone, and a *private key*, which must be kept strictly secret by the owner. In asymmetric encryption, the sender uses the recipient's public key to encrypt the data, and the recipient then uses their private key to decrypt it, offering a higher security level between two parties without the need for both parties to share the same secret key, which represents a huge advantage. Public-key cryptography is able to guarantee confidentiality and authenticity in modern cryptosystems, making it useful for data encryption and decryption as well as digital signatures, which encompass the data authentication process. This latter process is obtained by generating a unique cryptographic signature using the sender's private key, typically computed over a hash digest of the underlying data. Any receiver can then verify the integrity and origin of the message by processing the data alongside the signature using the sender's widely accessible public key. Moreover, digital signatures also provide non-repudiation, a service used to provide assurance of the integrity and origin of data in such a way that it can be verified and validated by a third party as having originated from a specific entity in possession of that private key. Among the most widely deployed public-key primitives is the Rivest–Shamir–Adleman (RSA) cryptosystem [RSA78], employed for asymmetric cryptography and named after its inventors Rivest, Shamir, and Adleman.

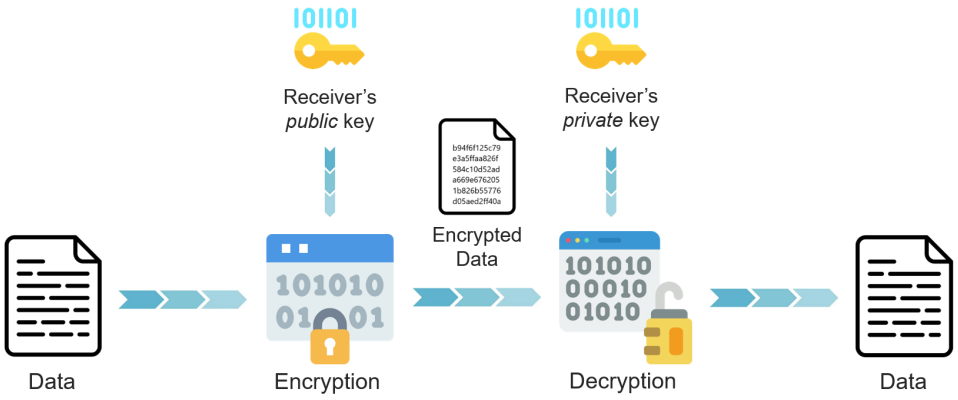


Figure 2.4 Asymmetric-Key Encryption [Fre]

To exploit the computational throughput of symmetric ciphers alongside the flexible key-management advantages of public-key systems, modern networks deploy hybrid cryptosystems supported by a Key Encapsulation Mechanism (KEM). A KEM (Figure 2.5) is a cryptographic primitive that allows anyone in possession of some party's public key to securely generate, encapsulate, and transmit a symmetric secret key to that party. A KEM can be viewed as a key-exchange protocol in which only a single message is transmitted, and its main application lies in securing symmetric key material for transmission using asymmetric public-key algorithms to achieve public-key encryption of messages of arbitrary length.

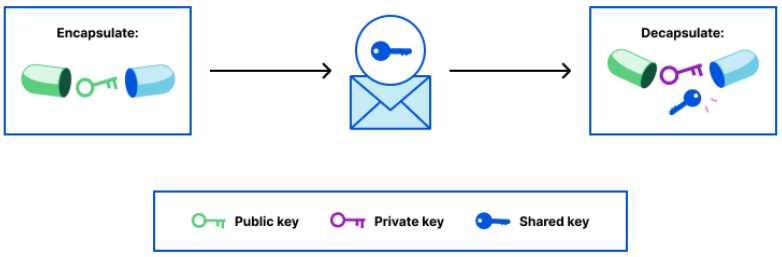


Figure 2.5 Key Encapsulation Mechanism (KEM) [GT]

Public-key cryptography is indispensable for the security of open computer networks, particularly the Internet, and most modern communications are protected by digital signatures and public-key cryptosystems. However, as Peter Shor's seminal 1994 work demonstrates [Sho94], the public-key cryptography used today is threatened by quantum computer attacks. This vulnerability arises because standard algorithms like RSA, which is based on the large integer factoring problem, or ECC, which relies on the discrete logarithm problem, are solvable in subexponential time by quantum machines. Over the last decades, huge research efforts have focused on the physical realization of quantum computers, and their success has brought substantial technological progress. If large-scale quantum computers are successfully constructed, they will be able to break many of the public-key cryptosystems currently in use, seriously compromising the confidentiality and integrity of digital communications on the Internet and elsewhere. The goal of post-quantum cryptography, also called quantum-resistant cryptography, is to develop cryptographic systems that are secure against both quantum and classical computers while interoperating with existing communications protocols and networks. Since converting our current digital infrastructure to post-quantum algorithms is a very long and complex process, NIST called for proposals with the intent of initiating a standardization procedure in 2017.

Post-Quantum Cryptography (PQC) is substantially divided into distinct mathematical families that offer different levels of confidence and trade-offs [Ste18]. Hash-based cryptography builds upon the classic example of Merkle's hash-tree public-key signature system established in 1979 [Mer79, Mer89], which built upon the one-message-signature ideas of Lamport [Lam79] and Diffie [DH22]. This family can only be used to generate digital signatures and cannot perform

public key encryption, requiring large signatures for many signature systems despite maintaining exceptionally high confidence. Code-based cryptography is exemplified by McEliece's hidden Goppa-code public-key encryption system introduced in 1978 [McE78], representing a long-studied class of cryptosystems with moderately high confidence for specific code families that faces engineering challenges regarding communication and key sizes. Multivariate-quadratic-equations cryptography is highlighted by examples like Patarin's HFEv- public-key signature system from 1996 [Pat96], which generalized a proposal by Matsumoto and Imai [MI88], offering a wide variety of systems with various levels of confidence and performance trade-offs. Elliptic curve isogenies represent a specialized but promising technique utilizing non-constant functions defined on an elliptic curve that take values on another elliptic curve and preserve point addition, achieving small communication footprints at the expense of slower computational latency. Finally, lattice-based cryptography has attracted the most intense interest, featuring historical examples like the Hoffstein-Pipher-Silverman NTRU public-key encryption system from 1998 [HPS98], as well as modern algorithms like CRYSTALS-Kyber [ABD+19] now standardized in FIPS 203 and recognizable as ML-KEM [NIS24b]. This family features a high level of academic interest, flexible constructions, and reasonable communication sizes while continuously developing confidence, encapsulating foundational formulations such as learning with errors (LWE), ring-LWE, and learning with rounding. This brief introduction provides the global context surrounding post-quantum cryptography, leading directly into our focused background analysis of lattice-based cryptography and the LWE problem over module lattices.

2.1 Lattice-based Cryptography

Lattice-based cryptography represents a modern paradigm in the field of information security, gaining significant prominence as a primary pillar of post-quantum cryptographic standards due to its exceptional structural flexibility and balanced communication footprints. This cutting-edge cryptographic approach harnesses the mathematical properties of multi-dimensional lattice structures to build encryption schemes, key-encapsulation mechanisms, and digital signature protocols. Rather than relying on classical algebraic problems that are vulnerable to quantum acceleration, its security is built upon the presumed computational hardness of geometric challenges defined over discrete point sets in high-dimensional spaces.

Intuitively, a *lattice* can be conceptualized as a regular, repeating arrangement of points spanning an n-dimensional real vector space $\mathbb{R}^n$, obtained by replicating a fundamental geometric pattern throughout the entire space. In abstract algebra, a lattice is formally modeled as a partially ordered set in which every arbitrary pair of elements possesses both a unique least upper bound (the join operation) and a greatest lower bound (the meet operation), connected systematically via the absorption law [DP02]. However, in modern post-quantum cryptography, the term refers strictly to a *geometric point lattice* [MG02]. From this geometric perspective, a lattice $\mathscr{L} \subset \mathbb{R}^n$ is defined as the set of all integer linear combinations of k linearly independent basis vectors

$\mathbf{b}_1, \ldots, \mathbf{b}_k \in \mathbb{R}^n$:

$$\mathscr{L}(\mathbf{B}) = \left\{ \sum_{i=1}^{k} z_i \mathbf{b}_i \;\middle|\; z_i \in \mathbb{Z} \right\} \tag{2.1}$$

These vectors constitute a *basis* $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_k)$, where k represents the rank of the lattice. When the rank equals the dimension of the ambient space ($k = n$), the lattice is said to be full-rank. A graphical representation of such point configurations is illustrated in Figure 2.6.

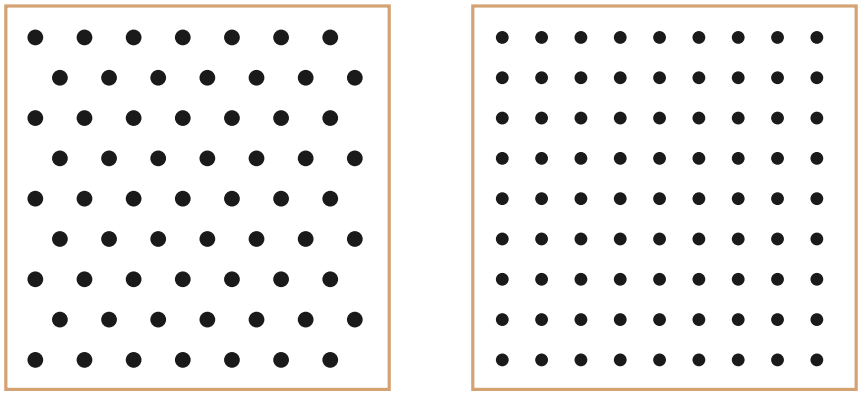


Figure 2.6 Lattice graphical examples showing an unstructured, staggered point distribution (left) alongside a highly regular, dense orthogonal grid structure (right).

As can be immediately noticed from the visual representation, different choices of basis vectors exhibit distinct geometric characteristics and can generate an infinite variety of lattice configurations, despite tiling the underlying space uniformly. The internal structural geometry of a lattice is frequently evaluated using its fundamental parallelepiped, which defines the n-dimensional unit volume spanned directly by the basis vectors with coefficients strictly bounded within the range $[0, 1)$, mathematically represented as:

$$\mathscr{F} = \left\{ \sum_{i=1}^{k} \mathbf{a}_i \mathbf{b}_i \middle| a_i \in [0, 1) \right\}$$

When a lattice is mapped such that $\mathscr{L} \subset \mathbb{Z}^n$, it is recognized as an *integer lattice*. Even though these repeating point structures appear visually simple, they lay the structural foundations for exceptionally difficult mathematical problems, paving a highly promising path for post-quantum algorithms.

To generate a lattice, it is mandatory to define an underlying vector basis. Crucially, this basis is never unique, meaning that multiple distinct sets of bases can generate the exact same lattice point set. For the sake of ease, considering a standard two-dimensional lattice, the available bases can be broadly divided into two sets: good bases and bad bases. A basis belonging to the good set features basis vectors that are reasonably short and nearly orthogonal to one another. On the contrary, a bad lattice basis is characterized by long, highly non-orthogonal, and skewed vectors, as illustrated in Figure 2.7. This geometric contrast establishes the core security asymmetry of

public-key lattice primitives, where the public keys expose only a bad basis of the lattice, while the corresponding secret keys safely conceal a good basis.

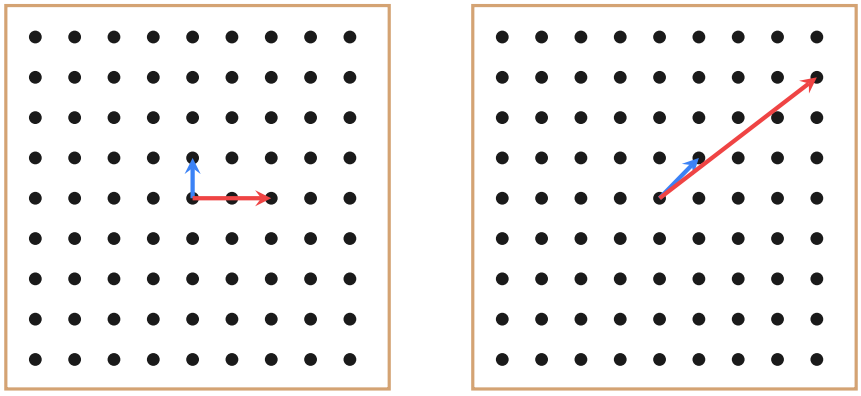


Figure 2.7 Geometric comparison of a good lattice basis (left), consisting of short, orthogonal vectors, versus a bad lattice basis (right) for the same point set, characterized by long, highly non-orthogonal, and skewed vectors.

The most thoroughly established hard computational problems defined over these discrete spaces are the Shortest Vector Problem (SVP) and the Closest Vector Problem (CVP). The SVP asks an algorithm to determine a non-zero lattice vector that minimizes the Euclidean length, which is formally defined as:

$$\|\mathbf{v}\| = \min\{\|\mathbf{u}\| \mid \mathbf{u} \in \mathscr{L},\ \mathbf{u} \neq \mathbf{0}\}$$

Similarly, the CVP requires identifying a specific lattice point that is geometrically closest to an arbitrary, non-lattice target point in space. While these problems are straightforward to solve in low-dimensional lattices by executing exhaustive searches or exploiting near-orthogonal vectors, they become increasingly harder as the considered lattice dimension n grows. In high-dimensional spaces, the geometry becomes immensely complex as lattice points grow exponentially dense, rendering the identification of the true shortest or closest vector practically impossible. Because the best-known algorithms demand running times that scale exponentially with n, these tasks become computationally intractable for dimensions in the low hundreds. Conversely, when a good basis is available, efficient approximation algorithms, such as Babai's nearest plane method [Bab86], can be applied to make these geometric operations more tractable.

To illustrate how this mathematical gap maps onto a practical public-key scenario, consider two characters, Sandy the sender and Reece the receiver, who need to exchange a secure message. Reece sets up a high-dimensional lattice and derives two separate bases capable of generating the exact same point distribution. The first is a short, near-orthogonal good basis, while the second is a highly skewed bad basis. Reece keeps the good basis secret to serve as their private key and exposes the bad basis to the network as their public key. Sandy can then use this public key to embed a message into the lattice structure, representing the underlying data as a specific coordinate point generated via the public bad basis. Before transmitting the coordinate, Sandy deliberately shifts the point, implying the addition of a small error vector. Upon receiving the

noisy coordinate, Reece can seamlessly decode the message by finding the closest genuine lattice point to the received coordinate, a task that reduces to solving the CVP efficiently by utilizing their private good basis. If an unauthorized eavesdropper intercepts the message, they are forced to solve the identical CVP instance using only the public bad basis, which is computationally unfeasible in high dimensions. This specific formulation was historically pioneered by the GGH encryption scheme [GGH97], which was unfortunately later proven insecure due to structural vulnerabilities inherent to its error padding and basis exposure [HJ16].

Despite the failure of early frameworks like GGH, lattices remain the most promising foundational core for quantum-resistant cryptography. To achieve rigorous security guarantees, most practical, modern lattice-based cryptographic constructions do not rely directly on worst-case instances of SVP or CVP. Instead, they leverage average-case problems that are mathematically designed to inherit the worst-case hardness of high-dimensional geometric lattices through reduction proofs. The most influential of these average-case formulations is the Learning With Errors (LWE) problem [Reg09]. In its standard decisional form over the ring of integers modulo q ($\mathbb{Z}_q$), the LWE problem requires an adversary to distinguish between a set of noisy linear equations and uniformly random data samples. Concretely, an individual LWE sample is represented as a tuple:

$$(\mathbf{a}, \mathbf{b}) = (\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e) \pmod{q}$$

where the vector $\mathbf{a} \in \mathbb{Z}_q^n$ is chosen uniformly at random, $\mathbf{s} \in \mathbb{Z}_q^n$ is a fixed, secret vector, and e is a small error term sampled from a specific probability distribution χ. The insertion of this small error term is mathematically vital as, without it, recovering the secret vector s would be a trivial task achievable via standard Gaussian elimination. With the error present, however, the problem becomes intractable in high dimensions.

To optimize operational throughput and minimize communication bandwidth overheads in resource-constrained hardware environments, modern protocols deploy structured variants of the LWE problem. One prominent variant is RLWE, which substitutes vectors with polynomials defined over a cyclotomic ring $R_q = \mathbb{Z}_q[x]/(f(x))$, allowing for faster polynomial multiplications using the Number-Theoretic Transform (NTT). An even more flexible generalization is the MLWE problem, where secrets, samples, and errors are elements of structured modules over R_q of dimension k. By manipulating the module rank k, designers can tune the balance between hardware execution performance and cryptographic security margins. These advanced, structured mathematical frameworks form the theoretical core of primary post-quantum standards, including the key-encapsulation mechanism ML-KEM (governed by FIPS 203 [NIS24b]) and the digital signature algorithm ML-DSA (governed by FIPS 204 [NIS24a]). Consequently, exploring the acceleration, optimization, and security of these underlying algebraic structures represents a vital frontier for modern hardware security engineering.

2.2 The Learning With Errors (LWE) Problem

The Learning With Errors (LWE) problem, formally introduced by Oded Regev in his seminal 2005 work, serves as the mathematical foundation for modern post-quantum public-key cryptography [Reg09]. Structurally, the problem generalizes classic parity learning paradigms to higher modular domains and can be conceptualized as decoding from random linear codes in the presence of errors [Reg09]. To establish an accessible framing for its structural properties, the LWE problem can be modeled as an exchange between two communication entities, Sandy the sender and Reece the receiver. Reece maintains a secret vector $\mathbf{s}$, which acts as a private key, while exposing a system of linear equations that serves as the corresponding public key. While evaluating a standard linear system is computationally trivial, the systematic introduction of localized error terms transforms these relations into an intractable computational puzzle, ensuring long-term cryptographic security against both classical and quantum adversaries [Reg09].

2.2.1 Problem Definitions

The LWE problem is formally evaluated in two distinct mathematical variants: the *Search formulation* and the *Decisional formulation* [Reg09]. Both definitions operate over a finite ring of integers $\mathbb{Z}_q$, where $q \geq 2$ represents a fixed modulus parameter [Reg09].

1. **Search LWE ($LWE_{q,\chi}$):** Let $\mathbf{s} \in \mathbb{Z}_q^n$ be a fixed, secret target vector chosen uniformly at random. We define $\mathscr{A}_{\mathbf{s},\chi}$ as the probability distribution over $\mathbb{Z}_q^n \times \mathbb{Z}_q$ obtained by selecting a coordinate vector $\mathbf{a} \in \mathbb{Z}_q^n$ uniformly at random, sampling an independent error term $e \in \mathbb{Z}_q$ from a specified error distribution χ, and outputting the tuple:

$$(\mathbf{a}, b) = (\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e) \pmod{q} \tag{2.2}$$

   where $\langle \cdot, \cdot \rangle$ denotes the standard inner product operation. Given access to an arbitrary number of independent samples drawn from this distribution $\mathscr{A}_{\mathbf{s},\chi}$, the objective of the Search LWE problem is to fully recover the exact secret vector $\mathbf{s}$.

2. **Decisional LWE:** In the decisional variant, the secret vector $\mathbf{s} \in \mathbb{Z}_q^n$ is generated uniformly at random, defining the sample distribution $\mathscr{A}_{\mathbf{s},\chi}$. The goal of the adversary is to function as a statistical distinguisher, determining with non-negligible advantage whether a provided sequence of polynomial samples is sampled genuinely from $\mathscr{A}_{\mathbf{s},\chi}$, or if it is drawn entirely from the uniform distribution $\mathscr{U}$ over $\mathbb{Z}_q^n \times \mathbb{Z}_q$.

2.2.2 Problem Structure

From an algebraic standpoint, when multiple equations are grouped together to process a transaction, the underlying system can be modeled using compact matrix notation. Let n

represent the security dimension, and let m define the total number of available equations or samples. We define the public coefficient matrix as $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$, the error vector as $\mathbf{e} \in \mathbb{Z}_q^m$, and the public result vector as $\mathbf{b} \in \mathbb{Z}_q^m$.

When no error is present within the configuration, the system corresponds to a standard over-determined system of linear equations where $m > n$:

$$\mathbf{As} = \mathbf{b} \pmod{q} \tag{2.3}$$

As noted, extracting the secret vector $\mathbf{s}$ from Equation 2.3 is computationally trivial, requiring only $O(n)$ independent equations to solve completely via Gaussian elimination in polynomial time. To construct a secure cryptographic primitive, the structure is deliberately perturbed by introducing the error vector $\mathbf{e}$, mapping the configuration onto the matrix-vector relation shown in Figure 2.8:

$$\mathbf{As} + \mathbf{e} = \mathbf{b} \pmod{q} \tag{2.4}$$

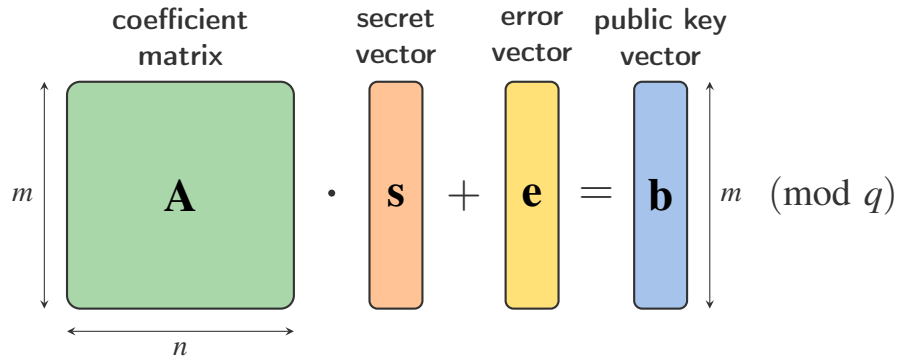


Figure 2.8 Matrix-vector parameter composition of the Learning With Errors problem framework.

All operations within this equation are strictly evaluated using the principles of modular arithmetic over a fixed modulus q. Because the added error vector alters the expected results, the system becomes overdefined with high probability, yielding no exact, clean solution through classical matrix inversion techniques.

2.2.3 Role of Randomness

Randomness acts as a foundational component in establishing the security guarantees of the LWE framework. The coefficient matrix $\mathbf{A}$ must be sampled uniformly at random from $\mathbb{Z}_q^{m \times n}$ to ensure that every individual equation represents an entirely independent, random linear combination across the modular space. Similarly, the secret vector $\mathbf{s}$ must be selected uniformly from $\mathbb{Z}_q^n$ to prevent attackers from exploiting localized structural biases or non-uniformities within the search space. This extensive randomization ensures that the resulting vectors $\mathbf{b}$ generated via genuine $\mathscr{A}_{\mathbf{s},\chi}$ samples closely mimic a completely uniform distribution. By randomizing both the transformation matrix and the underlying secret variables, the system successfully eliminates

fixed geometric patterns, preventing specialized algebraic optimization attacks and forcing an adversary to evaluate the problem as a generalized decoding instance over an entirely random linear code.

2.2.4 Role of the Error in LWE

The insertion of the error term is mathematically fundamental as, without it, recovering the secret vector $\mathbf{s}$ would be a trivial task achievable via standard Gaussian elimination using a small number of equations in polynomial time. The error masks the precise algebraic intersection of the linear equations, breaking the perfect alignment required by classical reduction algorithms. In theoretical security proofs, the error distribution χ is modeled as a discrete Gaussian distribution centered closely at zero with a standard deviation parameter σ, which corresponds to the discretized distribution $\bar{\Psi}_\alpha$ evaluated in foundational mathematical reductions [Reg09]. This continuous-to-discrete distribution mapping ensures that the error effectively conceals the underlying lattice structure from unauthorized observers without destabilizing the core algebraic relationship for authorized keyholders. Crucially, the architectural viability of modern lattice-based design relies on the fact that the geometric profile of this error can be altered without compromising security. As demonstrated in [MP13], the LWE problem remains hard even if the errors are non-Gaussian, provided that they are unpredictable enough to mask the secret $\mathbf{s}$ from an attacker, yet sufficiently small to ensure that the underlying algebraic relationship remains recoverable. This theoretical boundary provides the mathematical justification for practical cryptographic primitives to modify their error structures to optimize hardware performance and mitigate deployment vulnerabilities. For example, ML-KEM employs a centered binomial distribution that closely mirrors a Gaussian curve but allows hardware accelerators to generate error via simple bit-counting operations, completely bypassing expensive look-up tables. To safeguard designs against side-channel timing leaks associated with variable-time non-uniform sampling routines, ML-DSA implements a bounded uniform distribution, allowing raw pseudo-random bits to map directly onto error terms without requiring statistical shaping logic. Finally, alternative frameworks completely eliminate explicit randomized sampling units through LWR, generating the requisite public error deterministically through simple bit-truncation bottlenecks during modular scaling operations.

2.2.5 Hardness of LWE

The cryptographic utility of the LWE problem depends on its provable worst-case to average-case hardness reduction. [Reg09] demonstrates through a quantum reduction that if an efficient classical algorithm exists to solve typical, average-case instances of the LWE problem, then there must exist an equivalent, efficient quantum algorithm capable of solving worst-case instances of foundational geometric lattice problems. These worst-case challenges include the Shortest Independent Vectors Problem (SIVP) and the decisional version of the Shortest Vector

Problem (GapSVP) scaled within a polynomial factor $\tilde{O}(n/\alpha)$. The security margin of an LWE deployment is regulated by an interconnected triad of parameters, namely the security dimension n, the modular size q, and the error distribution width σ. The dimension n functions as the primary security parameter. Because the best-known classical and quantum algorithms require execution times that scale exponentially with n, increasing the dimension rapidly shifts the problem beyond the bounds of computational feasibility. Concurrently, the ratio between σ and q must be carefully balanced. If the error-to-modulus ratio is configured too low, the system becomes vulnerable to specialized algebraic combinations; conversely, if the ratio is set too high, the accumulated error completely overwhelms the mathematical relationship, rendering the system correct but non-functional for legitimate decryption or verification operations.

2.2.6 Variants of LWE-based Problems

While the standard formulation of the Learning With Errors problem provides robust worst-case security guarantees, its large key sizes and quadratic memory footprints pose significant deployment bottlenecks. To transition these primitives into practical cryptographic implementations, several structural variants have been proposed. These variants introduce specialized algebraic rings, deterministic functions, or alternative error behaviors to optimize hardware execution and storage requirements.

Ring-LWE (RLWE)

To compress the substantial memory footprint of standard public keys, Lyubashevsky et al. introduced the Ring-LWE variant [LPR10]. This formulation shifts the mathematical operations from standard vectors and matrices to polynomial rings evaluated over a quotient ring:

$$R_q = \mathbb{Z}_q[x]/\langle f(x)\rangle$$

where $f(x)$ is an irreducible polynomial of degree d, most commonly chosen as $f(x) = x^d + 1$ where d is a power of two. In this setting, the secret element $s(x)$, the public uniform challenge $a(x)$, and the error term $e(x)$ are all polynomials within R_q. A public sample is represented as a polynomial pair $(a(x), b(x))$ defined by:

$$b(x) = a(x) \cdot s(x) + e(x) \pmod{f(x), q}$$

From a structural perspective, a single polynomial multiplication in R_q is algebraically equivalent to a matrix-vector multiplication using a structured, quasi-cyclic matrix block. This layout allows a single row of d polynomial coefficients to implicitly represent an entire $d \times d$ matrix, compressing public key storage requirements from quadratic to linear scaling.

Module-LWE (MLWE)

Module-LWE, introduced by Langlois and Stehlé, acts as a flexible, hybrid framework that bridges the architectural gap between standard LWE and Ring-LWE [LS15]. Instead of utilizing a single massive polynomial ring or a completely unstructured matrix, Module-LWE expresses linear relations using vectors and matrices composed of small polynomials. The secret is structured as a vector of k polynomials $\mathbf{s} \in R_q^k$, and the public matrix $\mathbf{A} \in R_q^{m \times k}$ consists of uniform polynomial elements. The public vector $\mathbf{b} \in R_q^m$ is generated via:

$$\mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e} \pmod{q}$$

where each component of the error vector $\mathbf{e}$ is an independent polynomial with small coefficients. This configuration provides ideal microarchitectural scalability since, instead of restructuring hardware acceleration units to accommodate a completely different polynomial degree d when security requirements change, designers can simply adjust the module dimension k to scale the cryptographic strength. This modular agility underpins finalized standards such as ML-KEM and ML-DSA.

Learning with Rounding (LWR)

Introduced by Banerjee et al., Learning with Rounding offers a deterministic approach to generating public keys by eliminating randomized error sampling [BPR12]. Given a uniform public matrix $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$ and a secret vector $\mathbf{s} \in \mathbb{Z}_q^n$, the public observation vector $\mathbf{b}$ is computed by scaling and rounding the unperturbed matrix product from the operational modulus q down to a smaller target modulus p:

$$\mathbf{b} = \left\lfloor \frac{p}{q} \cdot (\mathbf{A}\mathbf{s}) \right\rceil \pmod{p}$$

In this environment, no standalone error vector is explicitly generated or added to the system. Instead, dropping the least significant bits via the rounding function introduces a deterministic truncation error that behaves mathematically like the random error vectors found in traditional LWE. This structural modification simplifies microarchitectural logic by entirely removing the need for dedicated pseudorandom error generators.

Learning With Errors with Output Dependencies (LWE-OD)

To unify abstract mathematical theory with the physical realities of hardware execution, Hoffmann et al. formalized the Learning With Errors with Output Dependencies (LWE-OD) framework [HMRS26]. In all prior configurations, the error distribution is strictly independent of the error-free product. LWE-OD addresses scenarios where the distribution of the error term explicitly depends on the exact value of the underlying inner product $\langle \mathbf{a}, \mathbf{s} \rangle$. This advanced model acts as a broad mathematical umbrella that captures standard LWE, LWR, and hardware

implementations involving physical data dependencies. It provides a formal tool to analyze the cryptographic hardness of designs where hardware acceleration paradigms, such as approximate computing or low-power processing units, inject an operational error profile that remains intrinsically tied to the computational output.

2.2.7 LWE-based Public Key Encryption

In order to fully understand the role of the error term in LWE-based Post-Quantum Cryptography (PQC), it is necessary to understand how the LWE can be used to build a Public Key Encryption (PKE) scheme, where two parties aim at sharing a message *msg*, which can be used as a symmetric key for future encrypted communications:

- **Key Generation (Algorithm 1):** This procedure generates the static key pair starting from two seeds, $seed_A$ and $seed_{s,e}$, obtained from a True Random Number Generator (TRNG). The matrix $\mathbf{A}$ is generated by expanding $seed_A$ through a Pseudorandom Number Generator (PRNG). Similarly, the secret $\mathbf{s}$ and the error $\mathbf{e}$ are produced by passing a PRNG-expanded sequence from $seed_{s,e}$ into a Gaussian sampler ($\text{SAMPLE}_{\text{Gauss}}$). The public vector $\mathbf{b}$ is then computed via the LWE matrix-vector product, resulting in the public key $pk = (\mathbf{A}, \mathbf{b})$ and the secret key $sk = \mathbf{s}$.

- **Encryption (Algorithm 2):** To encrypt a message *msg*, the sender generates an ephemeral secret $\mathbf{s}'$ and error terms $\mathbf{e}', \mathbf{e}''$ using a similar PRNG and Gaussian sampling chain. Using the public key $(seed_A, \mathbf{b})$, the sender re-generates $\mathbf{A}$ and computes the ciphertext vectors $\mathbf{u}$ and $\mathbf{v}$. Specifically, the $\text{ENCODE}()$ function maps the message into the higher-order bits of the finite field to maximize the error margin, ensuring that the introduced error terms do not interfere with the original data. The procedure outputs the ciphertext $(\mathbf{u}, \mathbf{v})$.

- **Decryption (Algorithm 3):** The recipient uses the private key $\mathbf{s}$ to extract the noisy message msg_{noisy} from the ciphertext by computing $\mathbf{v} - \mathbf{us}$. Expanding this operation reveals the complete algebraic derivation:

$$msg_{noisy} = (\mathbf{s}'\mathbf{As} + \mathbf{s}'\mathbf{e} + \mathbf{e}'' + \text{ENCODE}(msg)) - (\mathbf{s}'\mathbf{A} + \mathbf{e}')\mathbf{s}$$

  By the associative property of matrix multiplication, the term $(\mathbf{s}'\mathbf{A})\mathbf{s}$ is mathematically equivalent to $\mathbf{s}'(\mathbf{As})$. This equivalence allows the algebraic term $\mathbf{s}'\mathbf{As}$ to cancel out, which isolates the encoded message alongside the accumulated error without requiring commutativity:

$$msg_{noisy} = \text{ENCODE}(msg) + \mathbf{s}'\mathbf{e} + \mathbf{e}'' - \mathbf{e}'\mathbf{s}$$

  Finally, to retrieve the original message, the $\text{DECODE}()$ function performs a threshold-based rounding operation to remove these remaining error terms.

Correct decryption is only guaranteed if the total accumulated error remains below a specific threshold (typically $q/4$). If the error, including any hardware-induced approximation errors, exceeds this budget, the DECODE() function will round to an incorrect value, causing a decryption failure. Consequently, when replacing the sampler with approximate hardware, controlling the magnitude and bias of the intrinsic hardware error is mandatory.

Algorithm 1 LWE - Key Generation

Procedure KEYGENERATION()
  $seed_A \leftarrow \text{TRNG}()$
  $seed_{s,e} \leftarrow \text{TRNG}()$
  $\mathbf{A} \leftarrow \text{PRNG}(seed_A)$
  $\mathbf{s}, \mathbf{e} \leftarrow \text{SAMPLE}_{Gauss}(\text{PRNG}(seed_{s,e}))$
  $\mathbf{b} \leftarrow \mathbf{As} + \mathbf{e} \pmod q$
  $pk \leftarrow (seed_A, \mathbf{b}), \quad sk \leftarrow \mathbf{s}$
  return (pk, sk)
End Procedure

Algorithm 2 LWE - Encryption

Procedure ENCRYPT(pk, msg)
  $(seed_A, \mathbf{b}) \leftarrow pk$
  $\mathbf{A} \leftarrow \text{PRNG}(seed_A)$
  $seed_{s,e} \leftarrow \text{TRNG}()$
  $\mathbf{s}', \mathbf{e}', \mathbf{e}'' \leftarrow \text{SAMPLE}_{Gauss}(\text{PRNG}(seed_{s,e}))$
  $\mathbf{u} \leftarrow \mathbf{s}'\mathbf{A} + \mathbf{e}' \pmod q$
  $v \leftarrow \mathbf{s}'\mathbf{b} + e'' + \text{ENCODE}(msg) \pmod q$
  $ciphertext \leftarrow (\mathbf{u}, \mathbf{v})$
  return $ciphertext$
End Procedure

Algorithm 3 LWE - Decryption

Procedure DECRYPT($sk, ciphertext$)
  $(\mathbf{u}, \mathbf{v}) \leftarrow ciphertext$
  $\mathbf{s} \leftarrow sk$
  $msg_{noisy} \leftarrow v - \mathbf{us} \pmod q$
  $msg \leftarrow \text{DECODE}(msg_{noisy})$
  return msg
End Procedure

2.2.8 Decryption Failure Rate

The reference metric for the maximum acceptable error is the *Decryption Failure Rate* (DFR), defined as the probability that a decryption operation will fail to correctly recover the original plaintext from the ciphertext. In many modern cryptographic schemes, particularly those based on the LWE problem, there is a small, inherent chance that the involved parties fail to derive the

correct shared message. It has been established that the ability to induce decryption failures in such schemes can leak information about the secret key, potentially leading to full key recovery if enough failures are collected [DVV18]. Consequently, maintaining a negligible DFR is a critical design constraint for LWE-based schemes.

In brief, this is because such failures are not purely random events but they are directly influenced by the specific secret key being used. Because of this dependency, a failing ciphertext indicates a correlation between the secret key and the randomness used during encryption. An adversary can exploit this by searching for weak ciphertexts that are more likely to trigger a failure, a technique known as failure boosting. By collecting and analyzing these failures, an attacker can derive clues about the secret, effectively reducing the overall security of the system by estimating the key from the leaked information. Therefore, when modifying the underlying arithmetic, it is essential to ensure that the resulting error remains within the bounds necessitated by these security considerations to prevent creating new vulnerabilities through increased decryption failures.

2.2.9 Summary

This section presented the foundational properties, structural configurations, and cryptographic variants of the Learning With Errors (LWE) problem. The core takeaways of the LWE framework comprise its computational formulation, parametric error boundaries, and architectural adaptations.

At its mathematical core, the security of an LWE-based primitive relies on the computational hardness of distinguishing linear relationships containing errors from completely uniform randomness, a property backed by rigorous quantum reductions to worst-case geometric lattice challenges. The problem is fundamentally characterized by the security dimension n, the number of available equations m, and the operational modulus q. From an algebraic standpoint, this configuration is expressed as an overdefined matrix-vector product operating within the finite modular field $\mathbb{Z}_q$:

$$\mathbf{As} + \mathbf{e} = \mathbf{b} \pmod{q}$$

where $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$ represents a public, uniformly random coefficient matrix, $\mathbf{s} \in \mathbb{Z}_q^n$ acts as the hidden secret key, $\mathbf{e} \in \mathbb{Z}_q^m$ is a localized error vector, and $\mathbf{b} \in \mathbb{Z}_q^m$ is the resulting public key vector observed by an adversary.

The scope and magnitude of the error term $\mathbf{e}$ are governed by strict mathematical boundaries that establish a critical operational trade-off. If the error term is configured to be too small or is completely omitted, the system simplifies to a standard linear system that can be solved in polynomial time via classical Gaussian elimination, compromising the security guarantees of the cryptosystem. On the other hand, when utilizing the LWE problem to construct practical cryptographic systems, the error magnitude must be carefully managed to preserve functional

correctness. To ensure that a legitimate recipient can unambiguously separate the underlying message from the introduced error, the protocol relies on a deterministic threshold-based decoding mechanism. This decoding routine operates effectively only if the total accumulated error remains within an allocated budget, which strictly limits the error to an absolute magnitude below $q/4$. If the error breaches this $q/4$ boundary, it crosses the decoding threshold, causing the rounding operation to incorrectly map the perturbed state to an invalid plaintext symbol and triggering a decryption failure. Beyond disrupting the basic functionality of the system, an elevated Decryption Failure Rate (DFR) introduces severe security vulnerabilities. Because these failures are not purely random occurrences but are structurally correlated with the underlying secret key, they enable an adversary to gather critical algorithmic hints. By tracking which ciphertexts fail, a malicious actor can mount side-channel failure-boosting attacks to systematically reconstruct the private key variables.

To balance these stringent security boundaries with real-world deployment requirements, modern post-quantum standards implement specialized algebraic variations of the problem. While theoretical proofs rely on discrete Gaussian error distributions, practical protocols deploy structural adaptations to optimize hardware throughput. These adaptations include Ring-LWE (RLWE) and Module-LWE (MLWE) to compress public key storage footprints via structured polynomial matrices, Learning with Rounding (LWR) to replace randomized error sampling with deterministic bit truncation, and LWE-OD to model execution environments where physical data dependencies alter the underlying error behavior. Understanding this delicate interplay between parametric boundaries and error generation mechanisms forms the mandatory theoretical foundation for analyzing approximate hardware acceleration paradigms.

Chapter 3

LWE Implementations

The transition of lattice-based primitives from abstract mathematical foundations into deployable cryptographic software and hardware systems introduces distinct structural and microarchitectural challenges. While theoretical security proofs assume access to ideal random sources and exact probability distributions, practical implementations must construct these distributions deterministically from pseudorandom bitstreams. This chapter explores the concrete structural and algorithmic components that bridge the gap between Learning With Errors (LWE) theory and functional protocol execution.

First, the architecture of pseudorandom and true randomness generation is analyzed, contrasting lightweight shift-register streams with standardized, permutation-based sponge constructions specified by the FIPS 202 standard. Second, the structural error generation chains required to shape raw uniform bitstreams into localized, non-uniform error distributions are detailed to expose their underlying architectural and computational complexity, offering a targeted overview of the primary discrete Gaussian sampling methodologies found in cryptographic literature. Finally, these individual mathematical and algorithmic blocks are mapped onto complete protocol workflows by analyzing two prominent post-quantum Key Encapsulation Mechanisms (KEMs): the standard unstructured LWE baseline framework of FrodoKEM, and the module-structured Module-LWE (MLWE) framework of ML-KEM.

Crucially, tracing the data flow across these distinct protocols reveals that symmetric cryptographic permutations, specifically those driven by the Keccak primitive, are executed iteratively throughout every phase of key generation, encapsulation, and decapsulation, creating a massive computational bottleneck. In parallel, the practical instantiation of these error generation chains introduces non-trivial complexities, particularly regarding side-channel resistance, memory allocation, and precision constraints. To mitigate these design challenges, this thesis also addresses optimizations within the sampling pipeline, exploring implementation paradigms that aim to minimize or eliminate the requirement for complex, resource-intensive Gaussian samplers entirely. By establishing these comprehensive protocol definitions, this chapter provides the foundational system-level context and algorithmic baseline necessary to justify and guide the

development of the dedicated hardware acceleration architectures and co-design optimizations presented in the subsequent parts of this work.

3.1 Pseudo-Randomness Generation

In the formal mathematical formulations of the LWE problem (Section 2.2), foundational parameters such as the coefficient matrix $\mathbf{A}$ and the secret vector $\mathbf{s}$ are strictly defined as being sampled from uniform distributions over the finite modular space $\mathbb{Z}_q$. This theoretical assumption ensures that every individual element within the algebraic structures possesses an identical probability of selection, maximizing the entropy of the system and preventing adversaries from exploiting structural or directional biases.

When transitioning from these abstract mathematical definitions to real-world cryptographic systems, the requirement for a uniform distribution directly dictates the architecture of the underlying system randomness. In practical hardware and software deployments, these vast uniform matrices and vectors cannot be assumed inherently but they must be actively constructed using robust, unpredictable entropy sources. Modern schemes materialize this theoretical uniformity by integrating physical True Random Number Generators (TRNG) to capture localized environmental entropy, generating a highly secure, unpredictable seed. This seed is subsequently expanded through deterministic, cryptographically secure Pseudo-Random Number Generators (PRNGs) to generate the large, uniform public blocks efficiently. Consequently, the mathematical necessity of a uniform distribution in the theory translates directly into a strict engineering requirement for secure, high-throughput cryptographic randomness within the practical implementation.

3.1.1 Introduction to True Random Number Generators (TRNG)

To instantiate the uniform distributions required by the theoretical foundations of lattice-based schemes, practical electronic systems must integrate dedicated hardware mechanisms capable of generating physical entropy. A True Random Number Generator (TRNG) fulfills this role by harvesting non-deterministic microscopic phenomena inherent to the underlying hardware substrate. In modern complementary metal-oxide-semiconductor (CMOS) architectures and field-programmable gate array (FPGA) deployments, common practices for entropy harvesting exploit thermal noise, shot noise, telegraph noise, or radioactive decay. On an architectural level, these microscopic fluctuations are typically captured via clock jitter in Ring Oscillators (RO), metastable behavior in cross-coupled latches or flip-flops, or phase noise in phase-locked loops [BLBF11, WT08]. The captured analog perturbations are subsequently digitized through sampling circuitry to produce a continuous stream of high-entropy bits.

Despite their theoretical necessity for absolute security, the physical implementation of a TRNG presents significant engineering difficulties and strict resource trade-offs. Physical entropy

sources are intrinsically susceptible to environmental conditions, including Process, Voltage, and Temperature (PVT) variations, as well as localized power supply noise and aging effects [VL03]. These non-ideal environmental factors can systematically introduce statistical biases, reduce the min-entropy of the digitized output, or cause the entropy source to stall entirely. To guarantee that the harvested bits comply with stringent statistical standards such as NIST SP 800-90B, practical architectures must implement elaborate digital post-processing blocks, including Von Neumann correctors, linear feedback shift registers, or cryptographic hash functions, to compress and sanitize the raw bitstream [TBK+18]. Furthermore, TRNG cores require continuous online health testing circuitry to detect active adversarial manipulation or environmental drift in real time. Consequently, generating a single truly random bit demands substantial hardware area, imposes an elevated power consumption envelope, and exhibits an inherently low throughput that scales poorly with high-speed processing requirements.

Owing to these throughput limitations and microarchitectural overheads, contemporary cryptographic engineering avoids utilizing raw TRNG streams to fulfill the massive data requirements of entire matrices or vectors. Instead, modern hardware deployments adopt a hybrid security paradigm governed by seed expansion. In this practical framework, the expensive true randomness is exploited to generate a compact, high-quality bitstring that serves as a secure, unpredictable cryptographic seed. Once this seed is successfully registered within an internal memory, it is handed over to a high-throughput, deterministic algorithm designed to expand the short seed into the vast, uniform coordinate matrices required during the key generation and encryption routines.

This architectural compromise effectively minimizes the operational duty cycle of the physical entropy source, reducing the overall hardware power and area overhead while guaranteeing that the structural uniformity of the expanded data remains computationally bounded by the security margins of the underlying cryptographic expansion primitive.

3.1.2 Introduction to Pseudorandom Number Generators (PRNG)

To bridge the practical throughput gap of hardware entropy sources, modern cryptographic systems rely heavily on algorithmic expansion structures. A Pseudo-Random Number Generator (PRNG), also referred to as a Deterministic Random Bit Generator (DRBG), is a deterministic algorithm that ingests a short, high-entropy seed (typically captured by a TRNG) and processes it to produce a significantly longer sequence of output bits [BK15]. Unlike a TRNG, a PRNG operates entirely deterministically; given the identical initial seed, the algorithm will invariably compute the exact same output sequence. Consequently, the cryptographic security of a PRNG is evaluated not by absolute physical randomness, but by computational indistinguishability. The output bitstream must be statistically indistinguishable from a uniform distribution to any adversary lacking knowledge of the underlying seed, thereby preventing forward and backward state reconstruction attacks [BK15]. Within contemporary hardware acceleration architectures,

two primary structural methodologies exist for implementing high-throughput seed expansion: Linear Feedback Shift Register (LFSR) based lightweight stream ciphers and permutation-based symmetric hash functions.

LFSR-Based Lightweight Stream Ciphers (Bivium and Trivium)

For applications constrained by tight silicon area budgets and rigid power consumption envelopes, shift-register-based stream ciphers offer a highly optimized solution for pseudorandom bit generation. A prominent example is Trivium [DC06], designed by De Cannière and Preneel as a selection for the European eSTREAM portfolio and standardized as part of the lightweight stream cipher standard ISO/IEC 29192-3. Structurally, Trivium is based on a combination of three Non-Linear Feedback Shift Registers (NLFSRs) of degree 93, 84, and 111, maintaining a 288-bit internal state in total. The update mechanism combines linear feedback operations (XOR gates) with localized, low-degree non-linear terms (two-input AND gates) connecting the registers in a circular loop. This specific layout provides robust resistance against classical algebraic and linear cryptanalysis while allowing the microarchitectural implementation to easily parallelize bit generation by a factor of up to 64 without introducing structural hazards. The cipher operates using two input parameters: an 80-bit secret key and an 80-bit public Initialization Vector (IV), which are loaded into the state registers at the beginning of the execution. Following this loading phase, the cipher is clocked for 1152 cycles without producing any keystream. This warm-up phase corresponds to exactly four complete rotations of the state ($4 \times 288 = 1152$) and randomizes the content of the registers. After this initialization phase is completed, the online phase begins, enabling the continuous generation of a valid pseudorandom keystream.

To analyze the exact security margins and algebraic vulnerabilities of the full Trivium construction, Raddum introduced two hardware-reduced variants, designated Bivium A and Bivium B [Rad, Rad06]. This work does not focus on Bivium A because it has been demonstrated to be highly insecure, yielding to full cryptanalysis in approximately one day [Rad]. On the other hand, Bivium B, referred to simply as Bivium, consists of only two of the Trivium NLFSRs, namely the 93-bit and the 84-bit sub-registers. Bivium operates on identical algorithmic principles but compresses the internal state to a total of 177 bits. In this configuration, the keystream is generated as the bitwise sum of four specific state bits, taking two elements from each NLFSR block. While Bivium exhibits reduced resistance to specialized algebraic attacks compared to Trivium, it remains a highly valuable baseline architecture in hardware benchmarking and approximate computing frameworks where minimizing total gate count is prioritized over long-term cryptographic security margins. An interesting feature of hardware implementations of both Trivium and Bivium is that their structural layout can be unrolled. Hardware unrolling implies that multiple algorithmic instances are realized in silicon without any intermediary memory elements or registers between them, allowing the combinational logic of multiple execution steps to be evaluated within a single clock cycle [CMM$^+$23]. The structural layout of such an optimization is exemplified by the 2-bit unrolled version of Bivium presented in Figure 3.1. As

illustrated, the dotted line indicates the unrolled path, which allows the architecture to evaluate multiple steps concurrently and generate the pseudorandom keystream with an operational throughput rate that is exactly twice that of a standard, regular Bivium implementation.

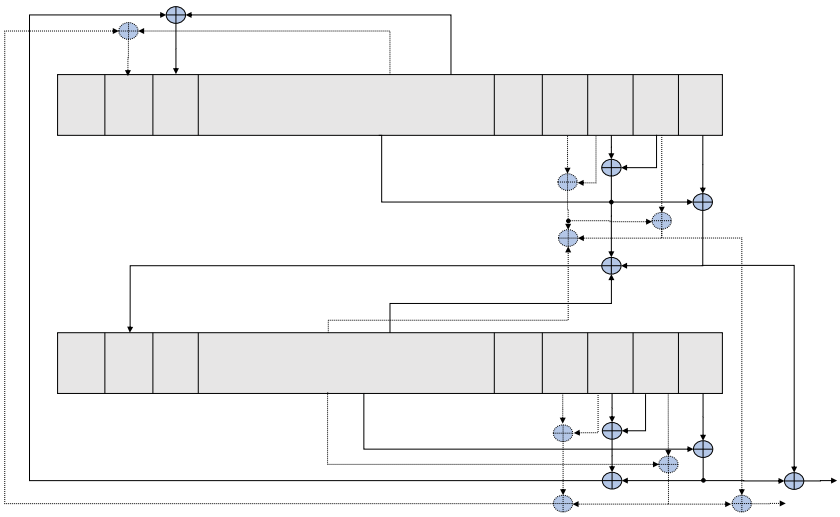


Figure 3.1 2-bit unrolled Bivium architectural schematic.

Symmetric Hash Functions and the FIPS 202 Standard

When public-key primitives demand compliance with standardized security metrics, symmetric hash functions are deployed as high-security PRNG blocks. The primary governing framework for modern permutation-based designs is the FIPS 202 standard specified by the NIST [Nat15b], which formalizes the SHA-3 cryptographic hash family based on the *Keccak* sponge construction [BDVA11]. The standard defines two distinct families of functions tailored for separate cryptographic objectives: standard SHA-3 functions and SHAKE functions.

A traditional hash function is a deterministic algorithm that transforms input data of arbitrary size into a fixed-size output, termed a hash or digest, ensuring that even slight variations in the input data generate significantly different, uncorrelated outputs. On the other hand, eXtendable-Output Functions (XOFs) act as a functional extension of hash functions where the produced output can be of variable and arbitrary length. This architecture allows for the generation of arbitrarily long sequences by continually extracting additional output bits from the internal structure. Consequently, while SHA-3 functions are primarily utilized for data hashing and integrity verification, SHAKE functions are explicitly optimized to operate as XOFs for high-capacity pseudorandom stream generation when instantiated with a random cryptographic seed. As detailed in Table 3.1, the FIPS 202 standard regulates the structural characteristics of four distinct hash functions and two distinct XOFs, all constructed using the underlying Keccak sponge function.

Keccak is a highly versatile cryptographic primitive capable of supporting hashing, authenticated encryption, and pseudorandom number generation [BDVA11]. Its core architecture utilizes

Table 3.1 Parameters of FIPS 202 Functions (bits)

Function	r	c	Output Size
SHA3-224	1152	448	224
SHA3-256	1088	512	256
SHA3-384	832	768	384
SHA3-512	576	1024	512
SHAKE128	1344	256	Unlimited
SHAKE256	1088	512	Unlimited

the well-known sponge construction illustrated in Figure 3.2. From a general standpoint, a sponge construction operates as a functional mapping that ingests an input of arbitrary size and yields an output sequence of any length required by the system application. The sponge layout is composed primarily of three components: a state register, a fixed-length permutation function (f), and a padding function. The state register maintains a total width of $b = 1600$ bits and is partitioned into two distinct internal fields: the rate section r, where the active message elements are processed, and the capacity section c, which contains additional bits required to enforce the design's security boundaries. The padding function appends a specific bit pattern to the input message to ensure that the total length of the input data is a whole multiple of the rate parameter r.

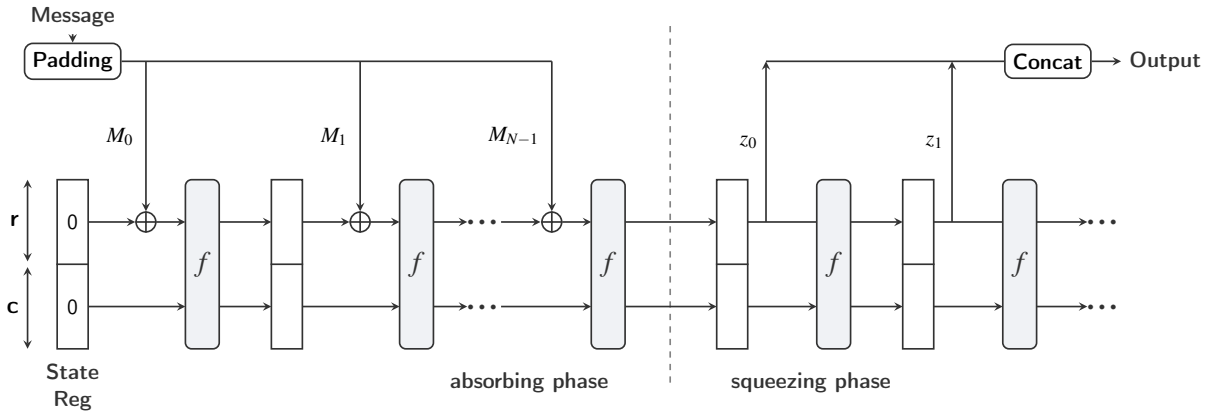


Figure 3.2 Structural operational mapping of the Keccak sponge framework highlighting the absorbing and squeezing phases.

The underlying permutation function of Keccak, formally designated as Keccak-p[1600,24], executes 24 iterations of an identical, specialized round function. Each individual round comprises five specialized permutation steps: four linear operations (θ, ρ, π, ι) based on simple bit-wise operations including eXclusive-OR (XOR) logic, bitwise rotations, and the injection of round constants, followed by a single non-linear step (χ) that introduces logical AND operators to break algebraic linear tracking.

To implement the SHA-3 family of hash functions, the input text must be appropriately padded, divided into discrete blocks matching the size of r, and systematically absorbed into

the rate portion of the state register. The resulting digest of the message is extracted exclusively from the rate section of the state register after the permutation function has been applied to the last absorbed message block. The precise length of this extracted digest depends on the required security level, which determines the operational assignment of the rate and capacity parameters according to Table 3.1.

For the SHAKE family of XOFs, the fixed-length input seed is absorbed in an analogous manner. However, the squeezing phase differs significantly. Because the required output can be of arbitrary length, pseudorandom bits are extracted iteratively across the entire rate section of the state register, executing additional permutation rounds as needed to continuously generate additional data. Because these XOFs can squeeze uniform bit sequences indefinitely without resetting the internal state register or compromising security bounds, they serve as the default structural mechanism for public key matrix expansion in post-quantum lattice algorithms, including ML-KEM and ML-DSA. Although implementing a full Keccak permutation block requires an order of magnitude more silicon area and registers than lightweight shift-register solutions, hash-based expansion guarantees full theoretical resilience against state-compromise and distinguisher attacks. By providing an expansive seed space that resists quantum-accelerated brute-force search, this standardized approach preserves the long-term post-quantum security bounds of the overall cryptographic primitive.

Crucially, within the context of LWE-based architectures, these high-throughput pseudorandom bitstreams are employed for the expansion of uniform public matrices and they also serve as the foundational randomness engine that feeds subsequent non-uniform sampling blocks. Whether an architecture utilizes lightweight shift registers or standardized extendable-output functions, most commonly, the generated pseudorandom bitstream must ultimately be mapped through specialized mathematical layers, such as rejection samplers, inversion tables, or bit-counting transformation logic, to shape raw uniform bits into the bounded error distributions required to protect the underlying lattice secret.

3.2 Error Sampling in LWE

In the context of the LWE problem, the generation of error terms is a critical operational phase that directly impacts both the theoretical security bounds and the implementation efficiency of the cryptosystem. Because the cryptographic validity of the primitive depends on the error distribution strictly conforming to its defined parametric boundaries, the underlying sampling mechanisms require a source of randomness that is both statistically secure and computationally indistinguishable from uniform data.

As established in Section 3.1, the construction of the public coefficient matrix $\mathbf{A}$ relies on uniform distributions. In standardized implementations, this high-throughput uniform pseudorandomness generation is handled directly by specific XOFs derived from the FIPS 202 guidelines,

as detailed in Section 3.1.2. However, generating the localized error vector **e** requires an additional layer to transform these uniform bitstreams into a targeted non-uniform distribution.

In practical deployments, this comprehensive parameter and error generation follows a multi-stage process, as illustrated in Figure 3.3. The operational workflow follows a sequential pipeline that transitions from randomness extraction to algorithmic expansion and distribution shaping. This process begins with a TRNG producing a compact, high-entropy cryptographic seed. Once generated, this seed is expanded by means of a PRNG. The expansion algorithmically processes the compact input to generate a massive, uniform pseudorandom bitstream. At this stage of the matrix product chain, these uniform bits can be routed directly to instantiate the coordinates of the public coefficient matrix **A**, completely satisfying the requirement for uniform distribution. In contrast, constructing the error vector **e** requires an additional phase dedicated to distribution shaping, as raw uniform bits cannot satisfy the bounded error criteria mandated by the cryptosystem. To achieve this, a designated portion of the uniform bitstream from the PRNG is directed into a sampler, which may be realized as a discrete Gaussian, centered binomial, or bounded uniform sampler depending on the specific protocol standard. This block mathematically transforms and maps the uniform inputs, shaping them into the localized, small-magnitude error distribution required to securely mask the linear relationships of the lattice challenge without exceeding the functional decoding boundaries of the system.

In Figure 3.3, the dependency path for the secret vector **s** is designated with a dashed line to reflect its structural flexibility across different implementation contexts. Unlike the public matrix **A**, which is routinely expanded from a shared public seed to minimize communication overhead, the private key variables comprising **s** can be instantiated through alternative architectural workflows. Depending on the specific protocol configuration or operational mode, **s** may be derived directly from TRNGs, sampled from an alternative non-uniform distribution matching the error profile, or treated as a pre-computed input variable during ephemeral encryption routines.

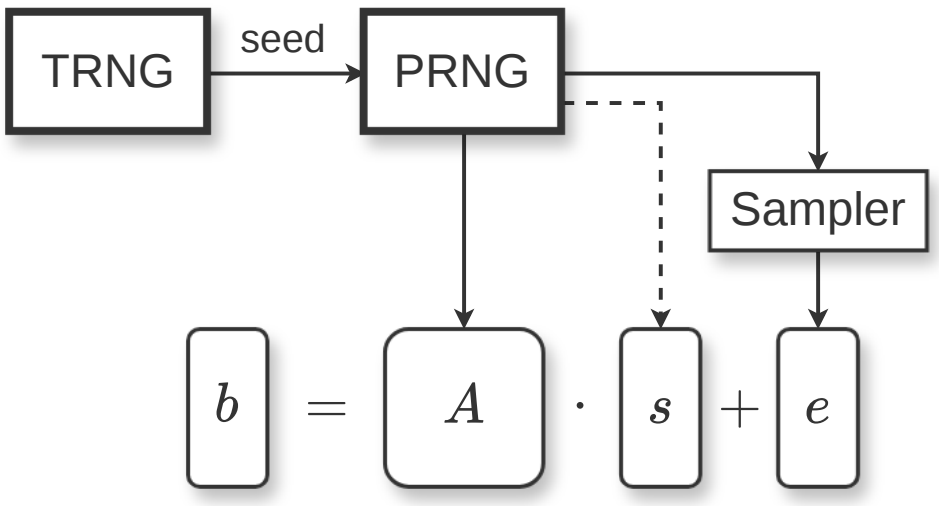


Figure 3.3 Inner matrix product generation chain.

3.2.1 Methodologies for Discrete Gaussian Sampling

As established in the preceding sections, cryptographic primitives relying on the LWE problem require the injection of an error term to guarantee theoretical security margins. While various error distributions can be mathematically instantiated, contemporary lattice-based constructions frequently leverage the discrete Gaussian distribution. Generating samples from this distribution requires specialized algorithmic components known as *discrete Gaussian samplers*. Structurally, these samplers ingest high-entropy, uniformly distributed pseudo-random bitstreams and transform them into discrete outputs that conform precisely to a target Gaussian distribution defined by a specific standard deviation σ and mean μ.

Numerous algorithms [KRR+18, KLL20] have been proposed to execute sampling from the discrete Gaussian distribution, each introducing distinct design trade-offs regarding computational throughput, hardware area, memory consumption, precision bounds, and timing-attack resistance. The most prominent and widely studied methodologies in the literature include rejection sampling, Bernoulli-based trials, the Cumulative Distribution Table (Cumulative Distribution Table (CDT)) approach, Ziggurat-based techniques, and the Knuth–Yao (KY) random walk algorithm [HKR+16].

In the following, a high-level overview of these foundational sampling approaches is provided. Particular attention is devoted to the structural properties of both the CDT approach and the Knuth–Yao algorithm, as these two specific methodologies constitute the underlying sampling primitives implemented and evaluated in the remaining sections of this work.

Rejection Sampling

Originally introduced by von Neumann [VN+63], rejection sampling is a flexible methodology used to generate samples from a target distribution $f(x)$ by utilizing an easier-to-sample proposal distribution $g(x)$. Candidate samples x are drawn from the proposal distribution and are subsequently accepted with a probability defined by:

$$P(\text{accept} \mid x) = \frac{f(x)}{c \cdot g(x)} \tag{3.1}$$

where c is a bounding constant tightly chosen such that $c \geq \sup_x \frac{f(x)}{g(x)}$ across the entire operational support. For discrete Gaussian sampling within lattice-based primitives, simple proposals such as uniform or geometric distributions are frequently employed due to their straightforward realization.

While this method is conceptually simple and avoids the memory overhead of large precomputed distribution tables, its non-deterministic execution profile introduces significant challenges in secure implementations. Because the number of iterations required to achieve a valid acceptance is a random variable following a geometric distribution with an expected value of c,

the runtime of a naive implementation is inherently variable. This timing variability introduces severe vulnerabilities to side-channel analysis, specifically timing and power attacks, as the execution duration can leak information regarding the rejected candidates and the internal state of the sampler. Implementing constant-time countermeasures, such as executing a fixed, worst-case number of dummy iterations, remedies this vulnerability but substantially reduces the operational throughput. Consequently, the practical efficiency of this approach remains heavily dependent on how closely the chosen proposal distribution mirrors the target discrete Gaussian curve to minimize the rejection rate.

Bernoulli-Based Samplers

Bernoulli-based samplers construct discrete Gaussian distributions by evaluating a sequence of randomized binary outcomes known as Bernoulli trials [DDLL13]. Typically, this methodology operates as a multi-stage refinement pipeline where a candidate value is first drawn from an easily generated base distribution, such as a geometric or half-Gaussian distribution, and subsequently accepted or rejected through successive Bernoulli validation steps. By definition, a Bernoulli trial with a probability parameter $p \in [0,1]$ yields a binary outcome $B \in \{0,1\}$ characterized by the probability distribution:

$$\Pr[B = 1] = p. \tag{3.2}$$

Within the specific context of discrete Gaussian sampling, the mathematical targets require the evaluation of transcendental exponential probabilities. Consequently, the operational parameter p is typically formulated as:

$$\Pr[B = 1] = e^{-t} \tag{3.3}$$

where t represents a positive parameter whose value is directly proportional to the squared distance of the candidate sample from the mean, scaled by the target variance.

A significant advantage of Bernoulli-based samplers is that these exponential evaluation trials can be executed efficiently using bit-level operations. For instance, the probability e^{-t} can be decomposed into a product of precomputed constants corresponding to the bitwise representation of the input parameter. This algorithmic property makes Bernoulli constructions highly attractive for practical implementations because they eliminate the need for large lookup tables, requiring minimal memory footprints and elementary logic structures. Furthermore, because the evaluation of these binary probabilities can be designed to process a fixed sequence of operations regardless of the input data, the methodology is highly amenable to constant-time execution profiles, thereby providing robust resistance against timing and power-based side-channel analysis. However, their global computational efficiency remains strictly dependent on both the expected number of sequential Bernoulli trials required to output a single valid sample and the precision limits required to accurately evaluate and represent the underlying e^{-t} probabilities.

Cumulative Distribution Table (CDT) Sampling

The Cumulative Distribution Table (CDT) methodology provides an alternative, table-driven approach to discrete Gaussian sampling by shifting the computational complexity from the online execution phase into an offline precomputation phase [Pei10]. Fundamentally, this technique precomputes the cumulative distribution function (CDF) of the discrete Gaussian distribution and stores the resulting boundary probabilities within a structured array or table. During the online sampling phase, the algorithm generates a uniform random number $x \in [0,1)$ and maps it to a discrete target sample by identifying the specific probability interval in the table into which x falls. This technique guarantees that the output elements strictly match the desired target distribution up to a mathematically bounded precision error.

As detailed by Peikert [Pei10], when deploying discrete Gaussian operations within lattice-based cryptosystems, the rounding parameter r is typically fixed, relatively small, and known well in advance. This structural constraint enables the implementation of highly optimized and compact tables. Because the probability mass of a Gaussian distribution decays exponentially, the precomputed table does not need to cover an infinite support. Instead, the table is safely truncated outside of a specific interval bounded by the tail factor $r \cdot \omega(\sqrt{\log n})$ [Pei10]. The probability mass outside this window is mathematically negligible, meaning its exclusion introduces only a negligible statistical distance from the ideal distribution.

A notable optimization enabled by the CDT approach in this context is the concept of "lazy bit consumption". When searching for the correct discrete value within the cumulative table, the uniform random variable x does not need to be evaluated to full precision immediately. Instead, its random bits can be generated and processed sequentially, from the most-significant to the least-significant bit, until the target interval is uniquely identified. Despite its high efficiency and simple control flow, the primary drawback of the CDT approach resides in its substantial memory footprint. For cryptographic parameters requiring broad standard deviations or exceptionally high precision, the memory allocation required to store the tables scales rapidly with the tail bounds and precision requirements. Furthermore, a naive implementation utilizing a standard binary search introduces severe side-channel vulnerabilities. Because a standard search creates data-dependent memory access patterns and variable execution timing based on the selected interval, it can leak critical details regarding the sampled values. Consequently, to ensure robust side-channel resistance, the lookup phase must be implemented using constant-time techniques, typically requiring a complete memory scan of the table elements to disguise the true access index.

Ziggurat Method

The Ziggurat method, originally designed for continuous probability distributions, is an algorithmic approach that partitions a probability density function into a series of horizontal rectangular sections of equal area, stacked to resemble a stepped pyramid. The primary advantage of this

geometric arrangement is that the vast majority of candidate samples fall into the inner regions of these rectangles that lie entirely beneath the target distribution curve. For these dominant regions, the algorithm can output a valid sample immediately with a very low, constant computational cost. Only a small fraction of candidate points fall into the boundary wedges located between the rectangular steps and the true curve, or into the infinite tail region, where a slower rejection sampling mechanism must be invoked to evaluate the exact distribution boundaries.

While discrete adaptations of the Ziggurat method have been explored for cryptographic lattice frameworks [BCG$^+$13], they remain significantly less common in practical implementations. Although the algorithm offers exceptional average-case throughput on high-performance processors, the necessity of the occasional rejection step introduces an inherently non-deterministic execution profile. This runtime variability is directly proportional to the total area of the boundary segments and the distribution's tail, which heavily complicates the realization of strictly constant-time execution profiles required to mitigate timing-based side-channel attacks. Furthermore, the Ziggurat method inherently requires storing precomputed lookup tables containing the exact geometric step boundaries. In its standard formulation, it also relies on floating-point arithmetic to accurately evaluate candidate coordinates against the non-linear curve. These combined storage and arithmetic overheads make it less suitable for highly constrained Internet of Things (IoT) devices where fixed-point or integer-only operations are strictly prioritized.

Knuth–Yao Algorithm

Knuth and Yao [KY76] introduced a foundational method to represent any discrete probability distribution using a binary tree structure, commonly referred to as a Discrete Distribution Generating (DDG) tree. In this model, each path from the root node to a terminal leaf node uniquely maps to a valid sample value, where the path topology itself encodes the precise probability of selecting that value. Every level of the tree corresponds to a binary decision guided by a single uniformly random input bit, meaning the cumulative path length represents a specific summation of negative powers of two. Sampling from the target distribution is executed via a random walk down the tree structure, driven by the continuous injection of these random bits until a terminal leaf node is reached and the corresponding value is output.

To construct the DDG tree, a formal probability matrix (P-matrix) is defined, where each row corresponds to a potential discrete sample value and each column represents the binary fraction expansions of the associated probabilities at successive precision levels. The total number of columns establishes the statistical precision of the sampler and determines the exact number of uniform input bits required during the traversal. The binary tree architecture is subsequently extracted from this matrix, forming the structural baseline for the operational execution.

Building on this theoretical foundation, Dwarakanath and Galbraith [DG14] adapted the Knuth–Yao algorithm for discrete distributions where probabilities are expressed as fixed-precision binary fractions, focusing explicitly on the discrete Gaussian distribution required

by cryptographic primitives. To optimize the storage and traversal overhead, their architecture limits the tree's support by sampling exclusively from the non-negative half of the Gaussian distribution, halving the probability mass allocated to the zero coordinate, and subsequently utilizing an additional independent random bit to dynamically determine the final algebraic sign.

Further extending this paradigm, Karmakar et al. [KRR+18] observed that each discrete sample generated by a Knuth–Yao tree is uniquely and deterministically dictated by the sequence of uniform input bits consumed during the random walk. Consequently, each individual bit of the final output sample can be mathematically formulated as a synthesized Boolean function of these random input bits. Leveraging this property, they introduced a software-based bit-slicing technique capable of evaluating these functions across multiple data words simultaneously to achieve highly parallelized sample generation. Furthermore, Karmakar et al. [KRVV19] proposed a constant-time software evaluation framework for these Boolean representations, an optimization necessary to eliminate data-dependent timing channels and mitigate timing-based side-channel vulnerabilities.

Given their inherent spatial parallelism, custom hardware platforms such as FPGA and ASIC represent an ideal medium for the high-throughput evaluation of these Boolean function chains. Lyons and Gaj [LG20] exploited this microarchitectural advantage by developing an automated methodology to derive minimized combinational Boolean expressions directly from a target distribution's DDG tree. This framework synthesizes highly optimized, synthesizable VHDL descriptions of the combinational logic, yielding high-throughput, area-efficient discrete Gaussian samplers that are exceptionally well-suited for deployment in resource-constrained post-quantum cryptographic hardware accelerators.

3.2.2 LWE-Based Key Encapsulation Mechanisms

Having established the basic principles of pseudorandom number generation and Gaussian error sampling, the next step consists of analyzing how these separate operational blocks are integrated into complete, deployable cryptographic primitives. A *key-encapsulation mechanism* (KEM) is a framework composed of three foundational cryptographic algorithms that allows two parties to securely establish a shared secret key over an insecure public channel. Once successfully established, this shared secret key can be utilized alongside symmetric-key cryptographic algorithms to perform basic tasks in secure communications, such as symmetric data encryption and message authentication.

Formally, a generic KEM scheme is defined by a collection of parameter sets alongside three core algorithms:

- **Key Generation** (KeyGen): An algorithmic routine that generates a fresh key pair consisting of a public encapsulation key (`ek`) and a private decapsulation key (`dk`).

- **Encapsulation** (Encaps): An operation executed by the initiating party that ingests the recipient's public encapsulation key (ek) to compute a random shared secret key K and an associated ciphertext c.
- **Decapsulation** (Decaps): An operation executed by the receiving party that processes the incoming ciphertext c using the private decapsulation key (dk) to recover the identical shared secret key K.

The accompanying collection of parameter sets contains specific numerical constants chosen to define precise operational trade-offs, enabling implementations to scale smoothly between high-throughput performance and conservative post-quantum security margins.

A graphical representation of this key establishment mechanism is illustrated in Figure 3.4, where the roles of Alice and Bob correspond directly to the interactions between Sandy and Reece described in preceding sections of this work.

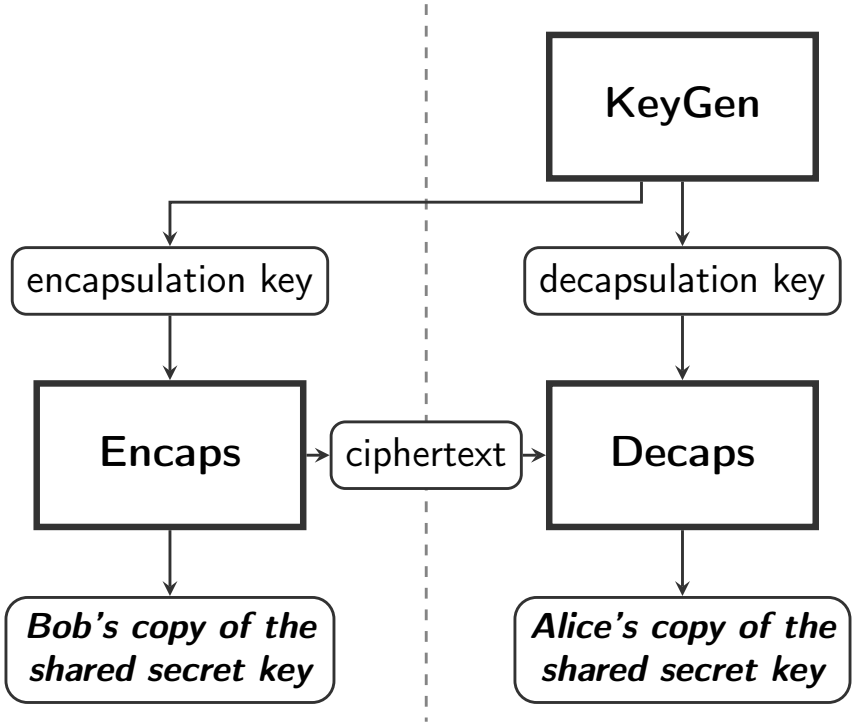


Figure 3.4 Operational data flow of a Key Encapsulation Mechanism (KEM) framework during key establishment.

To trace this operational sequence, Sandy initiates the process by running the KeyGen routine locally to produce her public encapsulation key and private decapsulation key. Upon acquiring Sandy's encapsulation key over the public channel, Reece executes the Encaps algorithm. This routine generates Reece's independent copy of the shared secret key, denoted as K_B, along with an associated ciphertext c that binds the encapsulated secret. Reece transmits this ciphertext back across the network to Sandy. Sandy concludes the protocol by running the Decaps routine, feeding the received ciphertext c and her private decapsulation key into the execution engine to compute her local copy of the secret, denoted as K_A. Following a successful, uncorrupted

execution flow, both parties arrive at identical outputs such that $K_A = K_B$, establishing a secure, random, shared secret key ready for cryptographic deployment [oST24].

Within modern post-quantum cryptography, the concrete translation of the LWE problem into standardized, deployable KEMs splits into two primary structural paradigms, depending entirely on the underlying algebraic framework utilized to build the system matrices.

The first paradigm adheres strictly to the generic, standard LWE formulation using unstructured matrices over the modular integer ring $\mathbb{Z}_q$. This highly conservative design approach is applied by FrodoKEM, which deliberately avoids structured algebraic rings to provide maximal long-term security margins against structural lattice reduction attacks, willingly accepting the resulting trade-off of expanded key sizes and increased network communication overhead. Conversely, the second paradigm introduces ring-based module structures to drastically reduce key footprints and accelerate operational matrix arithmetic, giving rise to Module-LWE (MLWE). This algebraically optimized framework serves as the mathematical baseline for ML-KEM (originally derived from the round-three CRYSTALS-Kyber submission [oST]), which has been finalized by the NIST as the primary standardized baseline for general-purpose post-quantum encryption [NIS24b].

The following subsections detail the operational mechanics, algorithmic workflows, and specific parameter selections of these two distinct cryptographic families, explicitly mapping the previously discussed randomness engines and discrete sampling primitives to their concrete functional roles within the respective encapsulation and decapsulation protocols.

Standard LWE-based: FrodoKEM

FrodoKEM is a family of key-encapsulation mechanisms submitted to the NIST post-quantum cryptography standardization process [A+20]. Unlike ring- or module-based primitives, the security of FrodoKEM is anchored directly in the hardness of the standard, algebraically unstructured LWE problem over generic lattices. This conservative design rationale deliberately avoids the extra algebraic ring structure to maximize long-term immunity against potential structural lattice reduction attacks. Like its module-oriented counterparts, FrodoKEM satisfies the indistinguishability under adaptive chosen-ciphertext attacks (IND-CCA2) security criteria. Under this security model, a scheme is classified as secure if an adversary remains incapable of distinguishing the encapsulated shared secret from a uniformly random key, even when granted continuous access to a decapsulation oracle (with the strict exception of decapsulating the challenge ciphertext itself) [A+22].

The FrodoKEM specification defines three primary parameter configurations designed to balance post-quantum security margins with memory and communication overhead: FrodoKEM-640, FrodoKEM-976, and FrodoKEM-1344. These target NIST security levels 1, 3, and 5, respectively, matching or exceeding the brute-force security of the corresponding symmetric

AES block size variants. The explicit bit-length requirements for data structures across the three standardized security levels are compiled in Table 3.2.

	Encaps. Key	Decaps. Key	Ciphertext	Shared Secret Key
FrodoKEM-640	9,616	19,888	9,720	16
FrodoKEM-976	15,632	31,296	15,744	24
FrodoKEM-1344	21,520	43,088	21,632	32

Table 3.2 Sizes (in bytes) of keys and ciphertexts across FrodoKEM variants [A+20]

A defining microarchitectural feature of FrodoKEM is its extensive execution footprint in symmetric cryptographic operations. To minimize public key transmission sizes, the massive $n \times n$ public matrix $\mathbf{A}$ is deterministically expanded on-the-fly from a compact 16-byte seed via the `Frodo.Gen` subroutine, which can be instantiated using either AES128 or SHAKE128. Furthermore, error mapping is achieved through constant-time inversion sampling arrays, where pseudorandom bitstreams from a SHAKE instance are parsed to generate discrete localized noise matrices. Because these iterative permutations and large-scale matrix-vector transformations modulo a power of two form the absolute primary computational bottleneck of the protocol, they provide a strong structural justification for the dedicated hardware accelerator developed in this work.

To establish where these symmetric operations interact with the generic lattice layers, the operational execution flow is organized into three central algorithmic subroutines:

- Key Generation (`FrodoKEM.KeyGen`);
- Encapsulation (`FrodoKEM.Encaps`);
- Decapsulation (`FrodoKEM.Decaps`).

The foundational logic of `FrodoKEM.KeyGen` (Algorithm 5) encapsulates the underlying public-key encryption scheme routine `FrodoPKE.KeyGen` (Algorithm 4). The framework samples uniform high-entropy seeds to derive the public matrix seed and expands error blocks $\mathbf{S}$ and $\mathbf{E}$ from localized SHAKE streams. The public encapsulation key stores the raw expanded seed along with the packed modular matrix result $\mathbf{B}$. The complete decapsulation private key vector expands upon the public elements by appending the secret transpose $\mathbf{S}^T$, an independent rejection seed s, and the computed key hash string pkh.

The execution flow of `FrodoKEM.Encaps` (Algorithm 7) utilizes the public-key encryption layer `FrodoPKE.Enc` (Algorithm 6) to bind an ephemeral, randomly generated message block μ into a dual-component ciphertext string $(c_1||c_2)$. Hashing functions derive specific pseudorandom strings from the public key configuration, isolating the active encryption noise layers from long-term parameters. The final shared secret ss is established by passing the combined string layout and intermediate pseudorandom values through a concluding SHAKE permutation step.

Finally, `FrodoKEM.Decaps` (Algorithm 9) implements a robust variant of the Fujisaki–Okamoto (FO) transform featuring implicit rejection to maintain active side-channel immunity. The incoming ciphertext layers are parsed and decoded via the underlying decryption operation `FrodoPKE.Dec` (Algorithm 8) to extract a candidate plaintext μ'. To guarantee that this plaintext has not been actively altered during transmission, the decapsulation engine completely replicates the encapsulation pipeline to generate a local verification ciphertext pair $\mathbf{B}''$ and $\mathbf{C}'$. A data-independent, constant-time comparison check is performed on the verification variables. If a discrepancy is caught, the protocol applies implicit rejection, overriding the generated keying variables with a fallback pseudorandom string derived from the long-term secret seed component s and the ciphertext string to prevent information leakage.

Algorithm 4 FrodoPKE.KeyGen()

Generates a public encryption key and a corresponding secret decryption key.
Output: Public key $pk = (seed_A, \mathbf{B}) \in \{0,1\}^{len_{seed_A}} \times \mathbb{Z}_q^{n \times \bar{n}}$.
Output: Secret key $sk = \mathbf{S}^T \in \mathbb{Z}_q^{\bar{n} \times n}$.

$seed_A \xleftarrow{\$} \{0,1\}^{len_{seed_A}}$ ▷ choose a uniformly random matrix seed
$\mathbf{A} \leftarrow \text{Frodo.Gen}(seed_A)$ ▷ generate pseudorandom matrix $\mathbf{A}$
$seed_{SE} \xleftarrow{\$} \{0,1\}^{len_{seed_{SE}}}$ ▷ choose a random seed for error sampling
$(r^{(0)}, r^{(1)}, \ldots, r^{(2n\bar{n}-1)}) \leftarrow \text{SHAKE}(0\text{x}5\text{F} || seed_{SE}, 2n\bar{n} \cdot len_\chi)$
$\mathbf{S}^T \leftarrow \text{Frodo.SampleMatrix}((r^{(0)}, \ldots, r^{(n\bar{n}-1)}), \bar{n}, n, T_\chi)$
$\mathbf{E} \leftarrow \text{Frodo.SampleMatrix}((r^{(n\bar{n})}, \ldots, r^{(2n\bar{n}-1)}), n, \bar{n}, T_\chi)$
$\mathbf{B} \leftarrow \mathbf{AS} + \mathbf{E} \pmod q$
return $(pk = (seed_A, \mathbf{B}), sk = \mathbf{S}^T)$

Algorithm 5 FrodoKEM.KeyGen()

Generates a public encapsulation key and a corresponding secret decapsulation key.
Output: Public key $pk \in \{0,1\}^{len_{seed_A} + D \cdot n \cdot \bar{n}}$.
Output: Secret key $sk' \in \{0,1\}^{len_s + len_{seed_A} + D \cdot n \cdot \bar{n}} \times \mathbb{Z}_q^{\bar{n} \times n} \times \{0,1\}^{len_{pkh}}$.

$s || seed_{SE} || z \xleftarrow{\$} \{0,1\}^{len_s + len_{seed_{SE}} + len_z}$ ▷ choose random seeds uniformly
$seed_A \leftarrow \text{SHAKE}(z, len_{seed_A})$ ▷ generate pseudorandom seed for matrix $\mathbf{A}$
$\mathbf{A} \leftarrow \text{Frodo.Gen}(seed_A)$ ▷ generate pseudorandom matrix $\mathbf{A}$
$(r^{(0)}, r^{(1)}, \ldots, r^{(2n\bar{n}-1)}) \leftarrow \text{SHAKE}(0\text{x}5\text{F} || seed_{SE}, 2n\bar{n} \cdot len_\chi)$
$\mathbf{S}^T \leftarrow \text{Frodo.SampleMatrix}((r^{(0)}, \ldots, r^{(n\bar{n}-1)}), \bar{n}, n, T_\chi)$
$\mathbf{E} \leftarrow \text{Frodo.SampleMatrix}((r^{(n\bar{n})}, \ldots, r^{(2n\bar{n}-1)}), n, \bar{n}, T_\chi)$
$\mathbf{B} \leftarrow \mathbf{AS} + \mathbf{E} \pmod q$
$b \leftarrow \text{Frodo.Pack}(\mathbf{B})$ ▷ pack matrix into a compressed bit string
$pkh \leftarrow \text{SHAKE}(seed_A || b, len_{pkh})$ ▷ compute public key hash string
return $(pk = seed_A || b, sk' = (s || seed_A || b, \mathbf{S}^T, pkh))$

Algorithm 6 FrodoPKE.Enc(μ, pk)

Encrypts a plaintext message using the public key parameters.
Input: Message $\mu \in \{0,1\}^{len_\mu}$, public key $pk = (seed_A, \mathbf{B})$.
Output: Ciphertext $c = (\mathbf{C}_1, \mathbf{C}_2) \in \mathbb{Z}_q^{\bar{m}\times n} \times \mathbb{Z}_q^{\bar{m}\times \bar{n}}$.

$\mathbf{A} \leftarrow$ Frodo.Gen$(seed_A)$ ▷ regenerate matrix $\mathbf{A}$ from seed
$seed_{SE} \xleftarrow{\$} \{0,1\}^{len_{seed_{SE}}}$ ▷ choose random error seed
$(r^{(0)}, \ldots, r^{(2\bar{m}n+\bar{m}\bar{n}-1)}) \leftarrow$ SHAKE$(0\text{x}96||seed_{SE}, (2\bar{m}n+\bar{m}\bar{n}) \cdot len_\chi)$
$\mathbf{S}' \leftarrow$ Frodo.SampleMatrix$((r^{(0)}, \ldots, r^{(\bar{m}n-1)}), \bar{m}, n, T_\chi)$
$\mathbf{E}' \leftarrow$ Frodo.SampleMatrix$((r^{(\bar{m}n)}, \ldots, r^{(2\bar{m}n-1)}), \bar{m}, n, T_\chi)$
$\mathbf{E}'' \leftarrow$ Frodo.SampleMatrix$((r^{(2\bar{m}n)}, \ldots, r^{(2\bar{m}n+\bar{m}\bar{n}-1)}), \bar{m}, \bar{n}, T_\chi)$
$\mathbf{C}_1 \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}' \pmod q$
$\mathbf{C}_2 \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}'' \pmod q$
return $c = (\mathbf{C}_1, \mathbf{C}_2 +$ Frodo.Encode$(\mu))$

Algorithm 7 FrodoKEM.Encaps(pk)

Generates a shared secret key and an associated ciphertext string pair.
Input: Public key $pk = seed_A||b \in \{0,1\}^{len_{seed_A}+D\cdot n\cdot \bar{n}}$.
Output: Ciphertext string $c_1||c_2 \in \{0,1\}^{(\bar{m}n+\bar{m}\bar{n})D}$, shared secret $ss \in \{0,1\}^{len_{ss}}$.

$\mu \xleftarrow{\$} \{0,1\}^{len_\mu}$ ▷ choose a random ephemeral key string uniformly
$pkh \leftarrow$ SHAKE(pk, len_{pkh}) ▷ compute explicit public key hash
$seed_{SE}||k \leftarrow$ SHAKE$(pkh||\mu, len_{seed_{SE}} + len_k)$ ▷ derive internal execution tokens
$(r^{(0)}, \ldots, r^{(2\bar{m}n+\bar{m}\bar{n}-1)}) \leftarrow$ SHAKE$(0\text{x}96||seed_{SE}, (2\bar{m}n+\bar{m}\bar{n}) \cdot len_\chi)$
$\mathbf{S}' \leftarrow$ Frodo.SampleMatrix$((r^{(0)}, \ldots, r^{(\bar{m}n-1)}), \bar{m}, n, T_\chi)$
$\mathbf{E}' \leftarrow$ Frodo.SampleMatrix$((r^{(\bar{m}n)}, \ldots, r^{(2\bar{m}n-1)}), \bar{m}, n, T_\chi)$
$\mathbf{A} \leftarrow$ Frodo.Gen$(seed_A)$ ▷ regenerate matrix $\mathbf{A}$
$\mathbf{B}' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}' \pmod q$
$c_1 \leftarrow$ Frodo.Pack$(\mathbf{B}')$ ▷ compress and pack matrix $\mathbf{B}'$
$\mathbf{E}'' \leftarrow$ Frodo.SampleMatrix$((r^{(2\bar{m}n)}, \ldots, r^{(2\bar{m}n+\bar{m}\bar{n}-1)}), \bar{m}, \bar{n}, T_\chi)$
$\mathbf{B} \leftarrow$ Frodo.Unpack$(b, n, \bar{n})$ ▷ decompress packed public matrix
$\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}'' \pmod q$
$\mathbf{C} \leftarrow \mathbf{V} +$ Frodo.Encode$(\mu) \pmod q$
$c_2 \leftarrow$ Frodo.Pack$(\mathbf{C})$ ▷ compress and pack matrix $\mathbf{C}$
$ss \leftarrow$ SHAKE$(c_1||c_2||k, len_{ss})$ ▷ compute final shared secret string
return $(c_1||c_2, ss)$

Algorithm 8 FrodoPKE.Dec(c, sk)

Decrypts a packed ciphertext using the secret matrix exponent.
Input: Ciphertext $c = (\mathbf{C}_1, \mathbf{C}_2)$, secret key $sk = \mathbf{S}^T$.
Output: Decrypted message $\mu' \in \{0,1\}^{len_\mu}$.

$\mathbf{M} \leftarrow \mathbf{C}_2 - \mathbf{C}_1\mathbf{S} \pmod q$
$\mu' \leftarrow$ Frodo.Decode$(\mathbf{M})$ ▷ decode raw bitstream components
return μ'

Algorithm 9 FrodoKEM.Decaps($c_1||c_2, sk'$)

Extracts the shared secret key from a ciphertext string using the secret key profile.
Input: Ciphertext string $c_1||c_2 \in \{0,1\}^{(\bar{m}n+\bar{m}\bar{n})D}$, secret key $sk' = (s||seed_A||b, \mathbf{S}^T, pkh)$.
Output: Shared secret $ss \in \{0,1\}^{len_{ss}}$.

$\mathbf{B}' \leftarrow$ Frodo.Unpack$(c_1, \bar{m}, n)$
$\mathbf{C} \leftarrow$ Frodo.Unpack$(c_2, \bar{m}, \bar{n})$
$\mathbf{M} \leftarrow \mathbf{C} - \mathbf{B}'\mathbf{S} \pmod q$
$\mu' \leftarrow$ Frodo.Decode$(\mathbf{M})$ ▷ internal decryption step
Parse $pk \leftarrow seed_A||b$
$seed'_{SE}||k' \leftarrow$ SHAKE$(pkh||\mu', len_{seed_{SE}} + len_k)$ ▷ regenerate operational tokens
$(r^{(0)}, \ldots, r^{(2\bar{m}n+\bar{m}\bar{n}-1)}) \leftarrow$ SHAKE$(0x96||seed'_{SE}, (2\bar{m}n + \bar{m}\bar{n}) \cdot len_\chi)$
$\mathbf{S}' \leftarrow$ Frodo.SampleMatrix$((r^{(0)}, \ldots, r^{(\bar{m}n-1)}), \bar{m}, n, T_\chi)$
$\mathbf{E}' \leftarrow$ Frodo.SampleMatrix$((r^{(\bar{m}n)}, \ldots, r^{(2\bar{m}n-1)}), \bar{m}, n, T_\chi)$
$\mathbf{A} \leftarrow$ Frodo.Gen$(seed_A)$
$\mathbf{B}'' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}' \pmod q$
$\mathbf{E}'' \leftarrow$ Frodo.SampleMatrix$((r^{(2\bar{m}n)}, \ldots, r^{(2\bar{m}n+\bar{m}\bar{n}-1)}), \bar{m}, \bar{n}, T_\chi)$
$\mathbf{B} \leftarrow$ Frodo.Unpack$(b, n, \bar{n})$
$\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}'' \pmod q$
$\mathbf{C}' \leftarrow \mathbf{V} +$ Frodo.Encode$(\mu') \pmod q$
if $(\mathbf{B}' == \mathbf{B}'')$ **and** $(\mathbf{C} == \mathbf{C}')$ **then** ▷ executed in constant time
    $\bar{k} \leftarrow k'$
else
    $\bar{k} \leftarrow s$ ▷ implicit rejection safeguard string assignment
end if
$ss \leftarrow$ SHAKE$(c_1||c_2||\bar{k}, len_{ss})$
return ss

Module LWE-Based: ML-KEM (FIPS 203)

The FIPS 203 standard [NIS24b], announced in August 2023 by the NIST, specifies a Key Encapsulation Mechanism (KEM) called ML-KEM. ML-KEM is a standardized key-encapsulation mechanism derived from the CRYSTALS-Kyber framework [ABD+19]. It satisfies the rigorous criteria of indistinguishability under adaptive chosen-ciphertext attacks (IND-CCA2), rendering it semantically secure for general cryptographic deployments.

The ML-KEM standard natively supports three separate parameter sets to realize distinct trade-offs between computational performance and post-quantum security margins: ML-KEM-512, ML-KEM-768, and ML-KEM-1024. These configurations target NIST security levels 1, 3, and 5, respectively. The architectural differences regarding key lengths and data overhead are organized in Table 3.3.

	Encaps. Key	Decaps. Key	Ciphertext	Shared Secret Key
ML-KEM-512	800	1632	768	32
ML-KEM-768	1184	2400	1088	32
ML-KEM-1024	1568	3168	1568	32

Table 3.3 Sizes (in bytes) of keys and ciphertexts across ML-KEM variants [NIS24b]

A defining operational feature of ML-KEM is its deep architectural reliance on symmetric cryptographic functions. Virtually every structural routine—ranging from pseudorandom matrix generation via extendable-output functions (XOF), error distribution mapping through pseudorandom functions (PRF), to explicit key derivation pipelines—is instantiated using the Keccak-based primitives specified in the FIPS 202 standard [NIS24b]. Because these symmetric permutations are executed iteratively across all phases of the protocol, they form a major computational bottleneck alongside polynomial arithmetic. This heavy execution footprint directly underscores and justifies the development of the dedicated hardware accelerator presented in this work, which is designed to offload and optimize these Keccak execution layers.

To map out exactly where these symmetric blocks interface with the underlying lattice structures, the protocol is split into three foundational algorithmic workflows:

- Key Generation (`ML-KEM.KeyGen`);
- Encapsulation (`ML-KEM.Encaps`);
- Decapsulation (`ML-KEM.Decaps`).

The core subroutine of `ML-KEM.KeyGen` (Algorithm 10) is the internal key generation engine of the public-key encryption layer, `K-PKE.KeyGen` (Algorithm 11). The global ML-KEM encapsulation key (`ek`) directly mirrors the underlying $\texttt{ek}_{PKE}$ string. Conversely, the decapsulation key (`dk`) is structured as a composite vector concatenating the raw $\texttt{dk}_{PKE}$, the public encapsulation

key ek, a localized hash evaluation $H(\texttt{ek}||z)$, and a high-entropy 32-byte pseudorandom sequence z. This stored value z functions as an alternative cryptographic path within the decapsulation algorithm to handle malformed data safely.

The functional layout of ML-KEM.Encaps (Algorithm 12) utilizes the public-key encryption component K-PKE.Encrypt (Algorithm 13) to transform a 32-byte ephemeral seed m into a secure ciphertext c. Concurrently, the final shared secret key K and the active encryption randomness r are jointly derived by binding the message m to a digest of the encapsulation key through a specialized evaluation of the symmetric hash function G. The routine finishes by packing and transmitting the output tuple (K, c).

Finally, ML-KEM.Decaps (Algorithm 14) begins by parsing the segmented elements of the incoming decapsulation key dk. The target ciphertext c is processed via the decryption function K-PKE.Decrypt (Algorithm 15) to extract a candidate message plaintext m'. To enforce rigid protection against active manipulation, the decapsulation engine replicates the original encapsulation environment: it computes matching candidate keying variables (K', r') via $G(m'||h)$ and reconstructs an internal verification ciphertext c' by re-encrypting m' with randomness r'. At this stage, the sampler checks whether the reconstructed string c' matches the received ciphertext c. If a mismatch is detected, the primitive triggers an *implicit rejection* step; rather than failing loudly and leaking info, it rewrites the shared secret K into a pseudorandom fallback string generated by hashing the ciphertext c together with the embedded key component z via the function J. Otherwise, it securely returns the authentic derived key K'.

Algorithm 10 ML-KEM.KeyGen()

Generates an encapsulation key and a corresponding decapsulation key.
Output: Encapsulation key $\texttt{ek} \in \mathbb{B}^{384k+32}$.
Output: Decapsulation key $\texttt{dk} \in \mathbb{B}^{768k+96}$.

1: $z \xleftarrow{\$} \mathbb{B}^{32}$ ▷ z is 32 random bytes
2: $(\texttt{ek}_{PKE}, \texttt{dk}_{PKE}) \leftarrow$ K-PKE.KeyGen() ▷ run key generation for K-PKE
3: $\texttt{ek} \leftarrow \texttt{ek}_{PKE}$ ▷ KEM encaps key is just the PKE encryption key
4: $\texttt{dk} \leftarrow (\texttt{dk}_{PKE}||\texttt{ek}||H(\texttt{ek}||z))$ ▷ KEM decaps key includes PKE decryption key
5: **return** $(\texttt{ek}, \texttt{dk})$

Algorithm 11 K-PKE.KeyGen()

Generates an encryption key and a corresponding decryption key.
Output: Encryption key $\texttt{ek} \in \mathbb{B}^{384k+32}$.
Output: Decryption key $\texttt{dk} \in \mathbb{B}^{384k}$.

$z \xleftarrow{\$} \mathbb{B}^{32}$ ▷ z is 32 random bytes
$(\rho, \sigma) \leftarrow G(d)$ ▷ expand to two pseudorandom 32-byte seeds
$N \leftarrow 0$
for $(i \leftarrow 0;\ i < k;\ i++)$ **do** ▷ generate matrix $\hat{\mathbf{A}} \in (\mathbb{Z}_q^{256})^{k \times k}$
  for $(j \leftarrow 0;\ j < k;\ j++)$ **do**
    $\hat{\mathbf{A}}[i, j] \leftarrow \textbf{SampleNTT}(\text{XOF}(\rho, i, j))$
    ▷ each entry of $\hat{\mathbf{A}}$ uniform in NTT domain
  end for
end for
for $(i \leftarrow 0;\ i < k;\ i++)$ **do** ▷ generate $\mathbf{s} \in (\mathbb{Z}_q^{256})^k$
  $\mathbf{s}[i] \leftarrow \textbf{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(\sigma, N))$
  ▷ $\mathbf{s}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
  $N \leftarrow N + 1$
end for
for $(i \leftarrow 0;\ i < k;\ i++)$ **do** ▷ generate $\mathbf{e} \in (\mathbb{Z}_q^{256})^k$
  $\mathbf{e}[i] \leftarrow \textbf{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(\sigma, N))$
  ▷ $\mathbf{e}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
  $N \leftarrow N + 1$
end for
$\hat{\mathbf{s}} \leftarrow \text{NTT}(\mathbf{s})$ ▷ NTT is run k times (once for each coordinate of $\mathbf{s}$)
$\hat{\mathbf{e}} \leftarrow \text{NTT}(\mathbf{e})$ ▷ NTT is run k times
$\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$ ▷ noisy linear system in NTT domain
$\texttt{ek}_{PKE} \leftarrow \text{ByteEncode}_{12}(\hat{\mathbf{t}}) || \rho$
▷ ByteEncode_{12} is run k times; include seed for $\hat{\mathbf{A}}$
$\texttt{dk}_{PKE} \leftarrow \text{ByteEncode}_{12}(\hat{\mathbf{s}})$ ▷ ByteEncode_{12} is run k times
return $(\texttt{ek}_{PKE}, \texttt{dk}_{PKE})$

Algorithm 12 ML-KEM.Encaps(ek)

Uses the encapsulation key to generate a shared key and an associated ciphertext.
Validated input: encapsulation key $\texttt{ek} \in \mathbb{B}^{384k+32}$.
Output: shared key $K \in \mathbb{B}^{32}$.
Output: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.

$m \xleftarrow{\$} \mathbb{B}^{32}$ ▷ m is 32 random bytes
$(K, r) \leftarrow G(m || H(\texttt{ek}))$ ▷ derive shared secret key K and randomness r
$c \leftarrow \text{K-PKE.Encrypt}(\texttt{ek}, m, r)$ ▷ encrypt m using K-PKE with randomness r
return (K, c)

Algorithm 13 K-PKE.Encrypt($\mathtt{ek}_{PKE}, m, r$)

Uses the encryption key to encrypt a plaintext message using the randomness r.

Input: encapsulation key $\mathtt{ek} \in \mathbb{B}^{384k+32}$.
Input: message $m \in \mathbb{B}^{32}$.
Input: encryption randomness $r \in \mathbb{B}^{32}$.
Output: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.

$N \leftarrow 0$
$\hat{\mathbf{t}} \leftarrow \text{ByteDecode}_{12}(\mathtt{ek}_{PKE}[0 : 384k])$
$\rho \leftarrow \mathtt{ek}_{PKE}[384k : 384k+32]$ ▷ extract 32-byte seed from $\mathtt{ek}_{PKE}$
for ($i \leftarrow 0$; $i < k$; $i++$) **do** ▷ re-generate matrix $\hat{\mathbf{A}} \in (\mathbb{Z}_q^{256})^{k \times k}$
  for ($j \leftarrow 0$; $j < k$; $j++$) **do**
    $\hat{\mathbf{A}}[i, j] \leftarrow \textbf{SampleNTT}(\text{XOF}(\rho, i, j))$
  end for
end for
for ($i \leftarrow 0$; $i < k$; $i++$) **do** ▷ generate $\mathbf{r} \in (\mathbb{Z}_q^{256})^k$
  $\mathbf{r}[i] \leftarrow \textbf{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(r, N))$
  ▷ $\mathbf{r}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
  $N \leftarrow N + 1$
end for
for ($i \leftarrow 0$; $i < k$; $i++$) **do** ▷ generate $\mathbf{e}_1 \in (\mathbb{Z}_q^{256})^k$
  $\mathbf{e}_1[i] \leftarrow \textbf{SamplePolyCBD}_{\eta_2}(\text{PRF}_{\eta_2}(r, N))$
  ▷ $\mathbf{e}_1[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
  $N \leftarrow N + 1$
end for
$e_2 \leftarrow \textbf{SamplePolyCBD}_{\eta_2}(\text{PRF}_{\eta_2}(r, N))$
▷ $e_2 \in \mathbb{Z}_q^{256}$ sampled from CBD
$\hat{\mathbf{r}} \leftarrow \text{NTT}(\mathbf{r})$ ▷ NTT is run k times
$\mathbf{u} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$ ▷ NTT^{-1} is run k times
$\mu \leftarrow \text{Decompress}_1(\text{ByteDecode}_1(m))$
$v \leftarrow \text{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \mu$ ▷ encode plaintext m into polynomial v
$c_1 \leftarrow \text{ByteEncode}_{d_u}(\text{Compress}_{d_u}(\mathbf{u}))$ ▷ ByteEncode_{d_u} is run k times
$c_2 \leftarrow \text{ByteEncode}_{d_v}(\text{Compress}_{d_v}(v))$
return $c \leftarrow (c_1 || c_2)$

Algorithm 14 ML-KEM.Decaps(c,dk)

Uses the decapsulation key to produce a shared key from a ciphertext.
Validated input: ciphertext $c \in \mathbb{B}^{32(d_u k+d_v)}$.
Validated input: decapsulation key $\texttt{dk} \in \mathbb{B}^{768k+96}$.
Output: shared key $K \in \mathbb{B}^{32}$.

$\texttt{dk}_{PKE} \leftarrow \texttt{dk}[0 : 384k]$
▷ extract PKE decryption key from KEM decaps key
$\texttt{ek}_{PKE} \leftarrow \texttt{dk}[384k : 768k+32]$ ▷ extract PKE encryption key
$h \leftarrow \texttt{dk}[768k+32 : 768k+64]$ ▷ extract hash of PKE encryption key
$z \leftarrow \texttt{dk}[768k+64 : 768k+96]$ ▷ extract implicit rejection value
$m' \leftarrow$ K-PKE.Decrypt$(\texttt{dk}_{PKE}, c)$ ▷ decrypt ciphertext
$(K', r') \leftarrow G(m'||h)$
$\bar{K} \leftarrow J(z||c, 32)$
$c' \leftarrow$ K-PKE.Encrypt$(\texttt{ek}_{PKE}, m', r')$
▷ re-encrypt using the derived randomness r'
if $c \neq c'$ **then**
    $K \leftarrow \bar{K}$ ▷ if ciphertexts do not match, "implicitly reject"
else
    $K \leftarrow K'$ ▷ otherwise, accept and output derived key
end if
return K

Algorithm 15 K-PKE.Decrypt($\texttt{dk}_{PKE}, c$)

Uses the decryption key to decrypt a ciphertext.
Input: decryption key $\texttt{dk}_{PKE} \in \mathbb{B}^{384k}$.
Input: ciphertext $c \in \mathbb{B}^{32(d_u k+d_v)}$.
Output: message $m \in \mathbb{B}^{32}$.

$c_1 \leftarrow c[0 : 32d_u k]$
$c_2 \leftarrow c[32d_u k : 32(d_u k+d_v)]$
$\mathbf{u} \leftarrow \text{Decompress}_{d_u}(\text{ByteDecode}_{d_u}(c_1))$ ▷ ByteDecode_{d_u} invoked k times
$v \leftarrow \text{Decompress}_{d_v}(\text{ByteDecode}_{d_v}(c_2))$
$\hat{\mathbf{s}} \leftarrow \text{ByteDecode}_{12}(\texttt{dk}_{PKE})$
$w \leftarrow v - \text{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \text{NTT}(\mathbf{u}))$ ▷ NTT^{-1} and NTT invoked k times
$m \leftarrow \text{ByteEncode}_1(\text{Compress}_1(w))$ ▷ decode plaintext m from polynomial w
return m

3.3 Summary

This chapter has systematically mapped the operational progression required to transition Learning With Errors (LWE) primitives from abstract mathematical equations into structured, implementable cryptographic protocols. By tracing this continuum, the analysis presented herein establishes a clear structural hierarchy that connects entropy sources to high-level key-encapsulation algorithms, providing the foundational baseline for the architectural optimizations proposed in the remainder of this thesis.

The practical operational sequence begins with the generation of uniform data. While lattice-based theory assumes immediate access to perfect, uniform distributions over $\mathbb{Z}_q$, practical deployments handle this data requirement through a hybrid approach. Dedicated physical architectures (TRNGs) harvest certain forms of non-determinism to isolate a short, high-entropy cryptographic seed. This seed is subsequently handed over to deterministic expansion primitives (PRNGs), such as lightweight shift registers or standardized Keccak sponge permutations specified by the FIPS 202 standard, to algorithmically synthesize the computationally uniform bitstreams required to populate public system matrices.

A central focus of this structural review is the multi-stage *error generation chain*. Because raw uniform bits cannot satisfy the bounded, small-magnitude error profiles mandated to mask the lattice secrets safely, the uniform streams must be actively routed through specialized distribution shapers. This thesis has outlined the core methodologies utilized to map uniform inputs into targeted non-uniform profiles, such as discrete Gaussian curves or centered binomial distributions. Reviewing these error generation chains highlights the distinct algorithmic trade-offs, structural complexities, and inherent execution challenges associated with different sampling techniques, illustrating why optimizing or mitigating these sampling stages is a central objective of this work.

Finally, these independent randomness primitives, expansion mechanisms, and error generation chains are composed into complete post-quantum protocols. Specifically, this chapter presents two distinct algorithmic frameworks: the algebraically unstructured FrodoKEM (LWE-based) and the modular ring-structured ML-KEM (Module-LWE-based, FIPS 203). Tracing the data flow within these standardized subroutines highlights that symmetric cryptographic permutations, specifically the underlying Keccak (FIPS 202) core, are executed iteratively across all key generation, encapsulation, and decapsulation operations. By documenting these exact execution boundaries and tracking how the error generation chains compose the final public keys and ciphertexts, this chapter highlights the computational bottlenecks that guide and define the dedicated hardware accelerator implementations presented in the subsequent chapters.

Chapter 4

FIPS 202 Masked Hardware Accelerator

Following the foundational context established in Chapter 3, the practical performance of lattice-based post-quantum cryptography is heavily determined by symmetric primitives. While schemes based on the Learning With Errors (LWE) framework, such as FrodoKEM and the standardized Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM) (FIPS 203), are mathematically defined over high-dimensional lattice structures, their actual runtime is largely spent expanding short seed values into large pseudorandom public matrices and generating discrete error distributions. In compliant implementations, these operations rely on the Keccak sponge construction standardized in FIPS 202 (detailed in Section 3.1.2), specifically eXtendable-Output Functions (XOFs) such as SHAKE. Consequently, hardware acceleration of the FIPS 202 core is essential not only to address this primary pseudorandomness generation bottleneck, but also to establish a concrete baseline for evaluating the silicon area, latency, and power costs of post-quantum implementations.

Beyond purely algorithmic performance, evaluating these implementation costs becomes particularly insightful when taking physical security into account for practical implementations. In embedded environments, cryptographic devices operating in the field may be exposed to Side-Channel Analysiss (SCAs), such as Differential Power Analysis (DPA), which exploit physical side-channel leakage, including transient current consumption and electromagnetic radiation, to reconstruct sensitive intermediate variables. In this context, considering algorithmic countermeasures like Domain-Oriented Masking (DOM) based on secret sharing provides a realistic perspective on the area, power, and latency trade-offs inherent to protected hardware designs. Beyond the register and datapath replication inherent to hardware masking, the continuous generation of fresh randomness represents a challenge. In a high-throughput, one-round-per-cycle Keccak core, the non-linear χ transformation requires up to 1600 bits of fresh randomness at every clock cycle to satisfy non-completeness and prevent security degradation caused by physical circuit glitches. Producing this volume of pseudorandom bits at hardware speed introduces significant implementation costs, since physical True Random Number Generators (TRNGs) cannot deliver multi-gigabit throughput without excessive area overhead, and simple

linear structures like Linear Feedback Shift Registers (LFSRs) suffer from cryptographic weaknesses and state-recovery vulnerabilities. As a result, the randomness generation infrastructure itself becomes a primary contributor to the overall area, power, and complexity of a protected cryptographic core.

Within the broader scope of this thesis, which aims to mitigate the overheads of explicit error and pseudorandomness generation in post-quantum hardware, this chapter establishes a real-world baseline by evaluating the physical implementation costs of conventional FIPS 202 acceleration. Investigating this core is essential to quantify the concrete hardware footprint of standard pseudorandom generation pipelines, particularly when accounting for the additional entropy demands introduced by physical side-channel countermeasures. To this end, this chapter details the hardware design and cost trade-offs of a FIPS 202 accelerator capable of operating in both unmasked and first-order masked configurations. To manage the massive entropy consumption of the DOM-protected datapath without relying on prohibitive hardware resources, we evaluate a dedicated *Randomness Dispatcher* module based on parallelized, unrolled stream ciphers (Trivium and Bivium).

To obtain an accurate, realistic assessment of these implementation costs in an operational environment, the accelerator is integrated as an OBI-connected memory-mapped peripheral within a 32-bit RISC-V System-on-Chip (SoC) based on the CV32E40P core. This system framework enables the concrete measurement of latency, throughput, and silicon overhead when offloading the symmetric hashing and pseudorandom expansion stages of a standardized ML-KEM decapsulation sequence (FIPS 203). Furthermore, the physical security margins and leakage resilience of the masked core are experimentally validated through Test Vector Leakage Assessment (TVLA) using electromagnetic measurements. Ultimately, this quantitative evaluation highlights the substantial hardware overhead associated with conventional pseudorandomness generation and masking infrastructures, motivating the exploration of alternative design strategies that reduce explicit randomness chains across post-quantum architectures.

4.1 Side-Channel Attacks and Masking Countermeasures

While cryptographic algorithms offer robust mathematical guarantees, translating them into physical silicon inevitably exposes them to implementation-level vulnerabilities. Within the broader domain of hardware security, this analysis focuses on non-invasive Side-Channel Analysis (SCA). Rather than exploiting structural design flaws, SCA targets the inherent physical byproducts generated by integrated circuit logic during execution. By monitoring data-dependent observables, such as transient power consumption and localized electromagnetic emissions, an adversary can reconstruct sensitive intermediate variables and extract secret keys, bypassing the theoretical security of the cryptosystem. Consequently, protecting cryptographic hardware against these physical leakages requires dedicated countermeasures, most notably algorithmic masking, to decorrelate the processed secret data from the device's physical emissions.

4.1.1 Side-Channel Analysis

SCA encompasses a specialized category of cryptographic attacks aimed at extracting sensitive assets, such as secret cryptographic keys or ephemeral data, by monitoring the physical phenomena generated by a device during execution. As noted by [CGF21], while abstract cryptosystems are intuitively modeled as ideal *black boxes* that interact exclusively through well-defined input and output interfaces, this abstraction collapses when evaluating physical hardware. Real-world physical devices routinely emit unintended signals, or *leaks*, stemming from their underlying analog behavior. This unintended leakage provides an adversary with an indirect window into internal signals and intermediate computational states that are otherwise inaccessible via standard software boundaries. Crucially, side-channel analysis relies entirely on exploiting these physical execution byproducts rather than attempting to mathematically compromise or cryptanalyze the underlying cryptographic algorithm itself.

A side-channel attack exploits physical vulnerabilities in an attempt to extract confidential data from an integrated circuit or embedded system. Typical vectors leveraged by these techniques include supply current fluctuations, execution timing variations, and localized electromagnetic emissions. Such attacks severely compromise the security of integrated cryptographic architectures. Indeed, practical demonstrations have proven their capability to extract secret cryptographic keys even from mathematically robust algorithms, such as in the seminal timing attacks against RSA implementations [DH96]. Consequently, these methodologies do not target the mathematical structure of a cryptographic algorithm or software suite directly, but rather, they reconstruct internal operations by observing the passive, indirect physical byproducts of the underlying circuitry. The literature classifies side-channel analysis into several distinct operational vectors, including:

- **Timing Attack**: Exploits data-dependent variations in the total execution time required to complete a cryptographic routine.
- **Simple Power Analysis (SPA)**: Relies on the direct visual or qualitative observation of power consumption waveforms to map out macro-level operations, such as identifying key-dependent conditional branches.
- **Differential Power Analysis (DPA)**: Partitions a large population of execution traces based on a guessed bit of information and utilizes difference-of-means testing to isolate minute, data-dependent correlations hidden beneath ambient noise.
- **Correlation Power Analysis (CPA)**: A more powerful evolution of DPA that models the physical leakage of the device (e.g., using Hamming weight or distance) and employs the Pearson correlation coefficient across the traces to extract the secret key with significantly higher efficiency.

- **Template Attack**: Profiles a fully controlled replica device to construct a highly precise, multivariate statistical template of the hardware's leakages, subsequently mapping minimal trace data from the target device to extract secret keys.

DPA stands as one of the most prevalent and effective side-channel threats against secure hardware [KJJR11]. Rather than relying on the visual inspection of macro-operations, it applies rigorous statistical analysis across a large population of physical measurements. The technique exploits data-dependent variations in the electrical power consumption or electromagnetic emissions of the target hardware during computation. This physical leakage is recorded in the form of execution traces, where a *trace* is a time-series sequence of digitized physical measurements captured over the operational duration of a cryptographic or, more generally, electronic device. The foundational methodology of a classical DPA attack involves partition testing, where a collection of traces is split into distinct subsets governed by a selection function that predicts a specific intermediate algorithmic state. By computing the difference between the mathematical averages of these subsets, ambient algorithmic and environmental noise can be effectively cancelled out. Consequently, given a sufficient number of traces, precise data-dependent correlations can be successfully isolated, exposing the underlying secret variables.

Building upon the foundational principles of differential analysis, Correlation Power Analysis (CPA) represents a more powerful and highly efficient evolution of statistical side-channel threats [BCO04]. Rather than simply partitioning execution traces into binary subsets, CPA leverages a formalized, linear model to represent the target device's physical leakage. The technique maps predicted intermediate algorithmic states to expected power or electromagnetic emissions using specific hardware leakage abstractions, most commonly the Hamming weight of a processed data word or the Hamming distance between consecutive state transitions. The core methodology of a CPA attack involves calculating these hypothetical leakage values for every possible guess of a secret sub-key across the entire trace population. These theoretical predictions are then systematically compared against the actual digitized physical measurements using the Pearson correlation coefficient. By evaluating the statistical correlation between the modeled leakage and the physical traces, the correct key hypothesis emerges as the one exhibiting the highest correlation peak at the exact time instance the target operation occurs. Because it utilizes the combined leakage of all switching bits simultaneously rather than targeting isolated partitions, CPA typically converges much faster than classical DPA, requiring significantly fewer traces to successfully expose the underlying secret variables.

4.1.2 The Masking Countermeasure

Masking is a formally verifiable countermeasure designed to protect cryptographic implementations against side-channel analysis, such as DPA. It represents one of the most widely adopted defense mechanisms due to its architectural versatility, as it can be integrated at both the software

algorithmic level and the hardware logic gate level. This gate-level paradigm is of primary interest in this work, as it directly addresses physical silicon implementations.

At its core, masking relies on the principles of secret sharing, specifically Boolean secret sharing, where a sensitive intermediate variable x is split into $d+1$ randomized variables called *shares* $(x_0, x_1, \ldots, x_d)$. Here, the parameter d denotes the *masking order* (or protection order), which dictates the maximum number of independent shares an adversary can jointly observe without gaining information about the native secret. In a basic linear configuration, these shares are generated such that their bitwise XOR sum reconstructs the original secret $(x = x_0 \oplus x_1 \oplus \cdots \oplus x_d)$. Because of the random nature of the injected masks, any sub-combination of fewer than $d+1$ shares remains completely statistically independent of the unmasked value. In the context of a CPA attack, this statistical independence forces an adversary to execute higher-order analyses, which exponentially increases the number of physical execution traces required to isolate the secret key. At the conclusion of the processing pipeline, the independent output shares are recombined via a final XOR operation to retrieve the unmasked result.

Despite its theoretical elegance, masking carries a non-negligible implementation cost. The data footprint and the underlying processing pathways increase proportionally to the number of shares, as each independent share must undergo identical operations while continuously consuming fresh entropy for mask generation. For a hardware implementation, this replication of logic structures translates to a silicon area footprint and power consumption envelope that can be several times larger than the corresponding unmasked cryptographic system, scaling rapidly with the masking order d.

Furthermore, in real hardware environments, naive Boolean masking fails to provide security due to physical glitches (i.e., transient, non-ideal switching activities in combinational circuits) which can prematurely re-couple independent shares and leak the secret data. To mitigate these hardware-specific vulnerabilities and manage the associated area overheads, advanced non-linear masking frameworks such as Threshold Implementations (TI) and Domain-Oriented Masking (DOM) must be deployed. The operational mechanics, structural properties, and implementation paradigms of these two specific methodologies are detailed in the remainder of this section.

Threshold Implementation

Threshold Implementations (TI) is a provably secure masking paradigm specifically engineered to protect hardware implementations against side-channel analysis, even in the presence of physical glitches. Rather than masking logic gates individually, TI decomposes the underlying cryptographic algorithms and non-linear operations into multiple shared sub-operations known as *component functions*. Each component function processes a targeted subset of the independent shares concurrently, effectively hiding the intermediate computational values that would otherwise leak sensitive information. To guarantee both algebraic validity and side-

channel resilience, these component functions must strictly satisfy three foundational properties: correctness, non-completeness, and uniformity.

Following the formalizations in [GMK16], the *correctness* property dictates that the combination or sum of the outputs from all component functions must exactly reconstruct the true, unshared result of the original cryptographic operation. For first-order secure designs, the *non-completeness* property mandates that each individual component function must be completely independent of at least one share from every input variable. From a hardware engineering perspective, preventing transient glitches from violating non-completeness across cascading non-linear stages requires inserting pipeline registers at these component function boundaries. Finally, the *uniformity* property requires the input and output distributions of the shares to remain perfectly uniform, regardless of the underlying unshared secrets they represent. Because non-linear mapping operations inherently tend to degrade distribution entropy, maintaining uniformity across cascaded stages represents a significant design challenge, frequently necessitating share-refreshing techniques that consume fresh random bits to restore statistical uniformity [RBN+15].

Domain-Oriented Masking

Unlike TI, implementations leveraging Domain-Oriented Masking (DOM) [GMK16] assign each individual share of a secret variable to a strictly designated, independent share domain. For example, the shares A_x and B_x of a sensitive variable x are allocated exclusively to domain A and domain B, respectively. To achieve d-th order security under the d-probing model, a DOM architecture utilizes $d+1$ shares per variable distributed across $d+1$ mutually isolated domains.

The core paradigm of the DOM approach relies on preserving the strict statistical independence of these domains during execution. In a first-order secure configuration ($d = 1$), if a component function processes inputs A_x and A_y belonging entirely to domain A, all internal intermediate values calculated within that circuit remain completely independent of the unshared secrets x and y (where $x = A_x \oplus B_x$ and $y = A_y \oplus B_y$). This independence holds because the complementary shares B_x and B_y are excluded from this specific component function, being processed instead by a parallel structure within domain B.

For linear operations, maintaining domain isolation is trivial because the underlying calculations combine shares exclusively within their native domains. The primary challenge, as with all hardware masking schemes, arises during the evaluation of non-linear functions. When computing non-linear terms (i.e., the logical AND of two masked variables, where $z = (x_A \oplus x_B) \cdot (y_A \oplus y_B)$ expanding into cross-domain products like $x_A \cdot y_B$), domain boundaries must inevitably be crossed, requiring dedicated countermeasures to prevent security degradation. DOM addresses these cross-domain vulnerabilities by systematically injecting fresh randomness (Z) and inserting pipeline registers to block cross-domain glitch propagation.

A structural schematic of a first-order DOM multiplier is illustrated in Figure 4.1. Within the linear execution paths where operations remain confined to their native domains, such as the domain-A multiplication of A_x and A_y (represented by the black path on the left) and the domain-B multiplication of B_x and B_y (represented by the blue path on the right), no random refreshing or resharing is required. Conversely, the cross-domain multiplication terms (highlighted by the red central paths) are highly critical for side-channel security. To safeguard these boundaries, the intermediate cross-products are immediately blended with the fresh random mask Z and isolated by subsequent pipeline registers, successfully preventing transient physical glitches from propagating across domains and preserving the required non-completeness properties.

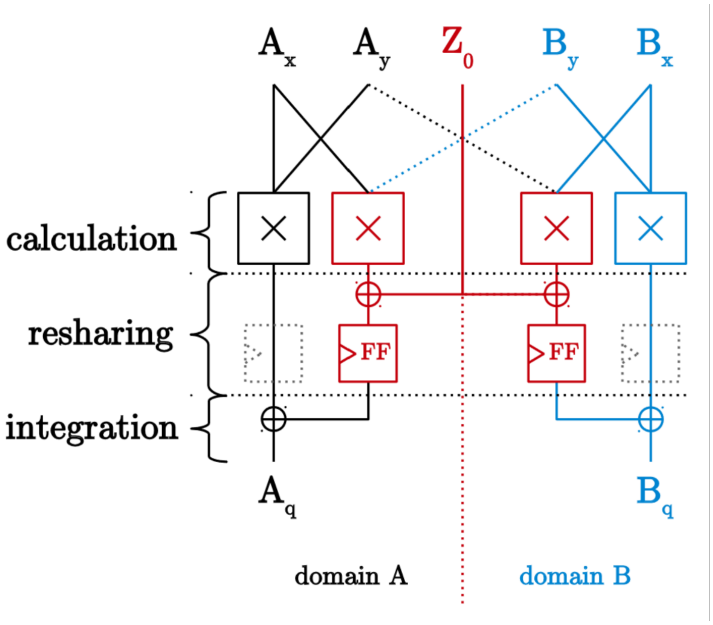


Figure 4.1 First-order DOM multiplier [GMK16]

4.1.3 Introduction to Test Vector Leakage Assessment (TVLA)

To validate the physical security of cryptographic implementations, evaluation laboratories require testing methodologies that are fast, reproducible, and independent of specific adversarial strategies [GGJR+11, SM16]. Traditional side-channel evaluations rely on executing a battery of known key-recovery attacks, such as classical Differential Power Analysis (DPA) or Correlation Power Analysis (CPA). However, this approach is fundamentally constrained by the expertise of the evaluator and the specific choice of a hypothetical leakage model. If an evaluator selects a poorly matching power model, the key-recovery attack may fail, leading to a false sense of security even though the device remains highly vulnerable to an adversary utilizing a more advanced or accurate model. To overcome this limitation, the Test Vector Leakage Assessment (TVLA) methodology, originally proposed by Goodwill et al. [GGJR+11] and widely adopted within the security community, shifts the evaluation focus from explicit key extraction to the more fundamental detection of statistically significant information leakage.

The core framework of TVLA is built upon statistical hypothesis testing. Rather than predicting internal intermediate states to guess a secret key, the procedure assesses whether any sensitive computational variable influences the physical side-channel emissions in a statistically significant manner. The underlying data collection stage segregates a population of recorded physical execution traces into two separate subsets, $\mathcal{Q}_0$ and $\mathcal{Q}_1$, based on the underlying data being processed. The methodology then evaluates these subsets against a defined null hypothesis (H_0), which states that the two trace populations possess identical means, meaning that the sensitive intermediate data exerts no detectable influence on the device's physical emissions. Conversely, the alternate hypothesis (H_1) asserts that the distributions' means are significantly different, proving the presence of data-dependent physical leakage.

To mathematically evaluate these hypotheses without assuming equal variances or sample sizes between the partitioned sets, TVLA utilizes the two-tailed Welch's t-test. For each discrete sample point across the operational time-series traces, the t-statistic is computed point-wise as follows:

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}}} \tag{4.1}$$

where μ_0 and μ_1 denote the sample means, s_0^2 and s_1^2 represent the sample variances, and n_0 and n_1 indicate the total number of traces (cardinality) within the respective subsets $\mathcal{Q}_0$ and $\mathcal{Q}_1$.

In practical leakage evaluations, two primary tracking modalities are deployed depending on the objective of the validation test:

- **Specific t-test:** This approach mirrors standard side-channel attacks by partitioning the trace population according to a specific, known internal algorithmic state, such as a targeted bit or byte of an intermediate function during execution. It determines whether a specific intermediate value actively leaks information through the physical substrate.
- **Non-specific t-test (Fixed vs. Random):** This modality isolates the evaluation from any specific power model or internal intermediate variable, enabling a comprehensive baseline assessment of a countermeasure's absolute physical efficacy. During data collection, the device under test is fed either a fixed, preselected input vector or a dynamically changing random input vector in a non-deterministic, randomly interleaved fashion governed by a random selection bit. The traces are then partitioned exclusively based on whether they processed the fixed or the random dataset.

To eliminate environmental noise, time-dependent variations, and equipment-induced drifts that could create false-positive results, the data collection for the subsets must be handled in a strictly interleaved manner. In standard cryptographic validation testing, a constant significance threshold of $|t| > 4.5$ is conventionally defined to reject the null hypothesis. Choosing a threshold of 4.5 ensures that the probability of a false positive occurring purely by chance at any given time instant is extremely low ($p < 0.00001$), providing a statistical confidence level exceeding

99.999%. Consequently, any point along the execution time-series where the computed t-statistic breaches these threshold boundaries ($t > 4.5$ or $t < -4.5$) serves as definitive mathematical proof of information leakage, allowing reviewers to rapidly identify implementation flaws or countermeasure failures.

4.1.4 Masked ML-KEM Implementation

We summarize the construction of the ML-KEM scheme (Section 3.2.2), which proceeds in two main steps: the first leverages the hardness of the MLWE problem to build a PKE scheme, which is used to encrypt and decrypt a shared secret, providing Indistinguishability under Chosen Plaintext Attack (IND-CPA) security. The second step converts the PKE scheme into a key-encapsulation mechanism using the Fujisaki-Okamoto (FO) transform, providing Indistinguishability under Chosen Ciphertext Attack (IND-CCA) security.

In ML-KEM, the decapsulation function is the most critical for both performance and security reasons compared to the others. According to the FO transform, the decapsulation function must decrypt the received ciphertext, re-encrypt it, and verify the equality between the received and re-encrypted ciphertexts. This procedure is the most time-consuming compared to key generation and encapsulation. Additionally, the decapsulation function may use the private key multiple times for different executions of the KEM protocol, which opens the door to a range of side-channel attacks that require multiple traces (e.g., power or electromagnetic emissions), such as DPA [FVBR+22]. For this reason, the decapsulation function is the main target for masking countermeasures, which protect against or at least mitigate such attacks.

Algorithm 16 ML-KEM.Decaps

Require: Ciphertext c
Require: Decapsulation key dk
1: $dk = (s, pk, \text{SHA3-HASH}(pk), z)$
Ensure: Shared key K
2: $m' \leftarrow \text{PKE.Decrypt}(s, c)$
3: $(K', r') \leftarrow \text{SHA3-HASH}(m' \,\|\, \text{SHA3-HASH}(pk))$
4: $K'' \leftarrow \text{SHA3-HASH}(z \,\|\, c)$
5: $c' \leftarrow \text{PKE.Encrypt}(pk, m', r')$
6: **if** $c = c'$ **then**
7: **return** K'
8: **else**
9: **return** K''
10: **end if**

Algorithm 16 reports the ML-KEM.Decaps function: it exploits the PKE.Decrypt function (line 1) to retrieve a shared secret message m', the $G()$ function (SHA3-512) to compute the shared key K', the $J()$ function (SHAKE-256) to compute an ephemeral shared key K'', and the PKE.Encrypt function to recompute the ciphertext to guarantee CCA security. Algorithm 17 and

Algorithm 17 PKE.Decrypt

Require: Secret key s
Require: Ciphertext $c = (c_1, c_2)$
Ensure: Message m
1: $\mathbf{u} \leftarrow \text{Decompress}_q(c_1, d_u)$
2: $v \leftarrow \text{Decompress}_q(c_2, d_v)$
3: $m \leftarrow \text{Compress}_q\left(v - s \cdot \mathbf{u}^\top, 1\right)$
4: **return** m

Algorithm 18 PKE.Encrypt

Require: Public key $pk = (\text{seed}_A, \mathbf{b})$
Require: Message m
Require: Seed r
Ensure: Ciphertext $c = (c_1, c_2)$
1: $\mathbf{A} \leftarrow \mathscr{U}(\mathbb{Z}_q^{k \times k})^{\text{seed}_A}$
2: $(\mathbf{s}', \mathbf{e}_1, \mathbf{e}_2) \leftarrow \Psi_{\eta_1} \times \Psi_{\eta_2} \times \Psi_{\eta_2}^{\text{SHA3-XOF}(r)}$
3: $\mathbf{u} \leftarrow \mathbf{A} \cdot \mathbf{s}' + \mathbf{e}_1$
4: $v \leftarrow \mathbf{b}^\top \cdot \mathbf{s}' + \mathbf{e}_2 + \lfloor q/2 \rfloor \cdot m$
5: $c_1 \leftarrow \text{Compress}_q(\mathbf{u}, d_u)$
6: $c_2 \leftarrow \text{Compress}_q(v, d_v)$
7: **return** (c_1, c_2)

Algorithm 18 report the PKE.Decrypt and PKE.Encrypt functions, respectively. SHAKE-256 is used to transform seeds into pseudo-random values for generating lattice parameters (line 2 of Algorithm 18), while SHAKE-128 is used to expand the matrix $\mathbf{A}$ from seed_A (line 1 of Algorithm 18).

A first-order masked implementation of the ML-KEM decapsulation function requires that all sensitive data are split into two independent shares. A schematic representation is shown in Figure 4.2. Here, variables in red are sensitive and split into shares (e.g., s^0, s^1), while variables in green are non-sensitive and remain unmasked. In Figure 4.2 we emphasize the role of SHA-3 functions: red boxes correspond to masked instances of SHA-3 functions operating on sensitive data, whereas green boxes correspond to unmasked instances operating on public data.

The masked PKE.Decrypt function (Algorithm 17) decrypts the ciphertext (c_1, c_2) with the masked secret key shares s^0 and s^1. The resulting masked message shares m'^0 and m'^1 are concatenated with $\mathscr{H}(pk)$ and hashed with the masked $G()$ function: the least significant 256 bits of the result correspond to the masked shared key (K'^0, K'^1), and the most significant 256 bits serve as a seed for the masked XOF (masked SHAKE-256) functions in the masked PKE.Encrypt. The sequence of XOF–CBD–B2Aq operations in Figure 4.2 corresponds to line 2 of Algorithm 18. The `B2Aq` block converts Boolean masked variables into their arithmetic representation. The public matrix $\mathbf{A}$ is generated from seed_A using the unmasked SHAKE-128. All polynomial operations of PKE.Encrypt (lines 3 and 4 of Algorithm 18) are grouped in the `Poly` box. The `Masked Compress` block (lines 5 and 6 of Algorithm 18) produces the masked

ciphertext shares, which must be securely compared with the original unmasked ciphertext: this can be done by verifying $\mathscr{H}(c^1 \oplus c_1^0) = \mathscr{H}(c_1^1)$ and $\mathscr{H}(c^2 \oplus c_2^0) = \mathscr{H}(c_2^1)$. If equality holds, the masked shared key shares K'^0 and K'^1 are returned; otherwise, the ciphertext and the masked random value shares z^0 and z^1 are hashed with the masked $J()$ function to produce the final output.

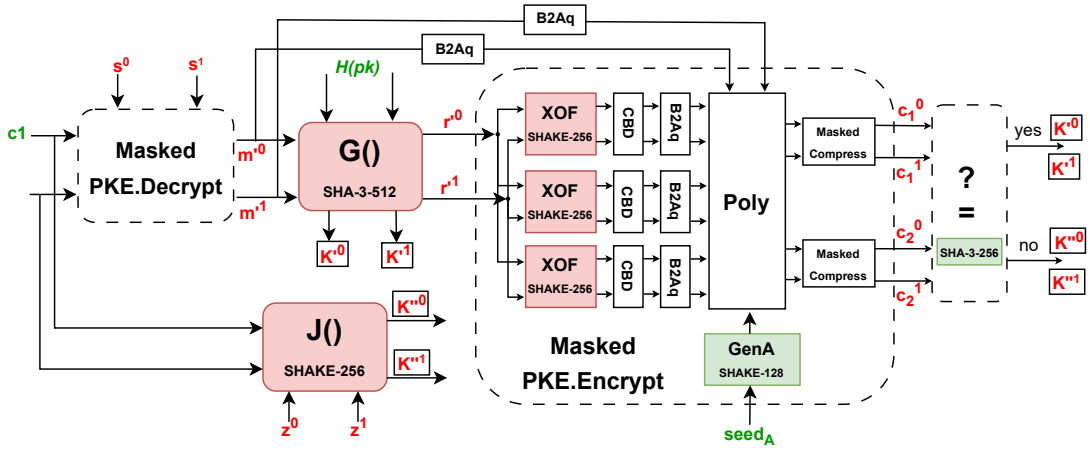


Figure 4.2 Schematic view of the first-order masked ML-KEM.Decaps function. Red boxes indicate FIPS 202 functions in masked form, while green boxes in unmasked form.

4.2 Existing Works

4.2.1 Masked Keccak Hardware Architectures

The first masked accelerator for Keccak was introduced by the Keccak developers in [BDPVA10]. This architecture employs a three-share scheme for first-order security. However, [BDN+13] later identified a security flaw due to the absence of the *uniformity property* in the non-linear χ step, which can lead to information leakage in side-channel attacks. To mitigate this, researchers proposed two solutions: adding a fourth data share or incorporating *fresh randomness* after each χ step. These approaches were implemented on ASIC hardware using the evaluation platform from [MKDH15], which includes both the original three-share non-uniform solution and its improved variants. Subsequent works introduced further enhancements. Hardware *glitches*, a known source of leakage, were addressed in [ABP+18] by combining *threshold implementations* (TI) with DOM. However, [MKDH15] demonstrated that applying TI resulted in the loss of the non-completeness property. To counteract this, [SC19] proposed an algorithm-level solution offering a low-latency implementation with countermeasure costs dependent on the Keccak permutation width.

The final design we analyzed, from [ZSS+21], was selected for our SHA-3 masked hardware accelerator. This implementation is optimized for a minimal number of shares, making it more efficient than TI-based solutions while maintaining a low-latency operation of one round per

clock cycle, similar to [BDPVA10]. Figure 4.3 shows the architectural scheme of the masked Keccak core from [ZSS+21]. The masking order d is set to 1, meaning the state is split into two independent shares. Each share has a width of 1600 bits, corresponding to the state size of the Keccak function used in the FIPS 202 standard.

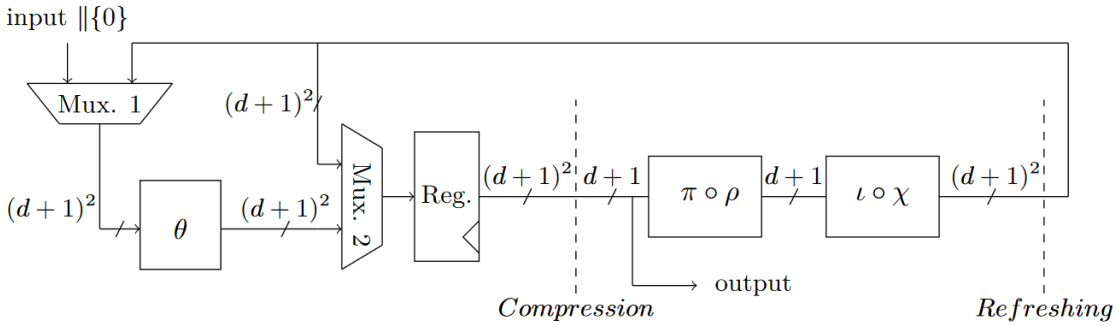


Figure 4.3 High-level architecture of the masked Keccak core [ZSS+21]. d is equal to the masking order. The input passes through diffusion (θ), registers, and the permutation steps ($\pi \circ \rho$) for shuffling/rotation and $\iota \circ \chi$ for nonlinear mixing. Refreshing ensures the state remains securely masked against side-channel leakage.

Here, we describe the Keccak core from a hardware perspective, focusing on the scenario considered in this paper, i.e., when $d = 1$. The input data can assume a bit size that depends on the selected FIPS 202 algorithm, as reported in column r of Table 3.1. Before *Mux. 1*, the inputs are padded with zeros, and four shares are constructed. The θ function acts on four shares, and it is followed by one 1600-bit register per share (6400 bits in total). This register is needed for two main purposes: it samples the results of each round of the Keccak transformations (functional purpose) and it prevents the recombination of the shares due to glitches (security purpose). In addition, to preserve *non-completeness*, the χ and θ functions are aligned so that each χ component receives only a single input share. This structure prevents unintended recombination, thereby mitigating glitches. The compression stage compresses four shares into two; this point of the circuit is used as the output point for the last round of Keccak, thanks also to *Mux. 2*, which bypasses the θ function at the end of the last round. According to the DOM principle, the refreshing function is applied to cross-domains to restore uniformity; this operation requires a 1600-bit share made of fresh randomness.

Unlike previous works that focus solely on accelerating masked Keccak, our contribution stands out by proposing a complete SHA-3 masked accelerator, fully integrated and operational within a realistic cryptographic context. Existing designs, while offering valuable solutions for masking Keccak, often neglect essential aspects of practical deployment, including the efficient management of randomness generation.

4.2.2 Randomness Generation for Masked Implementations

Masked implementations of cryptographic algorithms typically require a substantial amount of randomness per clock cycle, a factor often overlooked in research focused on masking techniques. However, for masked SHA-3 implementations, this issue is particularly critical: in one-round-per-cycle designs, executing Keccak demands up to 1600 bits of fresh randomness per clock cycle.

Most works in the literature adopt Linear Feedback Shift Registers (LFSRs) for this purpose, without further analysis of the security of this method. The work in [ZSS+21], for instance, employs a 31-bit LFSR with the feedback polynomial $x^{31} + x^{28} + 1$, which is seeded randomly at device power-up and updated at every clock cycle. A similar approach is adopted in [SM21] and [KSM22]. The work in [KM22] considers a 64-bit LFSR with the feedback polynomial $x^{64} + x^{63} + x^{61} + x^{60} + 1$ in order to increase the period of oscillation. Moreover, the authors state that LFSRs do not provide security against state recovery, meaning that knowledge of a certain number of outputs leads to the full recovery of the shift register. Nevertheless, the authors remark that security against state recovery may not be required in the context of SCA. In addition, the authors estimate the area overhead of LFSR-based solutions compared with a PRNG based on the Keccak function (unrolled and round-based). This analysis shows that only unrolled versions of Keccak can be suitable for randomness generation, but at a high cost in terms of resource consumption.

The work in [CMM+23] deeply analyzed different ways to generate randomness for secure hardware masking schemes; in particular, the authors evaluated the efficiency and security of three classes of random number generators: True Random Number Generators (TRNGs), LFSRs, and PRNGs based on stream ciphers, block ciphers, and permutations. In the case of TRNGs, the authors explored the architectures proposed in [YRG+18, LBS22, CFAF13, CFFA13], which represent the most efficient designs in the state of the art. The authors concluded that PRNG-based solutions present a higher throughput-to-area ratio compared to TRNGs. Regarding LFSRs, the authors demonstrated security weaknesses of unrolled LFSRs in three use cases with a practical analysis (i.e., a statistical t-test) on an FPGA and a theoretical attack. For PRNG implementations, the authors compared the most cost-efficient NIST lightweight block ciphers like Subterranean 2.0, Gimli, and SPEEDY, alongside the stream cipher Trivium and its reduced version, Bivium B, submitted to the eSTREAM competition.

The usage of stream ciphers as PRNGs has been shown to be of particular interest. Stream ciphers are commonly used for encryption, relying on internal architectures based on Non-Linear Shift Registers (NLFSRs). Stream ciphers take as input a secret key and an Initialization Vector (IV), and they produce a serial output of pseudo-random bits, commonly called a *keystream*. In the encryption scenario, the keystream can be used to encrypt a message through a bit-wise XOR of the message bits with the keystream bits. Alternatively, the keystream can be directly used as a *pseudo-random* sequence of bits. This latter usage is the one of interest for randomness

generation in masked implementations, as has been shown in [CMM+23]. The authors consider unrolled versions of the target stream ciphers, thus being able to generate multiple bits of pseudo-randomness per clock cycle. In particular, they show that the usage of the Trivium or Bivium stream ciphers with an unrolling level of 64 bits (i.e., 64 random bits generated per clock cycle) is the most efficient in terms of the area-performance trade-off.

4.3 Architecture Description

In this section, we describe the architecture of the proposed masked SHA-3 accelerator. The accelerator has been designed to be compatible with 32-bit systems, which is the word size for both input and output modes. In fact, specific control signals to handle bus transactions have been included in the design, both for control and data bits. Figure 4.4 shows the top-level architecture of the accelerator.

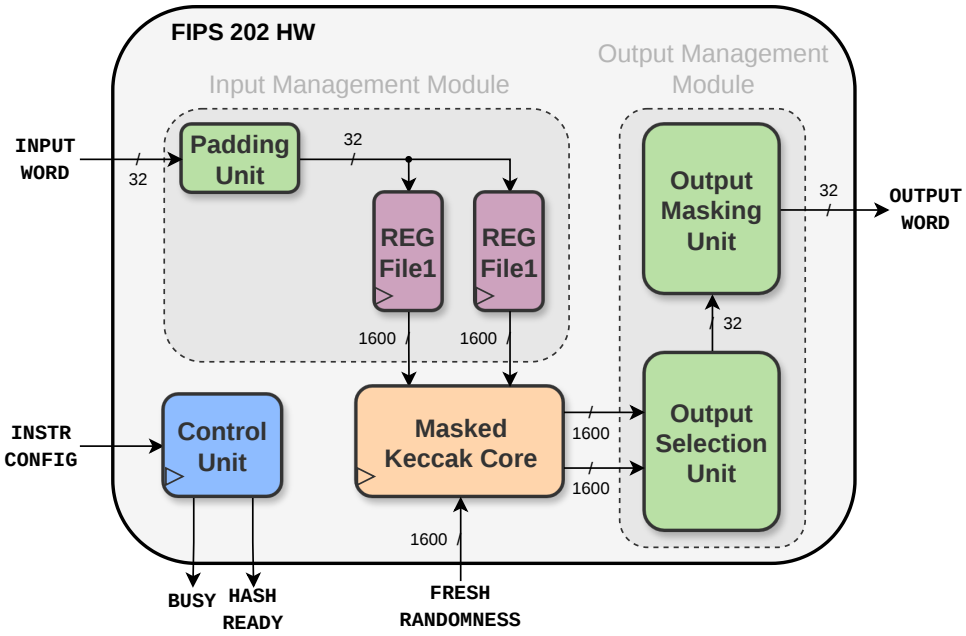


Figure 4.4 Architecture of the proposed masked SHA-3 accelerator.

The accelerator includes four main parts: the Input Management Module, the Output Management Module, the Masked Keccak core, and the Control Unit. The Input Management Module prepares input data, including padding, before it is stored in one of the register files (REG File in Figure 4.4). Two distinct 1344-bit register files are employed, each dedicated to a distinct share. Byte-wise padding, consisting of a delimiter, zero bytes, and an end marker, is applied exclusively to the 32-bit words of a single share. This design choice is crucial to avoid the deterministic cancellation of the padding during recombination: applying identical padding to both shares would cause the padding bytes to cancel out when they are combined. The hardware circuit is implemented through four byte-selection multiplexers forming a 32-bit output word.

These multiplexers select among original data, zero bytes, padding delimiters for SHA-3 and SHAKE, and padding end markers.

The Masked Keccak core is based on the work of [ZSS+21], as described in Subsection 4.2.1. The core executes one round of the Keccak function in one clock cycle on both shares. It takes as input two 1600-bit shares, so the outputs of the register files (1344 bits) are zero-padded before being sent to the Keccak core. The zero-padding is necessary for those bits that always belong to the capacity part of the state. The `FRESH RANDOMNESS` input is required by the DOM countermeasure, and it should be fed with 1600 bits of fresh randomness per clock cycle during the execution of Keccak. The Output Management Module constructs the output stream starting from the internal state of the Keccak core by selecting internal registers from the register files (via the Output Selection Unit) and zeroing bytes (via the Output Masking Unit), enabling byte-aligned outputs for SHAKE and SHA-3 functions. The Control Unit manages the operation flow and the modes of operation of the accelerator.

4.3.1 Modes of Operation

The proposed accelerator supports all the FIPS 202 algorithms, as reported in Table 3.1, in both masked and unmasked versions. In fact, the KEM.Decaps function requires both masked implementations of FIPS 202 algorithms (i.e., SHA-3-512 and SHAKE-256 in the red boxes of Figure 4.2) and unmasked implementations (i.e., SHAKE-128 and SHA-3-256 in the green boxes of Figure 4.2). This requirement can also be extended to other masked implementations of PQC algorithms. This functionality has been implemented thanks to a set of commands managed by the Control Unit. This module is in charge of orchestrating all the datapath components described in Section 4.3.

The Control Unit recognizes and manages the following commands:

- `RESET`: this command resets the entire accelerator by setting all internal registers to zero and forcing the internal Finite State Machine (FSM) to an idle state.
- `RD_SH1`: this command configures the datapath to store the first share of the input message into *REG File 1*. It expects one message word per clock cycle whenever input data is available; otherwise, it suspends the read operation.
- `RD_SH2`: this command configures the datapath to receive the second share of the input message, similar to the `RD_SH1` command, but the incoming share is stored into *REG File 2*. When the storing phase is complete, it proceeds with the computation of one of the supported FIPS 202 functions. This command must be used when the incoming data corresponds to the first block of a new message.
- `RD_SH2_ABSORB`: this command performs the storing phase in the exact same way as the previous one. When this phase is complete, it proceeds with a hash function absorption.

This command is used when the passed input data is a new block of the same input message whose computation was previously started.

- `SQUEEZE`: whenever a SHAKE function has been computed, this command allows the system to request the computation of additional bytes to realize the XOF functionality.
- `OUTPUT`: with this command, the accelerator outputs the two shares word-wise. They are output one after the other, potentially at a rate of one word per clock cycle.

The instruction format of the accelerator includes one of the commands described above plus the size of the message block to be processed. This field is accepted by the accelerator only for the commands `RD_SH1`, `RD_SH2`, and `RD_SH2_ABSORB`, which require knowledge of the message size to pad it according to the selected mode of operation; otherwise, it is ignored. Thanks to this mechanism, it is possible to manage both the masked and unmasked computation of FIPS 202 functions. In the case of masked computation, the first share shall be written after the command `RD_SH1`, while also indicating the size of the data block. Afterward, the second share can be written using `RD_SH2` or `RD_SH2_ABSORB`, depending on the absorption phase (first or non-first block of the message), again specifying the size of the block. It should be noted that in the masked computation, fresh randomness should be provided (1600 bits of fresh randomness per clock cycle during the computation of Keccak). In the unmasked case, the unique share (where the plaintext message is not divided into shares) should be provided in the same way as in the masked case. Afterward, one of the commands `RD_SH2` or `RD_SH2_ABSORB` shall be set (for the first or non-first block of the message), but the size of the message must be set to zero. In this way, the accelerator immediately starts the computation of the selected FIPS 202 algorithm in the unmasked version. In this case, the fresh randomness shall be set to zero.

4.3.2 Randomness Dispatcher

We have implemented a hardware module, called *Randomness Dispatcher*, in order to efficiently provide the 1600 bits per clock cycle of randomness required by the proposed masked SHA-3 accelerator. The Randomness Dispatcher is based on PRNGs, which can be implemented relying on the two stream cipher architectures, Trivium and Bivium, detailed in Subsection 3.1.2.

To fulfill the substantial throughput demands of the masking countermeasure, the implementation leverages the structural unrolling capabilities of these target ciphers. Rather than relying on the baseline configurations or the introductory 2-bit parallel layout shown in Figure 3.1, the Randomness Dispatcher scales up this concurrent evaluation strategy. By expanding the combinational path to process multiple algorithmic updates simultaneously within a single clock cycle, the module successfully scales the keystream generation rate to continuously deliver the required pseudorandom bits to the core datapath.

Following this principle, the proposed hardware architecture of the Randomness Dispatcher is presented in Figure 4.5. It includes 25 instances of a 64-bit unrolled version of Trivium, a

five-stage 32-bit shift register, and a finite state machine. The unrolled Trivium can be substituted by the unrolled Bivium cipher during the synthesis phase of the circuit. The output stream is composed of 1600 bits, which are the parallel outputs of the 25 Trivium ciphers, each of 64 bits.

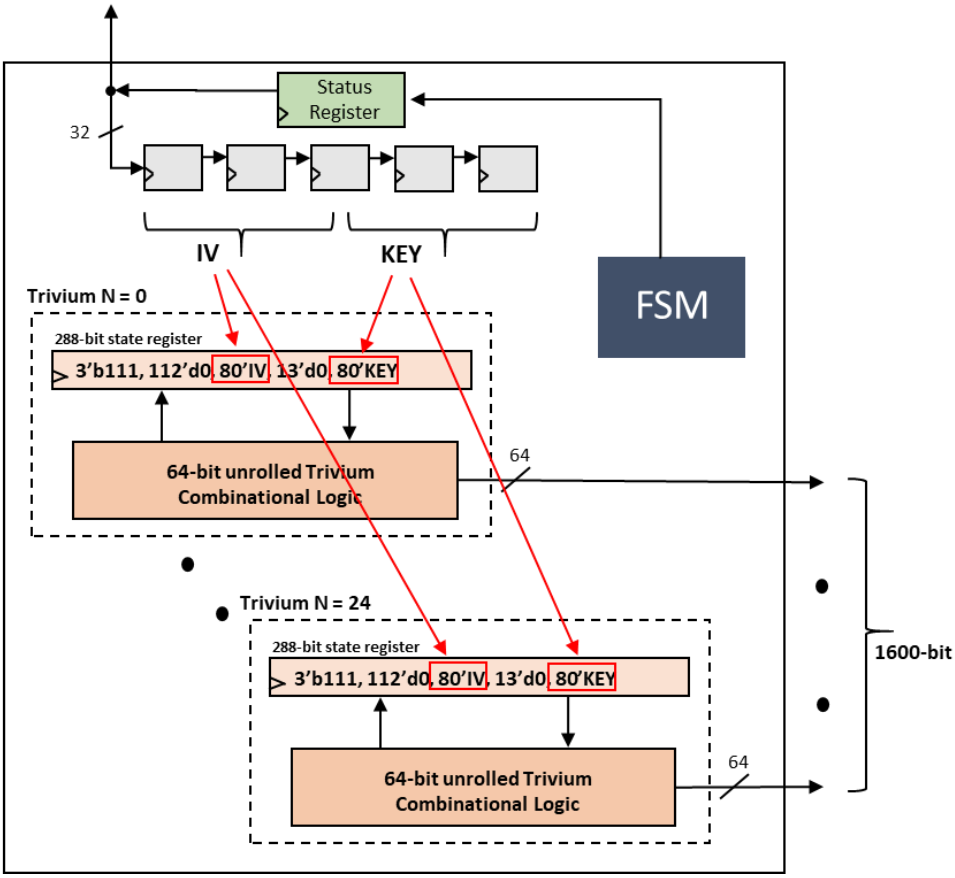


Figure 4.5 Hardware architecture of the Randomness Dispatcher. The design employs 25 parallel instances of Trivium or Bivium, each initialized with independent IV and key inputs. These parallel cores generate the required 1600-bit randomness output, controlled by the FSM and status register.

The circuit has four operating states, managed by the state machine:

- `Idle`: the Randomness Dispatcher is not initialized and the output is set to zero.
- `Initialization`: in this phase, all 25 Trivium ciphers must be initialized with the IV and the key. The shift register (160 bits in total) is used for this purpose: once all 160 bits have been written to the shift register (they correspond to the size of the IV and the key of each Trivium), its content is sampled by the state register of one of the Trivium stages. From a software point of view, this operation is completely transparent: the user shall only write the initialization sequence (160 bits for each Trivium) to the corresponding address where the Randomness Dispatcher has been mapped in the SoC. Each state register of each Trivium is automatically filled by the finite state machine. This mechanism also enables on-demand reseeding via software at any time, allowing the internal states to be refreshed without additional hardware.

- `Warm-up`: in this state, the content of the state registers of the 25 Trivium ciphers is randomized by means of four rotations of the state. This operation takes 18 clock cycles.
- `Running`: in this state, the Randomness Dispatcher generates 1600 bits of random strings per clock cycle.

4.3.3 Integration in a 32-bit RISC-V System-on-Chip

We integrated the proposed masked accelerator for SHA-3 and the Randomness Dispatcher into a RISC-V System-on-Chip (SoC), shown in Figure 4.6. It is built around the CV32E40P, a 32-bit RISC-V core optimized for embedded applications. The processor is connected to separate instruction and data memory blocks, thanks to a 32-bit Open Bus Interface (OBI). To support development and debugging, the SoC includes standard peripherals such as a JTAG interface and a UART module. At the core of the design is our masked SHA-3 accelerator, which is tightly coupled with the dedicated Randomness Dispatcher. This dispatcher ensures the secure and timely delivery of fresh randomness required by the Domain-Oriented Masking scheme. The accelerator is memory-mapped and controlled via the RISC-V core, enabling flexible integration with higher-level cryptographic protocols. It should be noted that in our SoC, no performance degradation due to bus contention occurs, since the CPU is the sole active bus master during software execution.

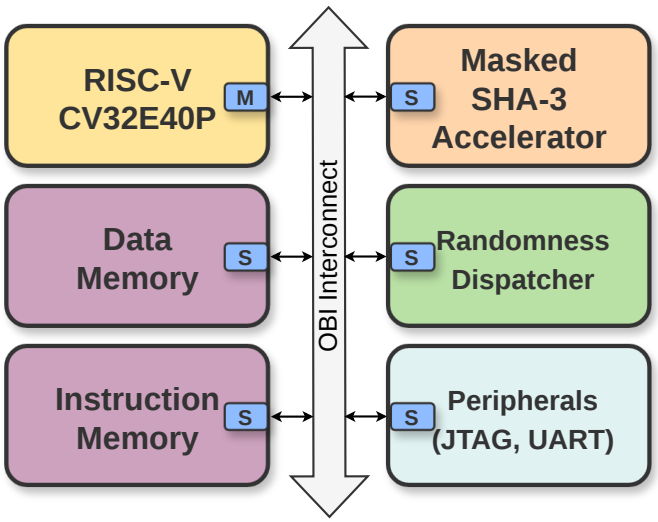


Figure 4.6 Integration of the Masked SHA-3 Accelerator and the Randomness Dispatcher in the target SoC.

4.4 Results and Comparison

4.4.1 Implementation Results on FPGA and ASIC

Table 4.1 reports the synthesis results of the proposed SHA-3 accelerator plus the Randomness Dispatcher on the Artix-7 FPGA contained in the Digilent Arty A7-100T development board. The synthesis tool is Vivado 2021.1, and no particular optimization strategies have been applied (Vivado Synthesis Defaults). The maximum frequency on the target FPGA is 163 MHz, with an area of 18,805 LUTs and 9,168 FFs. In the first row of Table 4.1, we also reported the synthesis results of our SHA-3 accelerator without the masking countermeasure. The maximum frequency is 166 MHz and the resource consumption is 6,990 LUTs and 4,706 FFs. It should be noted that both architectures follow the one-round-per-clock-cycle approach: this means that the Keccak function takes the same number of clock cycles (i.e., 24 clock cycles). Thus, excluding the fact that the masked version works with two shares (two shares shall be written in the register file of the module), the performance of the two implementations is the same in terms of clock cycles. The synthesis results show that the maximum frequency is not affected by the masking countermeasure on the FPGA. However, the countermeasure has a significant cost in terms of hardware resources: 269% for LUTs and 194% for FFs. Concerning the Randomness Dispatcher, the version with the Trivium stream cipher consumes 11,966 LUTs and 7,403 FFs. The version with the Bivium stream cipher is smaller, with 9,201 LUTs and 4,663 FFs. This analysis clearly shows that the area overhead of the masking countermeasure (first-order) on a one-round-per-clock-cycle SHA-3 architecture should also take into account the area of the Randomness Dispatcher, which cannot be neglected. So, the real overhead is around 440% for LUTs and 352% for FFs in the case of the Randomness Dispatcher implemented with the Trivium cipher, and around 400% of LUTs and 293% of FFs in the case of the Randomness Dispatcher implemented with the Bivium cipher.

Table 4.1 Synthesis Results of the SHA-3 Accelerator and the Randomness Dispatcher on an Artix-7 FPGA

Module	**LUTs**	**FFs**
SHA-3 Accelerator (Unmasked)	6,990	4,706
SHA-3 Accelerator (Masked)	18,805	9,168
Randomness Dispatcher (Trivium)	11,966	7,403
Randomness Dispatcher (Bivium)	9,201	4,663

We synthesized our accelerators on a standard cell 22nm FD-SOI technology. The synthesis is performed with Synopsys Design Compiler [Inc20]. Figure 4.7 presents a design space exploration (frequency vs. area) in this technology. In black, we show the area of the unmasked SHA-3 accelerator; in red, the area of the masked version. The orange section represents the combined area of the masked SHA-3 accelerator and the Randomness Dispatcher based on Bivium, while in pink, Trivium replaces Bivium as the Randomness Dispatcher. We evaluated

the area consumption in Gate Equivalents (GE), which is calculated as the ratio between the area of the circuit and the area of the smallest NAND2 gate of the technological library. The maximum frequency of the SHA-3 accelerator is 1.76 GHz. At this frequency, the area of the SHA-3 accelerator is 213 kGE, the area of the Randomness Dispatcher with Trivium is 130 kGE, while the Bivium version takes 86 kGE. At 735 MHz, the area of the SHA-3 accelerator becomes 186 kGE, while the area of the Randomness Dispatcher remains unchanged. The area of the unmasked version of the SHA-3 accelerator ranges from 54 kGE to 64 kGE, so the overhead of the masking countermeasure ranges from 500% (at 1.76 GHz with the Randomness Dispatcher based on Trivium) to 300% (at 735 MHz with the Randomness Dispatcher based on Bivium). Extending the accelerator to second-order masking would increase the accelerator area by more than 2× and require roughly 3× the amount of randomness [ZSS+21], making higher-order masking impractical in hardware.

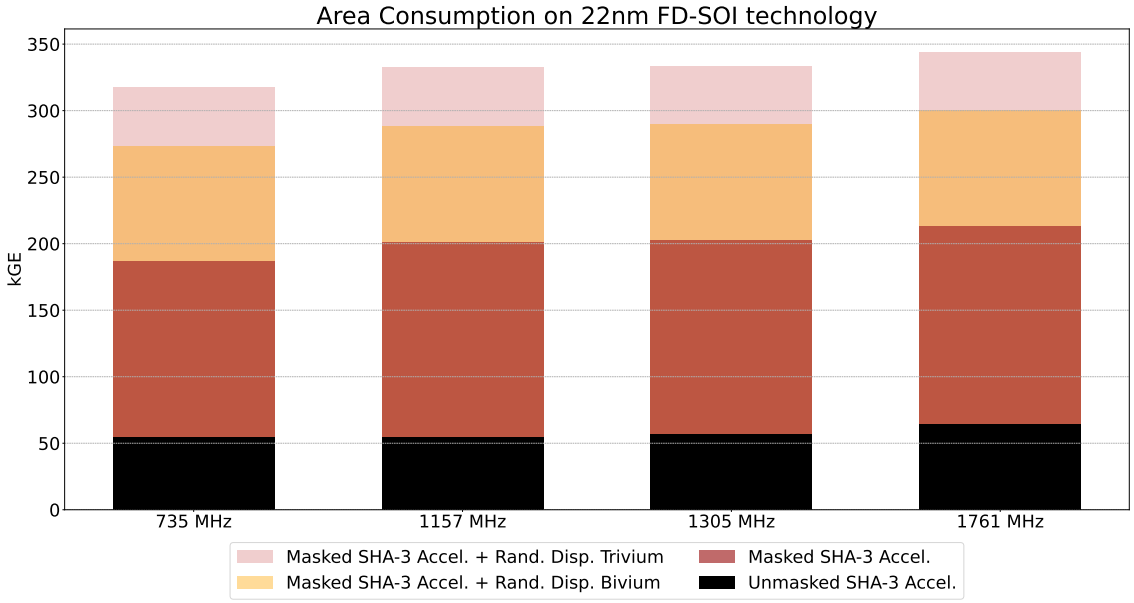


Figure 4.7 Synthesis results and design space exploration of the proposed accelerators on a 22nm FD-SOI standard cell technology.

It is worth noting that the observed area overhead of 300–500% is an inherent feature of masking-based countermeasures, which should be taken into account when considering the cost of masking. In fact, as we have shown, randomness generation, essential for ensuring the security of the DOM implementation, is a very expensive feature that is often overlooked in the literature. While this naturally results in notable overhead, our work provides a transparent and comprehensive analysis, reporting both the cost of the masked SHA-3 accelerator and the associated randomness generation infrastructure. The absolute overhead, though high, reflects the basic cost of achieving side-channel resistance and the trade-offs involved.

4.4.2 Security Analysis with Test Vector Leakage Assessment

In this section, we present the security analysis that has been conducted in order to validate the hardware countermeasure that has been implemented in the proposed SHA-3 accelerator. The chosen methodology is the Test Vector Leakage Assessment (TVLA) [GGJR+11, SM16], which is a well-established methodology in the side-channel community for leakage detection during implementation evaluation and for points-of-interest selection in attack scenarios. It is a pass/fail test based on Welch's t-statistic. Specifically, the statistic tests the hypothesis of whether the difference between the moments of two sets of samples is significant. Our security evaluation relies on TVLA to confirm the absence of first-order leakage. Although stronger attacks like CPA could offer additional insights, they are outside the scope of this study. The t-statistic is defined as:

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}}} \tag{4.2}$$

where μ_0 and μ_1 are the mean values of the two sets of measurements, s_0^2 and s_1^2 are their variances, and n_0 and n_1 are the number of samples in each set. The variant of the methodology that we adopt in this security analysis is the *non-specific* t-test, where samples from random inputs are compared against samples captured when executing SHA-3 functions with a previously selected fixed input. As for the threshold for the pass/fail test, we consider a sample to be leaking when $|t| > 4.5$, as widely employed in the literature [SM16]. We performed statistical leakage detection according to the TVLA methodology on the Arty A7-100 FPGA board, and we measured the electromagnetic emissions during the execution of SHA-3 and SHAKE algorithms. We used an electromagnetic probe connected to a Tektronix MSO 6 series oscilloscope for trace acquisition. Figure 4.8(a) reports the t-statistic obtained with 20k traces using the SHA-3 accelerator in the unmasked mode of operation. We remark that leakage appears throughout the execution. Figure 4.8(b) shows the t-statistic obtained with 20k traces using the SHA-3 accelerator in the masked version with the Randomness Dispatcher deactivated (OFF). In this case, the first-order masking is active, but the DOM countermeasure is not, and leakage appears due to the lack of randomness in the nonlinear operations. Specifically, the presence of shares without proper refreshing allows leakage to arise from cross-domain operations such as AND gates. Figure 4.8(c) shows the t-statistic obtained with one million traces using the SHA-3 accelerator in the masked version with the Randomness Dispatcher activated (Trivium version). In this case, no leakage is detected ($|t| < 4.5$). Figure 4.9 shows the t-statistic obtained with 1M traces using Trivium (a) or Bivium (b) as the Randomness Dispatcher. No leakage is detected in either configuration.

We acknowledge that the TVLA experiments were performed with up to 1 million traces. This number of traces is sufficient to reliably detect first-order leakage in our implementation, as widely accepted in the side-channel community [GGJR+11, SM16]. While attackers could, in principle, collect more traces, the results provide strong evidence of first-order resistance, and extending the experiments to larger trace counts is expected to confirm the trends observed.

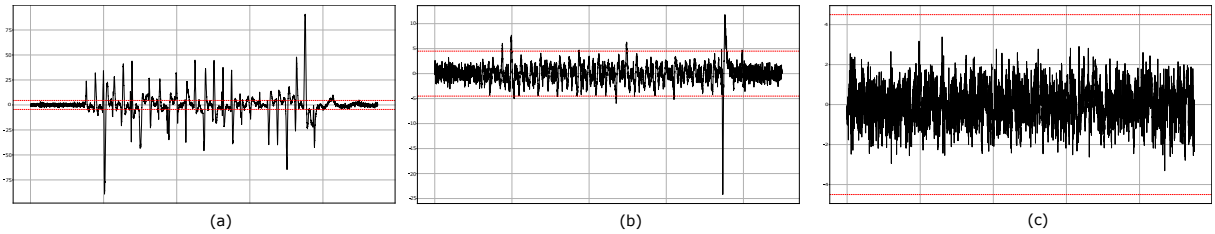


Figure 4.8 Leakage assessment of the masked SHA-3 accelerator during one round of the Keccak function. Horizontal red lines correspond to the threshold of $|t| = 4.5$ above which we are in the presence of information leakage. (a) t-statistic obtained with 20k traces with the SHA-3 accelerator used in the unmasked mode of operation. (b) t-statistic obtained with 20k traces with the SHA-3 accelerator used in masked version with the Randomness Dispatcher OFF. (c) t-statistic obtained with 1M traces with the SHA-3 accelerator used in masked version with Randomness Dispatcher (Trivium version) ON.

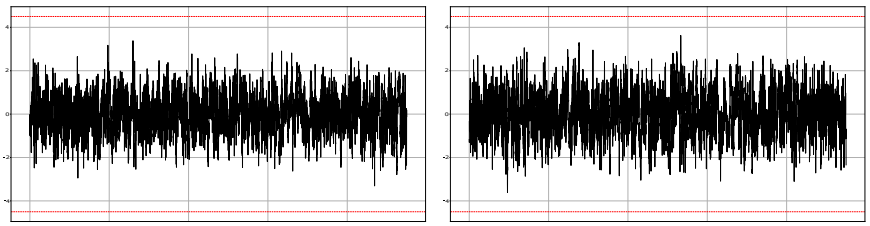


Figure 4.9 In (a) the t-statistic obtained with 1M traces with the SHA-3 accelerator used in masked version with Randomness Dispatcher (Trivium version) ON. In (b) the t-statistic obtained with 1M traces with the SHA-3 accelerator used in masked version with Randomness Dispatcher (Bivium version) ON.

4.4.3 Masked ML-KEM performance with masked SHA-3 Accelerator on a RISC-V SoC

The proposed hardware accelerator for SHA-3 and the Randomness Dispatcher have been deployed in the SoC described in Subsection 4.3.3, which has been used for the execution of a first-order masked implementation of ML-KEM. We considered the masked implementation from [HKL$^+$22] as a baseline for our masked ML-KEM.

As a first step, we updated the code according to the latest NIST standard for ML-KEM (FIPS 203 [NIS24b]). The implementation from [HKL$^+$22] includes dedicated assembly instructions on ARM processors for several functions: NTT, masking conversions, binomial sampling, and others. Consequently, we rewrote all such functions in clean C code to be compatible with our CV32E40P RISC-V processor. Table 4.2 reports the execution time of the masked KEM.Decaps running on the target platform. We report the clock cycles taken by the full-SW implementation running on the CV32E40P processor (second column of Table 4.2) and the SW implementation accelerated through the masked SHA-3 accelerator (third column of Table 4.2). We hierarchically divided the contributions of the subroutines included in KEM.Decaps by considering the hash functions G() and J(), PKE.Decrypt, and PKE.Encrypt. In addition, we extended this analysis to PKE.Decrypt and PKE.Encrypt. The full-SW implementation of KEM.Decaps takes approximately 6.8 million clock cycles, with around 62% of this time consumed by the SHA-3 functions, as shown in Figure 4.10. In particular, the J() function alone

Table 4.2 Execution Time (in Clock Cycles) of the Masked `KEM.Decaps` Function Using Full Software and a SHA-3 Hardware Accelerator

Operation	Full-SW	SHA-3 Accel.	Gain
ML-KEM.decaps	6,868,657	3,164,677	**2.1**×
PKE.Decrypt	427,786	427,786	–
G()	179,540	678	**265**×
PKE.Encrypt	4,684,373	2,729,732	**1.7**×
J()	1,576,958	6,481	**243**×
PKE.Decrypt	427,786	427,786	–
masked_poly_tomsg	67,833	67,833	–
poly arithmetic	350,589	350,589	–
decompression	9,364	9,364	–
PKE.Encrypt	4,684,373	2,729,732	**1.7**×
CBD	572,362	572,362	–
poly_frommsg	546,009	546,009	–
hash	1,975,347	20,706	**95**×
poly arithmetic	1,244,689	1,244,689	–
compare	345,966	345,966	–

takes over 1.5 million clock cycles, while the combined contribution of all SHA-3 functions within PKE.Encrypt accounts for nearly 2 million clock cycles. Thanks to hardware acceleration, the execution time of the J() function is reduced by a factor of 243, that of the G() function by a factor of 264, and the total contribution of the hash functions in PKE.Encrypt is reduced by a factor of 95. Overall, KEM.Decaps is accelerated by a factor of 2.

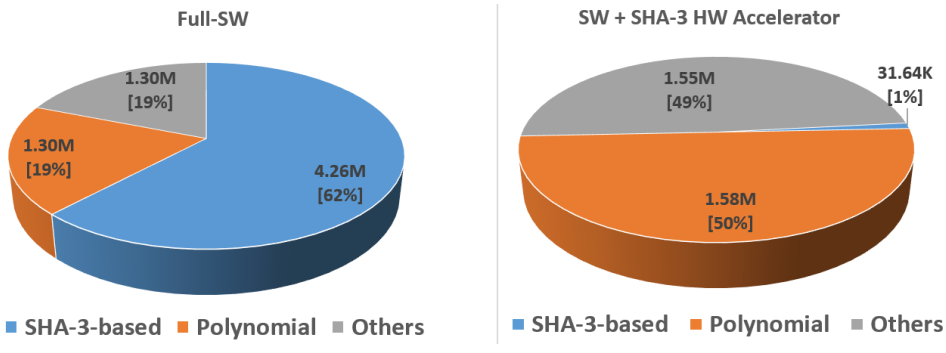


Figure 4.10 On the left side, the cycle count and percentage of the full-SW implementation of KEM.Decaps are divided by family of primitives (SHA-3 functions, polynomial computation, and other operations). On the right side, the SW implementation accelerated by the proposed masked SHA-3 accelerator is shown.

Most notably, the share of SHA-3 functions in the total runtime drops from 62% (in the full-software implementation) to just 1% when using the proposed SHA-3 hardware accelerator.

4.4.4 Comparison with the State of the Art

This section presents a deployable SHA-3 accelerator for post-quantum cryptography featuring a masking-based countermeasure against physical attacks. It supports both masked and unmasked modes, efficiently handles randomness generation through a dedicated hardware module, and is evaluated within a real-world use case. Specifically, it assesses the cost of randomness, measures the performance gain in masked ML-KEM decapsulation, and validates security through TVLA. A comprehensive comparison with related work is challenging, as the literature offers limited results on fully masked, deployable SHA-3 accelerators with integrated randomness generation and system-level evaluation. Compared to [FVBR+22], which proposes a set of hardware accelerators and instruction set extensions targeting post-quantum cryptographic workloads, including support for NTT operations and masking conversions, this section focuses specifically on the SHA-3 functions, which are the dominant computational component in masked ML-KEM decapsulation. While [FVBR+22] addresses a broader scope by accelerating other arithmetic primitives, their SHA-3 implementation differs significantly from the one presented here: we propose a one-round-per-cycle masked SHA-3 accelerator designed for high throughput and full compliance with the FIPS 202 standard in both masked and unmasked modes. This architectural choice enables substantially higher performance. However, a direct numerical comparison is not possible, as [FVBR+22] does not provide detailed latency or cycle-level results for their SHA-3 implementation.

A comparison with [HKL+22], which presents a full software implementation of masked Kyber on an ARM Cortex-M4, highlights important differences in both scope and methodology. Their implementation achieves faster performance than the software baseline developed in this study, primarily because it targets an earlier draft of the ML-KEM specification, whereas this section adopts the final standardized version (FIPS 203). Additionally, [HKL+22] includes heavily optimized assembly routines for computationally intensive components such as the NTT and masking conversions. In contrast, the implementation proposed in this section is entirely written in portable C to ensure compatibility with a generic 32-bit RISC-V core and focuses specifically on hardware acceleration of SHA-3 functions, which are the dominant runtime component in masked ML-KEM decapsulation. The acceleration of NTT and masking conversions is explicitly out of the scope of this section, which concentrates instead on analyzing the cost and performance trade-offs of SHA-3 masking and randomness generation. Due to these differences in algorithm version, optimization level, and system targets, a direct performance comparison is not meaningful, but the two works are complementary: [HKL+22] advances optimized software for lattice arithmetic, while this section focuses on secure and efficient hardware acceleration of the cryptographic hash layer.

Other relevant works include [KNAH22] and [JGCS24], which explore full hardware implementations of masked Kyber with side-channel protections. However, a direct comparison with this research is not possible due to the different focus of those papers: they target lattice-based

arithmetic and the masking of polynomial operations, while this section concentrates specifically on the cost, performance, and security of masked SHA-3 acceleration.

4.5 Summary

This chapter has presented the complete design, microarchitectural implementation, and system-level evaluation of a fully deployable, first-order masked SHA-3 hardware accelerator optimized for post-quantum cryptography. Designed specifically for integration into 32-bit systems, the proposed accelerator provides full architectural compliance with the FIPS 202 standard, supporting all hash functions and eXtendable-Output Functions (XOFs) in both masked and unmasked operational modes. The core datapath executes a round-per-cycle implementation of the Keccak permutation based on Domain-Oriented Masking (DOM) to protect sensitive variables across two independent shares. To prevent the padding bytes from canceling each other out when the shares are eventually recombined, the Input Management Module applies the byte-wise padding to only one of the two shares. If identical padding were applied to both shares, the padding data would completely disappear during the bitwise XOR recombination phase.

To satisfy the intensive entropy requirements of the non-linear DOM operations, which demand up to 1600 bits of fresh randomness per clock cycle to protect intermediate states against glitch-induced leakage, this chapter has introduced an integrated hardware module termed the Randomness Dispatcher. Operating under a four-state finite state machine (`Idle`, `Initialization`, `Warm-up`, and `Running`), this module scales up the throughput of pseudo-random bit generation by deploying 25 parallel instances of a 64-bit unrolled stream cipher. Depending on synthesis constraints, the module is fully configurable to implement either the ISO/IEC 29192-3 standardized Trivium cipher or its area-reduced variant, Bivium B, and features a software-transparent configuration layer that allows for on-demand reseeding directly via software without additional hardware.

The complete cryptographic ecosystem was validated at the system level by embedding both the masked SHA-3 accelerator and the Randomness Dispatcher into a complete RISC-V System-on-Chip (SoC). The platform is organized around an OBI-connected 32-bit CV32E40P processor core acting as the sole active bus master during runtime, thereby ensuring that no performance degradation occurs due to bus contention. As a realistic post-quantum use case, the hardware was evaluated by accelerating a first-order masked decapsulation function (`ML-KEM.Decaps`) fully updated to the final NIST FIPS 203 standard. To achieve compatibility with the target RISC-V pipeline, assembly routines from prior literature were entirely rewritten into portable, clean C code.

The experimental and synthesis results detailed within this chapter establish clear hardware trade-offs across multiple implementation targets:

- **Software Acceleration and Performance Gains:** In the full-software baseline implementation running on the CV32E40P core, the execution of `ML-KEM.Decaps` requires approximately 6.8 million clock cycles, with the underlying SHA-3 functions accounting for 62% of the total runtime overhead. Offloading these operations to the proposed hardware accelerator provides an overall system-level speedup factor of 2×, reducing the total decapsulation time to 3,164,677 clock cycles. Looking closely at individual subroutines, the hardware accelerator reduces the execution time of the $J()$ function by a factor of 243, the $G()$ function by a factor of 264, and the total contribution of the hash layers within `PKE.Encrypt` by a factor of 95. Consequently, the performance footprint of the cryptographic hashing operations drops from 62% of the total system runtime down to just 1%.

- **FPGA Synthesis Profiles:** When mapped onto an Artix-7 FPGA (Digilent Arty A7-100T board) via Vivado 2021.1 using default optimization strategies, the baseline unmasked accelerator operates at a maximum frequency of 166 MHz while occupying 6,990 LUTs and 4,706 FFs. The standalone masked accelerator achieves a maximum operating frequency of 163 MHz, demonstrating that the side-channel countermeasure does not degrade operating frequency on an FPGA. However, the masked core demands 18,805 LUTs and 9,168 FFs, representing a standalone resource penalty of 269% for LUTs and 194% for FFs. When incorporating the necessary randomness infrastructure, the Trivium-based dispatcher consumes an additional 11,966 LUTs and 7,403 FFs, while the Bivium-based dispatcher requires 9,201 LUTs and 4,663 FFs. Accounting for this indispensable infrastructure, the comprehensive real area overhead reaches approximately 440% for LUTs and 352% for FFs using Trivium, and 400% for LUTs and 293% for FFs using Bivium.

- **ASIC Design Space Exploration:** Synthesized using Synopsys Design Compiler on a standard cell 22nm FD-SOI technology, the SHA-3 accelerator achieves a maximum operational frequency of 1.76 GHz. At this peak frequency, the masked accelerator consumes 213 kGE, the Trivium-based dispatcher consumes 130 kGE, and the Bivium-based dispatcher requires 86 kGE. When optimized for a lower operating frequency of 735 MHz, the masked accelerator area decreases to 186 kGE, while the area of the Randomness Dispatcher remains unchanged. Given that the unmasked SHA-3 baseline ranges between 54 kGE and 64 kGE depending on the target constraints, the comprehensive ASIC masking overhead ranges from 300% (at 735 MHz with the Bivium dispatcher) to 500% (at 1.76 GHz with the Trivium dispatcher). The analysis notes that extending this implementation to second-order masking would increase the accelerator area by more than 2× and triple the randomness throughput requirements, making higher-order hardware masking highly impractical.

- **Side-Channel Security Validation:** The physical security margins of the hardware countermeasure were rigorously validated through electromagnetic Test Vector Leakage Assessment (TVLA) on the FPGA development platform, using an electromagnetic probe

and a Tektronix MSO 6 series oscilloscope for trace acquisition. Non-specific fixed-vs-random t-tests proved that while the unmasked accelerator configuration and the masked configuration with the Randomness Dispatcher deactivated (OFF) exhibit massive, immediate first-order information leakage ($|t| > 4.5$) due to un-refreshed cross-domain operations in the non-linear logic, activating the Randomness Dispatcher completely eliminates these vulnerabilities. Both the Trivium and Bivium configurations demonstrate robust first-order side-channel resistance, remaining securely bounded below the strict statistical threshold ($|t| < 4.5$) across one million physical execution traces.

In conclusion, this chapter highlights that while the computationally intensive FIPS 202 primitives can be successfully accelerated in a deployable, SCA-protected hardware design, the physical implementation costs are severe. Specifically, the silicon area expands drastically when side-channel vulnerabilities are addressed, primarily due to the high volume of pseudorandom bits required to sustain the masking countermeasures. The necessity of FIPS 202 primitives within LWE-based schemes, coupled with their heavy hardware area impact, justifies the exploration of novel optimizations to bypass these architectural constraints. Motivated by these findings, Chapter 5 introduces the AxLWE paradigm, proposing approximate computing to reduce the complexity of the error sampling chains in LWE. In this way, it is possible to minimize the overall demand for FIPS 202 primitive execution while providing more area-efficient post-quantum architectures.

Chapter 5

Approximate Computing as Controlled Error Source for LWE-based Schemes

This chapter explores the deployment of Approximate Computing (AxC) as a controlled, hardware-driven error source to fundamentally restructure the execution of lattice-based cryptographic primitives. Energy efficiency represents a critical bottleneck for modern hardware design, particularly within resource constrained Internet of Things (IoT) end-nodes where traditional silicon scaling under Moore's law is hitting its physical limits. To manage these processing overheads, the paradigm of approximate computing has emerged as a viable structural alternative, exploiting the fact that many digital workloads possess an intrinsic resilience to computational errors and can accept "good enough" outputs in exchange for substantial resource savings. This architectural trade-off is uniquely suited for lattice-based Post-Quantum Cryptography (PQC), specifically for schemes built upon the Learning With Errors (LWE) problem. Because LWE architectures are fundamentally probabilistic by design, intentionally introducing an error term during matrix computations that is subsequently eliminated via threshold decoding or rounding, they present a natural, mathematically bounded error budget. This chapter investigates the intentional exploitation of this error margin by deploying hardware-level approximations directly into the arithmetic datapath, transitioning from traditional explicit error generation to a streamlined, implicit error embedding methodology.

The core idea guiding this exploration is the proposed Approximate Learning With Errors (AxLWE) paradigm, which bridges the gap between physical resource optimization and cryptographic compliance. Traditional baseline implementations rely heavily on an explicit error sampling chain, mandating dedicated hardware units like Cumulative Distribution Function (CDF)-based Gaussian samplers and independent memory blocks to pre-compute and store standalone error vectors. While prior literature approaches have attempted to replace this explicit generation by utilizing analog Voltage-Frequency Over-Scaling (VOS) with custom variable delay lines, they suffer from extreme technology dependencies and require costly per-chip calibration routines to counter manufacturing process variations. In contrast, our work investigates a

purely digital alternative that integrates directly into standard ASIC design flows. By embedding digital approximate adders (AxADD) directly within the intermediate accumulation stages of the matrix-vector inner products, the hardware perturbations are introduced natively during computation. This framework explores whether the deterministic, input-dependent errors of digital approximate operators can faithfully mirror target discrete Gaussian distributions when combined with natural hardware diffusion and dynamic index randomization via an independent entropy source. Ultimately, this exploration aims to completely excise the explicit sampling logic and memory footprint, highlighting area and power efficiencies without compromising the theoretical security tier of the underlying cryptosystem.

5.1 Introduction to Approximate Computing

Energy efficiency represents a dominant driving force across the modern computer industry, impacting architectures ranging from supercomputers and clouds to small portable personal electronics and sensors. Projections indicate that by the year 2040, the unconstrained energy demand of computing systems will exceed total global energy production unless major efficiency breakthroughs are achieved [Sem20]. This compounding energy crisis is visible within mobile broadband networks and terminals, and it is further accelerated by the rapid expansion of the Internet of Things (IoT), which is expected to connect up to 50 billion devices. Analyzing the root causes of this energy consumption reveals that physical energy costs depend heavily on the bit-width of the manipulated variables, the mathematical type of operations being executed (where floating-point operations are more expensive than fixed-point or integer alternatives), and the physical distance of the component where the target data is stored. Accessing external main memory (DRAM) represents the worst-case operating condition, incurring an energy cost that can be up to three orders of magnitude higher than standard arithmetic operations, while increasing data access and execution times.

These consumption characteristics are intrinsically linked to the underlying Complementary Metal-Oxide-Semiconductor (CMOS) technology. While classical technology scaling under Moore's law historically delivered massive performance gains alongside a dramatic reduction in cost per transistor and energy per operation, modern silicon deployment is reaching its definitive physical limits. For technology nodes lower than 10 nm, several manufacturing and physical challenges arise, including high leakage currents that escalate static power consumption, diminished performance gains, complex manufacturing processes leading to low yield, complex testing procedures, and extreme mask fabrication costs. Furthermore, deep-submicron devices are increasingly less reliable, exhibiting a higher probability of runtime errors during their operational lifetimes. Mitigating these reliability concerns traditionally requires fault-tolerant mechanisms that introduce extra silicon area, power, and timing overheads, alongside the enforcement of design guard bands, such as higher supply voltages or lowered clock frequencies, to ensure correct hardware behavior under process variations.

To manage these processing demands and improve overall energy efficiency, the paradigm of Approximate Computing (AxC) [BMS22] has emerged as a viable structural alternative. AxC addresses the problems of maintaining system performance and managing runtime errors within an acceptable resource overhead envelope. The paradigm is based on the foundational observation that while performing fully exact computation requires a high concentration of hardware resources, allowing selective approximations or occasional specification violations can provide massive efficiency advantages. In practice, AxC exploits the gap between the precise output quality required by the application or end user and the strict computational precision provided by the underlying computing system. A wide variety of digital workloads display an intrinsic resilience to computational errors or noisy inputs, maintaining the capability to provide satisfactory or "good enough" outputs. For example, modifying an algorithm to selectively skip or remove a portion of its computations directly translates to a reduced number of accesses to main memory. By minimizing memory data transfers, significantly less energy is consumed and higher hardware performance is achieved, presenting a non-linear trade-off between accuracy degradation and cost reduction across the approximate computing design space.

5.1.1 Customizing Number Representation and Precision

The first category of approximate computing techniques focuses directly on the abstraction layer of number representation. Implementing approximations at this level addresses the fundamental root causes of hardware energy consumption, where physical costs scale with the size of the manipulated data. Specifically, a higher bit-width inherently increases the energy required to manipulate the data. Furthermore, the specific types of operations executed on a given dataset dictate energy demands, as floating-point operations are significantly more expensive than fixed-point or integer operations. The resource footprint of memory elements and register files is heavily dependent on this word-length selection. Consequently, data width choices directly impact overall performance, as both data access and execution times increase alongside the width of the data representation. By customizing the precision and formatting of numbers, designers can minimize the energy dissipated during arithmetic manipulation and data movement across wires or memory structures.

5.1.2 Hardware-Level Approximations

Hardware-level approximations modify the physical macro-layer of the computing system, which is divided into three distinct sub-layers: technology, circuit, and architecture. The technology layer defines the physical properties and the specific semiconductor technology utilized to construct the underlying hardware components, such as CMOS transistors. Moving up the stack, the circuit layer specifies the basic components of the hardware in terms of discrete logic and memory elements, including individual logic gates and flip-flops. In addition, the architecture layer specifies the high-level hardware components and their structural interconnections, en-

compassing elements such as central processing units (CPUs), cache memory systems, and input/output (I/O) systems. Approximations across these hardware layers help address the limits of modern CMOS technology at nodes lower than 10 nm, where engineers face high leakage currents, static power consumption, and reduced performance gains. Instead of using expensive fault-tolerant mechanisms or conservative guard bands like higher supply voltages and lower clock frequencies to handle process variations and runtime errors, hardware-level approximate computing allows selective specification violations to achieve substantial improvements in area, power, and energy efficiency.

5.1.3 Software-Level Approximations

Software-level approximations operate within the software macro-layer of the computing stack, targeting the core systems software responsible for program execution and translation. This macro-layer is further divided into operating systems (OS), application programming interfaces (APIs), and compilers. The OS and APIs specify the primary operating system components and the complete set of application programming interfaces, controlling vital system operations such as process scheduler execution and file system access. Working in tandem with these systems, the compiler layer is responsible for translating a software specification from high-level programming languages down to low-level machine instructions, such as assembly language. Software-level approximation techniques introduce flexibility into these translation and management layers, allowing the system to dynamically or statically alter execution parameters. By developing specialized compilers and runtime environments, software-level approximations can actively manage accuracy budgets and exploit hardware layer capabilities across diverse computing frameworks.

5.1.4 Application-Level Approximations

The final abstraction layer for approximate computing techniques is the application layer, which consists of the user applications running directly on the computing system. Approximations at this level exploit the intrinsic resilience of certain application types to computational errors or noisy inputs. These resilient applications maintain the capability to provide satisfactory, "good enough" outputs even when errors occur during execution. For example, a software implementation can be modified to selectively skip or remove a certain amount of its baseline instructions and computations, such as skipping 40% to 80% of instructions in an image filter. While this can cause minor localized artifacts like wrong pixels, the overall output remains completely recognizable to the user due to human perceptual limitations. The direct benefit of skipping these instructions is a reduced number of accesses to main memory to retrieve instructions and data operands, which consumes less energy and enables higher performance. Application-level approximations can be applied to a wide range of emerging real-world fields,

spanning from artificial intelligence, machine learning, and deep learning to high-performance computing, cyber-security, digital circuit testing, and safety-critical or mission-critical systems.

5.2 Related Work on Approximate Lattice Cryptography

The application of Approximate Computing (AxC) to Post-Quantum Cryptography (PQC) is an emerging research direction that primarily exploits the inherent error tolerance of lattice-based primitives. Cryptographic frameworks built on the Learning With Errors (LWE) problem and its algebraic variant, Ring-LWE (RLWE), are fundamentally probabilistic by design. During decryption, these schemes execute threshold decoding or rounding operations to eliminate an error vector introduced intentionally and recover the original plaintext message. Because the underlying mathematical structures naturally accommodate a specific error budget before decryption failures occur, they present an ideal opportunity for hardware-level approximation. By intentionally introducing controlled algorithmic or microarchitectural errors into the datapath, designers can exchange excess error margins for reductions in silicon area, power consumption, and execution latency, which is beneficial for resource-constrained applications.

The foundational work bridging lattice-based cryptography and approximate computing was introduced by Bian et al. under the Decrypting Learning with Errors (DWE) methodology [BHS18]. The authors recognized that because the LWE decryption process in these cryptosystems is inherently probabilistic and approximate by design, the system can naturally tolerate a certain amount of hardware-induced errors without compromising its overall operational correctness. To leverage this characteristic, they applied hardware-level approximations to the decryption function's most computationally intensive component, which heavily relies on dot product calculations. By utilizing an approximate multiplier known as the Dynamic Range Unbiased Multiplier (DRUM) instead of exact arithmetic units, the implementation introduces a suitable level of calculation error to significantly simplify the underlying circuitry. When evaluated on a standard public-key encryption scheme, this architectural shift successfully reduced silicon area and power consumption for the multiplier component while simultaneously increasing processing speed. These physical efficiency gains scaled more significantly when applied to a highly complex Fully Homomorphic Encryption (FHE) architecture, demonstrating that introducing controlled hardware errors is a viable strategy for optimizing lattice-based cryptographic workloads.

Expanding upon this paradigm, Kundi et al. shifted the focus toward addressing the primary computational bottleneck in Ring-LWE cryptosystems, which is the modular polynomial multiplication process [KBK$^+$20]. Recognizing that this operation often limits the practical realization of these cryptosystems on resource-constrained embedded and IoT devices, they proposed an approximate modular multiplier design denoted as AxMM. This hardware optimization leverages the inherent error-tolerant structure of the underlying cryptographic scheme to trade a permissible amount of accuracy for significant performance returns. Rather than utilizing

traditional exact arithmetic components, the architecture combines an approximate multiplication mechanism with an approximate modular reduction stage. When implemented, this high-level design strategy successfully mitigates resource overheads, delivering substantial reductions in both silicon area and power consumption while simultaneously increasing operational speed compared to conventional exact modular multipliers.

To provide a systematic approach to hardware optimization, Kundi et al. subsequently formalized a multi-level approximate framework for Ring-LWE (RLWE) architectures, known as AxRLWE, designed specifically to meet the strict resource budgets of lightweight IoT devices [KBW+21]. This co-processor architecture introduces a configurable approach that allows designers to easily trade off hardware resource savings against the resulting changes in security strength. A central architectural feature of this work is the shift away from complex, traditional error sampling methods toward a more compact and hardware-friendly binomial sampling approach. By utilizing this simplified sampling distribution alongside approximate polynomial multiplication techniques, the design achieves substantial reductions in both physical area and operational energy consumption. This multi-level optimization strategy enables the post-quantum cryptographic core to better fit within the highly restrictive power and area constraints of small end-node devices, making it a possible alternative to existing lightweight public-key processors.

Recognizing that aggressive bit truncation introduces a strict penalty on cryptographic strength, O'Connor et al. provided rigorous security estimates and failure rate characterizations for the approximate R-LWE landscape. Focusing directly on the error truncation mechanisms utilized in the AxRLWE framework, the authors leveraged the automated LWE Estimator tool [APS15] to map out the exact degradation of the lattice under modern adversarial vectors, including quantum sieving algorithms. By evaluating these truncation effects across two distinct medium-level security R-LWE parameter sets, the work quantified the effective new standard deviations (σ_{new}) and calculated the precise mathematical upper bounds for the resulting decryption failure rates. Their rigorous statistical profiling concluded that a 2-bit truncation level represents the optimal Pareto-efficient operating point across the design space, successfully maximizing power and memory efficiency while avoiding catastrophic security drops and keeping decryption failures securely within safe architectural boundaries.

While these prior works demonstrate the significant advantages of applying AxC to optimize the performance and footprint of post-quantum primitives, they do not apply AxC to the general LWE problem and still rely on the generation of the error through a Gaussian sampler.

5.3 Replacing Gaussian Error Generation via VOS

Other works have attempted to go further and propose approximation techniques aimed at replacing the explicit Gaussian error generation. The first proposal of this kind was the Learning Parity with Physical Noise (LPPN) by Kamel et al. [K+20]. This is a hardware-focused variant

of the Learning Parity with Noise (LPN) problem [BKW03]. LPN is a lightweight instance of an LWE problem defined over the binary field $\mathbb{Z}_2$. Referring to Figure 5.1, where k represents the secret key and x the input data, the inner product is computed such that $y = \langle x, k \rangle$. Given that x and k are binary vectors, y is a 1-bit value. In the standard LPN formulation, y is perturbed by a 1-bit error e. Traditionally, this error bit is generated by a hardware Random Number Generator (RNG), followed by a Bernoulli sampler that filters its output to ensure $e = 1$ with a specific *error probability* ε.

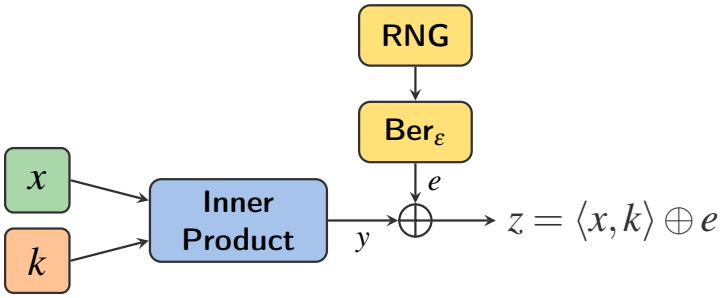


Figure 5.1 Computational pipeline for the Learning Parity with Noise (LPN) sample generation framework.

Unlike the standard formulation that relies on an external error sampling chain, LPPN introduces the error directly through the inexact computation of the inner product [K+20]. As illustrated in Figure 5.2, this intrinsic computational inaccuracy replaces the need for external error addition, yielding an output expressed as:

$$\langle x, k \rangle_\varepsilon = z \tag{5.1}$$

By transforming the standard LPN problem into a physical framework, LPPN fundamentally shifts the security foundation from a purely mathematical assumption to a physical one. In traditional LPN architectures, two independent hardware components are mandatory: one to execute the precise binary inner product and another to generate the error values using a Random Number Generator (RNG). This dual-component approach introduces severe physical vulnerabilities, as a single side-channel probe targeting the explicit output of the RNG can completely compromise the security of the protocol. LPPN elegantly circumvents this vulnerability by eliminating the explicit generation, calculation, and storage of Bernoulli noise. Instead, the noise is implicitly embedded within the physical function of the inner product computation itself, preventing straightforward side-channel probing attacks on the randomness source while providing natural resistance features against fault injection attempts. Furthermore, LPPN expands the microarchitectural design space by enabling implementations to exploit both deterministic and probabilistic physical phenomena within microelectronic devices. While standard LPN implementations can only leverage probabilistic noise via an external RNG, LPPN can harness deterministic physical characteristics, such as circuit glitches and data-dependent timing propagation delays, alongside purely probabilistic noise like thermal transistor variations or clock jitter. For this approach to be cryptographically viable, the hardware's internal error must be

statistically controllable to maintain a stable error probability ε. The authors utilized Voltage-Frequency Over-Scaling (VOS) techniques to induce controllable errors. While voltage and frequency scaling are traditionally employed to balance power consumption and performance, they serve here as a controlled approximation mechanism. By intentionally operating a circuit beyond its safe limits, one can induce a controllable error probability ε. Crucially, this error probability must be regulated by an internal, autonomous block within the device architecture. This autonomous regulation ensures that an external adversary cannot manipulate or override the circuit's operational parameters, such as externally altering the supply voltage or clock frequency, to reduce the error rate or eliminate the noise to crack the secret key.

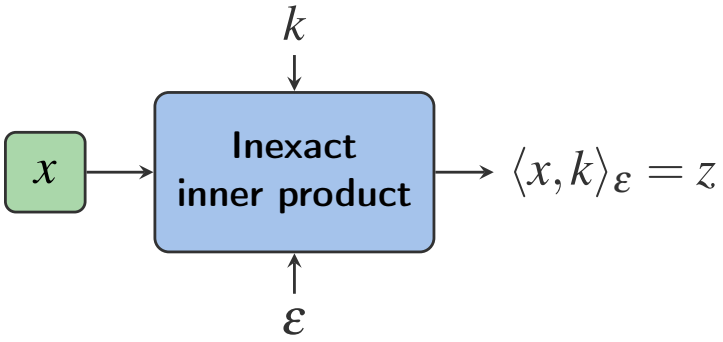


Figure 5.2 Computational layout of the Learning with Physical Parity Noise (LPPN) inexact inner product framework.

The first practical realization of the LPPN principle was published in [B+21] via an FPGA prototype designed to analyze physical imperfections. This implementation highlighted that practical hardware instantiations suffer from output data dependencies, where the physical error probabilities are inherently tied to the correct underlying computations. To evaluate the impact of this defect, the authors provided a formal security reduction establishing that LPPN with such data dependencies is polynomially equivalent to an asymmetric variant of the standard LPN problem, thereby proving that these imperfections do not significantly degrade the security of the cryptosystem. At the microarchitectural level, the prototype utilizes a Variable Delay Line (VDL) controlled by a Successive Approximation Register (SAR) calibration loop and a finite state machine to deliberately sample data at unstable states, while incorporating an on-chip voltage sensor to monitor circuit working conditions against environmental fluctuations. The LPPN concept was subsequently generalized to the larger-modulus LWE problem and presented as Learning With Physical Errors (LWPE) in [B+22]. The authors discovered that a direct extension of the LPPN bit-level error technique to the final processor output creates severe mathematical data dependencies because the physical errors alter individual bits rather than full algebraic field elements, distorting the target Centered Binomial Distribution (CBD). To mitigate this issue, an improved architecture was developed that relocates the incorrect sampling flip-flops away from the final output and places them inside an internal stage of the processor's arithmetic adder tree. By introducing errors exclusively through the least significant bits of multiple intermediate bytes, the design exploits natural hardware diffusion across the subsequent adder layers to dilute and

effectively cancel out the mathematical data dependencies while generating a reliably bounded CBD.

The importance of these foundational works lies in the fact that they demonstrate the feasibility of implementing hard learning schemes using approximate computations, which are able to entirely replace the external error generation traditionally performed through an explicit Gaussian or binomial sampler. However, the main limitation of these approaches is that they rely on voltage and clock skew tuning techniques managed by specialized variable delay lines. Because these structures depend on full-custom cell layouts or analog-like delay tuning configurations, they are inherently technology-dependent. Furthermore, these techniques are highly sensitive to manufacturing process variations, meaning that achieving a stable and uniform error rate requires a precise calibration phase for every individual chip. In the remainder of this chapter, we improve upon this methodology by relying on fully-digital approximate circuits. Unlike delay-tuned configurations, our proposed solution is entirely based on a standard ASIC digital design flow, ensuring portability across any technology node without the need for custom delay macros, specialized analog sensors, or per-device calibration.

5.4 From Explicit Error Sampling to Embedded Errors with AxLWE

The traditional formulation of lattice-based cryptography relies on an explicit error sampling chain to produce the necessary error distributions. A high-level view of the traditional, explicit error sampling architecture is represented in Figure 5.3. In standard implementations, once these error values are generated via a dedicated pseudo or true random number source, they must be stored and subsequently combined with the outputs of exact matrix-vector multiplications, requiring extra registers and further addition stages. While state-of-the-art physical learning paradigms like LPN, Learning Parity with Physical Noise (LPPN), and LWPE demonstrate the feasibility of removing these external sampling chains by embedding errors directly through inexact computation, their reliance on analog VOS and delay-line skews introduces severe technology dependencies and requires per-device calibration. To overcome these limitations, we introduce the Approximate Learning With Errors (AxLWE) paradigm, which achieves embedded error introduction in a purely digital manner, fully compatible with standard ASIC digital design flows.

Mathematically, the standard, exact inner product computed during the $\mathbf{b} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e}$ operation can be formulated as follows:

$$b_i = \sum_{k=0}^{n-1} a_{ik} s_k \underbrace{+ e_i}_{\text{error term}} = a_{i0} s_0 + \cdots + a_{i,n-1} s_{n-1} + e_i \tag{5.2}$$

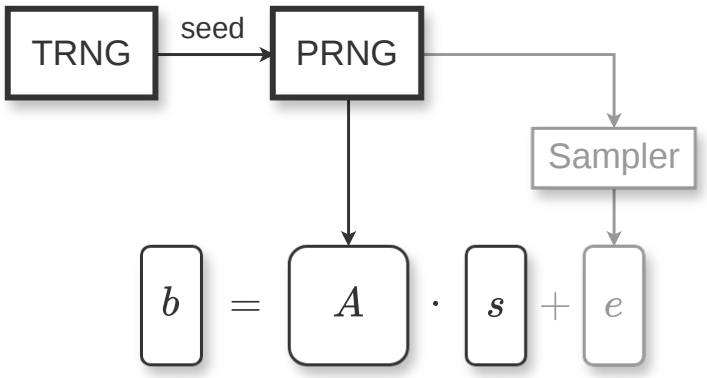


Figure 5.3 Inner matrix product generation chain.

where the summation represents the inner product of the i-th row of matrix $\mathbf{A}$ and the secret vector $\mathbf{s}$, followed by the separate addition of the explicitly sampled error term e_i to compute the i-th element of vector $\mathbf{b}$.

By employing approximate digital arithmetic operators directly within the functional logic of the matrix multiplication, the required error distribution can instead be produced natively during computation. Removing this explicit error addition step and the accompanying external generation chain, represented in gray in Figure 5.3, leads to a substantial simplification and resource reduction of the hardware layout. Figure 5.4 reports a high-level view of this reduced generation chain utilizing digital approximate additions.

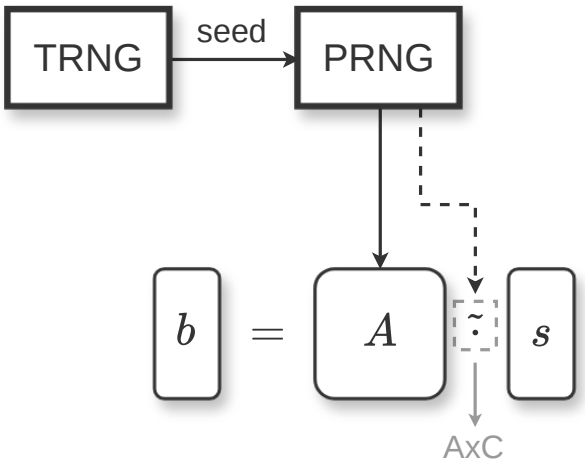


Figure 5.4 Reduced inner matrix product generation chain with approximate additions.

Instead of adding a separately generated error vector as shown in Equation 5.2, AxLWE embeds the error directly into the arithmetic operations by integrating an approximate adder (AxADD) within the accumulation tree of the partial products. A core challenge in hardware approximation is managing the exploitable data dependencies between the inputs of the approximate circuit and the resulting error distribution. AxLWE mitigates these dependencies through a multi-layered digital strategy. First, by employing the AxADD at an internal, intermediate

stage of the inner product calculation, any local data dependencies are diluted and canceled out via natural hardware diffusion across the subsequent exact accumulation layers as highlighted in [B+22]. Additionally, we dynamically randomize the exact stage at which the AxADD is applied during the accumulation process for each element b_i. This runtime variation increases the error entropy by leveraging $\lceil \log_2(n) \rceil$ bits of digital randomness, represented by the dashed line in Figure 5.4.

Hence, the AxLWE matrix-vector product is defined as:

$$b_i = a_{i0}s_0 + \ldots \underbrace{+}_{\text{AxADD}} a_{i,n-1}s_{n-1} \approx \sum_{k=0}^{n-1} a_{ik}s_k \tag{5.3}$$

While this architectural choice maintains suitable computational and area efficiency, embedding the errors relies on characterizing the approximate adders in advance to select the most suitable operators as will be described in Section 5.5. The cryptographic viability of this method relies fundamentally on the additive property of the hardware error, specifically the invariance of the error distribution under addition. This property ensures that the statistical distribution of the hardware-introduced perturbation is preserved as it propagates to the final result, causing the computed value b_i to deviate from the exact result by a magnitude and shape identical to the error profile of the approximate operator itself. Because the elements of the public matrix $\mathbf{A}$ are uniformly distributed, they activate the internal logic gates of the digital approximate adder with equal probability, ensuring that the resulting hardware-induced perturbations faithfully follow the intrinsic error distribution of the chosen operator without introducing specific statistical biases. To fully evaluate the effects of this behavior, the following sections analyze the exact error distributions introduced by these approximate adders and perform a comprehensive analysis of the decryption failure rate (DFR) implications that this technique has on a real KEM.

5.5 Choosing a Suitable Approximate Operator

As presented in Section 5.1, the paradigm of Approximate Computing (AxC) revolves around the deliberate relaxation of strict Boolean equivalence in digital circuits to trade numerical precision for significant improvements in power consumption, silicon area, and processing latency [BMS22, HO13]. Unlike traditional design flows that treat any deviation from exact arithmetic as a structural failure, AxC leverages the fact that specific algorithms can yield acceptable results even when the underlying hardware is computationally imprecise.

A prominent framework for exploring these trade-offs is EvoApproxLib [M+17], an open-source repository designed to accelerate the hardware design process and support the fair benchmarking of circuit approximation methods. As the authors highlight, the evaluation of approximate arithmetic operators in prior literature was often hindered by practical challenges, such as the lack of standardized exact baselines, disparate technology libraries, and the absence of

open-source HDL descriptions evaluated under consistent post-synthesis benchmarks. To address these limitations, the repository introduces a comprehensive collection of ready-to-use hardware components across multiple precision levels (including 8-bit, 12-bit, and 16-bit operators), providing hundreds of non-dominated approximate adders and multipliers that populate a dense multi-dimensional Pareto front. In this multi-objective evolutionary space, a circuit configuration is classified as non-dominated if no other candidate implementation simultaneously outperforms it across all targeted optimization metrics, namely circuit delay, power consumption, and numerical error. These arithmetic blocks were systematically evolved via Cartesian Genetic Programming from an initial population of 13 conventional accurate adder architectures (including ripple-carry, carry-select, carry-lookahead, and tree adder topologies) alongside 6 conventional multiplier structures. In this chapter, the 16-bit approximate arithmetic units from the library are of particular interest for accelerating modular lattice arithmetic.

These components are synthesized via multi-objective Cartesian Genetic Programming (CGP) [HMV16] in combination with the NSGA-II sorting algorithm [DPAM02], an evolutionary methodology that treats candidate circuit representations as directed acyclic graphs mapped across a Cartesian grid of programmable nodes. The parallel search method operates by applying a point mutation operator to the circuit chromosomes to simultaneously minimize circuit delay, power consumption, and numerical error. To ensure immediate utility across different design abstractions, the framework provides complete Verilog, Matlab, and C source models for every component, allowing developers to integrate the blocks into both physical hardware design tools and software simulation pipelines. Furthermore, the library provides a thorough characterization of each circuit by detailing its exact performance across seven distinct error metrics, including error probability, mean absolute error, mean squared error, and worst-case absolute error, alongside standard physical circuit metrics like area, path delay, and power consumption. This multi-layered characterization enables a direct evaluation of the intrinsic deviations from exact arithmetic built into the circuit gates, offering highly reproducible, ready-to-use building blocks for error-resilient applications where microarchitectural hardware inaccuracies can be traded for substantial physical resource savings.

While AxC is traditionally applied in error-tolerant domains such as image processing or signal filtering, where small perturbations in output are often imperceptible [CCRR13], its suitability for cryptographic design relies on specific statistical properties, such as the Mean Error (μ), the Standard Deviation (σ), and the Maximum Absolute Error (E_{max}). Approximate circuits introduce a controlled but strictly deterministic perturbation of the results. Consequently, when these circuits are stimulated with uniformly random inputs, they generate an output error distribution that is intrinsically dependent on those inputs. This behavior is analogous to the approximate implementations of LWE described in Section 5.2 and Section 5.3, based on numerical truncation or physical hardware perturbations. In this work, we explore whether the deterministic and input-dependent errors provided by digital approximate circuits, such as the ones from the EvoApproxLib, can satisfy the requirements for providing an implementation

of the LWE problem with an implicit generation of the error term, thus without relying on a Gaussian sampler.

5.5.1 Approximate Adder Characterization Methodology

As previously discussed, the EvoApproxLib provides a diverse repository of approximate arithmetic circuits, each offering a distinct trade-off between efficiency and accuracy. While the library characterizes each operator using standard metrics such as Mean Absolute Error (MAE), Worst-Case Absolute Error (WCE), Mean Relative Error (MRE), and Error Probability (EP), these numerical aggregates are insufficient for the requirements of the AxLWE paradigm. Because the security and functional correctness of LWE-based schemes are intrinsically linked to the statistical profile of the error, a qualitative and quantitative understanding of the error distribution shape is necessary. To perform this analysis, the C-based behavioral models provided by the library are employed to extract the raw error data necessary for statistical characterization through functional simulation.

To ensure algorithmic correctness and prevent decryption failures, the error magnitude in LWE-based schemes must be rigorously constrained. The proposed methodology evaluates the suitability of approximate adders by comparing their intrinsic error profiles against a target discrete Gaussian distribution with a specific standard deviation σ and support values. The selection follows a two-stage approach, as illustrated in the methodology flowchart in Figure 5.5. Initially, the WCE of each candidate is compared against the target support values. This first filter ensures that the hardware-induced errors remain strictly bounded, preventing violations of LWE security assumptions. In the second stage, the statistical profile of the remaining adders is isolated through a software-based characterization environment. Within this characterization framework, as detailed in Listing A.1, the error extraction is carried out by exposing the selected approximate operator to a sequence of pseudo-randomly generated input operands. Over a defined number of experimental simulation trials (N_{EXP}), the program generates uniform 16-bit input pairs to serve as the arithmetic stimuli. For each iteration, an exact arithmetic baseline is calculated alongside the approximate output produced by the specific library component under test. The intrinsic error term is then isolated by computing the difference between this approximate sum and the exact reference. These individual arithmetic deviations are subsequently logged to a sample file and mapped onto a centered histogram array, providing the raw statistical data required to verify the shape and boundary support constraints of the hardware-induced error distribution. This is followed by an error characterization step to generate the empirical Probability Density Function (PDF) and Cumulative Distribution Function (CDF) profiles required for the final evaluation. To ensure a fair comparison, a reference dataset is generated using a standard algorithmic Gaussian sampler configured with the target σ. As implemented in the Python-based analysis detailed in Listing A.2, the script processes both the raw approximate error samples and the reference error dataset to calculate their respective discrete Probability Mass Functions (PMFs), the discrete equivalent of a continuous PDF profile. This processing enables

a rigorous statistical evaluation through the automated computation of multiple distance and goodness-of-fit metrics. Specifically, the Hellinger Distance (HD) is used to measure distribution overlap, while the Kullback-Leibler (KL) Divergence evaluates potential information loss and entropy degradation. To assess structural similarity, the discrete Kolmogorov-Smirnov (KS) test determines the maximum deviation between the respective empirical cumulative distribution functions. Additionally, the Chi-Squared test evaluates the statistical significance of the frequency differences between the observed hardware error counts and the expected Gaussian profile. For all these metrics, lower values indicate a higher degree of similarity to the target distribution and are thus desirable for our application. Finally, the script handles the visual mapping by rendering a comprehensive comparison plot, which overlays the approximate operator's error distribution as a bar chart against the reference Gaussian step plot, supplemented by an integrated data box displaying the exact calculated metrics for the final operator evaluation.

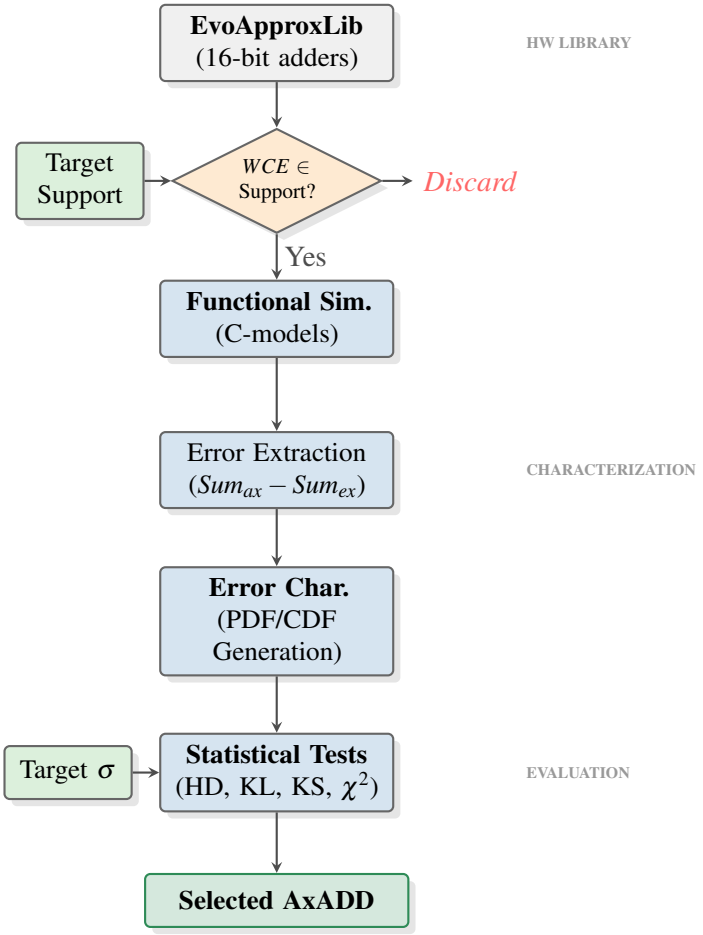


Figure 5.5 Adder selection methodology: integrating hardware constraints, behavioral simulation, and statistical mapping to identify the optimal AxADD.

5.6 Approximate FrodoKEM Case Study

This section explores the practical application of the AxLWE paradigm to the FrodoKEM [A+20] Key Encapsulation Mechanism described in Section 3.2.2. By replacing explicit Gaussian sampling with the intrinsic inaccuracies of approximate adders, we evaluate the feasibility of introducing errors with AxC in a real-world cryptographic context. FrodoKEM [A+20] is a practical implementation of the LWE problem, translating theoretical lattice-based foundations into a robust cryptographic scheme. This case study focuses on the FrodoKEM-640 parameter set.

The choice of FrodoKEM as our reference scheme is driven by several distinct technical and strategic factors. First, FrodoKEM directly extends the standard, unstructured LWE problem, ensuring that our microarchitectural modifications act on the foundational formulation of the problem rather than on structured ring or module variants. Second, although the scheme was eliminated from the standardization pool during the NIST Post-Quantum Cryptography Standardization process due to its larger public key sizes and performance overheads, often considered a redundant cost when compared to structured lattice alternatives, this very overhead stems from its conservative security design. By avoiding any additional algebraic ring structures, FrodoKEM eliminates potential vulnerabilities to structural lattice attacks, a consideration that has led the French National Agency for the Security of Information Systems (ANSSI) to maintain an active interest in the scheme and recommend its use as a secure, conservative option. Finally, from a validation perspective, the availability of official NIST Known Answer Tests (KATs) provides a highly reliable, deterministic framework to perform rigorous testing and verify our approximate implementation against established baselines.

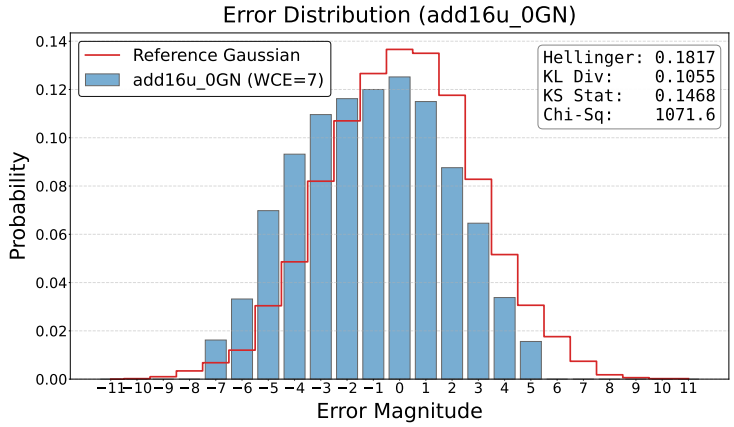
While the fundamental LWE problem is often defined using secret and error vectors, FrodoKEM employs a matrix-based formulation. Specifically, the secret matrices $(\mathbf{S}, \mathbf{S}')$ and error matrices $(\mathbf{E}, \mathbf{E}', \mathbf{E}'')$ are composed of multiple column vectors, where each column represents an independent LWE sample sharing the same public matrix $\mathbf{A}$. This matrix-centric design is the primary structural refinement over the original vector-based LWE problem, allowing the scheme to remain practical while prioritizing a conservative security approach. All the necessary pseudo-random bits are generated dynamically using hash functions compliant with the FIPS 202 standard [Nat15b]. The coefficients of the secret and error matrices are additionally passed to a discrete Gaussian sampler to achieve the required distribution.

Suitable Adder Selection

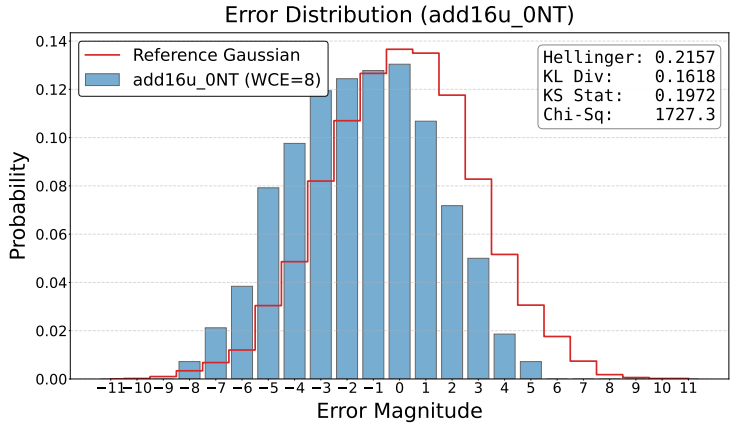
To select a proper approximate adder, we leverage the evaluation and characterization methodology described in Subsection 5.5.1. For this practical evaluation, we adopt $\sigma = 2.8$ as a reference benchmark with error support values strictly confined to the discrete interval $[-12, 12]$, as required by the selected FrodoKEM parameter scheme. The quantitative results for this screening

process are detailed in Table 5.1, which ranks the candidates using statistical distance and goodness-of-fit metrics to identify those most closely aligned with the target distribution, where lower values indicate a higher degree of similarity. To further substantiate the final component selection, Table 5.1 also reports Power, Performance, and Area (PPA) comparison data for all evaluated candidate approximate adders, obtained at their maximum operating frequency on a 22nm technology. While the primary statistical metrics (Hellinger, KL, KS, and Chi-Squared) act as the mandatory first filter to guarantee cryptographic reliability, PPA serves as a secondary selection criterion. Specifically, it is utilized to identify the Pareto-optimal choice among candidate designs that exhibit equivalent or highly competitive statistical distance metrics.

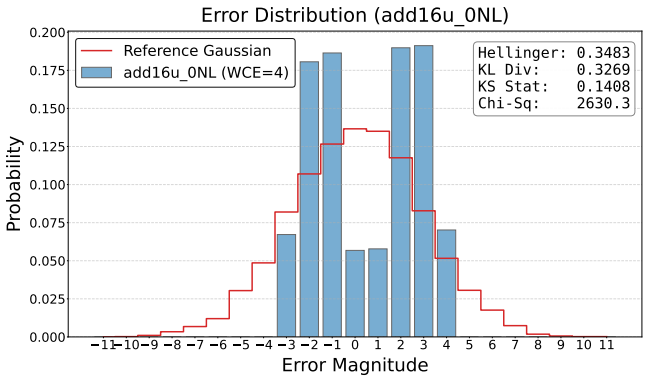
The statistical behavior of the candidate approximate adders is visually summarized in Figure 5.6, which is arranged vertically in a single-column layout and ordered by suitability from top to bottom across consecutive pages. In these plots, the x-axis represents the Error Magnitude, while the y-axis indicates the Probability of each error occurrence. The bars represent the intrinsic error distribution of the AxADD, which is labeled with its respective Worst-Case Absolute Error (WCE). The solid red line provides the reference discrete Gaussian distribution for direct comparison. These visualizations highlight the substantial diversity of error shapes within the EvoApproxLib, ranging from the centralized and quasi-symmetric profiles of the top candidates to the highly skewed and discrete distributions of the least suitable components. The degree of alignment between the hardware-induced error and the target error model for the selected `add16u_0GN` component is detailed in Figure 5.6a, which includes an annotation of its Hellinger distance (0.1817), KL divergence (0.1055), KS statistic (0.1468), and Chi-Squared value (1071.6). Because it achieves the highest statistical fidelity while maintaining an efficient physical footprint, this combined quantitative and PPA-driven correspondence confirms `add16u_0GN` as the most suitable choice for replacing explicit sampling units in LWE computations. Conversely, the least suitable operator, `add16u_0TA`, is shown in Figure 5.6i. This adder exhibits severe structural bias and significant statistical divergence, as confirmed by its high Hellinger distance (0.7560) and Chi-Squared value (22868.8), which renders it incapable of approximating the required cryptographic error profile.



(a) add16u_0GN

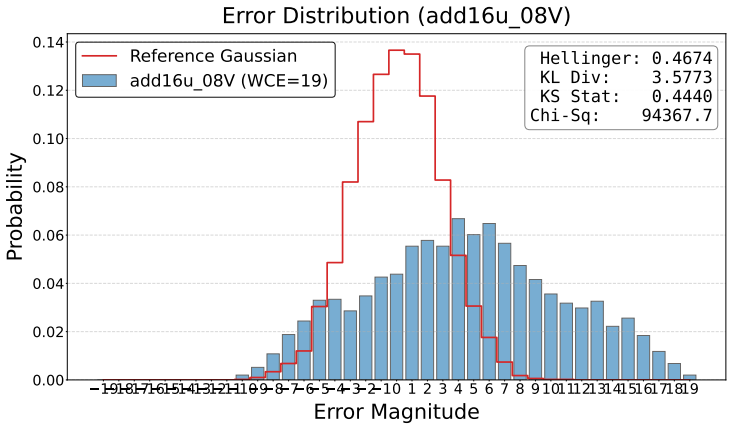


(b) add16u_0NT

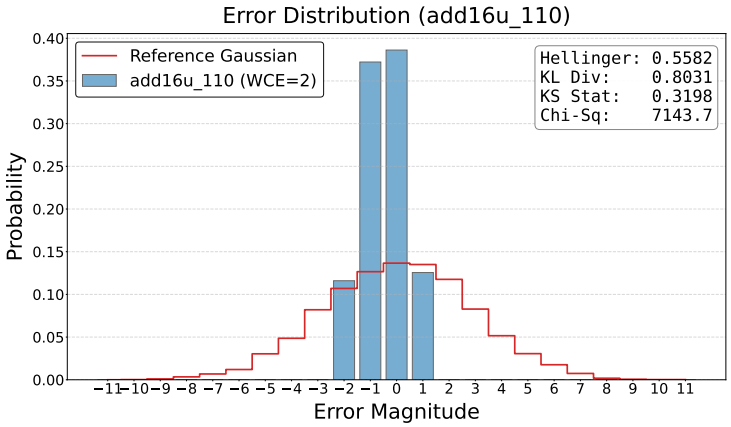


(c) add16u_0NL

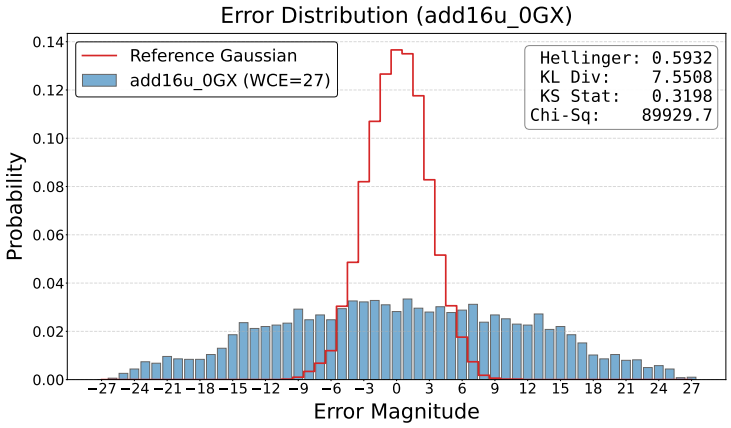
Figure 5.6 Comparative error distribution plots of evaluated EvoApproxLib adders, arranged vertically in a single-column layout and ordered by suitability from top to bottom (continued on next page). Each subfigure displays the Probability on the y-axis against the Error Magnitude on the x-axis. The most suitable candidates exhibit centralized and symmetric profiles, whereas lower-ranked designs display increasing levels of statistical divergence from the target baseline. The bar charts represent the empirical intrinsic error distribution of the hardware, labeled with the respective Worst Case Absolute Error (WCE), while the solid red lines represent the analytical Reference Gaussian distribution from the FrodoKEM parameters.



(d) add16u_08V

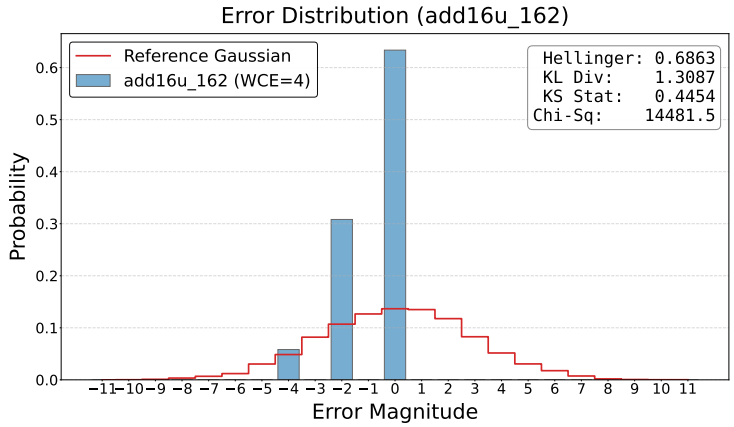


(e) add16u_110

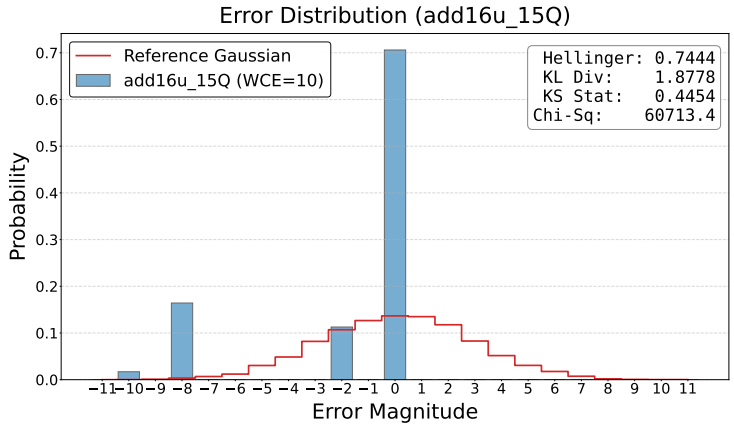


(f) add16u_0GX

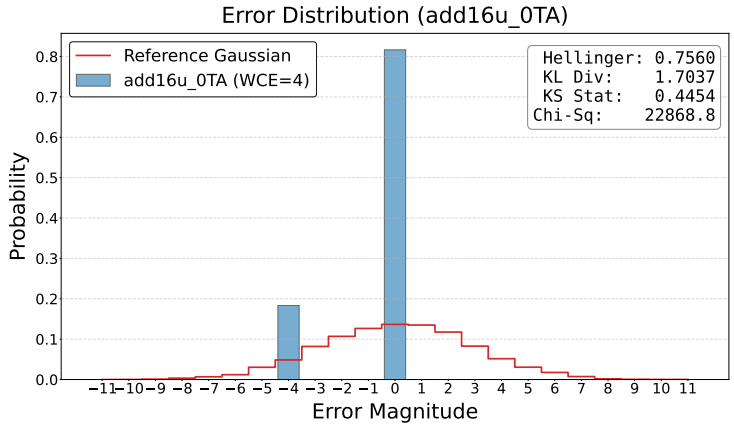
Figure 5.6 Comparative error distribution plots of evaluated EvoApproxLib adders (continued on next page).



(g) add16u_162



(h) add16u_15Q



(i) add16u_0TA

Figure 5.6 Comparative error distribution plots of evaluated EvoApproxLib adders (concluded). Annotation boxes provide the specific Hellinger distance, KL divergence, KS statistic, and Chi-Squared values computed for each operator.

Table 5.1 Statistical distance, goodness-of-fit tests, and 22nm PPA metrics at maximum frequency for candidate approximate adders against the FrodoKEM-640 reference distribution, ordered by suitability.

	Statistical Metrics				PPA Metrics (22nm)		
EvoApproxLib Adder	**Hellinger Distance**	**KL Divergence**	**Discrete KS**	**Chi-Squared**	**Max. Frequency (GHz)**	**Power (mW)**	**Area (μm^2)**
add16u_0GN	0.1817	0.1055	0.1468	1071.6	1.1824	11.4174	20.6336
add16u_0NT	0.2157	0.1618	0.1972	1727.3	1.1974	12.1308	20.5670
add16u_0NL	0.3483	0.3269	0.1408	2630.3	1.1491	12.3285	21.3658
add16u_08V	0.4674	3.5773	0.4440	94367.7	1.3103	11.3939	19.1693
add16u_110	0.5582	0.8031	0.3198	7143.7	3.0794	24.2242	49.3875
add16u_0GX	0.5932	7.5508	0.3198	89929.7	1.1074	11.1092	17.9712
add16u_162	0.6863	1.3087	0.4454	14481.5	1.1102	12.6875	22.6970
add16u_15Q	0.7444	1.8778	0.4454	60713.4	1.1859	12.8380	23.0963
add16u_0TA	0.7560	1.7037	0.4454	22868.8	1.0994	13.3121	23.8950

AxFrodoKEM Implementation

The proposed Approximate FrodoKEM implementation derives directly from the reference one [The26], removing the sections of code involving explicit error generation, placing approximate additions as explained in Section 5.4, and emulating their operation by means of the C function call provided in the EvoApproxLib library [M+17]. This architectural shift is realized across the core matrix arithmetic operations by restructuring inner-loop accumulations to natively embed the approximate hardware primitive. Specifically, in `frodo_mul_add_as_plus_e` (Listing B.1, B.2), the initial buffer assignment of the explicit error matrix is entirely stripped, and the accumulator invokes the `AxADD` macro to introduce error profiles directly during the final loop step. A parallel modification strategy is applied to `frodo_mul_add_sa_plus_e` (Listing B.3, B.4) and `frodo_mul_add_sb_plus_e` (Listing B.5, B.6), where the inner loop bounds are shortened to aggregate the terminal element product via the approximate operator. In this way, errors are introduced implicitly through approximate arithmetic, and the scheme completely bypasses the software-intensive explicit Gaussian sampling process.

This inline error introduction leads to a reduction in both data structure sizes and high-level cryptographic function calls, as systematically detailed in the compact unified diff of Listing B.7. At the scheme's top level, the procedures `crypto_kem_keypair`, `crypto_kem_enc`, and `crypto_kem_dec` are freed from allocating and tracking standalone error matrices ($\mathbf{E}, \mathbf{E}', \mathbf{E}''$). Memory structures dedicated to Gaussian sampled data are halved. For instance, the allocation size for array `S` is reduced from `2*PARAMS_N*PARAMS_NBAR` down to `PARAMS_N*PARAMS_NBAR` since the independent error matrix pointer `E` is no longer required, and pointers `Ep` and `Epp` are entirely removed from the `Sp` allocation footprint. Consequently, the input byte sizes requested from the pseudo-random `shake` extendable-output function are scaled down, and the discrete Gaussian sampling routine `frodo_sample_n` is omitted for all error components. This

optimization consequently eliminates the corresponding security cleanups via `clear_bytes`, removing substantial software processing overhead from the execution flow.

The approximate hardware characterization methodology presented above (Figure 5.5) using a target Gaussian distribution $\sigma = 2.8$, as required by the FrodoKEM-640 specifications, successfully identified the most suitable adder for these streamlined blocks. The selection results are reported in Table 5.1 from the best fitting adder (on top) to the least optimal solutions. The `add16u_0GN` approximate adder shows the smallest deviation from the reference Gaussian distribution in all statistical tests (HD $= 0.1817$, KL $= 0.1055$, KS $= 0.1468$, $\chi^2 = 1071.6$). Figure 5.7 reports a direct comparison between the `add16u_0GN` and the reference error distribution. As observed, the hardware error distribution shows a standard deviation of $\sigma = 2.7815$ and a mean of $\mu = -1.02$, while the reference Gaussian distribution exhibits $\sigma = 2.8381$ and $\mu = 0.07$. Furthermore, the observed shift of the mean toward a value of -1 is not a critical factor, as the error values remain well within the established distribution boundaries.

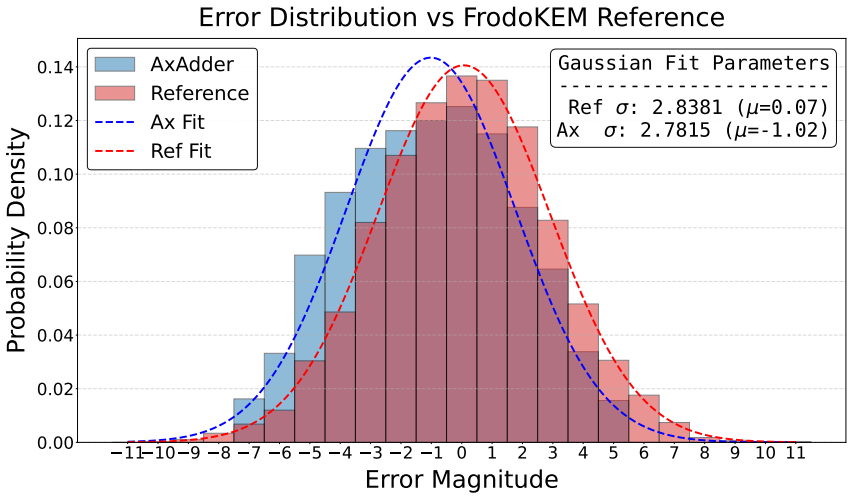


Figure 5.7 Gaussian fit comparison between the intrinsic error distribution of the `add16u_0GN` adder and the FrodoKEM-640 reference profile.

Decryption Failure Rate Estimation

To ensure this discrepancy, highlighted in Figure 5.7, does not compromise the security of the cryptosystem, we proceed with the Decryption Failure Rate (DFR) analysis of the error distribution introduced by the approximate adder. In FrodoKEM, because the error distribution behaves as a discrete Gaussian, the DFR can be modeled using the Gaussian tail bound for subgaussian distributions [Pei16].[1] This bound quantifies the probability that the cumulative decoding error e exceeds a specific critical threshold t, and is formulated as:

$$\Pr[|e| > t] \leq 2 \cdot \exp\left(-\pi \frac{t^2}{\sigma^2}\right) \tag{5.4}$$

[1] A subgaussian distribution is a probability distribution whose tail probabilities decay at least as fast as a canonical Gaussian distribution, a property satisfied by the discrete Gaussian distribution.

where t denotes the failure threshold, σ represents the standard deviation of the error distribution, and σ^2 is its variance.

To formalize this error threshold, the underlying symbol encoding of the cryptosystem must be evaluated, as structurally illustrated in Figure 5.8. The space of integers modulo q ($\mathbb{Z}_q$) is partitioned into intervals to embed bit patterns. For the FrodoKEM-640 parameter set, each symbol encodes $B = 2$ bits, mapping the binary pairs `00`, `01`, `10`, and `11` to ideal centers spaced at intervals of $q/4$. During decapsulation, successful decoding relies on rounding the received values to the nearest multiple of $q/4$. Consequently, a decryption failure is triggered if the absolute magnitude of the cumulative error component crosses the decision boundary, which lies exactly halfway between adjacent codewords. The maximum tolerable error threshold is therefore defined as $t = q/8$ (yielding an absolute value of 4096 given the modulus $q = 32768$). The reference standard deviation for this distribution is defined in the parameters as $\sigma = 2.8$.

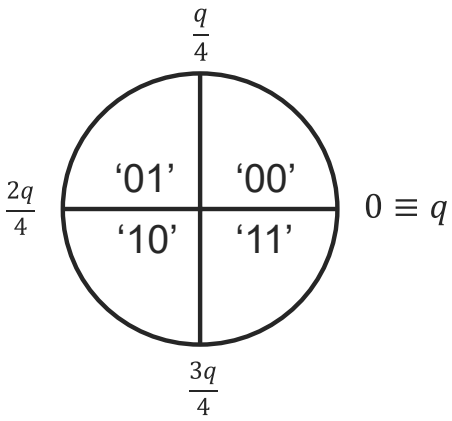


Figure 5.8 Modulo-q quadrant mapping for the FrodoKEM-640 bit encoding scheme ($B = 2$ bits per symbol). The space of integers modulo q ($\mathbb{Z}_q$) is partitioned into four distinct intervals centered around multiples of $q/4$ to encode the binary pairs `00`, `01`, `10`, and `11`.

From the FrodoKEM specification, the total accumulated error altering the decrypted message is defined as $E''' = S'E + E'' - E'S$. Given that the underlying error components follow a discrete Gaussian distribution with a reference parameter $\sigma = 2.8$, the total expanded variance across the matrix multiplication dimension ($N = 640$, for FrodoKEM-640) is computed as:

$$\sigma_{tot}^2 = 2N\sigma^4 + \sigma^2 = 2 \cdot 640 \cdot (2.8)^4 + (2.8)^2 = 78683.808 \tag{5.5}$$

Plugging this expanded variance alongside the correct decision boundary threshold $t = \frac{q}{8} = 4096$ (for a system modulus $q = 32768$) into the adjusted Gaussian tail bound allows for the formal estimation of the DFR:

$$DFR \approx 2 \cdot \exp\left(-\pi \cdot \frac{(\frac{q}{8})^2}{\sigma_{tot}^2}\right) = 2^{\left(1 - \frac{\pi \cdot (\frac{q}{8})^2}{\sigma_{tot}^2 \cdot \ln(2)}\right)} \tag{5.6}$$

To facilitate a direct comparison with the FrodoKEM specification maximum limit of $2^{-138.7}$ as reported in Table 5.2, the exponential formula is converted into an explicit power-of-two form by mapping the system parameters directly into the base-2 exponent definition:

$$1 - \frac{\pi \cdot 4096^2}{78683.808 \cdot \ln(2)} = 1 - \frac{52707178.5813}{54539.4627} \tag{5.7}$$

Executing the division yields an intermediate exponential value of $1 - 966.4045$, which resolves to a final exponent of -965.4045. Therefore, the theoretical upper bound for the decryption failure rate evaluates to:

$$DFR \leq 2^{-965.40} \tag{5.8}$$

This cryptographically negligible probability confirms that the original framework maintains an exceptionally high error margin before a bit-rounding error can trigger an invalid decapsulation state.

When using the `add16u_0GN` approximate adder, the total error variance formula is updated to reflect the interaction between the secret matrix and the hardware-induced error introduction, and results in the following:

$$\sigma_{tot}^2 = 2N\sigma_{ax}^2\sigma^2 + \sigma_{ax}^2 \tag{5.9}$$

where $N = 640$, $\sigma = 2.8$ represents the standard deviation of the reference discrete Gaussian distribution for the secret vector $\mathbf{s}$, and $\sigma_{ax} = 2.7815$ denotes the intrinsic standard deviation of the approximate adder's error distribution. Substituting these parameters yields the concrete variance value for the approximate implementation:

$$\sigma_{tot}^2 = 2 \cdot 640 \cdot (2.7815)^2 \cdot (2.8)^2 + (2.7815)^2 = 77647.493 \tag{5.10}$$

By plugging this updated total variance along with the critical failure threshold $t = \frac{q}{8} = 4096$ into the subgaussian tail bound, the modified DFR can be evaluated as follows:

$$DFR_{approx} \approx 2 \cdot \exp\left(-\pi \cdot \frac{(\frac{q}{8})^2}{\sigma_{tot}^2}\right) = 2^{\left(1 - \frac{\pi \cdot (\frac{q}{8})^2}{\sigma_{tot}^2 \cdot \ln(2)}\right)} \tag{5.11}$$

To compute the final bound, these values are evaluated directly within the base-2 algebraic representation:

$$1 - \frac{\pi \cdot 4096^2}{77647.493 \cdot \ln(2)} = 1 - \frac{52707178.5813}{53820.5910} \tag{5.12}$$

Evaluating the fraction simplifies the exponential term to $1 - 979.3025$, resolving to a final target exponent of -978.3025. The theoretical upper bound for the decryption failure rate under

the proposed approximate paradigm is therefore bounded by:

$$DFR_{approx} \leq 2^{-978.30} \tag{5.13}$$

Although this calculation yields a slightly more negative theoretical exponent than the reference baseline ($2^{-965.40}$), this behavior stems from the fact that the characterized standard deviation of the `add16u_0GN` operator ($\sigma_{ax} = 2.7815$) is marginally lower than the nominal cryptographic target ($\sigma = 2.8$). This indicates that the approximate hardware execution achieves cryptographic equivalence to the reference scheme. As summarized in Table 5.2, both configurations inhabit an identical, negligible security tier situated safely below the maximum specification limit, verifying that the algorithmic-level error introduction remains transparent to the security guarantees of the cryptosystem.

Table 5.2 Comparison of Decryption Failure Rates (DFR) for AxFrodoKEM vs. reference values.

	Configuration	**Estimated DFR**
[A+20]	FrodoKEM (Specification Max Limit)	$2^{-138.7}$
[A+20]	FrodoKEM-640 (Reference)	$2^{-965.40}$
This work	**FrodoKEM with AxC (`add16u_0GN`)**	$\mathbf{2^{-978.30}}$

Experimental Setup and Results

After the mathematical analysis of the DFR conducted above, and to evaluate the practical feasibility and confirm the execution correctness of the scheme, the modified variant of FrodoKEM-640 was tested against the official NIST Known Answer Tests (KATs). The primary objective of this implementation was not to achieve an overwhelming software speedup, but rather to serve as a structural feasibility check, verifying that the error profiles inherently introduced via arithmetic-level approximation could completely replace the software-intensive explicit error-generation flow without disrupting functional and mathematical compliance.

The evaluation was conducted on a Linux system running kernel version 4.18.0, equipped with an Intel Core i3-2120 CPU at 3.30 GHz, 16 GB of RAM, and compiled using GCC version 8.5.0. Clock cycle metrics were captured by reading the processor's hardware Time Stamp Counter (TSC) register. The detailed performance breakdown across the core cryptographic routines (namely Key Generation, Encryption, and Decryption) is listed in Table 5.3. Because the approximate adders from the EvoApproxLib library were emulated via C functions during this verification stage, such a software-emulated execution incurs a massive processing penalty, taking roughly 20 to 30 times more clock cycles than a native assembly instruction. Therefore, to isolate and accurately evaluate the performance profile of an eventually integrated hardware accelerator, the execution cycles reported for the approximate design were calculated assuming that each approximate addition executes on-board in a single processor clock cycle, matching

a standard hardware adder. As shown in Table 5.3, the approximate implementation yields a 1.22% reduction in required pseudo-randomly generated bytes for an entire run, which translates directly into a 2.04% reduction in total execution clock cycles. Notably, the relative reduction in execution time surpasses the raw byte savings. This behavior occurs because error generation in standard FrodoKEM involves significant computational overhead beyond simple random byte sampling. The extracted bytes from the FIPS 202 `SHAKE128` extendable-output function must undergo a secondary, intensive discrete Gaussian sampling routine. By introducing the errors inline through arithmetic truncation, our approach completely eliminates these processing blocks concerning the error matrices $\mathbf{E}$, $\mathbf{E}'$, and $\mathbf{E}''$.

Table 5.3 AxFrodoKEM-640 execution time breakdown in kilo clock cycles (kcc) and pseudorandom error-generation byte requirements for the reference and proposed approximate architectures.

Metric / Configuration	**KeyGen**	**Encrypt**	**Decrypt**	**Total Run**
Execution Time (kcc)				
Reference [The26]	55,756	72,418	73,215	201,388
This work (with `add16u_0GN`)	**54,580**	**71,078**	**71,078**	**197,272**
Percentage Savings	*(-2.11%)*	*(-1.85%)*	*(-2.19%)*	*(-2.04%)*
Error Generation Volume (Bytes)				
Reference [The26]	839,680	839,808	850,048	2,529,536
This work (with `add16u_0GN`)	**829,440**	**829,440**	**839,680**	**2,498,560**
Percentage Savings	*(-1.22%)*	*(-1.23%)*	*(-1.22%)*	*(-1.22%)*

While these initial execution cycle gains are structurally limited because the main computational bottleneck remains the heavy symmetric cryptography primitives required to build the essential components of the scheme (such as the dynamic expansion of matrix $\mathbf{A}$ via `SHAKE128`), the architectural advantages are substantial. The requirement for Gaussian sampled data is effectively halved rather than entirely discarded because the secret matrices ($\mathbf{S}$ and $\mathbf{S}'$) must still undergo explicit generation and discrete Gaussian sampling (via `frodo_sample_n`), as shown in Listing B.7. However, by replacing the explicit error generation with inline error introduction via approximate arithmetic, the storage allocations for the standalone error matrices are completely removed from the execution footprint. This architectural elimination directly results in a significant reduction in data RAM usage within the stack allocation space during runtime.

Based on the top-level implementations detailed in Listing B.7, the exact memory savings can be estimated from the size of the excised data structures ($\mathbf{E}$, $\mathbf{E}'$, and $\mathbf{E}''$), where each matrix element is stored as a 16-bit unsigned integer (`uint16_t`, occupying 2 bytes). The reduction is quantified as follows:

- In `crypto_kem_keypair`, the elimination of the error matrix $\mathbf{E}$ removes an allocation of `PARAMS_N` $\times$ `PARAMS_NBAR` $\times 2$ bytes. For the FrodoKEM-640 parameter set ($N = 640, \bar{n} = 8$), this yields a direct stack savings of $640 \times 8 \times 2 = 10,240$ bytes (10 KiB).
- In both `crypto_kem_enc` and `crypto_kem_dec`, the removal of the error matrices $\mathbf{E}'$ (`Ep`) and $\mathbf{E}''$ (`Epp`) entirely cleans up an allocation footprint of (`PARAMS_N` + `PARAMS_NBAR`) $\times$ `PARAMS_NBAR` $\times 2$ bytes. This translates to an additional data RAM optimization of $(640 + 8) \times 8 \times 2 = 10,368$ bytes (10.125 KiB) per execution instance.

This functional validation confirms that error introduced via targeted arithmetic approximation is a viable alternative to explicit sampling, establishing a solid foundation for the area-optimized LWE hardware implementation detailed in the following.

5.7 Approximate LWE Architecture

The conceptual framework for Approximate Learning With Errors (AxLWE), established in Section 5.4, identifies the accumulation stages of the $\mathbf{A} \cdot \mathbf{s}$ inner product as the optimal insertion point for Approximate Adders (AxADD). By substituting exact arithmetic with these approximate operators, errors are embedded directly into the computation, potentially eliminating the requirement for standalone sampling hardware. Building upon the characterization of EvoApproxLib operators (Section 5.5) and the functional validation within the FrodoKEM framework (Section 5.6), this section presents the hardware implementation of the core AxLWE primitives. As demonstrated by the previous software case study, the runtime performance gains in execution cycles were relatively limited because the high-level computational flow remains heavily bottlenecked by the sequential execution of the FIPS 202 symmetric cryptographic primitives (such as the generation of matrix $\mathbf{A}$ via `SHAKE128`). Because general-purpose software metrics are intrinsically constrained by these symmetric building blocks, we explore whether the proposed AxLWE paradigm can deliver a more profound impact on dedicated hardware implementations, where silicon area cost and power consumption represent critical physical design constraints. By completely removing the dedicated Gaussian sampling circuits and their corresponding memory storage structures from the hardware footprint, the physical implementation may unlock significant area and power efficiencies that are otherwise hidden in software runtimes.

To ensure an evaluation independent of specific protocol constraints, the design focuses on a generalized matrix-vector multiplication architecture. The evaluated LWE instance is characterized by a standard parameter tuple consisting of a problem size $n = 512$, a modulus $q = 2^{16}$ (16-bit coefficients), and an error distribution with a standard deviation of $\sigma = 2.8$. Under the concrete cryptanalytic hardness framework evaluated using the lattice estimator tool detailed in [APS15], these parameters guarantee a security level equivalent to 100 bits. At a high level, a security level of λ bits represents a standard structural metric of cryptographic strength. This implies that an adversary attempting to compromise the system using the most

effective known mathematical attacks must expend a minimum computational workload of 2^{λ} operations. Consequently, a 100-bit security margin guarantees that solving this specific LWE instance requires a minimum of 2^{100} operations across the main established cryptanalytic strategies, such as utilizing lattice reduction algorithms to find an underlying short vector, or solving it as a bounded distance decoding problem to isolate the target vector from the embedded error components. This parameter selection establishes a robust and realistic testing baseline that directly balances security verification with the physical constraints of embedded cryptographic implementations.

By aligning our experimental setup with these dimensions, we ensure that the architectural benefits observed for the AxLWE paradigm remain meaningful, secure, and directly applicable to real-world lattice-based cryptographic hardware. To quantify the hardware benefits of the AxLWE paradigm, the evaluation methodology follows a comparative approach: we first implement a baseline LWE architecture utilizing a conventional Gaussian Sampling (GS) unit, and subsequently an implementation leveraging the proposed AxLWE technique (completely removing the GS hardware) to assess the exact physical reductions in silicon area cost and power consumption.

5.7.1 Baseline LWE Hardware Accelerator

The baseline hardware architecture of the LWE accelerator is designed for optimized throughput by utilizing independent, parallelizable modules. A high-level schematic of the system is provided in Figure 5.9, detailing the main functional blocks.

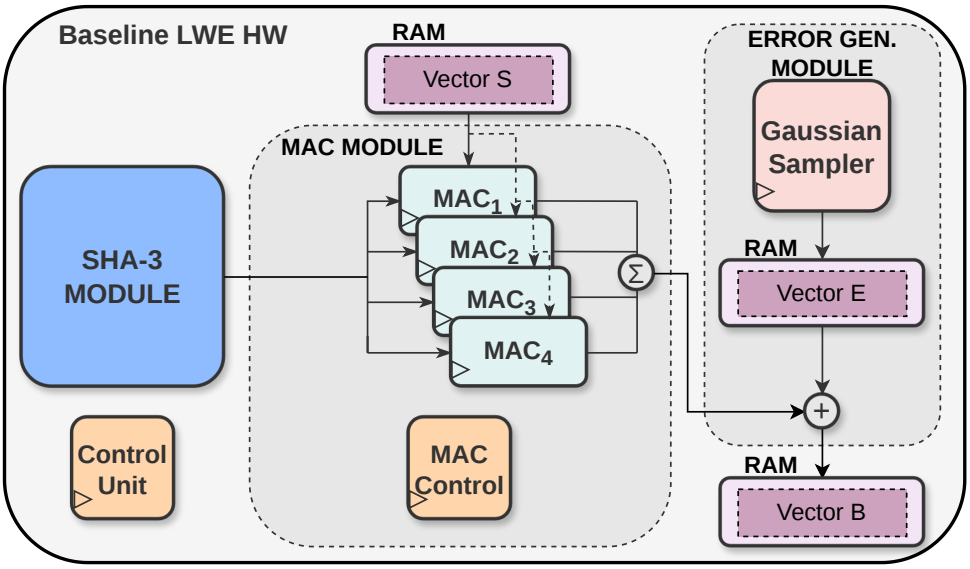


Figure 5.9 Baseline hardware architecture for LWE acceleration. The design features a SHA-3 Module for pseudo-random data generation and a MAC module for inner product accumulation. The right-most dashed block (Gaussian sampler and error Vector E memory) represents the explicit error generation logic.

The main Control Unit (CU) serves as the central orchestrator, managing the parallelization and dataflow across all architectural modules. Implemented as a synchronous Finite State Machine (FSM), this CU decodes external execution commands to coordinate internal memory mapping and cryptographic pipelines. During data preparation phases, the FSM manages sequential data flows with internal storage units, handling the tracking of secret vector retrieval during the `LOAD_S` and `MEM_S_READ_ALL` states, and regulating the tracking of final vector elements within the `MEM_B_READ_ALL` state. Crucially, the main matrix-vector multiplication loop is driven by structural state transitions designed to interleave the data generation of the symmetric SHAKE core with the operations of the arithmetic Multiply and ACcumulate (MAC) engine. Upon receiving a `COMPUTE_B` instruction, the FSM cycles through an initialization and transition phase (`SHAKE_A_TRANSITION` and `WAIT_SHAKE_A`) to hand over execution control to the dedicated MAC Control unit, thereby freeing the main CU to concurrently instruct other architectural modules or process incoming commands in the meantime. To mitigate processing latencies, the controller coordinates concurrent hardware execution in the `START_MAC` state, triggering a sequential squeezing operation on the SHAKE core to extract fresh pseudo-random matrix coefficients in parallel with ongoing dot-product calculations. Synchronization between modules is finalized within the `WAIT_MAC` state. If the arithmetic core signals that it requires additional error or coefficient data, the FSM loops back to the generation wait state. Conversely, once a complete row calculation concludes, the controller safely terminates execution and reverts to the `IDLE` state.

To prepare for computation, the Error Generation Module produces 16-bit error elements by means of a Gaussian Sampler. Governed by a dedicated synchronous FSM, this block functions as an arithmetic wrapper around a core sampling unit (CDF Gaussian Sampler) to transform raw uniform random words into a vector of Gaussian-distributed error values. The architecture handles a wide 64-bit uniform random input interface, which consolidates the required data for multiple operations into a single transfer. Upon activation via an external start signal coming from the CU, the local FSM transitions from an `IDLE` state into a `RUNNING` routine to process the wide random word sequentially. During this active execution, the module automatically partitions the incoming data into individual 16-bit uniform streams, which are sequentially delivered to the underlying Gaussian sampling core. The submodule converts each uniform slice into a mathematically valid discrete Gaussian error sample. The FSM iteratively captures these 16-bit signed error elements, populating an internal array structure over consecutive clock cycles. Once the sequential sampling loop completes all four execution iterations, the FSM transitions to a final state, packing the gathered samples into a unified 64-bit parallel output bus and asserting a completion flag to notify the system controller before resetting back to its initial waiting configuration.

Internally, as structurally illustrated in Figure 5.10, the CDF Gaussian Sampler converts uniform random data into mathematically compliant discrete Gaussian outputs by utilizing a table-inversion method mapped to an internal Cumulative Distribution Function (CDF) lookup table. The CDF-based sampling technique was selected for this implementation, as it represents

the most popular approach in practical realizations and is the specific method adopted by the FrodoKEM specification [A+20]. When a 16-bit uniform random input is provided, the block extracts a segment for distribution comparison and preserves a single bit to determine the sign of the final element. During its active processing phase, the local controller steps sequentially through the hardcoded CDF thresholds, updating a counter each time the random value exceeds a table boundary, which guarantees constant-time sampling behavior required to prevent timing attacks. Once this scanning procedure traverses the table entries, the module applies the sign bit to the accumulated count to compute the final signed 16-bit error sample, immediately asserting its completion flag to signal that a valid error sample is ready for storage in memory.

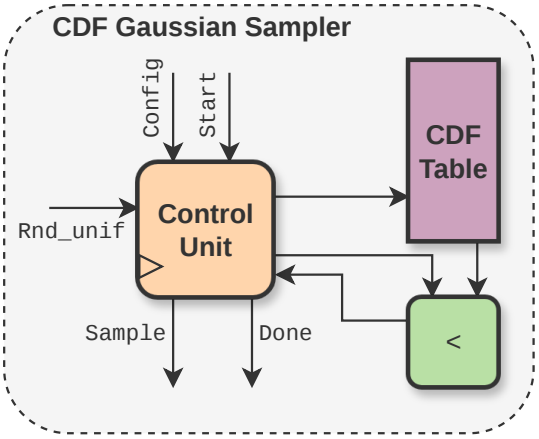


Figure 5.10 Hardware schematic of a CDF-based Gaussian sampler, featuring a control unit, internal CDF table, and interval comparison logic.

For randomness generation, the SHA-3 Module block (on the left in Figure 5.9) follows the FIPS 202 standard to implement the SHAKE128 extendable-output function. While structurally derived from the core architecture presented in [CRA+24], this variant differs by utilizing only a single input register file and a standard, unmasked `Keccak` core for area optimization compared to the design introduced in Section 4.3. Once an initial seed is provided, the module operates autonomously under its own local control logic, executing independently of the main CU. The secret vector $\mathbf{S}$ is stored in a RAM block configured with 64-bit words, where each word packs four 16-bit secret elements. This optimal memory configuration was identified as the most suitable architectural choice by utilizing the Global Foundries Memory Explorer [Glo24].

The central computational core, namely the MAC Module, performs the $\mathbf{A} \cdot \mathbf{s}$ inner product. This is implemented as an autonomous module managed by a dedicated local MAC control unit. Structurally, the hardware instantiates four individual MAC units operating in parallel. To support this parallel execution, data is routed through a wide 64-bit data path that is divided into independent 16-bit segments for each execution unit. Each channel simultaneously ingests a 16-bit word from the secret vector $\mathbf{S}$ RAM and a 16-bit word from the SHA-3 Module representing one coefficient of the matrix $\mathbf{A}$. This parallel configuration is critical for balancing system dataflow. The SHA-3 Module generates 1280 bits of pseudo-randomness at each full permutation, and deploying four distinct arithmetic units ensures that all available random bits

are efficiently consumed by the time a new set is computed. Internally, each individual MAC element (Figure 5.11) performs its arithmetic operations through synchronized multiplication and accumulation stages. The two incoming 16-bit data words are multiplied, and the resulting product is added to the current feedback state managed by an internal 16-bit register to update the running sum. After 128 accumulation steps, the independent outputs from the four units are reduced via a downstream adder tree (represented by the Σ symbol) into a single combined sum. Finally, a 16-bit error element is added to this total to produce a single element of the result vector **B**, which is then committed to its dedicated 16-bit word RAM. This highly decoupled architectural design ensures that the arithmetic core and the SHA-3 engine can operate concurrently, maximizing overall system throughput by completely overlapping pseudo-randomness generation with matrix computation.

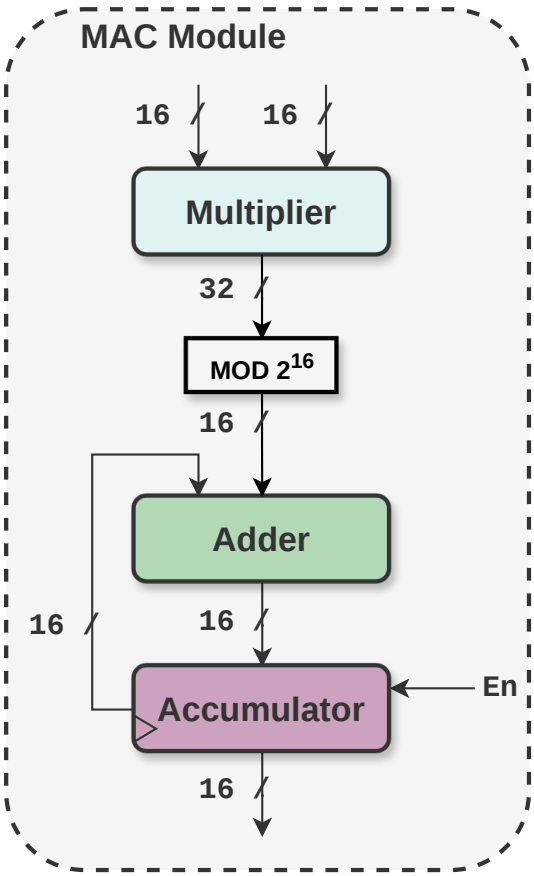


Figure 5.11 Hardware schematic of an individual MAC element, featuring a multiplier, an adder, and an enabled accumulator register with feedback routing.

5.7.2 AxLWE Hardware Accelerator

The AxLWE hardware accelerator is derived from the baseline architecture described in Subsection 5.7.1, incorporating targeted modifications to transition from explicit to implicit error introduction.

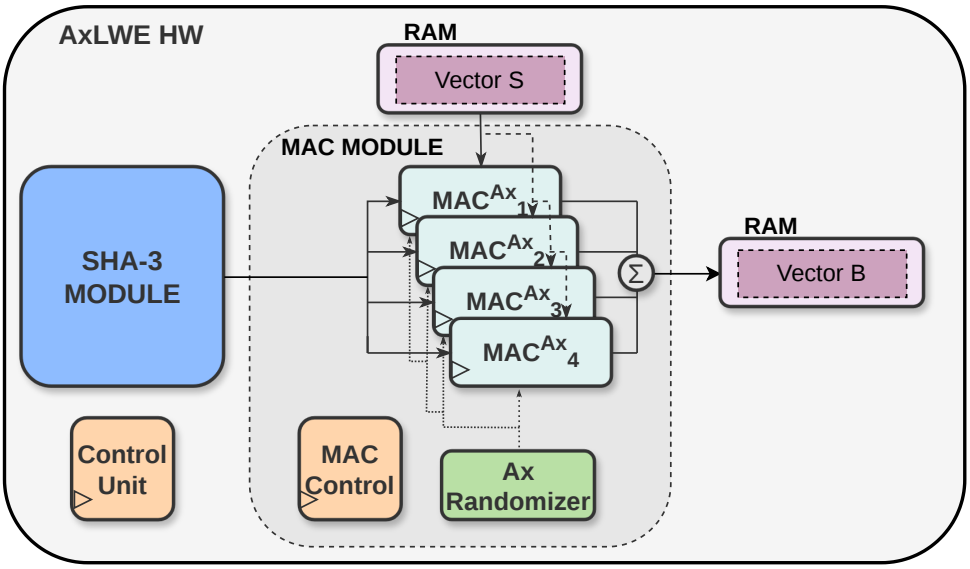


Figure 5.12 Proposed hardware architecture of the AxLWE accelerator. Error introduction is performed via MAC^{Ax} units and the Ax Randomizer.

As structurally illustrated in Figure 5.12, the most significant architectural modification is the complete removal of the Error Generation Module. This entirely eliminates the Gaussian sampler core hardware along with its associated 16-bit word RAM memory that was previously required to store the error vector $\mathbf{E}$. By pruning the design and bypassing this entire explicit error generation path, the hardware achieves substantial savings in silicon area. This structural simplification also removes the operational latency normally incurred during explicit error pre-calculation, allowing the system to achieve a more compact and direct execution timeline. Moreover, the conventional MAC units are replaced with Approximate MAC (i.e., MAC^{Ax}) blocks, which integrate an approximate adder (namely the `add16u_0GN`) in parallel with the standard exact adder at the accumulation stage, as represented in Figure 5.13. While the initial multiplication stage of the two incoming 16-bit words remains exact, the subsequent accumulation pipeline is modified to facilitate structural error introduction directly within the arithmetic datapath. To achieve this, the block introduces a dedicated approximation enable control signal that allows for dynamic, runtime configuration of the matrix calculation. When this control signal is asserted, the internal combinational logic switches the accumulation path to utilize the approximate adder, combining the multiplication product with the current feedback state through a simplified arithmetic structure to generate the implicit error. Conversely, when the control signal is driven low, the module bypasses the approximate logic and functions as a standard exact accumulator. This dual-mode capability provides the hardware with the flexibility to selectively toggle between exact computation and hardware-driven approximate accumulation before capturing the final 16-bit sum into the synchronous accumulator register.

To ensure the statistical robustness of the introduced error and prevent data dependencies, the AxRandomizer (indicated in green in Figure 5.12) governs the `Ax_En` control signal. This signal dictates an intermittent activation strategy for the approximate adder rather than an always-on

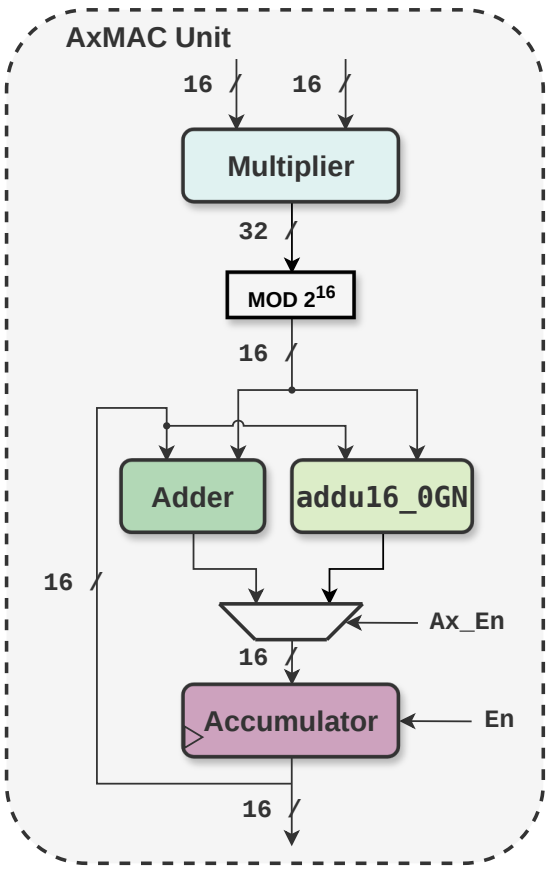


Figure 5.13 Hardware schematic of the Approximate Multiply-Accumulate unit. The architecture features a 16-bit Multiplier followed by a selectable accumulation stage. A multiplexer facilitates switching between a standard exact Adder and the approximate `addu16_0GN` operator, enabling either conventional accumulation or approximate computation for implicit error introduction within the same hardware footprint.

operational mode. Due to the invariance property of the error distribution under addition, a single activation of the approximate adder successfully propagates its intrinsic statistical error characteristics to the final computed result b_i. Conversely, repeated activations during the same inner product computation would cause an uncontrolled additive accumulation of errors. Such an accumulation would quickly exceed the tight error boundaries mandated by the cryptographic specifications, inevitably leading to decryption failures. Therefore, the approximate addition must be introduced exactly once per b_i element. Because digital approximate hardware is deterministic, processing identical inputs $(\mathbf{A}, \mathbf{s})$ while executing the approximate addition at the exact same accumulation index would always yield the same error value. To prevent this predictability, the AxRandomizer randomizes the insertion point of the approximate addition during the partial sum accumulation, thereby dynamically modifying the final introduced error value. Crucially, the AxRandomizer exploits 9 random bits originating from an entropy source that is independent of the one used to generate the pseudorandom matrix $\mathbf{A}$. This strict separation between the two entropy sources ensures that the error insertion logic remains decoupled from the input data path. Specifically, because the inner product requires $n = 512$ accumulation steps, a 9-bit random value perfectly maps the entire space of possible step configurations ($2^9 = 512$).

Under a uniform distribution, each accumulation index is selected with an equal probability of $\frac{1}{512}$, maximizing the behavioral entropy of the execution path. This uniform uncertainty guarantees the unpredictability of the error insertion point and actively prevents deterministic hardware patterns, satisfying a critical requirement for secure cryptographic primitives [ZBK22].

5.8 Results

This analysis evaluates the performance benefits of the AxLWE paradigm compared to the baseline implementation, with a summary of these results reported in Table 5.4. To establish a software baseline for our LWE and AxLWE hardware implementations, we chose to use an embedded-class CPU, namely the CV32E40P RISC-V core. We implemented the core $\mathbf{b} = \mathbf{As} + \mathbf{e}$ operation in C and compiled it using GCC 10.1.0. This software implementation required 93,906,925 clock cycles to complete the error generation and the full computation of vector $\mathbf{b}$. For the hardware evaluation, we modeled both the baseline LWE and the AxLWE accelerators at the Register-Transfer Level (RTL) using Verilog and simulated them with QuestaSim 2021.4.2. To compute the same vector $\mathbf{b}$, the AxLWE hardware required only 96,910 clock cycles, achieving a 7.73% reduction in execution time compared to the baseline LWE implementation utilizing CDF-based error sampling.

Table 5.4 Performance and Efficiency Comparison of LWE and AxLWE HW Implementations

Metric / Configuration	**LWE-cdf**	**AxLWE**
SW Reference		
Total Cycles	93,906,925	–
Execution (CC)		
Total Cycles	105,030	**96,910**
Gain	–	*(-7.73%)*
Speedup vs. SW	894.1×	969.0×
Power (mW)		
Total Power	3.5493	**2.9118**
Power Reduction	–	*(-17.96%)*
Energy (nJ)		
Total Energy	643.05	**486.76**
Energy Reduction	–	*(-24.30%)*

[a]Measured at a 1.725 ns clock period. Gains are relative to LWE-cdf.

To evaluate the area, power, and energy efficiency of our proposed solution, we performed logic synthesis for an ASIC target using Synopsys Design Compiler U-2022.12-SP6, targeting a

22 nm FDSOI standard cell library from GlobalFoundries under typical operating conditions. By eliminating the explicit Error Generation Module, our AxLWE accelerator achieves a **13.9%** reduction in area, occupying 111.0 kGE (kilo-gate equivalents) compared to the 128.8 kGE required by the baseline. As illustrated in Figure 5.14, the primary area savings stem from the reduced memory requirements and a simplified overall control logic, which more than offset the slight area increase of the AxMAC unit containing the `add16u_0GN` approximate adder. This architectural simplification directly translates to improved power efficiency, yielding a **17.96%** reduction in post-synthesis total power dissipation for the AxLWE implementation, as reported in Table 5.4. Furthermore, combining this lower power demand with the reduced clock cycle count required to compute **b** allows our AxLWE implementation to achieve overall energy savings of **24.30%**. These energy metrics were computed by multiplying the post-synthesis total power consumption by the total execution time, derived from the cycle count at the target clock period of 1.725 ns. We note that while static post-synthesis power reports typically provide conservative (often overestimated) absolute values compared to post-layout dynamic simulations with activity traces, they offer a consistent and reliable basis for relative comparative analysis across both architectures under identical synthesis constraints.

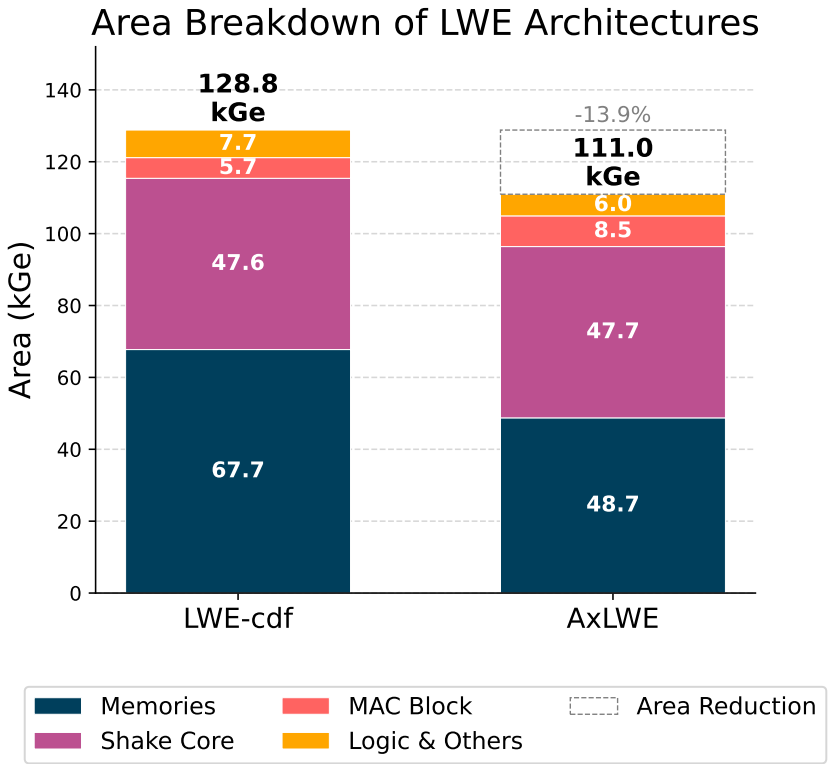


Figure 5.14 Detailed area breakdown comparison (in kilo-gate equivalent kGe) of the LWE-cdf and AxLWE accelerators.

Table 5.5 provides a qualitative comparison with related approaches in the literature. While a strict one-to-one benchmark across all studies is not feasible, existing works such as [BHS18, KBK+20, KBW+21, O+22] focus primarily on the R-LWE problem, utilizing AxC to reduce the bit-widths of operands or error terms. However, these studies generally do not characterize the

resulting impact of the approximation on the final error distribution. Techniques leveraging VOS in LPPN and LWPE [K+20, B+21, B+22] demonstrate that hardware perturbations can provide flexibility in error shaping. However, these frameworks rely on analog-driven mechanisms that are inherently difficult to control compared to purely digital implementations. Furthermore, a direct quantitative comparison with these VOS-based methods is fundamentally hindered by differing architectural evaluation assumptions and limited available data. In contrast, our work provides a detailed quantitative analysis comparing LWE implementations with and without explicit error generation, fully accounting for the hardware area and power consumption of the complete Gaussian sampling chain. Conversely, the evaluations in [B+21, B+22] only isolated and measured the cost of VOS-specific hardware compared directly against the baseline inner-product architecture. While the authors of those works stated that they reduced the cost of error generation by an amount equivalent to an RNG, this perspective does not fully reflect total system requirements. Executing the underlying cryptographic scheme still fundamentally requires an entropy source for other primitives, such as generating the pseudorandom public matrix $\mathbf{A}$. Consequently, the only real architectural gain provided by that technique on the error generation side is the removal of the Gaussian sampler.

The proposed AxLWE paradigm bridges these methodological gaps by leveraging the area efficiency of AxC to completely eliminate the dedicated Gaussian sampler. It utilizes digital hardware perturbations to introduce the required LWE errors while providing a statistical characterization of the resulting error distribution.

Table 5.5 State-of-the-Art Comparison on Approximate LWE-based Schemes

Ref.	Problem	Approximation Type	Gaussian Sampler	Error Analysis
[BHS18]	R-LWE	DRUM	Yes	No
[KBK+20]	R-LWE	AxMM	Yes	No
[KBW+21]	R-LWE	Truncation	Yes	No
[O+22]	R-LWE	Truncation	Yes	No
[K+20]	LPPN	Analog (VOS)	No	Yes
[B+21]	LPPN	Analog (VOS)	No	Yes
[B+22]	LWPE	Analog (VOS)	No	Yes
This work	**LWE**	**Digital (AxC)**	**No**	**Yes**

5.9 Summary

This chapter presented the architectural framework, characterization methodology, and hardware implementation of the Approximate Learning With Errors (AxLWE) paradigm, demonstrating the viability of utilizing fully digital Approximate Computing (AxC) as a controlled hardware error

source for lattice-based cryptographic primitives. By exploiting the inherent error-tolerant and probabilistic properties of the Learning With Errors (LWE) problem, it established a methodology to select suitable approximate operators to replace conventional Gaussian sampler hardware, directly introducing the error within computations to achieve substantial gains in silicon area, power dissipation, and energy consumption.

A thorough review of existing state-of-the-art implementations highlights a clear dichotomy in current literature approaches. Traditional approximate frameworks, such as DWE [BHS18], AxMM [KBK$^+$20], and AxRLWE [KBW$^+$21], focus primarily on the algebraic Ring-LWE problem and optimize performance through arithmetic truncation or approximate multipliers. However, they do not characterize the resulting impact on the error profile and still strictly mandate a dedicated hardware Gaussian sampler. Conversely, alternative approaches leveraging Voltage-Frequency Over-Scaling (VOS) within LPPN and LWPE frameworks [K$^+$20, B$^+$21, B$^+$22] successfully eliminate explicit error generation blocks by utilizing analog-driven hardware perturbations. Nonetheless, because these VOS structures rely on custom full-cell layouts, variable delay lines, and analog calibration feedback loops, they are highly technology-dependent and susceptible to manufacturing process variations, requiring an expensive per-device calibration phase for every individual chip. The proposed AxLWE paradigm extends the current state-of-the-art to resolve these limitations. By operating completely within a standard digital ASIC flow, it guarantees portability across any technology node without the need for custom macroblocks or per-device tuning, while simultaneously providing a statistical characterization and analysis of the introduced hardware error.

The architectural transition from an explicit sampling chain to native error embedding was formalized by analyzing the core matrix-vector operation $\mathbf{b} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e}$. As illustrated by comparing the standard inner product chain (Figure 5.3) with the reduced variant (Figure 5.4), AxLWE enables the removal of the standalone explicit Gaussian error generation and addition stages. Instead, an approximate adder (AxADD) is integrated directly within the internal accumulation step of the $\mathbf{A} \cdot \mathbf{s}$ inner product, introducing the required LWE error implicitly during calculation. To identify an arithmetic operator capable of mimicking a Gaussian error source, a multi-stage hardware selection and statistical characterization methodology was established (Figure 5.5). Candidate 16-bit approximate adders from the EvoApproxLib library [M$^+$17] were evaluated through functional behavioral simulations to isolate their intrinsic error distributions. These empirical profiles were directly benchmarked against target discrete Gaussian distributions utilizing an automated suite of goodness-of-fit and distance tests, including the Hellinger Distance (HD), Kullback-Leibler (KL) Divergence, discrete Kolmogorov-Smirnov (KS) test, and Chi-Squared (χ^2) analysis.

To validate this approach within a real-world cryptographic scheme, a software-emulated case study was conducted on the FrodoKEM-640 parameter set. FrodoKEM was selected due to its conservative, unstructured LWE formulation, active ANSSI recommendation, and the availability of official NIST KATs to verify execution correctness. Restructuring the matrix func-

tions to embed the approximate operators inline yielded a 1.22% reduction in pseudo-randomly requested bytes and a direct 2.04% reduction in total runtime clock cycles (Table 5.3). Furthermore, this optimization cut data RAM usage within the stack allocation space by removing the standalone error matrices ($\mathbf{E}, \mathbf{E}', \mathbf{E}''$), saving 10 KiB during keypair generation and 10.125 KiB per encryption/decryption instance. The characterization flow successfully identified the `add16u_0GN` operator as the optimal candidate, exhibiting minimal statistical divergence (HD = 0.1817, hardware $\sigma = 2.7815$) relative to the nominal cryptographic target ($\sigma = 2.8$). Incorporating this empirical variance into the subgaussian tail bound confirmed that the approximate configuration maintains an exceptionally high security margin, yielding a cryptographically negligible DFR of $2^{-978.30}$, which sits safely below the maximum specification limit (Table 5.2).

Finally, to decouple the evaluation from specific protocol bottlenecks, a generalized, protocol-independent matrix-vector hardware prototype ($n = 512$, $q = 2^{16}$, 100-bit security) was implemented to contrast a baseline LWE accelerator with the proposed AxLWE core. To support throughput balancing, the baseline design instantiates four parallel MAC elements backed by a CDF-based Gaussian sampler and a dedicated error memory block (Figure 5.9). In the AxLWE hardware accelerator (Figure 5.12), the entire Error Generation Module is removed, the standard arithmetic units are replaced with MAC^{Ax} blocks containing the `add16u_0GN` approximate adder (Figure 5.13), and an AxRandomizer utilizes 9 independent uniformly distributed bits to dynamically alter the insertion index across the 512 accumulation steps, maximizing behavioral entropy and preventing deterministic hardware patterns. Both designs were synthesized for an ASIC target utilizing a 22 nm FDSOI standard cell library from GlobalFoundries with Synopsys Design Compiler U-2022.12-SP6 at a target clock period of 1.725 ns.

The quantitative physical synthesis and simulation results compiled in Table 5.4 and Figure 5.14 confirm the efficiency advantages of the AxLWE paradigm:

- **Area Footprint Optimization:** Bypassing the explicit error sampling logic and its associated memory arrays reduces total silicon area by **13.9%**, shrinking the design from 128.8 kGE down to a more compact 111.0 kGE.
- **Execution Latency Reduction:** Eliminating the sequential latency penalties incurred during explicit error pre-calculation optimizes the internal control flow, lowering execution time from 105,030 down to **96,910 clock cycles** (a 7.73% speedup).
- **Power Dissipation Savings:** The significant structural simplification of the macro-architecture directly translates to physical power efficiency, achieving a **17.96%** reduction in total post-synthesis power consumption (2.9118 mW vs. 3.5493 mW).
- **Cumulative Energy Efficiency:** Driven by the simultaneous reduction in active power demands and clock cycle count, the AxLWE accelerator delivers a substantial post-synthesis energy reduction of **24.30%**, requiring only 486.76 nJ compared to the 643.05 nJ demanded by the baseline configuration.

These findings highlight that embedding targeted digital hardware approximations within the arithmetic execution path can completely replace conventional software-intensive and area-heavy sampling chains, establishing a highly portable and low-power hardware baseline for post-quantum cryptographic implementations.

Chapter 6

Conclusion

Implementing post-quantum cryptographic primitives in physical hardware introduces significant design trade-offs between theoretical security and microarchitectural efficiency. While theoretical formulations like Module-LWE (MLWE) and standard, unstructured Learning With Errors (LWE) provide mathematically sound security bounds against classical and quantum adversaries, their practical realizations are bounded by severe computational bottlenecks, strict memory footprints, and critical physical vulnerabilities. This thesis has systematically investigated this cross-domain layer, introducing targeted microarchitectural optimizations, hardware accelerators, and alternative arithmetic paradigms to bridge the gap between algorithmic theory and deployable, secure hardware systems. The investigative approach of this research has been driven by a two-fold strategy designed to optimize the two primary computational bottlenecks inherent to lattice-based post-quantum cryptography, namely the heavy symmetric cryptographic expansion layer and the resource-intensive error generation chain.

The first phase of this work established a physically secure, high-throughput foundation for the initial pseudo-randomness generation stage. Because standardized post-quantum algorithms, including the finalized key-encapsulation standard ML-KEM (FIPS 203), rely continuously on the Keccak sponge permutation (FIPS 202) for matrix generation and seed expansion, the total processing latency of the entire cryptosystem is fundamentally constrained by the speed of these symmetric hashing operations. On embedded platforms, these operations are highly vulnerable to non-invasive side-channel analysis, such as Differential Power Analysis (DPA) and Electromagnetic Analysis (EMA). To protect these building blocks, this thesis detailed the complete development of a first-order masked hardware accelerator compliant with the FIPS 202 standard, protected via Domain-Oriented Masking (DOM). To preserve the required non-completeness properties, a custom Input Management Module was designed to handle byte-aligned formatting and apply structural padding to only one of the two independent data shares, successfully preventing deterministic padding data from canceling out during bitwise recombination. This microarchitectural design directly exposed and resolved a critical, frequently hidden bottleneck in hardware masking frameworks, which centers on the massive entropy consumption required

to preserve domain isolation across non-linear logic transformations. To satisfy the intensive 1600-bit-per-cycle entropy requirements of the non-linear DOM operations, we introduced an integrated hardware module termed the Randomness Dispatcher. Operating under a four-state finite state machine, this module parallelizes bit generation using 25 independent instances of 64-bit unrolled lightweight stream ciphers, configurable between the standardized Trivium architecture and its area-reduced variant, Bivium B, while facilitating software-transparent, on-demand reseeding directly via software.

The complete masked cryptographic subsystem was validated at the system level by embedding both the accelerator and the Randomness Dispatcher into a complete 32-bit RISC-V System-on-Chip (SoC) built around an OBI-connected CV32E40P processor core (Figure 4.6). The hardware was evaluated in a practical application scenario by offloading the cryptographic hash operations of a standardized `ML-KEM.Decaps` routine, where all assembly routines from prior literature were entirely rewritten into portable C code. The accelerator delivered an overall system-level speedup factor of $2\times$, reducing total execution latency from approximately 6.8 million down to 3,164,677 clock cycles (Table 4.2), while successfully dropping the performance footprint of the hashing layers from 62% of the runtime down to just 1% (Figure 4.10). Physical security margins were rigorously validated through electromagnetic Test Vector Leakage Assessment (TVLA) on the FPGA development platform using an electromagnetic probe and a digital oscilloscope, proving the complete absence of first-order information leakage ($|t| < 4.5$) across one million physical execution traces (Figure 4.8, Figure 4.9). Comprehensive synthesis mapping on an Artix-7 FPGA using Vivado 2021.1 (Table 4.1) and on a standard-cell 22nm FD-SOI library using Synopsys Design Compiler (Figure 4.7) provided a transparent account of the real physical costs of masking, revealing a comprehensive area overhead ranging from 300% to 500% depending on the target frequency constraints and choice of dispatcher core.

The second phase of this thesis transitioned from securing the baseline expansion layers to fundamentally redefining the subsequent error generation chain, moving from traditional explicit error sampling to an alternative, fully-digital implicit error embedding methodology. Conventional lattice architectures utilize expanded pseudorandom bitstreams to explicitly construct precise, Gaussian distributed error vectors via complex look-up tables or combinational networks, introducing substantial latency penalties and independent RAM storage overheads. Conventional lattice architectures utilize expanded pseudorandom bitstreams to explicitly construct precise, Gaussian distributed error vectors via complex look-up tables or combinational networks, introducing substantial latency penalties and independent RAM storage overheads. Earlier approximate computing frameworks addressed these bottlenecks by applying arithmetic optimizations like truncation or inexact multipliers to streamline the algebraic data path [BHS18, KBK+20, KBW+21, O+22], yet they did not characterize the resulting impact on the final error distribution and ultimately retained conventional explicit samplers. Conversely, prior literature attempts to entirely bypass these explicit samplers relied on analog-driven voltage and frequency scaling or clock-skew tuning techniques within physical noise learning paradigms [K+20, B+21, B+22], which carry severe technology dependencies and demand costly per-chip

calibration to mitigate manufacturing process variations. This research solved these deficiencies by introducing the AxLWE paradigm. This research extended the current literature by introducing the AxLWE paradigm. By aligning the probabilistic error requirements of lattice-based cryptography with the intrinsic combinational inaccuracies of digital approximate computing elements, an approximate hardware adder (AxADD) is integrated directly within the internal accumulation step of the $\mathbf{A} \cdot \mathbf{s}$ inner product, introducing the required LWE error implicitly during calculation. To identify an arithmetic operator capable of mimicking a Gaussian error source, a multi-stage hardware selection and statistical characterization methodology was established. Candidate 16-bit approximate adders from the open-source EvoApproxLib library [M+17] were simulated behaviorally and benchmarked against target profiles using an automated testing environment driving Hellinger Distance (HD), Kullback-Leibler (KL) Divergence, discrete Kolmogorov-Smirnov (KS), and Chi-Squared (χ^2) distance metrics.

The characterization flow successfully identified the `add16u_0GN` operator as the optimal candidate (Table 5.1, HD $= 0.1817$, hardware $\sigma = 2.7815$) relative to the nominal cryptographic target ($\sigma = 2.8$, Figure 5.7). Integrating its behavior into a real-world post-quantum case study focusing on the conservative, unstructured lattice-based protocol FrodoKEM-640 verified full functional compliance with official NIST Known Answer Tests (KATs) (Table 5.3). Concurrently, removing the standalone error matrices ($\mathbf{E}, \mathbf{E}', \mathbf{E}''$) optimized data RAM stack usage by up to 10.125 KiB per execution instance. Subgaussian tail-bound estimation proved that this approximate execution preserves cryptographic integrity, guaranteeing a negligible Decryption Failure Rate (DFR) of $2^{-978.30}$ situated safely below the maximum specification limit of $2^{-138.7}$ (Table 5.2). To decouple the evaluation from specific protocol symmetric-cryptography bottlenecks, a protocol-independent matrix-vector hardware prototype ($n = 512$, $q = 2^{16}$, 100-bit security baseline) was modeled at the Register-Transfer Level (RTL) using Verilog to compare a parallel LWE baseline architecture (Figure 5.9) against the proposed AxLWE implementation (Figure 5.12). To ensure long-term cryptanalytic unpredictability and prevent deterministic hardware patterns, the AxLWE core incorporates an AxRandomizer block governed by 9 independent, uniformly distributed bits to dynamically alter the approximate adder's insertion index across the 512 internal accumulation steps.

Both layout accelerators were synthesized for an ASIC target utilizing a 22nm FD-SOI standard cell library from GlobalFoundries with Synopsys Design Compiler at a target clock period of 1.725 ns. By completely removing the explicit Error Generation Module, the Gaussian sampler logic, and its associated error vector memory, the AxLWE hardware achieved both hardware and performance optimizations relative to the parallel, CDF-driven baseline LWE accelerator. Specifically, the AxLWE accelerator optimized the total silicon area footprint by 13.9%, shrinking the design from 128.8 kGE down to a more compact 111.0 kGE. This structural simplification optimized the internal control flow and bypassed sequential pre-calculation delays, lowering execution time from 105,030 down to 96,910 clock cycles, which represents a 7.73% speedup. This combination of reduced active switching activity and shorter execution windows directly enhanced physical power and energy metrics, delivering a 17.96% reduction in total

power consumption (2.9118 mW vs. 3.5493 mW) and a substantial cumulative energy saving of 24.30%, requiring only 486.76 nJ compared to the 643.05 nJ demanded by the baseline configuration for a complete $\mathbf{A} \cdot \mathbf{s}$ computation (Table 5.4).

In summary, this thesis explored the optimization of post-quantum cryptography with a two-fold approach. To begin with, a classical hardware accelerator was designed and embedded in a SoC, and a thorough analysis mapped out the transparent, unavoidable physical costs and resource complexities necessary to securely harden essential symmetric building blocks against physical side-channel exploitation, which are often overlooked in the state-of-the-art. For the innovative research path, this thesis introduced the Approximate Learning With Errors (AxLWE) paradigm, extending the literature with a highly portable digital solution and challenging traditional cryptographic design conventions by demonstrating that hardware approximation is not inherently a security risk. Instead, within the mathematically bounded parameters of the LWE problem, AxC can be harnessed as an implicit source of controlled errors leading to area and power optimization in physical implementations. By demonstrating that fully digital approximate components can eliminate explicit sampling circuits while preserving cryptographic security bounds and algorithmic compliance, this work establishes a portable solution for resource-constrained post-quantum implementations.

Future Perspectives

The methodologies and hardware implementations validated in this thesis establish the foundational viability of the AxLWE paradigm, yet they represent only the initial steps toward fully integrating hardware approximation into cryptographic engineering. To guide subsequent developments, several promising research trajectories can be envisioned, spanning automated hardware synthesis, microarchitectural scaling, and formal security theory.

From an AxC perspective, a compelling direction for future research involves transitioning from the screening of pre-characterized libraries toward the automated, generative design of custom approximate operators. While the methodology established in this work successfully identified suitable approximate adders within existing repositories, those circuits were originally evolved to optimize conventional arithmetic aggregates like mean absolute error for error-resilient multimedia workloads. Future efforts could focus on developing a formal synthesis methodology to design hardware structures with explicitly controlled, target error distributions when stimulated by uniform inputs. By integrating statistical goodness-of-fit and distance metrics, such as the Hellinger distance or Kullback-Leibler divergence, directly into the fitness functions of automated circuit generation tools like Cartesian Genetic Programming, it could be possible to evolve approximate operators tailored to specific cryptographic error constraints. This generative framework would allow for the on-demand synthesis of approximate adders that natively produce ideal discrete Gaussian or binomial curves, providing a predictable and highly customizable hardware-driven error baseline for various lattice-based configurations.

Beyond CGP-based design tools, another promising trajectory lies in expanding both the algorithmic scope and the microarchitectural complexity of this implicit error framework. While this thesis validated the digital approximation paradigm within the conservative bounds of unstructured lattice-based structures using a single, area-efficient approximate operator (`add16u_0GN`) to demonstrate explicit sampler removal with minimal hardware overhead, future research could explore its application across a broader spectrum of post-quantum algorithms and standards, in particular structured ring and module variants like ML-KEM. Adapting the paradigm to these protocols would require analyzing how hardware approximations interact with structured polynomial arithmetic and modular reductions. Concurrently, future investigations could address the current profile limitations of a single fixed operator by exploring the statistical implications of utilizing cascaded combinations of multiple distinct approximate operators within deeper accumulation loops. By exploiting the additive property of error, the independent errors introduced by different operators applied during the matrix-vector accumulation process will convolve, a mathematical behavior that can be leveraged to shape the hardware-introduced distribution and achieve a higher fidelity match against diverse target distributions. Nevertheless, implementing a multi-operator strategy requires that the cumulative distribution be strictly controlled, as the unconstrained accumulation of hardware inaccuracies would increase the total error variance, exceed the strict error budget mandated by the cryptographic parameters, and trigger decryption failures that compromise system security. Therefore, developing a systematic method to combine these operators while rigorously bounding the worst-case error remains a primary objective for future research, establishing an advanced framework where dynamically routing data paths through a heterogeneous array of approximate components allows a single hardware core to safely adjust its internal error variance and statistical boundaries at runtime to satisfy diverse cryptographic security levels.

For a more mathematically-oriented future research path, a possible direction involves establishing a formal connection between the proposed AxLWE paradigm and the recent LWE-OD framework [HMRS26]. The objective of this trajectory is to formalize a security reduction that unifies the hardware-approximation error profiles introduced in this work under the same theoretical umbrella as other output-dependent learning instances. By mapping the algorithmic inaccuracies of approximate data paths directly to the generalized noise definitions of LWE-OD, the AxLWE paradigm could formally inherit the established computational hardness assumptions of standard lattice-based cryptography. This theoretical bridge would eliminate the need for heuristic security arguments, providing rigorous mathematical guarantees that validate the long-term cryptographic integrity of approximate hardware implementations.

Dissemination

- **Poster Presentation: D. S. Crescenzo**, E. Valea, A. Bosio. *Randomness Generation in PQC and New Implementation Strategies Based on Approximate Computing*. 18^{e} Colloque du GDR SoC^{2}, June 2024, Toulouse, France.
- **Oral Presentation: D. S. Crescenzo**, E. Valea, A. Bosio. *AxC for Randomness Generation in Post-Quantum Cryptography Algorithms*. Approximate Computing Workshop, July 2024, Lyon, France.
- **Oral Presentation: D. S. Crescenzo**, E. Valea, A. Bosio. *Hardware Accelerator for FIPS 202 Hash Functions in Post-Quantum Ready SoCs*. PQ-TLS Project Meeting, December 2024, Paris, France.
- **Poster Presentation: D. S. Crescenzo**, E. Valea, A. Bosio. *Accelerating LWE-Based Post-Quantum Cryptography with Approximate Computing*. 19^{e} Colloque du GDR SoC^{2}, June 2025, Lorient, France.
- **Preprint: D. S. Crescenzo**, E. Valea, A. Bosio. *Accelerating LWE-Based Post-Quantum Cryptography with Approximate Computing*. IACR Cryptology ePrint Archive, Report 2025/1927, October 2025.
- **Poster Presentation: D. S. Crescenzo**, E. Valea, A. Bosio. *Replacing Explicit Error Generation with Approximate Computing in LWE-Based Schemes*. CEA-List Days Conference, June 2026, Palaiseau, France.
- **Poster Presentation: D. S. Crescenzo**, E. Valea, A. Bosio. *Replacing Explicit Error Generation With Approximate Computing in LWE-Based Schemes*. 20^{e} Colloque du GDR SoC^{2}, June 2026, Marseille, France.

Publications

- **Conference Paper: D. S. Crescenzo**, R. Carrera Rodriguez, R. Alidori, F. Bruguier, E. Valea, P. Benoit, A. Bosio. *Hardware Accelerator for FIPS 202 Hash Functions in Post-Quantum Ready SoCs*. 2024 IEEE 30th International Symposium on On-Line Testing and Robust System Design (IOLTS), July 2024, Rennes, France. DOI: 10.1109/IOLTS60994.2024.10616067
- **Journal Article:** S. Di Matteo, **D. S. Crescenzo**, R. Carrera Rodriguez, E. Valea, F. Bruguier, P. Benoit. *Accelerating first-order secure ML-KEM with masked SHA-3: cost, randomness, and security evaluation*. IEEE Access, October 2025. DOI: 10.1109/ACCESS.2025.3616775

- **Journal Article: D. S. Crescenzo**, E. Valea, A. Bosio. *Replacing Explicit Error Generation With Approximate Computing in LWE-Based Schemes*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), Special Issue on ESWEEK/CASES 2026, 2026 (to appear). DOI: 10.1109/TCAD.2026.3725523

List of Figures

2.1 Hash Function 11

2.2 eXtendable-Output Function 11

2.3 Symmetric-Key Encryption [Fre] 12

2.4 Asymmetric-Key Encryption [Fre] 12

2.5 Key Encapsulation Mechanism (KEM) [GT] 13

2.6 Lattice graphical examples showing an unstructured, staggered point distribution (left) alongside a highly regular, dense orthogonal grid structure (right). 15

2.7 Geometric comparison of a good lattice basis (left), consisting of short, orthogonal vectors, versus a bad lattice basis (right) for the same point set, characterized by long, highly non-orthogonal, and skewed vectors. 16

2.8 Matrix-vector parameter composition of the Learning With Errors problem framework. 19

3.1 2-bit unrolled Bivium architectural schematic. 31

3.2 Structural operational mapping of the Keccak sponge framework highlighting the absorbing and squeezing phases. 32

3.3 Inner matrix product generation chain. 34

3.4 Operational data flow of a Key Encapsulation Mechanism (KEM) framework during key establishment. 40

4.1 First-order DOM multiplier [GMK16] 59

4.2 Schematic view of the first-order masked ML-KEM.Decaps function. Red boxes indicate FIPS 202 functions in masked form, while green boxes in unmasked form. 63

4.3 High-level architecture of the masked Keccak core [ZSS+21]. d is equal to the masking order. The input passes through diffusion (θ), registers, and the permutation steps ($\pi \circ \rho$) for shuffling/rotation and $\iota \circ \chi$ for nonlinear mixing. Refreshing ensures the state remains securely masked against side-channel leakage. 64

4.4 Architecture of the proposed masked SHA-3 accelerator. 66

4.5 Hardware architecture of the Randomness Dispatcher. The design employs 25 parallel instances of Trivium or Bivium, each initialized with independent IV and key inputs. These parallel cores generate the required 1600-bit randomness output, controlled by the FSM and status register. 69

4.6 Integration of the Masked SHA-3 Accelerator and the Randomness Dispatcher in the target SoC. 70

4.7 Synthesis results and design space exploration of the proposed accelerators on a 22nm FD-SOI standard cell technology. 72

4.8 Leakage assessment of the masked SHA-3 accelerator during one round of the Keccak function. Horizontal red lines correspond to the threshold of $|t| = 4.5$ above which we are in the presence of information leakage. (a) t-statistic obtained with 20k traces with the SHA-3 accelerator used in the unmasked mode of operation. (b) t-statistic obtained with 20k traces with the SHA-3 accelerator used in masked version with the Randomness Dispatcher OFF. (c) t-statistic obtained with 1M traces with the SHA-3 accelerator used in masked version with Randomness Dispatcher (Trivium version) ON. 74

4.9 In (a) the t-statistic obtained with 1M traces with the SHA-3 accelerator used in masked version with Randomness Dispatcher (Trivium version) ON. In (b) the t-statistic obtained with 1M traces with the SHA-3 accelerator used in masked version with Randomness Dispatcher (Bivium version) ON. 74

4.10 On the left side, the cycle count and percentage of the full-SW implementation of KEM.Decaps are divided by family of primitives (SHA-3 functions, polynomial computation, and other operations). On the right side, the SW implementation accelerated by the proposed masked SHA-3 accelerator is shown. 75

5.1 Computational pipeline for the Learning Parity with Noise (LPN) sample generation framework. 87

5.2 Computational layout of the Learning with Physical Parity Noise (LPPN) inexact inner product framework. 88

5.3 Inner matrix product generation chain. 90

5.4 Reduced inner matrix product generation chain with approximate additions. . . 90

5.5 Adder selection methodology: integrating hardware constraints, behavioral simulation, and statistical mapping to identify the optimal AxADD. 94

5.6 Comparative error distribution plots of evaluated EvoApproxLib adders, arranged vertically in a single-column layout and ordered by suitability from top to bottom (continued on next page). Each subfigure displays the Probability on the y-axis against the Error Magnitude on the x-axis. The most suitable candidates exhibit centralized and symmetric profiles, whereas lower-ranked designs display increasing levels of statistical divergence from the target baseline. The bar charts represent the empirical intrinsic error distribution of the hardware, labeled with the respective Worst Case Absolute Error (WCE), while the solid red lines represent the analytical Reference Gaussian distribution from the FrodoKEM parameters. 97

5.6 Comparative error distribution plots of evaluated EvoApproxLib adders (continued on next page). 98

5.6 Comparative error distribution plots of evaluated EvoApproxLib adders (concluded). Annotation boxes provide the specific Hellinger distance, KL divergence, KS statistic, and Chi-Squared values computed for each operator. 99

5.7 Gaussian fit comparison between the intrinsic error distribution of the `add16u_0GN` adder and the FrodoKEM-640 reference profile. 101

5.8 Modulo-q quadrant mapping for the FrodoKEM-640 bit encoding scheme ($B = 2$ bits per symbol). The space of integers modulo q ($\mathbb{Z}_q$) is partitioned into four distinct intervals centered around multiples of $q/4$ to encode the binary pairs `00`, `01`, `10`, and `11`. 102

5.9 Baseline hardware architecture for LWE acceleration. The design features a SHA-3 Module for pseudo-random data generation and a MAC module for inner product accumulation. The right-most dashed block (Gaussian sampler and error Vector E memory) represents the explicit error generation logic. 107

5.10 Hardware schematic of a CDF-based Gaussian sampler, featuring a control unit, internal CDF table, and interval comparison logic. 109

5.11 Hardware schematic of an individual MAC element, featuring a multiplier, an adder, and an enabled accumulator register with feedback routing. 110

5.12 Proposed hardware architecture of the AxLWE accelerator. Error introduction is performed via MAC^{Ax} units and the Ax Randomizer. 111

5.13 Hardware schematic of the Approximate Multiply-Accumulate unit. The architecture features a 16-bit Multiplier followed by a selectable accumulation stage. A multiplexer facilitates switching between a standard exact Adder and the approximate `addu16_0GN` operator, enabling either conventional accumulation or approximate computation for implicit error introduction within the same hardware footprint. 112

5.14 Detailed area breakdown comparison (in kilo-gate equivalent kGe) of the LWE-cdf and AxLWE accelerators. 114

List of Tables

3.1 Parameters of FIPS 202 Functions (bits) 32

3.2 Sizes (in bytes) of keys and ciphertexts across FrodoKEM variants [A+20] . . . 42

3.3 Sizes (in bytes) of keys and ciphertexts across ML-KEM variants [NIS24b] . . 46

4.1 Synthesis Results of the SHA-3 Accelerator and the Randomness Dispatcher on an Artix-7 FPGA 71

4.2 Execution Time (in Clock Cycles) of the Masked `KEM.Decaps` Function Using Full Software and a SHA-3 Hardware Accelerator 75

5.1 Statistical distance, goodness-of-fit tests, and 22nm PPA metrics at maximum frequency for candidate approximate adders against the FrodoKEM-640 reference distribution, ordered by suitability. 100

5.2 Comparison of Decryption Failure Rates (DFR) for AxFrodoKEM vs. reference values. 104

5.3 AxFrodoKEM-640 execution time breakdown in kilo clock cycles (kcc) and pseudorandom error-generation byte requirements for the reference and proposed approximate architectures. 105

5.4 Performance and Efficiency Comparison of LWE and AxLWE HW Implementations 113

5.5 State-of-the-Art Comparison on Approximate LWE-based Schemes 115

References

[A+20] Erdem Alkim et al. FrodoKEM: Learning With Errors Key Encapsulation Algorithm Specifications and Supporting Documentation. Technical report, NIST PQC Standardization Round 3 Submission, September 2020.

[A+22] Gorjan Alagic et al. Status report on the third round of the NIST post-quantum cryptography standardization process. NIST Interagency/Internal Report (NISTIR) 8413, National Institute of Standards and Technology (NIST), 2022.

[ABB+22] N. Aragon, P. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J. C. Deneuville, P. Gaborit, S. Ghosh, S. Gueron, T. Güneysu, et al. BIKE: Bit Flipping Key Encapsulation. Nist post-quantum cryptography standardization submission, National Institute of Standards and Technology, 2022.

[ABD+19] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, Damien Stehlé, et al. Crystals-kyber algorithm specifications and supporting documentation. *NIST PQC Round*, 2(4):1–43, 2019.

[ABP+18] Victor Arribas, Begül Bilgin, George Petrides, Svetla Nikova, and Vincent Rijmen. Rhythmic keccak: Sca security and low latency in hw. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 269–290, 2018.

[APS15] Martin R Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Math. Cryptol.*, 9(3):169–203, 2015.

[B+21] Davide Bellizia et al. Learning Parity with Physical Noise: Imperfections, Reductions and FPGA Prototype. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(3):390–417, Jul. 2021.

[B+22] Davide Bellizia et al. When Bad News Become Good News: Towards Usable Instances of Learning with Physical Errors. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 1–24, 2022.

[Bab86] László Babai. On Lovász' lattice reduction and the nearest point problem. *Combinatorica*, 6(1):1–13, 1986.

[BCG+13] Johannes Buchmann, Daniel Cabarcas, Florian Göpfert, Andreas Hülsing, and Patrick Weiden. Discrete ziggurat: A time-memory trade-off for sampling from a gaussian distribution over the integers. In *International Conference on Selected Areas in Cryptography*, pages 402–417. Springer, 2013.

[BCL+17] Daniel J Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, et al. Classic McEliece: conservative code-based cryptography. *NIST submissions*, 1(1):1–25, 2017.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *International workshop on cryptographic hardware and embedded systems*, pages 16–29. Springer, 2004.

[BDN+13] Begül Bilgin, Joan Daemen, Ventzislav Nikov, Svetla Nikova, Vincent Rijmen, and Gilles Van Assche. Efficient and first-order dpa resistant implementations of keccak. In *International Conference on Smart Card Research and Advanced Applications*, pages 187–199. Springer, 2013.

[BDPVA10] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Building power analysis resistant implementations of keccak. In *Second SHA-3 candidate conference*, volume 142, 2010.

[BDVA11] Guido Bertoni, Michaël Daemen, Joan Peeters, and Gilles Van Assche. *The Keccak Reference*. Keccak Team Submission to NIST, 2011.

[BHS18] Song Bian, Masayuki Hiromoto, and Takashi Sato. Dwe: Decrypting learning with errors with errors. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

[BK15] Elaine Barker and John Kelsey. Recommendation for random number generation using deterministic random bit generators. NIST Special Publication 800-90A Rev. 1, National Institute of Standards and Technology (NIST), June 2015.

[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.

[BLBF11] Mathieu Baudet, David Lubicz, Nathalie Bochard, and Viktor Fischer. A closer look at security limits of oscillator-based TRNGs. *IEEE Transactions on Computers*, 60(12):1774–1783, 2011.

[BMS22] Alberto Bosio, Daniel Ménard, and Olivier Sentieys. *Approximate computing techniques*. Springer, 2022.

[BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 719–737. Springer, 2012.

[CCRR13] Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference*, pages 1–9, 2013.

[CFAF13] Abdelkarim Cherkaoui, Viktor Fischer, Alain Aubert, and Laurent Fesquet. A self-timed ring based true random number generator. In *2013 IEEE 19th international symposium on asynchronous circuits and systems*, pages 99–106. IEEE, 2013.

[CFFA13] Abdelkarim Cherkaoui, Viktor Fischer, Laurent Fesquet, and Alain Aubert. A very high speed true random number generator with entropy assessment. In *Cryptographic Hardware and Embedded Systems-CHES 2013: 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings 15*, pages 179–196. Springer, 2013.

[CGF21] Ana Covic, Fatemeh Ganji, and Domenic Forte. Circuit masking: from theory to standardization, a comprehensive survey for hardware security researchers and practitioners. *arXiv preprint arXiv:2106.12714*, 2021.

[CMM+23] Gaëtan Cassiers, Loïc Masure, Charles Momin, Thorben Moos, Amir Moradi, and François-Xavier Standaert. Randomness generation for secure hardware masking-unrolled trivium to the rescue. *Cryptology ePrint Archive*, 2023.

[CRA+24] Diamante Simone Crescenzo, Rafael Carrera Rodriguez, Riccardo Alidori, Florent Bruguier, Emanuele Valea, Pascal Benoit, and Alberto Bosio. Hardware accelerator for fips 202 hash functions in post-quantum ready socs. In *2024 IEEE 30th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–6. IEEE, 2024.

[CRPS26] Lorenzo Casalino, Maria Méndez Real, Jean-Christophe Prévotet, and Rubén Salvador. Double strike: Breaking approximation-based side-channel countermeasures for dnns. In *2026 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 285–296. IEEE, 2026.

[DC06] Christophe De Canniere. Trivium: A stream cipher construction inspired by block cipher design principles. In *International Conference on Information Security*, pages 171–186. Springer, 2006.

[DDLL13] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In *Annual Cryptology Conference*, pages 40–56. Springer, 2013.

[DG14] Nagarjun C Dwarakanath and Steven D Galbraith. Sampling from discrete gaussians for lattice-based cryptography on a constrained device. *Applicable Algebra in Engineering, Communication and Computing*, 25(3):159–180, 2014.

[DH96] RSA Diffie-Hellman. Timing attacks on implementations of. In *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18–22, 1996, Proceedings*, volume 1109, page 104. Springer, 1996.

[DH22] Whitfield Diffie and Martin E. Hellman. *New Directions in Cryptography*, page 365–390. Association for Computing Machinery, New York, NY, USA, 1 edition, 2022.

[DP02] Brian A Davey and Hilary A Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.

[DPAM02] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[DVV18] Jan-Pieter D'Anvers, Frederik Vercauteren, and Ingrid Verbauwhede. On the Impact of Decryption Failures on the Security of LWE/LWR Based Schemes. In *Post-Quantum Cryptography – PQCrypto 2018*, volume 10786 of *Lecture Notes in Computer Science*, pages 91–112. Springer, 2018.

[Dwo01] M. Dworkin. Recommendation for Block Cipher Modes of Operation: Methods and Techniques. NIST Special Publication (SP) 800-38A, National Institute of Standards and Technology, 2001.

[Fre] SBTS2018 Smashicons Freepik.

[FVBR+22] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked accelerators and instruction set extensions for post-quantum cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 414–460, 2022.

[GGH97] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction problems. In *Annual International Cryptology Conference*, pages 112–131. Springer, 1997.

[GGJR+11] Benjamin Jun Gilbert Goodwill, Josh Jaffe, Pankaj Rohatgi, et al. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011.

[Glo24] GlobalFoundries. *GlobalFoundries Memory Explorer*, 2024. Available via the GlobalFoundries Customer Portal; access restricted under NDA.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. *Cryptology ePrint Archive*, 2016.

[Gro96] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.

[GT] Sofia Celi Goutam Tamvada. Deep dive into a post-quantum key encapsulation algorithm.

[HJ16] Yupu Hu and Huiwen Jia. Cryptanalysis of ggh map. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 537–565. Springer, 2016.

[HKL+22] Daniel Heinz, Matthias J Kannwischer, Georg Land, Thomas Pöppelmann, Peter Schwabe, and Amber Sprenkels. First-order masked kyber on arm cortex-m4. *Cryptology ePrint Archive*, 2022.

[HKR+16] James Howe, Ayesha Khalid, Ciara Rafferty, Francesco Regazzoni, and Máire O'Neill. On practical discrete gaussian samplers for lattice-based cryptography. *IEEE Transactions on Computers*, 67(3):322–334, 2016.

[HMRS26] Clément Hoffmann, Pierrick Méaux, Mélissa Rossi, and François-Xavier Standaert. Learning with errors with output dependencies: Lwe, lwr, and physical learning problems under the same umbrella. In *International Conference on Post-Quantum Cryptography*, pages 287–318. Springer, 2026.

[HMV16] Radek Hrbacek, Vojtech Mrazek, and Zdenek Vasicek. Automatic design of approximate circuits by means of multi-objective evolutionary algorithms. In *2016 International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6. IEEE, 2016.

[HO13] Jie Han and Michael Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *2013 18th IEEE European test symposium (ETS)*, pages 1–6. IEEE, 2013.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. Ntru: A ring-based public key cryptosystem. In *International algorithmic number theory symposium*, pages 267–288. Springer, 1998.

[Inc20] Synopsys Inc. *Design Compiler*, 2020. Available at https://www.synopsys.com/.

[JGCS24] Arpan Jati, Naina Gupta, Anupam Chattopadhyay, and Somitra Kumar Sanadhya. A configurable crystals-kyber hardware implementation with side-channel protection. *ACM transactions on embedded computing systems*, 23(2):1–25, 2024.

[K+20] Dina Kamel et al. Learning with Physical Noise or Errors. *IEEE Transactions on Dependable and Secure Computing*, 17(5):957–971, 2020.

[KBK+20] Dur E Shahwar Kundi, Song Bian, Ayesha Khalid, Chenghua Wang, Máire O'Neill, and Weiqiang Liu. Axmm: Area and power efficient approximate modular multiplier for r-lwe cryptosystem. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2020.

[KBW+21] Ayesha Khalid, Song Bian, Chenghua Wang, Máire O'Neill, Weiqiang Liu, et al. Axrlwe: A multilevel approximate ring-lwe co-processor for lightweight iot applications. *IEEE Internet of Things Journal*, 9(13):10492–10501, 2021.

[KJJR11] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011.

[KLL20] Liang Kong, Shuguo Li, and Ruirui Liu. High-performance constant-time discrete gaussian sampling. *IEEE Transactions on Computers*, 70(7):1019–1033, 2020.

[KM22] David Knichel and Amir Moradi. Composable gadgets with reused fresh masks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(3):114–140, 2022.

[KNAH22] Tendayi Kamucheka, Alexander Nelson, David Andrews, and Miaoqing Huang. A masked pure-hardware implementation of kyber cryptographic algorithm. In *2022 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–1. IEEE, 2022.

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[KRR+18] Angshuman Karmakar, Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede. Constant-time discrete gaussian sampling. *IEEE Transactions on Computers*, 67(11):1561–1571, 2018.

[KRVV19] Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Pushing the speed limit of constant-time discrete gaussian sampling. a case study on the falcon signature scheme. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[KSM22] David Knichel, Pascal Sasdrich, and Amir Moradi. Generic hardware private circuits: Towards automated generation of composable secure gadgets. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 323–344, 2022.

[KY76] Donald E. Knuth and Andrew C. Yao. The complexity of nonuniform random number generation. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 357–428. Academic Press, San Diego, CA, USA, 1976.

[Lam79] Leslie Lamport. Constructing digital signatures from a one-way function. Technical report, Technical Report CSL-98, SRI International Palo Alto, 1979.

[LBS22] Itamar Levi, Davide Bellizia, and François-Xavier Standaert. Tight-es-trng: Improved construction and robustness analysis. *SN Computer Science*, 3(4):321, 2022.

[LG20] Michael X Lyons and Kris Gaj. Sampling from discrete distributions in combinational hardware with application to post-quantum cryptography. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France*, 2020.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 1–23. Springer, 2010.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015.

[M+17] Vojtech Mrazek et al. EvoApprox8b: Library of approximate adders and multipliers for circuit design. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 258–261. IEEE, 2017.

[MAB+18] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loıc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, and I Bourges. Hamming Quasi-Cyclic (HQC). *NIST PQC Round*, 2(4):13, 2018.

[McE78] Robert J McEliece. A public-key cryptosystem based on algebraic. *Coding Thv*, 4244(1978):114–116, 1978.

[Mer79] Ralph Charles Merkle. *Secrecy, authentication, and public key systems.* Stanford university, 1979.

[Mer89] Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer, 1989.

[MG02] Daniele Micciancio and Shafi Goldwasser. *Complexity of lattice problems: a cryptographic perspective*, volume 671. Springer Science & Business Media, 2002.

[MI88] Tsutomu Matsumoto and Hideki Imai. Public quadratic polynomial-tuples for efficient signature-verification and message-encryption. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 419–453. Springer, 1988.

[Mil85] Victor S Miller. Use of elliptic curves in cryptography. In *Conference on the theory and application of cryptographic techniques*, pages 417–426. Springer, 1985.

[Mit16] Sparsh Mittal. A survey of techniques for approximate computing. *ACM Computing Surveys*, 48(4):1–33, 2016.

[MKDH15] Michael Muehlberghuber, Thomas Korak, Philipp Dunst, and Michael Hutter. Towards evaluating dpa countermeasures for keccak k1012eccak on a real asic. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 222–236. Springer, 2015.

[MP13] D. Micciancio and C. Peikert. Hardness of SIS and LWE with small parameters. In *Proc. of SODA*, pages 1105–1118, 2013.

[N+19] Hamid Nejatollahi et al. Post-quantum lattice-based cryptography implementations: A survey. *ACM Computing Surveys (CSUR)*, 51(6):1–41, 2019.

[Nat13] National Institute of Standards and Technology. Digital Signature Standard (DSS). Federal Information Processing Standards Publication (FIPS PUB) 186-4, U.S. Department of Commerce, July 2013.

[Nat15a] National Institute of Standards and Technology. Secure hash standard (SHS). Technical Report 180-4, National Institute of Standards and Technology (NIST), 2015.

[Nat15b] National Institute of Standards and Technology. SHA-3 standard: Permutation-based hash and extendable-output functions. Federal Information Processing Standards Publication (FIPS PUB) 202, National Institute of Standards and Technology (NIST), Gaithersburg, MD, August 2015.

[Nat16] National Institute of Standards and Technology. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. Technical report, NIST, December 2016.

[NF24] National Cybersecurity Agency of France (ANSSI) and Federal Office for Information Security (BSI). Securing tomorrow, today: Transitioning to post-quantum cryptography. Joint statement, ANSSI and BSI, November 2024.

[NIS24a] NIST. Module-lattice-based digital signature standard. Technical Report 204, National Institute of Standards and Technology, August 2024.

[NIS24b] NIST. Module-Lattice-Based Key-Encapsulation Mechanism Standard. Technical Report FIPS PUB 203, National Institute of Standards and Technology, August 2024.

[O+22] Ruth O'Connor et al. Better Security Estimates for Approximate, IoT-Friendly R-LWE Cryptosystems. In *IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pages 611–615, 2022.

[OANWO20] Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves, and Zooko Wilcox-O'Hearn. BLAKE3: One function, fast everywhere. Specification paper, available online, 2020.

[oST] National Institute of Standards and Technology. PQC - Round 3 Submissions.

[oST24] National Institute of Standards and Technology, 2024. NIST. Special Publication 800-227: Recommendations for key-encapsulation mechanisms.

[Pat96] Jacques Patarin. Hidden fields equations (hfe) and isomorphisms of polynomials (ip): Two new families of asymmetric algorithms. In *International conference on the theory and applications of cryptographic techniques*, pages 33–48. Springer, 1996.

[Pei10] Chris Peikert. An efficient and parallel gaussian sampler for lattices. In *Annual Cryptology Conference*, pages 80–97. Springer, 2010.

[Pei16] Chris Peikert. A decade of lattice cryptography. *Foundations and TrendsŴ in Theoretical Computer Science*, 10(4):283–424, 2016.

[Rad] Havard Raddum. Cryptanalytic results on trivium. estream, ecrypt stream cipher project, report 2006/039, 2006.

[Rad06] Håvard Raddum. An analysis of Trivium. Report 2006/056, IACR Cryptology ePrint Archive, 2006.

[RBN+15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In *Annual Cryptology Conference*, pages 764–783. Springer, 2015.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.

[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.

[SBK+17] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1. In *Annual international cryptology conference*, pages 570–596. Springer, 2017.

[SC19] Against Combined Side-Channel. Trade-offs in protecting keccak against combined side-channel and fault attacks. In *Constructive Side-Channel Analysis and Secure Design: 10th International Workshop, COSADE 2019, Darmstadt, Germany, April 3–5, 2019, Proceedings*, page 285. Springer, 2019.

[Sem20] Semiconductor Research Corporation (SRC) and Semiconductor Industry Association (SIA). The decadal plan for semiconductors: Full report. Technical report, Semiconductor Research Corporation, 2020.

[Sho94] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[SM16] Tobias Schneider and Amir Moradi. Leakage assessment methodology: Extended version. *Journal of Cryptographic Engineering*, 6(2):85–99, 2016.

[SM21] Aein Rezaei Shahmirzadi and Amir Moradi. Second-order sca security with almost no fresh randomness. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 708–755, 2021.

[Ste18] Douglas Stebila. Introduction to post-quantum cryptography and learning with errors, 6 2018.

[TBK+18] Meltem Sönmez Turan, Elaine Barker, John Kelsey, Kerry McKay, Mary Baish, and Mike Boyle. Recommendation for the entropy sources used for random bit generation. NIST Special Publication 800-90B, National Institute of Standards and Technology (NIST), January 2018.

[The26] The FrodoKEM Team. PQCrypto-LWEKE: FrodoKEM reference and optimized implementations. https://github.com/microsoft/PQCrypto-LWEKE, 2026. GitHub Repository.

[VCRR15] Swagath Venkataramani, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Approximate computing and the quest for computing efficiency. In *Proceedings of the 52nd Annual Design Automation Conference (DAC)*, pages 1–6, 2015.

[VL03] Vesselin Vassilev and Dimitris Loukas. Silicon true random number generators and environmental susceptibility. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 412–415. IEEE, 2003.

[VN+63] John Von Neumann et al. Various techniques used in connection with random digits. *John von Neumann, Collected Works*, 5:768–770, 1963.

[WT08] Knut Wold and Chik How Tan. Analysis and enhancement of random number generator in FPGA based on oscillator rings. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 385–390. IEEE, 2008.

[WY05] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 19–35. Springer, 2005.

[YRG+18] Bohan Yang, Vladimir Rožic, Miloš Grujic, Nele Mentens, and Ingrid Verbauwhede. Es-trng: A high-throughput, low-area true random number generator based on edge sampling. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 267–292, 2018.

[ZBK22] Behrouz Zolfaghari, Khodakhast Bibak, and Takeshi Koshiba. The odyssey of entropy: Cryptography. *Entropy*, 24(2):266, 2022.

[ZSS+21] Sara Zarei, Aein Rezaei Shahmirzadi, Hadi Soleimany, Raziyeh Salarifard, and Amir Moradi. Low-latency keccak at any arbitrary order. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 388–411, 2021.

Note on AI Assistance: *The author acknowledges the use of Gemini AI for text revision, proofreading advice, and language optimization during the final writing phase of this manuscript.*

List of Acronyms

AES Advanced Encryption Standard

ANSSI French National Agency for the Security of Information Systems

AxC Approximate Computing

AxLWE Approximate Learning With Errors

BSI German Federal Office for Information Security

CBD Centered Binomial Distribution

CDF Cumulative Distribution Function

CDT Cumulative Distribution Table

CGP Cartesian Genetic Programming

CPA Correlation Power Analysis

CU Control Unit

CVP Closest Vector Problem

DDG Discrete Distribution Generating

DES Data Encryption Standard

DFR Decryption Failure Rate

DOM Domain-Oriented Masking

DPA Differential Power Analysis

DRUM Dynamic Range Unbiased Multiplier

DSA Digital Signature Algorithm

DSS Digital Signature Standard

DWE Decrypting Learning with Errors

ECC Elliptic Curve Cryptography

EMA Electromagnetic Analysis

FHE Fully Homomorphic Encryption

FIPS Federal Information Processing Standard

FSM Finite State Machine

GS Gaussian Sampling

HQC Hamming Quasi-Cyclic

IND-CCA Indistinguishability under Chosen Ciphertext Attack

IND-CPA Indistinguishability under Chosen Plaintext Attack

KAT Known Answer Test

KEM Key Encapsulation Mechanism

LFSR Linear Feedback Shift Register

LPN Learning Parity with Noise

LPPN Learning Parity with Physical Noise

LWE Learning With Errors

LWE-OD Learning With Errors with Output Dependencies

LWPE Learning With Physical Errors

LWR Learning with Rounding

MAC Multiply and ACcumulate

MD5 Message Digest 5

ML-DSA Module-Lattice-Based Digital Signature Algorithm

ML-KEM Module-Lattice-Based Key-Encapsulation Mechanism

MLWE Module-LWE

NIST National Institute of Standards and Technology

NLFSR Non-Linear Feedback Shift Register

NTT Number-Theoretic Transform

PKE Public Key Encryption

PQC Post-Quantum Cryptography

PRNG Pseudo-Random Number Generator

RLWE Ring-LWE

RNG Random Number Generator

RSA Rivest–Shamir–Adleman

SCA Side-Channel Analysis

SHA Secure Hashing Algorithm

SIVP Shortest Independent Vectors Problem

SoC System-on-Chip

SVP Shortest Vector Problem

TI Threshold Implementations

TRNG True Random Number Generator

TVLA Test Vector Leakage Assessment

VOS Voltage-Frequency Over-Scaling

XOF eXtendable-Output Function

XOR eXclusive-OR

Appendix A

Source Code for Approximate Adders Characterization

Listing A.1 C implementation for the behavioral simulation and characterization of an approximate adder operator.

```c
/*****************************************************************************
 * Behavioral characterization of the absolute error distribution introduced
 * by a digital approximate adder.
 *****************************************************************************/

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <evoApprox.h>
#include <time.h>

#define GAUSS_LIM 10      /* Defined boundary support for target error distribution */
#define N_EXP     5000    /* Number of random simulation trials */

/* Select target approximate adder template from the hardware library */
#define AxADD     add16u_0GN

int main() {
    /* Open file pointers to save characterization statistics */
    FILE *fptr;
    fptr = fopen("err_dist.txt", "w");
    if (fptr == NULL) {
        printf("Error opening distribution log file.\n");
        exit(1);
    }

    FILE *fptr1;
    fptr1 = fopen("err_samp.txt", "w");
    if (fptr1 == NULL) {
```

```
30          printf("Error opening sample log file.\n");
31          exit(1);
32      }
33
34      /* Seed pseudo-random generator with system time for uniform stimuli generation */
35      srand(time(NULL));
36
37      uint16_t op1, op2;
38      uint64_t res;
39      uint64_t res_ax;
40      int16_t abs_err;
41
42      /* Initialize the histogram array centered around the zero-error offset */
43      int distribution[2 * GAUSS_LIM + 1] = { 0 };
44      int i;
45
46      /* Simulation loop: expose the operator to uniformly distributed random inputs */
47      for (i = 0; i < N_EXP; i++) {
48          op1 = rand() % 0xFFFF;
49          op2 = rand() % 0xFFFF;
50
51          res = op1 + op2;                            /* Exact arithmetic baseline */
52          /* Approximate arithmetic execution */
53          res_ax = AxADD((uint64_t)op1, (uint64_t)op2);
54
55          abs_err = (int16_t)(res_ax - res);          /* Extract arithmetic deviation */
56
57          /* Verify if the observed deviation falls within the acceptable distribution
                support */
58          if (abs(abs_err) > GAUSS_LIM) {
59              printf("ERROR: gaussian distribution support is too small\n");
60              exit(1);
61          }
62
63          fprintf(fptr1, "%d ", abs_err);
64          /* Map error to the centered histogram */
65          distribution[GAUSS_LIM + abs_err]++;
66      }
67
68      /* Stream the compiled error distribution histogram to file */
69      for (i = 0; i < (2 * GAUSS_LIM + 1); i++) {
70          fprintf(fptr, "%d ", distribution[i]);
71      }
72
73      fclose(fptr);
74      fclose(fptr1);
75
76      return 0;
```

```
}
```

Listing A.2 Python script for the statistical evaluation, distance mapping, and plotting of approximate error distributions against a reference baseline.

```python
import numpy as np
import matplotlib.pyplot as plt
import argparse
import sys
import os
from scipy import stats
from scipy.spatial import distance

# --- Configuration for Paper-Quality Plots ---
FONT_LABEL = 49
FONT_LEGEND = 40
FONT_TICKS = 34
FONT_TITLE = 52
FONT_STATS = 40
LINE_WIDTH = 4
FIG_SIZE = (24, 14)

def get_scientific_stats(ax_samples, ref_samples, ax_pmf, ref_pmf, ax_counts,
    ref_counts):
    """
    Computes statistical distance metrics and goodness-of-fit tests
    between the approximate hardware noise and reference distribution.
    """
    # 1. Kolmogorov-Smirnov (KS) Test on raw sample vectors
    ks_stat, ks_pvalue = stats.ks_2samp(ax_samples, ref_samples)

    # 2. Hellinger Distance calculation
    h_dist = distance.euclidean(np.sqrt(ax_pmf), np.sqrt(ref_pmf)) / np.sqrt(2)

    # 3. Kullback-Leibler Divergence with epsilon offset to prevent log(0) conditions
    epsilon = 1e-10
    pk = ax_pmf + epsilon
    qk = ref_pmf + epsilon
    pk /= np.sum(pk)
    qk /= np.sum(qk)
    kl_div = stats.entropy(pk, qk)

    # 4. Chi-Squared Goodness-of-Fit Test
    total_obs = np.sum(ax_counts)
    f_exp = ref_pmf * total_obs

    # Filter out empty expected bins to maintain test stability
    valid_bins = f_exp > 0

```

```
    if np.any(valid_bins):
        f_obs_filtered = ax_counts[valid_bins]
        f_exp_filtered = f_exp[valid_bins]

        # Rescale expected counts to guarantee matching alignment sums
        f_exp_filtered = f_exp_filtered * (np.sum(f_obs_filtered) / np.sum(
            f_exp_filtered))
        chi2_stat, chi2_p = stats.chisquare(f_obs=f_obs_filtered, f_exp=f_exp_filtered)
    else:
        chi2_stat = float('inf')

    return h_dist, kl_div, ks_stat, chi2_stat

def load_data(filepath):
    try:
        if not os.path.exists(filepath):
            print(f"Error: File '{filepath}' not found.")
            sys.exit(1)
        with open(filepath, 'r') as f:
            data = f.read().split()
        return np.array(data, dtype=float)
    except Exception as e:
        print(f"Error loading {filepath}: {e}")
        sys.exit(1)

def main():
    parser = argparse.ArgumentParser(description="Compare Adder Error vs Reference
        Noise")
    parser.add_argument('--name', type=str, help='Name of the adder (e.g. add16u_0GX)')
    parser.add_argument('--ax', type=str, help='Path to AxAdder raw samples')
    parser.add_argument('--ref', type=str, default='generated_noise_640_5000samples.
        data', help='Path to reference noise')
    parser.add_argument('--output', type=str, help='Output filename for the plot')
    args = parser.parse_args()

    plot_dir = 'plots'
    if not os.path.exists(plot_dir):
        os.makedirs(plot_dir)

    if args.name:
        ax_path = args.ax if args.ax else f"err_samp_{args.name}.txt"
        default_out_name = f"comparison_plot_{args.name}.pdf"
        title_suffix = f"({args.name})"
    else:
        ax_path = args.ax if args.ax else "err_samp.txt"
        default_out_name = "comparison_plot.pdf"
        title_suffix = ""

```

```
    out_path = args.output if args.output else os.path.join(plot_dir, default_out_name)

    # Load raw execution traces
    ax_samples = load_data(ax_path)
    ref_samples = load_data(args.ref)

    max_err_ax = np.max(np.abs(ax_samples))
    max_err_ref = np.max(np.abs(ref_samples))
    limit = int(np.ceil(max(max_err_ax, max_err_ref)))

    bins = np.arange(-limit - 0.5, limit + 1.5, 1)
    x_axis = np.arange(-limit, limit + 1)

    # Compute discrete frequency counts
    ax_counts, _ = np.histogram(ax_samples, bins=bins)
    ref_counts, _ = np.histogram(ref_samples, bins=bins)

    # Normalize counts to formulate Probability Mass Functions (PMFs)
    ax_pmf = ax_counts / ax_counts.sum()
    ref_pmf = ref_counts / ref_counts.sum()

    # Extract statistical profiles
    hellinger, kl_div, ks_stat, chi2_stat = get_scientific_stats(
        ax_samples, ref_samples, ax_pmf, ref_pmf, ax_counts, ref_counts
    )
    wce_ax = np.max(np.abs(ax_samples))

    # --- Plot Generation ---
    plt.figure(figsize=FIG_SIZE)

    # Plot empirical approximate operator distribution
    plt.bar(x_axis, ax_pmf, alpha=0.6, label=f'{args.name if args.name else "AxAdder"}
        (WCE={int(wce_ax)})',
            color='tab:blue', width=0.8, edgecolor='black', linewidth=2)

    # Plot target analytical reference distribution step
    plt.step(x_axis, ref_pmf, where='mid', label='Reference Gaussian',
             color='tab:red', linewidth=LINE_WIDTH)

    plt.title(f'Error Distribution {title_suffix}', fontsize=FONT_TITLE, pad=25)
    plt.xlabel('Error Magnitude', fontsize=FONT_LABEL, labelpad=15)
    plt.ylabel('Probability', fontsize=FONT_LABEL, labelpad=15)
    plt.xticks(fontsize=FONT_TICKS)
    plt.yticks(fontsize=FONT_TICKS)

    # Inject overlay textbox containing computed distance metrics
    stats_text = (
        f"Hellinger: {hellinger:.4f}\n"
```

```
        f"KL Div:   {kl_div:.4f}\n"
        f"KS Stat:  {ks_stat:.4f}\n"
        f"Chi-Sq:   {chi2_stat:.1f}"
    )
    props = dict(boxstyle='round', facecolor='white', alpha=0.95, edgecolor='gray', linewidth=2)
    plt.gca().text(0.98, 0.95, stats_text, transform=plt.gca().transAxes, fontsize=FONT_STATS,
            verticalalignment='top', horizontalalignment='right', bbox=props, fontfamily='monospace')

    plt.legend(loc='upper left', fontsize=FONT_LEGEND, framealpha=0.95, edgecolor='black')
    plt.grid(axis='y', linestyle='--', alpha=0.5, linewidth=2)

    if limit > 20:
        step_size = max(1, limit // 8)
        plt.xticks(np.arange(-limit, limit+1, step=step_size), fontsize=FONT_TICKS)
    else:
        plt.xticks(x_axis, fontsize=FONT_TICKS)

    plt.tight_layout()
    plt.savefig(out_path, format='pdf', bbox_inches='tight')
    plt.close()

if __name__ == "__main__":
    main()
```

Appendix B

Source Code Samples for Approximate FrodoKEM Implementation

Listing B.1 Original FrodoKEM implementation of `frodo_mul_add_as_plus_e` highlighting the lines targeted for deletion or replacement (marked with -).

```
--- src/matrix_orig.c
+++ src/matrix_orig.c
@@ -1,37 +1,37 @@
-int frodo_mul_add_as_plus_e(uint16_t *out, const uint16_t *s, const uint16_t *e,
    const uint8_t *seed_A)
 { // Generate-and-multiply: generate matrix A (N x N) row-wise, multiply by s on the
    right.
   // Inputs: s, e (N x N_BAR)
-  // Output: out = A*s + e (N x N_BAR)
     int i, j, k;
     ALIGN_HEADER(32) int16_t a_row[4*PARAMS_N] ALIGN_FOOTER(32) = {0};

-    for (i = 0; i < (PARAMS_N*PARAMS_NBAR); i += 2) {
-        *((uint32_t*)&out[i]) = *((uint32_t*)&e[i]);
-    }
-
     uint8_t seed_A_separated[2 + BYTES_SEED_A];
     uint16_t* seed_A_origin = (uint16_t*)&seed_A_separated;
     memcpy(&seed_A_separated[2], seed_A, BYTES_SEED_A);
     for (i = 0; i < PARAMS_N; i += 4) {
         seed_A_origin[0] = UINT16_TO_LE(i + 0);
         shake128((unsigned char*)(a_row + 0*PARAMS_N),
                  (unsigned long long)(2*PARAMS_N), seed_A_separated,
                  2 + BYTES_SEED_A);
         seed_A_origin[0] = UINT16_TO_LE(i + 1);
         shake128((unsigned char*)(a_row + 1*PARAMS_N),
                  (unsigned long long)(2*PARAMS_N), seed_A_separated,
                  2 + BYTES_SEED_A);
         seed_A_origin[0] = UINT16_TO_LE(i + 2);
```

```
        shake128((unsigned char*)(a_row + 2*PARAMS_N),
                 (unsigned long long)(2*PARAMS_N), seed_A_separated,
                 2 + BYTES_SEED_A);
        seed_A_origin[0] = UINT16_TO_LE(i + 3);
        shake128((unsigned char*)(a_row + 3*PARAMS_N),
                 (unsigned long long)(2*PARAMS_N), seed_A_separated,
                 2 + BYTES_SEED_A);

        for (k = 0; k < 4 * PARAMS_N; k++) {
            a_row[k] = LE_TO_UINT16(a_row[k]);
        }
        for (k = 0; k < PARAMS_NBAR; k++) {
            uint16_t sum[4] = {0};
            // Matrix-vector multiplication
-           for (j = 0; j < PARAMS_N; j++) {
-               uint16_t sp = s[k*PARAMS_N + j];
-               // Go through four lines with same s
-               sum[0] += a_row[0*PARAMS_N + j] * sp;
+           // Matrix-vector multiplication
+           for (j = 0; j < PARAMS_N; j++) {
+               uint16_t sp = s[k*PARAMS_N + j];
+               // Go through four lines with same s
+               sum[0] += a_row[0*PARAMS_N + j] * sp;
                sum[1] += a_row[1*PARAMS_N + j] * sp;
                sum[2] += a_row[2*PARAMS_N + j] * sp;
                sum[3] += a_row[3*PARAMS_N + j] * sp;
            }
-           out[(i+0)*PARAMS_NBAR + k] += sum[0];
-           out[(i+2)*PARAMS_NBAR + k] += sum[2];
-           out[(i+1)*PARAMS_NBAR + k] += sum[1];
-           out[(i+3)*PARAMS_NBAR + k] += sum[3];
        }
    }

    return 1;
}
```

Listing B.2 Approximate variant of `frodo_mul_add_as_plus_e` explicitly highlighting the software optimizations and hardware operator injection (marked with +).

```
--- src/matrix_approx.c
+++ src/matrix_approx.c
@@ -1,48 +1,48 @@
+int frodo_mul_add_as_plus_e(uint16_t *out, const uint16_t *s, const uint8_t *seed_A)
 { // Generate-and-multiply: generate matrix A (N x N) row-wise, multiply by s on the
     right.
   // Inputs: s, e (N x N_BAR)
+  // Output: out = A*s (+ e) (N x N_BAR)
```

```
+  // Note: the error e is no more generated but introduced by means of an approximate
    adder
     int i, j, k;
     ALIGN_HEADER(32) int16_t a_row[4*PARAMS_N] ALIGN_FOOTER(32) = {0};

     uint8_t seed_A_separated[2 + BYTES_SEED_A];
     uint16_t* seed_A_origin = (uint16_t*)&seed_A_separated;
     memcpy(&seed_A_separated[2], seed_A, BYTES_SEED_A);
     for (i = 0; i < PARAMS_N; i += 4) {
+        // generate matrix A
         seed_A_origin[0] = UINT16_TO_LE(i + 0);
         shake128((unsigned char*)(a_row + 0*PARAMS_N),
                  (unsigned long long)(2*PARAMS_N), seed_A_separated,
                  2 + BYTES_SEED_A);
         seed_A_origin[0] = UINT16_TO_LE(i + 1);
         shake128((unsigned char*)(a_row + 1*PARAMS_N),
                  (unsigned long long)(2*PARAMS_N), seed_A_separated,
                  2 + BYTES_SEED_A);
         seed_A_origin[0] = UINT16_TO_LE(i + 2);
         shake128((unsigned char*)(a_row + 2*PARAMS_N),
                  (unsigned long long)(2*PARAMS_N), seed_A_separated,
                  2 + BYTES_SEED_A);
         seed_A_origin[0] = UINT16_TO_LE(i + 3);
         shake128((unsigned char*)(a_row + 3*PARAMS_N),
                  (unsigned long long)(2*PARAMS_N), seed_A_separated,
                  2 + BYTES_SEED_A);

         for (k = 0; k < 4 * PARAMS_N; k++) {
             a_row[k] = LE_TO_UINT16(a_row[k]);
         }
         for (k = 0; k < PARAMS_NBAR; k++) {
+            uint16_t arow_sp[4] = {0};
+            uint16_t sum[4] = {0};
+            uint16_t sp;
+            // Matrix-vector multiplication
+            for (j = 0; j < PARAMS_N-1; j++) {
+                sp = s[k*PARAMS_N + j];
+
+                // inner A*s elements product
                 // Go through four lines with same s
+                sum[0] += a_row[0*PARAMS_N + j] * sp;
+                sum[1] += a_row[1*PARAMS_N + j] * sp;
+                sum[2] += a_row[2*PARAMS_N + j] * sp;
+                sum[3] += a_row[3*PARAMS_N + j] * sp;
+            }
+
+            // last A*s elements product
+            sp = s[k*PARAMS_N + j];
```

```
+
+           arow_sp[0] = a_row[0*PARAMS_N + j] * sp;
+           arow_sp[1] = a_row[1*PARAMS_N + j] * sp;
+           arow_sp[2] = a_row[2*PARAMS_N + j] * sp;
+           arow_sp[3] = a_row[3*PARAMS_N + j] * sp;
+
+           // in the last addition, use approximate adder
+           // adds the last A*s elements product to the B element
+           //    with the approximate adder
+           out[(i+0)*PARAMS_NBAR + k] = AxADD((uint64_t)(sum[0]),
+                                              (uint64_t)arow_sp[0]);
+           out[(i+1)*PARAMS_NBAR + k] = AxADD((uint64_t)(sum[1]),
+                                              (uint64_t)arow_sp[1]);
+           out[(i+2)*PARAMS_NBAR + k] = AxADD((uint64_t)(sum[2]),
+                                              (uint64_t)arow_sp[2]);
+           out[(i+3)*PARAMS_NBAR + k] = AxADD((uint64_t)(sum[3]),
+                                              (uint64_t)arow_sp[3]);
        }
    }

    return 1;
}
```

Listing B.3 Original FrodoKEM implementation of `frodo_mul_add_sa_plus_e` highlighting the lines targeted for deletion or replacement (marked with -).

```
--- src/matrix_orig.c
+++ src/matrix_orig.c
@@ -1,43 +1,35 @@
-int frodo_mul_add_sa_plus_e(uint16_t *out, const uint16_t *s,
-                            uint16_t *e, const uint8_t *seed_A)
 { // Generate-and-multiply: generate matrix A (N x N) column-wise, multiply by s' on
     the left.
   // Inputs: s', e' (N_BAR x N)
-  // Output: out = s'*A + e' (N_BAR x N)
   // The matrix multiplication uses the row-wise blocking and packing (RWCF) approach
      described in: J.W. Bos, M. Ofner, J. Renes,
   // T. Schneider, C. van Vredendaal, "The Matrix Reloaded: Multiplication Strategies
      in FrodoKEM". https://eprint.iacr.org/2021/711
    int i, j, q, p;
    ALIGN_HEADER(32) uint16_t A[PARAMS_N*8] ALIGN_FOOTER(32) = {0};

    uint8_t seed_A_separated[2 + BYTES_SEED_A];
    uint16_t* seed_A_origin = (uint16_t*)&seed_A_separated;
    memcpy(&seed_A_separated[2], seed_A, BYTES_SEED_A);

    % Start matrix multiplication
    for (i = 0; i < PARAMS_N; i+=8) {
        seed_A_origin[0] = UINT16_TO_LE(i + 0);
```

```
        shake128((unsigned char*)(A + 0*PARAMS_N), (unsigned long long)(2*PARAMS_N),
                 seed_A_separated, 2 + BYTES_SEED_A);
        seed_A_origin[0] = UINT16_TO_LE(i + 1);
        shake128((unsigned char*)(A + 1*PARAMS_N), (unsigned long long)(2*PARAMS_N),
                 seed_A_separated, 2 + BYTES_SEED_A);
        seed_A_origin[0] = UINT16_TO_LE(i + 2);
        shake128((unsigned char*)(A + 2*PARAMS_N), (unsigned long long)(2*PARAMS_N),
                 seed_A_separated, 2 + BYTES_SEED_A);
        seed_A_origin[0] = UINT16_TO_LE(i + 3);
        shake128((unsigned char*)(A + 3*PARAMS_N), (unsigned long long)(2*PARAMS_N),
                 seed_A_separated, 2 + BYTES_SEED_A);
        seed_A_origin[0] = UINT16_TO_LE(i + 4);
        shake128((unsigned char*)(A + 4*PARAMS_N), (unsigned long long)(2*PARAMS_N),
                 seed_A_separated, 2 + BYTES_SEED_A);
        seed_A_origin[0] = UINT16_TO_LE(i + 5);
        shake128((unsigned char*)(A + 5*PARAMS_N), (unsigned long long)(2*PARAMS_N),
                 seed_A_separated, 2 + BYTES_SEED_A);
        seed_A_origin[0] = UINT16_TO_LE(i + 6);
        shake128((unsigned char*)(A + 6*PARAMS_N), (unsigned long long)(2*PARAMS_N),
                 seed_A_separated, 2 + BYTES_SEED_A);
        seed_A_origin[0] = UINT16_TO_LE(i + 7);
        shake128((unsigned char*)(A + 7*PARAMS_N), (unsigned long long)(2*PARAMS_N),
                 seed_A_separated, 2 + BYTES_SEED_A);

        for (j = 0; j < PARAMS_NBAR; j++) {
            uint16_t sum = 0;
            int16_t sp[8];
            for (p = 0; p < 8; p++) {
                sp[p] = s[j*PARAMS_N + i + p];
            }
            for (q = 0; q < PARAMS_N; q++) {
-               sum = e[j*PARAMS_N + q];
-               for (p = 0; p < 8; p++) {
-                   sum += sp[p] * A[p*PARAMS_N + q];
-               }
-               e[j*PARAMS_N + q] = sum;
            }
        }
    }
-   memcpy((unsigned char*)out, (unsigned char*)e, 2*PARAMS_N*PARAMS_NBAR);

    return 1;
}
```

Listing B.4 Approximate variant of `frodo_mul_add_sa_plus_e` explicitly highlighting the software optimizations and hardware operator injection (marked with +).

```
--- src/matrix_approx.c
+++ src/matrix_approx.c
```

```
@@ -1,46 +1,46 @@
+int frodo_mul_add_sa_plus_e(uint16_t *out, const uint16_t *s, const uint8_t *seed_A)
 { // Generate-and-multiply: generate matrix A (N x N) column-wise, multiply by s' on the left.
   // Inputs: s', e' (N_BAR x N)
+  // Output: out = s'*A (+ e') (N_BAR x N)
+  // Note: the error e' is no more generated but introduced by means of an approximate adder
   // The matrix multiplication uses the row-wise blocking and packing (RWCF) approach described in: J.W. Bos, M. Ofner, J. Renes,
   // T. Schneider, C. van Vredendaal, "The Matrix Reloaded: Multiplication Strategies in FrodoKEM". https://eprint.iacr.org/2021/711
     int i, j, q, p;
     ALIGN_HEADER(32) uint16_t A[PARAMS_N*8] ALIGN_FOOTER(32) = {0};

     uint8_t seed_A_separated[2 + BYTES_SEED_A];
     uint16_t* seed_A_origin = (uint16_t*)&seed_A_separated;
     memcpy(&seed_A_separated[2], seed_A, BYTES_SEED_A);

     % Start matrix multiplication
     for (i = 0; i < PARAMS_N; i+=8) {
         seed_A_origin[0] = UINT16_TO_LE(i + 0);
         shake128((unsigned char*)(A + 0*PARAMS_N), (unsigned long long)(2*PARAMS_N),
                  seed_A_separated, 2 + BYTES_SEED_A);
         seed_A_origin[0] = UINT16_TO_LE(i + 1);
         shake128((unsigned char*)(A + 1*PARAMS_N), (unsigned long long)(2*PARAMS_N),
                  seed_A_separated, 2 + BYTES_SEED_A);
         seed_A_origin[0] = UINT16_TO_LE(i + 2);
         shake128((unsigned char*)(A + 2*PARAMS_N), (unsigned long long)(2*PARAMS_N),
                  seed_A_separated, 2 + BYTES_SEED_A);
         seed_A_origin[0] = UINT16_TO_LE(i + 3);
         shake128((unsigned char*)(A + 3*PARAMS_N), (unsigned long long)(2*PARAMS_N),
                  seed_A_separated, 2 + BYTES_SEED_A);
         seed_A_origin[0] = UINT16_TO_LE(i + 4);
         shake128((unsigned char*)(A + 4*PARAMS_N), (unsigned long long)(2*PARAMS_N),
                  seed_A_separated, 2 + BYTES_SEED_A);
         seed_A_origin[0] = UINT16_TO_LE(i + 5);
         shake128((unsigned char*)(A + 5*PARAMS_N), (unsigned long long)(2*PARAMS_N),
                  seed_A_separated, 2 + BYTES_SEED_A);
         seed_A_origin[0] = UINT16_TO_LE(i + 6);
         shake128((unsigned char*)(A + 6*PARAMS_N), (unsigned long long)(2*PARAMS_N),
                  seed_A_separated, 2 + BYTES_SEED_A);
         seed_A_origin[0] = UINT16_TO_LE(i + 7);
         shake128((unsigned char*)(A + 7*PARAMS_N), (unsigned long long)(2*PARAMS_N),
                  seed_A_separated, 2 + BYTES_SEED_A);

         for (j = 0; j < PARAMS_NBAR; j++) {
             uint16_t sum = 0;
```

```
            int16_t sp[8];
            for (p = 0; p < 8; p++) {
                sp[p] = s[j * PARAMS_N + i + p];
            }
            for (q = 0; q < PARAMS_N; q++) {
+               sum = out[j*PARAMS_N + q];
+               for (p = 0; p < 8-1; p++) {
+                   sum += sp[p] * A[p * PARAMS_N + q];
+               }
+               % check at last p index only, previous for cycle shortened by 1
+               if (i == PARAMS_N-8){
+                   sum = AxADD((uint64_t)sum,
+                               (uint64_t)((uint16_t)sp[p] * A[p * PARAMS_N + q]));
+               } else {
+                   sum += sp[p] * A[p * PARAMS_N + q];
+               }
+
+               out[j * PARAMS_N + q] = sum;
            }
        }
    }

    return 1;
}
```

Listing B.5 Original FrodoKEM implementation of `frodo_mul_add_sb_plus_e` highlighting the lines targeted for deletion or replacement (marked with -).

```
--- src/matrix_orig.c
+++ src/matrix_orig.c
@@ -1,18 +1,18 @@
-void frodo_mul_add_sb_plus_e(uint16_t *out, const uint16_t *b, const uint16_t *s,
-                             const uint16_t *e)
 { // Multiply by s on the left
   // Inputs: b (N x N_BAR), s (N_BAR x N), e (N_BAR x N_BAR)
-  // Output: out = s*b + e (N_BAR x N_BAR)
    int i, j, k;

    for (k = 0; k < PARAMS_NBAR; k++) {
        for (i = 0; i < PARAMS_NBAR; i++) {
-           out[k*PARAMS_NBAR + i] = e[k*PARAMS_NBAR + i];
-           for (j = 0; j < PARAMS_N; j++) {
-               out[k*PARAMS_NBAR + i] +=
-                   (int16_t)s[k*PARAMS_N + j] * b[j*PARAMS_NBAR + i];
-           }
-           out[k*PARAMS_NBAR + i] =
-               (uint32_t)(out[k*PARAMS_NBAR + i]) & ((1<<PARAMS_LOGQ)-1);
        }
    }
```

```
}
```

Listing B.6 Approximate variant of `frodo_mul_add_sb_plus_e` explicitly highlighting the software optimizations and hardware operator injection (marked with +).

```
--- src/matrix_approx.c
+++ src/matrix_approx.c
@@ -1,25 +1,25 @@
+void frodo_mul_add_sb_plus_e(uint16_t *out, const uint16_t *b, const uint16_t *s)
 { // Multiply by s on the left
   // Inputs: b (N x N_BAR), s (N_BAR x N), e (N_BAR x N_BAR)
+  // Output: out = s*b (+ e) (N_BAR x N_BAR)
+  // Note: the error e' is no more generated but introduced by means of an approximate
     adder
     int i, j, k;

     for (k = 0; k < PARAMS_NBAR; k++) {
         for (i = 0; i < PARAMS_NBAR; i++) {
+           uint16_t sum = 0; // Initialize output to zero
+           for (j = 0; j < PARAMS_N-1; j++) {
+               // Perform the multiplication
+               sum += (int16_t)s[k*PARAMS_N + j] * b[j*PARAMS_NBAR + i];
+           }
+
+           // approximate sum of partial result stored in out[]
+           //    with partial product sp[]*A[]
+           sum = AxADD((uint64_t)sum,
+                       (uint64_t)((uint16_t)s[k*PARAMS_N + j] * b[j*PARAMS_NBAR + i]));
+
+           // Ensure the result is reduced modulo Q
+           sum = (uint32_t)(sum) & ((1 << PARAMS_LOGQ) - 1);
+           out[k*PARAMS_NBAR + i] = sum;
         }
     }
 }
```

Listing B.7 Summarized unified diff of the core FrodoKEM top-level operations, illustrating the global reduction in error matrix memory spaces, omitted sampling calls, and streamlined function interfaces.

```
--- src/kem_orig.c
+++ src/kem_approx.c
@@ -1,25 +1,20 @@
 int crypto_kem_keypair(unsigned char* pk, unsigned char* sk, unsigned char*
     randomness)
 {
     uint16_t B[PARAMS_N*PARAMS_NBAR] = {0};
-    uint16_t S[2*PARAMS_N*PARAMS_NBAR] = {0};
-    uint16_t *E = (uint16_t *)&S[PARAMS_N*PARAMS_NBAR];
+    uint16_t S[PARAMS_N*PARAMS_NBAR] = {0};
```

```
10      ...
11  -   shake((uint8_t*)S, 2*PARAMS_N*PARAMS_NBAR*sizeof(uint16_t),
12  -               shake_input_seedSE, 1 + BYTES_SEED_SE);
13  +   shake((uint8_t*)S, PARAMS_N*PARAMS_NBAR*sizeof(uint16_t),
14  +                shake_input_seedSE, 1 + BYTES_SEED_SE);
15      ...
16      frodo_sample_n(S, PARAMS_N*PARAMS_NBAR);
17  -   frodo_sample_n(E, PARAMS_N*PARAMS_NBAR);
18  -   frodo_mul_add_as_plus_e(B, S, E, pk);
19  +   frodo_mul_add_as_plus_e(B, S, pk);
20      ...
21      clear_bytes((uint8_t *)S, PARAMS_N*PARAMS_NBAR*sizeof(uint16_t));
22  -   clear_bytes((uint8_t *)E, PARAMS_N*PARAMS_NBAR*sizeof(uint16_t));
23      return 0;
24  }
25
26  @@ -30,22 +25,17 @@
27  int crypto_kem_enc(unsigned char *ct, unsigned char *ss, const unsigned char *pk,
        unsigned char *mu_input)
28  {
29      ALIGN_HEADER(32) uint16_t Sp[(2*PARAMS_N+PARAMS_NBAR)*PARAMS_NBAR] ALIGN_FOOTER
            (32) = {0};
30  -   uint16_t *Ep = (uint16_t *)&Sp[PARAMS_N*PARAMS_NBAR];
31  -   uint16_t *Epp = (uint16_t *)&Sp[2*PARAMS_N*PARAMS_NBAR];
32      ...
33  -   shake((uint8_t*)Sp, (2*PARAMS_N+PARAMS_NBAR)*PARAMS_NBAR*sizeof(uint16_t),
34  -               shake_input_seedSE, 1 + BYTES_SEED_SE);
35  +   shake((uint8_t*)Sp, PARAMS_N*PARAMS_NBAR*sizeof(uint16_t),
36  +               shake_input_seedSE, 1 + BYTES_SEED_SE);
37      ...
38      frodo_sample_n(Sp, PARAMS_N*PARAMS_NBAR);
39  -   frodo_sample_n(Ep, PARAMS_N*PARAMS_NBAR);
40  -   frodo_mul_add_sa_plus_e(Bp, Sp, Ep, pk_seedA);
41  +   frodo_mul_add_sa_plus_e(Bp, Sp, pk_seedA);
42      ...
43  -   frodo_sample_n(Epp, PARAMS_NBAR*PARAMS_NBAR);
44      frodo_unpack(B, PARAMS_N*PARAMS_NBAR, pk_b,
45                   CRYPTO_PUBLICKEYBYTES - BYTES_SEED_A, PARAMS_LOGQ);
46  -   frodo_mul_add_sb_plus_e(V, B, Sp, Epp);
47  +   frodo_mul_add_sb_plus_e(V, B, Sp);
48      ...
49  -   clear_bytes((uint8_t *)Ep, PARAMS_N*PARAMS_NBAR*sizeof(uint16_t));
50  -   clear_bytes((uint8_t *)Epp, PARAMS_NBAR*PARAMS_NBAR*sizeof(uint16_t));
51      return 0;
52  }
53
54  @@ -60,21 +50,15 @@
```

```
int crypto_kem_dec(unsigned char *ss, const unsigned char *ct, const unsigned char *
    sk)
{
    ALIGN_HEADER(32) uint16_t Sp[(2*PARAMS_N+PARAMS_NBAR)*PARAMS_NBAR] ALIGN_FOOTER
        (32) = {0};
-   uint16_t *Ep = (uint16_t *)&Sp[PARAMS_N*PARAMS_NBAR];
-   uint16_t *Epp = (uint16_t *)&Sp[2*PARAMS_N*PARAMS_NBAR];
    ...
-   shake((uint8_t*)Sp, (2*PARAMS_N+PARAMS_NBAR)*PARAMS_NBAR*sizeof(uint16_t),
-             shake_input_seedSEprime, 1 + BYTES_SEED_SE);
+   shake((uint8_t*)Sp, PARAMS_N*PARAMS_NBAR*sizeof(uint16_t),
+             shake_input_seedSEprime, 1 + BYTES_SEED_SE);
    ...
    frodo_sample_n(Sp, PARAMS_N*PARAMS_NBAR);
-   frodo_sample_n(Ep, PARAMS_N*PARAMS_NBAR);
-   frodo_mul_add_sa_plus_e(BBp, Sp, Ep, pk_seedA);
+   frodo_mul_add_sa_plus_e(BBp, Sp, pk_seedA);
    ...
-   frodo_sample_n(Epp, PARAMS_NBAR*PARAMS_NBAR);
    frodo_unpack(B, PARAMS_N*PARAMS_NBAR, pk_b,
                 CRYPTO_PUBLICKEYBYTES - BYTES_SEED_A, PARAMS_LOGQ);
-   frodo_mul_add_sb_plus_e(W, B, Sp, Epp);
+   frodo_mul_add_sb_plus_e(W, B, Sp);
    ...
-   clear_bytes((uint8_t *)Ep, PARAMS_N*PARAMS_NBAR*sizeof(uint16_t));
-   clear_bytes((uint8_t *)Epp, PARAMS_NBAR*PARAMS_NBAR*sizeof(uint16_t));
    return 0;
}
```