INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2025IPPAT032

Thèse de doctorat

Secure implementation for post-quantum cryptography

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom Paris

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (ED IP Paris)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, le 13 octobre 2025, par

MAXIME SPYROPOULOS

Composition du Jury :

Jacques Patarin Professeur, Université de Versailles Saint-Quentin-en-Yvelines	Président / Examinateur
Jean-Sébastien Coron Professeur, Université du Luxembourg	Rapporteur
Louis Goubin Professeur, Université de Versailles Saint-Quentin-en-Yvelines	Rapporteur
Cécile Dumas Ingénieur-Chercheur, CEA-LETI	Examinatrice
Guénaël Renault Responsable Adj. de Laboratoire, Chercheur HDR, ANSSI	Examinateur
Renaud Pacalet Directeur d'études, Télécom Paris	Directeur de thèse
Agathe Cheriere Docteur, Direction Générale de l'Armement	Invitée
Fabrice Perion Ingénieur, Thales CDI	Invité

"Never lift, never stop believing."
– Sebastian Vettel

Remerciements

J'aime beaucoup la coïncidence qui fait que je soutiens ma thèse le 13 octobre, sachant que le 14 octobre 2013 je commençais mon stage d'observation de 3ème chez Gemalto. Pendant ce stage, j'ai visité un labo d'attaques et rencontré deux cryptologues, Fabrice et Sylvain, qui m'ont présenté leur métier. Au moment où l'on m'a dit « tu sais, si tu nous donnes ton téléphone, on peut retrouver le code PIN de ta carte SIM », j'ai eu une révélation. Adieu pilote d'avions de chasse, je voulais désormais être cryptologue. Ça s'impressionne d'un rien, hein, un gamin de 13 ans...

Puisque je ne crois pas qu'on puisse dire « merci » sans rien derrière et que j'ai cru comprendre que c'était la partie du manuscrit où j'avais le droit de dire à peu près tout ce que je voulais, croyez bien que je vais en profiter.

Tout d'abord, merci Fabrice pour ton encadrement tout au long de ma thèse dans cette maison sérieuse. J'ai énormément apprécié t'avoir à mes côtés durant ces 3 ans, en particulier quand je faisais relire mes écrits et que tu me rassurais toujours par un « bon, ça m'paraît pas mal ». C'était la première thèse que tu encadrais et je ne pense pas que j'aurais pu espérer mieux.

Merci David pour ta bienveillance et ton temps. Tu avais beau avoir parfois 3 réunions par créneau, tu trouvais toujours le temps de discuter de ma thèse ou de relire mes productions. Tes idées et ton regard académique ont grandement contribué à la réussite de ce travail.

Renaud, merci d'avoir accepté d'être mon directeur de thèse et d'avoir été aussi impliqué malgré la distance géographique. J'ai beaucoup aimé nos discussions enrichissantes. Grâce à toi, j'ai aussi grandement progressé en rédaction LaTeX, même si je n'ai pas encore ta rigueur.

Merci beaucoup Gilles d'avoir bien voulu d'un nouveau « Spyro » dans tes équipes.

Je suis très reconnaissant à l'Agence de l'Innovation de Défense pour la confiance qu'elle m'a accordée, en particulier je veux remercier Benoît, Margaux et Agathe pour leur suivi.

Merci à Fabienne Jézéquel et Charles Bouillaguet d'avoir bien voulu m'écrire deux lettres de recommandation sans lesquelles ma candidature en thèse n'aurait jamais été considérée.

Petit coucou à Gabriel et Mathieu qui m'ont donné le goût de la recherche et l'envie de poursuivre en thèse après mon stage.

Merci à Jean-Sébastien Coron et Louis Goubin d'avoir accepté d'être rapporteurs de mon manuscrit. Merci également à Guénaël Renault et Cécile Dumas d'avoir accepté de faire partie de mon jury. Je tiens aussi à remercier Mélissa Rossi, qui avait accepté de participer à ma soutenance mais qui n'a malheureusement pas pu y assister.

Merci Jacques d'avoir bien voulu présider mon jury de thèse, surtout dans un délai aussi court. J'ai beaucoup aimé cette tendance que tu as à transformer chaque présentation en cours d'histoire, quel que soit le sujet initial.

Merci à Benoît et Guénaël pour leur suivi en tant que membres de mon CSI.

Merci à Laurent d'avoir soutenu ma candidature de thèse.

Maya, Andersson, merci à tous les deux d'avoir si souvent répondu à mes questions concernant la thèse. Du début à la fin, j'ai eu de la chance d'avoir des personnes qui étaient déjà passées par là et prêtes à m'éclairer. Vos manuscrits ont été pour moi une véritable mine d'or et une source d'inspiration dans l'écriture de ces pages.

Je tiens à remercier Nicolas, Philippe et tout particulièrement Guillaume, pour cette collaboration fructueuse et stimulante qu'a été le masquage de HQC.

Merci à Steven, Li Xun et Benoît, avec vous je ne manquais pas d'experts pour répondre à mes questions, qu'elles portent sur les side-channel, le fonctionnement du labo, le Deep Learning ou les preuves de sécurité.

Merci Sylvain pour ta disponibilité et surtout ton aide précieuse. Parmi les personnes non impliquées dans l'encadrement de ma thèse, tu es celui qui y a le plus contribué, que ce soit pour m'aider à débuguer des STM32 que j'avais du mal à prendre en main ou coder des prototypes de programmes dont j'avais l'idée mais aucun moyen pour la réaliser.

Je sais que dans ses remerciements, Andersson m'avait décerné le titre (assurément usurpé) de « doctorant le plus stylé ». Je profite de ces lignes pour le restituer à toi, Paco, qui le mérite réellement.

Merci à Quentin et Tuan, les boss du CSE Games et à toutes les personnes qui font vivre le club. J'ai passé de super pauses midi à me faire retourner le cerveau sur Avalon. Désolé par contre, il va falloir trouver une nouvelle personne pour finir dernier (premier en partant de la fin) à chaque partie de Skull King.

Merci à Loup, Nyts, Axel, Richard et Christian pour nos discussions passionnantes durant nos pauses café au 4ème, nos parties de babyfoot, nos excursions du mardi chez le colonel Sanders et nos inoubliables soirées au Falstaff.

Loïc, merci de partager ma passion pour le rougail saucisse, la pana cota framboise mais surtout le Speed Duel et le mod casino de Lethal Company. Merci Pierre d'être un

philhellène si remarquable, aussi balèze en Histoire que sur Hearts of Iron. Éric, merci pour toutes nos discussions sur les séries et l'univers de G.R.R. Martin mais surtout pour tes recommandations (Band of Brothers quelle claque !). Merci Youssouf pour ta bonne humeur communicative. Avec toi, chaque mardi ressemblait à un vendredi. Simon, bon courage pour ta thèse et merci de m'avoir fait découvrir The Great Review. Amélie, merci pour ton humour.

Merci à Ayoub, Loïc, Joseph, Baptiste, Dylan et Joris pour nos soirées JDR, les parties de Yu-Gi-Oh du dimanche aprèm, les week-ends à Grenoble et nos aventures au Japon. Promis Loïc, faire une thèse ça ne sert pas juste à pouvoir se la péter en répondant « PhD student » quand la police aux frontières de l'aéroport d'Osaka nous demande ce que l'on fait dans la vie.

Merci Clément pour nos débriefs de GP, ton enthousiasme à chaque race week (RAWE CEEK ?) et cette superbe 92e édition des 24h du Mans.

Même si l'on s'est moins vu pendant ces trois ans, merci aux Rangers du SFPN pour les bons moments.

Reda, Alex, je pense que vous n'auriez pas pu trouver pire timing pour convertir Pierre à R6 qu'un mois avant le début de ma thèse. Merci à tous les trois pour nos soirées à mourir de rire sur Phasmo, R6, RoN, ... (j'en oublie), et CS2 (même si ça me fait mal de l'admettre). Elles m'ont souvent permis de souffler pendant les périodes les plus compliquées.

Louis, merci d'être là depuis 19 ans. De ces trois dernières années je retiendrai : la Saint-Patrick à Dublin (pour le meilleur comme pour le pire), nos sessions dix heures non-stop sur Pokémon et tes capacités de diplomate avec mes voisins.

Une thèse c'est de la recherche et je peux m'estimer très chanceux de t'avoir trouvée, Laurine. Dès le premier rendez-vous c'était évident, même si nous avons des avis très différents concernant la saison 2021 de F1... Merci pour tes messages presque quotidiens pour m'encourager durant la rédaction de ce manuscrit. Mon plus beau succès de ces trois années de thèse, c'est toi.

Hugo, Sacha, c'est une évidence à quel point vous comptez pour moi. Je vous rassure que ce n'est pas seulement aujourd'hui que je m'en rends compte. Vous êtes les seuls capables de me faire aimer jouer à des jeux solos avec vous, même si l'on doit se passer la manette. Dans ces moments-là, je ne voudrais être nulle part ailleurs.

Maman, Papa, ce n'est pas spécialement maintenant que ma thèse se termine que je dois vous dire merci. Je vous remercie pour tout, pour votre soutien depuis toujours et de m'avoir supporté trois ans de plus.

Merci à Maze et surtout à Lucifer de m'avoir soutenu pendant la rédaction de ce manuscrit en sautant, au choix, sur mon bureau ou sur mes genoux, ou en passant la journée à (ronfler) dormir sur mon lit.

Enfin, un mot pour Dietrich Mateschitz, visionnaire disparu peu après le début de ma thèse et dont les canettes m'auront souvent aidé à tenir durant les longues sessions de codage et de rédaction.

Ce travail a été partiellement financé par la subvention 2022154 de l'appel à projets 2022 thèses AID Cifre-Défense de l'Agence de l'Innovation de Défense (AID), Ministère des Armées.

Résumé de la thèse en français

L'avènement des ordinateurs quantiques menace la cryptographie classique, ce qui motive le développement de la cryptographie post-quantique (PQC). En effet, un ordinateur quantique suffisamment puissant, exécutant l'algorithme de Shor, pourrait résoudre les problèmes de factorisation et du logarithme discret sur lesquels reposent notre cryptographie à clef publique actuelle. Depuis 2016, cet effort de développement de la PQC est porté par le NIST (National Institute of Standards and Technology), qui a organisé une compétition internationale ayant pour but de décider des prochains algorithmes standards.

Cette thèse porte sur l'analyse et l'implémentation sécurisée de l'algorithme à base de codes HQC, candidat du quatrième tour du concours de standardisation post-quantique du NIST.

Après avoir examiné les candidats participants au quatrième tour, nous avons choisi d'étudier HQC. D'une part, tout comme BIKE, HQC repose sur des codes quasi-cycliques rendant ses tailles de clefs et de chiffré compatibles avec les contraintes propres aux systèmes embarqués (ce qui n'était pas le cas de McEcliece). D'autre part, HQC se distingue par des performances plus compétitives que celles de BIKE, en particulier en termes de vitesse d'exécution pour la génération de clefs et la décapsulation.

Ce travail débute par une étude détaillée des attaques par canaux auxiliaires (SCA) applicables à HQC. Elle présente également une amélioration d'une attaque en Timing de Guo et al., rendue possible par une analyse fine de la capacité de correction du décodeur de HQC, permettant de diviser par trois le nombre de requêtes initialement nécessaires pour retrouver la clef secrète de HQC. Sur la base de cet état de l'art, cette thèse présente la première analyse de sensibilité du schéma et évalue la contre-mesure de masquage, une technique couramment utilisée consistant à randomiser les calculs intermédiaires. La contribution principale est la conception et la validation expérimentale de la première implémentation entièrement masquée de HQC.

L'analyse de sensibilité a été utile pour déterminer quelles étaient les fonctions et variables sensibles lors de l'exécution de HQC. Elle nous a également permis de définir les variables non-sensibles, lesquelles ne nécessitaient pas de protection, ce qui a permis d'éviter des surcoûts inutiles dus au masquage dans notre implémentation.

Aussi, une attention particulière a été portée au masquage de l'algorithme de vector sampling ou tirage de vecteurs aléatoires. Le vector sampling de HQC n'était pas constant en temps, cette différence de temps d'exécution pouvait être exploitée pour retrouver la clef secrète (comme démontré par Guo et al.). Un algorithme de vector sampling en temps constant a été proposé par Nicolas Sendrier pour BIKE et cette solution a été également retenue par l'équipe de HQC. Cependant, le tirage de l'aléa pour HQC s'effectue avec un calcul de modulo, une opération qui n'est généralement pas constante en temps, dépendamment des opérandes et de l'architecture exécutant le programme. Cette observation a été faite par Schröder et al., qui ont suggéré de faire ce calcul de modulo en codant en dur la réduction de Barrett dans le code de HQC. En masquant cette réduction de Barrett, nous avons remarquée que 80 % du temps d'exécution était passé dans le calcul de la multiplication. En utilisant le masquage arithmétique et des conversions de masque nous avons pu accélérer la multiplication et réduire de 30 % le temps d'exécution total de la réduction de Barrett masquée.

Afin de démontrer la sécurité théorique d'une implémentation masquée, on prouve généralement la sécurité de chaque fonction masquée (appelées gadgets) indépendamment, puis on compose ces gadgets de manière sécurisée en suivant des règles de composition. Ainsi, on prouve la sécurité de l'ensemble de la construction. Notre implémentation sécurisée de HQC repose sur l'utilisation de gadgets MIMO-SNI. Cette notion, introduite

par Cassiers et Standaert, définit un type de gadgets qui se compose entre eux, permettant de prouver la sécurité théorique de l'ensemble uniquement à partir de la preuve de sécurité des briques élémentaires. Nous avons également soumis notre implémentation a une analyse de fuite side-channel afin de prouver sa sécurité aussi bien en théorie qu'en pratique.

Enfin, nous avons fourni des benchmarks de notre implémentation masquée de HQC, afin de rendre compte de l'impact du masquage sur ses performances en temps d'exécution. Nous avons pu ainsi la comparer avec une implémentation masquée de l'état de l'art, celle de BIKE, et conclure que notre implémentation était compétitive. Cette dernière est disponible publiquement sur Github.

Contents

Part I

Introduction, Context and Background

Chapter 1

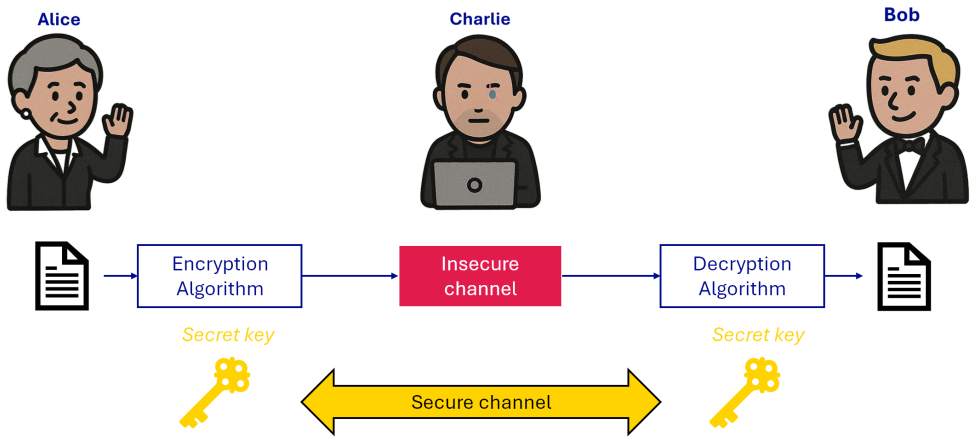
Introduction

Contents

1.1 Cryptography

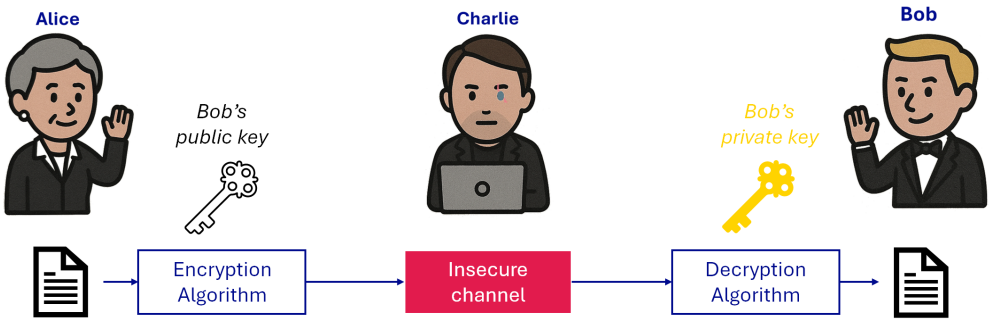
Without realizing it, we rely on cryptography every day to exchange private messages or pay with our credit card. Its origins date back to Antiquity, after the invention of writing, mankind searched for a way to write down secret messages. Throughout history, both cryptography (the art of writing secrets) and cryptanalysis (the art of recovering them) evolved. In the beginning, the goal of cryptography was to convey messages between two parties without the possibility for eavesdroppers to understand them (confidentiality). Nowadays, cryptography stills serves for confidentiality, but also covers integrity (guarantee that the message was not modified), authenticity (guarantee the identity of the communicating parties), non-repudiation (prevent a party from denying having carried out an action), etc. Two types of cryptography are commonly distinguished: symmetric and asymmetric.

Symmetric cryptography The same key is used to encrypt the plaintext (the clear message) and to decrypt the resulting ciphertext. One drawback of the symmetric key paradigm is that it requires a shared secret among the communicating parties, that is, a



secure channel to exchange such a secret. This has been mitigated with the emergence of asymmetric cryptography in the 1970s, especially with its public discovery by Diffie and Hellman in 1976 [DH76].

Asymmetric cryptography In asymmetric cryptography a pair of keys is used: a public key is used to encrypt and a secret key is used to decrypt. The two keys are linked via a mathematical relation that allows the owner of a secret key to decrypt any message encrypted using the corresponding public key. It is similar to Bob giving Alice an open padlock so she can store her message in a chest and lock it: only Bob owns the key that can open the chest to read Alice's message. Asymmetric cryptography is also used for other services like, for example, authentication: to prove that she is the author of a message Alice uses her private key to compute a digital signature of the message, and Bob uses Alice's public key to verify that the signature and the message he received match. As only Alice knows her private key and can compute such signatures, this verifies that Alice is indeed the author.



1.2 Quantum threat

Asymmetric cryptography relies on the difficulty to recover a secret key with the only knowledge of the corresponding public key. This difficulty comes from a computationally hard mathematical problem requiring intractable amounts of computing power and time to be solved. RSA (Rivest–Shamir–Adleman), for instance, relies on the hardness of the factoring of numbers (find the prime factors of a number) and of the discrete logarithm (given x and y two elements of a group, find integer d such that $x^d = y$).

However, these security guaranties are challenged by the emergence of quantum computing.

While a classic computer uses bits to perform computations, quantum computers rely on Qbits. Qbits are quantum elements (particles, groups of particles...) like electrons, and quantum computers compute by acting on a particular quantum property of the Qbits like, for instance, the spin of electrons.

Different from a classic bit, that can only be in one of two states (0 or 1), Qbits can be in a *superposition* of states. For example, if the spin of an electron, measured along a given axis, can take values $|\uparrow\rangle$ or $|\downarrow\rangle$, the electron can be in a superposition of 1/3 of $|\uparrow\rangle$ and 2/3 of $|\downarrow\rangle$, meaning that a measure would output $|\uparrow\rangle$ with probability 1/3 and $|\downarrow\rangle$ with probability 2/3.

Another fundamental property of Qbits is the *entanglement*: the state of a group of entangled Qbits is not the simple juxtaposition of their individual states, as it would be with classic bits, it is a superposition of all possible combinations of their fundamental states. A pair of entangled electrons, for example, can be in a superposition of 1/4 of $|\uparrow\uparrow\rangle$, 0 of $|\uparrow\downarrow\rangle$, 1/4 of $|\downarrow\uparrow\rangle$ and 1/2 of $|\downarrow\downarrow\rangle$; if the first electron is measured as $|\uparrow\rangle$ the other can only be measured as $|\uparrow\rangle$ too, no matter the distance between them, because the entangled state has weight (probability) 0 in the $|\uparrow\downarrow\rangle$ direction.

Superposition and entanglement allow quantum computers to (theoretically) outperform classic computers on tasks where their ability to process as a whole the joint state of a large number of entangled Qbits can be exploited. In particular Peter Shor proposed in 1994 several quantum algorithms [Sho94] to efficiently solve the factoring problem and the discrete logarithm problem. As these two problems are the underlying hard problems of RSA, if a powerful enough quantum computer was developed, RSA and similar algorithms (ElGammal, Diffie-Hellman, elliptic curves...) would not be secure any more.

Symmetric cryptography, integrity checking based on cryptographic hash functions, and other cryptographic primitives are also threatened by quantum computers. In 1996 Lov Grover proposed a quantum algorithm providing a quadratic speed-up of an exhaustive search [Gro96]. This implies that in order to maintain the same level of security of a symmetric block cipher, a twice larger secret key must be used. AES (Advanced Encryption Standard) with 256 bits secret keys can be considered as post-quantum while its version with 128 bits secret keys, cannot: $\sqrt{2^{256}} = 2^{128} \approx 3.4 \times 10^{38}$ AES encryptions by an efficient and large quantum computer is likely intractable but $\sqrt{2^{128}} = 2^{64} \approx 1.9 \times 10^{19}$ is probably not. A quantum computer capable of one trillion (10^{12}) AES encryptions per second would need almost one billion (10^9) times the age of the universe to break AES-256 but only half a year to break AES-128. Nevertheless, some experts argue that it will never be possible to break AES-128 with quantum computers, citing in particular that they parallelize badly (see the invited talk at CHES 2024 [che]). This opinion is shared by NIST (National Institute of Standards and Technology) [nis] but some certification authorities like ANSSI (Agence Nationale de la Sécurité des Systèmes d'Information) prefer more conservative security parameters for block ciphers and hash functions [ans].

Companies like IBM (International Business Machines Corporation) and Google already developed quantum computers, and aim to build larger, fault-tolerant quantum computers in the future [goo, ibm]. Governments have also expressed ambitions to build their own [fre, usa]. At the time of writing (2025) existing quantum computers are not powerful enough to be a serious threat for classic asymmetric cryptography. However, an adversary could record sensitive communications today and wait until the advent of a quantum computer capable of decrypting them. This "Harvest Now, Decrypt Later" threat puts at risk information that we want to keep secret in the coming years. This

concern is formalized in Mosca's Theorem [mos13], which considers three time frames:

x: the desired confidentiality period of a given encrypted piece of data,

y: the time required to migrate to post-quantum cryptographic solutions,

z: the time before quantum computers become powerful enough to break current cryptography.

According to the theorem, if $x + y > z$, then action must be taken urgently, as the data being protected today may be compromised in the future. This highlights the necessity of accelerating the transition to PQC (Post-Quantum Cryptography).

1.3 New cryptographic algorithms

For all these reasons, the NIST of the USA started a worldwide competition in 2016 to select the post-quantum (i.e., resistant to both classic and quantum attackers) cryptographic algorithms to become the next standards [nis17]. Ideally, these post-quantum algorithms would in time replace our current algorithms to ensure that we are ready when a powerful enough quantum computer emerges. Initially, NIST received 82 candidate algorithms and accepted 69 of them as first-round contestants [MAAS+19]. There were five distinct algorithmic families: code-based (HQC (Hamming Quasi-Cyclic), BIKE), isogeny-based (SIKE (Supersingular Isogeny Key Encapsulation)), hash-based (SPHINCS+, Picnic), lattice-based (ML-KEM (Module Lattice based Key Encapsulation Mechanism) — formerly CRYSTALS-Kyber, CRYSTALS-Dilithium), and multivariate-based (Rainbow, GeMSS). To determine their suitability as new standards, the contestants were evaluated on several criteria. First, their security levels were benchmarked, through cryptanalysis, against security strengths offered by existing standards (namely AES and SHA (Secure Hash Algorithm)). These levels were established to facilitate performance comparisons between the contestants, they are recalled in Table 1.1. Second their computational cost were evaluated, both in terms of execution time and memory footprint. Lastly, their resistance against side-channel attacks (see Section1.4) was also taken into account.

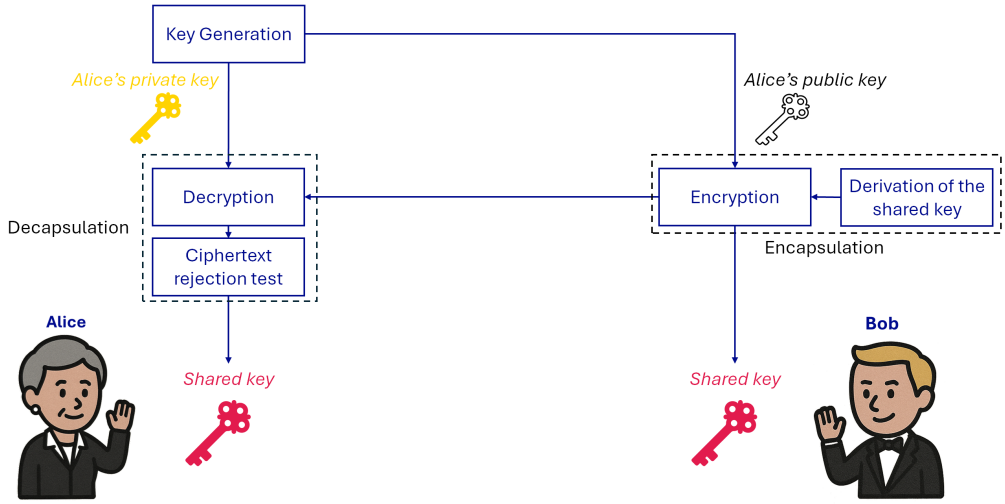
Level	**Security description**	**Equivalent standard**
I	Key search on a block cipher with a 128-bit key	(AES-128)
II	Collision search on a 256-bit hash function	(SHA256/SHA3-256)
III	Key search on a block cipher with a 192-bit key	(AES-192)
IV	Collision search on a 384-bit hash function	(SHA384/SHA3-384)
V	Key search on a block cipher with a 256-bit key	(AES-256)

Table 1.1: NIST security levels

At the time of writing, June 2025, the main competition has already ended, with the selection of two KEMs (Key Encapsulation Mechanisms) and three DSAs (Digital Signature Algorithms) [ABC+25]. A smaller, still ongoing competition, started in 2022 to select other DSAs that would not rely on the difficulty of solving lattice-based problems.

The reader should note that an algorithm is considered "post-quantum" not because it is proved to resist any quantum attack, but because there are currently no known efficient quantum algorithms to solve the underlying mathematical problem.

Key Encapsulation Mechanisms A KEM allows two parties to securely exchange a shared key using a PKE (Public Key Encryption). This is achieved through the FO (Fujisaki-Okamoto) transform [FO99], which strengthens a PKE into a CCA-secure KEM (i.e., secure against adaptive chosen ciphertext attacks). Once the key pair is generated, the sender randomly selects a symmetric key, encrypts it using the recipient's public key and sends the resulting ciphertext to them. The ciphertext is then decrypted using the recipient's secret key to recover the shared key. The FO transform includes a check to ensure the ciphertext is valid; if this check passes, both parties now share a common secret key. If not, a random key is returned to preserve security.



1.3.1 BIKE vs HQC

In October 2022, NIST announced that four algorithms were selected for standardization, out of the original 69 contestants [AAC+22]. Three out of four relied on the hardness of lattice-based problems. Fearing a major cryptanalytic advance against lattice-based schemes, NIST issued a fourth round of competition to diversify its selection [AAC+22]. The four algorithms partaking in the fourth round of selection were BIKE, HQC, McEliece and SIKE. They provided NIST a useful fall-back line.

SIKE, the isogeny based scheme, was broken soon after in a paper by Castryck and Decru [CD23]. Their attack ran on a laptop and could recover the secret key in ten minutes. SIKE team acknowledged it and announced that they withdraw from the competition.

The three remaining schemes where all based on error-correcting codes. McEliece suffered from its huge secret key (6 kB) and even larger public key (255 kB). The two favourites were BIKE and HQC, both relying on quasi-cyclic codes to mitigate the size problems faced by McEliece. BIKE had a smaller public-key and ciphertext size. This limited its bandwidth usage, enabling faster and more efficient communication over networks. On the other hand, HQC had faster key generation and, more important, its encapsulation plus decapsulation was also about twice faster than that of BIKE. The byte sizes and computation times in kilocycles on an x86_64 platform are shown in tables 1.2 and 1.3 where ML-KEM values are also given for comparison.

In March 2025, NIST selected HQC for standardization as an alternative KEM to ML-KEM. The main reason behind this choice was its better and more stable DFR (Decryption Failure Rate) analysis, which was a concern in BIKE [ABC+25].

Name	Private key	Public key	Ciphertext
HQC	40	2,249	4,497
BIKE	281	1,541	1,573
McEliece	6,492	261,120	96
ML-KEM	1,632	800	768

Table 1.2: Key and ciphertext sizes (bytes) of fourth round contestants

Name	KeyGen	Encaps	Decaps
HQC	105	197	360
BIKE	599	105	1,642
McEliece	80,376	36	92
ML-KEM	24	26	30

Table 1.3: Speed comparison (kilocycles) of fourth round contestants on x86_64 [oqs]

1.4 Side Channel Attacks

Even when cryptographic algorithms are considered as secure, they run on real hardware, which components (memory, processor, ...) unintentionally leak information on sensitive data via different channels called "side-channels". These leaks are intrinsic to the hardware platform and their magnitude directly depend on the operations carried out during the computation. They take different forms: electromagnetic radiation, power consumption, heat, noise, execution time, etc. An attacker can exploit these leaks to recover parts of secrets without breaking the cryptographic algorithm itself.

In 1996 Paul Kocher was the first to publicly highlight and exploit the link between the execution time of a cryptographic algorithm and the secret data it manipulates [Koc96]. The operands of modular multiplications and squares involved in the modular exponentiation of RSA depend on the secret exponent, and the time taken by these operations usually depend on the value of their operands, for various reasons mostly related with performance optimisation techniques. Leveraging on this P. Kocher showed that the execution time of a software implementation of RSA can serve as a distinguisher between correct and wrong hypotheses on portions of the secret exponent. This discovery gave rise to the discipline of SCA (Side Channel Attack).

In 1999 the same P. Kocher extended his previous work to the power consumption side channel and to symmetric block ciphers [KJJ99].

SCA are generally classified into two main families: non-profiled and profiled.

1.4.1 Non-profiled SCA

In "non-profiled" SCA attackers try to extract the secret directly from the target system.

Timing Attack Timing attacks exploit variations in the execution time of cryptographic operations to extract secret information. In practice, the duration of such operations often depends on the processed input and, crucially, on the secret key. By precisely measuring these timings, an attacker can gradually recover sensitive data. Kocher's 1996 attack is one of the most prominent examples of timing attacks.

DPA (Differential Power Analysis) The DPA [KJJ99] targets the result of an operation that depends on both a known input and the secret key. The attacker begins by recording power traces for different inputs. Next, they make a guess about the value of (part of) the secret key, and compute the expected result of the target operation. The traces are then divided in two groups, according to the value of a specific bit in this result. If the guess is correct, the difference between the average power consumption of the two groups will exhibit statistically significant peaks, corresponding to the moment when this bit influences the power consumption. If the guess is wrong however, then this classification will be arbitrary and no visible difference will be observed between the mean traces. By testing multiple key hypotheses and repeating this process across all bits of the key, the attacker reconstructs the full secret key.

CPA (Correlation Power Analysis) The CPA [BCO04] is a frequently encountered technique in non-profiled power SCA. It exploits the linear correlation between the power consumption of the cryptosystem and the transitions on internal nodes of the target. Attackers record power traces and input (or output) messages during operations of the cryptosystem. Thanks to hypotheses on small portions of the secret, and the knowledge of the input or output messages of the cryptosystem, attackers can estimate hypothetical internal node transitions, and compute statistical correlations with the recorded power traces. The highest correlation then indicates which hypothesis on the secret is the most likely.

1.4.2 Profiled SCA

"Profiled" SCA assume that the adversary has access to a clone of the target system, on which they can record as many side channel traces as they wish, with full control over the inputs and keys used. A common example of profiled power SCA is the Template Attack [CRR02] in which the adversary first varies the keys and inputs on the clone, while recording the resulting power traces. Using statistical methods, they next build a "leakage profile" that models how the system behaves for each possible key value. Comparing the models and the recorded power traces then allow to deduce the value of the secret.

1.4.3 Leakage Assessment

Before performing an attack, an attacker may run a leakage assessment to identify the presence of side-channel leakage and determine the samples in the power traces where it occurs.

Welch's t-test Welch's t-test is a statistical method used to assess whether there is a significant difference between the means of two independent samples. It is a more robust adaptation of Student's t-test, particularly suited for situations where the two populations have unequal variances and possibly unequal sample sizes.

Here, $\mathcal{S}_0$ and $\mathcal{S}_1$ denote two sets of respective cardinalities n_0 and n_1, with means μ_0 and μ_1, and variances σ_0^2 and σ_1^2. The Welch's t-test result, commonly known as t-value is computed as follows:

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\left(\frac{\sigma_0^2}{n_0} + \frac{\sigma_1^2}{n_1}\right)}} \tag{1.1}$$

It is statistically proven that a threshold value of $|t| = 4.5$ represents a classifier with a confidence greater than 99.999%. An absolute t-value exceeding this threshold indicates that the two distributions S_0 and S_1 are statistically distinguishable [DCE16]. Conversely, an absolute t-value below this threshold suggests that the two sets cannot be distinguished by a linear classifier. Welch's t-test is commonly used in side-channel analysis to detect leakage. By varying an input or parameter between two sets of recorded traces, the resulting t-value indicates whether the variation correlates with observable leakage. In the remainder of this thesis, we will exclusively use Welch's t-test. Therefore, we will simply refer to it as the t-test.

ANOVA (Analysis of Variance) The ANOVA [YJ21] is a statistical test used to determine whether the means of multiple groups of power traces differ significantly from each other. If this is the case, and if the traces are grouped according to an intermediate value, then it reveals the presence of data-dependent leakage. Unlike the t-test, the ANOVA is not limited to the comparison of two groups.

1.4.4 CMOS logic

This section provides a short insight on why electronic components leak information through power consumption and, by extension, electromagnetic radiation. Current digital electronic systems are built using the CMOS (Complementary Metal Oxide Semiconductor) technology. CMOS components, such as memory and processors, are made of transistors that behave like controllable switches. Depending on the voltage applied on its control input (the *grid*) a transistor lets the current flow through it or blocks it.

We distinguish two types of power consumption in a CMOS system.

- The static power consumption (P_{stat}), is due to all the small leakages between the power supply and the ground of the circuit. The more transistors in the system, the higher the static power consumption.
- The dynamic power consumption (P_{dyn}), occurs when a transistor becomes passing and the current flows trough it to charge (or discharge) its output capacitance.

In general $P_{dyn} \gg P_{stat}$; the power consumption that can be observed with an equipment like an oscilloscope is primarily dynamic. It is also the one that is most correlated with the activity of the circuit, the operations it computes, and the data it manipulates. So, most power SCA exploit the dynamic power consumption. It is less common but the static power consumption can also be exploited to uncover secrets [LKMM21].

1.5 Goals of the Thesis

As we have seen, the threat posed by quantum computers has motivated a worldwide competition to develop PQC. The goal of this thesis is to analyse an algorithm competing in the fourth round of NIST selection process. This decision was motivated by the fact that lattice-based algorithms, having been selected for standardisation, already received significant attention from the community. Among the code-based candidates BIKE and HQC were the most promising and the best suited for embedded devices. We ultimately chose to focus on HQC due to its faster execution speed.

Since resistance to SCA is mandatory for future standards, our first objective is to obtain a comprehensive understanding of implementation attacks and to explore potential improvements to some of them. Then, our goal is to provide the community with a proposal for a secure implementation, while identifying potential pain points. In particular, we employ masking, one of the most widely used countermeasures against SCA. Its principle is to randomise intermediate computations in order to decorrelate them from sensitive data. This work culminates in the design of the first fully-masked implementation of HQC.

1.6 Contributions and Publications

1.6.1 Contributions

The main contributions of this thesis are as follows:

- Comprehensive survey of SCA on HQC and first complete sensitivity analysis of the scheme.
- Improved version of Guo *et al.*'s attack [GHJ+22], requiring 3× fewer decapsulations and a new version working against constant-time implementations.
- First fully-masked implementation of HQC.

1.6.2 Publications

The work presented in this thesis has led to two publications:

- *Masked Vector Sampling for HQC* *Maxime Spyropoulos, David Vigilant, Fabrice Perion, Renaud Pacalet, Laurent Sauvage In Proceedings of the 22nd International Conference on Security and Cryptography, ISBN 978-989-758-760-3, ISSN 2184-7711, pages 750-758.*

  In 2022, Guo *et al.* introduced a timing attack that exploited a weakness in HQC rejection sampling function to recover its secret key in 866,000 calls to an oracle. The authors of HQC updated its specification by applying an algorithm to sample vectors in constant time. A masked implementation of this function was later proposed for BIKE but it is not directly applicable to HQC. In this paper, we propose a specification compliant masked version of the HQC vector sampling which relies, to our knowledge, on the first masked implementation of the Barrett reduction.

- *Side-Channel Sensitivity Analysis on HQC: Towards a Fully Masked Implementation* *Guillaume Goy, Maxime Spyropoulos, Nicolas Aragon, Philippe Gaborit, Renaud Pacalet, Fabrice Perion, Laurent Sauvage, David Vigilant*

This paper presents two major contributions to secure HQC against Side-Channel Attacks. First, we present a detailed sensitivity analysis of HQC, highlighting the critical variables and critical internal functions that need to be protected. Second and main contribution, we propose the first fully masked HQC implementation. It is also the first PQC masked implementation that is formally proved to be secure in the MIMO-SNI security model. This security, introduced by Cassiers and Standaert in 2020, ensures the security of gadgets composition against propagating probes [CS20]. In this paper, we provide benchmarks of our implementation, showing that our masked implementation is competitive in the state-of-the-art masked PQC implementations.

1.7 Structure of the Thesis

The remainder of this thesis is organized as follows. Chapter 2 recalls all the necessary notions to understand both HQC, its sensitivity analysis and how it can be protected against SCA. Chapter 3 presents our contributions regarding the side-channel evaluation of HQC. Chapter 4 details the work carried out to achieve a masked implementation of HQC. Chapter 5 summarizes the benchmarking of our implementation and its practical security analysis. Finally, Chapter 6 concludes the thesis.

Chapter 2

Background and state of the art

Contents

2.1 Notation

Vectors are denoted with bold lower-case letters, matrices with bold upper-case letters. Polynomials are interchangeably considered as such or as row vectors of their coefficients.

2.2 Error-correcting codes

Coding theory originates from the field of information theory, formalised by Claude Shannon in 1948 [Sha48]. In this article, Shannon demonstrates that digital information can be transmitted reliably over noisy channels, provided that appropriate redundancy is introduced. Building on this foundation, coding theory studies how to design such redundancy mechanisms. In particular, error-correcting codes are a set of techniques used to transmit information over a noisy channel. They allow to detect and eventually correct errors due to noise or interferences during the transmission. One example is the NATO (North Atlantic Treaty Organization) phonetic alphabet where each letter is replaced by a word with same

Symbol	Description
$\mathbb{N}$	Set of non-negative integers
$\mathbb{F}_q$	Field with q elements
$\mathcal{R}$	Polynomial ring $\mathbb{F}_2[X]/(X^n - 1)$
$\mathtt{HW}(\boldsymbol{v})$	Hamming weight of vector $\boldsymbol{v}$ (number of non-zero components)
$\boldsymbol{v}^\top$	Transpose of vector $\boldsymbol{v}$
$\mathcal{R}_\omega$	$\{\boldsymbol{p} \in \mathcal{R} \mid \mathtt{HW}(\boldsymbol{p}) = \omega\}$ (set of polynomials in $\mathcal{R}$ with Hamming weight ω)
$\xleftarrow{\$}{}^{\theta}\mathcal{R}$	Random sampling of an element from $\mathcal{R}$, seeded with θ
$\lvert\mathcal{S}\rvert$	Size of set $\mathcal{S}$
$\lvert x\rvert$	Absolute value of number x
$\boldsymbol{a}\lVert\boldsymbol{b}$	Concatenation of vectors $\boldsymbol{a}$ and $\boldsymbol{b}$
$\boldsymbol{v} \ggg n$	Right rotation (circular shift) of vector $\boldsymbol{v}$ by n positions

Table 2.1: Notation and Symbols

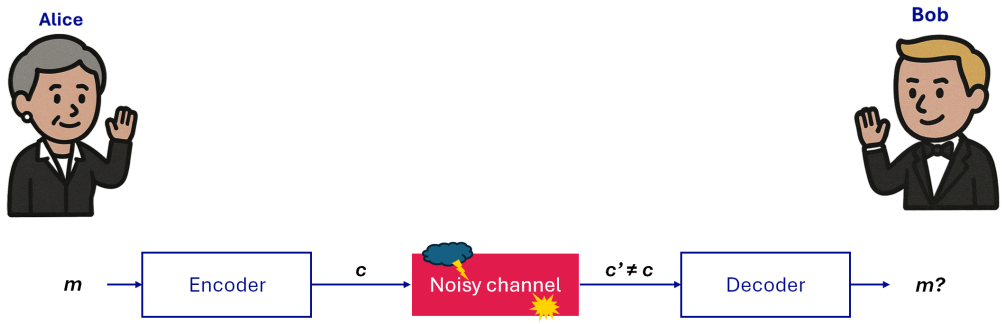


Figure 2.1: Message transmission using an error-correcting code

initial letter, like MIKE for M or NOVEMBER for N, to avoid confusion between letters which names sound similar. Each word is carefully chosen to be as different as possible from the others; the *distance* between words is maximized.

Error-correcting codes are useful not only to transmit information but also for long-term storage on physical devices like Blu-ray discs or hard drives.

Definition 2.2.1: Linear code

Let $k, n \in \mathbb{N}$ with $k \leq n$. An $\mathbb{F}_q$-linear code $\mathcal{C}$ of length n and dimension k is a linear subspace of dimension k of $\mathbb{F}_q^n$; that is, a subset of $\mathbb{F}_q^n$ that is closed under vector addition and scalar multiplication over $\mathbb{F}_q$ and contains exactly q^k elements.

Definition 2.2.2: Minimal distance

The minimal distance (denoted d) of a code $\mathcal{C}$ is the smallest distance between two distinct elements (codewords) of $\mathcal{C}$.

A linear code of length n, dimension k and minimal distance d is designated as a $[n, k, d]$ code.

Definition 2.2.3: Generator matrix

$\boldsymbol{G} \in \mathbb{F}_q^{k \times n}$ is a generator matrix for the $[n, k, d]$ code $\mathcal{C}$ if

$$\mathcal{C} = \{\boldsymbol{mG} \mid \boldsymbol{m} \in \mathbb{F}_q^k\} \tag{2.1}$$

Example. (with q=2) Alice and Bob communicate through a noisy channel. They use code $\mathcal{C}$ to correct transmission errors. To send Bob a k bits message $\boldsymbol{m}$ Alice maps $\boldsymbol{m}$ to a n bits codeword $\boldsymbol{c} = \boldsymbol{mG}$ (encoding) using the generator matrix $\boldsymbol{G}$ of the code. Alice sends $\boldsymbol{c}$ to Bob. Due to the noise Bob receives $\boldsymbol{c'} \neq \boldsymbol{c}$, a corrupted version of Alice's original codeword $\boldsymbol{c}$. Exploiting the structure of $\mathcal{C}$, Bob tries to recover $\boldsymbol{m}$ from $\boldsymbol{c'}$ (decoding).

Definition 2.2.4: Parity-check matrix

Given an $[n, k, d]$ code $\mathcal{C}$, $\boldsymbol{H} \in \mathbb{F}_q^{(n-k) \times n}$ is a parity-check matrix for $\mathcal{C}$ if $\boldsymbol{H}$ is a generator matrix of the dual code $\mathcal{C}^\perp$ or, equivalently, if

$$\mathcal{C} = \{\boldsymbol{c} \in \mathbb{F}_q^n \mid \boldsymbol{Hc}^\top = 0\} \tag{2.2}$$

Definition 2.2.5: Syndrome

Let $\mathcal{C}$ be a $[n, k, d]$ code, $\boldsymbol{H} \in \mathbb{F}_q^{(n-k) \times n}$ its parity-check matrix, and $\boldsymbol{c} \in \mathbb{F}_q^n$. Then the syndrome of $\boldsymbol{c}$ is $\boldsymbol{Hc}^\top$ and:

$$\boldsymbol{c} \in \mathcal{C} \Leftrightarrow \boldsymbol{Hc}^\top = 0$$

Definition 2.2.6: Hamming distance

The Hamming distance between two codewords of $\mathbb{F}_q^n$ is the number of coordinates where they differ.

Proposition 1. *Let $\mathcal{C}$ be a $[n, k, d]$ code and $t < d/2$. The Hamming balls with radius t and centred on the codewords of $\mathcal{C}$ are disjoint. A (theoretical) decoding strategy that selects the codeword that is closest to the received word (in terms of Hamming distance) correctly decodes any word with up to t errors.*

Definition 2.2.7: Support

Let $\boldsymbol{v} = \{v_1, v_2, \ldots, v_n\} \in \mathbb{F}_q^n$. The support of $\boldsymbol{v}$, denoted $\mathrm{Supp}(\boldsymbol{v})$, is defined as:

$$\mathrm{Supp}(\boldsymbol{v}) = \{i \in \{1, \ldots, n\} \mid v_i \neq 0\}$$

Definition 2.2.8: Spectrum

Let $\boldsymbol{v} \in \mathbb{F}_q^n$, $\omega = \mathtt{HW}(\boldsymbol{v})$, and $\mathrm{Supp}(\boldsymbol{v}) = \{s_1, \ldots, s_\omega\}$. The spectrum of $\boldsymbol{v}$ denoted $\mathrm{Spec}(\boldsymbol{v})$ is defined as

$$\mathrm{Spec}(\boldsymbol{v}) = \{|s_i - s_j|, 1 \leq i < j \leq \omega\}$$

Definition 2.2.9: Cyclic code

Let T :

$$\mathbb{F}_q^n \to \mathbb{F}_q^n$$
$$(x_1, x_2, \ldots, x_n) \mapsto (x_n, x_1, \ldots, x_{n-1})$$

be the cyclic shift map. Let $\mathcal{C}$ be a $[n, k, d]$ code.

$$\mathcal{C} \text{ is cyclic} \iff \text{For all } \boldsymbol{c} \in \mathcal{C}, \quad T(\boldsymbol{c}) \in \mathcal{C}$$

Definition 2.2.10: Quasi-cyclic code

Let $\mathcal{C}$ be a $[n, k, d]$ code and T the cyclic shift map. $\mathcal{C}$ is QC (Quasi-Cyclic) of index s (or s-QC) if there exist an integer s dividing n such that:

$$\forall \boldsymbol{c} \in \mathcal{C}, \quad T^s(\boldsymbol{c}) \in \mathcal{C}$$

where T^s denotes the composition of T applied s times.

s-QC codes have a compact representation because the generator matrix can be deduced from the first row via rotations ($\boldsymbol{g_1}, \ldots, \boldsymbol{g_s} \in \mathbb{F}_q^n$):

$$\boldsymbol{G} = \begin{bmatrix} \boldsymbol{g_1} & \boldsymbol{g_2} & \cdots & \boldsymbol{g_s} \\ \boldsymbol{g_s} & \boldsymbol{g_1} & \cdots & \boldsymbol{g_{s-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{g_2} & \boldsymbol{g_3} & \cdots & \boldsymbol{g_1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{g_1} & \boldsymbol{g_2} & \cdots & \boldsymbol{g_s} \end{bmatrix}$$

Definition 2.2.11: Reed — Muller code [Ree54]

Let $m, r \in \mathbb{N}$ with $0 \leq r \leq m$. Let $\mathbb{F}_2[x_1, \ldots, x_m]$ be the set of polynomials of m variables $x_1, \ldots, x_m$ with coefficients in $\mathbb{F}_2$. Let $\mathbb{F}_2[x_1, \ldots, x_m]/r = \{f \in \mathbb{F}_2[x_1, \ldots, x_m] \mid \deg(f) \leq r\}$ (polynomials of $\mathbb{F}_2[x_1, \ldots, x_m]$ with degree less or equal r). The RM (Reed — Muller) code of order r and length 2^m is defined as:

$$\texttt{RM}(r, m) = \left\{ (f(a))_{a \in \mathbb{F}_2^m} \mid f \in \mathbb{F}_2[x_1, \ldots, x_m]/r \right\}$$

Each codeword corresponds to a polynomial $f \in \mathbb{F}_2[x_1, \ldots, x_m]/r$, and is represented by the 2^m evaluations of f on the 2^m points of $\mathbb{F}_2^m$. It is thus a 2^m bits string. The dimension of the code is the number of distinct monomials of degree $\leq r$ of m variables:

$$k = \sum_{i=0}^{r} \binom{m}{i}$$

Its minimal distance is $d = 2^{m-r}$ [Ree54].

Definition 2.2.12: Reed-Solomon code [RS60]

Let $\mathbb{F}_q$ be the finite field with q elements. Let $\alpha_1, \ldots, \alpha_n \in \mathbb{F}_q$ be n distinct elements of the field, with $n < q$. A RS (Reed — Solomon) code of length n and dimension k is defined as:

$$\texttt{RS}(n,k) = \{(f(\alpha_1), f(\alpha_2), \ldots, f(\alpha_n)) \mid f \in \mathbb{F}_q[X],\ \deg(f) < k\}$$

The minimal distance of a $\texttt{RS}(n,k)$ code is $d = n - k + 1$ [RS60].

Definition 2.2.13: Concatenated code [For65]

A concatenated code is a code built from two simpler codes: an *outer code* and an *inner code*. The outer code $\mathcal{C}_o$ is a $[n_o, k_o, d_o]$ code over a a_o symbols alphabet, and the inner code $\mathcal{C}_i$ is a $[n_i, k_i, d_i]$ code over a a_i symbols alphabet, where $a_o = a_i^{k_i}$ (each symbol of $\mathcal{C}_o$ can be mapped to an input of $\mathcal{C}_i$ and encoded by $\mathcal{C}_i$).

A well known property of concatenated codes is that their distance is at least the product of the distances of the outer and inner codes. The concatenated code is thus a $[N = n_o n_i, K = k_o k_i, D \geq d_o d_i]$ code over the alphabet of the inner code.

Example. Let $\mathcal{C}_o$ be a $[n_o, k_o, d_o]$ code over $\mathbb{F}_q$ and $\mathcal{C}_i$ a $[n_i, k_i, d_i]$ code over $\mathbb{F}_2$, with $q = 2^{k_i}$. To encode a message $\boldsymbol{m} = (m_1, \ldots, m_{k_o}) \in \mathbb{F}_q^{k_o}$, it is first encoded with $\mathcal{C}_o$, producing an intermediate codeword $\boldsymbol{a} = \boldsymbol{m}\boldsymbol{G_o} = (a_1, \ldots, a_{n_o}) \in \mathbb{F}_q^{n_o}$. Each a_i is mapped to a $\boldsymbol{b_i} = \phi(a_i) \in \mathbb{F}_2^{k_i}$ thanks to a bijection ϕ from $\mathbb{F}_q$ to $\mathbb{F}_2^{k_i}$, and encoded with $\mathcal{C}_i$ ($\boldsymbol{c_i} = \boldsymbol{b_i}\boldsymbol{G_i}$). The n_o output blocks are finally concatenated to form the final codeword:

$$\boldsymbol{c} = \boldsymbol{c_1} \| \boldsymbol{c_2} \| \cdots \| \boldsymbol{c_{n_o}} \in \mathbb{F}_2^{n_i n_o}$$

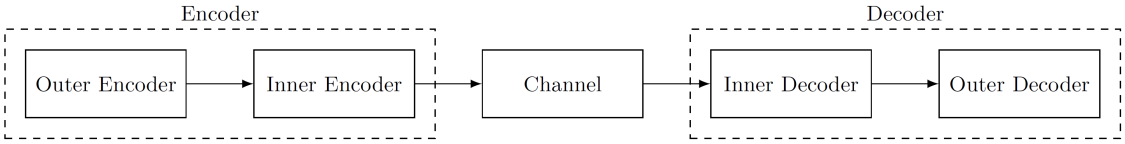


Figure 2.2: Schematic depiction of a concatenated code

2.3 Hard problems in codes and code-based cryptography

In this section we detail some mathematically hard problems in coding theory.

In 1948, Shannon observed that decoding a random code (i.e., a code without any structure) was a hard problem [Sha48]. Thirty years later, McEliece extended this by noting that decoding a code without any known structure was also hard [McE78]. This observation was the starting point of code-based cryptography. McEliece's idea was to hide a trapdoor (the secret key) in the generator matrix of a code (the public key). The generator matrix enabled anyone to encrypt a message with it, but only someone with knowledge of the trapdoor could decode the ciphertext in polynomial time.

Definition 2.3.1: Syndrome Decoding Distribution

Let $n, k, \omega \in \mathbb{N}$. The Syndrome Decoding Distribution $\texttt{SDD}(n, k, \omega)$ samples $\boldsymbol{H} \in \mathbb{F}_2^{(n-k)\times n}$ and $\boldsymbol{x} \in \mathbb{F}_2^n$ such that $\texttt{HW}(\boldsymbol{x}) = \omega$, computes $\boldsymbol{y}^\top = \boldsymbol{H}x^\top$ and outputs $(\boldsymbol{H}, \boldsymbol{y})$.

Definition 2.3.2: Syndrome Decoding Problem

Given $(\boldsymbol{H}, \boldsymbol{y}) \in \mathbb{F}_2^{(n-k)\times n} \times \mathbb{F}_2^{(n-k)}$ from the $\texttt{SDD}(n, k, \omega)$ distribution, the Syndrome Decoding Problem $\texttt{SDP}(n, k, \omega)$ asks to find $\boldsymbol{x} \in \mathbb{F}_2^n$ such that $\boldsymbol{y}^\top = \boldsymbol{H}\boldsymbol{x}^\top$ and $\texttt{HW}(\boldsymbol{x}) = \omega$.

Definition 2.3.3: Decisional Syndrome Decoding Problem

Given $(\boldsymbol{H}, \boldsymbol{y})$ from the $\texttt{SDD}(n, k, \omega)$ distribution, the Decisional Syndrome Decoding Problem $\texttt{DSDP}(n, k, \omega)$ asks to decide with non-negligible advantage whether $(\boldsymbol{H}, \boldsymbol{y})$ came from the $\texttt{SDD}(n, k, \omega)$ distribution or the uniform distribution over $\mathbb{F}_2^{(n-k)\times n} \times \mathbb{F}_2^{(n-k)}$.

The SDP was proven NP-complete by Berlekamp *et al.* [BMvT78]. Informally, this means that it is impossible to decode all codes at any decoding distance in polynomial time. The DSDP has been shown to be polynomially equivalent to the SDP in [AIK07].

2.4 HQC

HQC [AMAB+17] is a code-based post-quantum KEM. It was selected on March 2025 by the NIST to become a new standard for PQC [AAC+25]. It has been noticed especially for its "*DFR*" which "*has remained stable throughout the NIST PQC standardization process*".

HQC is a KEM based on a PKE scheme, thanks to the HHK (Hofheinz — Hovelmanns — Kiltz) transform [HHK17], a variant of the well-known FO transform [FO99]. HQC uses two codes: *(i)* a random code to ensure the security of secret keys and *(ii)* an efficient public code for encryption and decryption. This public code $\mathcal{C}$ is formed by concatenation of a duplicated RM code with a shortened RS code. In the following we denote $\boldsymbol{G}$ its generator matrix. Unlike cryptosystems based on the McEliece construction, there is no trapdoor hidden in the decoding code.

HQC-PKE consists of three algorithm: Key Generation (1), Encryption (2) and Decryption (3). These algorithms are summarized in Figure 2.3.

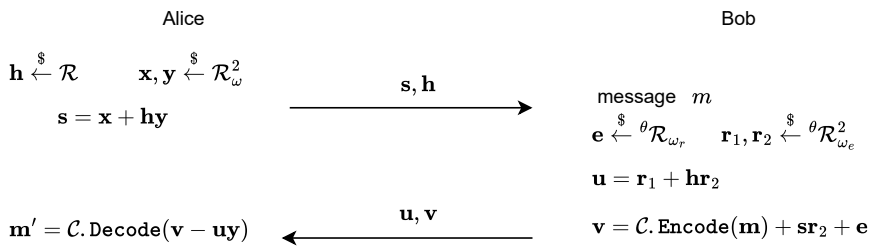


Figure 2.3: Summarized HQC-PKE scheme

2.4.1 Key Generation

The key generation algorithm is described in Algorithm 1. It generates two pairs of vectors of $\mathcal{R}$: $\mathsf{pk} = (\boldsymbol{h}, \boldsymbol{s})$ make up the public key and $\mathsf{sk} = (\boldsymbol{x}, \boldsymbol{y})$ make up the secret key. $\boldsymbol{h}$ defines the random code and is randomly sampled in $\mathcal{R}$, $\boldsymbol{x}$ and $\boldsymbol{y}$ are randomly sampled in $\mathcal{R}_\omega$.

In the implementation, pk and sk are stored as two 40 bytes seeds (respectively $\mathsf{pk\_seed}$ and $\mathsf{sk\_seed}$) that allow to re-generate $(\boldsymbol{h}, \boldsymbol{s})$ and $(\boldsymbol{x}, \boldsymbol{y})$.

Algorithm 1 KeyGen

Input: $n, k, \omega, l, \mathsf{pk\_seed}, \mathsf{sk\_seed}$
$\boldsymbol{h} \xleftarrow{\$}{}^{\mathsf{pk\_seed}}\mathcal{R}$
$\boldsymbol{\sigma} \xleftarrow{\$} \mathbb{F}_2^{512}$
$(\boldsymbol{x}, \boldsymbol{y}) \xleftarrow{\$}{}^{\mathsf{sk\_seed}}\mathcal{R}_\omega^2$
$\mathsf{sk} \leftarrow (\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\sigma})$
$\boldsymbol{s} \leftarrow \boldsymbol{x} + \boldsymbol{h}\boldsymbol{y}$
$\mathsf{pk} \leftarrow (\boldsymbol{h}, \boldsymbol{s})$
return $(\mathsf{sk}, \mathsf{pk})$

2.4.2 Encryption

The encryption algorithm is described in Algorithm 2. It generates a ciphertext composed of two vectors: $\boldsymbol{u}$ and $\boldsymbol{v}$. $\boldsymbol{u}$ is derived from $\boldsymbol{h}$, $\boldsymbol{r_1}$ and $\boldsymbol{r_2}$. $\boldsymbol{v}$ is obtained from $\boldsymbol{s}$, $\boldsymbol{r_2}$, $\boldsymbol{e}$ and $\boldsymbol{mG}$, the encoding of the message to encrypt $\boldsymbol{m}$.

Algorithm 2 Encrypt

Input: pk, $\boldsymbol{m}, \theta, \omega_r, \omega_e$
$\boldsymbol{e} \xleftarrow{\$}{}^{\theta}\mathcal{R}_{\omega_e}$
$(\boldsymbol{r_1}, \boldsymbol{r_2}) \xleftarrow{\$}{}^{\theta}\mathcal{R}_{\omega_r}^2$
$\boldsymbol{u} \leftarrow \boldsymbol{r_1} + \boldsymbol{h}\boldsymbol{r_2}$
$\boldsymbol{v} \leftarrow \mathcal{C}.\mathrm{Encode}(\boldsymbol{m}) + \boldsymbol{s}\boldsymbol{r_2} + \boldsymbol{e}$
$\mathsf{ct} \leftarrow (\boldsymbol{u}, \boldsymbol{v})$
return ct

2.4.3 Decryption

The decryption algorithm is described in Algorithm 3. It consists of decoding the vector $\boldsymbol{v} - \boldsymbol{uy}$. Note that although the secret key additionally consists of $\boldsymbol{x}$, most attacks concentrate on retrieving $\boldsymbol{y}$, as it is the only secret needed for a successful decryption.

Correctness

From the specifications:

$$\begin{aligned}
\mathcal{C}.\mathrm{Decode}(\boldsymbol{v} - \boldsymbol{uy}) &= \mathcal{C}.\mathrm{Decode}((\mathcal{C}.\mathrm{Encode}(\boldsymbol{m}) + \boldsymbol{s}\boldsymbol{r_2} + \boldsymbol{e}) - (\boldsymbol{r_1} + \boldsymbol{h}\boldsymbol{r_2})\boldsymbol{y}) && (2.3)\\
&= \mathcal{C}.\mathrm{Decode}((\mathcal{C}.\mathrm{Encode}(\boldsymbol{m}) + (\boldsymbol{x} + \boldsymbol{h}\boldsymbol{y})\boldsymbol{r_2} + \boldsymbol{e}) - (\boldsymbol{r_1} + \boldsymbol{h}\boldsymbol{r_2})\boldsymbol{y}) && (2.4)\\
&= \mathcal{C}.\mathrm{Decode}(\mathcal{C}.\mathrm{Encode}(\boldsymbol{m}) + \boldsymbol{x}\boldsymbol{r_2} + \boldsymbol{e} - \boldsymbol{r_1}\boldsymbol{y}) && (2.5)
\end{aligned}$$

Name	Description	Security Level HQC-128	HQC-192	HQC-256
mul	multiplicity of RM code	3	5	5
k_1	dimension of RM code	16	24	32
k_2	dimension of RS code	8	8	8
n_1	length of RM code	46	56	90
n_2	length of RS code	384	640	640
$n_1 n_2$	length of concatenated RM-RS code	17664	35840	57600
n	length of the random code	17669	35851	57637
l	truncated bits ($l = n - n_1 n_2$)	5	11	37
δ	correction capacity of RS code	15	16	29
ω	weight of secret vectors	66	100	131
$\omega_e = \omega_r$	weight of error vectors	75	114	149
	public key size (in bytes)	2249	4522	7245
	secret key size (in bytes)	56	64	72
	ciphertext size (in bytes)	4433	8878	14421
	shared secret size (in bytes)	64	64	64

Table 2.2: HQC parameters from [AMAB+17]

And, if $\mathcal{C}$ correctly decodes:

$$\mathcal{C}.\text{Decode}(\boldsymbol{v} - \boldsymbol{uy}) = \boldsymbol{m} \tag{2.6}$$

Let Δ be the number of errors that the code can correct. The condition for correct decoding is:

$$\texttt{HW}(\boldsymbol{xr_2} + \boldsymbol{e} - \boldsymbol{r_1 y}) \leq \Delta \tag{2.7}$$

If Equation 2.7 does not hold a decryption failure can occur. HQC parameters are chosen so that the DFR is at most $2^{-\lambda}$, where λ is the security level (128, 192 or 256).

Algorithm 3 Decrypt

Input: sk, ct
return $\mathcal{C}.\text{Decode}(\boldsymbol{v} - \boldsymbol{uy})$

In HQC-KEM, the encryption function is de-randomized thanks to a hash of the message and public key, used as a seed. This allows to re-encrypt the message after decryption to verify if the received ciphertext is equal to the computed one (this step is required by the HHK transform). The session key, that corresponds to a hash of the concatenation of the message and the ciphertext, is shared only if the equality is verified. The KEM is composed of Encapsulation (Figure 4) and Decapsulation (Figure 5), where $\mathcal{G}, \mathcal{H}$ and $\mathcal{K}$ are cryptographically secure hash functions and pk_{32} represents the first 32 bytes of pk.

2.4.4 HQC security

HQC-PKE has been proved IND-CPA secure, and the application of the HHK transform, combined with the strong DFR analysis, allows HQC-KEM to reach IND-CCA2 security. The security of HQC relies on the hardness of solving the QC-SDP [BMvT78] without any

Algorithm 4 Encapsulate

Input: pk
$\boldsymbol{m} \xleftarrow{\$} \mathbb{F}_2^k$
$\mathtt{salt} \xleftarrow{\$} \mathbb{F}_2^{128}$
$\theta \leftarrow \mathcal{G}(\boldsymbol{m} \parallel \mathsf{pk}_{32} \parallel \mathtt{salt})$
$\boldsymbol{c} \leftarrow \text{Encrypt}(\mathsf{pk}, \boldsymbol{m}, \theta)$
$K \leftarrow \mathcal{K}(\boldsymbol{m}, \boldsymbol{c})$ ▷ shared key
return $(K, \boldsymbol{c}, \mathtt{salt})$

Algorithm 5 Decapsulate

Input: $\mathsf{sk}, \boldsymbol{c}, \mathtt{salt}$
$\boldsymbol{m'} \leftarrow \text{Decrypt}(\mathsf{sk}, \boldsymbol{c})$
$\theta' \leftarrow \mathcal{G}(\boldsymbol{m'} \parallel \mathsf{pk}_{32} \parallel \mathtt{salt})$
$\boldsymbol{c'} \leftarrow \text{Encrypt}(\mathsf{pk}, \boldsymbol{m'}, \theta')$
if $\boldsymbol{c} \neq \boldsymbol{c'}$ **then**
  return $\boldsymbol{\sigma}$
else
  return $\mathcal{K}(\boldsymbol{m}, \boldsymbol{c})$
end if

other assumptions. More details on the security reduction proofs can be found in section 5 of HQC documentation [AMAB+17].

2.5 Side-channel attacks on HQC

Although no quantum attacks are known against HQC, its implementations have been shown to be vulnerable to physical attacks, particularly SCA. Since 2019, these attacks have become more and more effective. The number of observations required to recover secret information has been progressively reduced. SCA against HQC fall in two main categories: timing attacks and power analysis attacks.

2.5.1 Timing attacks

The BCH (Bose–Chaudhuri–Hocquenghem codes) decoder used in the reference implementation of HQC submitted for the first round of the competition was not constant-time.

In [PT19], authors noticed that the decryption time depended on the number of errors being decoded. With this information and 400 million ciphertexts they were able to recover the spectrum of the secret key vector $\boldsymbol{y}$. They then used a result from [GJS16] to reconstruct a sparse vector from its spectrum.

Similarly, authors from [WTBB+20] proposed a chosen-ciphertext attack that exploits the correlation between the weight of the error to be decoded and the runtime of the decoding algorithm. Using a dichotomic search and an oracle that detects whether an error was decoded or not, they recover the value of $\boldsymbol{y}$ in 5400 to 6600 queries (depending on the security level). They also propose a countermeasure that achieves constant time decoding with an overhead of less than 11%.

Although HQC switched from the BCH decoder to the current RM-RS decoder in 2020 (third round submission), data-dependent computation times persisted in implementations. The June 2021 HQC specification suffered from a timing flaw in the encryption function, which relies on the sampling of three small weight random vectors. The support of each vector was computed from random bytes generated by an XOF (eXtendable Output Function) called `seedexpander`, seeded with a hash of the message $\boldsymbol{m}$ to encrypt. In most cases there was no collision between the indexes drawn and `seedexpander` was called only 3 times, once per vector. However, if at least one collision occurred, a second call to `seedexpander` was made (see Algorithm 6). This is because, ***rand_bytes*** only contains an amount of randomness sufficient for ω drawings. If a collision occurs on Line 13, the algorithm will run out of random bytes and will need to fetch new ones (Line 5). This variation in the number of calls was noticeable in timing, and related to the value of the

input message $\boldsymbol{m}$. The authors used this as a distinguisher to perform a timing attack and recover the secret key in $\approx \mathbf{8.7 \times 10^5}$ decapsulations [GHJ+22].

Algorithm 6 HQC not constant-time sampling

Input: ω, n and ctx the context of the seed expander
Output: $\boldsymbol{v}[0], \ldots, \boldsymbol{v}[\omega - 1] \in \{0, \ldots, n-1\}$

$i \leftarrow 0$
$j \leftarrow 3 \times \omega$
while $i < \omega$ **do**
  if $j == 3 \times \omega$ **then**
    $\boldsymbol{rand\_bytes} \leftarrow$ `seedexpander`$(ctx, 3 \times \omega)$ ▷ draw $3 \times \omega$ random bytes
    $j \leftarrow 0$
  end if
  $\boldsymbol{v}[i] \leftarrow (\boldsymbol{rand\_bytes}[j] \ll 16) \oplus (\boldsymbol{rand\_bytes}[j+1] \ll 8) \oplus \boldsymbol{rand\_bytes}[j+2]$
  $j \leftarrow j + 3$
  $\boldsymbol{v}[i] \leftarrow \boldsymbol{v}[i]$ `mod` n
  $inc \leftarrow 1$
  for $k = 0$ **to** $i - 1$ **do**
    if $\boldsymbol{v}[k] == \boldsymbol{v}[i]$ **then**
      $inc \leftarrow 0$
    end if
  end for
  $i \leftarrow i + inc$
end while
return $\boldsymbol{v}[0], \ldots, \boldsymbol{v}[\omega - 1]$

Guo *et al.* suggested using an algorithm designed by Sendrier [Sen21] to sample vectors in constant time in order to patch this vulnerability. Following this recommendation, the HQC team explicitly included this countermeasure in their 2023 update of the scheme. The idea behind this new sampling is to always ask for the same amount of randomness and handle potential collisions with the addition of a small bias. This bias was shown to have no significant impact on the output distribution [Sen21]. The countermeasure is detailed in Algorithm 7 where `compare` is a constant-time function that returns `0xF...F` if the two input values are equal and `0x0...0` otherwise.

Later, Leander *et al.* [SGG24] warned that, in some cases, the modulo operation used in the countermeasure was replaced by compilers with a division instruction. When the divisor is known at compile-time, compilers sometimes optimize modulo operations by transforming them into constant-time Barrett reductions. However, this optimization is not guaranteed: compilers may instead emit division instructions to carry out the computation. Depending on the target processor and on the value of the numerator, division instructions can have variable execution time, which makes the countermeasure constant-time in theory only. To address this, the authors proposed computing the modulo using a hard-coded Barrett reduction.

2.5.2 Power analysis attacks

In [SRSWZ20] Schamberger *et al.* built an oracle using a power side-channel targeting the BCH decoder. The oracle indicates whether the decoder corrected an error or not. With carefully chosen ciphertexts, they exploited the collisions between the support of $\boldsymbol{y}$ and $\boldsymbol{v}$ to recover $\boldsymbol{y}$. When HQC switched to a RM-RS decoder, the authors adapted their

Algorithm 7 Sendrier's constant time vector sampling in HQC

Input: $s, seed, n$
Output: $support[0], \ldots, support[s-1]$

```
prng ← prng_init(seed)
for i = 0 to s − 1 do
    support[i] ← i + rand(prng) mod (n − i)          ▷ support[i] ∈ [i, n[
end for
for i = s − 1 downto 0 do
    found ← 0                                        ▷ collision flag
    for j = i + 1 to s − 1 do
        found ← found ∨ compare(support[i], support[j])
    end for
    support[i] ← (i ∧ found) ⊕ (support[i] ∧ ¬found)
end for
return support[0], ..., support[s − 1]
```

strategy to attack the new design [SHR+22]. Most of the time, the RM decoder corrects all the errors and the RS decoder processes an error-free codeword. But when errors persist after the RM decoder, the power consumption of the RS decoder is significantly different. It is then possible to build a new oracle, similar to the one in the previous attack, which in turn can be used to recover the secret key.

Also targeting the RS decoder, authors of [GLG22b] showed a vulnerability in the Galois field multiplication that enables the recovery of the 64 bytes of the shared secret key. They attack the input of the RS decoder (that is, the output of the RM decoder). At this point, the intermediate codeword is no longer a ciphertext, it was decrypted before being fed to the RM decoder. Assuming that the RS decoder processes an error-free codeword, Goy *et al.* described how to use a correlation analysis to recover it. Then, since the RS decoder is public, they used the recovered codeword to compute the exchanged message and deduce the shared secret key. Yet, this attack is not practical, as it requires 2^{96} Galois field operations [GLG22b].

In [GMGL23], authors exploited the same vulnerability, this time using a SASCA (Soft Analytical Side-Channel Attack) to recover the shared key in a single trace.

The RM decoder was targeted by [GLG22a] and [BMG+24]. In [GLG22a], the authors targeted the FHT (Fast Hadamard Transform) to build an oracle capable of revealing the number of corrected errors in a block. By sending ciphertexts where $\boldsymbol{u}$ is set to 1, the victim effectively decodes $\boldsymbol{v} - \boldsymbol{y}$, so the decoder has to correct errors that correspond to $\boldsymbol{y}$. With a divide and conquer strategy, they take advantage of the collisions between the support of $\boldsymbol{v}$ and the support of $\boldsymbol{y}$ to recover $\boldsymbol{y}$. In [BMG+24], the authors also target the FHT, but this time using a SASCA to recover both the secret key and the shared secret.

Leveraging soft information from the publicly available decoder, Dong and Guo build what they call *offline templates* [DG24]. Unlike traditionnal Template Attacks, these templates are not device-specific. Instead, they exploit the interactions that occur during the decoding between predefined error patterns and sparse vectors sampled from HQC secret key distribution. With these templates, Dong and Guo significantly reduce the number of oracle calls required in some attacks. For example, they report a 2.4× reduction in the number of calls needed in [SGG24] and an 87× reduction compared to [SHR+22].

While most SCA targeting PQC canditates were mounter on reference implementations, Maillet *et al.* proposed two attacks against the optimized implementation of

HQC [MNMD25]. Both attacks exploit the power consumption of the RM decoder. In the first, the authors analyze the power consumption at the start of the decoding process to recover the value of the ciphertext after noise removal. This information enables them to recover the secret key $\boldsymbol{y}$ in a single trace, assuming access to a side-channel oracle with 100% accuracy. Though the first attack was tested using a simulator, the second one was carried out under real-world conditions. The authors targeted an STM32F303 and measured power consumption with a ChipWhisperer. In this practical scenario, micro-architectural leakage allowed them to recover the secret key in 83 traces, with 99% success rate.

2.6 Masking

The state of the art in SCA against code-based PQC underscores the need for robust countermeasures, especially in embedded systems. Masking [CJRR99, GP99] is a widely used countermeasure to protect implementations of cryptographic algorithms against side-channel attacks. It consists in representing a sensitive variable x as a tuple of d shares $(x_1, \ldots, x_d)$, denoted $[\![x]\!]$ in the following. Among the d shares $d-1$ are chosen uniformly at random, and the last one such that the "sharing relation" is satisfied. Any set of $d-1$ shares is thus always statistically independent of any sensitive variable. $d-1$ is called the masking order. The most frequently encountered sharing relations are boolean:

$$x = \bigoplus_{i=1}^{d} x_i \tag{2.8}$$

and arithmetic:

$$x = \left(\sum_{i=1}^{d} x_i \right) \bmod q \tag{2.9}$$

In our work we select the sharing relation depending on the nature of the operations to mask, in order to reduce the masking overhead. Of course, we also consider the cost of conversions between boolean and arithmetic masking.

2.6.1 Security in the t-probing model

In the t-probing model [ISW03] an attacker has access to at most t-probes on the implementation, giving an access to t intermediate variables. By splitting the sensitive variables of an algorithm in $d > t$ shares, and maintaining the statistical independence during the execution, attackers cannot learn information. The algorithm is said t-probing secure if any set of t probes is statistically independent of the unmasked values of the sensitive inputs.

Proving the theoretical security is a hard problem that does not scale well with the number of variables and shares. A common solution is based on composability: break down the algorithm into elementary operations referred to as "gadgets", and to compose them to build larger functions [BBD+16]. The composing gadgets and the composition must obey rules that preserve the security.

We recall two commonly used security models, that we can define with the simulatability framework introduced in [BBP+16]:

- A gadget is t-NI (Non-Interfering) if and only if any set of $t' \leq t$ probes can be perfectly simulated from at most t' shares of each input [BBP+16].

- A gadget is t-SNI (Strong Non-Interfering) [BBD+16] if and only if any set of $t_{int} + t_{out} \leq t$ probes, where t_{int} is the number of internal probes and t_{out} is the number of output probes, can be perfectly simulated from at most t_{int} shares of each input.

Algorithm 8 A $(d-1)$-SNI version of the boolean d shares `refresh` gadget

Input: $[\![x]\!]$
Output: $[\![x]\!]$ with refreshed shares
1: **for** $i \in \{1 \dots d\}$ **do**
2: **for** $j \in \{i+1 \dots d\}$ **do**
3: $r \leftarrow \$$
4: $x_1 \leftarrow x_1 \oplus r$
5: $x_j \leftarrow x_j \oplus r$
6: **end for**
7: **end for**
8: **return** $[\![x]\!]$

Combining t-NI secure gadgets does not generally yield a t-NI algorithm: an attacker could exploit the correlation between reused shares across different masked operations to recover sensitive information. To maintain the security guarantees of masking throughout the execution of an algorithm, it is often necessary to perform *refresh* operations. The `refresh` gadget was first introduced by Rivain and Prouff [RP10] to re-randomize a masked value by generating new random shares that encode the same underlying secret. A SNI version of `refresh` (see Algorithm 8 for an example) is crucial since Barthe *et al.* demonstrated that it allows secure composition of gadgets [BBD+16]. It is therefore a critical component in masked implementations, as it prevents cumulative leakage and reinforces the security of the masked scheme against side-channel attacks.

Remark 1. Any d shares gadget implementing a linear operation trivially, that is, share-wise, is $(d-1)$-NI. A d shares gadget implementing a linear operation trivially, is generally **not** $(d-1)$-SNI: to perfectly simulate a probe on one share of an output, one would need the corresponding share of each input.

A well known result is that a gadget over $d > t$ shares satisfying t-NI or t-SNI is t-probing secure. A composition rule was established by Barthe *et al.* in Proposition 4 of [BBD+16,]:

Proposition. An algorithm is t-NI provided all its gadgets are t-NI, and all masked variables are used at most once as input of a gadget other than `refresh`.

In the following we sometimes write that a d shares gadget is NI or SNI for short, instead of $(d-1)$-NI or $(d-1)$-SNI. We represent algorithms as computation graphs, that is, DAGs (Directed Acyclic Graphs) where the composing gadgets, the inputs and the outputs are the vertexes, and the connections between them are the edges. In the figure representing a computation graph the light blue boxes are NI gadgets (□) and the light red boxes are SNI gadgets (□).

2.6.2 PINI and MIMO-SNI security

PINI (Probe Isolating Non-Interfering) and Multiple Inputs-Multiple Outputs Strong Non-Interfering are notions proposed by Cassiers and Standaert [CS20] that extend SNI to allow

the secure composition of gadgets. They were developed after the authors observed that the SNI model applies only to single output gadgets, and that the trivial implementation of a linear function is not SNI. As a result, the design of compositions involving linear functions is not straightforward.

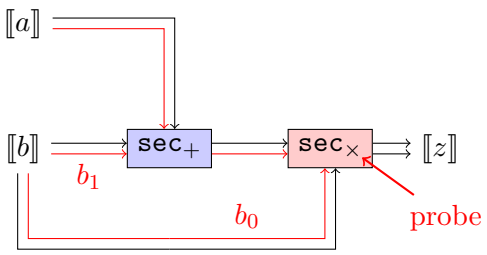
To reason about security, instead of using the simulatability definition from [BBP+16], they introduce the equivalent concept of back-propagating probes towards the inputs. The properties of NI and SNI gadgets can then be rephrased as:

Proposition 2. *For a t-NI gadget with n_o probes on its output shares and n_i internal probes with $n_i + n_o \leq t$, there are propagated probes on $n_i + n_o$ shares of each input.*

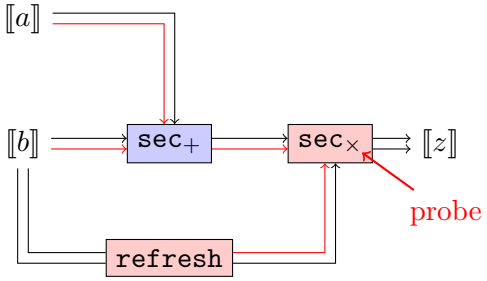
Proposition 3. *For a t-SNI gadget with n_o probes on its output shares and n_i internal probes with $n_i + n_o \leq t$, there are propagated probes on n_i shares of each input.*

SNI gadgets such as SNI `refresh` guarantee independence between input and output shares. Hence, SNI gadgets stop the propagation of probes towards the inputs.

Example. An illustration of these properties is shown on Figure 2.4 where $d = 2$. In Figure 2.4a, one internal probe in the SNI $\texttt{sec}_\times$ gadget propagates to one share of each of its inputs, finally leading to both b_0 and b_1. With this knowledge, the attacker can recover the secret value $[\![b]\!]$, because it was used twice without a `refresh`, thus violating the composition rule from [BBD+16]. Figure 2.4b shows how to correct this situation with a well placed `refresh` that prevents the propagation. This example illustrates the threat posed by propagating probes, especially the duplications caused by multiple, recombining, paths.



(a) Computation graph without `refresh`



(b) Computation graph with `refresh`

Figure 2.4: Illustration of propagating probes

In order to propose secure, yet simple, composition rules, Cassiers and Standaert [CS20] introduce the PINI security model. Different from SNI, it applies to multiple outputs gadgets. Another major difference is that PINI not only constrains the number of propagated probes, as in NI or SNI, but also their position: in a t-PINI gadget, if there are t_1 internal probes and if A is the set of output share indexes that are probed, with $|A| = t_2$ and

$t_1 + t_2 \leq t$, then there is a set B of share indexes with $|B| \leq t_1$ such that the probes (output and internal) propagate to input share indexes in $A \cup B$. This important constraint limits the duplication effect and guarantees that multiple paths, if any, do not increase the number of exposed shares. The authors prove that trivial implementations of linear gadgets are PINI, and that any composition of t-PINI gadgets is also t-PINI. They also prove that any d shares t-PINI gadget with $d > t$ is t-probing secure.

Remark 2. Any d shares gadget implementing a linear operation trivially is $(d-1)$-PINI.

As designing PINI gadgets from NI and SNI gadgets and verifying them is not easy the authors of [CS20] also introduce the MIMO-SNI model. They prove that MIMO-SNI implies PINI, so MIMO-SNI gadgets can be freely composed to form secure PINI larger gadgets. And they propose a small set of simple rules to design and verify MIMO-SNI gadgets from NI and SNI gadgets, based on the computation graph.

A gadget is t-MIMO-SNI if for any set of t_1 internal probes and any set of probes on output shares where the number t_2 of probes on each output is such that $t_1 + t_2 \leq t$, the probes can be perfectly simulated with at most t_1 input shares.

To ensure structural MIMO-SNI correctness, the computation graph forbids multiple edges connected to the same gadget output or to a single input. In cases where an input or a gadget output needs to be reused, the authors introduce a special vertex called `split` that we represent as a light green box (□) in the computation graphs. These vertexes have one input and n identical outputs, and they perform no operation. Without loss of generality, a SNI gadget other than the SNI refresh is implicitly considered as an equivalent NI gadget followed by the SNI refresh gadget.

By leveraging the connection between the computation graph and the probe propagation framework, and exploiting the fact that SNI `refresh` gadgets block the back-propagation of probes, the authors propose a simplification step. This consists in removing all SNI `refresh` gadgets and their incident edges from the graph. The internal probes of these `refresh` gadgets are conservatively reassigned to their inputs, preserving security equivalence in the probing model. This results in a **simplified graph** that remains faithful to the original structure from a security standpoint.

Finally, the authors demonstrate that a gadget composition satisfies the MIMO-SNI property if the following conditions are satisfied:

1. The composite gadget uses only single output NI gadgets, single output SNI gadgets, and single input multiple outputs `split` gadgets.
2. There exists no path between an input and an output.
3. Between any two vertexes of the simplified graph there is at most one path.
4. For any pair of output vertexes u_1, u_2 there is no vertex v such that there is a path from v to u_1 and a path from v to u_2.
5. For any pair of input vertexes u_1, u_2 there is no vertex v such that there is a path from u_1 to v and a path from u_2 to v.

The security-preserving compositions are a key advantage of the framework, as they enable scalable and modular design of secured masked implementations. The PINI and MIMO-SNI frameworks suggest two ways to securely implement large and complex algorithms like the ones of HQC:

1. Implement the complete algorithm from NI, SNI and `split` gadgets, according the above conditions. The result is MIMO-SNI which implies PINI.

2. Implement intermediate MIMO-SNI gadgets as building blocks, and compose them to obtain the full algorithm. The composite gadgets are MIMO-SNI, which implies PINI, so the result is also PINI.

In both approaches the final implementation is PINI and thus t-probing secure. As we will see in Chapter 4, considering the complexity of HQC we preferred a modular approach, that is, the second one.

2.6.3 Prior art regarding Countermeasures and secure implementations

HQC is not the only post-quantum cryptosystem requiring protection against side-channel leakage. Prior works explored masking implementations for other schemes.

Migliore et al. [MGTF19] proposed a masked version of the CRYSTALS-Dilithium lattice-based signature scheme in 2019. Three years later, Azouaoui et al. [ABC+22a] identified two flaws in this implementation: *(i)* unmasked sensitive variables, creating a security gap, and *(ii)* unnecessary masking of public information, degrading performance. Azouaoui et al. improved the sensitivity analysis by categorizing intermediate computations and their security requirements.

Regarding ML-KEM, the other NIST-selected KEM standard, several studies have referenced masked implementations or accelerations of masked operations, particularly focusing on the NTT (Number Theoretic Transform). Notably, a number of works have proposed first-order masked implementations, such as [HKL+22] and [FVBR+22], the latter of which also includes first-order masking for SABER and KECCAK. For higher-order masking, Bos et al. [BGR+21] introduced a generic masking scheme that is provably secure at any orders under the SNI model. In 2022, Bronchain and Cassiers [BC22] ML-KEM implementation achieves PINI security level.

In 2024, Demange et al. [DR24a], introduced the first masked implementation of a code-based algorithm. This implementation of BIKE is proven NI secure, for any masking order. The authors used only boolean masking (avoiding arithmetic masking common in lattice-based schemes). The advantages of this choice are twofold: *(i)* it does not require mask conversions, which are known to be computationally expensive, and *(ii)* BIKE is mainly based on binary operations, which makes boolean masking more natural and efficient. They developed many new gadgets specific to BIKE, including the masked version of Sendrier's countermeasure.

2.7 Problem statement

At the start of this research, our objective was to analyse side-channel vulnerabilities in HQC and propose tailored countermeasures. Early on, a major challenge became apparent: many SCA on HQC had already been published, and all components of the scheme seemed thoroughly vetted. While countermeasures against timing attacks were regularly integrated to the HQC reference implementation, there was a lack of a unified implementation that was resistant to timing, power and EM (Electro-Magnetic) SCA. This gap became more evident with the publication of BIKE masked implementation [DR24a], which motivated us to pursue a similar effort for HQC. But before working on the implementation itself, it was essential to identify which variables and functions in HQC are sensitive and need protection. This step aimed to avoid the issues encountered by

Migliore *et al.* [MGTF19]. To maximize the chances of success, we began a joint work with part of the HQC team to design a fully masked implementation of HQC, based on their reference implementation. Our work is supported by the first detailed sensitivity analysis of HQC and relies on state-of-the-art masking techniques to achieve the first fully masked implementation of HQC at any order. Additionally the formal verification is backed by practical side-channel analysis and benchmarks, showing that our masked implementation is competitive in the state-of-the-art masked PQC implementations.

Part II

Contributions

Chapter 3

Evaluating HQC

Contents

3.1 Improving Guo *et al.* attack

3.1.1 Attack Description

We begin our side-channel analysis of HQC by reproducing an attack from the state of the art. Our goal is to find an attack that we could then build upon, either by targeting another part of the algorithm or reduce its complexity in terms of assumptions and setup or computational power. The dual aim is also to familiarize ourselves with HQC itself. After a survey of the existing attacks on the October 2022 implementation of HQC, we choose Guo *et al.* Timing attack [GHJ+22]. In this article, the authors target the re-encryption step in the decapsulation. They recover the secret key by exploiting a timing variation in the vector sampling step of the re-encryption. This leakage can be exploited by mounting an attack that recovers the secret key of HQC-128 in around 866,000 decapsulations.

The timing variation is caused by collisions during the vector sampling. To sample a vector $\boldsymbol{v}$ of weight ω the algorithm draws ω distinct values between 0 and $n-1$ that will constitute the support of $\boldsymbol{v}$ ($n = 17669$ for HQC-128). Theses random values are obtained through a call to the XOF `seedexpander` at the start of the sampling. If two drawn values are equal there is a collision and the algorithm must ask `seedexpander` for more randomness because it needs ω distinct values (see Algorithm 6). This additionnal call to `seedexpander` is noticeable in timing and and may occur during each of the three vector samplings required for encryption. The seed of the XOF comes form a hash of the

plaintext. As such, we can categorize plaintexts according to the number of additional `seedexpander` calls (zero, one, two or three) that occur during their encryption. Figure 3.1 provides an overview of the strategy used by Guo *et al.* to extract the secret key from a target device running an HQC implementation. We abbreviate "*additional seedexpander call(s)*" as "*asec*".

- In the first place, the attacker selects a message $\boldsymbol{m}$ that yields 3 additional calls to `seedexpander`. Since they are rare ($\approx 0.6\%$) and their encryption is noticeable in timing, the attacker will use this properties as a distinguisher when asking the target for a decapsulation.
- For the second step, they create a ciphertext such that $\boldsymbol{c} = (\boldsymbol{u}, \boldsymbol{v}) = (1, \boldsymbol{mG})$. It is not a valid ciphertext as $\boldsymbol{u}$ is set to 1 and $\boldsymbol{m}$ is simply encoded, not encrypted. This choice was made so that the target, when decrypting this ciphertext computes:

$$\mathcal{C}.\text{Decode}(\boldsymbol{v} - \boldsymbol{uy}) = \mathcal{C}.\text{Decode}(\boldsymbol{mG} - \boldsymbol{y}) \tag{3.1}$$

  In other terms, the target has to decode a codeword that is a valid encoding of the message $\boldsymbol{m}$ with added errors that corresponds to the secret key $\boldsymbol{y}$.
- The third step is to iteratively add random errors (bit by bit) to the ciphertext and query the target for a decapsulation. When the timing differs from that of the original message, it indicates that the XOF has have been seeded with a different value. This can only happen if the decoded message was different from the original one, revealing the attacker that the correction capacity of the decoder has been exceeded. From this point, the attacker stops adding error bits and saves the faulted ciphertext.
- The fourth step will consist in going through each bit in the faulted ciphertext and flipping it (one at a time), before querying the target. Two outcomes are possible: if the timing remains different from the one of the original message, then they just added another error to the ciphertext. However, if the timing is the same, then an error bit has been corrected. Again, there are two possibilities: if this error bit was *not* added during the third step, then it originates from $\boldsymbol{y}$. It is a bit that is set in $\boldsymbol{y}$.

Repeating this method until each bit of $\boldsymbol{y}$ has been decided, the attacker recovers the value of secret key.

3.1.2 Improving the attack

We searched for a method to reduce the number of decapsulations needed to recover the secret key. We found the answer by drawing inspiration from Guo *et al.* optimized strategy. To reach a total of 866,000 decapsulations, their idea is to exploit the block structure of the ciphertext to speedup the third step: the search of the point where the faulted ciphertext exceeds by one bit the error correction capacity of the decoder. The ciphertext can be seen as 46 blocks of 384 bits each, among which HQC can correct up to 15 faulted blocks. Their strategy was to fault 15 random blocks at once, and begin the third step by searching for the tipping point in a 16th block. However, it was clear to us that this bit by bit search in the 16th block was suboptimal. Since each block is 384 bits long, it is highly unlikely that faulting just a few bits (e.g., 1, 2, or even 10) would lead to a decoding error. We needed to take a closer look at the DFR of duplicated RM codes. A

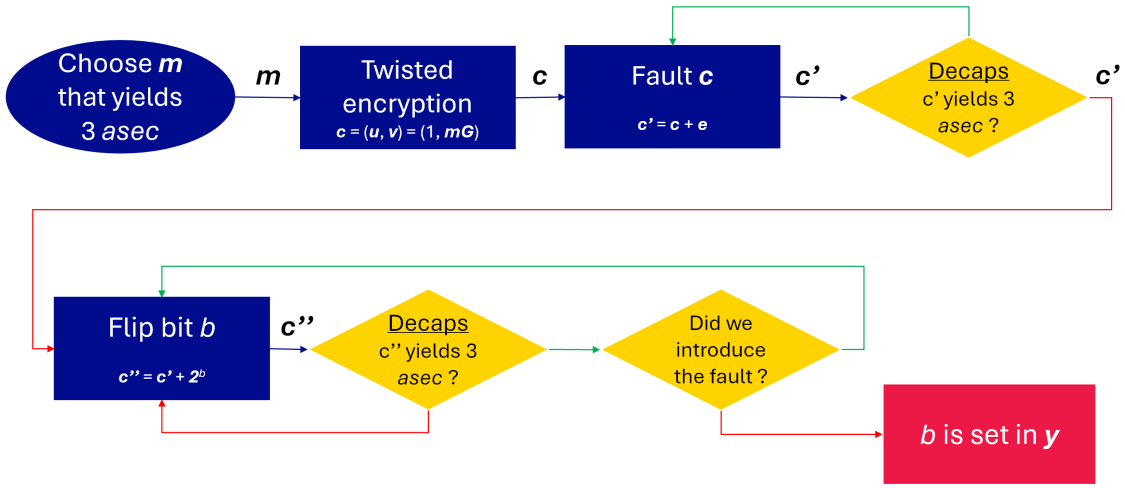


Figure 3.1: Outline of Guo *et al.* attack in [GHJ+22]

pessimistic (but simple) upper bound to compute the probability of a decryption failure is given by the authors of HQC in their reference paper [AMAB+17], Proposition 2.5.1. :

$$DFR \leq 255 \sum_{j=d/2}^{d} \binom{d}{j} p^j (1-p)^{d-j}$$

with p the error rate and d the minimal distance.

In our case, $d = 192$ and we needed to determine a suitable error ratio p. We initially chose $p = \frac{137}{384}$, which results in a decryption failure probability of less than 0.85% when randomly faulting 137 out of 384 bits in a block.

3.1.3 Practical experiments

We performed a test to verify that our implementation produced the same distribution as in the original paper, which it did. Figure 3.2 shows the distribution of plaintexts across these four categories. To obtain it, we used the PQM4 implementation of HQC [KPR+], optimized for ARM Cortex-M4, and ran it on our STM32F4 Discovery board equipped with that processor. We connected our board to a laptop through the serial port. The board acts as a target device that continuously listens to the serial port and executes the requests (encryption, decapsulation, ...) it receives from the computer. An internal performance counter allows us to measure precisely the elapsed number of cycles taken to execute an operation. Once completed, this number is sent to our laptop by the board. We sent 100,000 plaintexts for the board to encrypt and recorded the number of cycles taken for the sampling. Note that the more additional calls are made, the rarer the plaintexts that provoke them become.

We also recorded EM traces to compare their aspect according to their timing class (see Figure 3.3). The four timing classes can effectively be distinguished, for example by concentrating our focus on the pattern pointed by the red arrow. It represents the end of the vector sampling and we can notice it shifts to the right until it disappears. This indicates that the sampling effectively takes longer as more collisions occur. We can observe some desynchronization (small yellow arrows) in the second and third group of traces.

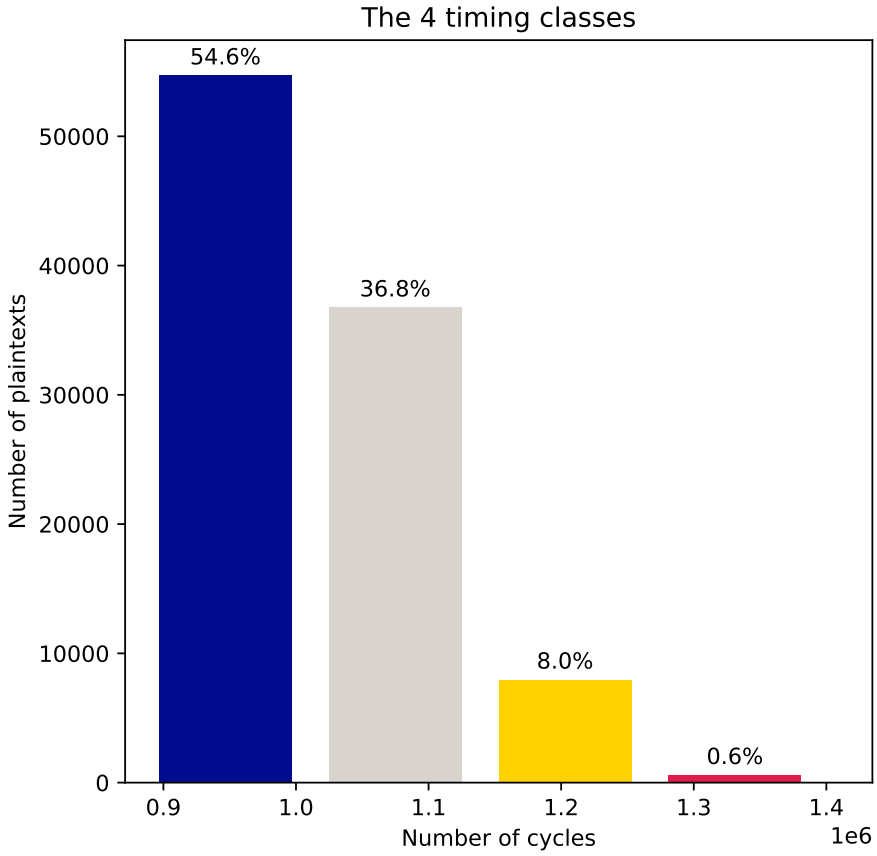


Figure 3.2: Distribution of the number of cycles needed for the sampling of 3 small-weight random vectors.

These correspond to the traces were there was respectively one and two additional calls to `seedexpander`. Although we know the number of additional calls, we ignore in which vector sampling they happened, thus the apparent desynchronization. These EM traces serve only for visualisation, to understand the internal behaviour of the implementation.

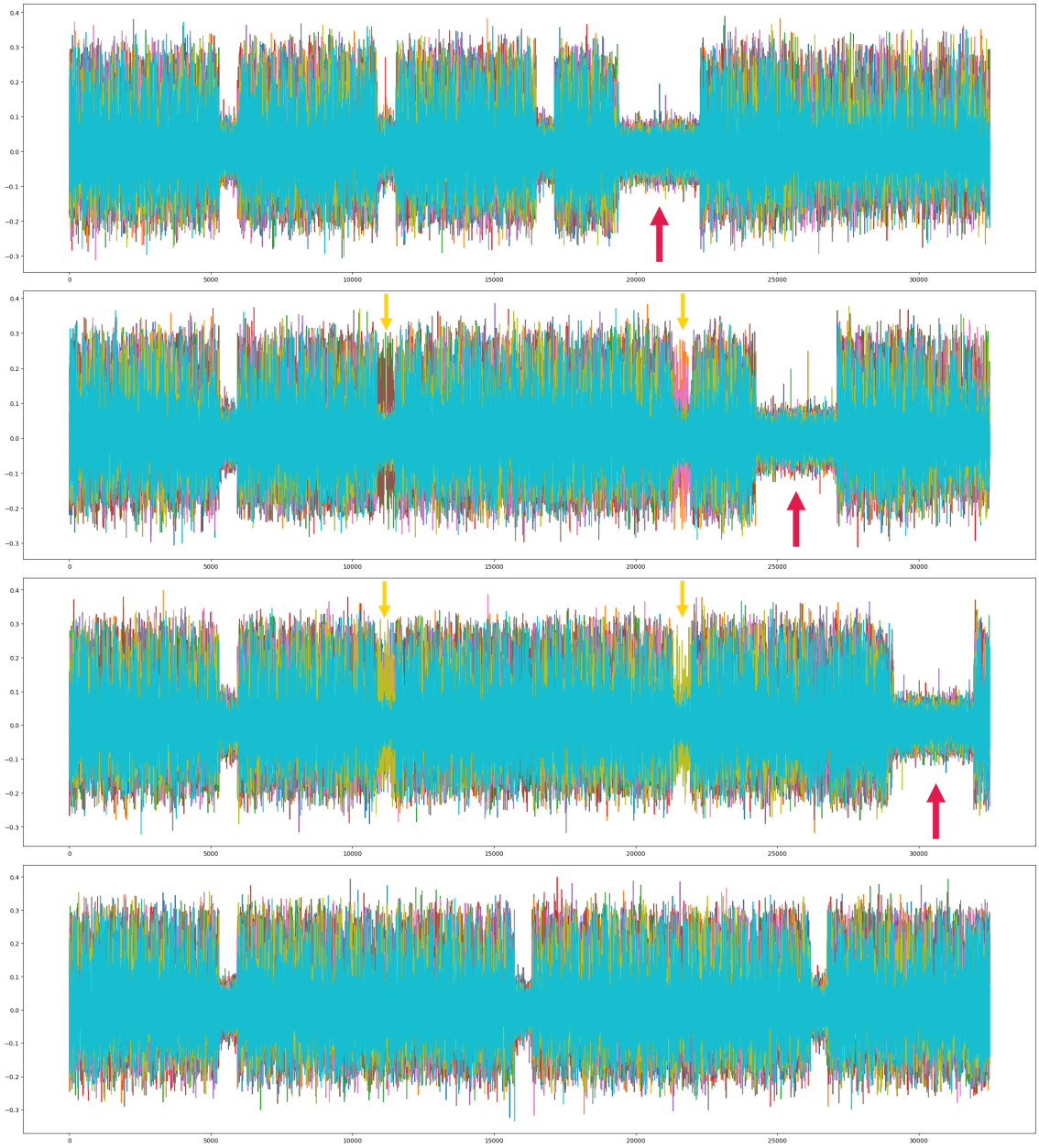


Figure 3.3: EM traces of the four timing classes.

Next, we reproduced the timing attack with our STM32F4 Discovery board as the target, using the internal performance counter to deduce the timing class of the decoded message. An initial test showed that a single decapsulation takes approximately 0.22 seconds to run on this platform. Performing the full attack requires around 866,000 decapsulations, which would take more than two days. To shorten this duration, there was no other viable option than to reduce the number of decapsulations. The microprocessor on the STM32F4 is mono-thread, meaning we can not parallelize the attack by doing multiple

decapsulations at the same time. Additionally, the code was already compiled using the -O3 optimization flag, we can not further reduce the time taken for one decapsulation.

We tested our new faulting approach with an error rate $p = \frac{137}{384}$, and found out experimentally that the optimal value was closer to $p = \frac{140}{384}$. Subsequently, we added a condition in the third step to start querying the oracle only if we had already faulted 140 bits in the 16th block. Thanks to this observation, we managed to divide by three the number of requests needed to recover the secret key. As a result, the attack only requires 290,000 requests to execute, $\frac{1}{3}$ of the original amount, and the key was practically recovered in around 17 hours.

3.2 Analysis of Sendrier's countermeasure

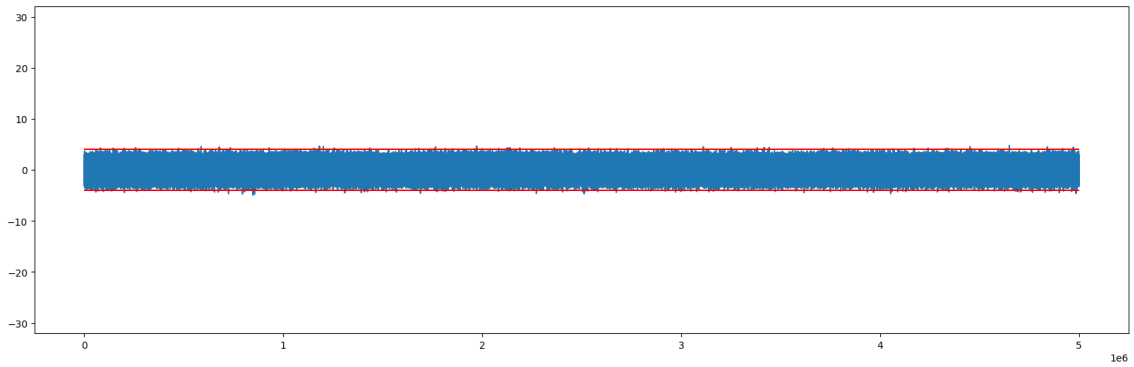
3.2.1 Finding a new attack vector

Our contribution here is to analyse the sensitivity of the countermeasures beyond timing, by exploring power and EM leakage. A solution to Guo *et al.*'s attack was already known, an algorithm to sample a random vector in constant time (see Algorithm 7). Proposed by Nicolas Sendrier, it was initially designed to correct a similar issue in BIKE. As suggested by the authors of [GHJ+22], the countermeasure has been integrated in the reference implementation of HQC. Although it should no longer be possible to distinguish between two messages based on their encryption time, other attack vectors can be considered. For example, the power consumption observed during the sampling of two identical vectors should be roughly the same, as the same intermediate values are processed. Conversely, sampling different vectors is likely to result in distinguishable power traces due to the manipulation of different data. This observation suggests that we could build a new distinguisher, based of power consumption traces. The idea would be to pick a message $\boldsymbol{m}$ and record traces of its encryption. If we are able to identify whether a given encryption trace corresponds to $\boldsymbol{m}$ or to a different message, we could adapt Guo *et al.*'s attack to a constant-time setup. The t-test should be sufficient to perform this binary classification.

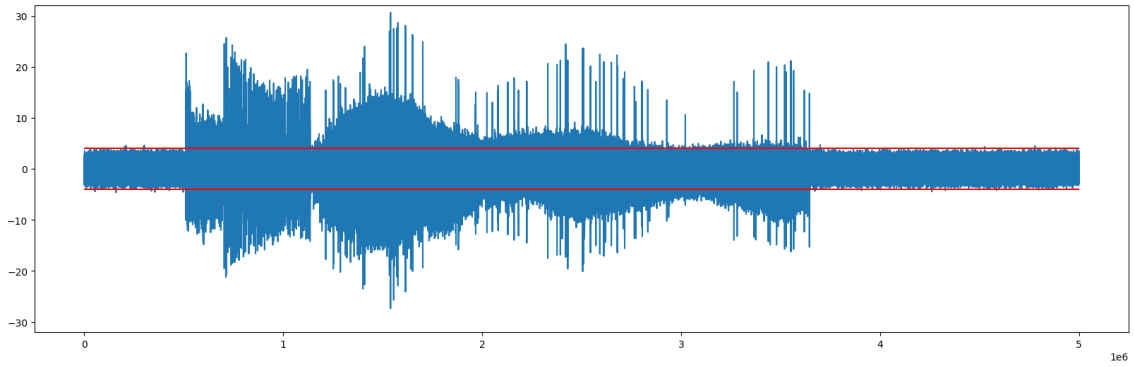
3.2.2 Practical Experiments

We replaced the old vector sampling on our target with the code of the new constant-time vector sampling. As expected, the variation in timing disappeared. We recorded EM traces of the vector sampling and performed a t-test between two set of 1,000 traces with two different input messages (see Figure 3.4b). We also ran a t-test on two set of 1,000 traces with the same input message (see Figure 3.4a). Crucially, there were no false positives, meaning we were effectively able to determine whether the same vector was being sampled or not. Experimentally, we found that this distinguisher was effective with sets as small as 10 traces.

Building on Guo *et al.*'s attack strategy we replaced the timing distinguisher with our new EM-based one. Unlike the original attack, which only worked for 0.6% of messages (those that triggered 3 additional calls to `seedexpander`), our attack works with any initial message. The main drawback is that our distinguisher requires 10 traces per oracle call, so the total number of decapsulations is multiplied by 10. Using our improved strategy of Section 3.1.2, this new attack can recover the secret key of HQC-128 in less than 3 million oracle calls, which would take about a week on our experimental setup. More efficient attacks on HQC already exist and can recover the secret key using significantly fewer traces. Our goal here was to highlight the importance of protecting HQC against side-channel attacks.



(a) with the same inputs.



(b) with two different inputs.

Figure 3.4: t-test of Sendrier's vector sampling

3.3 HQC Side-Channel Sensitivity Analysis

This section is adapted from a text written by Guillaume Goy, as part of our co-authored article *Side-Channel Sensitivity Analysis on HQC: Towards a Fully Masked Implementation*, with his kind permission.

In this section, we provide a strong side-channel sensitivity analysis of variables of HQC. We first represent variables and functions of HQC-KEM in an oriented graph (see Figure 3.5). A variable node is connected to a function node with a descending arrow if it is an input or with an ascending arrow if it is an output. The purpose of this section is to decide if a variable leaks information about a secret and must thus be masked. Variables holding a secret key (sk) or part of the shared key are considered as sensitive while variables holding part of the public key (pk) or ciphertext (ct) are considered as non-sensitive. For any other variable v, we show that the knowledge of v (denoted $\mathscr{K}(v)$) leads to the recovery of another sensitive data or secret. We summarize our analyzes in Figure 3.5 where sensitive variables and functions are red and non-sensitive variables and functions are green.

3.3.1 Variables sensitivity

The secret key of HQC is the pair of small Hamming weight vectors $(\boldsymbol{x}, \boldsymbol{y})$. The shared key K is derived from the message $\boldsymbol{m}$, denoted $\boldsymbol{m}'$ in the Decapsulation process. The public information in HQC is the public key $(\boldsymbol{h}, \boldsymbol{s})$, the public known code represented by generator matrix $\boldsymbol{G}$ and parity check matrix $\boldsymbol{H}$, and ciphertext vectors $\boldsymbol{c} = (\boldsymbol{u}, \boldsymbol{v})$ and $\boldsymbol{d}$.

KeyGen The following variables must be secured:

(i) $(\boldsymbol{x}, \boldsymbol{y})$ since it is the secret key and

(ii) $\boldsymbol{t}_0$ since $\mathscr{K}(\boldsymbol{t}_0) \implies \mathscr{K}(\boldsymbol{x})$ by: $\boldsymbol{x} = \boldsymbol{s} - \boldsymbol{t}_0$.

Encaps The Encaps function involves the generation of ephemeral random values $\boldsymbol{r}_1, \boldsymbol{r}_2$ and $\boldsymbol{e}$. From [AMAB+17], we know that the security reduction relies on the hardness of solving an instance of QC-SDP. The knowledge of any of the three vectors $\boldsymbol{r}_1$,$\boldsymbol{r}_2$ or $\boldsymbol{e}$ dramatically decreases the complexity of this operation which can be solved by inverting the residual matrix. We thus have that:

$$\mathscr{K}(\boldsymbol{r}_1) \iff \mathscr{K}(\boldsymbol{r}_2) \iff \mathscr{K}(\boldsymbol{e}) \tag{3.2}$$

any of them would break the IND-CPA security of HQC, they must be secured. The following other variables must also be secured:

(i) K since it is the shared key.

(ii) $\boldsymbol{m}$ since $\mathcal{K}$ is a publicly known hash function, so $\mathscr{K}(\boldsymbol{m}) \implies \mathscr{K}(K)$.

(iii) θ since it allows to generate all encryption random values $\boldsymbol{e}, \boldsymbol{r}_1, \boldsymbol{r}_2$.

(iv) $\boldsymbol{t}_1$ since $\mathscr{K}(\boldsymbol{t}_1) \implies \mathscr{K}(\boldsymbol{r}_1)$ by: $\boldsymbol{r}_1 := \boldsymbol{u} - \boldsymbol{t}_1$.

(v) $\boldsymbol{t}_2$ since $\mathscr{K}(\boldsymbol{t}_2) \implies \mathscr{K}(\boldsymbol{r}_2)$ by: $\boldsymbol{r}_2 = \boldsymbol{s}^{-1}\boldsymbol{t}_2$.

(vi) $\boldsymbol{t}_3$ since $\mathscr{K}(\boldsymbol{t}_3) \implies \mathscr{K}(\boldsymbol{m})$ as $\boldsymbol{t}_3 = \boldsymbol{m}\boldsymbol{G}$ can be reversed.

(vii) $\boldsymbol{t}_4$ since $\mathscr{K}(\boldsymbol{t}_4) \implies \mathscr{K}(\boldsymbol{e})$, by: $\boldsymbol{e} = \boldsymbol{v} - \boldsymbol{t}_4$.

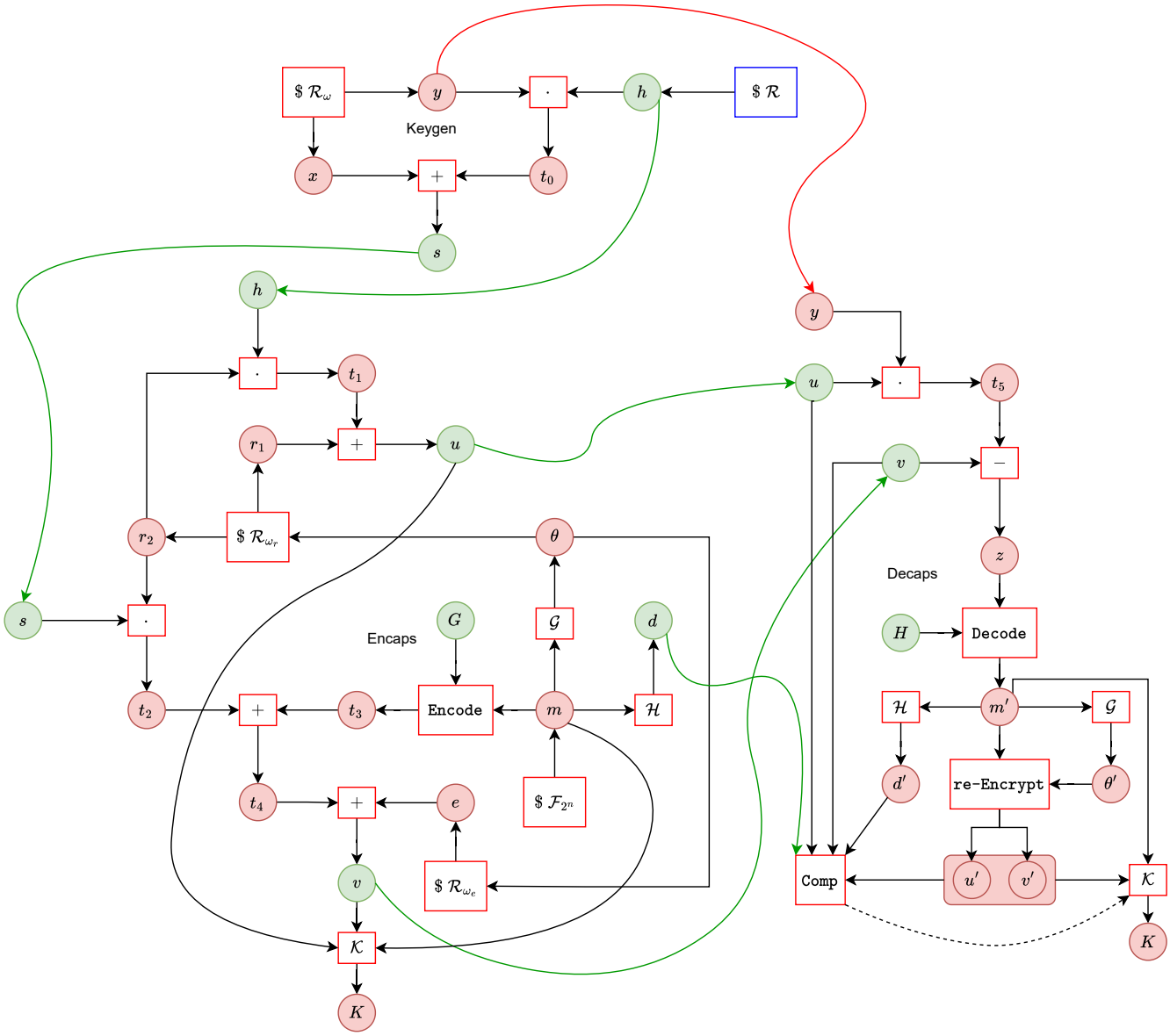


Figure 3.5: Variables and functions of HQC KEM, red: sensitive, blue: non-sensitive, green: public data

Decaps The following variables must be secured:

(i) $K, \boldsymbol{y}$ as they are respectively the shared key and part of the secret key.

(ii) $\boldsymbol{m'}$ since $\mathcal{K}$ is a publicly known hash function, so $\mathscr{K}(\boldsymbol{m'}) \implies \mathscr{K}(K)$.

(iii) $\boldsymbol{z}$ since the decoder is publicly known and decoding $\boldsymbol{z}$ allows $\mathscr{K}(\boldsymbol{m'})$.

(iv) $\boldsymbol{t}_5$ since $\mathscr{K}(\boldsymbol{t}_5) \implies \mathscr{K}(\boldsymbol{z})$, by: $\boldsymbol{z} = \boldsymbol{v} - \boldsymbol{t}_5$.

(v) θ' since it allows to generate all re-encryption random values $\boldsymbol{e}, \boldsymbol{r}_1, \boldsymbol{r}_2$.

(vi) $\boldsymbol{u}'$ and $\boldsymbol{v}'$ since the result of `Comp` could be used to build a CCA and bypass the re-encryption security.

Re-encryption We have shown that all variables manipulated during the encryption step are sensitive. In a typical use case of HQC, the re-encryption step processes the exact same variables as those used during encryption. It follows that the variables and functions involved in re-encryption must also be protected using the same security measures as those applied during encryption.

3.3.2 Functions sensitivity

Any function that manipulates at least one sensitive variable is considered a sensitive function. Each sensitive function must be transformed into a secure gadget to ensure that the manipulation of sensitive variables is carried out in a protected manner. These functions are:

(i) $+$ addition in characteristic 2.

(ii) $\cdot$ binary vector multiplication.

(iii) $\times$ binary matrix vector multiplication.

(iv) $\$\mathcal{R}_\omega$ random sample from a given set (with Hamming weight constraint).

(v) $\mathcal{G}, \mathcal{K}$ and $\mathcal{H}$ hash functions.

(vi) `Comp` equality verification (inputs are secret and output must be secured to prevent CCA).

(vii) HQC `Decoder` is a complex operation, the next section is dedicated to its sensitivity analysis.

3.3.3 Sensitivity of the HQC decoder

The decoder of HQC is a concatenation of RM and RS decoders. It is a complex function composed of numerous operations.

Many side-channel attacks [GLG22a, PRJB, BMG+24] exploit some leakage of the RM decoder at various steps of the processing. It follows that all operations of this decoder must be protected to thwart these attacks.

On the other hand, [GLG22b] and [GMGL23] showed the RS syndrome computation leaks information that lead to recover the shared key of HQC. The authors also showed that the encoding process of HQC can leak information about the shared key and thus also needs to be secured. Furthermore, the attack in [SHR+22] demonstrates that the whole RS decoder can be exploited for key recovery. Hence, each operation of the HQC encoder and decoder needs to be masked to ensure the global security of the scheme.

Wrap-up The attack presented in [SHR+22], highlights the necessity of protecting the entire RS decoder against side-channel attacks. Moreover, numerous state-of-the-art attacks [GLG22a] [PRJB] [BMG+24] targeting the RM decoding step demonstrate that this component must also be secured. In addition, we have established that all variables and functions involved in HQC (excluding public key generation and manipulation) are sensitive (see Figure 3.5), in the sense that knowledge of any one of them could lead to the disclosure of a cryptographic secret. It follows that all such elements must be properly protected to ensure the overall security of the implementation.

Chapter 4

Protecting HQC

Contents

Though HQC was selected at the end of round 4 of NIST competition, there was no implementation protected against SCA. Our goal is thus to propose the first end-to-end masked implementation of HQC. In collaboration with part of the HQC team, we develop a publicly available masked implementation of HQC, and prove that it is secure both on paper and in practice. We also develop new generic gadgets to best suit our needs. Using all these basic blocks, we design and code masked versions of the HQC functions. We apply the MIMO-SNI [CS20] framework to ensure the correct composition of our gadgets and to benefit from its efficient security proof method. Finally, we analyse the side-channel leakage of selected functions to demonstrate the efficiency of our solution.

4.1 Masking strategy

Following the sensitivity analysis presented in Section 3.3, we gained a clear understanding of which parts of HQC required protection. This allowed us to establish a precise strategy to secure sensitive variables and functions, ensuring the resilience of our implementation against SCA.

We started the masking of HQC by identifying the gadgets, that we would need for its protection. Most of these were state-of-the-art gadget that were proved NI or SNI-secure (see Section 4.2). In addition, we developed new gadgets tailored to specific operations in HQC, where existing solutions were lacking (see Section 4.3).

Using this collection of NI and SNI gadgets, we set out to build MIMO-SNI composite gadgets. As all gadgets in our collection have only one output, the first prerequisite of the MIMO-SNI model is already satisfied. The remain four composition conditions, as established in [CS20], that can be informally summarized as follows:

1. No path from an input to an output.
2. At most one path between two gadgets.
3. No path from a gadget to two inputs.
4. No path from a gadget to two outputs.

These conditions must be verified by the "simplified" DAG, once all refresh vertices and their incident edges have been removed. To ensure compliance, our implementation was designed to enforce these four conditions. Condition 1 and 3 were systematically satisfied by always refreshing all the inputs within a composite MIMO-SNI gadget. Condition 2 may only be violated in cases where a variable is reused and its copies later converge as n inputs to the same gadget. To prevent this, we refresh (n–1) of the inputs originating from the same variable. Most of our composite gadgets produce a single output, making Condition 4 trivial to satisfy. In the rare cases where a gadget produces n outputs ($n > 1$), we ensure that $(n - 1)$ of them are refreshed.

Once we had these composite MIMO-SNI gadgets, we assembled them to build the core functions of HQC (KeyGen, Encaps, Decaps, Encrypt, Decrypt). As MIMO-SNI implies PINI, we leverage the composition property of PINI gadgets, to obtain a fully-masked PINI implementation of HQC.

In parallel, we also dedicated significant effort on researching and designing efficient gadgets (see Section 4.4.3 and 4.5). Finally, we assessed our implementation through a series of benchmarks and practical side-channel evaluations to validate both its efficiency and its security (see Chapter 5).

4.2 State-of-the-art gadgets

To complete our fully masked HQC implementation, we use some generic gadgets (see Table 4.1) from the state-of-the-art. Some are used *as-in*, we re-implement from scratch the conversions from arithmetic masking to boolean masking (`AtoB`) and from boolean masking to arithmetic masking (`BtoA`), following the NI composition rules.

Note: the $\texttt{sec}_{\neg}$ gadget is the simplest of all in boolean masking; it simply consist in flipping one of the input shares: $(a_1, a_2, \ldots, a_d) \rightarrow (\neg a_1, a_2, \ldots, a_d)$. It is trivially NI.

Gadget	References	Security	Function
$\texttt{sec}_{\neg}$		NI	Bitwise boolean NOT
$\texttt{sec}_{\&}$	[CGTV15]	SNI	Bitwise boolean AND
$\texttt{sec}_{+}$	[CGTV15]	NI	Addition in $\mathbb{Z}$
`refresh`	[BCPZ16]	SNI	Refresh mask
`AtoB`	[CGV14]	NI	Arithmetic to boolean conversion
`BtoA`	[CGV14]	NI	Boolean to arithmetic conversion
$\texttt{sec}_{=}$	[DR24a]	NI	Masked variables equality
$\texttt{sec}_{\text{if}}$	[DR24a]	NI	Conditional branch
$\texttt{sec}_{\text{SHA-3}}$	[CGL+24, Ger24]	NI	Secure `SHA-3`

Table 4.1: Secure gadgets used in our implementation

4.3 New generic gadgets

We develop new generic gadgets using the ones from the state of the art as building blocks. All binary secure operations for which it makes sense come in two flavours: $\texttt{sec}_{\text{op}}(\llbracket A \rrbracket, \llbracket B \rrbracket)$ (both operands masked), $\texttt{sec}_{\text{op}}(\llbracket A \rrbracket, B)$ (left operand masked, right operand public and unmasked).

Since trivial implementations of linear operations are NI, we do not represent them as gadgets, to lighten the notation. We write: $\llbracket a \rrbracket \oplus \llbracket b \rrbracket$ to indicate a sharewise XOR.

Under boolean masking, shift and AND (when the second operand is not masked) are the same operation but sharewise. For example, we write $\llbracket a \rrbracket \gg b$ ($\llbracket a \rrbracket \ll b$) to indicate a right (respectively left) shift by b positions, and $\llbracket a \rrbracket \& b$ for a sharewise AND between $\llbracket a \rrbracket$ and b.

We represent a gadget with its algorithm, its computation graph and, when it is different, its simplified computation graph. Then we prove that it is MIMO-SNI by verifying that it satisfies the conditions listen in Section 2.6.2.

4.3.1 Boolean masked opposite

This gadget returns the two's complement opposite of an integer masked in boolean masking. We use the two's complement property: $-x = \neg x + 1$ (where $\neg x$ is x with all bits flipped).

Algorithm 9 $\texttt{sec}_{\text{oppos}}$

Input: $\llbracket a \rrbracket$
Output: $\llbracket z \rrbracket \mid z = -a$
1: $\llbracket \neg a \rrbracket \leftarrow \texttt{sec}_{\neg} \llbracket a \rrbracket$
2: $\llbracket z \rrbracket \leftarrow \texttt{sec}_{+}(\llbracket \neg a \rrbracket, 1)$
3: **return** $\llbracket z \rrbracket$

Theorem 1. $\texttt{sec}_{\text{oppos}}$ *is NI.*

Proof. $\texttt{sec}_{+}$ and $\texttt{sec}_{\neg}$ are NI and no masked variable is used twice. □

4.3.2 Boolean masked subtraction

This gadget computes the subtraction of two integers masked in boolean masking.

Algorithm 10 $\texttt{sec}_-$

Input: $[\![a]\!], [\![b]\!]$
Output: $[\![z]\!] \mid z = a - b$
1: $[\![t]\!] \leftarrow \texttt{sec}_{\text{oppos}}([\![b]\!])$
2: $[\![z]\!] \leftarrow \texttt{sec}_{+}([\![a]\!], [\![t]\!])$
3: **return** $[\![z]\!]$

Theorem 2. *$\texttt{sec}_-$ is NI.*

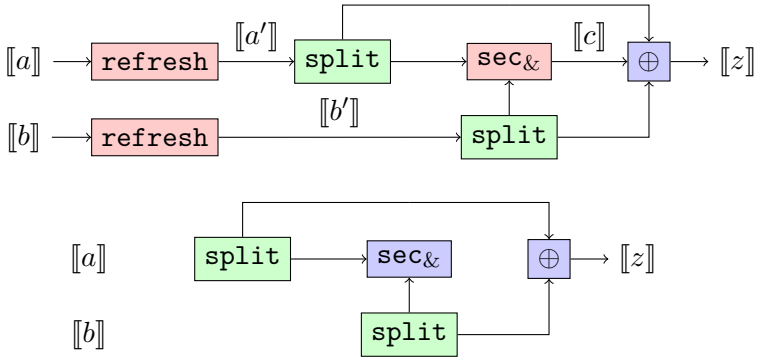
Proof. $\texttt{sec}_{\text{oppos}}$ and $\texttt{sec}_{+}$ are NI and no masked variable is used twice. □

4.3.3 Building a composite MIMO-SNI gadget: Boolean masked OR

In this section, we will give an insight on how we built our MIMO-SNI composite gadgets. To this end, we will take the Boolean masked OR as an example. It computes the logical OR between two masked values. We use the relation: $a \vee b = a \oplus b \oplus (a \wedge b)$.

Input: $[\![a]\!], [\![b]\!]$
Output: $[\![z]\!] \mid z = a \vee b$
1: $[\![a']\!] \leftarrow \texttt{refresh}([\![a]\!])$
2: $[\![b']\!] \leftarrow \texttt{refresh}([\![b]\!])$
3: $[\![c]\!] \leftarrow \texttt{sec}_{\&}([\![a']\!], [\![b']\!])$
4: $[\![z]\!] \leftarrow [\![a']\!] \oplus [\![b']\!] \oplus [\![c]\!]$
5: **return** $[\![z]\!]$

(a) Algorithm



(b) Computation graph (top: full, bottom: simplified)

Figure 4.1: $\texttt{sec}_{\text{or}}$ gadget

Theorem 3. *$\texttt{sec}_{\text{or}}$ is MIMO-SNI.*

Proof. After erasure of the `refresh` gadgets on $[\![a]\!]$ and $[\![b]\!]$ and their incident edges, there is no path anymore from an input to an output, and no path from two different inputs to the same vertex. The gadget has only one output $[\![z]\!]$, so there cannot be a path from a vertex to two different outputs. After erasure of the implicit output `refresh` gadget of $\texttt{sec}_{\&}$ and its incident edges, there are no paths any more between the output of $\texttt{sec}_{\&}$ and another vertex. This leaves at most one single path between any two vertexes. □

Proposition of a simplified $\texttt{sec}_{\text{or}}$ We developed a simplified version of the above $\texttt{sec}_{\text{or}}$ gadget. We use the relation: $a \vee b = \neg(\neg a \wedge \neg b)$.

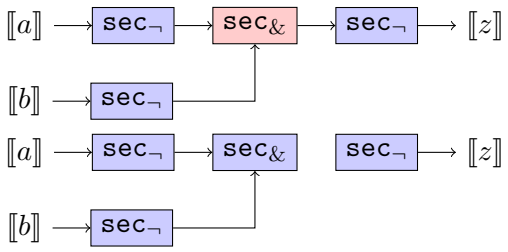
Theorem 4. *The simplified $\texttt{sec}_{\text{or}}$ gadget is SNI.*

Proof. The $\texttt{sec}_{\&}$ gadget is SNI, which is equivalent to an NI $\texttt{sec}_{\&}$ followed by a refresh. All gadgets are NI and all masked variables are used at most once as input of a gadget. By construction the simplified $\texttt{sec}_{\text{or}}$ gadget is NI and since its output shares are independent from the input shares, its is SNI. □

For performance concerns, this is the version we will use in the following algorithms when referring "$\texttt{sec}_{\text{or}}$".

Input: $[\![a]\!], [\![b]\!]$
Output: $[\![z]\!] \mid z = a \vee b$
1: $[\![\neg a]\!] \leftarrow \texttt{sec}_{\neg}([\![a]\!])$
2: $[\![\neg b]\!] \leftarrow \texttt{sec}_{\neg}([\![b]\!])$
3: $[\![c]\!] \leftarrow \texttt{sec}_{\&}([\![\neg a]\!], [\![\neg b]\!])$
4: $[\![z]\!] \leftarrow \texttt{sec}_{\neg}([\![c]\!])$
5: **return** $[\![z]\!]$

(a) Algorithm



(b) Computation graph (top: full, bottom: simplified)

Figure 4.2: Simplified $\texttt{sec}_{\text{or}}$ gadget

4.3.4 Refresh

For the SNI `refresh` gadget we decided to use the one based on a recursive algorithm proposed by Battistello *et al.* [BCPZ16]. It has a quasi-linear complexity, which offers a significant speedup compared to the quadratic complexity of the other state-of-the-art SNI refresh gadgets.

4.4 Masked Vector Sampling for HQC

We wanted to use the masked vector sampling from [DR24a] because it was proven secure and its implementation was publicly available. Though both HQC and BIKE vector sampling are based on the algorithm described in [Sen21] they differ in the way they draw randomness. Given a p bits random number a, BIKE multiplies it by n and shifts p bits to the right (Algorithm 11, Line 4), whereas HQC returns the remainder of a modulo n (Algorithm 12, Line 3).

Algorithm 11 BIKE vector sampling

Input: $seed, len, wt$
Output: $wlist$, a list of wt distinct elements of $\{0, \ldots, len - 1\}$.
1: $wlist \leftarrow ()$ ▷ empty list
2: $s_0, \ldots, s_{wt-1} \leftarrow \texttt{SHAKE256-Stream}(seed, 32 \cdot wt)$
▷ parse as a sequence of wt non negative 32-bits integers
3: **for** $i = wt - 1$ **downto** 0 **do**
4: $\quad pos \leftarrow i + \lfloor (wt - i)s_i / 2^{32} \rfloor$
5: $\quad wlist \leftarrow wlist, (pos \in wlist)\ ?\ i : pos$
6: **end for**
7: **return** wlist

This difference means that the `SecFisherYates` algorithm from [DR24a] cannot be directly transposed to an HQC implementation, as it would not follow the specifications. We thus needed to design a side-channel resistant function that computes the remainder of a boolean masked value a modulo a public value n. Our design does not use division or modulo instructions because on some architectures their execution time depends on the numerator which, in our case, is a secret variable; a variation in execution time could leak information about the secret. Moreover, they are not directly applicable to boolean masking. We solve these problems with a masked Barrett reduction. Non-masked Barrett reduction was already implemented in September 2023 in the PQClean version of HQC [KSSW22] before being added to the HQC specification in February 2024. The

Algorithm 12 HQC vector sampling

Input: $n, w, seed$
Output: w distinct elements of $\{0, \ldots, n\}$
1: $prng \leftarrow \text{prng\_init}(seed)$
2: **for** $i = w - 1$ **downto** 0 **do**
3: $\quad l \leftarrow i + (\text{rand}(prng) \bmod (n - i))$
4: $\quad pos[i] \leftarrow (l \in \{pos[j], i < j < t)\ ?\ i : l$
5: **end for**
6: **return** $pos[0], \ldots, pos[w-1]$

idea of manually coding the Barrett reduction was a proposition from [SGG24]. They suggested it to fix a timing leakage in the randomness drawing caused by non-constant time division operations.

4.4.1 Barrett Reduction

The Barrett reduction [Bar87] efficiently computes the remainder of an integer division in constant-time. To compute $x \bmod n$ one can use $r = x - \lfloor x/n \rfloor \times n$. Instead of performing a division, Barrett reduction precomputes an integer m such that $\frac{m}{2^p} \approx \frac{1}{n}$, and uses it to approximate the quotient x/n when computing $x \bmod n$. We usually take $m = \lfloor \frac{2^p}{n} \rfloor$.

Remark 3. In our case, $p = 32$, because the function rand of HQC (Algorithm 12, Line 3) outputs pseudo-random 32-bits unsigned integers.

The main advantages are that variable m can be precomputed, and dividing by 2^p comes down to a shift p bits to the right, which is virtually free.

Algorithm 13 Barrett reduction

Input: $a, n, p, m \mid m = \lfloor \frac{2^p}{n} \rfloor$
Output: $r = a \bmod n$
1: $q \leftarrow (a \times m) \gg p$
2: $r \leftarrow a - q \times n$
3: **if** $r \geq n$ **then**
4: $\quad r \leftarrow r - n$
5: **end if**
6: **return** r

Since we use the floor function, the quotient $\frac{m}{2^p}$ is only guaranteed to be less or equal to $\frac{1}{n}$, a final subtraction is sometimes required. This cannot be permitted in a secure implementation as it would induce a timing inconsistency that could potentially be exploited. Our solution will compute both possible answers and return the correct one. It is constant-time by design, provably secure and works with any masking degree.

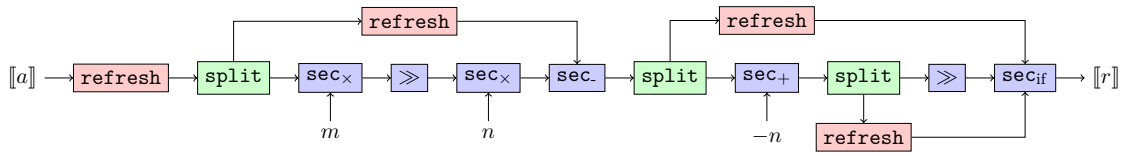
4.4.2 Boolean Barrett

To preserve the constant-time property, we always compute the conditional subtraction. We first subtract n from the computed value $a\_qn = a - q \times n$, and store the result in A (Line 8). There are 2 possibilities: either A is negative and a_qn is the correct result or A is positive and the correct result. To evaluate the sign of A we shift it by 31 bits to the

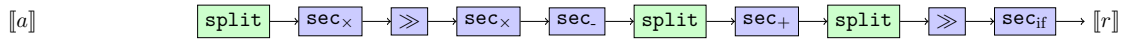
Algorithm 14 Masked Barrett reduction — "Boolean Barrett"

Input: $[\![a]\!], m, n$
Output: $[\![r]\!] = [\![a]\!] \bmod n$

1: $[\![a']\!] \leftarrow \texttt{refresh}([\![a]\!])$
2: $[\![q]\!] \leftarrow \texttt{sec}_{\times}([\![a']\!], m)$
3: $[\![q]\!] \leftarrow [\![q]\!] \gg 32$ ▷ Sharewise shift
4: $[\![qn]\!] \leftarrow \texttt{sec}_{\times}([\![q]\!], n)$
5: $[\![a'']\!] \leftarrow \texttt{refresh}([\![a']\!])$
6: $[\![a\_qn]\!] \leftarrow \texttt{sec}_{-}([\![a'']\!], [\![qn]\!])$ ▷ $(a - q \times n)$
7: $minus\_n \leftarrow 2^{32} - n$
8: $[\![A]\!] \leftarrow \texttt{sec}_{+}([\![a\_qn]\!], minus\_n)$ ▷ conditional final subtraction
9: $[\![z]\!] \leftarrow [\![A]\!] \gg 31$
10: $[\![A']\!] \leftarrow \texttt{refresh}([\![A]\!])$
11: $[\![a\_qn']\!] \leftarrow \texttt{refresh}([\![a\_qn]\!])$
12: $[\![r]\!] \leftarrow \texttt{sec}_{\text{if}}([\![a\_qn']\!], [\![A']\!], [\![z]\!])$ ▷ $[\![z]\!]$? $[\![a\_qn]\!]$: $[\![A]\!]$
13: **return** $[\![r]\!]$



(a) Full



(b) Simplified

Figure 4.3: Computation graph of Boolean Barrett

right and store the result in z; if A is negative then z is equal to 1, else A is positive and z is equal to 0.

Theorem 5. *"Boolean Barrett" is MIMO-SNI.*

Proof. m and n are public values, we only need to refresh $[\![a]\!]$ to ensure there is no path from an input to an output (Line 1). As it operates independently on each share, the shift operation (Lines 3, 9) is NI. All used gadgets are NI and produce only one output. Three variables are used twice, they are a (Lines 2, 6), A (Lines 9, 12) and a_qn (Lines 8, 12). They are all refreshed (Lines 5, 10, 11) to ensure there is at most one path between two gadgets. Finally, "Boolean Barrett" has only one output, there is no possible path from a gadget to two outputs. □

4.4.3 Accelerating the Multiplication

Boolean masking is one of several masking techniques. Arithmetic masking, for instance, splits a secret variable x into d shares such that:

$$x = x_1 + \cdots + x_d \mod q \tag{4.1}$$

In [DR24a] the authors decided to avoid conversions between boolean and arithmetic masking (`BtoA` and `AtoB`) because they are considered as expensive and because most

BIKE operations are binary. As HQC also uses binary vectors and operations, taking advantage of this structure to do efficient computations (e.g., the addition of two vectors is a bitwise exclusive OR (XOR) of their components), the choice of boolean masking makes sense.

We profiled our Boolean Barrett to identify potential performance improvements. It turned out that the $\texttt{sec}_\times$ gadget represented nearly 80% of the cycles of the function's total cycles. A possible way to reduce this number would be the use arithmetic masking: multiplying by a public value n in arithmetic masking simply consists in multiplying each share of $[\![x]\!]$ by n; the cost is in $\mathcal{O}(d)$. The performance of the multiplication could be improved, provided that the cost of the conversions does not surpass the cost of the boolean masked multiplication. We thus compared the cost of a multiplication in boolean masking and the cost of a sequence `BtoA`, arithmetic multiplication, `AtoB`:

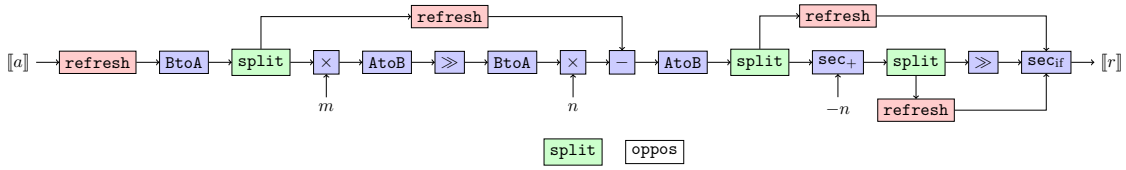
1. Boolean multiplication, no conversion: $\mathcal{O}(k \times \log k \times d^2)$
2. Arithmetic multiplication, with conversions: $\mathcal{O}(\log k \times d^2)$

with $k = 32$ bits and d the order of masking.

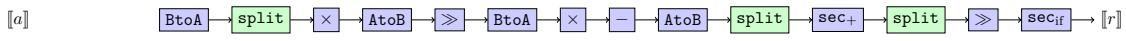
It appears that, theoretically, the multiplication could be significantly accelerated. We designed a new version of our masked Barrett reduction, replacing the boolean multiplications with arithmetic ones and the appropriate conversions (see Algorithm 15). For simplicity sake, we will refer to this version as *Arithmetic Barrett*, to contrast our first solution which relied exclusively on boolean masking. We based the code for the conversions on the pseudo-code described in [CGV14, Algorithm 4 & 6].

We have to convert $[\![q]\!]$ back to boolean on Line 4 before doing the bit shift. This is because bit shifting is a boolean operation and we don't know any shift gadget in arithmetic. We perform both multiplications and one subtraction under arithmetic masking.

The last subtraction (that is no longer conditional) is done in boolean for efficiency purposes. Indeed, if we had performed it in arithmetic, we would have had to pay for two `AtoB` conversions (one for $[\![A]\!]$ and one for $[\![a\_qn]\!]$). It would have been more expensive than a boolean masked addition and one `AtoB` conversion.



(a) Full



(b) Simplified

Figure 4.4: Computation graph of Arithmetic Barrett

Theorem 6. *Arithmetic Barrett is MIMO-SNI.*

Proof. Again, m and n are public values. We refresh $[\![a]\!]$ so it does not exist a path from an input to an output. Once we erase the `refresh` gadgets and their incident edges (Figure 4.4b), we can verify that there is at most one path between any pair of vertices. Finally, this gadget has only one output and no path to an input: there is trivially no path from a vertex to either two inputs or two outputs. □

Algorithm 15 Arithmetic Barrett

Input: $[\![a]\!], m, n$
Output: $[\![r]\!] = [\![a]\!] \bmod n$
1: $[\![a']\!] \leftarrow \texttt{refresh}([\![a]\!])$
2: $[\![ari\_a]\!] \leftarrow \texttt{BtoA}([\![a']\!])$
3: $[\![ari\_q]\!] \leftarrow [\![ari\_a]\!] \times m$ ▷ Multiplication on each share
4: $[\![q]\!] \leftarrow \texttt{AtoB}([\![ari\_q]\!])$
5: $[\![q]\!] \leftarrow [\![q]\!] \gg 32$ ▷ Shift on each share
6: $[\![ari\_q]\!] \leftarrow \texttt{BtoA}([\![q]\!])$
7: $[\![ari\_qn]\!] \leftarrow [\![ari\_q]\!] \times n$
8: $[\![ari\_a']\!] \leftarrow \texttt{arithmetic\_refresh}([\![ari\_a]\!])$
9: $[\![ari\_a\_qn]\!] \leftarrow [\![ari\_a']\!] - [\![ari\_qn]\!]$ ▷ Sharewise subtraction
10: $[\![a\_qn]\!] \leftarrow \texttt{AtoB}([\![ari\_a\_qn]\!])$
11: $minus\_n \leftarrow 2^{32} - n$
12: $[\![A]\!] \leftarrow \texttt{sec}_{+}([\![a\_qn]\!], minus\_n)$
13: $[\![z]\!] \leftarrow [\![A]\!] \gg 31$
14: $[\![a\_qn']\!] \leftarrow \texttt{refresh}([\![a\_qn]\!])$
15: $[\![A']\!] \leftarrow \texttt{refresh}([\![A]\!])$
16: $[\![r]\!] \leftarrow \texttt{sec}_{\text{if}}([\![a\_qn']\!], [\![A']\!], [\![z]\!])$
17: **return** $[\![r]\!]$

4.5 GF multiplication

In HQC, multiplication between polynomials of $\mathbb{GF}(2^{64})$ is done very efficiently thanks to an algorithm from [BGTZ08]. Our approach to protecting HQC against side-channel attacks with masking aims to ensure security while preserving performance. Not knowing how long it would take to protect such a lengthy function and having no guarantees that the resulting masked multiplication would remain efficient, we searched for a simpler, faster alternative: a code that would perform the same operation, with boolean gadgets in much fewer lines and less cycles. Fortunately, such a function already existed in the masked BIKE implementation [DR24b]. We adapted it to HQC: in BIKE, polynomials over $\mathbb{GF}(2^{64})$ are represented as unsigned 64-bit integers where the Least Significant Bit (LSB) corresponds to the coefficient of the highest degree term, whereas in HQC the representation is reversed. Then, we added `refresh` gadgets to make it MIMO-SNI.

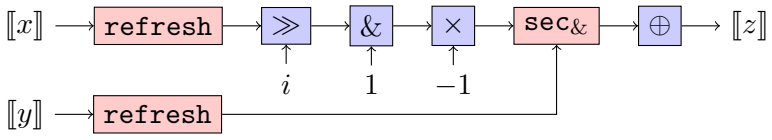
Once the entire masked implementation was functional, we protected the original polynomial multiplication and compared both solutions. Results are available in Section 5.4. As expected, the solution from masked BIKE was not only the easiest to implement but also significantly faster, with a speedup of approximately 3× for orders 1 to 5.

Theorem 7. *GF multiplication is MIMO-SNI.*

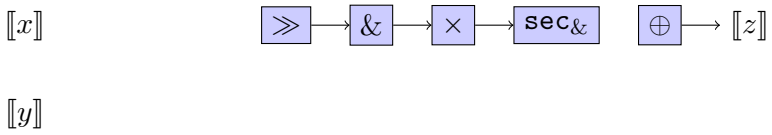
Proof. The multiplication uses the SNI $\texttt{sec}_{\&}$ gadget and $[\![z]\!]$ is updated in each loop iteration. There is no path from an input to an output, no path from a gadget to two inputs. There is only one output and there is at most one path between any two gadgets since $[\![y]\!]$ is refreshed after each use. □

Algorithm 16 $sec_{GFMult}()$: GF multiplication from [DR24b]

Input: $[\![x]\!] \in \mathbb{F}_2^B, [\![y]\!] \in \mathbb{F}_2^B$
Output: $[\![z]\!] = [\![a]\!] \cdot [\![b]\!] \in \mathbb{F}_2^{2B}$
1: $[\![x]\!] \leftarrow \texttt{refresh}([\![x]\!])$
2: $[\![y]\!] \leftarrow \texttt{refresh}([\![y]\!])$
3: $[\![z]\!] \leftarrow 0$
4: **for** $i \leftarrow 0$ to B **do**
5: $\quad [\![tmp]\!] \leftarrow (([\![x]\!] \gg i)\ \&\ 1)$
6: $\quad [\![t]\!] \leftarrow [\![tmp]\!] \times -1$ ▷ tmp = 0 or 0xF... F
7: $\quad [\![u]\!] \leftarrow \texttt{sec}_{\&}([\![t]\!], [\![y]\!])$
8: $\quad [\![x]\!] \leftarrow \texttt{refresh}([\![x]\!])$
9: $\quad [\![y]\!] \leftarrow \texttt{refresh}([\![y]\!])$
10: $\quad [\![z]\!] \leftarrow [\![z]\!] \oplus [\![u]\!]$
11: **end for**
12: **return** $[\![z]\!]$



(a) Full



(b) Simplified

Figure 4.5: Computation graph of GF multiplication

4.6 Masked HQC

In this section we present how we build the core functions of HQC (KeyGen, Encaps, Decaps, Encrypt, Decrypt) to obtain a fully-masked PINI implementation of HQC.

4.6.1 Key generation

In the HQC specification $[\![\boldsymbol{y}]\!]$ is actually sampled before $[\![\boldsymbol{x}]\!]$, and this order must be preserved to remain compliant with the specification. According to the NIST convention, the public key is appended to the end of the private key. We sample with a masked seed $[\![\mathsf{sk\_seed}]\!]$ using the masked SHA-3 gadget from [CGL+24].

In the HQC C implementation, vectors are represented as arrays of 8-bit unsigned integers. To concatenate a 64-bit masked vector $[\![\boldsymbol{a}]\!]$ with a 32-bit masked vector $[\![\boldsymbol{b}]\!]$, we treat $[\![\boldsymbol{a}]\!]$ as an array of 8 rows and d columns, where d is the masking order. We then append the 4 rows of $[\![\boldsymbol{b}]\!]$ to obtain the concatenated vector $[\![\boldsymbol{a}]\!]||[\![\boldsymbol{b}]\!]$, which has 12 rows and d columns. To concatenate a masked vector with an unprotected vector, we treat the latter as a masked vector in which only the first share (column) is populated, while all other shares are set to zero.

Algorithm 17 Key Generation

Input: $n, k, \omega, l,$ $\mathsf{pk\_seed}$,$[\![\mathsf{sk\_seed}]\!]$
Output: sk, pk
$\boldsymbol{h} \xleftarrow{\$} {}^{\mathsf{pk\_seed}}\mathcal{R}$
$[\![\boldsymbol{\sigma}]\!] \xleftarrow{\$} \mathbb{F}_2^{512}$
$[\![\boldsymbol{y}]\!] \xleftarrow{\$} {}^{[\![\mathsf{sk\_seed}]\!]}\mathcal{R}_\omega^2$
$[\![\boldsymbol{x}]\!] \xleftarrow{\$} {}^{[\![\mathsf{sk\_seed}]\!]}\mathcal{R}_\omega^2$
$[\![\boldsymbol{tmp}]\!] \leftarrow \mathsf{sec}_{\text{vect\_mul}}(\boldsymbol{h}, [\![\boldsymbol{y}]\!])$
$[\![\boldsymbol{s}]\!] \leftarrow \mathsf{sec}_{\text{vect\_add}}([\![\boldsymbol{tmp}]\!], [\![\boldsymbol{x}]\!])$
$\mathsf{pk} \leftarrow \mathsf{pk\_seed}||\boldsymbol{s}$
$\mathsf{sk} \leftarrow [\![\mathsf{sk\_seed}]\!]||[\![\boldsymbol{\sigma}]\!]||\mathsf{pk}$

4.6.2 Encrypt

Here we present a version of Encrypt with the output protected. A second version exists in which the output remains unprotected. The two versions are identical, only the output differs. Because of the re-encryption step in the Decapsulation (see Algorithm 21), the output must be protected; otherwise, an attacker could exploit it to determine whether the key exchange was successful. Or alternatively, being able to tell if the re-encrypted ciphertext is equal to the received ciphertext gives a distinguisher to the adversary. This adversary could then reproduce Guo's attack in a constant-time setting (like we have in Section 3.2.1).

The only moment when we allow to unmask the ciphertext is in the Encapsulation. It is not sensitive since it's supposed to be send publicly. Moreover, sending masked ciphertexts would be detrimental to the bandwidth of HQC.

Again, we follow the sampling order: $[\![\boldsymbol{r_2}]\!], [\![\boldsymbol{e}]\!], [\![\boldsymbol{r_1}]\!]$, to comply with HQC specification. The gadgets $\texttt{secRSEncode}()$, $\texttt{secRMEncode}()$ and $\mathsf{sec}_{\text{vect\_mul}}$ are unfortunately too large to be fully detailed in this section. However, their complete implementations are available in Appendix B. The $\mathsf{sec}_{\text{vect\_add}}$ gadget performs a sharewise XOR between the corresponding entries of the two vectors.

Algorithm 18 Encrypt

Input: $\mathsf{pk}, [\![\boldsymbol{m}]\!], [\![\boldsymbol{\theta}]\!]$
Output: $[\![\boldsymbol{u}]\!], [\![\boldsymbol{v}]\!]$
$[\![\boldsymbol{r_2}]\!] \xleftarrow{\$} {}^{[\![\theta]\!]}\mathcal{R}_{\omega_r}$
$[\![\boldsymbol{e}]\!] \xleftarrow{\$} {}^{[\![\theta]\!]}\mathcal{R}_{\omega_e}$
$[\![\boldsymbol{r_1}]\!] \xleftarrow{\$} {}^{[\![\theta]\!]}\mathcal{R}_{\omega_r}$
$[\![\boldsymbol{u}]\!] \leftarrow \mathsf{sec}_{\text{vect\_mul}}([\![\boldsymbol{r_2}]\!], \boldsymbol{h})$
$[\![\boldsymbol{u}]\!] \leftarrow \mathsf{sec}_{\text{vect\_add}}([\![\boldsymbol{u}]\!], [\![\boldsymbol{r_1}]\!])$
$[\![\boldsymbol{cdw_{RS}}]\!] \leftarrow \texttt{secRSEncode}([\![\boldsymbol{m}]\!])$
$[\![\boldsymbol{v}]\!] \leftarrow \texttt{secRMEncode}([\![\boldsymbol{cdw_{RS}}]\!])$
$[\![\boldsymbol{tmp}]\!] \leftarrow \mathsf{sec}_{\text{vect\_mul}}([\![\boldsymbol{r_2}]\!], \boldsymbol{s})$
$[\![\boldsymbol{tmp}]\!] \leftarrow \mathsf{sec}_{\text{vect\_add}}([\![\boldsymbol{tmp}]\!], [\![\boldsymbol{e}]\!])$
$[\![\boldsymbol{v}]\!] \leftarrow \mathsf{sec}_{\text{vect\_add}}([\![\boldsymbol{v}]\!], [\![\boldsymbol{tmp}]\!])$

4.6.3 Encapsulation

Algorithm 19 Encapsulation

Input: pk
$[\![\boldsymbol{m}]\!] \xleftarrow{\$} \mathbb{F}_2^k$
$\mathsf{pk}_{1,\dots,32} \leftarrow (\mathsf{pk}_1, \dots, \mathsf{pk}_{32})$ ▷ extract first 32 bytes of pk
$\boldsymbol{salt} \xleftarrow{\$} \mathbb{F}_2^{128}$
$[\![\boldsymbol{tmp}]\!] \leftarrow [\![\boldsymbol{m}]\!] || \mathsf{pk}_{1,\dots,32} || \boldsymbol{salt}$
$[\![\boldsymbol{\theta}]\!] \xleftarrow{\$} {}^{[\![\boldsymbol{tmp}]\!]}$
$\boldsymbol{u}, \boldsymbol{v} \leftarrow \mathsf{sec}_{\text{encrypt}}(\mathsf{pk}, [\![\boldsymbol{m}]\!], [\![\boldsymbol{\theta}]\!])$
$[\![\boldsymbol{mc}]\!] \leftarrow [\![\boldsymbol{m}]\!] || \boldsymbol{u} || \boldsymbol{v}$
$[\![\boldsymbol{ss}]\!] \xleftarrow{\$} {}^{[\![\boldsymbol{mc}]\!]}\mathbb{F}_2^{512}$
$\mathsf{ct} \leftarrow \boldsymbol{u} || \boldsymbol{v} || \boldsymbol{salt}$
return ct

4.6.4 Decrypt

In the Decryption step, we are supposed to compute the quantity: $\mathcal{C}.\text{Decode}(\boldsymbol{v} - \boldsymbol{uy})$. Since we work in characteristic 2, it is equivalent to computing $\mathcal{C}.\text{Decode}(\boldsymbol{v} + \boldsymbol{uy})$.

The gadgets `secRSDecode`() and `secRMDecode`() are unfortunately too large to be fully detailed in this section. However, their complete implementations are available in Appendix B.

Algorithm 20 Decrypt

Input: $[\![\mathsf{sk\_seed}]\!], \mathsf{ct}$
Output: $[\![\boldsymbol{m}]\!]$
$[\![\boldsymbol{y}]\!] \xleftarrow{\$} {}^{[\![\mathsf{sk\_seed}]\!]}\mathcal{R}_\omega^2$
$[\![\boldsymbol{tmp}]\!] \leftarrow \mathsf{sec}_{\text{vect\_mul}}([\![\boldsymbol{u}]\!], [\![\boldsymbol{y}]\!])$
$[\![\boldsymbol{cdw_{RM}}]\!] \leftarrow \mathsf{sec}_{\text{vect\_add}}([\![\boldsymbol{v}]\!], [\![\boldsymbol{tmp}]\!])$
$[\![\boldsymbol{cdw_{RS}}]\!] \leftarrow \texttt{secRMDecode}([\![\boldsymbol{cdw_{RM}}]\!])$
$[\![\boldsymbol{m}]\!] \leftarrow \texttt{secRSDecode}([\![\boldsymbol{cdw_{RS}}]\!])$

4.6.5 Decapsulation

In the Decapsulation step, the received ciphertext is first decrypted, then re-encrypted, and the result is compared to the original ciphertext. The algorithm then perfoms a multiplexer (MUX) operation: if the two vectors from both ciphertexts are identical, the correct shared secret is returned; otherwise, a random key σ is returned. $[\![res]\!]$ is an 8-bit value equal to 0 if the key exchange was successful, and 255 if not. The pseudocode for $\mathsf{sec}_{\text{vect\_compare}}$ is provided in Algorithm 22 in the Appendix.

Algorithm 21 Decapsulation

Input: $[\![\mathsf{sk\_seed}]\!], \mathsf{ct}$
Output: $[\![res]\!], [\![\boldsymbol{ss}]\!]$

$\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{salt} \leftarrow \mathsf{ct}$
$[\![\boldsymbol{m'}]\!], [\![\boldsymbol{\sigma}]\!] \leftarrow \mathsf{sec}_{\text{decrypt}}([\![\mathsf{sk\_seed}]\!], \mathsf{ct})$
$\mathsf{pk}_{1,\ldots,32} \leftarrow \mathsf{pk}_1, \ldots, \mathsf{pk}_{32}$ ▷ extract first 32 bytes of pk and mask them
$[\![\boldsymbol{tmp}]\!] \leftarrow [\![\boldsymbol{m'}]\!] || \mathsf{pk}_{1,\ldots,32} || \boldsymbol{salt}$
$[\![\boldsymbol{\theta'}]\!] \xleftarrow{\$} {}^{[\![\boldsymbol{tmp}]\!]}$
$[\![\boldsymbol{u'}]\!], [\![\boldsymbol{v'}]\!] \leftarrow \mathsf{sec}_{\text{encrypt}}([\![\boldsymbol{\theta'}]\!], [\![\boldsymbol{m'}]\!], \mathsf{pk})$
$[\![cmp_1]\!] \leftarrow \mathsf{sec}_{\text{vect\_compare}}(\boldsymbol{u}, [\![\boldsymbol{u'}]\!])$
$[\![cmp_2]\!] \leftarrow \mathsf{sec}_{\text{vect\_compare}}(\boldsymbol{v}, [\![\boldsymbol{v'}]\!])$
$[\![res]\!] \leftarrow \mathsf{sec}_{\text{or}}([\![cmp_1]\!], [\![cmp_2]\!])$
$[\![res]\!] \leftarrow \mathsf{sec}_{-}([\![res]\!], 1)$
$[\![notres]\!] \leftarrow \mathsf{sec}_{\neg}([\![res]\!])$
for $i \leftarrow 1$ to 16 **do**
  $[\![mar]\!] \leftarrow \mathsf{sec}_{\&}([\![\boldsymbol{m'}]\!]_i, [\![res]\!])$
  $[\![sar]\!] \leftarrow \mathsf{sec}_{\&}([\![\boldsymbol{\sigma}]\!]_i, [\![notres]\!])$
  $[\![\boldsymbol{mc}]\!]_i \leftarrow [\![mar]\!] \oplus [\![sar]\!]$
end for
$[\![\boldsymbol{mc}]\!] \leftarrow [\![\boldsymbol{mc}]\!] || \boldsymbol{u} || \boldsymbol{v}$
$[\![\boldsymbol{ss}]\!] \xleftarrow{\$} {}^{[\![\boldsymbol{mc}]\!]}\mathbb{F}_2^{512}$

4.7 Verifying MIMO-SNI properties

We developed a tool that automatically generates a graphical representation of an algorithm, where the nodes are the elementary operations and the edges represent the data flows. This *ad hoc* solution is built upon a Python parser inspired by white box analysis methods. The parser scans a pre-compiled C file in order to extract function calls and link their inputs and outputs together. It then produces a file containing the graph's vertexes and edges in DOT format, that we plot with the `igraph` library [igr]. With this tool, we are able to generate the computation graph of simple masked algorithms, like the Barrett reduction as illustrated in Figure 4.6. In addition, a Python-based verifier reads the DOT file [dot] and traverses the graph to ensure that the four conditions for compliance with the MIMO-SNI model are met. If any violation is detected, the graphical representation helps us identify the most efficient placement of refresh operations, with the dual objective of minimizing their number and ensuring that the required security properties are satisfied.

4.7.1 Limitations and future directions

As previously mentioned, the tool that parses the code and generates a graphical representation is currently limited to simple algorithms. At this stage, it does not handle branching structures such as loops or conditional statements (`if`/`else`). Moreover, it relies on inconvenient conventions: for example, code manipulating masked variables must follow a strict pattern where each operation is written as a function call, so it can be correctly mapped to a vertex in the graph. The main challenge resides in the fact that we are not interested in the program's call tree (that could be obtain via many existing tools), but rather in tracing how masked variables evolve during execution. This is the difference between a function call graph and a data flow graph. The first one represents how a program invokes

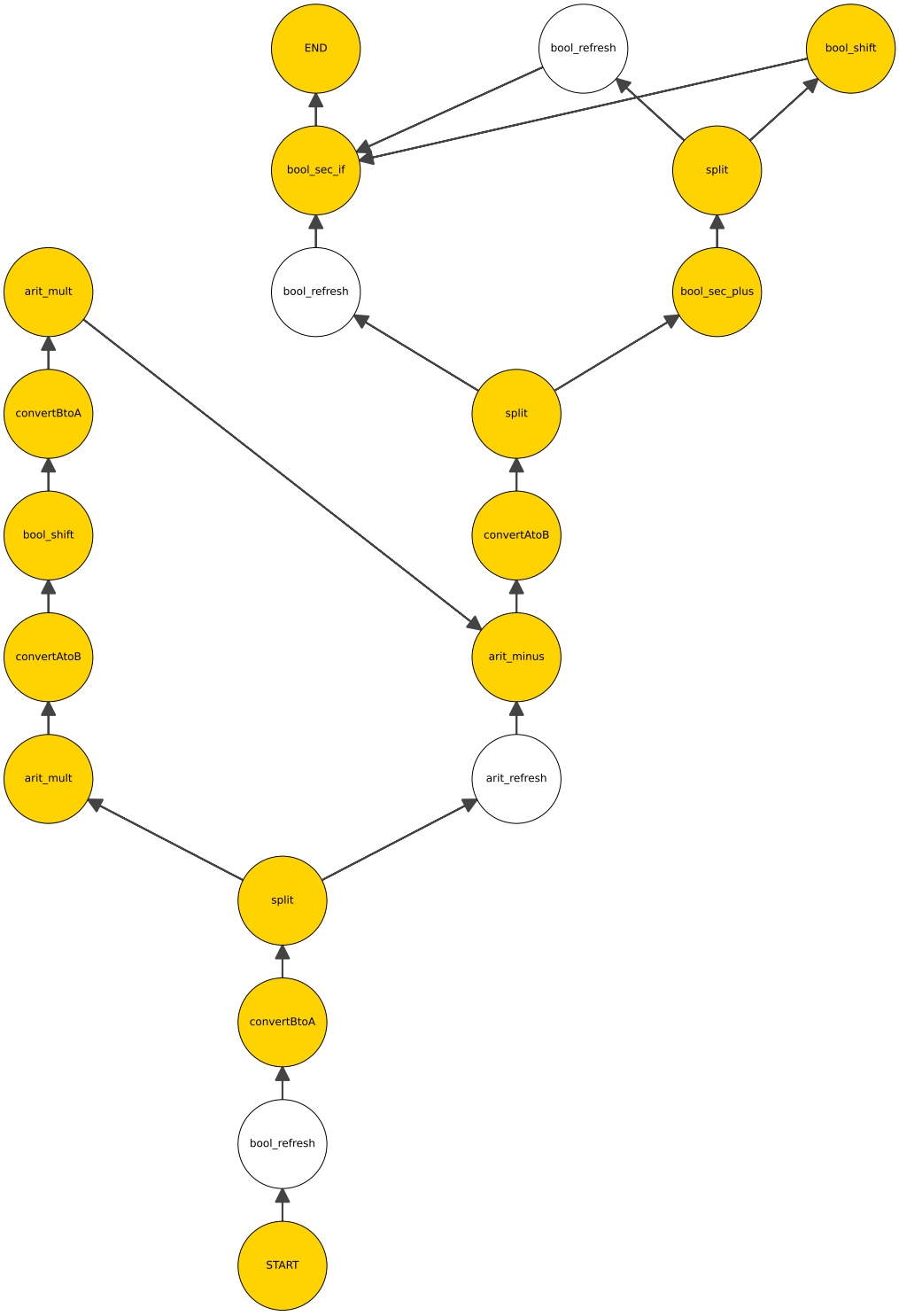


Figure 4.6: Barrett reduction DAG

functions, with nodes corresponding to functions and directed edges indicating that one function calls another during execution. In contrast, the second one represents how data (in our case, masked variables), evolve through a program. Its nodes represent operations (in our case, masked gadgets), and its directed edges represent the flow of data between them.

While the first improvement would be to generalize the tool to support more complex gadgets, we could also envision automating the verifier. Instead of just verifying graph properties, it could automatically fix violations by inserting appropriate refresh gadgets where needed. The ultimate goal would be to minimize the number of refreshes while ensuring compliance with the MIMO-SNI composition conditions. To this end, AI-assisted methods [ZLY+22] might offer promising solutions by tackling this as a global optimization problem: identifying the most efficient refresh placements across the entire algorithm, not just within individual gadgets, while still preserving MIMO-SNI security guarantees.

Chapter 5

Experimental evaluation

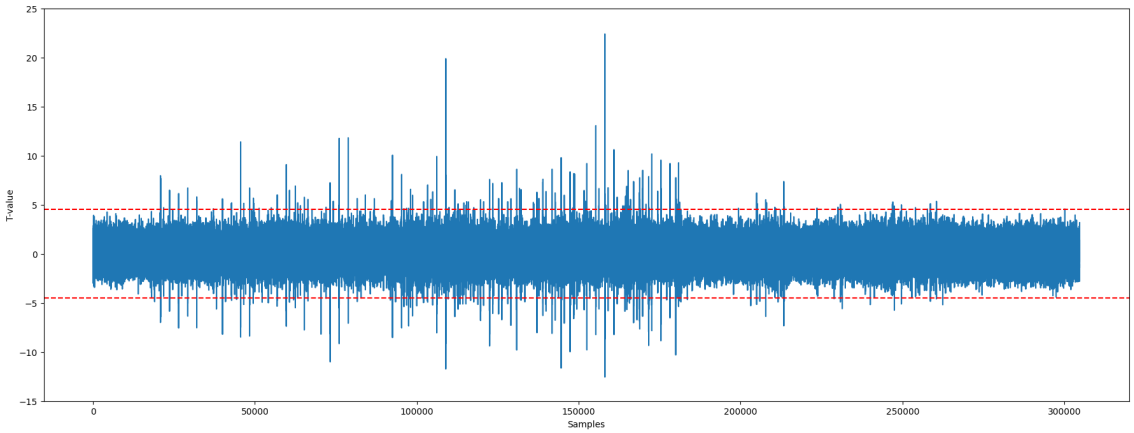
Contents

5.1 Experimental setup

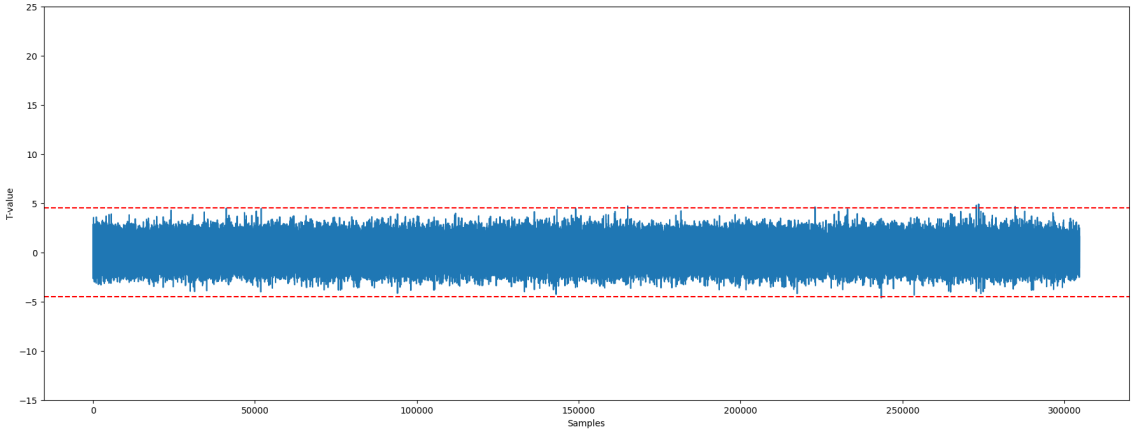
To conduct our experiments we used the STM32 Nucleo-F439ZI development board which is equipped with an ARM Cortex-M4 core running at 168 MHz. For our tests on the unprotected version of HQC, we selected its *pqm4* [KPR+] implementation, which is specially designed for the ARM Cortex-M4. We implemented some 32-bit variants of the gadgets to best fit our STM32 Nucleo-144 target board.

5.2 Boolean Barrett

We retrieved the code of the gadgets from Demange and Rossi's Github [DR24b] and implemented our solution. After setting the masking order to one, we ran 10,000 executions of our masked Barrett reduction, once with fixed inputs and once with random ones; and recorded the resulting electro-magnetic traces. For comparison purposes, we ran this first experiment while fixing the masks to zero (virtually unmasking the secret value). Using a TLVA (Test Vector Leakage Assessment) [BCD+13] we confront these two sets of traces and obtain Figure 5.1a. The numerous peaks well above and below the ±4.5 threshold [SM16] (dotted red lines) inform us of the presence of leaks. We then ran a second experiment, this time relying on the integrated TRNG (True Random Number Generator) of the board to produce the random masks required for the computation. This time, there are no visible peaks (Figure 5.1b) which is expected from a first order leakage



(a) with masks set to zero.



(b) with random masks.

Figure 5.1: TVLA of the masked Barrett reduction

analysis of a first order masking. Hence, the absence of leaks in the second experiment gives us better confidence concerning the soundness of our solution.

Remark 4. A formal security proof on the pseudo or source code is not a complete guarantee. Due to all optimizations performed by the compilation tool chain and even by the hardware, executing the corresponding compiled software on a real hardware CPU can leak secret data, even when the abstract algorithm was proved t-NI secure. Further analyses on the assembly code, on the linked and loaded binary or even on the actual execution by the processor are needed before one can conclude that the masking provides the expected security level, as demonstrated for instance in [CGP+12].

Our results were obtained with the compilation flag `-Og` and all caches ON.

5.2.1 Example of Leak Introduced by the Compiler Optimizations

When testing our solution with the more aggressive `-O3` compilation option we discovered that the compiler decided to reuse the same register for two shares as can be seen on Listing 5.1. Since we use a first order masking, the transition between the two states of the register induces a XOR between the two values, which unmasks the secret as illustrated by Figure 5.2.

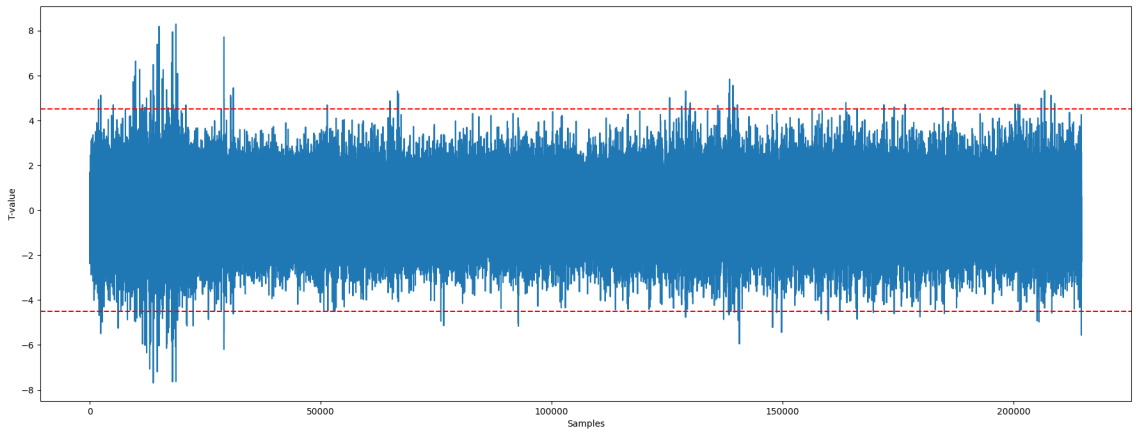


Figure 5.2: TVLA of the masked Barrett reduction compiled with `-O3`.

On Line 6, the value of `y[0]` is loaded in `r2` then on Line 8 the value of `y[1]` is loaded in `r2`. Physically, for the register to switch from the value `y[0]` to `y[1]` there is an implicit XOR between the two values. As $\mathtt{y} = \mathtt{y}[0] \oplus \mathtt{y}[1]$, for all 0-bits of `y` there is no transition (that is, no consumed energy) between the corresponding bits of `y[0]` and `y[1]`, while for all 1-bits of `y` there is a transition from 0 to 1 or from 1 to 0, that consumes energy.

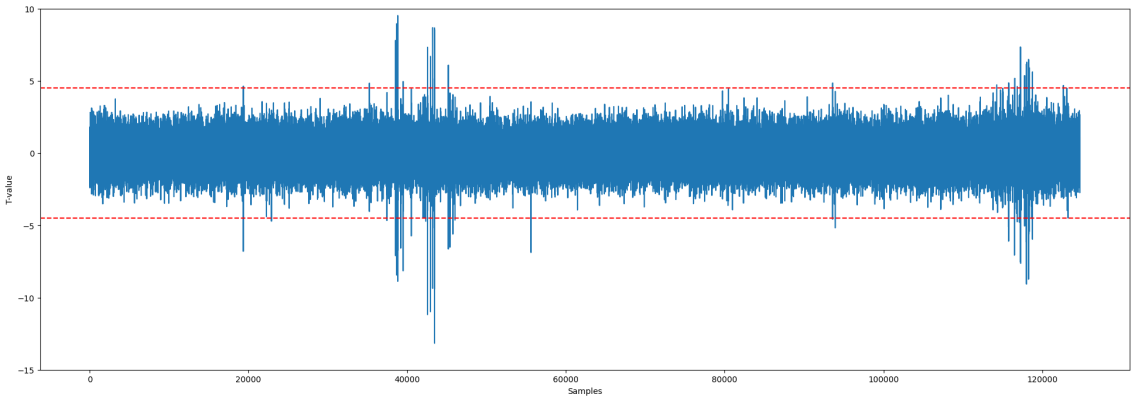
This involuntarily exposes the secret `y` and advocates for further analyses on the assembly code, on the linked and loaded binary or even on the actual execution by the processor as proposed for instance by [GHP+21]. Increasing the masking order, as suggested by [BGG+15], is another option but it is costly.

```
1 <boolean_sec_and>:
2 push {r3,r4,r5,r6,r7,lr}
3 mov  r5,r1       ;store y address in r5
4 ldrd r1,r3,[r0] ;r1 = x[0], r3 = x[1]
5 mov  r4,r2
6 ldr  r2,[r5,#0] ;r2 = y[0]
7 ands r1,r2       ;r1 = x[0] & y[0]
8 ldr  r2,[r5,#4] ;r2 = y[1]
9 ...
```

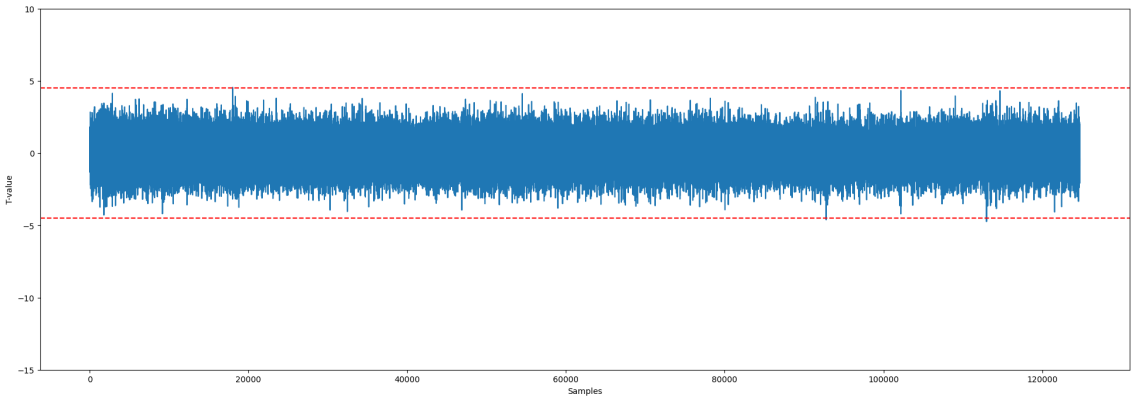
Listing 5.1: Abstract of the assembly code of $\texttt{sec}_{\&}$ for `-O3`.

5.3 Arithmetic Barrett

We ran the same experiments as described at the beginning of Section 5.2 to check if this new solution was still secure. The results are presented in Figure 5.3. As expected, using mask conversions and arithmetic masking does not affect the security of our solution.



(a) with masks set to zero.



(b) with random masks.

Figure 5.3: TVLA of the masked Barrett reduction with mask conversions.

We ran a benchmark (see Table 5.1), where we compare the sampling of a random vector of weight 75 in five different settings: the reference implementation of HQC and BIKE, HQC using our masked Barrett solutions (Boolean or arithmetic with conversions) and BIKE with the masking scheme designed by Demange and Rossi. The following results were all obtained with the compilation flag `-Og`.

Table 5.1: Comparison of the number of cycles needed to sample a random vector of weight $w = 75$ (ARM Cortex-M4 @168 MHz, caches ON).

Algorithm (vector sampling)	Execution time (cycles)	Overhead
HQC	747,500	–
HQC Boolean Barrett (order 1)	17,496,000	2340%
HQC Arithmetic Barrett (order 1)	12,203,000	1632%
BIKE	746,500	–
Masked BIKE (order 1)	11,606,000	1554%

Our fully Boolean masked solution is 51% more computationally intensive compared to the one proposed for BIKE. This is due to the fact that BIKE specification only requires a masked multiplication and a shift. In comparison, following HQC specification requires to execute the entire masked Barrett reduction (Algorithm 14). However, by using the arithmetic masking to accelerate the multiplications, our second solution achieves a performance only 5% slower than BIKE, despite the conversions. Thus, we think that the impact of our second solution on the performance is limited while providing the key advantage of preserving the HQC specification.

5.4 GF multiplication

To compare the different multiplication methods, we ran 10000 multiplications and recorded the elapsed number of cycles with the "__rdtsc()" intrinsic. Then we calculated the average number of cycles for one multiplication and compared for different masking orders. This benchmark was run on a laptop equipped with an Intel Core i7-13700H. The results are shown in Table 5.2.

Masking order	1	2	3	4	5
Masked HQC mult	7919	20244	36540	53362	71846
Masked BIKE mult	2938	7497	12090	16177	24338
Speedup	×3.40	×2.70	×3.02	×3.36	×2.95

Table 5.2: Comparison of the number of cycles for a masked GF multiplication (BIKE vs HQC).

5.5 PINI vs. NI

Since the masked BIKE implementation is NI-secure, we wanted to study the impact of the two security models on performance. To this end, we developed an NI implementation of HQC, based on the composition rule established in [BBD+16]. We performed a series of benchmarks on our masked implementations to compare the execution times for different numbers of shares and different security models. We conducted these experiments on a laptop equipped with an Intel Core Ultra 7 165U processor, 16 GB of RAM, a 512 GB SSD, and running a Windows operating system. The processor was configured to maintain a fixed frequency of 2.69GHz, ensuring stable experimental conditions. To measure executions times, we directly accessed the processor's time-stamp counter unsing the `rdtsc`

intrinsic, which directly provides the number of CPU clock cycles. We compiled our HQC masked implementations using `gcc` 11.4.0 with the `-O3` optimization flag.

For each masking order from 0 to 7 (1 to 8 shares) we repeated the experiment 120 times, preceded by a warm-up step, to obtain an average time for KeyGen, Encaps, and Decaps. We did a benchmark for two different HQC implementations with two security models, NI and PINI, to estimate the impact of increased security on performance. The results of the benchmarks are shown in Table 5.3 for the PINI security model and in Table 5.4 for the NI security model. We also provide scaled values, with reference 1 for masking order 0, for easier comparison of the impact of masking orders.

The order-0 masking configuration corresponds to an implementation using a single share per variable. While this configuration provides no protection for sensitive variables, it serves as a baseline for comparing execution time scaling across implementations with masking structures.

Order	0	1	2	3	4	5	6	7
HQC-128								
KeyGen	6.3	43.7	95.1	247.0	328.1	447.1	644.4	845.3
KeyGen scaled	×1.0	×7.0	×15.1	×39.3	×52.2	×71.2	×102.6	×134.6
Encaps	11.7	81.7	181.5	463.9	622.5	843.4	1230.8	1608.9
Encaps scaled	×1.0	×7.0	×15.6	×39.8	×53.4	×72.4	×105.6	×138.1
Decaps	25.8	128.3	270.9	675.1	911.5	1220.5	1827.0	2374.5
Decaps scaled	×1.0	×5.0	×10.5	×26.2	×35.4	×47.4	×70.9	×92.2
HQC-192								
KeyGen	19.0	122.5	262.9	690.8	916.1	1211.5	1787.7	2338.2
KeyGen scaled	×1.0	×6.5	×13.9	×36.4	×48.3	×63.8	×94.2	×123.2
Encaps	35.2	227.5	492.9	1285.5	1718.8	2297.1	3378.0	4433.1
Encaps scaled	×1.0	×6.5	×14.0	×36.5	×48.9	×65.3	×96.0	×126.0
Decaps	61.9	332.2	705.4	1799.7	2429.1	3231.9	4800.1	6267.4
Decaps scaled	×1.0	×5.4	×11.4	×29.1	×39.3	×52.2	×77.6	×101.3
HQC-256								
KeyGen	36.9	232.3	509.1	1352.8	1779.7	2340.4	3484.1	4574.9
KeyGen scaled	×1.0	×6.3	×13.8	×36.7	×48.3	×63.5	×94.5	×124.1
Encaps	67.7	428.2	947.3	2486.2	3303.9	4371.5	6509.0	8517.3
Encaps scaled	×1.0	×6.3	×14.0	×36.7	×48.8	×64.5	×96.1	×125.7
Decaps	115.8	618.3	1340.8	3461.1	4630.2	6129.8	9162.8	11975.6
Decaps scaled	×1.0	×5.3	×11.6	×29.9	×40.0	×52.9	×79.1	×103.4

Table 5.3: Benchmarks of PINI HQC masked implementation (in millions of cycles) with `-O3` optimization flag.

As expected, increasing the security requirements introduces additional performance and computational overhead. We observe an overhead of approximately 5 to 10% for the PINI implementation compared to the NI version. In this performance-security trade-off, one may choose between *(i)* an implementation secure in the NI model, which remains comparable to most state-of-the-art masked PQC schemes [DR24a, ABC+22b], or *(ii)* an implementation that satisfies the PINI security model.

Figure 5.4 represents the scaling of the number of cycles of our implementation with the masking order. We use the execution time of our order-0 masked implementation as the reference value to construct the plot.

Order	0	1	2	3	4	5	6	7
HQC-128								
KeyGen	6.2	43.0	93.6	239.6	318.4	421.4	620.8	817.6
KeyGen scaled	×1.0	×7.0	×15.2	×38.8	×51.6	×68.2	×100.5	×132.4
Encaps	11.4	79.7	178.0	446.5	610.1	805.6	1179.8	1538.8
Encaps scaled	×1.0	×7.0	×15.6	×39.1	×53.5	×70.6	×103.4	×134.8
Decaps	25.3	125.5	263.7	650.4	878.4	1170.7	1749.3	2264.9
Decaps scaled	×1.0	×5.0	×10.4	×25.7	×34.7	×46.2	×69.1	×89.4
HQC-192								
KeyGen	18.3	117.0	257.2	670.2	878.8	1175.5	1735.3	2256.4
KeyGen scaled	×1.0	×6.4	×14.0	×36.6	×48.0	×64.1	×94.7	×123.1
Encaps	34.0	217.0	483.1	1231.6	1648.2	2223.5	3277.5	4253.9
Encaps scaled	×1.0	×6.4	×14.2	×36.2	×48.4	×65.3	×96.3	×125.0
Decaps	60.1	317.2	687.4	1726.1	2312.7	3117.0	4616.8	5996.5
Decaps scaled	×1.0	×5.3	×11.4	×28.7	×38.5	×51.9	×76.8	×99.8
HQC-256								
KeyGen	35.8	227.5	501.8	1305.9	1718.0	2279.9	3364.8	4357.2
KeyGen scaled	×1.0	×6.4	×14.0	×36.5	×48.0	×63.7	×94.1	×121.8
Encaps	65.3	417.8	928.0	2387.0	3171.0	4265.2	6345.3	8161.0
Encaps scaled	×1.0	×6.4	×14.2	×36.6	×48.6	×65.4	×97.2	×125.1
Decaps	112.8	604.7	1310.9	3318.5	4453.5	5972.6	8783.2	11445.3
Decaps scaled	×1.0	×5.4	×11.6	×29.4	×39.5	×52.9	×77.8	×101.4

Table 5.4: Benchmarks of NI HQC masked implementation (in millions of cycles) with `-O3` optimization flag.

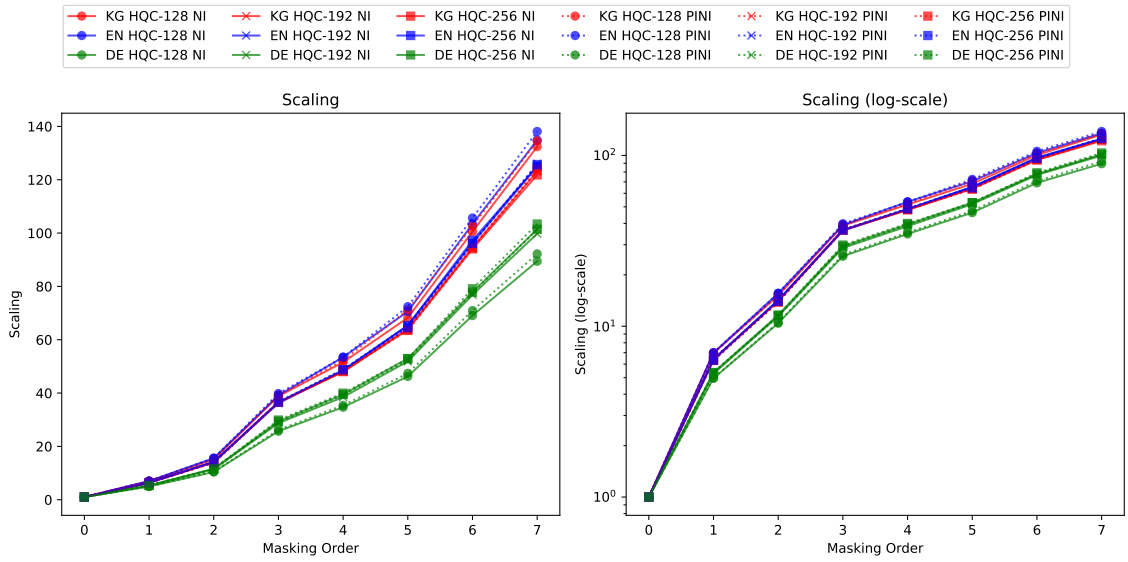


Figure 5.4: Scaling of the number of cycles with masking order, for all security levels, based on benchmarks with `-O3` optimization flag (KG: Key Generation, EC: Encapsulation, DC: Decapsulation).

5.5.1 Update of the GF multiplication

Our previous implementation of the GF multiplication contained an error. To address this issue, we adopted the masked implementation from BIKE [DR24b]. This approach remains faster than applying masking to the original HQC GF multiplication (see Section 5.4). The modification introduces a 14–30% overhead compared to our previous implementation, depending on the masking order. However, the BIKE-based multiplication improves scalability due to its better asymptotic complexity. As a result, the performance gap between the previous and the current version decreases as the masking order increases. These differences are shown in Tables 5.5 and 5.6.

Version	KeyGen	Encaps	Decaps
Previous	43.7	81.7	128.3
Current	55.1	106.4	159.4
Overhead	+26.1%	+30%	+24.2%

Table 5.5: Speed comparison (in millions of cycles) of PINI HQC (previous and current version) on x86_64 at masking order 1.

Version	KeyGen	Encaps	Decaps
Previous	447	843	1220
Current	509	990	1422
Overhead	+13.8%	+17.4%	+16.6%

Table 5.6: Speed comparison (in millions of cycles) of PINI HQC (previous and current version) on x86_64 at masking order 5.

5.5.2 Comparison to masked BIKE

Our PINI-masked implementation still remains competitive when compared to BIKE for the same masking order. For example, at masking order 5, HQC key generation requires 509 million cycles, compared to 1330 million for BIKE [DR24a]. For the same masking order, the encapsulation and decapsulation phases require 2412 million cycles for PINI HQC and 2971 million cycles for NI BIKE (see Table 5.8). On the other hand, comparing the unprotected and masked implementations shows that masking reduces HQC's competitive advantage over BIKE, suggesting that HQC is inherently more complex to mask (see Tables 5.7 and 5.8).

Name	KeyGen	Encaps	Decaps
HQC	105	197	360
BIKE	599	105	1642
Ratio	×5.7	×0.53	×4.56

Table 5.7: Speed comparison (in kilocycles) of BIKE and HQC on x86_64.

Name	KeyGen	Encaps	Decaps
HQC	509	990	1422
BIKE	1330	278	2693
Ratio	×2.61	×0.28	×1.89

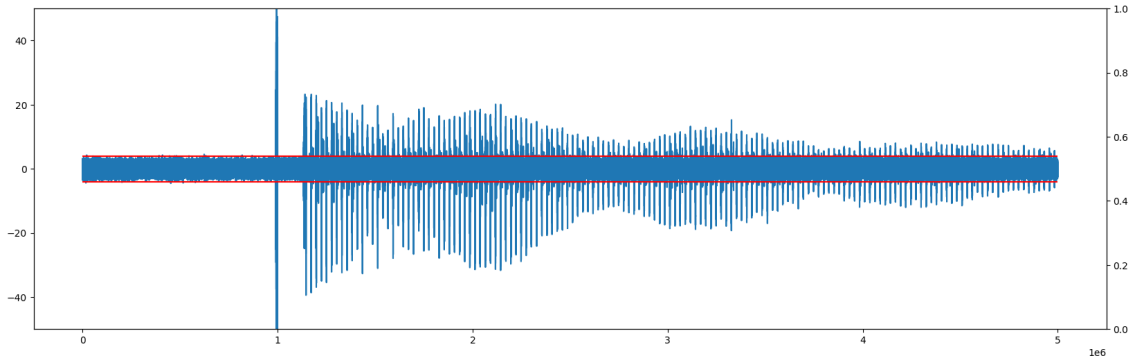
Table 5.8: Speed comparison (in millions of cycles) of NI BIKE and PINI HQC on x86_64 at masking order 5.

5.6 Masked Reed-Solomon decoder leakage analysis

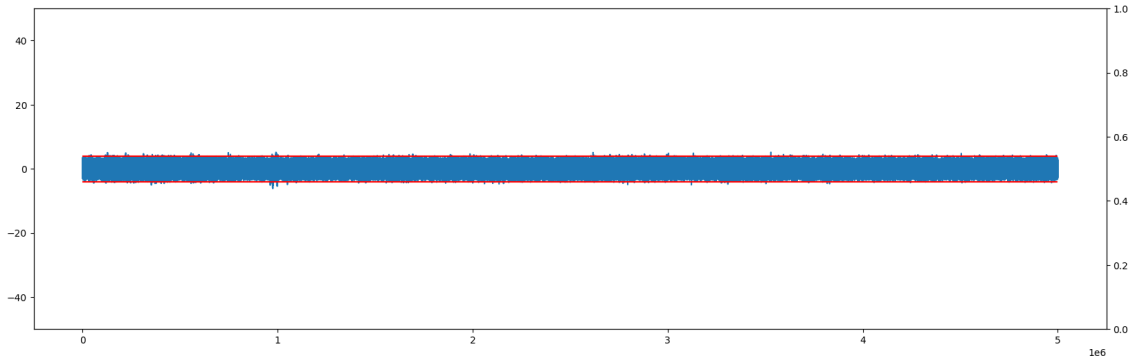
We tested the practical security of our implementation on several components; here, we focus on the masked RS decoder as a representative case. It was chosen because of its sensibility to SCA, as recalled in our sensitivity analysis (see Section 3.3). We performed a fixed-vs-random t-test, first with masks to zero to assess the leakage level. Then we ran a second t-test with random masks to find out the leakage level of our solution in its typical

use case. The results are visible in Figure 5.5. The noticeable peak at x = 1e6 corresponds to the moment when we create the input codeword that will be decoded. This is not part of the decoder, but we wanted to capture what comes before, to assess the leakage level of the algorithm when it is not manipulating the codeword. This is an important information, because we can then compare this level to the apparent leakage level when using random masks. Both have a similar amplitude, contained between -4.5/+4.5 (red horizontal lines), which conforts us in the relevance of our solution. In Figure 5.5b, small peaks are still visible around x = 1e6. Again, they are due to the creation of the codeword, they are not part of the decoder, thus not significant in the leakage analysis.

For completeness, we also ran an ANOVA leakage analysis (see Figure 5.6). At the input of the decoder, the codeword can be viewed as 40 blocks of 8 bits (i.e., 40 bytes). For this analysis, we targeted the Hamming weight of each of these 40 bytes individually. Once again, it demonstrates that the leakage is mitigated by the use of random masks, reinforcing the relevance of our countermeasure.

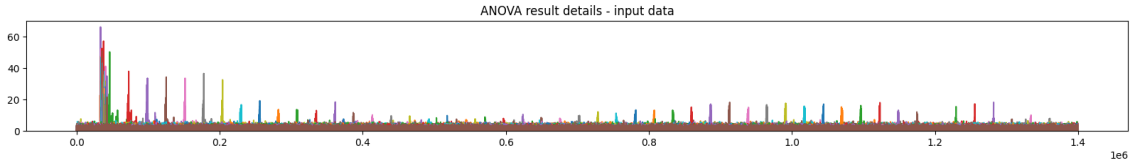


(a) with masks set to zero.

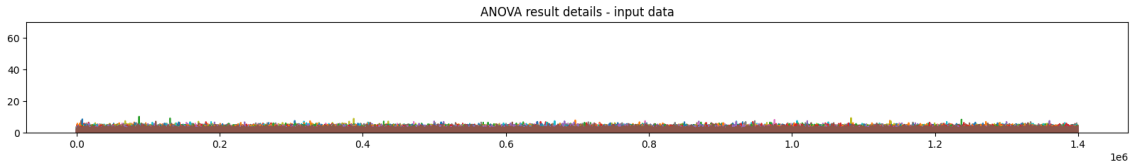


(b) with random masks.

Figure 5.5: TVLA of the masked RS decoder.



(a) with masks set to zero.



(b) with random masks.

Figure 5.6: ANOVA of the masked RS decoder

Part III

Conclusion and Open Topics

Chapter 6

Conclusion

At the start of this thesis, the goal was to analyse an algorithm competing in the fourth round of NIST post-quantum standardisation process. We selected HQC and gained a comprehensive understanding of existing implementation attacks, while exploring potential improvements. In particular, we proposed a new version of Guo *et al.*'s attack [GHJ+22], reducing by a factor of three the number of decapsulations needed to recover the secret key, and introduced a new distinguisher allowing the attack to succeed even against constant-time implementations.

Since side-channel resistance is a mandatory requirement for future post-quantum standards [AAC+22], our second objective was to provide the community with a proposal for a secure implementation, while identifying potential pain points in the process. With the use of masking and the design of new generic gadgets, we developed the first fully-masked implementation of HQC. This work was carried out in collaboration with members of the HQC team, who contributed a complete sensitivity analysis of their algorithm to guide our protection strategy. Beyond theoretical MIMO-SNI and PINI design, a full C implementation has been made available for the community as an example of put in practice. Ultimately, our masked implementation was validated through practical side-channel evaluation and benchmarked to assess its performance. It compares favourably to BIKE, the only other masked implementation of a code-based scheme, for the same masking order.

Three years later, HQC has transitioned from being a candidate to becoming a standard in the making. In this regard, our masked implementation can serve as a foundational contribution towards a side-channel-resistant standard. Although the specification of HQC may evolve during the standardisation process, and with it the implementation, our work provides a basis for further research and development.

Several directions could be explored to improve it, notably reducing its memory footprint to better suit constrained environments, and optimising execution time. Efforts could focus on reducing stack usage and improving memory reuse. Additionally, combining masking with other countermeasures may offer improved security. On the one hand, the verification of the MIMO-SNI construction could be automated, and even its design, to optimize its cost by finding the minimal number of required refresh gadgets. On the other hand, the gap between the source code and the actual executed code could be further investigated, in order to assess the impact of compilers, linkers, and micro-architectural factors, as well as the available techniques to mitigate their effects on security. Finally, although HQC has yet to be vetted against FIA (Fault Injection Attacks), developing a fault-resistant version could be a possible evolution.

Personal Statement

On a personal note, beyond the satisfaction of having worked for three years on an algorithm that was ultimately selected for standardisation, this thesis has taught me two valuable lessons. First, while a doctoral journey is inherently individual, it does not mean one must carry out everything alone. Collaboration and exchange were key in triggering the late breakthrough that led to the success of this work. Second, I learned that apprehension about ambitious goals is more limiting than the challenge itself. The masking of HQC could have started a month earlier; I delayed it, fearing the workload it would represent. In retrospect, aiming higher sooner would have been both possible and beneficial.

Bibliography

[AAC+22] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, et al. Status report on the third round of the NIST post-quantum cryptography standardization process. *US Department of Commerce, NIST*, 2022. (see pages 6 and 67)

[AAC+25] Gorjan Alagic, D Apon, D Cooper, Q Dang, T Dang, J Kelsey, J Lichtinger, C Miller, D Moody, R Peralta, et al. Status Report on the Fourth Round of the NIST Post-Quantum Cryptography Standardization Process, 2025. (see page 17)

[ABC+22a] Melissa Azouaoui, Olivier Bronchain, Gaëtan Cassiers, Clément Hoffmann, Yulia Kuzovkova, Joost Renes, Markus Schönauer, Tobias Schneider, François-Xavier Standaert, and Christine van Vredendaal. Protecting Dilithium against Leakage: Revisited Sensitivity Analysis and Improved Implementations. *Cryptology ePrint Archive*, 2022. (see page 27)

[ABC+22b] Melissa Azouaoui, Olivier Bronchain, Gaëtan Cassiers, Clément Hoffmann, Yulia Kuzovkova, Joost Renes, Markus Schönauer, Tobias Schneider, François-Xavier Standaert, and Christine van Vredendaal. Protecting Dilithium against Leakage: Revisited Sensitivity Analysis and Improved Implementations. *Cryptology ePrint Archive*, 2022. (see page 61)

[ABC+25] Gorjan Alagic, Maxime Bros, Pierre Ciadoux, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, et al. *Status report on the fourth round of the nist post-quantum cryptography standardization process*. US Department of Commerce, National Institute of Standards and Technology, 2025. (see pages 5 and 6)

[AIK07] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography with Constant Input Locality. In Alfred Menezes, editor, *Advances in Cryptology - CRYPTO 2007*, pages 92–110, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. (see page 17)

[AMAB+17] Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, and Gilles Zémor. Hamming Quasi-Cyclic (HQC). 2017. (see pages 17, 19, 20, 32, 37, and 78)

[ans] Avis de l'ANSSI sur la migration vers la cryptographie post-quantique. `https://cyber.gouv.fr/publications/avis-de-lanssi-sur-la-migration-vers-la-cryptographie-post-quantique-0`. (see page 4)

[Bar87] Paul Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 311–323, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg. (see page 46)

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 116–129, 2016. (see pages 23, 24, 25, and 60)

[BBP+16] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Randomness Complexity of Private Circuits for Multiplication. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 616–648, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. (see pages 23 and 25)

[BC22] Olivier Bronchain and Gaëtan Cassiers. Bitslicing arithmetic/boolean masking conversions for fun and profit: with application to lattice-based kems. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 553–588, 2022. (see page 27)

[BCD+13] Georg T. Becker, Jim Cooper, Elizabeth K. DeMulder, Gilbert Goodwill, Joshua Jaffe, Gary Kenworthy, T. Kouzminov, Andrew J. Leiserson, Mark E. Marson, Pankaj Rohatgi, and Sami Saab. Test Vector Leakage Assessment (TVLA) methodology in practice. 2013. (see page 56)

[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. (see page 8)

[BCPZ16] Alberto Battistello, Jean-Sébastien Coron, Emmanuel Prouff, and Rina Zeitoun. Horizontal side-channel attacks and countermeasures on the ISW masking scheme. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 23–39. Springer, 2016. (see pages 43 and 45)

[BGG+15] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the Cost of Lazy Engineering for Masked Software Implementations. In Marc Joye and Amir Moradi, editors, *Smart Card Research and Advanced Applications*, pages 64–81, Cham, 2015. Springer International Publishing. (see page 58)

[BGR+21] Joppe W Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking kyber: first-and higher-order implementations, 2021. (see page 27)

[BGTZ08] Richard P. Brent, Pierrick Gaudry, Emmanuel Thomé, and Paul Zimmermann. Faster Multiplication in GF(2)[x]. In Alfred van der Poorten and Andreas Stein, editors, *Algorithmic Number Theory Symposium*, volume 5011 of *Lecture Notes in Computer Science*, pages 153–166, Banff, Canada, May 2008. Springer-Verlag. (see page 49)

[BMG+24] Chloé Baïsse, Antoine Moran, Guillaume Goy, Julien Maillard, Nicolas Aragon, Philippe Gaborit, Maxime Lecomte, and Antoine Loiseau. Secret and Shared Keys Recovery on Hamming Quasi-Cyclic with SASCA. *Cryptology ePrint Archive*, 2024. (see pages 22, 39, and 40)

[BMvT78] E. Berlekamp, R. McEliece, and H. van Tilborg. On the inherent intractability of certain coding problems (Corresp.). *IEEE Transactions on Information Theory*, 24(3):384–386, 1978. (see pages 17 and 19)

[CD23] Wouter Castryck and Thomas Decru. An efficient key recovery attack on SIDH. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 423–447. Springer, 2023. (see page 6)

[CGL+24] Jean-Sébastien Coron, François Gérard, Tancrède Lepoint, Matthias Trannoy, and Rina Zeitoun. Improved High-Order Masked Generation of Masking Vector and Rejection Sampling in Dilithium. *Cryptology ePrint Archive*, 2024. (see pages 43 and 50)

[CGP+12] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, and Praveen Kumar Vadnala. Conversion of Security Proofs from One Leakage Model to Another: A New Issue. In Werner Schindler and Sorin A. Huss, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 69–81, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. (see page 58)

[CGTV15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from arithmetic to boolean masking with logarithmic complexity. In *Fast Software Encryption: 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers 22*, pages 130–149. Springer, 2015. (see page 43)

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between boolean and arithmetic masking of any order. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 188–205. Springer, 2014. (see pages 43 and 48)

[che] Invited talk II by Sam Jaques (CHES 2024). `https://www.youtube.com/watch?v=eB4po9Br1YY`. (see page 4)

[CJRR99] Suresh Chari, Charanjit S Jutla, Josyula R Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*, pages 398–412. Springer, 1999. (see page 23)

[CRR02] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. Template attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 13–28. Springer, 2002. (see page 8)

[CS20] Gaëtan Cassiers and François-Xavier Standaert. Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Transactions on Information Forensics and Security*, 15:2542–2555, 2020. (see pages 11, 24, 25, 26, 41, and 42)

[DCE16] A. Adam Ding, Cong Chen, and Thomas Eisenbarth. Simpler, Faster, and More Robust T-Test Based Leakage Detection. In François-Xavier Standaert and Elisabeth Oswald, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 163–183, Cham, 2016. Springer International Publishing. (see page 9)

[DG24] Haiyue Dong and Qian Guo. OT-PCA: New Key-Recovery Plaintext-Checking Oracle Based Side-Channel Attacks on HQC with Offline Templates. Cryptology ePrint Archive, Paper 2024/1715, 2024. (see page 22)

[DH76] WHITFIELD DIFFIE and MARTIN E HELLMAN. New Directions in Cryptography. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 22(6), 1976. (see page 3)

[dot] Graphviz: DOT Language. `https://graphviz.org/doc/info/lang.html`. (see page 53)

[DR24a] Loïc Demange and Mélissa Rossi. A provably masked implementation of BIKE Key Encapsulation Mechanism. *Cryptology ePrint Archive*, 2024. (see pages 27, 43, 45, 47, 61, and 63)

[DR24b] Loïc Demange and Mélissa Rossi. Provably masked implementation of BIKE Key Encapsulation Mechanism, 2024. `https://github.com/loicdemange/masked_BIKE_code`. (see pages 49, 50, 56, and 63)

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Annual international cryptology conference*, pages 537–554. Springer, 1999. (see pages 6 and 17)

[For65] G David Forney. Concatenated codes. 1965. (see page 16)

[fre] Portail de la stratégie nationale quantique. `https://quantique.france2030.gouv.fr/`. (see page 4)

[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. (see page 27)

[Ger24] François Gerard. Dilithim masked implementaion, 2024. `https://github.com/fragerar/tches24_masked_Dilithium`. (see page 43)

[GHJ+22] Qian Guo, Clemens Hlauschek, Thomas Johansson, Norman Lahr, Alexander Nilsson, and Robin Leander Schröder. Don't reject this: Key-recovery timing attacks due to rejection-sampling in HQC and BIKE. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 223–263, 2022. (see pages 10, 21, 30, 32, 35, 67, and 77)

[GHP+21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1469–1468. USENIX Association, August 2021. (see page 58)

[GJS16] Qian Guo, Thomas Johansson, and Paul Stankovski. A key recovery attack on MDPC with CCA security using decoding errors. In *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22*, pages 789–815. Springer, 2016. (see page 20)

[GLG22a] Guillaume Goy, Antoine Loiseau, and Philippe Gaborit. A new key recovery side-channel attack on HQC with chosen ciphertext. In *International Conference on Post-Quantum Cryptography*, pages 353–371. Springer, 2022. (see pages 22, 39, and 40)

[GLG22b] Guillaume Goy, Antoine Loiseau, and Phlippe Gaborit. Estimating the Strength of Horizontal Correlation Attacks in the Hamming Weight Leakage Model: A Side-Channel Analysis on HQC KEM. In *WCC 2022: The Twelfth International Workshop on Coding and Cryptography*, page WCC_2022_paper_48, 2022. (see pages 22 and 39)

[GMGL23] Guillaume Goy, Julien Maillard, Philippe Gaborit, and Antoine Loiseau. Single trace HQC shared key recovery with SASCA. *Cryptology ePrint Archive*, 2023. `https://ia.cr/2023/1590`. (see pages 22 and 39)

[goo] Google - Our quantum computing roadmap. `https://quantumai.google/roadmap`. (see page 4)

[GP99] Louis Goubin and Jacques Patarin. DES and differential power analysis the "Duplication" method. In *Cryptographic Hardware and Embedded Systems: First InternationalWorkshop, CHES'99 Worcester, MA, USA, August 12–13, 1999 Proceedings 1*, pages 158–172. Springer, 1999. (see page 23)

[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. (see page 4)

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In *Theory of Cryptography Conference*, pages 341–371. Springer, 2017. (see page 17)

[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. (see page 27)

[ibm] IBM Quantum Computing — Technology and roadmap. `https://www.ibm.com/quantum/technology`. (see page 4)

[igr] igraph - The network analysis package. `https://igraph.org/`. (see page 53)

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology-CRYPTO 2003: 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003. Proceedings 23*, pages 463–481. Springer, 2003. (see page 23)

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*, pages 388–397. Springer, 1999. (see pages 7 and 8)

[Koc96] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996. (see page 7)

[KPR+] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. `https://github.com/mupq/pqm4`. (see pages 32 and 56)

[KSSW22] Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. Improving Software Quality in Cryptography Standardization Projects. In *IEEE European Symposium on Security and Privacy, EuroS&P 2022 - Workshops, Genoa, Italy, June 6-10, 2022*, pages 19–30, Los Alamitos, CA, USA, 2022. IEEE Computer Society. (see page 45)

[LKMM21] Oleksiy Lisovets, David Knichel, Thorben Moos, and Amir Moradi. Let's Take it Offline: Boosting Brute-Force Attacks on iPhone's User Authentication through SCA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, page 496–519, Jul 2021. (see page 9)

[MAAS+19] Dustin Moody, G Alagic, JM Alperin-Sheriff, DC Apon, DA Cooper, QH Dang, Yi-Kai Liu, CA Miller, RC Peralta, RA Perlner, et al. Status report on the first round of the nist post-quantum cryptography standardization process. Technical report, Technical report, National Institute of Standards and Technology, 2019. (see page 5)

[McE78] Robert J McEliece. A public-key cryptosystem based on algebraic. *Coding Thv*, 4244:114–116, 1978. (see page 16)

[MGTF19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking dilithium: Efficient implementation and side-channel evaluation. In *Applied Cryptography and Network Security: 17th International Conference, ACNS 2019, Bogota, Colombia, June 5–7, 2019, Proceedings 17*, pages 344–362. Springer, 2019. (see pages 27 and 28)

[MNMD25] Nathan Maillet, Cyrius Nugier, Vincent Migliore, and Jean-Christophe Deneuville. Key Recovery from Side-Channel Power Analysis Attacks on Non-SIMD HQC Decryption. Cryptology ePrint Archive, Paper 2025/1270, 2025. (see page 23)

[mos13] Setting the Scene for the ETSI Quantum-safe Cryptography Workshop, 2013. `https://docbox.etsi.org/Workshop/2013/201309_CRYPTO/e-proceedings_Crypto_2013.pdf`. (see page 5)

[nis] NIST Post-Quantum Cryptography FAQs. `https://csrc.nist.gov/Projects/post-quantum-cryptography/faqs`. (see page 4)

[nis17] Call for Proposals - Post-Quantum Cryptography, January 2017. `https://csrc.nist.gov/Projects/post-quantum-cryptography/`

post-quantum-cryptography-standardization/Call-for-Proposals. (see page 5)

[oqs] Open Quantum Safe - algorithm performance visualizations. https://openquantumsafe.org/benchmarking/. (see pages 7 and 78)

[PRJB] Thales Paiva, Prasanna Ravi, Dirmanto Jap, and Shivam Bhasin. Et tu, Brute? Side-Channel Assisted Chosen Ciphertext Attacks using Valid Ciphertexts. (see pages 39 and 40)

[PT19] Thales Bandiera Paiva and Routo Terada. A timing attack on the hqc encryption scheme. In *International Conference on Selected Areas in Cryptography*, pages 551–573. Springer, 2019. (see page 20)

[Ree54] Irving S Reed. A class of multiple-error-correcting codes and the decoding scheme. *IEEE Transactions on Information Theory*, 4(4):38–49, 1954. (see page 15)

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 413–427. Springer, 2010. (see page 24)

[RS60] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960. (see page 16)

[Sen21] Nicolas Sendrier. Secure sampling of constant-weight words–application to bike. *Cryptology ePrint Archive*, 2021. (see pages 21 and 45)

[SGG24] Robin Leander Schröder, Stefan Gast, and Qian Guo. Divide and Surrender: Exploiting Variable Division Instruction Timing in {HQC} Key Recovery Attacks. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6669–6686, 2024. (see pages 21, 22, and 46)

[Sha48] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948. (see pages 12 and 16)

[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. (see page 4)

[SHR+22] Thomas Schamberger, Lukas Holzbaur, Julian Renner, Antonia Wachter-Zeh, and Georg Sigl. A power side-channel attack on the reed-muller reed-solomon version of the HQC cryptosystem. In *International Conference on Post-Quantum Cryptography*, pages 327–352. Springer, 2022. (see pages 22, 39, and 40)

[SM16] Tobias Schneider and Amir Moradi. Leakage assessment methodology: Extended version. *Journal of Cryptographic Engineering*, 6:85–99, 2016. (see page 56)

[SRSWZ20] Thomas Schamberger, Julian Renner, Georg Sigl, and Antonia Wachter-Zeh. A Power Side-Channel Attack on the CCA2-Secure HQC KEM. In *19th Smart Card Research and Advanced Application Conference (CARDIS2020)*, 2020. (see page 21)

[usa] National Quantum Initiative. `https://www.quantum.gov/`. (see page 4)

[WTBB+20] Guillaume Wafo-Tapa, Slim Bettaieb, Loïc Bidoux, Philippe Gaborit, and Etienne Marcatel. A practicable timing attack against HQC and its countermeasure. *Advances in Mathematics of Communications*, 2020. (see page 20)

[YJ21] Wei Yang and Anni Jia. Side-Channel Leakage Detection with One-Way Analysis of Variance. *Security and Communication Networks*, 2021(1):6614702, 2021. (see page 9)

[ZLY+22] Jiayi Zhang, Chang Liu, Junchi Yan, Xijun Li, Hui-Ling Zhen, and Mingxuan Yuan. A Survey for Solving Mixed Integer Programming via Machine Learning, 2022. (see page 55)

List of Figures

List of Tables

Appendix A

Glossary

AES Advanced Encryption Standard 4, 5

ANOVA Analysis of Variance 9, 64, 65, 77

ANSSI Agence Nationale de la Sécurité des Systèmes d'Information 4

BCH Bose — Chaudhuri — Hocquenghem codes, a class of cyclic error correcting codes 20, 21

BIKE Bit Flipping Key Encapsulation, a code-based key encapsulation scheme ii, 5, 6, 7, 10, 27, 35, 45, 48, 56, 59, 60, 63, 67, 78

CMOS Complementary Metal Oxide Semi-conductor 9

CPA Correlation Power Analysis 8

DAG Directed Acyclic Graph 24, 42, 54, 77

DFR Decryption Failure Rate 6, 17, 19, 31

DPA Differential Power Analysis 8

DSA Digital Signature Algorithm 5

DSDP Decisional Syndrome Decoding Problem 17

EM Electro-Magnetic 27, 32, 34, 35, 77

FHT Fast Hadamard Transform 22

FIA Fault Injection Attacks 67

FO Fujisaki-Okamoto 6, 17

HHK Hofheinz — Hovelmanns — Kiltz 17, 19

HQC Hamming Quasi-Cyclic, a code-based key encapsulation scheme 5, 6, 7, 10, 11, 17, 19, 20, 21, 22, 23, 26, 27, 28, 30, 31, 32, 35, 37, 38, 39, 40, 41, 42, 45, 46, 48, 49, 50, 51, 56, 59, 60, 61, 62, 63, 67, 68, 77, 78

IBM International Business Machines Corporation 4

KEM Key Encapsulation Mechanism 5, 6, 17, 19, 27, 37, 38, 77

MIMO-SNI Multiple Inputs-Multiple Outputs Strong Non-Interfering ii, 11, 24, 26, 27, 41, 42, 43, 44, 47, 48, 49, 53, 55, 67

ML-KEM Module Lattice based Key Encapsulation Mechanism, a key encapsulation scheme based on module learning with errors, formerly CRYSTALS-Kyber 5, 6, 7, 27

NATO North Atlantic Treaty Organization 12

NI Non-Interfering 23, 24, 25, 26, 27, 42, 43, 44, 47, 58, 60, 61, 62, 63, 78

NIST National Institute of Standards and Technology 4, 5, 6, 10, 17, 27, 41, 50, 67

NTT Number Theoretic Transform 27

PINI Probe Isolating Non-Interfering 24, 25, 26, 27, 42, 50, 61, 63, 67, 78

PKE Public Key Encryption 6, 17, 19

PQC Post-Quantum Cryptography 5, 10, 11, 17, 22, 23, 28, 61

QC Quasi-Cyclic 15, 19, 37

RM Reed — Muller 15, 17, 19, 20, 21, 22, 23, 31, 39, 40

RS Reed — Solomon 16, 17, 19, 20, 21, 22, 39, 40, 63, 64, 65, 77, 86

RSA Rivest–Shamir–Adleman, a public-key cryptosystem 3, 4, 7

SASCA Soft Analytical Side-Channel Attack 22

SCA Side Channel Attack 7, 8, 9, 10, 11, 20, 22, 23, 27, 41, 42, 63

SDD Syndrome Decoding Distribution 17

SDP Syndrome Decoding Problem 17, 19, 37

SHA Secure Hash Algorithm 5, 50

SIKE Supersingular Isogeny Key Encapsulation, an isogeny-based key encapsulation scheme 5, 6

SNI Strong Non-Interfering 24, 25, 26, 27, 42, 43, 44, 45, 49

TLVA Test Vector Leakage Assessment 56

TRNG True Random Number Generator 56

XOF eXtendable Output Function 20, 30, 31

Appendix B

B.1 Secure vector comparison

Algorithm 22 Secure vector comparison

Input: $[\![\boldsymbol{u}]\!] \in \mathbb{F}_2^n, [\![\boldsymbol{v}]\!], \in \mathbb{F}_2^n$
Output: $[\![r]\!]$, with $r = 0$ if the vectors are identical and 1 otherwise

1: $[\![\boldsymbol{u'}]\!] \leftarrow \texttt{refresh}([\![\boldsymbol{u}]\!])$
2: $[\![\boldsymbol{v'}]\!] \leftarrow \texttt{refresh}([\![\boldsymbol{v}]\!])$
3: **for** $i \leftarrow 1$ to n **do**
4: $\quad tmp \leftarrow \texttt{sec}_{=}([\![\boldsymbol{u'}]\!]_i, [\![\boldsymbol{v'_i}]\!])$
5: $\quad r \leftarrow \texttt{sec}_{\text{or}}(tmp, r)$
6: **end for**

B.2 Reed-Muller

B.2.1 Encode

Algorithm 23 BIT0MASK

Input: $[\![a]\!] \in \mathbb{F}_2^8$
Output: $[\![out]\!]$, where out = 0x0...0 if a&1 == 0, else: out = 0xF...F

1: $[\![out]\!] \leftarrow \texttt{sec}_{\&}([\![a]\!], 1)$
2: $[\![out]\!] \leftarrow \texttt{sec}_{\text{oppos}}([\![out]\!])$

Algorithm 24 RM intermediate encode

Input: $[\![m]\!] \in \mathbb{F}_2^8$
Output: $[\![cword]\!] \in \mathbb{F}_2^{128}$
1: $[\![fw]\!] \leftarrow \texttt{secBit0Mask}([\![m]\!] \gg 7)$
2: $[\![m]\!] \leftarrow \texttt{refresh}([\![m]\!])$
3: $[\![fw]\!] \leftarrow [\![fw]\!] \oplus (\texttt{secBit0Mask}([\![m]\!])$ & 0xAAAAAAAA)
4: $[\![m]\!] \leftarrow \texttt{refresh}([\![m]\!])$
5: $[\![fw]\!] \leftarrow [\![fw]\!] \oplus (\texttt{secBit0Mask}([\![m]\!] \gg 1)$ & 0xCCCCCCCC)
6: $[\![m]\!] \leftarrow \texttt{refresh}([\![m]\!])$
7: $[\![fw]\!] \leftarrow [\![fw]\!] \oplus (\texttt{secBit0Mask}([\![m]\!] \gg 2)$ & 0xF0F0F0F0)
8: $[\![m]\!] \leftarrow \texttt{refresh}([\![m]\!])$
9: $[\![fw]\!] \leftarrow [\![fw]\!] \oplus (\texttt{secBit0Mask}([\![m]\!] \gg 3)$ & 0xFF00FF00)
10: $[\![m]\!] \leftarrow \texttt{refresh}([\![m]\!])$
11: $[\![fw]\!] \leftarrow [\![fw]\!] \oplus (\texttt{secBit0Mask}([\![m]\!] \gg 4)$ & 0xFFFF0000)
12: $[\![m]\!] \leftarrow \texttt{refresh}([\![m]\!])$
13: $[\![cword]\!]_0 \leftarrow [\![fw]\!]$
14: $[\![fw]\!] \leftarrow [\![fw]\!] \oplus \texttt{secBit0Mask}([\![m]\!] \gg 5)$
15: $[\![m]\!] \leftarrow \texttt{refresh}([\![m]\!])$
16: $[\![cword]\!]_1 \leftarrow [\![fw]\!]$
17: $[\![fw]\!] \leftarrow [\![fw]\!] \oplus \texttt{secBit0Mask}([\![m]\!] \gg 6)$
18: $[\![m]\!] \leftarrow \texttt{refresh}([\![m]\!])$
19: $[\![cword]\!]_3 \leftarrow [\![fw]\!]$
20: $[\![fw]\!] \leftarrow [\![fw]\!] \oplus \texttt{secBit0Mask}([\![m]\!] \gg 5)$
21: $[\![cword]\!]_2 \leftarrow [\![fw]\!]$

Algorithm 25 Reed-Muller encode

Input: $[\![m]\!] \in \mathbb{F}_2^{8\times n_1}$
Output: $[\![cdw]\!] \in \mathbb{F}_2^{n_1 \times n_2}$
1: **for** $i \leftarrow 1$ to 46 **do**
2: $\quad [\![cdw]\!]_{2\times i\times mul} \leftarrow \texttt{ENCODERMintermediaire}([\![m]\!]_i)$
3: $\quad$ **for** $j \leftarrow 1$ to $mul - 1$ **do** ▷ copy
4: $\quad\quad [\![cdw]\!]_{2\times i\times mul+2\times j} \leftarrow \texttt{refresh}([\![cdw]\!]_{2\times i\times mul})$
5: $\quad$ **end for**
6: **end for**

B.2.2 Decode

Algorithm 26 `secExpandAndSum`()

Input: $[\![src]\!]$, the codeword
Output: $[\![dest]\!]$, the expanded codeword

for $i \leftarrow 1$ to 2 **do** ▷ the first copy
  for $j \leftarrow 1$ to 64 **do**
    $[\![dest]\!]_{64\times i+j} \leftarrow (([\![src]\!]_i \gg j)\&1)$
  end for
end for
for $k \leftarrow 2$ to 3 **do** ▷ sum the rest of the copies
  for $i \leftarrow 1$ to 2 **do**
    for $j \leftarrow 1$ to 64 **do**
      $[\![tmp]\!] \leftarrow (([\![src]\!]_{2\times k+i} \gg j)\&1)$
      $[\![tmp]\!] \leftarrow \texttt{refresh}([\![tmp]\!])$
      $[\![dest]\!]_{64\times i+j} \leftarrow \texttt{sec}_+([\![dest]\!]_{64\times i+j}, [\![tmp]\!])$
    end for
  end for
end for

Algorithm 27 `secFHT`()

Input: $[\![src]\!]$, expanded codeword
Output: $[\![dst]\!]$, expanded codeword

$[\![p_1]\!] \leftarrow [\![src]\!]$
$[\![p_2]\!] \leftarrow [\![dst]\!]$
for $i \leftarrow 1$ to 7 **do**
  for $j \leftarrow 1$ to 64 **do**
    $[\![p_1]\!]_{2\times j} \leftarrow \texttt{refresh}([\![p_1]\!]_{2\times j})$
    $[\![p_2]\!]_j \leftarrow \texttt{sec}_+([\![p_1]\!]_{2\times j}, [\![p_1]\!]_{2\times j+1})$
    $[\![p_1]\!]_{2\times j} \leftarrow \texttt{refresh}([\![p_1]\!]_{2\times j})$
    $[\![p_2]\!]_{64+j} \leftarrow \texttt{sec}_-([\![p_1]\!]_{2\times j}, [\![p_1]\!]_{2\times j+1})$
  end for
  $[\![p_3]\!] \leftarrow [\![p_1]\!]$ ▷ swap $[\![p_1]\!]$ and $[\![p_2]\!]$
  $[\![p_1]\!] \leftarrow [\![p_2]\!]$
  $[\![p_2]\!] \leftarrow [\![p_3]\!]$
end for

Algorithm 28 `secFindPeaks()`

Input: $[\![transf]\!]$, the expanded codeword
Output: $[\![pos]\!] \in \mathbb{F}_2^8$, location of the highest value

$[\![peak_{abs}]\!] \leftarrow [\![0]\!]$
$[\![peak]\!] \leftarrow [\![0]\!]$
$[\![pos]\!] \leftarrow [\![0]\!]$
for $i \leftarrow 1$ to 128 **do**
  $[\![t]\!] \leftarrow [\![transf]\!]_i$
  $[\![t']\!] \leftarrow \texttt{refresh}(\texttt{sec}_{\text{oppos}}([\![t]\!] \gg 15)$
  $[\![t'']\!] \leftarrow \texttt{refresh}([\![t]\!])$
  $[\![t''']\!] \leftarrow \texttt{refresh}(\texttt{sec}_{\text{oppos}}([\![t]\!]))$
  $[\![abs]\!] \leftarrow [\![t]\!] \oplus ([\![t']\!] \& ([\![t'']\!] \oplus [\![t''']\!]))$ ▷ $abs = t \oplus ((-(t \gg 15)) \& (t \oplus -t))$
  $[\![peak_{abs}]\!] \leftarrow \texttt{refresh}([\![peak_{abs}]\!])$
  $[\![mask]\!] \leftarrow \texttt{sec}_{\text{oppos}}(\texttt{sec\_}([\![peak_{abs}]\!], [\![abs]\!]) \gg 15)$
  ▷ $mask = -(((peak_{abs} - abs)) \gg 15)$
  $[\![t]\!] \leftarrow \texttt{refresh}([\![t]\!])$
  $[\![peak']\!] \leftarrow \texttt{refresh}([\![peak]\!])$
  $[\![mask]\!] \leftarrow \texttt{refresh}([\![mask]\!])$
  $[\![peak]\!] \leftarrow [\![peak]\!] \oplus \texttt{sec}_{\&}([\![mask]\!], [\![peak']\!] \oplus [\![t]\!])$
  ▷ $peak = peak \oplus (mask \& (peak \oplus t))$
  $[\![pos']\!] \leftarrow \texttt{refresh}([\![pos]\!])$
  $[\![mask]\!] \leftarrow \texttt{refresh}([\![mask]\!])$
  $[\![pos]\!] \leftarrow [\![pos]\!] \oplus \texttt{sec}_{\&}([\![mask]\!], [\![pos']\!] \oplus i)$ ▷ $pos = pos \oplus (mask \& (pos \oplus i))$
  $[\![mask]\!] \leftarrow \texttt{refresh}([\![mask]\!])$
  $[\![abs]\!] \leftarrow \texttt{refresh}([\![abs]\!])$
  $[\![peak'_{abs}]\!] \leftarrow \texttt{refresh}([\![peak_{abs}]\!])$
  $[\![peak_{abs}]\!] \leftarrow [\![peak_{abs}]\!] \oplus \texttt{sec}_{\&}([\![mask]\!], [\![peak_{abs}]\!] \oplus [\![abs]\!])$
  ▷ $peak_{abs} = peak_{abs} \oplus (mask \& (peak_{abs} \oplus abs))$
  $[\![peak]\!] \leftarrow \texttt{refresh}([\![peak]\!])$
  $[\![pos]\!] \leftarrow \texttt{sec}_{\text{or}}([\![pos]\!], \texttt{sec\_}([\![peak]\!] \gg 15, 1) \& 128)$ ▷ $pos| = 128 \& ((peak \gg 15) - 1)$
end for

Algorithm 29 Reed-Muller decode

Input: $[\![cdw]\!] \in \mathbb{F}_2^{n_1 \times n_2}$, received codeword
Output: $[\![msg]\!] \in \mathbb{F}_2^{8 \times n_1}$, decoded message

for $i \leftarrow 1$ to n_1 **do**
  $[\![expanded]\!] \leftarrow \texttt{secExpandAndSum}([\![cdw]\!]_{2 \times i \times mul})$ ▷ mul :see Table 2.2
  $[\![transf]\!] \leftarrow \texttt{secFHT}([\![expanded]\!])$
  $[\![transf]\!]_0 \leftarrow \texttt{sec\_}([\![transf]\!]_0, 64 \times mul)$
  $[\![msg]\!]_i \leftarrow \texttt{secFindPeaks}([\![transf]\!])$
end for

B.3 GF

Algorithm 30 $sec_{demiGFMult}()$

Input: $[\![a]\!] \in \mathbb{F}_2^B, b \in \mathbb{F}_2^B$
Output: $[\![c]\!] \in \mathbb{F}_2^B$
1: **for** $i \leftarrow 1$ to D **do** ▷ D: order of masking
2: $\quad [\![c]\!]_i \leftarrow \texttt{GFMult}([\![a]\!]_i, b)$
3: **end for**

Algorithm 31 $sec_{GFInverse}()$

Input: $[\![a]\!] \in GF(2^8)$
Output: $[\![inv]\!]$, the inverse of $[\![a]\!]$ in $GF(2^8)$
1: $[\![a]\!] \leftarrow \texttt{refresh}([\![a]\!])$
2: $[\![inv]\!] \leftarrow sec_{GFSquare}([\![a]\!])$
3: $[\![a]\!] \leftarrow \texttt{refresh}([\![a]\!])$
4: $[\![tmp_1]\!] \leftarrow sec_{GFMult}([\![inv]\!], [\![a]\!])$
5: $[\![inv]\!] \leftarrow \texttt{refresh}([\![inv]\!])$
6: $[\![inv^2]\!] \leftarrow sec_{GFSquare}([\![inv]\!])$
7: $[\![tmp_2]\!] \leftarrow sec_{GFMult}([\![inv^2]\!], [\![tmp_1]\!])$
8: $[\![inv^2]\!] \leftarrow \texttt{refresh}([\![inv^2]\!])$
9: $[\![tmp_1]\!] \leftarrow sec_{GFMult}([\![inv^2]\!], [\![tmp_2]\!])$
10: $[\![inv^2]\!] \leftarrow \texttt{refresh}([\![inv^2]\!])$
11: $[\![inv]\!] \leftarrow sec_{GFMult}([\![tmp_1]\!], [\![tmp_2]\!])$
12: $[\![inv^2]\!] \leftarrow sec_{GFSquare}([\![inv]\!])$
13: $[\![inv]\!] \leftarrow sec_{GFSquare}([\![inv^2]\!])$
14: $[\![inv^2]\!] \leftarrow sec_{GFSquare}([\![inv]\!])$
15: $[\![tmp_2]\!] \leftarrow \texttt{refresh}([\![tmp_2]\!])$
16: $[\![inv]\!] \leftarrow sec_{GFMult}([\![inv^2]\!], [\![tmp_2]\!])$
17: $[\![inv^2]\!] \leftarrow sec_{GFSquare}([\![inv]\!])$
18: $[\![inv]\!] \leftarrow [\![inv^2]\!]$

Algorithm 32 $sec_{GFSquare}()$

Input: $[\![a]\!] \in GF(2^8)$
Output: $[\![res]\!] \in GF(2^8)$
1: $[\![b]\!] \leftarrow [\![a]\!]$
2: $[\![a]\!] \leftarrow \texttt{refresh}([\![a]\!])$
3: $[\![res]\!] \leftarrow sec_{GFMult}([\![a]\!], [\![b]\!])$
4: $[\![res]\!] \leftarrow \texttt{refresh}([\![res]\!])$

B.4 Reed-Solomon

B.4.1 Encode

Algorithm 33 `secRSEncode`()

Input: $[\![msg]\!] \in \mathbb{F}_2^{128}$
Output: $[\![cdw]\!] \in \mathbb{F}_2^{8\times n_1}$
1: **for** $i \leftarrow 1$ to k **do**
2: $\quad [\![cdw]\!]_{n_1-k+i} \leftarrow [\![msg]\!]_i$
3: $\quad [\![msg]\!]_i \leftarrow \texttt{refresh}([\![msg]\!]_i)$
4: **end for**
5: **for** $j \leftarrow 1$ to $n_1 - k$ **do**
6: $\quad$ **for** $i \leftarrow 1$ to k **do**
7: $\quad\quad [\![tmp]\!] \leftarrow sec_{demiGFMult}([\![msg]\!]_i, rsGen_{i,j})$
8: $\quad\quad$ ▷ $rsGen$:generator matrix of the RS code
9: $\quad\quad [\![cdw]\!]_j \leftarrow [\![cdw]\!]_j \oplus [\![tmp]\!]$
10: $\quad$ **end for**
11: **end for**

B.4.2 Decode

Algorithm 34 `secRSComputeSyndromes`()

Input: $[\![cdw]\!] \in \mathbb{F}_2^{8\times\delta}$
Output: $[\![synd]\!] \in \mathbb{F}_2^{8\times n_1}$
1: **for** $i \leftarrow 1$ to $2 \times \delta$ **do**
2: $\quad [\![synd]\!]_i \leftarrow 0$
3: $\quad$ **for** $j \leftarrow 2$ to n_1 **do**
4: $\quad\quad [\![tmp]\!] \leftarrow sec_{demiGFMult}([\![cdw]\!]_j, \alpha Pow_{i,j})$
5: $\quad\quad$ ▷ αPow: precomputed Galois-field powers for RS operations
6: $\quad\quad [\![synd]\!]_i \leftarrow [\![synd]\!]_i \oplus [\![tmp]\!]$
7: $\quad\quad [\![cdw]\!]_j \leftarrow \texttt{refresh}([\![cdw]\!]_j)$
8: $\quad$ **end for**
9: $\quad [\![synd]\!]_i \leftarrow [\![synd]\!]_i \oplus [\![cdw]\!]_1$
10: $\quad [\![cdw]\!]_1 \leftarrow \texttt{refresh}([\![cdw]\!]_1)$
11: **end for**

Algorithm 35 `secRSComputeZPoly()`

Input: $[\![\sigma]\!]$ error-locator polynomial, $[\![deg]\!]$ degree of σ, $[\![synd]\!] \in \mathbb{F}_2^{32\times\delta}$
Output: $[\![z]\!] \in \mathbb{F}_2^{16\times(\delta+1)}$

$[\![z]\!] \leftarrow 1$
for $i \leftarrow 2$ to $\delta + 1$ **do**
  $[\![tmp]\!] \leftarrow i - 1$
  $[\![tmp_2]\!] \leftarrow \texttt{sec\_}([\![tmp]\!], [\![degree]\!])$
  $[\![degree]\!] \leftarrow \texttt{refresh}([\![degree]\!])$
  $[\![tmp]\!] \leftarrow [\![tmp_2]\!] \gg 15$
  $[\![mask]\!] \leftarrow \texttt{sec}_{\text{oppos}}([\![tmp]\!])$
  $[\![z]\!]_i \leftarrow \texttt{sec}_{\&}([\![mask]\!], [\![\sigma]\!]_i)$
end for
$[\![z]\!]_2 \leftarrow [\![z]\!]_2 \oplus [\![synd]\!]_1$
$[\![synd]\!]_1 \leftarrow \texttt{refresh}([\![synd]\!]_1)$
for $i \leftarrow 3$ to δ **do**
  $[\![tmp]\!] \leftarrow i - 1$
  $[\![tmp_2]\!] \leftarrow \texttt{sec\_}([\![tmp]\!], [\![degree]\!])$
  $[\![degree]\!] \leftarrow \texttt{refresh}([\![degree]\!])$
  $[\![tmp]\!] \leftarrow [\![tmp_2]\!] \gg 15$
  $[\![mask]\!] \leftarrow \texttt{sec}_{\text{oppos}}([\![tmp]\!])$
  $[\![tmp]\!] \leftarrow \texttt{sec}_{\&}([\![mask]\!], [\![synd]\!]_{i-1})$
  $[\![synd]\!]_{i-1} \leftarrow \texttt{refresh}([\![synd]\!]_{i-1})$
  $[\![z]\!]_i \leftarrow [\![z]\!]_i \oplus [\![tmp]\!]$
  for $j \leftarrow 2$ to $i - 1$ **do**
    $[\![tmp]\!] \leftarrow sec_{GFMult}([\![\sigma]\!]_j, [\![synd]\!]_{i-j-1})$
    $[\![tmp_2]\!] \leftarrow \texttt{sec}_{\&}([\![mask]\!], [\![tmp]\!])$
    $[\![z]\!]_i \leftarrow [\![z]\!]_i \oplus [\![tmp_2]\!]$
  end for
end for

Algorithm 36: secRSComputeElp

Input: $[\![synd]\!]$
Output: $[\![\sigma]\!]$ error-locator polynomial, $[\![deg_\sigma]\!]$

$[\![X\sigma p]\!] \leftarrow [\,0, 1, 0 \ldots, 0\,]$ (length $\delta + 1$)
$[\![pp]\!] \leftarrow -1$
$[\![dp]\!] \leftarrow 1$
$[\![d]\!] \leftarrow [\![synd]\!]_1$
$[\![\sigma]\!]_1 \leftarrow 1$
$[\![deg_{\sigma p}]\!] \leftarrow 0$
$[\![deg_\sigma]\!] \leftarrow 0$
for $\mu \leftarrow 1$ to $2 \times \delta$ **do**
  $[\![mu]\!] \leftarrow \mu$
  $[\![\sigma_{copy}]\!] \leftarrow [\![\sigma]\!]$
  $[\![deg_{\sigma_{copy}}]\!] \leftarrow [\![deg_\sigma]\!]$
  $[\![tmp]\!] \leftarrow sec_{GFInverse}([\![dp]\!])$
  $[\![dd]\!] \leftarrow sec_{GFMult}([\![d]\!], [\![tmp]\!])$
  $[\![d]\!] \leftarrow \texttt{refresh}([\![d]\!])$

15: **for** $i \leftarrow 2$ to $\min(\mu + 1, \delta)$ **do**
16: $[\![tmp]\!] \leftarrow sec_{GFMult}([\![dd]\!], [\![X\sigma p]\!]_i)$
17: $[\![X\sigma p]\!]_i \leftarrow \texttt{refresh}([\![X\sigma p]\!]_i)$
18: $[\![dd]\!] \leftarrow \texttt{refresh}([\![dd]\!])$
19: $[\![\sigma]\!]_i \leftarrow [\![sigma]\!]_i \oplus [\![tmp]\!]$
20: **end for**
21: $[\![deg_X]\!] \leftarrow \texttt{sec\_}([\![mu]\!], [\![pp]\!])$
22: $[\![pp]\!] \leftarrow \texttt{refresh}([\![pp]\!])$
23: $[\![deg_{X\sigma p}]\!] \leftarrow \texttt{sec}_{+}([\![deg_X]\!], [\![deg_{\sigma p}]\!])$
24: $[\![tmp]\!] \leftarrow \texttt{sec}_{\text{oppos}}([\![d]\!])$
25: $[\![tmp]\!] \leftarrow [\![tmp]\!] \gg 15$
26: $[\![mask_1]\!] \leftarrow \texttt{sec}_{\text{oppos}}([\![tmp]\!])$
27: $[\![deg_\sigma]\!] \leftarrow \texttt{refresh}([\![deg_\sigma]\!])$
28: $[\![tmp]\!] \leftarrow \texttt{sec\_}([\![deg_\sigma]\!], [\![deg_{X\sigma p}]\!])$
29: $[\![tmp]\!] \leftarrow [\![tmp]\!] \gg 15$
30: $[\![mask_2]\!] \leftarrow \texttt{sec}_{\text{oppos}}([\![tmp]\!])$
31: $[\![mask_3]\!] \leftarrow \texttt{sec}_{\&}([\![mask_1]\!], [\![mask_2]\!])$
32: $[\![deg_\sigma]\!] \leftarrow \texttt{refresh}([\![deg_\sigma]\!])$
33: $[\![deg_{X\sigma p}]\!] \leftarrow \texttt{refresh}([\![deg_{X\sigma p}]\!])$
34: $[\![tmp]\!] \leftarrow [\![deg_{X\sigma p}]\!] \oplus [\![deg_\sigma]\!]$
35: $[\![tmp_2]\!] \leftarrow \texttt{sec}_{\&}([\![tmp]\!], [\![mask_3]\!])$
36: $[\![deg_\sigma]\!] \leftarrow \texttt{refresh}([\![deg_\sigma]\!])$
37: $[\![deg_\sigma]\!] \leftarrow [\![deg_\sigma]\!] \oplus [\![tmp_2]\!]$
38: **if** $\mu == 2 \times \delta - 1$ **then**
39: **break**
40: **end if**
41: $[\![mu]\!] \leftarrow \texttt{refresh}([\![mu]\!])$
42: $[\![tmp]\!] \leftarrow [\![mu]\!] \oplus [\![pp]\!]$
43: $[\![mask_3]\!] \leftarrow \texttt{refresh}([\![mask_3]\!])$
44: $[\![tmp_2]\!] \leftarrow \texttt{sec}_{\&}([\![tmp]\!], [\![mask_3]\!])$
45: $[\![pp]\!] \leftarrow \texttt{refresh}([\![pp]\!])$
46: $[\![pp]\!] \leftarrow [\![pp]\!] \oplus [\![tmp_2]\!]$
47: $[\![d]\!] \leftarrow \texttt{refresh}([\![d]\!])$
48: $[\![d_p]\!] \leftarrow \texttt{refresh}([\![d_p]\!])$
49: $[\![tmp]\!] \leftarrow [\![d]\!] \oplus [\![d_p]\!]$
50: $[\![mask_3]\!] \leftarrow \texttt{refresh}([\![mask_3]\!])$
51: $[\![tmp_2]\!] \leftarrow \texttt{sec}_{\&}([\![tmp]\!], [\![mask_3]\!])$
52: $[\![d_p]\!] \leftarrow \texttt{refresh}([\![d_p]\!])$
53: $[\![d_p]\!] \leftarrow [\![d_p]\!] \oplus [\![tmp_2]\!]$
54: **for** $j \leftarrow \delta$ to 1 **do**
55: $[\![mask_3]\!] \leftarrow \texttt{refresh}([\![mask_3]\!])$
56: $[\![tmp]\!] \leftarrow \texttt{sec}_{\&}([\![mask_3]\!], [\![\sigma_{copy}]\!]_{j-1})$
57: $[\![notmask_3]\!] \leftarrow \neg[\![mask_3]\!]$
58: $[\![X\sigma p]\!]_{j-1} \leftarrow \texttt{refresh}([\![X\sigma p]\!]_{j-1})$
59: $[\![tmp_2]\!] \leftarrow \texttt{sec}_{\&}([\![X\sigma p]\!]_{j-1}, [\![notmask_3]\!])$
60: $[\![X\sigma p]\!]_j \leftarrow [\![tmp]\!] \oplus [\![tmp_2]\!]$
61: $[\![tmp]\!] \leftarrow \texttt{refresh}([\![tmp]\!])$
62: **end for**

63: $[\![deg_{\sigma p}]\!] \leftarrow \texttt{refresh}([\![deg_{\sigma p}]\!])$
64: $[\![tmp]\!] \leftarrow [\![deg_{\sigma_{copy}}]\!] \oplus [\![deg_{\sigma p}]\!]$
65: $[\![mask_3]\!] \leftarrow \texttt{refresh}([\![mask_3]\!])$
66: $[\![tmp_2]\!] \leftarrow \texttt{sec}_{\&}([\![mask_3]\!], [\![tmp]\!])$
67: $[\![deg_{\sigma p}]\!] \leftarrow \texttt{refresh}([\![deg_{\sigma p}]\!])$
68: $[\![deg_{\sigma p}]\!] \leftarrow [\![deg_{\sigma p}]\!] \oplus [\![tmp_2]\!]$
69: $[\![d]\!] \leftarrow [\![synd]\!]_{\mu+1}$
70: **for** $k \leftarrow 2$ to $\min(\mu+1, \delta)$ **do**
71: $[\![tmp]\!] \leftarrow sec_{GFMult}([\![\sigma]\!]_k, [\![synd]\!]_{\mu+1-k})$
72: $[\![\sigma]\!]_k \leftarrow \texttt{refresh}([\![\sigma]\!]_k)$
73: $[\![synd]\!]_{\mu+1-k} \leftarrow \texttt{refresh}([\![synd]\!]_{\mu+1-k})$
74: $[\![d]\!] \leftarrow [\![d]\!] \oplus [\![tmp]\!]$
75: **end for**
76: **end for**

Algorithm 37: secComputeErrorValues

Input: $[\![z]\!] \in \mathbb{F}_2^{16\times\delta+1}, [\![err]\!] \in \mathbb{F}_2^{16\times\delta}$
Output: $[\![ev]\!] \in \mathbb{F}_2^{16\times\delta}$
1: $[\![\beta]\!] \leftarrow [\,0, \ldots, 0\,]$ (length δ)
2: $[\![e]\!] \leftarrow [\,0, \ldots, 0\,]$ (length δ)
3: $[\![ff]\!] \leftarrow -1$
4: $[\![zero]\!] \leftarrow 0$
5: $[\![\delta_{count}]\!] \leftarrow 0$
6: **for** $i \leftarrow 1$ to n_1 **do**
7: $[\![found]\!] \leftarrow 0$
8: $[\![tmp]\!] \leftarrow \texttt{sec}_{=}([\![err]\!]_i, [\![zero]\!])$
9: $[\![zero]\!] \leftarrow \texttt{refresh}([\![zero]\!])$
10: $[\![mask_1]\!] \leftarrow \texttt{sec}_{\text{if}}([\![ff]\!], [\![zero]\!], [\![tmp]\!])$
11: **for** $j \leftarrow 1$ to δ **do**
12: $[\![jm]\!] \leftarrow j$
13: $[\![tmp]\!] \leftarrow \texttt{sec}_{=}([\![jm]\!], [\![\delta_{count}]\!])$
14: $[\![zero]\!] \leftarrow \texttt{refresh}([\![zero]\!])$
15: $[\![ff]\!] \leftarrow \texttt{refresh}([\![ff]\!])$
16: $[\![\delta_{count}]\!] \leftarrow \texttt{refresh}([\![\delta_{count}]\!])$
17: $[\![mask_2]\!] \leftarrow \texttt{sec}_{\text{if}}([\![zero]\!], [\![ff]\!], [\![tmp]\!])$
18: $[\![tmp]\!] \leftarrow [\![mask_2]\!]$ & gf_exp_i
19: $[\![tmp_2]\!] \leftarrow \texttt{sec}_{\&}([\![mask_1]\!], [\![tmp]\!])$
20: $[\![\beta]\!]_j \leftarrow \texttt{sec}_{+}([\![\beta]\!]_j, [\![tmp_2]\!])$
21: $[\![mask_2]\!] \leftarrow \texttt{refresh}([\![mask_2]\!])$
22: $[\![tmp]\!] \leftarrow [\![mask_2]\!]$ & 1
23: $[\![mask_1]\!] \leftarrow \texttt{refresh}([\![mask_1]\!])$
24: $[\![tmp_2]\!] \leftarrow \texttt{sec}_{\&}([\![mask_1]\!], [\![tmp]\!])$
25: $[\![found]\!] \leftarrow \texttt{sec}_{+}([\![found]\!], [\![tmp_2]\!])$
26: **end for**
27: $[\![\delta_{count}]\!] \leftarrow \texttt{sec}_{+}([\![\delta_{count}]\!], [\![found]\!])$
28: **end for**
29: $[\![\delta_{rv}]\!] \leftarrow [\![\delta_{count}]\!]$

30: **for** $i \leftarrow 1$ to δ **do**
31: $[\![tmp_3]\!] \leftarrow 1$
32: $[\![tmp_4]\!] \leftarrow 1$
33: $[\![inv]\!] \leftarrow sec_{GFInverse}([\![\beta]\!]_i)$
34: $[\![ipj]\!] \leftarrow 1$
35: **for** $j = 2$ to $\delta + 1$ **do**
36: $[\![tmp]\!] \leftarrow sec_{GFMult}([\![ipj]\!], [\![inv]\!])$
37: $[\![inv]\!] \leftarrow \texttt{refresh}([\![inv]\!])$
38: $[\![ipj]\!] \leftarrow [\![tmp]\!]$
39: $[\![tmp]\!] \leftarrow sec_{GFMult}([\![ipj]\!], [\![z]\!]_j)$
40: $[\![z]\!]_j \leftarrow \texttt{refresh}([\![z]\!]_j)$
41: $[\![tmp_3]\!] \leftarrow [\![tmp_3]\!] \oplus [\![tmp]\!]$
42: **end for**
43: **for** $k \leftarrow 2$ to δ **do**
44: $[\![tmp]\!] \leftarrow sec_{GFMult}([\![inv]\!], [\![\beta]\!]_{(i+k) \mod \delta})$
45: $[\![inv]\!] \leftarrow \texttt{refresh}(inv)$
46: $[\![\beta]\!]_{(i+k) \mod \delta} \leftarrow \texttt{refresh}([\![\beta]\!]_{(i+k) \mod \delta})$
47: $[\![tmp]\!] \leftarrow [\![tmp]\!] \oplus 1$
48: $[\![tmp_2]\!] \leftarrow sec_{GFMult}([\![tmp_4]\!], [\![tmp]\!])$
49: $[\![tmp_4]\!] \leftarrow [\![tmp_2]\!]$
50: **end for**
51: $[\![im]\!] \leftarrow i$
52: $[\![tmp]\!] \leftarrow \texttt{secMax}([\![\delta_{rv}]\!], [\![im]\!])$
53: $[\![\delta_{rv}]\!] \leftarrow \texttt{refresh}([\![\delta_{rv}]\!])$
54: $[\![tmp_2]\!] \leftarrow \texttt{sec}_{=}([\![tmp]\!], [\![\delta_{rv}]\!])$
55: $[\![\delta_{rv}]\!] \leftarrow \texttt{refresh}([\![\delta_{rv}]\!])$
56: $[\![zero]\!] \leftarrow \texttt{refresh}([\![zero]\!])$
57: $[\![ff]\!] \leftarrow \texttt{refresh}([\![ff]\!])$
58: $[\![mask_1]\!] \leftarrow \texttt{sec}_{\text{if}}([\![zero]\!], [\![ff]\!], [\![tmp_2]\!])$
59: $[\![tmp]\!] \leftarrow sec_{GFInverse}([\![tmp_4]\!])$
60: $[\![tmp_2]\!] \leftarrow sec_{GFMult}([\![tmp_3]\!], [\![tmp]\!])$
61: $[\![e]\!]_i \leftarrow \texttt{sec}_{\&}([\![mask_1]\!], [\![mask_2]\!])$
62: $[\![\delta_{count}]\!] \leftarrow 0$
63: **end for**
64: **for** $i \leftarrow 1$ to n_1 **do**
65: $[\![found]\!] \leftarrow 0$
66: $[\![zero]\!] \leftarrow \texttt{refresh}([\![zero]\!])$
67: $[\![[\![err]\!]_i]\!] \leftarrow \texttt{refresh}([\![err]\!]_i)$
68: $[\![tmp]\!] \leftarrow \texttt{sec}_{=}([\![err]\!]_i, [\![zero]\!])$
69: $[\![zero]\!] \leftarrow \texttt{refresh}([\![zero]\!])$
70: $[\![mask_1]\!] \leftarrow \texttt{sec}_{\text{if}}([\![ff]\!], [\![zero]\!], [\![tmp]\!])$
71: **for** $j \leftarrow 1$ to δ **do**
72: $[\![jm]\!] \leftarrow j$
73: $[\![tmp]\!] \leftarrow \texttt{sec}_{=}([\![jm]\!], [\![\delta_{count}]\!])$
74: $[\![zero]\!] \leftarrow \texttt{refresh}([\![zero]\!])$
75: $[\![ff]\!] \leftarrow \texttt{refresh}([\![ff]\!])$
76: $[\![\delta_{count}]\!] \leftarrow \texttt{refresh}([\![\delta_{count}]\!])$
77: $[\![mask_2]\!] \leftarrow \texttt{sec}_{\text{if}}([\![zero]\!], [\![ff]\!], [\![tmp]\!])$

78: $[\![tmp]\!] \leftarrow \texttt{sec}_{\&}([\![mask_2]\!], [\![e]\!]_j)$
79: $[\![e]\!]_j \leftarrow \texttt{refresh}([\![e]\!]_j)$
80: $[\![tmp_2]\!] \leftarrow \texttt{sec}_{\&}([\![mask_1]\!], [\![tmp]\!])$
81: $[\![ev]\!]_i \leftarrow \texttt{sec}_{+}([\![ev]\!]_i, [\![tmp_2]\!])$
82: $[\![tmp]\!] \leftarrow [\![mask_2]\!]$ & 1
83: $[\![mask_1]\!] \leftarrow \texttt{refresh}([\![mask_1]\!])$
84: $[\![tmp_2]\!] \leftarrow \texttt{sec}_{\&}([\![mask_1]\!], [\![tmp]\!])$
85: $[\![found]\!] \leftarrow \texttt{sec}_{+}([\![found]\!], [\![tmp_2]\!])$
86: **end for**
87: $[\![\delta_{count}]\!] \leftarrow \texttt{sec}_{+}([\![\delta_{count}]\!], [\![found]\!])$
88: **end for**

Algorithm 38 `secRSCorrectErrors`()

Input: $[\![cdw]\!] \in \mathbb{F}_2^{8\times n_1}, [\![err]\!] \in \mathbb{F}_2^{16\times\delta}$
Output: $[\![res]\!] \in \mathbb{F}_2^{8\times n_1}$
1: **for** $i \leftarrow 1$ to n_1 **do**
2: $[\![res]\!]_i \leftarrow [\![cdw]\!]_i \oplus [\![err]\!]_i$
3: **end for**

Algorithm 39 `secRSDecode`()

Input: $[\![cdw]\!] \in \mathbb{F}_2^{8\times n_1}$
Output: $[\![msg]\!] \in \mathbb{F}_2^{128}$
1: $[\![synd]\!] \leftarrow$ `secRSComputeSyndromes`$([\![cdw]\!])$
2: $[\![sigma]\!], [\![deg]\!] \leftarrow$ `secRSComputeELP`$([\![synd]\!])$
3: $[\![err]\!] \leftarrow$ `secRSComputeRoots`$([\![sigma]\!])$
4: $[\![sigma]\!] \leftarrow$ `refresh`$([\![sigma]\!])$
5: $[\![z]\!] \leftarrow$ `secRSComputeZPoly`$([\![sigma]\!], [\![deg]\!], [\![synd]\!])$
6: $[\![ev]\!] \leftarrow$ `secRSComputeErrorValues`$([\![z]\!], [\![err]\!])$
7: $[\![cdw]\!] \leftarrow$ `secRSCorrectErrors`$([\![cdw]\!], [\![ev]\!])$
8: **for** $i \leftarrow 1$ to k **do**
9: $[\![msg]\!]_i \leftarrow [\![cdw]\!]_{i+G-1}$
10: **end for**
11: $[\![msg]\!] \leftarrow$ `refresh`$([\![msg]\!])$

B.5 FFT

Algorithm 40 `computeFFTbetas`()

Output: β the array of basis used in the additive FFT
1: **for** $i \leftarrow 1$ to 7 **do**
2: $\beta_i \leftarrow 2^{7-i}$
3: **end for**

Algorithm 41 `computeSubsetSums`()

Input: $set, size$
Output: $sums$
1: $sums_1 \leftarrow 0$
2: **for** $i \leftarrow 1$ to $size$ **do**
3: **for** $j \leftarrow 1$ to 2^i **do**
4: $sums_{2^i+j} \leftarrow set_i \oplus sums_j$
5: **end for**
6: **end for**

Algorithm 42: secFFTrec

Input: $[\![f]\!], f_coeffs, m$number of betas, mf number of coefficients of f, β
Output: $[\![w]\!]$
1: $[\![g]\!] \leftarrow [\,0, \ldots, 0\,]$ (length 4)
2: $[\![h]\!] \leftarrow [\,0, \ldots, 0\,]$ (length 4)
3: $[\![u]\!] \leftarrow [\,0, \ldots, 0\,]$ (length 64)
4: $[\![v]\!] \leftarrow [\,0, \ldots, 0\,]$ (length 64)
5: **if** $mf == 1$ **then**
6: **for** $i \leftarrow 0$ to m **do**
7: $[\![tmp]\!]_i \leftarrow$ `GF_mult`$(\beta_i, [\![f]\!]_2)$
8: **end for**
9: $[\![w]\!]_1 \leftarrow [\![f]\!]_1$
10: $x \leftarrow 1$
11: **for** $j \leftarrow 1$ to m **do**
12: **for** $k \leftarrow 1$ to x **do**
13: $[\![w]\!]_{x+k} \leftarrow [\![w]\!]_k \oplus [\![tmp]\!]_j$
14: $[\![w]\!]_k \leftarrow$ `refresh`$([\![w]\!]_k)$
15: **end for**
16: $x \leftarrow 2^x$
17: **end for**
18: **end if**
19: **if** $\beta_{m-1} \neq 1$ **then**
20: $\beta mp \leftarrow 1$
21: $x \leftarrow 2^{mf}$
22: **for** $i \leftarrow 2$ to x **do**
23: $\beta mp \leftarrow$ `GF_mult`$(\beta mp, \beta_{m-1})$
24: $[\![f]\!]_i \leftarrow$ `GF_mult`$(\beta mp, [\![f]\!]_i)$
25: **end for**
26: **end if**
27: $[\![g]\!], [\![h]\!] \leftarrow sec_{radix}([\![f]\!], mf)$
28: **for** $i \leftarrow 1$ to m **do**
29: $\gamma_i \leftarrow$ `GF_mult`$(\beta_i,$ `GF_inv`$(\beta_{m-1}))$
30: $\delta_i \leftarrow$ `GF_square`$(\gamma_i) \oplus \gamma_i$
31: **end for**
32: $sums \leftarrow$ `computeSubsetSums`$(\gamma, m-1)$
33: $[\![v]\!] \leftarrow$ `secFFTrec`$([\![g]\!], (f\_coeffs+1)/2, m-1, mf-1, \delta)$
34: $k \leftarrow 2^{m-1}$

35: **if** $f\_coeffs \leq 3$ **then**
36: $[\![w]\!]_1 \leftarrow [\![u]\!]_1$
37: $[\![u]\!]_1 \leftarrow \texttt{refresh}([\![u]\!]_1)$
38: $[\![w]\!]_k \leftarrow [\![u]\!]_1 \oplus [\![h]\!]_1$
39: **for** $i \leftarrow 2$ to k **do**
40: $[\![u]\!]_i \leftarrow \texttt{refresh}([\![u]\!]_i)$
41: $[\![w]\!]_i \leftarrow [\![u]\!]_i \oplus \texttt{GF\_mult}(sums_i, [\![h]\!]_1)$
42: $[\![h]\!]_1 \leftarrow \texttt{refresh}([\![h]\!]_1)$
43: $[\![w]\!]_{k+i} \leftarrow [\![w]\!]_i \oplus [\![h]\!]_1$
44: **end for**
45: **end if**
46: **if** $f\_coeffs > 3$ **then**
47: $[\![v]\!] \leftarrow \texttt{secFFTrec}([\![h]\!], f\_coeffs/2, m-1, mf-1, \delta)$
48: $[\![w]\!]_{k,\ldots,3\times k} \leftarrow [\![v]\!]_{1,\ldots,2\times k}$
49: $[\![w]\!]_1 \leftarrow [\![u]\!]_1$
50: $[\![w]\!]_1 \leftarrow \texttt{refresh}([\![w]\!]_1)$
51: $[\![w]\!]_k \leftarrow [\![w]\!]_k \oplus [\![u]\!]_1$
52: **for** $i \leftarrow 2$ to k **do**
53: $[\![w]\!]_i \leftarrow \texttt{GF\_mult}(sums_i, [\![v]\!]_i)$
54: $[\![w]\!]_i \leftarrow \texttt{refresh}([\![w]\!]_i)$
55: $[\![w]\!]_{k+i} \leftarrow [\![w]\!]_{k+i} \oplus [\![w]\!]_i$
56: **end for**
57: **end if**

Algorithm 43 `secRadixBig`()

Input: $[\![f]\!], mf$
Output: $[\![g]\!], [\![h]\!]$

1: $[\![P]\!] \leftarrow [\,0, \ldots, 0\,]$ (length 8)
2: $[\![Q]\!] \leftarrow [\,0, \ldots, 0\,]$ (length 8)
3: $[\![P0]\!] \leftarrow [\,0, \ldots, 0\,]$ (length 4)
4: $[\![Q0]\!] \leftarrow [\,0, \ldots, 0\,]$ (length 4)
5: $[\![P1]\!] \leftarrow [\,0, \ldots, 0\,]$ (length 4)
6: $[\![Q1]\!] \leftarrow [\,0, \ldots, 0\,]$ (length 4)
7: $n \leftarrow 2^{mf-2}$
8: $[\![Q]\!]_{1,\ldots,2\times n} \leftarrow [\![f]\!]_{3\times n,\ldots,5\times n}$
9: $[\![Q]\!]_{n,\ldots,3\times n} \leftarrow [\![f]\!]_{3\times n,\ldots,5\times n}$
10: $[\![P]\!]_{1,\ldots,4\times n} \leftarrow [\![f]\!]_{1,\ldots,4\times n}$
11: **for** $i \leftarrow 1$ to n **do**
12: $[\![Q]\!]_i \leftarrow [\![Q]\!]_i \oplus [\![f]\!]_{2\times n+i}$
13: $[\![P]\!]_{n+i} \leftarrow [\![P]\!]_{n+i} \oplus [\![Q]\!]_i$
14: **end for**
15: $[\![Q0]\!], [\![Q1]\!] \leftarrow sec_{radix}([\![Q]\!], mf-1)$
16: $[\![P0]\!], [\![P1]\!] \leftarrow sec_{radix}([\![P]\!], mf-1)$
17: $[\![g]\!]_{1,\ldots,2\times n} \leftarrow [\![P0]\!]_{1,\ldots,2\times n}$
18: $[\![h]\!]_{1,\ldots,2\times n} \leftarrow [\![P1]\!]_{1,\ldots,2\times n}$
19: $[\![g]\!]_{n,\ldots,3\times n} \leftarrow [\![Q0]\!]_{1,\ldots,2\times n}$
20: $[\![h]\!]_{n,\ldots,3\times n} \leftarrow [\![Q1]\!]_{1,\ldots,2\times n}$

Algorithm 44 `secFFTRetrieveErrorPoly()`

Input: $[\![w]\!]$
Output: $[\![err]\!]$

$\gamma \leftarrow$ `computeFFTbetas`()
$sums \leftarrow$ `computeSubsetSums`$(\gamma, 7)$
$k \leftarrow 128$
$[\![wso]\!] \leftarrow [\,0, \ldots, 0\,]$ (length 256)
for $i \leftarrow 1$ to $2 \times k$ **do**
    $[\![wso]\!]_i \leftarrow [\![w]\!]_i$
    $[\![wso]\!]_i \leftarrow \texttt{sec}_{\text{oppos}}([\![w]\!]_i)$
    $[\![wso]\!]_i \leftarrow [\![w]\!]_i \gg 15$
end for
$[\![err]\!]_1 \leftarrow [\![wso]\!]_1 \oplus [\![wso]\!]_k$
for $i \leftarrow 1$ to k **do**
    $idx1 \leftarrow 256 - GF\_log_{sums_i}$
    $idx2 \leftarrow 256 - GF\_log_{sums_i \oplus 1}$
    $[\![err]\!]_{idx1} \leftarrow [\![err]\!]_{idx1} \oplus 1$
    $[\![err]\!]_{idx2} \leftarrow [\![err]\!]_{idx2} \oplus 1$
    $[\![err]\!]_{idx1} \leftarrow [\![err]\!]_{idx1} \oplus [\![wso]\!]_i$
    $[\![err]\!]_{idx2} \leftarrow [\![err]\!]_{idx2} \oplus [\![wso]\!]_{i+k}$
end for

Algorithm 45 `secFFT()`

Input: $[\![f]\!], f\_coeffs$
Output: $[\![w]\!]$

$[\![g]\!] \leftarrow [\,0, \ldots, 0\,]$ (length 8)
$[\![h]\!] \leftarrow [\,0, \ldots, 0\,]$ (length 8)
$[\![u]\!] \leftarrow [\,0, \ldots, 0\,]$ (length 128)
$[\![v]\!] \leftarrow [\,0, \ldots, 0\,]$ (length 128)
$\beta \leftarrow$ `computeFFTbetas`()
$sums \leftarrow$ `computeSubsetSums`$(\beta, 7)$
$[\![g]\!], [\![h]\!] \leftarrow sec_{radix}([\![f]\!], 4)$
for $i \leftarrow 1$ to 7 **do**
    $\delta_i \leftarrow$ `GF_square`$(\beta_i) \oplus \beta_i$
end for
$[\![u]\!] \leftarrow$ `secFFTrec`$([\![g]\!], (f\_coeffs + 1)/2, 7, 3, \delta)$
$[\![v]\!] \leftarrow$ `secFFTrec`$([\![h]\!], f\_coeffs/2, 7, 3, \delta)$
$k \leftarrow 128$
$[\![w]\!]_{k,\ldots,3\times k} \leftarrow [\![v]\!]_{1,\ldots,2\times k}$
$[\![w]\!]_1 \leftarrow [\![u]\!]_1$
$[\![w]\!]_1 \leftarrow$ `refresh`$([\![w]\!]_1)$
$[\![w]\!]_k \leftarrow [\![w]\!]_k \oplus [\![u]\!]_1$
for $i \leftarrow 2$ to k **do**
    $[\![u]\!]_i \leftarrow$ `refresh`$([\![u]\!]_i)$
    $[\![w]\!]_i \leftarrow [\![u]\!]_i \oplus$ `GF_mult`$(sums_i, [\![v]\!]_i)$
    $[\![w]\!]_{k+i} \leftarrow [\![w]\!]_{k+i} \oplus [\![w]\!]_i$
end for

Algorithm 46: secRadix

Input: $[\![f]\!], m_f$
Output: $[\![g]\!], [\![h]\!]$

$[\![f]\!] \leftarrow \texttt{refresh}([\![f]\!])$
if $m_f == 4$ **then**
  $[\![g]\!]_5 \leftarrow [\![f]\!]_9 \oplus [\![f]\!]_{13}$
  $[\![f]\!]_{13} \leftarrow \texttt{refresh}([\![f]\!]_{13})$
  $[\![g]\!]_7 \leftarrow [\![f]\!]_{13} \oplus [\![f]\!]_{15}$
  $[\![f]\!]_{15} \leftarrow \texttt{refresh}([\![f]\!]_{15})$
  $[\![h]\!]_6 \leftarrow [\![f]\!]_{12} \oplus [\![f]\!]_{14}$
  $[\![f]\!]_{14} \leftarrow \texttt{refresh}([\![f]\!]_{14})$; $[\![f]\!]_{15} \leftarrow \texttt{refresh}([\![f]\!]_{15})$
  $[\![h]\!]_7 \leftarrow [\![f]\!]_{14} \oplus [\![f]\!]_{15}$
  $[\![h]\!]_8 \leftarrow [\![f]\!]_{16}$
  $[\![h]\!]_8 \leftarrow \texttt{refresh}([\![h]\!]_8)$; $[\![f]\!]_{13} \leftarrow \texttt{refresh}([\![f]\!]_{13})$
  $[\![g]\!]_6 \leftarrow [\![f]\!]_{11} \oplus [\![f]\!]_{13} \oplus [\![h]\!]_{16}$
  $[\![f]\!]_{14} \leftarrow \texttt{refresh}([\![f]\!]_{14})$
  $[\![g]\!]_5 \leftarrow [\![f]\!]_{10} \oplus [\![f]\!]_{14} \oplus [\![g]\!]_6$
  $[\![g]\!]_1 \leftarrow [\![f]\!]_1$
  $[\![f]\!]_{12} \leftarrow \texttt{refresh}([\![f]\!]_{12})$;$[\![f]\!]_{16} \leftarrow \texttt{refresh}([\![f]\!]_{16})$
  $[\![h]\!]_4 \leftarrow [\![f]\!]_8 \oplus [\![f]\!]_{12} \oplus [\![f]\!]_{16}$
  $[\![f]\!]_{11} \leftarrow \texttt{refresh}([\![f]\!]_{11})$; $[\![f]\!]_{15} \leftarrow \texttt{refresh}([\![f]\!]_{15})$
  $[\![g]\!]_4 \leftarrow [\![f]\!]_7 \oplus [\![f]\!]_{11} \oplus [\![f]\!]_{15} \oplus [\![h]\!]_4$
  $[\![h]\!]_4 \leftarrow \texttt{refresh}([\![h]\!]_4)$
  $[\![g]\!]_3 \leftarrow [\![f]\!]_5 \oplus [\![g]\!]_5 \oplus [\![g]\!]_4 \oplus [\![h]\!]_4$
  $[\![f]\!]_{10} \leftarrow \texttt{refresh}([\![f]\!]_{10})$;$[\![f]\!]_{14} \leftarrow \texttt{refresh}([\![f]\!]_{14})$; $[\![h]\!]_4 \leftarrow \texttt{refresh}([\![h]\!]_4)$
  $[\![h]\!]_2 \leftarrow [\![f]\!]_4 \oplus [\![f]\!]_6 \oplus [\![f]\!]_{10} \oplus [\![f]\!]_{14} \oplus [\![h]\!]_4$
  $[\![f]\!]_4 \leftarrow \texttt{refresh}([\![f]\!]_4)$;$[\![g]\!]_4 \leftarrow \texttt{refresh}([\![g]\!]_4)$
  $[\![h]\!]_3 \leftarrow [\![f]\!]_4 \oplus [\![h]\!]_2 \oplus [\![g]\!]_4$
  $[\![h]\!]_2 \leftarrow \texttt{refresh}([\![h]\!]_2)$
  $[\![g]\!]_2 \leftarrow [\![f]\!]_3 \oplus [\![g]\!]_3 \oplus [\![h]\!]_2$
  $[\![h]\!]_1 \leftarrow [\![f]\!]_2 \oplus [\![g]\!]_2$
end if
if $m_f == 3$ **then**
  $[\![g]\!]_1 \leftarrow [\![f]\!]_1$
  $[\![g]\!]_3 \leftarrow [\![f]\!]_5 \oplus [\![f]\!]_7$
  $[\![f]\!]_7 \leftarrow \texttt{refresh}([\![f]\!]_7)$
  $[\![g]\!]_4 \leftarrow [\![f]\!]_7 \oplus [\![f]\!]_8$
  $[\![f]\!]_8 \leftarrow \texttt{refresh}([\![f]\!]_8)$
  $[\![h]\!]_2 \leftarrow [\![f]\!]_4 \oplus [\![f]\!]_6 \oplus [\![f]\!]_8$
  $[\![f]\!]_6 \leftarrow \texttt{refresh}([\![f]\!]_6)$;$[\![f]\!]_7 \leftarrow \texttt{refresh}([\![f]\!]_7)$
  $[\![h]\!]_3 \leftarrow [\![f]\!]_6 \oplus [\![f]\!]_7$
  $[\![g]\!]_4 \leftarrow [\![f]\!]_8$
  $[\![g]\!]_4 \leftarrow \texttt{refresh}([\![g]\!]_4)$
  $[\![g]\!]_2 \leftarrow [\![f]\!]_3 \oplus [\![g]\!]_3 \oplus [\![h]\!]_2$
  $[\![h]\!]_1 \leftarrow [\![f]\!]_2 \oplus [\![g]\!]_2$
end if
if $m_f == 2$ **then**
  $[\![g]\!]_1 \leftarrow [\![f]\!]_1$

46: $[\![g]\!]_2 \leftarrow [\![f]\!]_3 \oplus [\![f]\!]_4$
47: $[\![h]\!]_1 \leftarrow [\![f]\!]_2 \oplus [\![g]\!]_2$
48: $[\![h]\!]_2 \leftarrow [\![f]\!]_4$
49: **end if**
50: **if** $m_f == 1$ **then**
51: $[\![g]\!]_1 \leftarrow [\![f]\!]_1$
52: $[\![h]\!]_1 \leftarrow [\![f]\!]_2$
53: **end if**
54: **if** $m_f > 4$ **then**
55: `secRadixBig`$([\![g]\!], [\![h]\!], [\![f]\!], m_f)$
56: **end if**

B.6 Vector Multiplication

Algorithm 47 Masked Karatsuba add 1

Input: $[\![a]\!], [\![b]\!], size_h, size_l$
Output: $[\![alh]\!], [\![blh]\!]$
1: **for** $i \leftarrow 1$ to $size_h$ **do**
2: $[\![alh_i]\!] \leftarrow [\![a_i]\!] \oplus [\![a_{i+size_l}]\!]$
3: $[\![blh_i]\!] \leftarrow [\![b_i]\!] \oplus [\![b_{i+size_l}]\!]$
4: **end for**
5: **if** $size_h < size_l$ **then**
6: $[\![alh_{size_h}]\!] \leftarrow [\![a_{size_h}]\!]$
7: $[\![blh_{size_h}]\!] \leftarrow [\![b_{size_h}]\!]$
8: **end if**

Algorithm 48 Masked Karatsuba add 2

Input: $[\![f]\!], [\![g]\!], size_h, size_l$
Output: $[\![out]\!]$
1: **for** $i \leftarrow 1$ to $2 \times size_l$ **do**
2: $[\![f_i]\!] \leftarrow [\![f_i]\!] \oplus [\![out_i]\!]$
3: **end for**
4: **for** $i \leftarrow 1$ to $2 \times size_h$ **do**
5: $[\![f_i]\!] \leftarrow [\![f_i]\!] \oplus [\![g_i]\!]$
6: **end for**
7: **for** $i \leftarrow 1$ to $2 \times size_l$ **do**
8: $[\![out_i]\!] \leftarrow [\![out_i]\!] \oplus [\![f_i]\!]$
9: **end for**

Algorithm 49 Masked Karatsuba

Input: $[\![a]\!], [\![b]\!], [\![stack]\!], size$
Output: $[\![o]\!]$

if $size = 1$ **then**
  $[\![o]\!] \leftarrow$ `myMaskedMult`$([\![a_0]\!], [\![b_0]\!])$
  return
end if
$size_h \leftarrow size/2$
$size_l \leftarrow (size + 1)/2$
$[\![alh]\!] \leftarrow$ (pointer to)$[\![stack_0]\!]$
$[\![blh]\!] \leftarrow$ (pointer to)$[\![alh_{size_l}]\!]$
$[\![f]\!] \leftarrow$ (pointer to)$[\![blh_{size_l}]\!]$
$[\![g]\!] \leftarrow$ (pointer to)$[\![o_{size_l \times 2}]\!]$
$[\![newstack]\!] \leftarrow$ (pointer to)$[\![stack_{size_l \times 4}]\!]$
$[\![ah]\!] \leftarrow$ (pointer to)$[\![a_{size_l}]\!]$
$[\![bh]\!] \leftarrow$ (pointer to)$[\![b_{size_l}]\!]$
$[\![o]\!] \leftarrow$ `secKaratsuba`$([\![a]\!], [\![b]\!], size_l, [\![newstack]\!])$
$[\![newstack]\!] \leftarrow$ `refresh`$([\![newstack]\!])$
$[\![g]\!] \leftarrow$ `secKaratsuba`$([\![ah]\!], [\![bh]\!], size_h, [\![newstack]\!])$
$[\![a]\!] \leftarrow$ `refresh`$([\![a]\!])$
$[\![b]\!] \leftarrow$ `refresh`$([\![b]\!])$
$[\![newstack]\!] \leftarrow$ `refresh`$([\![newstack]\!])$
$[\![alh]\!], [\![blh]\!] \leftarrow$ `secKaratsubaAdd1`$([\![a]\!], [\![b]\!], size_l, size_h)$
$[\![f]\!] \leftarrow$ `secKaratsuba`$([\![alh]\!], [\![blh]\!], size_l, [\![newstack]\!])$
$[\![o]\!] \leftarrow$ `secKaratsubaAdd2`$([\![a]\!], [\![b]\!], size_l, size_h)$

Algorithm 50 reduce

Input: $[\![x]\!]$
Output: $[\![out]\!]$

$s \leftarrow n\&63$ ▷ n: see Table 2.2
$n_{64} \leftarrow n/64$
$mask \leftarrow n \mod 64$
$[\![out]\!] \leftarrow [\![x_{1,\ldots,n_{64}}]\!]$
for $i \leftarrow 1$ to n_{64} **do**
  $[\![out_i]\!] \leftarrow [\![out_i]\!] \oplus ([\![x_{i-1+n_{64}}]\!] \gg s)$
  $[\![out_i]\!] \leftarrow [\![out_i]\!] \oplus ([\![x_{i+n_{64}}]\!] \ll (64 - s)))$
end for
$[\![out_{n_{64}}]\!] \leftarrow [\![out_{n_{64}}]\!] \& (2^{mask} - 1)$

Algorithm 51 $\texttt{sec}_{\text{vect\_mul}}$

Input: $[\![u]\!] \in \mathbb{F}_2^n, [\![v]\!] \in \mathbb{F}_2^n$
Output: $[\![o]\!] \in \mathbb{F}_2^n$

1: $n_{64} \leftarrow n/64$
2: $stack \leftarrow [\,0, \ldots, 0\,]$ (length $n_{64} \times 8$)
3: $[\![o_{karat}]\!] \leftarrow [\,0, \ldots, 0\,]$ (length $n_{64} \times 2$)
4: $[\![o_{karat}]\!] \leftarrow \texttt{secKaratsuba}([\![u]\!], [\![v]\!], n_{64}, [\![stack]\!])$
5: $o \leftarrow \texttt{reduce}([\![o_{karat}]\!])$

ECOLE
DOCTORALE

Titre : Implémentation sécurisée pour la cryptographie post-quantique

Mots clés : Implémentation sécurisée, Cryptographie post-quantique, Attaques par canaux auxiliaires, Masquage, HQC

Résumé : L'objectif de la thèse est d'améliorer la sécurité logicielle de composants mettant en œuvre la cryptographie post-quantique. Plus précieusement, il s'agira d'identifier des vulnérabilises vis-à-vis des attaques par canal auxiliaire puis les corriger.

Title : Secure implementation for post-quantum cryptography

Keywords : Secure implementation, Post-quantum cryptography, Side-channel attacks, Masking, HQC

Abstract : The goal of the thesis is to improve the software security of components implementing post-quantum cryptography. More specifically, the aim is to identify and correct vulnerabilities to side-channel attacks.

Institut Polytechnique de Paris
91120 Palaiseau, France