POLITECNICO
MILANO 1863

POLITECNICO DI MILANO
DEPARTMENT OF ELECTRONICS, INFORMATION AND BIOENGINEERING
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

HARDWARE DESIGN AND IMPLEMENTATION OF POST-QUANTUM CRYPTOGRAPHIC ALGORITHMS: THE CASE OF NTRU, HQC AND CROSS

Doctoral Dissertation of:
Francesco Antognazza

Supervisor:
Prof. Gerardo Pelosi
Prof. Alessandro Barenghi

Tutor:
Prof. Cristina Silvano

The Chair of the Doctoral Program:
Prof. Luigi Piroddi

Year 2025 – XXXVI cycle

Abstract

DURING the last decade, public key cryptography received renewed attention from academic, industrial and institutional stakeholders due to the consistent advancements in the development of more capable quantum computers, which are deemed able to break in the near future the security guarantees provided by the currently deployed cryptographic algorithms based on the hardness of the integer factorization or discrete logarithm problems (e.g., RSA and ECC) thanks to Shor's algorithm. In 2016, the U.S.A. National Institute of Standards and Technology published a call for public key schemes resistant to quantum-computer-aided attacks, intending to replace the vulnerable algorithms currently in use and asking the community for cryptanalysis and optimized software and hardware implementations of the proposals. After several rounds of selection, schemes based on lattices and forward error-correction codes proved to be the most promising. Taking a hardware design perspective, I focused on analyzing the key encapsulation mechanism based on the Hoffstein, Pipher, and Silverman's NTRUEncrypt encryption scheme, commonly referred to as NTRU, the key encapsulation mechanism named "Hamming Quasi-Cyclic" (HQC), and the digital signature algorithm named "Codes and Restricted Objects Signature Scheme" (CROSS), with a particular emphasis on the optimization of the latency and the efficiency of their building blocks such as polynomial multipliers, random polynomial samplers and encoders/decoders for the Reed-Muller/Reed-Solomon concatenated code, ideating and comparing novel specialized algorithms for the sub-procedures composing the cryptoschemes and providing results for both FPGA and ASIC design flows. The results show a substantially improved latency and efficiency compared to the state-of-the-art, and a proposed algorithmic optimization for HQC was recently included in the official specification.

Sommario

DURANTE l'ultimo decennio, la crittografia a chiave pubblica ha ricevuto una rinnovata attenzione da parte di accademia, industria, e parti istituzionali interessate a causa dei continui progressi nello sviluppo di computer quantistici, i quali sono ritenuti in grado di violare nel prossimo futuro le garanzie di sicurezza fornite dagli algoritmi a chiave pubblica attualmente utilizzati che sono basati sulla difficoltà della fattorizzazione di numeri interi estremamente grandi, o del logaritmo discreto (ad esempio, RSA ed ECC) grazie all'algoritmo di Shor. Nel 2016, il National Institute of Standards and Technology degli Stati Uniti d'America ha pubblicato un bando per nuovi schemi a chiave pubblica resistenti ad attacchi per mezzo di computer quantistici con l'intento di sostituire gli algoritmi vulnerabili attualmente in uso, e al chiedendo alla comunità di effettuare crittoanalisi e implementazioni software e hardware ottimizzate delle proposte. Dopo diversi round di selezione, gli schemi basati su reticoli e codici a correzione degli errori si sono dimostrati i più promettenti. Considerando lo sviluppo di sistemi digitali, la mia ricerca si focalizza sull'analisi del meccanismo di generazione delle chiavi di sessione basato sull'algoritmo NTRUEncrypt di Hoffstein, Pipher, e Silverman, comunemente chiamato NTRU, il meccanismo di condivisione "Hamming Quasi-Cyclic" (HQC) e dell'algoritmo di firma digitale "Codes and Restricted Objects Signature Scheme" (CROSS), ponendo particolare enfasi sull'ottimizzazione della latenza e dell'efficienza dei componenti chiave come moltiplicatori polinomiali, generatori di polinomi casuali e codificatori/decodificatori per il codice concatenato Reed-Muller/Reed-Solomon, ideando e confrontando nuovi algoritmi specializzati per le sotto-procedure che compongono i crittosistemi, e analizzando i risultato con flussi di progettazione per FPGA e ASIC. I risultati ottenuti mostrano una latenza e un'efficienza sostanzialmente migliorate rispetto ai risultati precedenti, e un'ottimizzazione algoritmica proposta per HQC è stata recentemente inclusa nella specifica ufficiale.

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

1.1 Brief overview on Quantum Computers

In recent times, research efforts pioneered by companies such as IBM, Google, Rigetti, and Intel in the field of development of quantum computers with high qubit count and improved reliability have spurred researchers' interests from many different fields such as chemistry, material science, and cryptography.

On December 4, 2023 IBM unveiled [Cas23] a 1121 qubits Quantum Processor Unit (QPU), codename Condor, surpassing the thousand qubit integrated on a single QPU mark, confirming the exponentially-growing qubits density provided by their underlying superconducting technology in the last decade. On the same day, IBM also presented Heron, a 133 qubit QPU able to improve coherency time allowing the implementation of circuits with up to 5k quantum gates. Intel, on the other end, recently switched to a promising silicon spin qubit quantum computer [Ney+24]. Leveraging its expertise in semiconductor technology and their advanced manufacturing plants, Intel could produce such a technology with an extremely high qubit density (100 nm pitch between qubits) and in high volume.

Nonetheless, quite a large number of limitations afflicts current technology, delaying the a practical showcase of a 'quantum advantage', also known as 'quantum supremacy'. Ideal qubits are isolated from the external environment, and maintain coherency indefinitely, but real qubits are far from the ideal ones. Coherency time, in the range of few tens or hundreds of μs, heavily limits the size of circuits implementable. Similarly,

noise-inducted errors coming due to the undesirable interaction with the environment, require the introduction of Error-Correcting Codes (ECCs) that, along with technology-specific constraints, impose huge overheads. Finally, there is no clear solution enabling scalable Quantum Random-Access Memory (QRAM) key technology [JR23] in the near future, severely limiting the capabilities of current quantum computers.

An important step towards the implementation of large-scale fault-tolerant quantum algorithms was achieved in 2024 by [Ach+24]. In their work they established that it is now possible to profitably apply ECC to quantum algorithms to generate reliable logical qubits starting from a larger number of physical qubits having a higher error rate. Furthermore, the superpolynomial speedup offered by Shor's algorithm [Sho94], compared to known classical algorithms, in the order finding problem is an exceptionally attractive motivation for the research community to restlessly pursue the development of quantum technologies. Moreover, the more broadly applicable Grover's algorithm [Gro96] for finding zeros in a function, which only provides quadratic speedup over the classical solution, is nonetheless another appealing reason.

1.2 Applications of Quantum Algorithms to Cryptography

Generally speaking, to apply Shor and Grover algorithms and gain some advantage over classical computers, an algorithm of interest must be reformulated and reduced to an order finding problem, or finding zeros in a function.

In the field of cryptography, a category of algorithms referred to as Public-Key Cryptography (PKC), or asymmetric cryptography, allow to ensure crucial attributes such as confidentiality, authenticity, and non-repudiation, all without the need to be sharing a single secret value among parties. This is made possible through the use of an underlying one-way function. Public-key cryptography is at the core of many network protocols, such as Transport Layer Security (TLS), Secure SHell (SSH), and Pretty Good Privacy (PGP). Furthermore, many other algorithms use this ubiquitous yet inconspicuous technology, guaranteeing many security aspects in our daily digital life, from shopping on online platforms to making electronic payments. As a result, public-key cryptography remains a fundamental pillar of modern digital security, seamlessly protecting our online interactions and transactions, often without our direct awareness.

One of the most commonly used asymmetric algorithm is Rivest-Shamir-Adleman (RSA), acronym from its inventors, who in 1977 designed an algorithm to either conceal a message or create its digital signature, starting from a pair of keys, one kept private by the owner and another made public and shared to anyone. RSA security, preventing an agent without the private key to decipher a message or forge signatures impersonating the owner of the keys, can be reduced to on the hardness of the Integer Factorization Problem (IFP): given a composite integer $N = p \cdot q$, where p and q are unknown sufficiently large primes, it is hard to determine p and q. A more efficient alternative to RSA is given by the algorithms belonging to the category of Elliptic Curve (EC), such as ECDSA, ECDH, and Ed25519. Those algorithms, by contrast, depend

on the hardness of solving the Elliptic Curve Discrete logarithm Problem (ECDLP): given a non-singular elliptic curve E defined over a field $\mathbb{F}$, a sufficiently large point G that generates a large cyclic subgroup in the additive group of the points of E, and a point P in such subgroup, it is hard to find an integer k such that $P = kG$

Both of these difficult problems are specific instances of the Hidden Subgroup Problem (HSP) for finite abelian groups. Given only a classical computer, for appropriate parameter choices, these problems require an impractical amount of computational resources to be solved, making it virtually impossible to break these one-way functions on conventional computers within a timeframe of a thousand years. Unfortunately, those hard problems can be reduced to an order finding problem, enabling the use of Shor's algorithm and thus be solved in exponentially faster (thus, in polynomial time) by an attacker having access to a quantum computer. Considering the current status of quantum computers, it is estimated that 20 million noisy qubits are required to break RSA 2048 in 8 hours [GE21], and similarly 13 million noisy qubits to break 256-bit ECDSA in 24 hours [Web+22]. A recent update of the document from the German Bundesamt für Sicherheit in der Informationstechnik (German Federal Office for Information Security) (BSI) [BSI25] reporting the status of the development of quantum computers stated that several technical obstacles in fault-tolerant quantum computing were recently solved, forecasting the availability of a cryptographically relevant quantum computer in 16 years.

Symmetric cryptography is a class of algorithms that provide guarantees of confidentiality more efficiently with respect to asymmetric cryptography, particularly for bulk encryption of data, both in case of communication between two agents on an unsecure channel and for data at rest encryption. The drawback is that the exchange of the shared secret encryption/decryption key between the communicating agents over an insecure communication channel still requires public key cryptography. Block ciphers belong to this symmetric cryptography class, with Advanced Encryption Standard (AES) being the most diffused example pervasively present in most of the devices. AES offers modes of operations, such as Galois Counter Mode (GCM), to additionally guarantee data integrity and authenticity, and is deemed immune to Shor-based cryptanalysis attacks. A recent work [SP24] extensively documented the impact of quantum computers on the security of AES, estimating the physical resources required by a Grover-based algorithm to break it in a reasonable amount of time, as symmetric cryptographic algorithms are not affected by Shor's algorithm.

Under very conservative assumptions, symmetric cryptography and Hash Message Authentication Code (HMAC) are deemed to remain secure against the application of Grover-like algorithms [ETS17], though with a roughly halved security margin since their underlying hard problems can be reduced to finding roots of a generic Boolean function. Doubling the key size is a practical and straightforward solution to secure future communications and data communications protected via symmetric encryption, or message authentication performed via HMACs. Conversely, PKC algorithms currently in use cannot simply adopt this strategy due to the more effective security reduction

offered by a Shor-based cryptanalysis attacks.

Therefore, one of the most pressing challenges introduced by the advancements in the hardware development of Quantum Computers is the design of new asymmetric cryptographic algorithms whose security is based on hard problems not reducible to those that quantum computers can solve exponentially faster than a classical computer, i.e. IFP, ECDLP. A paradigm shift towards Quantum-Safe Cryptography, sometimes also referred as Quantum-Resistant Cryptography or Post-Quantum Cryptography (PQC), is imperative. While numerous new proposals are being assessed, a critical effort is currently underway to validate their security properties to avoid catastrophic effect in case a post-quantum scheme is discovered to be insecure even against classical computers. In the immediate future, the best strategy consists in the use of hybrid schemes, which are combining a post-quantum cryptoscheme with an asymmetric scheme already in use, to have the proof that breaking their combination requires the ability of breaking both cryptographic schemes.

1.3 Post-Quantum Cryptography standardization

In December 2016, the National Institute of Standards and Technology (NIST), a US government agency promoting widely adopted adopted Federal Information Processing Standard (FIPS) such as *FIPS 197* [EBB23] for the AES and *FIPS 202* [PM15] for the Secure Hash Algorithm SHA-3, announced the Post-Quantum cryptography Standardization Process, a public call for Key Establishment Mechanisms (KEMs) and Digital Signatures (DSs) with security properties against both classical and quantum-computer-aided cryptanalysis and attacks. The submitted proposals should provide multiple security margins via parameters selection to align to the security strengths offered by existing NIST standards in symmetric cryptography, allowing a meaningful comparison of the algorithms and evaluating the security/performance trade-offs. Moreover, KEM candidates should also satisfy the INDistinguishability under adaptive Chosen Ciphertext Attack (IND-CCA2) property, providing a safer use in many contexts and protocols, as it is possible to generate a key once and use it many times. The working principle of a KEM is depicted in Figure 1.1, and a detailed overview of the security classification is reported in Table 1.1.

In November 2017, among the 82 candidate algorithms submitted, only 69 met the minimum acceptance criteria. In January 2019, after a first round of evaluation based on their security and performance, 26 candidates moved to the next evaluation phase [Ala+19]. The proposed schemes were roughly classified by their core characteristics in several categories: lattices, ECCs, multivariate equations, isogeny graphs, hash-based signatures. Lattice-based candidates are the most numerous and versatile group, the only one covering both KEM and DS categories, as depicted in Figure 1.2. The security of Lattice-based schemes rely on the hardness of the Shortest Vector Problem (SVP) or the Closest Vector Problem (CVP), challenges capturing the researchers' interest for the past 30 years. Code-based candidates belong to another promising group, and are

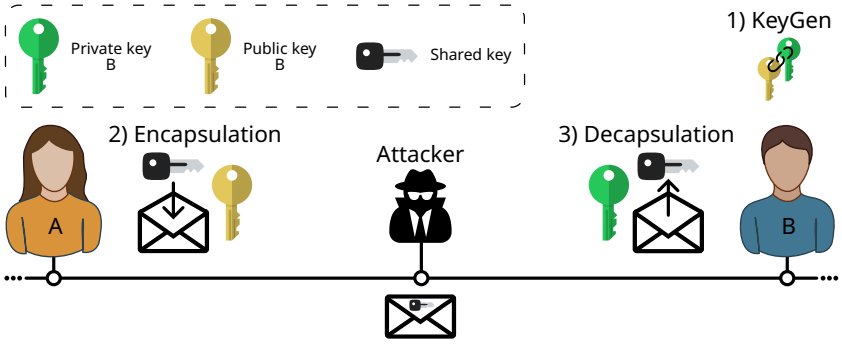


Figure 1.1: *Representation of the Key Establishment Mechanism of a shared session key between parties A and B over an unsecure channel wiretapped by an attacker. B initially generates a key pair, and transmits the public key to A. A then encrypts a fully random session key with B's public key obtaining the ciphertext containing the encapsulated key, and sends it to B. Only B, has the private key corresponding to the public key employed by A, is able to retrieve the shared session key.*

Table 1.1: *NIST security categories for PQC standardization process. Each security level defines the computational resources, measured as the number of classical boolean operations (gates) or the size of a quantum circuit, that are required to successful mount the described attack. δ is the quantum circuit depth, having realistic values ranging from 2^{40} (the number of quantum gates envisioned to be serially performed in a year), to 2^{64} (the number of boolean gates serially performed in a decade), and 2^{96} (the number of gates that atomic scale qubits with speed of light propagation times could perform in a millennium).*

Security level	Attack description	Estimated gates cost	
		quantum	classical
1	key search on a block cipher with a 128-bit key	$2^{157}/\delta$	2^{143}
2	collision search on a 256-bit hash function	–	2^{146}
3	key search on a block cipher with a 192-bit key	$2^{221}/\delta$	2^{207}
4	collision search on a 384-bit hash function	–	2^{210}
5	key search on a block cipher with a 256-bit key	$2^{285}/\delta$	2^{272}

primarily relying on the hardness of the Syndrome Decoding Problem (SDP) that also proved its importance in the past 45 years.

In July 2020, NIST announced [Moo+20] that Classic McEliece, CRYSTALS-Kyber, NTRU, and SABER were advancing to a third and final evaluation round in the KEM category, and that would elect one among them to be standardized. A similar procedure would involve CRYSTALS-DILITHIUM, FALCON, and Rainbow for the DS category. Moreover, as most of the finalists are relying on hard problems related to lattices, the BIKE, FrodoKEM, HQC, NTRU Prime, SIKE, GeMSS, Picnic, and SPHINCS+ algorithms would proceed on a parallel standardization branch as alternate candidates to promote variety in a portfolio of post-quantum algorithms in case of major security breakdowns. This choice proved wise, considering that in July 2022, a classic polynomial time cryptanalysis attack on SIKE [CD23] was revealed, leading to the retrieval of the secret keys in a few hours on a standard computer. In the same year

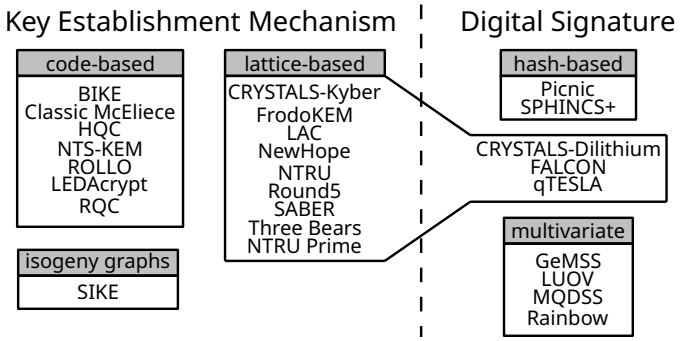


Figure 1.2: *Second round candidates in the NIST Post-Quantum Cryptography standardization process, categorized by their core characteristics. Among the* 26 *candidates,* 17 *of them are Key Establishment Mechanisms and* 9 *are Digital Signatures.*

Rainbow, a finalist in the DS category, was shown to be breakable by an exponential time, although fast enough attack [Beu22], able to recover the private key for the lowest security level using a laptop over a weekend.

The third selection round lasted two years, proving that a thorough evaluation of a new asymmetric algorithm is exceptionally challenging [Ala+22b]. In July 2022, NIST reported that CRYSTALS-Kyber would be standardized as the Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM) in *FIPS 203*, and that in the digital signatures category would proceed to standardize all the remaining proposals: CRYSTALS-Dilithium in *FIPS 204* under the name of Module-Lattice-Based Digital Signature Standard (ML-DSA), SPHINCS+ in *FIPS 205* under the name of Stateless Hash-Based Digital Signature Standard (SLH-DSA), and FALCON in *FIPS 206* under the name of Fast-Fourier Transform over NTRU-Lattice-Based Digital Signature Algorithm (FN-DSA). In the track for alternate candidates selection, three KEMs based on the hardness of the syndrome decoding problem BIKE [Ara+22], Classic McEliece [Ber+22], and HQC [Agu+24a] advanced to a final fourth round of selection. The PQC standardization was officially concluded in March 2025 when NIST announced in [Ala+25b] that only HQC was considered for the standardization, and expected to be finalized in 2027. Final specifications for *FIPS 203* [RL23c], *FIPS 204* [RL23b], and *FIPS 205* [RL23d] were released on August 13 2024, with the *FIPS 206* deferred later in the year.

Regarding the digital signatures, in October 2020 NIST issued the *SP 800-208* special publication [Coo+20] to supplement the *FIPS186* [RL23a] regulating the Digital Signature Standards, introducing two stateful hash-based signature schemes eXtended Merkle Signature Scheme (XMSS) and Leighton-Micali Signature (LMS), along with their multi-tree variants. Relying only on the security of cryptographic hash functions, both algorithms are confidently considered secure against both classical and quantum attacks, but they are not suitable for general use: an accidental reuse of a key-pair for a one-time signature on distinct messages enables an attacker to generate forgeries (valid signatures for other messages). Consequently, a state must be carefully maintained to

keep track of the one-time private keys that have been used, severely limiting the applicability to some specific applications, such as firmware code signing. By contrast, state-less hash-based signatures such as SLH-DSA are designed to be safely used as a drop-in replacement for already in use signature schemes, and are generally considered a more conservative choice in term of security compared to ML-DSA and FN-DSA, although having longer signatures and taking more time to generate a signature. To the end of determining other alternatives, in September 2022 NIST announced another public call for general-purpose Digital Signature specifically for schemes not based on structured lattices such as CRYSTALS-Dilithium and FALCON, although being open to lattice-based candidates significantly outperforming them in performance or security. In June 2023 NIST received 50 submissions, 40 of which were deemed compliant with the submission requirements, and after a first evaluation round only 14 candidates were selected [Mil+24] for further assessment in a second 12-18 months long review window. Notably, only two code-based DS schemes, namely CROSS and LESS, passed the first evaluation barrier, with the former being praised for the excellent performance and signature sizes when compared to SLH-DSA.

The draft version of NIST Interagency Report (IR) *8547* [Moo+24] published in November 2024 outlines NIST's planned strategy for shifting from cryptographic algorithms vulnerable to quantum computing threats to those designed for a post-quantum era. It highlights the current cryptographic standards at risk due to quantum advancements, as well as the quantum-resistant alternatives to be used during the transition, aiding federal agencies, industries, and standards organizations in updating their technology, products, services, and infrastructure to embrace post-quantum cryptography (PQC). In January 2025 NIST released the initial public draft of the *SP 800-227* [Ala+25a] special publication which provide a detailed guidance on how to properly use KEMs, giving some recommendations on their secure implementation. Starting from version 3.5.0 released in April 2025, the widespread software library OpenSSL [The25b] included the support for the standardized PQC algorithms ML-KEM, ML-DSA, and SLH-DSA, and set the hybrid KEM algorithm *X25519MLKEM768* the default TLS keyshare choice. At the same time, the release 10.0 of OpenSSH [The25a] introduced the support to the hybrid scheme *mlkem768x25519-sha256* combining the quantum resistant ML-KEM algorithm with the efficient ECDH scheme *X25519*.

Along with the U.S. based NIST standardization body, other national agencies are already defining guidelines and roadmaps for the post-quantum transition. The BSI, reported an extensive study on the current status of quantum computer development [BSI25]. Starting from 2019 the BSI agency provides a technical guideline for the use of cryptographic mechanisms [BSI24], particularly recommends the implementation of a KEM by means of FrodoKEM, Classic McEliece, and ML-KEM after its final standard becomes available. For the category of digital signatures, BSI recommends to use the stateful hash-based signature schemes XMSS and LMS/HSS, along with ML-DSA and SLH-DSA after their final standards becomes available. In any case, BSI encourage the use of hybrid solutions, and for the NIST standards the use of secu-

rity categories 3 and 5. The Agence nationale de la sécurité des systèmes d'information (French Cybersecurity Agency) (ANNSI) produces French national technical guidelines for cryptography in security products, regularly updating them to the best practices, and certifying products using cryptography for governmental use. The ANNSI agency recently released a position paper on post-quantum cryptography [ANS23], giving notice that post-quantum safety in the form of hybrid schemes will be mandatory starting from 2025 for some specific cases, with a list of accepted algorithms that might differ from the ones from NIST standards. Beyond 2030 it is intended to drop the requirement of hybrid schemes in favor of plain post-quantum algorithms.

A major concern moved by all those agencies is on how hybrid schemes should be implemented, particularly for the KEMs. The European Telecommunications Standards Institute (ETSI) released the technical specification *ETSI TS 103 744* [ETS20] which includes two suggested KEM combiner modes CatKDF and CasKDF, both making use of a Key Derivation Function (KDF). Some recommendation for KDF in KEMs are provided by NIST in the special publication *SP 800-56C rev.2* [BCD20]. The hybrid schemes for digital signatures are however simpler in design, as it is sufficient to both validate the pre-quantum and the post-quantum digital signatures.

1.4 Challenges of a Post-Quantum transition

Even if the threats posed by quantum computers appear not to be concerning in the near future, the current situation asks for immediate action to protect against retroactive attacks of the type 'store now, decrypt later' and 'verify now, forge later', undermining confidentiality and authenticity properties, respectively.

The complex life-cycle of a new cryptographic standard follows some rigorous steps. Starting from the ideation of a new cryptoscheme, an exploration of the parameters allow to identify several security-performance trade-offs, then assessed for few years by cryptanalyst, which eventually may impose some modifications to the scheme parameters. In case no devastating mathematical attacks are found, the performance and efficiency of Software (SW) prototypes are estimated, possibly proposing pre-computations improving the performance and new algorithms enriching the possible trade-offs. Hardware (HW) accelerators are then investigated, from full HW designs, HW/SW co-design and extensions of the Instruction Set Architecture (ISA).

In parallel Side-Channel Attacks (SCAs) are attempted to assess the practical security for in-the-field devices. These type of attacks do not rely on the mathematical properties of the scheme as for cryptanalysis, but rather on the fact that the device executing the algorithm is not an ideal black box and transmits sensitive information with the environment via multiple side-channels, such as the power consumption and the response delay. The large family of attacks can be classified as follows:

Timing attacks leverage variations in the computation runtime caused by cache memories and complex CPU instructions (e.g., division) to correlate the computation latency with the values of sensitive material such as the private key.

Simple Power Analysis (SPA) inspects the power consumption of a target device during the single execution of an operation employing the sensitive material to visually discern the individual bits composing it.

Differential Power Analysis (DPA) uses statistical techniques over many power traces (vertical attack) or several points of different operations in the same execution trace (horizontal attack) to find correlations between the intermediate values dependent on secret key and the power consumption.

Correplation Power Analysis (CPA) is a more targeted form of DPA that assumes a power model – such as the Hamming Weight approximating the dynamic power consumption of a switching transistor – to compute the correlations between the predicted power consumption under the model used and actual power traces to extract the private key.

Template attacks build detailed power models, called *templates*, for specific operations of a cryptographic scheme executed on a specific device, and then match them to new traces captured on a target device identical to the one used to build the models to deduce secrets. Instead of classical templates (e.g., Gaussian models), attackers now often use neural networks, decision trees, or support vector machines to model the relationship between traces and secret data more effectively, especially in complex or noisy environments.

Fault attacks intentionally induce faults during the computation on the device via glitches in the power or clock lines, or using electromagnetic (EM) or lasers pulses. By analyzing the behavior and outputs of the faulty execution and comparing them to expected ones, attackers can deduce secret information or bypass security checks.

In many cases the power analysis can be easily transformed into a less invasive and evident operation by merely observing the EM emissions of the device from a narrow distance.

To hinder these powerful attacks, it is common that in-the-field devices apply several countermeasures in forms of algorithms running in constant-time, hiding techniques, fault-tolerant masked algorithms, or a combination of them to have a broad range of cost/protection trade-offs. Employing constant-time algorithms safely address the timing attacks, while common hiding techniques introducing random runtime variations via casual clock frequency variations or the introduction of random stalls just raise the noise floor in the retrieved traces, and consequently the attacker either needs more sophisticated and expensive tools or must obtain more traces. Only the masking countermeasure is proved to be secure in the probing model, where an attacker can observe a limited number of intermediate values (or *probes*) without learning the secret. This is obtained by splitting all the sensitive variables into multiple random *shares*, so that no single share reveals information about the secret. However, masking is known to be computationally expensive, as when considering a d shares split, some operations can

have $O(d^2)$ quadratic complexity, and a non negligible quantity of randomness is consumed to refresh the share splitting. Glitch-resistant countermeasures such as Threashold Implementations (TI) [NRS08] and Domain-Oriented Masking (DOM) [GMK16] are promising techniques used to carefully design the masked circuits to achieve the desired probe security level. Compared to TI, DOM can arbitrary scale to high-order masking protection while having better efficiency than the TI approach, but requires fresh randomness during the operation of non-linear gates. The changing of the guards technique [Dae17] tries to maximize the randomness reuse. A comprehensive survey is provided in [CGF21], and some of the challenges in protecting PQC schemes are reported in [Saa23].

The new post-quantum cryptographic algorithm can also be assessed in real use-case scenarios as protocols in realistic simulated or testing environments, exciting any subtle incompatibility. If the new cryptoscheme shows promising metrics without roadblocking criticality, the standardization process and the update of regulations may begin, with new Request For Comments (RFC) for updated protocols. When everything gets approved, the incremental deployment of the new cryptographic scheme can start, with an initial phase where the replaced algorithm is deprecated but still available to use for legacy reasons, and then finally marked for removal. In case of cryptographic algorithms, there are several complications due to the large number of national regulations (military, commercial, etc.) that govern the use of cryptography. Moreover, as demonstrated by the old and unsecure hash algorithms MD5 and SHA-1, is quite difficult to deprecate in short time some cryptographic primitives, even more if deeply integrated in hardware. As the standardization process of post-quantum cryptoschemes goes on, employers of cryptographic primitives and network protocols should start preparing an inventory of the cryptoscheme they are relying on, with the ultimate goal of planning their phase out in favour of post-quantum alternatives, and even trying to improve the cryptographic agility of their products.

With the widespread use of cryptography in many applications and protocols comes the need of offloading the execution of the cryptographic primitives to dedicated accelerator, which nowadays comes at a relatively small area cost, allowing to free up Centra Processor Unit (CPU) time to other valuable tasks. Within this context, Smart Network Interface Cards (NICs) are programmable accelerators that are widely employed in data center networking to offload some tasks from server CPUs for security or efficiency purposes. To give some use cases, this performance boost can be particularly useful in bastion hosts where the traffic is encrypted/decrypted at the ends of two corporate networks, and frequent rekeying is desired for security reasons. Analogously, a front-end server of a three tier architecture typically manages a significant amount of TLS encrypted connections, while acting very little on the involved data which are forwarded to the second and third tier. In a REpresentational State Transfer (REST) based Application Programming Interfaces (APIs), which is a very common case nowadays, the amount of data per request is relatively limited, in turn making the overheads more visible. Furthermore, to improve the secure management and use of cryptographic keys,

a commercially established practice consists in the use of physically isolated Hardware Security Moduless (HSMs) providing a higher level of trustworthiness in terms of key material isolation and correctness and compliance verification.

Regarding the design of hardware accelerators for the cryptoscheme, there are many challenges derived by the large number of degrees of freedom.

Type of accelerator
Full hardware designs offer best performance, possibly exploiting any parallelism the cryptographic algorithm may offer, at cost of an extensive design effort and complex analysis and optimization. Hardware/software co-designs are maximizing the design effort rewards by offloading only the most demanding sub-algorithms composing the scheme from the CPU and interfacing with it via special purpose registers and directly working with the centralized memory. ISA extensions are moving the HW/SW synergy to a finer grain, accelerating particular operations in sub-algorithms, while also being deeply integrated in the CPU architecture.

Degree of parallelism
Software implementations must comply to some set of instruction set extensions to guarantee broad compatibility. In NIST PQC standardization, *Haswell* is the reference `x86_64` CPU architecture from Intel from 2013 compatible with the `AVX2` Single Instruction Multiple Data (SIMD) extension, enabling parallel execution of complex instructions on 256-bits registers. In case of hardware designs, there are no theoretical constraints on the size and number of parallel operations performed concurrently, leaving to the designer the freedom to choose the best solution suiting its needs, although many practical limitations are imposed by memories when computational units are interconnected to them.

Design Space Exploration (DSE)
Having many degrees of freedom requires to conduct a meticulous and extensive exploration of the possible solutions, determining many design points with performance, area, and efficiency values. Determining the Pareto front allows to restrict the attention to only a subset of meaningful choice of trade-offs. Notably, the exploration is conditioned by the target platform being a Field Programmable Gate Array (FPGA) or Application Specific Integrated Circuit (ASIC), the former largely used for prototypes or small volume production runs, and the latter having an even higher non-recurring engineering cost barrier that can only be amortized for very large production batches. FPGA deployments of cryptographic algorithms are also interesting from a flexibility standpoint, as testified by the employment of FPGAs in the automotive and telecommunication realms, where the long life cycle of the certified deployments takes precedence on the device power budget.

Validation
The complexity of the design along with the non-standardized interfaces make the validation process a particularly challenging task, and having a dedicated team for

that is not only a best practice but an actual need for extensive projects such as cryptographic accelerators. The *IEEE 1800.2* Universal Verification Methodology (UVM) [IEE20] is a standardized methodology widely adopted by the industry and supported by the major Electronic Design Automation (EDA) vendors such as Cadence, Synopsys, and Siemens. The verification framework help testing engineers to reuse most of the simulation code and simplifying the reproducibility.

Side-Channel Security
A discriminant factor in the evaluation of a cryptoscheme is the cost to be sustained to protect the sensitive operations in the cryptographic scheme from SCA. When comes to software-based protections, is common to assist a decrease in performance by one or even more orders of magnitudes, depending on how difficult is to protect the cryptographic algorithm. For hardware implementations such cost can be substantially reduced due to the inherently more flexible architecture with respect to the one of a CPU.

1.5 Contributions

Within the context of hardware accelerators, this thesis is presenting a thorough description of the essential components of two Post-Quantum cryptographic schemes in the standardization process held by NIST from the two most prominent categories, the lattice-based N-th degree Truncated polynomial Ring Units (NTRU) and the code-based Hamming Quasi-Cyclic (HQC) and Codes and Restricted Objects Signature Scheme (CROSS).

Enabled by the high flexibility of the designed components, some Design Space Exploration results will be reported for key operations, such as the multiplication between polynomials and the generation of random polynomials, with particular focus on the latency, occupied area, and efficiency metrics. A higher-level exploration for the NTRU scheme is also presented, showcasing the benefits of decoupled components with the ultimate goal of producing from the same design a high-performance unit and a compact solution.

The main investigation outcomes are three full hardware designs for the NTRU, HQC, and CROSS cryptographic schemes, and a polynomial multiplier unit for lattice-based schemes suited for a HW/SW co-design.

Many of these results were also presented in journals and international conferences [Ant+23b; Ant+23a; Ant+24a; Ant+24b; Ant+24c; ABP25]. Throughout this thesis, the term ‘we’ may appear to either involve the reader in the discussion, or to highlight the contributions of the other authors of the papers, my supervisors from both the academia and industry, notably professors Gerardo Pelosi and Alessandro Barenghi from the Politecnico di Milano university, and Ruggero Susella from ST Microelectronics. Moreover, the results concerning the CROSS design are obtained from the joint work with Patrick Karl, Ph.D. candidate at the Technical University of Munich. Credits to his contributions are attributed in the captions of tables and figures.

1.6 Thesis outline

A brief outlook of the theoretical background, the notation, and methodologies used throughout the thesis are presented in chapter 2. Lattice-based and code-based problems and cryptographic schemes are described in chapter 3 and chapter 4, respectively. The hash functions, one of the key components guaranteeing fundamental security properties, are introduced in chapter 5. The algorithms generating elements conforming to their defining set are presented in chapter 6. The chapter 7 describes the arithmetic algorithms used by the presented cryptographic schemes. Finally, the compositions of the top-level designs, with the description of units specific for each cryptographic scheme, are illustrated in chapter 8, and the results are compared to the state-of-the-art.

CHAPTER 2

Preliminaries and design methodology

This chapter provides the necessary foundation for the work presented in this thesis, introducing to the notation used throughout this document and the methodology employed to design, implement, and evaluate the proposed solutions. A brief theoretical background section describes the notions underpinning this research.

2.1 Theoretical background and notation

A binary string of length $l \geq 1$ is denoted with $\mathtt{a} \in \{0,1\}^l$. The concatenation $\mathtt{a} \| \mathtt{b}$ of two binary strings $\mathtt{a} \in \{0,1\}^l$ and $\mathtt{b} \in \{0,1\}^m$ is the $l+m$ long string formed by the first l binary digits of $\mathtt{a}$ immediately followed by the m binary digits of $\mathtt{b}$. The Boolean AND, OR, exclusive OR (XOR), and NOT operations are denoted by the symbols $\wedge$, $\vee$, $\oplus$, and $\neg$, respectively.

A set is denoted by an uppercase letter in calligraphic font $\mathcal{A}$, with the exception of notable sets such as the natural numbers $\mathbb{N}$, integers numbers $\mathbb{Z}$, and real numbers $\mathbb{R}$. The cardinality of the set $\mathcal{A}$ is represented by $|\mathcal{A}|$. An element a drawed with an uniform distribution from $\mathcal{A}$ is denoted as $a \xleftarrow{\$} \mathcal{A}$. When the procedure is made deterministic by using a Cryptographically Secure Random Number Generator (CSPRNG) and an input seed $\eta \in \{0,1\}^l$, $l \geq 1$, we refer to it with $a \leftarrow \mathrm{CSPRNG}(\eta, \mathcal{A})$.

With $[0,1)$ we denote the subset of $\mathbb{R}$ containing numbers i in the range $0 \leq i < 1$. Let $a, b \in \mathbb{N}$, then a divides b and conversely b is a multiple of a, for short $a \mid b$, if there

exist $c \in \mathbb{N}$ such that $ac = b$, otherwise $a \nmid b$.

2.1.1 Vector space

A finite field is indicated by a blackboard bold uppercase letter $\mathbb{F}$, and $\mathbb{F}^n$ is a vector space of dimension n over the finite field $\mathbb{F}$. A vector in a vector space is represented by a boldface lowercase letter, e.g. $\mathbf{v} \in \mathbb{F}$, and is composed by n scalars in $\mathbb{F}$ referenced with v_i: $\mathbf{v} = [v_0, v_1, \ldots, v_{n-1}]$ such that $v_i \in \mathbb{F}$, $\forall i \in \{0, 1, \ldots, n-1\}$. The null vector is represented by $\mathbf{0}$. The Euclidean norm of vector $\mathbf{v}$ is denoted by $\|\mathbf{v}\| = \sqrt{v_1 + v_2 + \ldots + v_{n-1}}$, and the Hamming weight of the same vector $\mathbf{v}$ is the number of non-zero coefficients in a vector and denoted as $\text{HW}(\mathbf{v})$. Two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{F}$ are orthogonal if their scalar product $\mathbf{u} \cdot \mathbf{v} \equiv \mathbf{u}^\top \mathbf{v} = 0$, where $\mathbf{u}^\top$ denotes the transpose of $\mathbf{u}$. To simplify some algorithmic operations, a polynomial $a(x) = a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$, for the sake of brevity also referred to as a, can be represented as a n-dimensional vector composed by its coefficients $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}] \in \mathbb{F}^n$.

A matrix of dimension $n \times m$ over the finite field $\mathbb{F}$ is represented by a boldface uppercase letter, e.g. $\mathbf{M} \in \mathbb{F}^{n \times m}$, where n is the number of rows, and m is the number of columns composing the matrix, and $M_{i,j}$ identifies the scalar element in $\mathbb{F}$ contained in the matrix $\mathbf{M}$ at row i and column j. The transpose of the matrix $\mathbf{M} \in \mathbb{F}^{n \times m}$ has m rows and n columns is indicated with $\mathbf{M}^\top \in \mathbb{F}^{m \times n}$. The identity matrix $\mathbf{I_n}$ is a $n \times n$ matrix with ones on its main diagonal, and zeros elsewhere. A permutation matrix $\mathbf{P}$ is an $n \times n$ square binary matrix such that each row and each column has a single 1 scalar (note that $\mathbf{P}^{-1} \equiv \mathbf{P}^\top$). Given $\mathbf{M} \in \mathbb{F}^{n \times n}$, $\mathbf{PM}$ contains the permuted rows of $\mathbf{M}$, whereas $\mathbf{MP}$ contains the permuted columns of $\mathbf{M}$. Let $\mathbf{v} \in \mathbb{F}^n$, then $\text{ROT}(\mathbf{v})$ is the $n \times n$ matrix obtained from the juxtaposition of $\mathbf{v}$ and a sequence of columns, each of which is obtained through a vertical cyclic rotation of the previous one by one cell. Let $\mathbf{M} \in \mathbb{F}^{n \times m}$ be a n rows by m columns matrix. Then $[\mathbf{M}, \mathbf{I}_n]$ is the matrix with n rows and $m + n$ columns constructed by the adjunction of the n columns of $\mathbf{I}_n$ after the ones of $\mathbf{M}$. Similarly, $[\mathbf{M} \mid \mathbf{I}_m]$ is the $n + m$ rows by m columns matrix constructed by the adjunction of the m rows of $\mathbf{I}_m$ after the ones of $\mathbf{M}$.

2.1.2 Algebraic structures

An *algebraic structure* consists of a set $\mathcal{A}$, a collection of binary operations working with the elements in $\mathcal{A}$, such as addition and multiplication, and a set of axioms that the operations need to satisfy, such as closure, commutativity and associativity.

A *group* $G := (\mathcal{A}, \cdot)$ is an algebraic structure with $\mathcal{A}$ close under one binary operation $(\cdot)$ that is associative, has an identity element e, and for which all elements are invertible. In an *additive group* the group operation is defined using addition $(+)$, with the inverse of $a \in \mathcal{A}$ being $-a$ and the identity element $e = 0$. Similarly, in a *multiplicative group* the group operation is the multiplication $(*)$, with the inverse element defined as a^{-1}, and the identity element $e = 1$. If the binary operation is also commutative, the algebraic structure is an *abelian group*. If the group G has a finite amount

of elements, then the *group order* $|G|$ corresponds to the cardinality of the set of its elements. An element $g \in G$ is called a *group generator* $\langle g \rangle = S$ when it produces every element of the group $S \subseteq G$ by repeatedly applying the group operation $(\cdot)$ to g or its inverse element. In particular, $S = \{ng \mid n \in \mathbb{Z}\}$ when G is an additive group, or $S = \{g^n \mid n \in \mathbb{Z}\}$ when G is a multiplicative group (in this case, S is a *cyclic subgroup*). The *order* of g is defined as the smallest positive integer m such that $mg = e$ or $g^m = e$ for additive or multiplicative groups, respectively. A set $\mathcal{G}$ can equally be a generator of the subgroup $\langle \mathcal{G} \rangle = S \subseteq G$ containing all the elements of G that can be expressed as the composition of the elements in $\mathcal{G}$ and their inverses. Considering an additive group, then $S = \langle \{g_0, \ldots, g_{m-1}\} \rangle = \left\{ \sum_{i=0}^{m-1} n_i g_i \mid n_i \in \mathbb{Z} \right\}$, whereas for a multiplicative group $S = \langle \{g_0, \ldots, g_{m-1}\} \rangle = \left\{ \prod_{i=0}^{m-1} g_i^{n_i} \mid n_i \in \mathbb{Z} \right\}$.

A *ring* $\mathbf{R} := (\mathcal{A}, +, *)$ is an algebraic structure with two binary operations, called addition $(+)$ and multiplication $(*)$, such that it is an abelian group with respect to the addition, and the multiplication is associative, and distributive over the addition. Rings do not generally require each element to have a multiplicative inverse. If the multiplication is also commutative, the algebraic structure is a *commutative ring*.

A *field* $\mathbb{F}$ is a commutative ring in which every non-zero element of $\mathcal{A}$ has a multiplicative inverse. A few examples are the real numbers $\mathbb{R}$, the rational numbers $\mathbb{Q}$, and the complex numbers $\mathbb{C}$. The multiplicative group of a field $\mathbb{F}$ is indicated as $\mathbb{F}^*$, and does not contain the zero element. A finite field is a field that contains a finite number of elements. The most used ones in cryptography are the prime field $\mathbb{F}_p$ having prime order p, or the extension fields $\mathbb{F}_{p^n}$ of elements from a n-dimensional vector space over $\mathbb{F}_p$.

A *vector space* $\mathbb{F}^n$ over a field $\mathbb{F}$ consists of a set $\mathcal{A}$ of n-dimension vectors of scalars in $\mathbb{F}$ that is an abelian group under the binary addition operation, and a ring homomorphism from the field $\mathbb{F}$ into the endomorphism ring of this group under the multiplication of a vector by a scalar. A vector subspace $\mathbb{F}^k$ of a vector space $\mathbb{F}^n$, with $k \leq n$, is a subset of $\mathbb{F}^n$ that is closed under vector addition and scalar multiplication. A set $\mathcal{B}$ of vectors form a basis for the vector space $\mathbb{F}^n$ if every element of $\mathcal{A}$ can be written as a linear combination of the vector in the basis, which are linearly independent. A vector space can have several basis, although $|\mathcal{B}|$ is fixed and defining the dimension of the vector space.

A *quotient ring* $\mathbf{R}/I$ constructed from a ring $\mathbf{R}$ and a two-sided ideal I in $\mathbf{R}$, has elements that are the cosets of I in $\mathbf{R}$, and therefore it also called *residue class ring*.

A *lattice* $(\mathcal{L}, \wedge, \vee)$ is a partially ordered set $\mathcal{L}$ with two or more binary operations, including the meet $(\wedge)$ and join $(\vee)$ operations, and connected by the absorption law.

2.1.3 Polynomials and Galois fields

A polynomial $f(x)$ with coefficients from GF(q) is called *monic* if the coefficient of the highest power of x is 1. A polynomial $p(x)$ of degree m over GF(q) is said to be *irreducible* if it is not divisible by any polynomial over GF(q) of degree less than m

but greater than zero. An irreducible polynomial $p(x)$ of degree m over GF(q) is called *primitive* if the smallest positive integer n for which $p(x)$ divides $x^n - 1$ is $n = q^m - 1$.

For any positive integer m, a Galois field GF(q^m) with q^m elements can be constructed from the ground field GF(q). The construction of GF(q^m) is based on a *monic primitive* polynomial $p(x)$ of degree m over GF(q). Let α be a root of $p(x)$. Then, $0, 1, \alpha, \alpha^2, \ldots, \alpha^{q^m-2}$ form all the elements of GF(q^m), and $\alpha^{q^m-1} = 1$. The element α is called a *primitive element*. Every element β in GF(q^m) can be expressed as a polynomial in α, $\beta = a_0 + a_1\alpha + a_2\alpha^2 + \ldots + a_{m-1}\alpha^{m-1}$ where $a_i \in$ GF(q) for $0 \leq i < m$. Then, $[a_0, a_1, \ldots, a_{m-1}]$ is a vector representation of β. Therefore, every element in GF(q^m) has three ways of being represented: power of a primitive element (0 is usually mapped to α^∞), polynomial, and vector.

The elements in GF(q^m) form all the roots of $x^{q^m} - x$. Let β be an element in GF(q^m). The minimal polynomial of β is the monic irreducible polynomial $\phi(x)$ of the smallest degree over GF(q) that has β as root; that is, $\phi(\beta) = 0$. Let e be the smallest non-negative integer for which $\beta^{q^e} = \beta$. The integer e is called the *exponent* of β and $e \leq m$. The elements $\beta, \beta^q, \beta^{q^2}, \ldots, \beta^{q^{e-1}}$ are conjugates. Then $\phi(x) = \prod_{i=0}^{e-1}(x - \beta^{q^i})$ and $\phi(x)$ divides $x^{q^m} - x$.

2.1.4 Complexity theory for classical computers

We can characterize the resources used by a specific algorithm in terms of completion time T (e.g., clock cycles) and space S (e.g., number of parallel processing units or memory cells). Determining the exact requirements may be extremely difficult for some algorithms, therefore they are usually defined as functions f of the input with binary size n, specifying only a few dominating terms, thus characterizing their asymptotic behaviors. Considering inputs with a fixed length n, the running time $T(n)$ of an algorithm may vary significantly. In this case, the average-case time complexity does not coincide with the worst-case time complexity $O(f(n))$ (upper bound) or the best-case time complexity $\Omega(f(n))$ (lower bound). If $O(f(n)) = \Omega(f(n))$, then $T(n) = \Theta(f(n))$ (tight bound). A problem can be classified according to the time complexity $T(n)$ of the best known algorithm solving it:

LOG problems solved by an algorithm in logarithmic time $\Theta(\log n)$

P class of the computational problems which can be solved by a deterministic Turing machine in polynomial time $\Theta(n^i),\ i \geq 1$. Problems in the P class are practically treatable for any n.

EXP class of computational problems that can be solved in exponential time $\Theta(2^{n^i}),\ i \in \mathbb{N}$. Solving a problem in this class is deemed not practically treatable even for moderately large n ($\mathtt{P} \subsetneq \mathtt{EXP}$).

NP contains the computational problems solvable in polynomial time by a non-deterministic Turing machine ($\mathtt{P} \subseteq \mathtt{NP}$). A deterministic Turing machine can verify in polynomial time if the solution is valid.

A specific problem is *hard* for a class C (C-hard) if every problem in C can be reduced to it in polynomial time. If the same problem is actually in C, then it is called C-*complete*.

2.2 Design Methodology

The aim of this thesis is to develop and assess efficient digital systems for lattice-based and code-based post-quantum asymmetric cryptographic schemes. Designs will be described at the Register Transfer Level (RTL) using the SystemVerilog (SV) language, and not relying on any proprietary Intellectual Properties (IPs) or specific features of FPGA macros, such as Digital Signal Processor (DSP) units.

The developed hardware designs are implemented on FPGAs for an easy and fair comparison with the current state-of-the-art. Those platforms are the preferred choice for fast validation of prototypes due to its streamlined and standardized synthesis and implementation flow compared to the one for ASICs. Nonetheless, the use of FPGAs goes beyond the prototyping use, as these platforms are extensively used in specific industry areas where the the number of samples to be produced are low enough to not justify the non-recurring expenses necessary to setup an ASIC production line, or when the flexibility of reprogram the hardware is desired. A few examples of industrial areas using FPGAs are aerospace and defense, telecommunications, automotive, industrial automation, medical devices, and even consumer electronics.

The most promising designs are also investigated in an ASIC design flow, giving insights into the designs for high-volume production purposes. However, given the substantial differences among the ASIC design and production processes in terms of Process Design Kit (PDK) and toolchains, the comparison with other works results is, structurally, more difficult.

2.2.1 Design Tools

To enhance the reproducibility of results and rapidly check for regressions in a Continuous Integration for Verification (CIV) automation, open-source programs and frameworks are used to synthesize, implement, and verify the designs, with the only exception being the *AMD Vivado* suite (formerly Xilinx's product). Nonetheless, the FPGA parts used in the benchmarks, the Artix-7 `xc7a200tfbg484-3` and Zynq UltraScale+ `xczu7ev-ffvc1156-3-e`, are covered by the WebPACK/Standard license, freely available once applied for the U.S. Government Export Approval. In this thesis is adopted the open-source *Verilator* [Sny+] SV simulator, with testbenches written either in `C++`, or in `Python` using the *cocotb* [FOS19] simulation library. In case of the verification of complex designs, it is followed the *IEEE*1800.2 Standard for UVM [IEE20], via the *pyuvm* [SS18] framework to produce reusable and scalable testbenches. The described verification toolchain, while using only open-source software, can only carry out behavioral simulations, thus not covering the need for post-synthesis and post-route simulations. However, this limitation can be worked around using any commercial simulator supported by the cocotb simulation library, such as *Synopsys VCS* or *Cadence*

Xcelium. In this work, a behavioral simulator was sufficient to iron out all the design bugs before generating a bitstream for the Digilent Arty A7-100T board and ultimately testing the design via Known Answer Tests (KATs) transmitted by a host PC via the Universal Asynchronous Receiver-Transmitter (UART) bus. Regarding the ASIC design flow, this thesis mainly adopts the *OpenROAD* project [Aja+19] via the *SiliconCompiler* [ORM22] framework, both open-source tools, along with the *FreePDK*45 [Sti+07] PDK, a reliable and widely adopted open-source predictive technology library having 45 nm planar transistors and 8 metal layers. The *OpenRAM* project [Gut+16] is a Static RAM (SRAM) compiler compatible with the FreePDK45 technology library that can produce the layout, netlists, timing and power models necessary to efficiently implement the large memories used in the designs, allowing to proceed with the back-end design flow up to the sign-off stage where the final Design Rule Check (DRC) and Layout Versus Schematic (LVS) checks are carried out. In addition to the previous tools, in some instances the results are reported using the *Synopsys Design Compiler* tool for the synthesis using a proprietary 40 nm technology library from ST Microelectronics.

2.2.2 Latency, area and efficiency metrics

NIST elected the AMD Artix-7 as the reference platform to evaluate the hardware implementation of accelerators for the post-quantum cryptographic schemes. Since some designs could not fit in the largest product of such family, the AMD Zynq UltraScale+ was widely adopted by many researchers as an alternative platform offering more resources.

The maximum working frequency of a design synthesized for a FPGA platform is determined using a binary search approach on the clock period defined in the constraint file passed to the Vivado suite in a out-of-context synthesis, making sure to specify a clock buffer cell in the FPGA fabric to define the root of the synthesized clock tree, and as a result having a more precise timing estimation. Thus, the latency figures are expressed in μs and not clock cycles in order to consider the complexity of the design determining the maximum working frequency.

To compare the area results of a target with many heterogeneous resources like FPGAs, in this thesis it is introduced a novel *equivalent Slice* (eSlice) synthetic area indicator, specific to the chosen FPGA target, encompassing the use of Look-Up Tables (LUTs), Flip-Flops (FFs), Block RAMs (BRAMs), and DSPs resources. LUTs and FFs are the fundamental building blocks for creating any digital circuit within an FPGA, implementing logic functions and small memory elements. BRAMs are more efficient memory blocks used to store larger amounts of data, with data accessed through a limited number of memory ports. DSPs blocks are specialized hardware units capable of performing specialized arithmetic operations, such as multiply-and-accumulate. The idea of the eSlice area indicator consists in converting the used BRAM and DSP resources to a set of FF and LUT resources using the results of an out-of-context synthesis on designs that are inferred in exactly one DSP or BRAM unit. In the former case we described a multiply-and-accumulate with pre-addition, carefully sized to match the

DSP48E1 architecture from the AMD UG479 user guide, and set the use_dsp = "no" directive, obtaining a synthesis result using 538 LUT and 232 FF. For the BRAM case, we annotated a 32 Kb Simple Dual Port RAM template with the ram_style = "distributed" directive, obtaining a resulting design composed by 848 LUT and 548 FF. Finally, considering that a Slice in a AMD 7-Series FPGA includes four 6-input LUTs and eight FFs, we derive the number of eSlices as $\mathbf{max}\{\lceil \mathrm{LUT}/4 \rceil, \lceil \mathrm{FF}/8 \rceil\}$. In case the AMD UltraScale+ FPGA is used, the number of eSlices are computed as $\mathbf{max}\{\lceil \mathrm{LUT}/8 \rceil, \lceil \mathrm{FF}/16 \rceil\}$, reflecting the composition of LUT and FF in a single Slice of the UltraScale+ fabric. This metric still has some shortcomings, as the conversions are a worst-case scenario estimate when DSP or BRAM are underutilized, while, by contrast the eSlice metric is a best-case scenario as it does not consider routing congestion. Moreover, other specialized units in the FPGA fabric, such as CARRY and MUX units, are not converted in a set of FF and LUT resources as those elements are available in every Slice unit, and most of the times the usage count of those units are omitted during the evaluation of the results.

The efficiency is computed as the product of the latency and the area occupied by the design, and the lowest resulting value denotes the most efficient solution. We are using eSlice as the area indicator and a proper latency time scale to produce an efficiency indicator with a suitable scale.

The procedure for evaluating the ASIC design results is less strict than the one for FPGAs. The exact maximum working frequency is not determined via a binary search algorithm but automatically computed by the OpenROAD toolchain from the minimum clock period required by the critical circuit path in the netlist. The occupied silicon area is generally expressed in μm^2. Considering the silicon area of a single NAND gate in the FreePDK45 technology library, the Gate Equivalent (GE) technology-independent complexity indicator is computed by dividing the whole circuit area by the NAND cell area. Finally, the efficiency indicator is expressed as time latency by the GE area.

CHAPTER 3

Lattice-based cryptography

Lattices are algebraic structures $(\mathcal{L}, \wedge, \vee)$ consisting of a partially ordered set $\mathcal{L}$ with two or more binary operations, denoted as the meet ($\wedge$) and join ($\vee$) operations, and connected by the absorption law. The Shortest Vector Problem (SVP) and Closest Vector Problem (CVP) are hard problems studied for more than a century, and researchers started to investigate their use to build trapdoor one-way functions to construct different public-key schemes from RSA and EC-based schemes.

Given $k \in \mathbb{N}$ linearly independent vectors $\mathbf{b}_i \in \mathbb{R}^n$, with $1 \leq i \leq k$ and $k \leq n$, an instance of a lattice $\mathcal{L}$ is the set of points in $\mathbb{R}^n$:

$$\mathcal{L} := \Lambda(\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_k) = \left\{ \sum_{i=1}^{k} a_i \mathbf{b}_i \mid a_i \in \mathbb{Z} \right\} \subseteq \mathbb{R}^n \tag{3.1}$$

Let $\mathbf{B}$ be the matrix constructed by $(\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_k)$, where $\mathbf{b}_i$ are the row vectors. We have that $\mathcal{L} = \textsc{RowSpan}(\mathbf{B})$ is a linear subspace of $\mathbb{R}^n$ generated by all possible linear combinations (span) of the row vectors. Therefore, $\mathbf{B}$ is a basis of the lattice $\mathcal{L}$ of dimension n and rank K, and there are multiple basis that generate the same lattice. The fundamental parallelepiped is the unit volume generated by the vectors in the basis $\mathbf{B}$, and is defined as $\mathcal{F} = \left\{ \sum_{i=1}^{k} a_i \mathbf{b}_i \mid a_i \in [0, 1) \right\}$. If $\mathcal{L} \subseteq \mathbb{Z}^n$, $\mathcal{L}$ is an integer lattice. Figure 3.1 depicts an integer two-dimensional lattice defined by two different basis, the fundamental parallelepiped, and the shortest vector in the lattice.

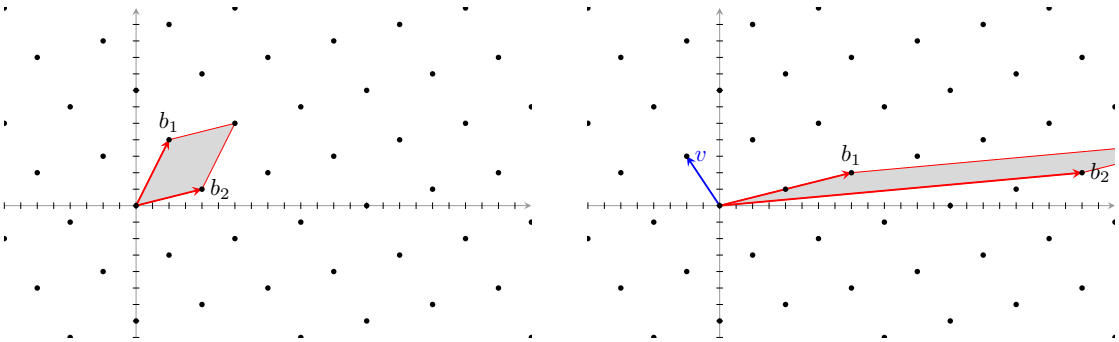


Figure 3.1: *Integer lattice $\mathcal{L} \subset \mathbb{Z}^2$ with $n = 2$ dimension, black dots are the lattice points in $\mathcal{L}$. In the picture on the left, the basis $\mathbf{B}$ is composed by two vectors $\mathbf{b}_1 = [4, 2]$ and $\mathbf{b}_2 = [2, 4]$, which are almost orthonormal among them, and the fundamental parallelepiped is the area highlighted in light gray generated by the two basis vectors. In the picture on the right, the basis $\mathbf{B}$ is composed by two vectors $\mathbf{b}_1 = [8, 2]$ and $\mathbf{b}_2 = [22, 2]$, and the shortest vector $\mathbf{v}$ in the lattice $\mathcal{L}$ is represented with a blue arrow. As the dimension of the lattice increase, finding the shortest vector becomes not so trivial.*

The shortest vector $\mathbf{v}$ in a lattice is not unique, and determining it is proved to be NP-hard problem even on average case, and it is known as the Shortest Vector Problem:

$$\|\mathbf{v}\| = \min(\|\mathbf{u}\| \mid \mathbf{u} \in \mathcal{L} \wedge \mathbf{u} \neq 0) \tag{3.2}$$

Denote as $\Delta(\mathbf{B})$ the determinant of the matrix $\mathbf{B}$ row-spanning $\mathcal{L}$. Hermite proved that $\mathcal{L}$ contains a short vector $\mathbf{v}$ such that $\|\mathbf{v}\| \leq \gamma(n) \sqrt[n]{\Delta(\mathbf{B})}$. The value for $\gamma(n)$ making the relation tight is known only for lattice dimension $n \leq 8$, whereas for large values of n, $\sqrt{\frac{n}{2\pi e}} \leq \gamma(n) \leq \sqrt{\frac{n}{\pi e}}$. Moreover, in a random instance of a lattice, we expect the shortest vector to be $\|\mathbf{v}\| \approx \sqrt{n} \sqrt[n]{\Delta(\mathbf{B})}$ in length due to the upper bound from Minkowki's first theorem.

A second lattice-based NP-hard problem is the Closest Vector Problem: given a random point in the vector space $\mathbf{u} \in \mathbb{R}^n$ not belonging to the lattice $\mathcal{L} \subseteq \mathbb{Z}^n$, find the lattice point $\mathbf{z} \in \mathcal{L}$ closest to it, which can be determined by searching for the vector $\mathbf{u} - \mathbf{z}$ with minimum euclidean norm. When vectors in the basis $\mathbf{B}$ of the lattice are almost orthonormal ($\|\mathbf{b_i}\| \approx 1$ and as orthogonal as possible, also referenced as a "good" basis), we can apply the Babai's closest vertex algorithm to easily find the vertex $\mathbf{v} \in \mathcal{L}$ of the fundamental parallelepiped containing the random point $\mathbf{u}$, that in this case coincides also to the closest point in the lattice $\mathbf{z}$. However, when the vector of the basis are not orthonormal, Babai's algorithm is not determining the actual closest lattice point. Figure 3.2 represents the execution of Babai's algorithm on a low-dimensional lattice $\mathcal{L} \subseteq \mathbb{Z}^2$ when using both a "good" basis and a generic basis.

Solving the SVP and CVP is straightforward when using a orthonormal basis since the euclidean norm of every single linear combination of the basis vectors degrades to $\|\mathbf{v}\| = \sum_{i \leq n} a_i^2 \|\mathbf{v_i}\|^2$. Therefore, the euclidean norm for some two vectors $\mathbf{u} \in \mathbb{R}^n$, $\mathbf{v} \in \mathcal{L}$, for $a_i \in \mathbb{Z}$ and $b_i \in \mathbb{R}$, is $\|\mathbf{u} - \mathbf{v}\| = \sum_{i \leq n} (a_i - b_i)^2 \|\mathbf{v_i}\|^2$. Consequently, the easiest way to minimize the scalars in the resulting vector consists in taking the closest

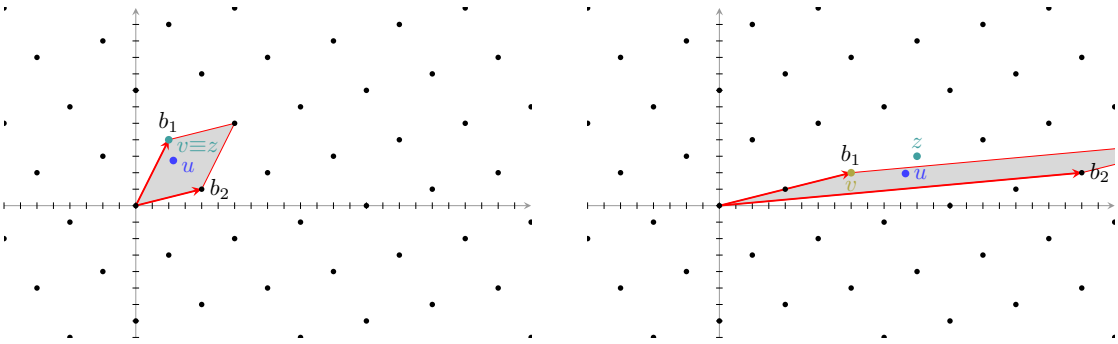


Figure 3.2: *Application of Babai's closest vertex algorithm in $\mathcal{L} \subset \mathbb{Z}^2$ with a basis having almost orthogonal vectors (left image), and using a non-ideal basis (right image), to determine a vertex $\mathbf{v}$ in the fundamental parallelepiped that minimizes the euclidean norm $\|\mathbf{u} - \mathbf{v}\|$, with $\mathbf{u} \in \mathbb{R}^2$. The actual closest lattice vector $\mathbf{z}$ in $\mathcal{L} \subseteq \mathbb{Z}^2$ cannot be determined with Babai's algorithm when is used a non-ideal basis.*

integer to each b_i. However, the best known algorithms solving the SVP on a random lattice have time complexity $T(n) = \Omega(2^n)$, thus are deemed practically infeasible to solve even for small values of n (e.g. n above 100).

Sometimes, it is not necessary to find the exact shortest vector $\mathbf{v} \in \mathcal{L}$, but a reasonably short one with approximation factor μ. Hence, this new problems take the name of Approximate Shortest Vector Problem (μSVP) and Approximate Closest Vector Problem (μCVP). A lattice reduction algorithm is producing a basis that is reasonably orthonormal starting from a generic integer lattice basis. The Lenstra-Lenstra-Lovász (LLL) algorithm produces in polynomial time with guaranteed worst-case performance a lattice basis with approximately orthonormal vectors if the lattice dimension $n < 100$, but in practice generates basis with non-negligible defects for $n > 300$ since the guaranteed approximation factor μ is exponential. More advanced lattice reduction techniques try to provide trade-offs between running time and approximation factor μ. The Block Korkine-Zolotarev (BKZ) lattice reduction has exponential time complexity, but the resulting lattice has better orthogonality with respect to LLL.

The first use of lattices for public-key cryptography appeared in 1996 by Ajtai and Dwork [AD97], proposing a probabilistic public-key cryptoscheme relying on the hardness of solving the SVP, and proving that solving the problem in a random lattice instance is as hard as solving the problem in the worst-case scenario. In a Probabilistic Public-Key Encryption (PPKE) scheme a valid ciphertext can fail to decrypt, and a careful selection of the parameters of the scheme can minimize or even remove the failure probability. However, the proposed scheme was impractical both in terms of public key size, in the order of few MiB, and computational cost, particularly for the generation of the public key taking few hours. Nonetheless, the achieved result encouraged the research efforts of lattice-based trapdoor one-way functions.

In short time, a new proposal came from [GGH97] in 1997 by Goldreich-Goldwasser-Halevi, this time having the security of the scheme based on the hardness of solving the

CVP, although not proving its security under the worst-case assumption. The results showed a substantially reduced computational complexity and public key size, improving the feasibility of a lattice-based public key schemes.

It was in 1998 that Hoffstein, Pipher and Silverman proposed the first practical scheme in [HPS98] called N-th degree Truncated polynomial Ring Units (NTRU), and sometimes also referred to NTRUEncrypt, based on truncated polynomial rings. The new proposal achieves the key generation, encryption and decryption in $O(n^2)$ basic operations, producing key-pair and ciphertext sizes in the order of $O(n \log n)$ bits, with n being the lattice dimension.

An extensive analysis of the early lattice-based public-key proposals is presented in [Har15], along with currently best known attacks to those schemes.

3.1 NTRU and LWE

Consequently to the first proposal of NTRU, numerous variants appeared in the following years to further optimize the scheme in terms of security and ciphertext/keys sizes:

IEEE 1363.1 NTRUEncrypt Institute of Electrical and Electronics Engineers (IEEE) standardized a scheme derived from NTRUEncrypt in 2008 that introduced an efficient improper-key variant and improved the security with the SVES padding scheme.

NTRU-HPS, NTRU-HRSS IND-CCA2 secure schemes presented in the NIST PQC standardization process. The original proposals were based on a PPKE scheme, which later in the third round was replaced with a Deterministic Public-Key Encryption (DPKE) scheme to remove any decryption failures NTRU-HPS is similar to the original NTRUEncrypt cryptoscheme, which still selects coefficients from fixed-weight sample spaces, while NTRU-HRSS selects coefficients in an arbitrary way as a countermeasure to the key mismatch attack proposed by Ding on the original NTRU cryptoscheme.

Streamlined NTRU Prime one of the two variants proposed by the NTRU Prime IND-CCA2 secure scheme that is not using a cyclotomic polynomial ring and sparse ternary secret to remove lattice structure not strictly necessary for the correctness of the scheme and potentially exploitable by attackers, at the cost of a probabilistic decryption, lower compactness, and slightly higher execution times. The errors are generated deterministically via rounding operations, reducing the size of the ciphertext and the opportunity to incorrectly generate small ring elements, and simplifying the protection against chosen-ciphertext attacks.

NTTRU IND-CCA2 secure scheme that uses lattice parameters to be compatible with Number-Theoretic Transform (NTT) multiplication to further reduce the timing complexity by $\approx 10\times$

A completely different family of lattice-based schemes, known under name of Learning With Errors (LWE) / Learning With Rounding (LWR) or product-ring schemes to differentiate from the quotient-ring NTRU-based schemes, appeared more recently following the proposal of the Gentry-Peikert-Vaikuntanathan (GPV) trapdoor function in [GPV08]. The most important practical differences are that the expensive computation of the inverse element in the polynomial ring is not used, and that the module variants use the same ring structure for multiple security margins offered. This in turn allowed the generation of schemes that are faster and slightly more efficient than the NTRU-based counterparts, although it is not at all clear which of product-ring and quotient-ring schemes is a safer option. During the NIST PQC standardization process, the proposed LWE candidates were numerous, most notably the Ring-LWE (R-LWE) NewHope, the Ring-LWR (R-LWR) NTRU LPRime, the Module-LWE (M-LWE) CRYSTALS-Kyber, and the Module-LWR (M-LWR) SABER. More details on the differences are detailed in the report [Ber+24].

A fundamental patent issue arose during the the evaluation phase in the NIST PQC standardization process as it appeared that many patents, such as the U.S. 9094189 and 9246675 patents, could potentially limit the broad diffusion of the future PQC standard due to license fees. By contrast, NTRU-derived schemes are not subject to any limitation as the U.S. patent 6081597A expired in August 2017. When NIST announced the standardization of ML-KEM, the patents owned by a US entity and French institutions were identified and two patent license agreements were redacted to facilitate adoption by guaranteeing a royalty-free use for the implementers even for commercial purposes, although limited to the exact specification of NIST's standard.

3.2 NTRU HPS and NTRU HRSS

In this section we report the mathematical background of the NTRU KEM submission to the NIST PQC standardization effort [Che+19] (from now referenced as NTRU for the sake of brevity), which includes two variants: NTRU-HPS [HPS98], and NTRU-HRSS [Hül+17].

3.2.1 Algebraic structures and parameters sets

Consider the prime integers n, p, and the integer q coprime with both p and n such that $p \ll q$. Denote with $\Phi_1 = x - 1 \in \mathbb{Z}[x]$ and $\Phi_n = \frac{x^n-1}{x-1} = x^{n-1} + x^{n-2} + \cdots + x + 1 \in \mathbb{Z}[x]$ the 1-st and the n-th irreducible cyclotomic polynomial, respectively. All operations in the NTRU cryptoscheme are operations between polynomials over the quotient rings $\mathbf{R}_q \cong \mathbb{Z}_q[x]/\langle \Phi_1 \Phi_n \rangle$, $\mathbf{S}_q \cong \mathbb{Z}_q[x]/\langle \Phi_n \rangle$ or $\mathbf{S}_p \cong \mathbb{Z}_p[x]/\langle \Phi_n \rangle$. Polynomial addition and subtraction operations between two polynomials $a, b \in \mathbf{R}_q$ to compute the resulting polynomial $a \pm b = c \in \mathbf{R}_q$ are regularly performed between scalars of the same unknown degree:

$$c_k = a_k \pm b_k \bmod q, \quad \forall k \in \{0, \ldots, n-1\} \tag{3.3}$$

Also polynomial multiplication is used, that in the quotient polynomial ring assumes the form of a circular convolution: let a, b be two polynomials in $\mathbf{R}_q$, their product $a \cdot b = c \in \mathbf{R}_q$ has coefficients

$$c_k = \sum_{i+j \equiv k \bmod n} a_i b_j \bmod q, \quad \forall k \in \{0, \ldots, n-1\} \tag{3.4}$$

Finally, the multiplicative inverse element of the polynomial $a \in \mathbf{R}_q$, if exists, is defined as $b = a^{-1} \in \mathbf{R}_q$, such that $a \cdot b = 1$. As the polynomial coefficients are in $\mathbb{Z}_q \equiv \mathbb{Z}/q\mathbb{Z}$, hence the integers group of the remainders modulo q, the result of all scalar operations are performed modulo q.

We represent polynomials in $\mathbf{R}_q$ and $\mathbf{S}_q$ with coefficients encoded in two's complement with $\varepsilon_q = \lceil \log_2(q) \rceil$ bits, i.e.: $\mathbb{Z}_q = \left\{-\frac{q}{2}, -\frac{q}{2}+1, \cdots, 0, \cdots, \frac{q}{2}-1\right\}$. The parameter p is set to 3 in all parameter sets of the NTRU specification [Che+19] thus, polynomials in $\mathbf{S}_p = S_3$ have coefficients encoded in ternary form, i.e.: $\mathbb{Z}_3 = \{-1, 0, 1\}$, with $\varepsilon_p = \lceil \log_2(p) \rceil = 2$ bits.

NTRU utilizes three additional sets of ternary polynomials within $\mathbb{Z}_p[x]/\langle \Phi_1 \Phi_n \rangle$ of degree at most $n-2$. Let $\mathcal{T}$ represent the set of ternary polynomials with a variable count of non-zero coefficients; $\mathcal{T}(d)$, where d is an even number, represents the set of fixed-weight polynomials having $d/2$ coefficients equal to $+1$ and $d/2$ coefficients equal to -1. Furthermore, $\mathcal{T}_+$ denotes the subset of $\mathcal{T}$ containing polynomials $v(x) = \sum_i v_i x^i$ satisfying the condition $\sum_i v_i v_{i+1} > 0$ (indicating a non-negative correlation property). Observe that the elements of $\mathbf{S}_p = S_3 \cong \mathbb{Z}_p/\langle \Phi_n \rangle$ are valid members of $\mathcal{T}$.

The map LIFT is an injective function $\text{LIFT} : \mathbb{Z}_p[x]/\langle \Phi_1 \Phi_n \rangle \to \mathbb{Z}[x]$ such that $\mathbf{S}_p(\text{LIFT}(m)) = m$, for all $m \in \mathcal{L}_m$, where $\mathbf{S}_p(a)$ represents the canonical representative element of the equivalence class in $\mathbf{S}_p$ to which the polynomial a belongs. In other words, LIFT is the map that allows to send a ternary polynomial m to its canonical representative element within a residue class in $\mathbf{S}_p \cong \mathbb{Z}_p[x]/\langle \Phi_n \rangle$.

An NTRU parameter set is specified by the tuple $(n, p, q, \mathcal{L}_f, \mathcal{L}_g, \mathcal{L}_r, \mathcal{L}_m, \text{LIFT})$, where $\mathcal{L}_f$, $\mathcal{L}_g$, $\mathcal{L}_r$, and $\mathcal{L}_m$ coincide, respectively, with one of the sets of polynomials denoted as $\mathcal{T}$, $\mathcal{T}(d)$, $\mathcal{T}_+$, depending on the specific instance of the cryptosystem. The NTRU parameter set is correct if, the ternary polynomials f, g, r, m are uniformly sampled from $\mathcal{L}_f, \mathcal{L}_g, \mathcal{L}_r, \mathcal{L}_m$, respectively, and $(f \cdot \text{LIFT}(m) + g \cdot r \cdot p) \bmod (\Phi_1 \Phi_n) \in \mathbb{Z}_q$, where $\mathbb{Z}_q = \left\{-\frac{q}{2}, -\frac{q}{2}+1, \cdots, 0, \cdots, \frac{q}{2}-1\right\}$. All NTRU parameter sets in [Che+19] are correct.

NTRU-HPS parameters sets

The NTRU-HPS parameter set takes $p = 3$, and q as a power of two and n as a prime number such that both 2 and 3 are generators of the multiplicative group $(\mathbb{Z}_n^*, \cdot)$. A uniform sampling procedure is defined over both the set of variable-weight and the set of fixed-weight polynomials with degree at most $n-2$, i.e.: $\mathcal{L}_f = \mathcal{T}$, $\mathcal{L}_g = \mathcal{T}(q/8-2)$, $\mathcal{L}_r = \mathcal{T}$, $\mathcal{L}_m = \mathcal{T}(q/8-2)$, while the LIFT function is defined as the identity map $\text{ID} : m \to m$, for all $m \in \mathcal{L}_m$.

Table 3.1: *Differences among NTRU-HPS and NTRU-HRSS cryptographic schemes*

	NTRU-HPS	NTRU-HRSS
n	prime, $\langle a \rangle = \mathbb{Z}_n^*$, $a \in \{2, 3\}$	prime, $\langle a \rangle = \mathbb{Z}_n^*$, $a \in \{2, 3\}$
p	3	3
q	power of two, $q/8 - 2 \leq 2n/3$	$2^{3.5+\log_2(n)}$
$\mathcal{L}_f$	$\mathcal{T}$	$\mathcal{T}_+$
$\mathcal{L}_g$	$\mathcal{T}(q/8 - 2)$	$\{\Phi_1 \cdot v \mid v \in \mathcal{T}_+\}$
$\mathcal{L}_r$	$\mathcal{T}$	$\mathcal{T}$
$\mathcal{L}_m$	$\mathcal{T}(q/8 - 2)$	$\mathcal{T}$
$\text{LIFT}(\cdot)$	$m \mapsto m$	$m \mapsto \Phi_1 \cdot (m/\Phi_1 \bmod (p, \Phi_n))$

The recommended parameter sets take $q/8 - 2 \leq 2n/3$. Parameter sets with larger q are advised to replace the $q/8 - 2$ in the definition of $\mathcal{L}_g$ and $\mathcal{L}_m$ with $2\lfloor n/3 \rfloor$. In order to remove probabilistic failures in decryption, $q > (6d + 1)\, p$. The official NTRU-HPS parameter sets [Che+19] are denoted as `ntruhps2048509`, `ntruhps2048677`, `ntruhps4096821`, concatenating the values of q and n.

The provided security levels complying to the NIST proposed rank in Table 1.1, are category 1 for $(q, n) = (2048, 509)$; category 3 for $(q, n) = (2048, 677)$, and category 5 for $(q, n) = (4096, 821)$.

NTRU-HRSS parameter sets

The NTRU-HRSS parameter set takes $p = 3$, $q = 2^{3.5+\log_2(n)}$, and n as a prime number such that both 2 and 3 are generators of $(\mathbb{Z}_n^*, \cdot)$. A uniform sampling procedure is required only for variable-weight polynomials with degree at most $n - 2$, i.e.: $\mathcal{L}_f = \mathcal{T}_+$, $\mathcal{L}_g = \{\Phi_1 \cdot v \mid v \in \mathcal{T}_+\}$, $\mathcal{L}_r = \mathcal{T}$, $\mathcal{L}_m = \mathcal{T}$, while the LIFT function is defined as the map $m \mapsto \Phi_1 \cdot \mathbf{S}_p(\text{LIFT}(m)/\Phi_1)$, for all $m \in \mathcal{L}_m$. The choice of the different LIFT function allowed to drop the requirement of sampling ternary polynomials with fixed weight, although requiring to increase q to maintain the perfect correctness property of the scheme. The single NTRU-HRSS parameter set proposed for standardization [Che+19] is denoted as `ntruhrss701` to point out an instance of the scheme with $n = 701$ providing a level of security equal to category 3.

3.2.2 NTRU DPKE

The NTRU DPKE cryptoscheme is built with the same deterministic structure, regardless of the fact that it uses the NTRU-HPS or NTRU-HRSS parameters, and consists of three algorithms: NTRU.DPKE-KEYGENERATION, NTRU.DPKE-ENCRYPTION, and NTRU.DPKE-DECRYPTION.

Algorithm 1 NTRU.DPKE-KEYGENERATION

Require: None

Ensure: $\mathsf{pk} = h \in \mathbf{R}_q$

$\mathsf{sk} = (f, f_p, h_q, s) \in \mathbf{S}_p \times \mathbf{S}_p \times \mathbf{R}_q \times \{0,1\}^{256}$

1: $\gamma \xleftarrow{\$} \{0,1\}^{320}$
2: $(s, \texttt{str\_f}, \texttt{str\_g}) \leftarrow \text{CSPRNG}(\gamma, \{0,1\}^{8\cdot(n-1)} \times \{0,1\}^{\beta})$
3: $f \leftarrow \text{CSPRNG}(\texttt{str\_f}, \mathcal{L}_f)$
4: $g \leftarrow \text{CSPRNG}(\texttt{str\_g}, \mathcal{L}_g)$
5: $f_p \leftarrow f^{-1} \bmod (p, \Phi_n)$
6: $G \leftarrow p \cdot g$
7: $v \leftarrow (G \cdot f) \bmod (q, \Phi_n)$
8: $v_q \leftarrow v^{-1} \bmod (q, \Phi_n)$
9: $h \leftarrow (v_q \cdot G \cdot G) \bmod (q, \Phi_1\Phi_n)$
10: $h_q \leftarrow (v_q \cdot f \cdot f) \bmod (q, \Phi_1\Phi_n)$
11: **return** $\mathsf{pk} = h$, $\mathsf{sk} = (f, f_p, h_q, s)$

NTRU.DPKE-KEYGENERATION

The key generation algorithm is summarized in Algorithm 1, denoting the computed keypair as $(\mathsf{sk}, \mathsf{pk})$, where the secret key is $\mathsf{sk} = (f, f_p, h_q)$, and the public key is $\mathsf{pk} = h$. The NTRU.DPKE-KEYGENERATION algorithm starts by sampling uniformly the coefficients of the ternary polynomials $f \in \mathcal{L}_f$ and $g \in \mathcal{L}_g$, respectively, by expanding a small high-quality random seed γ, obtained from a True-Random Number Generator (TRNG), into `str_f` and `str_g` using a CSPRNG, and then using the random binary strings with the appropriate ternary polynomial sampling algorithm depending on the cryptographic scheme variant. NTRU uses an instance of SHAKE256 to expand the seed into an arbitrary large output (further details will be provided in subsection 5.2.1). The size β of `str_g` depends on the cryptographic scheme variant, and is $8 \cdot (n-1)$ for NTRU-HRSS or $30 \cdot (n-1)$ for NTRU-HPS. Then, the ternary polynomial $f_p = f^{-1} \bmod (p, \Phi_n)$ and the polynomial with coefficients in $\mathbb{Z}_q$, $f_q = f^{-1} \bmod (q, \Phi_n)$ are determined to be the multiplicative inverse element of f in the ring $\mathbf{S}_p$ and $\mathbf{S}_q$, respectively. The public key $\mathsf{pk} = h$ is a polynomial in $\mathbf{R}_q$ obtained as $h = (p \cdot g \cdot f_q) \bmod (q, \Phi_1\Phi_n)$, while the last component of the private key h_q is pre-computed as $h_q = h^{-1} \bmod (q, \Phi_n)$.

Since the computation of the multiplicative inverse element is an expensive task, as an optimization at algorithmic level the NTRU scheme trades one of such operation for four multiplications:

$$v_q = v^{-1} \bmod (q, \Phi_n) = \left(\frac{1}{(p \cdot g) \cdot f}\right) \bmod (q, \Phi_n) \tag{3.5}$$

$$h = \left(\frac{(p \cdot g) \cdot (p \cdot g)}{p \cdot g \cdot f}\right) \bmod (q, \Phi_1\Phi_n) = \frac{p \cdot g}{f} \bmod (q, \Phi_1\Phi_n) \tag{3.6}$$

Algorithm 2 NTRU.DPKE-ENCRYPTION

Require: $\mathsf{pk} = h \in \mathbf{R}_q$
$(r, m) \in \mathcal{L}_r \times \mathcal{L}_m$
Ensure: $\mathsf{ctx} = c \in \mathbf{R}_q$
1: $m' \leftarrow \text{LIFT}(m)$
2: $c \leftarrow (r \cdot h + m') \bmod (q, \Phi_1\Phi_n)$
3: **return** c

$$h_q = \left(\frac{f \cdot f}{p \cdot g \cdot f}\right) \bmod (q, \Phi_1\Phi_n) = \frac{f}{p \cdot g} \bmod (q, \Phi_1\Phi_n) \tag{3.7}$$

Note that the computation of f_p and h_q, performed during the key generation, could be potentially deferred during the decryption, as they do not take part to the generation of the public key, at cost of a non-negligible hit in latency and throughput during an online key establishment. Moreover, the ring $\mathbf{R}_q$ does not contain the multiplicative inverse of every polynomial. In the (unlikely) case the inverse element does not exist, the NTRU.DPKE-KEYGENERATION is restarted sampling a new seed γ.

NTRU.DPKE-ENCRYPTION

The NTRU DPKE encryption algorithm, shown in Algorithm 2, receives as input the public key $h \in \mathbf{R}_q$, the encryption ephemeral value r, and the message to be encrypted m. While NTRU-HRSS uses both r and m from variable-weight ternary polynomials from $\mathcal{T}$, NTRU-HPS notably requires m to have fixed weight such that $m \in \mathcal{T}(\frac{q}{8} - 2)$.

The encryption function computes the ciphertext $c = (r \cdot h + \text{LIFT}(m)) \bmod (q, \Phi_1\Phi_n)$: a ternary to q-ary polynomial multiplication $(r \cdot h)$ modulo $\Phi_1\Phi_n$ is computed, and subsequently adding the resulting q-ary polynomial to the outcome of the $\text{LIFT}(\cdot)$ function applied to the message m. Depending on the NTRU-HPS or NTRU-HRSS scheme, the function $\text{LIFT}(\cdot)$ maps the input polynomial m in a specific way (see Table 3.1).

NTRU.DPKE-DECRYPTION

The NTRU DPKE decryption procedure, shown in Algorithm 3, receives a private key $\mathsf{sk} = (f, f_p, h_q)$ and a ciphertext $c \in \mathbf{R}_q$. As first step, it retrieves the encrypted message, and the ephemeral randomness r used during the encryption. The validity of the computed values is checked, producing a Boolean value δ indicating if the decryption operation failed. NTRU-HPS and NTRU-HRSS are correct schemes, therefore executing the NTRU.DPKE-DECRYPTION routine on the output of the NTRU.DPKE-ENCRYPTION function always recomputes the correct message and encryption randomness if the correct private key is used.

As first step, an intermediate q-ary polynomial $a = c \cdot f$ is obtained multiplying the ternary polynomial f and the q-ary polynomial c, and reducing the result modulo $\Phi_1\Phi_n$:

$$\begin{aligned} a &= c \cdot f = (r \cdot h + m') \cdot f = r \cdot h \cdot f + m' \cdot f \\ &= r \cdot (p \cdot g \cdot f_q) \cdot f + m' \cdot f = r \cdot p \cdot g + m' \cdot f \in \mathbf{R}_q \end{aligned} \tag{3.8}$$

Algorithm 3 NTRU.DPKE-DECRYPTION

Require: $\mathsf{sk} = (f, f_p, h_q) \in \mathbf{S}_p \times \mathbf{S}_p \times \mathbf{R}_q$
$\mathsf{ctx} = c \in \mathbf{R}_q$
Ensure: $(r, m, \delta) \in \mathcal{L}_r \times \mathcal{L}_m \times \{0, 1\}$
1: **if** $c \not\equiv 0 \bmod (q, \Phi_1)$ **then**
2: **return** $(0, 0, \texttt{false})$
3: $a \leftarrow (c \cdot f) \bmod (q, \Phi_1\Phi_n)$
4: $m \leftarrow (a \cdot f_p) \bmod (p, \Phi_n)$
5: $m' \leftarrow \text{LIFT}(m)$
6: $r \leftarrow ((c - m') \cdot h_q) \bmod (q, \Phi_n)$
7: **if** $(r, m) \in \mathcal{L}_r \times \mathcal{L}_m$ **then**
8: **return** $(r, m, \texttt{true})$
9: **else**
10: **return** $(0, 0, \texttt{false})$

Note that a is guaranteed to have small coefficients always fitting in $\mathbb{Z}_q$, hence no computations modulo q are actually performed.

Since a is already in $\mathbf{R}_q$ and no $\bmod q$ is performed, by computing $a \bmod p$ from the result of Equation 3.8 operation removes the contribution of the ephemeral r, obtaining $f \cdot m$, since m' is transformed to m when $\bmod p$ is applied due to the construction of the LIFT$(\cdot)$ map. This trick allows us to recover m multiplying a by f_p and applying the $\bmod p$ reduction. Once retrieved the message, the ephemeral randomness r is then computed via a simple manipulation derived from the encryption expression $c \leftarrow (r \cdot h + m') \bmod (q, \Phi_1\Phi_n)$: by using the h_q value pre-computed during the NTRU.DPKE-KEYGENERATION, it is possible to isolate r from the that equation obtaining $r = ((c - \text{LIFT}(m)) \cdot h_q) \in \mathbf{R}_q$ via a subtraction followed by a multiplication between two q-ary polynomials.

3.2.3 NTRU KEM

The NTRU KEM employs the NTRU DPKE primitives as components to meet the requirement by NIST to have an IND-CCA2 secure KEM, i.e., one resistant to active attackers, as proven in [SXY18].

We will denote with z^{pkd} the bit-packed encoding of a polynomial z, either ternary or q-ary, used to efficiently store on disk or transfer in a more compact way the polynomials between different systems. The PACK and UNPACK are auxiliary functions encoding the polynomials into a bit string, or viceversa.

NTRU.KEM-KEYGENERATION

The NTRU KEM key generation, reported in Algorithm 4, only performs the compression of the public and private keys obtained from NTRU.DPKE-KEYGENERATION, to reduce the bandwidth required for the transmission of the ciphertext and public key, while also reducing the size of the private key.

Table 3.2: *Size of keys and ciphertext of NTRU-HPS and NTRU-HRSS cryptographic algorithms. Note that also for the DPKE variant the elements are compressed using the appropriate* PACK$(\cdot)$ *function.*

parameters	ntruhps2048509	ntruhps2048677	ntruhps4096821	ntruhrss701
n	509	677	821	701
q	2048	2048	4096	8192
DPKE pk bytes	699	930	1230	1138
DPKE sk bytes	903	1202	1558	1418
DPKE ctx bytes	699	930	1230	1138
KEM pk bytes	699	930	1230	1138
KEM sk bytes	935	1234	1590	1450
KEM ctx bytes	699	930	1230	1138
KEM K bits	256	256	256	256

Algorithm 4 NTRU.KEM-KEYGENERATION

Require: None
Ensure: $\mathsf{pk} = h^{\mathtt{pkd}}$
$\mathsf{sk} = (f^{\mathtt{pkd}}, f_p^{\mathtt{pkd}}, h_q^{\mathtt{pkd}}, s \in \{0,1\}^{256})$
1: $h, (f, f_p, h_q, s) \leftarrow$ NTRU.DPKE-KEYGENERATION()
2: $h^{\mathtt{pkd}} \leftarrow \text{PACK}_q(h)$
3: $f^{\mathtt{pkd}} \leftarrow \text{PACK}_p(f)$
4: $f_p^{\mathtt{pkd}} \leftarrow \text{PACK}_p(f_p)$
5: $h_q^{\mathtt{pkd}} \leftarrow \text{PACK}_q(h_q)$
6: **return** $\mathsf{pk} = h^{\mathtt{pkd}}$, $\mathsf{sk} = (f^{\mathtt{pkd}}, f_p^{\mathtt{pkd}}, h_q^{\mathtt{pkd}}, s)$

NTRU.KEM-ENCAPSULATION

The NTRU KEM encapsulation augments the NTRU DPKE one specifying how the values of the ternary polynomials (r, m) are be sampled, how the secret session key K encapsulated by the KEM is derived from the ciphertext c, and how polynomials should be encoded in a bit-dense format.

Algorithm 5 shows the NTRU.KEM-ENCAPSULATION procedure, that starts by drawing a 320-bit binary string, γ, uniformly at random from a TRNG. This is the only true randomness used by the cryptosystem, which is then expanded by Pseudo-Random Number Generator (PRNG) to an arbitrary amount of bit strings. The random string γ is then employed by a deterministic procedure sampling the encryption ephemeral randomness r and the message m from the appropriate sets defined by either NTRU-HPS or NTRU-HRSS (see Table 3.1). The resulting polynomials are packed, concatenated, and absorbed via a HASH function, and the 256-bit digest represent session key K. NTRU implements the HASH function via the `SHA3-256` instance defined in the SHA-3 standard. The size of the bit string `str_r` is $8 \cdot (n-1)$, whereas the size of `str_m` depends on the cryptographic scheme variant in use: it could be either $8 \cdot (n-1)$ for NTRU-HRSS or $30 \cdot (n-1)$ for NTRU-HPS. Then the NTRU.DPKE-ENCRYPTION routine is called passing (r, m) and the public key h, obtaining the q-ary ciphertext polynomial c, which is packed as $c^{\mathtt{pkd}}$ and returned to the caller together with

Algorithm 5 NTRU.KEM-ENCAPSULATION

Require: $\mathsf{pk} = h^{\mathtt{pkd}}$
Ensure: $\mathsf{ctx} = c^{\mathtt{pkd}}$
$\mathsf{K} \in \{0,1\}^{256}$
1: $h \leftarrow \text{UNPACK}_q(h^{\mathtt{pkd}})$
2: $\gamma \xleftarrow{\$} \{0,1\}^{320}$
3: $(\mathtt{str\_r}, \mathtt{str\_m}) \leftarrow \text{CSPRNG}(\gamma, \{0,1\}^{8\cdot(n-1)} \times \{0,1\}^{\beta})$
4: $r \leftarrow \text{CSPRNG}(\mathtt{str\_r}, \mathcal{L}_r)$
5: $m \leftarrow \text{CSPRNG}(\mathtt{str\_m}, \mathcal{L}_m)$
6: $c \leftarrow \text{NTRU.DPKE-ENCRYPTION}(h, (r, m))$
7: $c^{\mathtt{pkd}} \leftarrow \text{PACK}_q(c)$
8: $r^{\mathtt{pkd}} \leftarrow \text{PACK}_p(r)$
9: $m^{\mathtt{pkd}} \leftarrow \text{PACK}_p(m)$
10: $\mathsf{K} \leftarrow \text{HASH}(r^{\mathtt{pkd}} \| m^{\mathtt{pkd}})$
11: **return** $c^{\mathtt{pkd}}, \mathsf{K}$

the derived session key K.

NTRU.KEM-DECAPSULATION

The NTRU KEM decapsulation procedure (NTRU.KEM-DECAPSULATION, Algorithm 6), uses the private key $\mathsf{sk} = (f^{\mathtt{pkd}}, f_p^{\mathtt{pkd}}, h_q^{\mathtt{pkd}}, s)$, composed by two ternary polynomials f, f_p and the q-ary polynomial h_q, and the q-ary ciphertext polynomial c, all in their packed format. Initially the algorithm unpacks all the inputs and invokes the NTRU.DPKE-DECRYPT algorithm, obtaining (r, m) and a failure flag. Two different 256-bit session keys are derived, K_1 and K_2. The former is obtained hashing the concatenation of r and m after being packed, similarly to what is performed in the encapsulation, while K_2, also known as the *implicit rejection key*, is composed as the digest of the HASH function obtained absorbing the received ciphertext $\text{PACK}_q(c)$ and the value s generated during NTRU.DPKE-KEYGENERATION and bounded to the keypair $(\mathsf{sk}, \mathsf{pk})$.

The implicit rejection key is used to not allow to establish the requested secure communication and is returned only in case a ciphertext manipulation is detected, either in case of decryption failure ((m, r) should belong to the appropriate set defined by NTRU-HPS or NTRU-HRSS), or if $c \equiv 0 \bmod (q, \Phi_1)$. In case of decryption success, K_1 is guaranteed to match the session key on the sender side. Both K_1 and K_2 are always computed to avoid an information leakage on the decryption failure via timing side channel.

3.3 Expressing NTRU hardness as lattice problems

The connection linking NTRU to the problems over a lattice is not immediate. How can we map the key recovery attack to SVP? Why is decryption attack reducible to solving the CVP?

Algorithm 6 NTRU.KEM-DECAPSULATION

Require: $\mathsf{sk} = (f^{\mathtt{pkd}}, f_p^{\mathtt{pkd}}, h_q^{\mathtt{pkd}}, s)$
$\mathsf{ctx} = c^{\mathtt{pkd}}$
Ensure: $\mathsf{K} \in \{0,1\}^{256}$
$f \leftarrow \text{UNPACK}_q(f^{\mathtt{pkd}})$
$f_p \leftarrow \text{UNPACK}_q(f_p^{\mathtt{pkd}})$
$h_q \leftarrow \text{UNPACK}_q(h_q^{\mathtt{pkd}})$
$c \leftarrow \text{UNPACK}_q(c^{\mathtt{pkd}})$
$(r, m, \delta) \leftarrow$ NTRU.DPKE-DECRYPTION$(f, (f_p, h_q, c))$
$r^{\mathtt{pkd}} \leftarrow \text{PACK}_p(r)$
$m^{\mathtt{pkd}} \leftarrow \text{PACK}_p(m)$
$\mathsf{K}_1 \leftarrow \text{HASH}(r^{\mathtt{pkd}} \| m^{\mathtt{pkd}})$
$\mathsf{K}_2 \leftarrow \text{HASH}(s \| c^{\mathtt{pkd}})$
if $\delta = \mathtt{true}$ **then**
    $\mathsf{K} = \mathsf{K}_1$
else
    $\mathsf{K} = \mathsf{K}_2$
return K

The Convolution Modular Lattice $\mathcal{L}$ associated to the vector $\mathbf{h}$, composed by the coefficients of the polynomial h and modulus q, is the $2n$ dimensional lattice with basis given by the rows of the following matrix:

$$\mathcal{L} = \text{ROWSPAN}\left(\left[\begin{array}{cccc|cccc} 1 & 0 & \dots & 0 & h_0 & h_1 & \dots & h_{n-1} \\ 0 & 1 & \dots & 0 & h_{n-1} & h_0 & \dots & h_{n-2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 & h_1 & h_2 & \dots & h_0 \\ \hline 0 & 0 & \dots & 0 & q & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & q & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & q \end{array}\right]\right) \tag{3.9}$$

or, for compactness, $\mathcal{L} = \text{ROWSPAN}\left(\begin{bmatrix} \mathbf{I_n} & \mathbf{H} \\ \mathbf{0} & q\mathbf{I_n} \end{bmatrix}\right) = \text{ROWSPAN}\left(\begin{bmatrix} x^0 & h \\ 0 & qx^0 \end{bmatrix}\right)$

The defined matrix spawning the lattice uses entirely public components: the public key h, the modulus q and the ring size n. The vectors of the basis $\mathbf{B}$ are the rows of the matrix, thus the lattice dimension is $2n$.

$\mathcal{L}$ contains all the vectors $(\mathbf{a}, \mathbf{b})$, obtained taking the coefficients of $a, b \in \mathbf{R}_q$ lifted in $\mathbb{Z}[x]/(\Phi_1\Phi_n)$ such that $a \cdot h = b$ for an invertible $a \in \mathbf{R}_q$:

$$\mathcal{L} = \left\{(\mathbf{a}, \mathbf{b}) \in \mathbb{Z}^{2n} : \mathbf{a} \cdot \mathbf{h} \equiv \mathbf{b} \,(\mathrm{mod} q)\right\} \tag{3.10}$$

NTRU public/private key pairs are constructed via $\mathbf{f} \cdot \mathbf{h} \equiv \mathbf{g} \,(\mathrm{mod} q)$, with small f and g. This implies that the NTRU lattice contains the short vector $[\mathbf{f}, \mathbf{g}]$, since $f \cdot h - g =$

$q \cdot u$ for some $\mathbf{u} \in \mathbb{Z}^n$

$$[\mathbf{f}, -\mathbf{u}] \begin{bmatrix} \mathbf{I_n} & \mathbf{H} \\ \mathbf{0} & q\mathbf{I_n} \end{bmatrix} = [f, f \cdot h - q \cdot u] = [\mathbf{f}, \mathbf{g}] \tag{3.11}$$

Key recovery attack

The public key is $h \equiv f^{-1} \cdot g$. Building the matrix requires only the public key, solving the SVP will either yield $[\mathbf{f}, \mathbf{g}]$ or an SVP good short vector to be used as a private key.

It can be easily proved that f and g used to generate the public key belong to the NTRU lattice, and are by construction short vectors (small polynomials). Solving the SVP and finding any valid short vector leads to a new valid private key and thus it is possible to impersonate one agent.

The ciphertext $c = r \cdot h + m' \,(\mathrm{mod} q)$ is the vector

$$[\mathbf{0}, \mathbf{c}] = [\mathbf{0}, \mathbf{r} \cdot \mathbf{h} + \mathbf{m'} \,(\mathrm{mod} q)] = [\mathbf{r}, \mathbf{r} \cdot \mathbf{h} \,(\mathrm{mod} q)] + [-\mathbf{r}, \mathbf{m'}] \tag{3.12}$$

where the vector $[\mathbf{r}, \mathbf{r} \cdot \mathbf{h} \,(\mathrm{mod} q)] \in \mathcal{L}$, and $[-\mathbf{r}, \mathbf{m'}]$ is a short error by construction

Decryption attack

First we should note that the ciphertext is a generic point in the vector space and (with high probability) not belonging to the NTRU lattice. Thus, we can rewrite this vector as the closest lattice vector plus a small error.

$$\mathbf{e} = [\mathbf{0}, \mathbf{c}] + [\mathbf{r}, -\mathbf{m'}] \tag{3.13}$$

where $\mathbf{e} \in \mathcal{L}$ is the closest vector to $[\mathbf{0}, \mathbf{c}]$.

Being able to easily solve the CVP, hence compute $\mathbf{e}$ from $[\mathbf{0}, \mathbf{c}]$, the attacker can easily find r and m required to generate the session key K:

$$[\mathbf{r}, -\mathbf{m'}] = \mathbf{e} - [\mathbf{0}, \mathbf{c}] \tag{3.14}$$

More details on that topic are available in [Car] and [Sil].

CHAPTER 4

Code-based cryptography

The integrity of data transmission over a noisy or unreliable communication channel is a well known problem that was tackled since the introduction of telecommunication lines and radio communications, particularly in case of unidirectional communication channels (e.g. broadcast television system) or where the cost for re-transmission is extremely expensive due to a long-latency (e.g. space communication) and possibly combined with inefficient re-transmission protocols.

To handle the errors introduced during the data transmission, a solution involves the encoding of the message with some redundant information to detect, and optionally correct, the introduced errors at cost of a reduced bandwidth. In the first case, parity check schemes can detect the errors, provided that the number of introduced errors are below some bound.

Error-Correcting Codes (ECCs) were invented halfway in the 20th century and provide a more versatile solution, as they can:

- both detect and correct the transmission errors
- deal with different type of error distribution (burst of errors or uniformly distributed errors)
- improve the error-correcting capability in case of erasures (the the location of the errors are known)
- work on fixed-sized blocks of data (block codes) or streams (convolutional code)

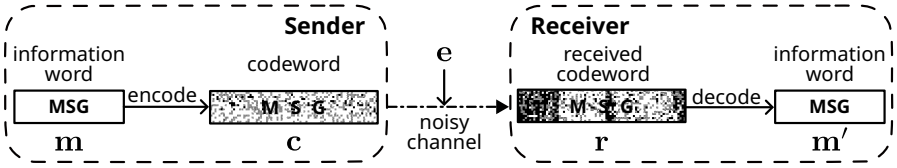


Figure 4.1: *Working principle of an Error Correction Code. The sender encodes a message* $\mathbf{m}$ *using a code* $\mathcal{C}$ *producing the codeword* $\mathbf{c}$ *embedding some redundant information, and sends it over a noisy channels. The receiver obtains a vector* $\mathbf{r}$*, also known as senseword, potentially corrupted by an error* $\mathbf{e}$*, and tries to decode it obtaining a message* $\mathbf{m}'$*. If the error* $\mathbf{e}$ *introduced by the noisy channel can be detected and corrected by the code* $\mathcal{C}$*, then* $\mathbf{m} \equiv \mathbf{m}'$*.*

- offer various trade-offs in terms of bandwidth consumption and error correcting capability achieving transmission rates close to the Shannon limit of the communication channel capacity

Figure 4.1 shows the working principle of a generic error correction code applied for the communication over a noisy channel.

A notorious block error correction code introduced in 1960 and still finding numerous applications in consumer products such as optical disks, QR codes and bar codes, but also in highly complex technologies such as satellite communications, Digital Video Broadcasting (DVB) and Digital Subscriber Line (DSL), is the Reed-Solomon (RS) code. The essence of the original proposal of the code was that the message to be transmitted needs to be transformed in a polynomial with maximum degree k, and the transmitted information are a set of $n > k$ evaluated points in the message polynomial at locations known both to the sender and the receiver. The decoder algorithm, however, was not practical for large block sizes, limiting the applicability of such solution. Shortly, it was conceived that using a Bose-Chaudhuri-Hocquenghem (BCH) encoding scheme lead to use more practical decoders such as the Petterson-Gorenstein-Zierler (PGZ) and the Berlekamp-Massey (BM) algorithms.

A $[n, k, d]$ linear forward ECC $\mathcal{C}$ over $\mathbb{F}_q$ is a subspace of $\mathbb{F}_q^n$ having dimension k, and minimum Hamming distance d among any two (row) vectors in $\mathcal{C}$. The value $r = n - k$ is called redundancy, and $R = \frac{k}{n}$ the rate of the code. The *generator matrix* $\mathbf{G} \in \mathbb{F}_q^{k \times n}$ of the code $\mathcal{C}$ allows to generate all the 2^k *codewords* $\mathbf{c} \in \mathcal{C} \subseteq \mathbb{F}_q^n$ from the *information words* $\mathbf{m} \in \mathbb{F}_q^k$:

$$\mathcal{C} = \left\{ \mathbf{mG} \mid \mathbf{m} \in \mathbb{F}_q^k \right\} \tag{4.1}$$

The generator matrix $\mathbf{G}$ is full-rank, and the rows forming the basis of the vector space are valid codewords. The *parity-check matrix* of the code $\mathcal{C}$ is the matrix $\mathbf{H} \in \mathbb{F}_q^{(n-k) \times n}$ such that

$$\mathcal{C} = \left\{ \mathbf{v} \in \mathbb{F}_q^n \mid \mathbf{H}\mathbf{v}^\top = \mathbf{0} \right\} \tag{4.2}$$

We denote as *syndrome* $\mathbf{s} \in \mathbb{F}_q^{n-k}$ of $\mathbf{v} \in \mathbb{F}_q^n$ with respect to $\mathbf{H}$ the result of $\mathbf{H}\mathbf{v}^\top$ for a generic $\mathbf{v} \in \mathbb{F}_q^n$. The parity-check matrix is the generator matrix of the *dual code* $\mathcal{C}^\perp$ with rank $n-k$ containing all the vectors orthogonal to the codewords in $\mathcal{C}$, and if $k > \frac{n}{2}$,

it is used to represent the code in a more compact way. A generator matrix formed as $\mathbf{G} = [\mathbf{I_k}|\mathbf{M}]$ is in its *systematic form*, and $\mathbf{H} = \left[-\mathbf{M}^\top \mid \mathbf{I_{n-k}}\right]$ is its corresponding parity-check matrix.

The encoding procedure is a map $\textsc{Encode} : \mathbb{F}_q^k \mapsto \mathbb{F}_q^n$, and the algebraic decoding procedure correcting up to $t = \lfloor \frac{d-1}{2} \rfloor$ errors is a map $\textsc{Decode}_t : \mathbb{F}_q^n \mapsto \mathbb{F}_q^k$. If a message $\mathbf{m} \in \mathbb{F}_q^k$ is transmitted over a channel encoded in the codeword $\mathbf{c} \in \mathcal{C} \subseteq \mathbb{F}_q^n$, the channel noise can be represented as an error $\mathbf{e} \in \mathbb{F}_q^n$ combined to the transmitted codeword. The received data $\mathbf{r} = \mathbf{c} + \mathbf{e}$ can be successfully recovered iff $\|\mathbf{e}\| \leq t$:

$$\textsc{Decode}_t\left(\textsc{Encode}\left(\mathbf{m}\right) + \mathbf{e}\right) = \mathbf{m},\ \mathbf{m} \in \mathbb{F}_q^k \tag{4.3}$$

A $[n, k, d]$ linear ECC is called a *perfect ECC* if all the codewords have the maximum possible d for a given code length n and dimension k, with the RS code being an example of a perfect ECC.

The Gilbert-Varshamov (GV) bound provides a lower bound on the maximum possible rate $R = \frac{k}{n}$ of a error-correcting code $\mathcal{C}$ over $\mathbb{F}_q$ for a given minimum Hamming distance d such that $0 \leq \frac{d}{n} < 1 - \frac{1}{q}$:

$$R \geq 1 - \mathrm{H}_q\left(\frac{d}{n}\right) \tag{4.4}$$

Where H_q is the q-ary entropy function defined as:

$$\mathrm{H}_q\left(x\right) = x\log_q\left(q-1\right) - x\log_q x - \left(1-x\right)\log_q\left(1-x\right) \tag{4.5}$$

4.1 Syndrome Decoding Problem

Decoding a general linear code is a NP-hard problem known as Generic Decoding Problem (GDP), but there exist efficient decoders for some linear code having some useful structure. Provided that an obfuscated generator matrix of such code is indistinguishable from a random code, the GDP can be used as the core for PKC schemes:

KeyGeneration Let $\mathbf{G} \in \mathbb{F}_q^{k\times n}$ be a random generator matrix of the q-ary $[n, k, d]$ linear code $\mathcal{C}$ for which an efficient decoding algorithm exists. Generate a random invertible matrix $\mathbf{S} \in \mathbb{F}_q^{k\times k}$, and a random permutation matrix $\mathbf{P} \in \mathbb{F}_2^{n\times n}$. The public key is the obfuscated generator matrix $\mathsf{pk} = \mathbf{SGP} = \hat{\mathbf{G}}$, and the private key is $\mathsf{sk} = (\mathbf{S}, \mathbf{G}, \mathbf{P})$.

Encryption Generate a random message $\mathbf{m} \in \mathbb{F}_q^k$ and the corresponding codeword $\mathbf{r} \in \mathbb{F}_q^n$ using $\hat{\mathbf{G}}$, and corrupt it with an error $\mathbf{e}$ of low weight $\|\mathbf{e}\| = t$. The ciphertext is $\mathsf{ctx} = \mathbf{m}\hat{\mathbf{G}} + \mathbf{e} = \mathbf{r}$.

Decryption Compute $\mathbf{r}\mathbf{P}^{-1} = (\mathbf{m}\hat{\mathbf{G}} + \mathbf{e})\mathbf{P}^{-1} = (\mathbf{mS})\mathbf{G} + \mathbf{e}\mathbf{P}^{-1}$, and use the efficient decoder algorithm to determine $\mathbf{mS}$, as $\mathbf{e}\mathbf{P}^{-1}$ maintains weight t due to $\mathbf{P}$ being a permutation matrix. Finally, retrieve the original message $\mathbf{m} = (\mathbf{mS})\mathbf{S}^{-1}$

The search or computational GDP (C-GDP) asks to find $\mathbf{m}$ from $\mathbf{r}$ and $\hat{\mathbf{G}}$. Fixing the code length n and dimension k, the number of solutions in a random code on average are $\binom{n}{t}|\mathcal{C}|/q^n = \binom{n}{t}/q^{n-k}$, and there is exactly one solution if $t < q^{n-k}$.

In 1978 McEliece chose $\mathcal{C}$ from the binary Goppa codes family as there exist efficient decoders and their number grows exponentially in the length of the code n, contrary to RS codes, to avoid an easy guess by the attacker of the random instance of $\mathbf{G}$. The scheme envisioned by McEliece has resisted cryptanalysis so far, with the most effective attacks using information-set decoding algorithms.

In 1986 Niederreiter applied the same idea for a cryptosystem based on the Syndrome Decoding Problem (SDP).

KeyGeneration Let $\mathbf{G} \in \mathbb{F}_q^{k\times n}$ be a random generator matrix of the q-ary $[n, k, d]$ linear code $\mathcal{C}$ for which an efficient decoding algorithm exists, and $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$ its parity-check matrix. Generate a random invertible matrix $\mathbf{S} \in \mathbb{F}_q^{(n-k)\times(n-k)}$, and a random permutation matrix $\mathbf{P} \in \mathbb{F}_2^{n\times n}$. The public key is the obfuscated parity-check matrix $\mathsf{pk} = \mathbf{SHP} = \hat{\mathbf{H}}$, and the private key is $\mathsf{sk} = (\mathbf{S}, \mathbf{H}, \mathbf{P})$.

Encryption Generate a random message $\mathbf{m}$ and encode in $\mathbf{e} \in \mathbb{F}_q^n$ such that $\|\mathbf{e}\| = t$, then compute the syndrome $\mathbf{s} \in \mathbb{F}_q^{n-k}$ using $\hat{\mathbf{H}}$. The ciphertext is $\mathsf{ctx} = \hat{\mathbf{H}}\mathbf{e}^\top = \mathbf{s}$.

Decryption Compute $\mathbf{S}^{-1}\mathbf{s} = \mathbf{HPe}^\top$, and apply the efficient syndrome decoding algorithm for $\mathcal{C}$ to recover $\mathbf{Pe}^\top$, and then recompute $\mathbf{m}$ from $\mathbf{P}^{-1}(\mathbf{Pe}^\top)$

The search or computational SDP (C-SDP) asks to find the error $\mathbf{e}$ with small weight from the syndrome $\mathbf{s}$ and the parity-check matrix $\hat{\mathbf{H}}$. If the weight $\|\mathbf{e}\| = t$ is large enough, there are multiple solutions which are easy to find. Given that $\mathcal{C}$ is a random linear code, it lies on the GV bound with very high probability, therefore it is possible to determine the relative distance $\delta_{GV} = d_{GV}/n$ of the code starting from Equation 4.4, and consequently find the maximum value of t for which there is an unique solution:

$$t \leq \frac{d_{GV} - 1}{2} = \frac{\delta_{GV} n - 1}{2} = \frac{\mathrm{H}_q^{-1}(1 - R)\, n - 1}{2} \tag{4.6}$$

The decisional SDP (D-SDP) only asks if there exist $\mathbf{e}$, with $\|\mathbf{e}\| \leq t$, such that $\hat{\mathbf{H}}\mathbf{e}^\top = \mathbf{s}$, which [BMT78] proved to be a NP-complete problem under the name of Coset Weights Problem. As the decisional variant of the presented problem can be trivially reduced to its search version, it implies that the C-SDP is at least as hard as D-SDP, and consequently it is a NP-hard problem.

There are many cryptographic schemes based on the Niederreiter framework, most notably Classic McEliece which is concurring in the standardization of a code-based scheme in the NIST PQC standardization process. As the GDP and the SDP can be reduced one into the other in both directions, the major advantage of a Niederreiter scheme with respect to McEliece schemes is the size reduction of the private key and the ciphertext when $k > \frac{n}{2}$, hence when the redundancy information $r = n - k$ is

larger than the message information k. Another technique to further reduce the size of the public key consists in computing the systematic form $[\mathbf{I_{n-k}} \mid \mathbf{T}]$ of the parity-check matrix $\hat{\mathbf{H}}$, whenever is possible, and consequently transmitting only $\mathbf{T} \in \mathbb{F}_q^{(n-k)\times k}$ and omitting the identity matrix $\mathbf{I_{n-k}}$. Considering that the computational cost of Gaussian elimination used to determine the matrix systematic form is $O(n^3)$, the reduction of the network traffic offered by this optimization comes at a cost of an increased keygen latency.

4.1.1 Quasi-Cyclic codes

For appropriate dimensions of k and k able to guarantee practical security against cryptanalysis, the public key size of McEliece and Niederreiter schemes is in the order of few MiB, which is $\approx 1000\times$ larger than EC-based public key schemes currently in use. For that reason, such schemes are primarily used to produce static keys, that are intended to be re-used for relatively long period of time, e.g. Virtual Private Network (VPN), and where the performance of the Key Generation and the size of the public key do not play a fundamental role. By contrast, ephemeral keys are generated for a single use in a cryptographic process, guaranteeing forward security in the cryptographic protocols making use of it, but are strongly influenced on key generation latency and size of the public key. Notably, the TLS `v1.3` protocol, which is widely used nowadays to establish secure connections between a web browser and a HTTP server, mandates the use of ephemeral keys for each key exchange.

Quasi-Cyclic (QC) codes introduced in [Gab05] aim to produce public keys having a size of just $O(n)$ in the code's length n by exploiting the quasi-cyclicity of BCH ECCs that allow to transmit only few rows of the generator or parity-check matrix and still be able to derive the whole matrix.

Let $\mathcal{C}$ be a $[n, k, d]$ binary quasi-cyclic code of order $s \mid n$. Its generator matrix $\mathbf{G}$ is composed of $r \times r$ circulant blocks $\mathbf{A}_i \in \mathbb{F}_2^{u\times s}$, with $r = \frac{n}{s}$, $0 \leq i < r$ and $ru = k$, such that every column (or row) is a cyclic rotation of the previous column (or row) by one cell:

$$\mathbf{G} = \begin{bmatrix} \mathbf{A}_0 & \mathbf{A}_{r-1} & \dots & \mathbf{A}_1 \\ \mathbf{A}_1 & \mathbf{A}_0 & \dots & \mathbf{A}_2 \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{r-1} & \mathbf{A}_{r-2} & \dots & \mathbf{A}_0 \end{bmatrix} = \text{ROT}([\mathbf{A}_0, \mathbf{A}_1, \dots, \mathbf{A}_{r-1}]) \tag{4.7}$$

As a consequence, the generator matrix $\mathbf{G} \in \mathbb{F}_2^{k\times n}$ can be reconstructed from the r smaller matrices $\mathbf{A}_i \in \mathbb{F}_2^{u\times s}$, having only $\frac{nk}{r}$ size in total. Let $\mathbf{c} = [\mathbf{c}_0, \mathbf{c}_1, \dots, \mathbf{c}_{s-1}]$ be a codeword of $\mathcal{C}$, then a simultaneous rotation of every $\mathbf{c}_i$ by the same amount of positions is also a codeword of $\mathcal{C}$.

The construction of $\mathbf{G}$ can be generalized to relax the $r \mid k$ condition, and therefore both McEliece and Niederreiter schemes can be instantiated with a QC code, considering that the permutation matrix $\mathbf{P} \in \mathbb{F}_2^{r\times r}$ is applied to each circulant block $\mathbf{A}_i$. Notice

that there are no complexity results proving the hardness of decoding random quasi-cyclic codes on average or worst cases, but it is still believed by the community to be hard, and the current best attacks are the same of generic codes.

In recent times there are numerous proposal of cryptographic schemes relying on quasi-cyclic codes, some featuring low-density or moderate-density parity-check matrix as a further step to decrease the public key size. Most notably some of them are KEMs proposals in the NIST PQC standardization process LEDAkem [Bal+18], BIKE [Ara+22], and HQC [Agu+24a].

4.1.2 Restricted error vectors

The Restricted Syndrome Decoding Problem (R-SDP) is an NP-complete problem similar to the SDP where the coefficients of the error vector $\mathbf{e}$ belong to the cyclic subgroup $\langle g \rangle = E \subseteq \mathbb{F}_p^*$ generated by the element $g \in \mathbb{F}_p^*$ of multiplicative order z, with $z < p$ prime numbers. It was initially presented in [Bal+20] for the $z = 2$ case, and later generalized for any z in [Bal+24b], with the proof for the NP-completeness being reported in [Weg+24]. Given a parity-check matrix $\mathbf{H} \in \mathbb{F}_p^{(n-k)\times n}$, a syndrome $\mathbf{s} \in \mathbb{F}_p^{n-k}$ and a restricted group $\langle g \rangle = E \subseteq \mathbb{F}_p^*$ with $g \in \mathbb{F}_p^*$ having prime order z, the search variant of the R-SDP asks to find a vector $\mathbf{e} \in E^n$ such that $\mathbf{s} = \mathbf{e}\mathbf{H}^\top$, while the decision variant of the R-SDP asks if there exists such $\mathbf{e}$. With this restriction of the ambient space of the error vector, the uniqueness of the solution can be guaranteed for any weight of the error vector when $\log_2(z) \leq (1 - R)\log_2(p)$, and the best solvers are the same ones for the SDP. Note that when $z = p - 1$, the R-SDP is similar to the SDP, whereas when $z = 1$, the R-SDP is close to the Subset Sum Problem (SSP) over finite fields.

Another way to further restrict the valid solution is to take the error vector $\mathbf{e}$ from the subgroup $G \subset E^n$ such that $|G| = z^m$ and $|E^n| = z^n$, where n is the dimension of the ECC and $m < n$. Note that the structure of G may be leveraged to mount more efficient attacks for the Restricted Syndrome Decoding Problem in the subgroup G (R-SDP(G)) variant of the problem, so the subgroup G must be carefully chosen to thwarts structural weaknesses. Given a parity-check matrix $\mathbf{H} \in \mathbb{F}_p^{(n-k)\times n}$, a syndrome $\mathbf{s} \in \mathbb{F}_p^{n-k}$ and a restricted group $G = \langle \{\mathbf{a_1}, \ldots, \mathbf{a_m}\} \rangle$, for $\mathbf{a_i} \in E^n$, the decisional R-SDP(G) asks if there exits a vector $\mathbf{e} \in G$ such that $\mathbf{e}\mathbf{H}^\top = \mathbf{s}$. When $m \log_p(z) \leq (1 - R)n$, it is guaranteed that only one solution exists.

Digital signatures can be constructed by leveraging a Zero-Knowledge (ZK) identification protocol where a prover $\mathcal{P}$ tries to convince a verifier $\mathcal{V}$ that it knows a particular secret that verifies some public statement, while not revealing any part of its secret during the process. During the process, the two actors exchange some messages, most of the times in an interactive manner. A prover $\mathcal{P}$ starts the protocol by making a *commitment* to its sk. Afterwards, the verifier $\mathcal{V}$ sends a random *challenge* asking the prover $\mathcal{P}$ to demonstrate the knowledge of the secret key in a follow-up *response* message. This challenge-response mechanism can be re-iterated multiple times if necessary. A ZK protocol must satisfy the following properties:

Completeness a honest prover $\mathcal{P}$ using a secret key sk and following the protocol always convinces a verifier $\mathcal{V}$.

Soundness a cheating prover $\mathcal{P}'$ can convince a verifier $\mathcal{V}$ only with a small probability $\varepsilon < 1$, called *soundness error*. To boost the soundness, the prover can perform t protocol executions using the same secret key sk, resulting in a overall soundness error $\approx \varepsilon^t$

Zero-knowledge starting from the transcript of the messages exchanged between $\mathcal{P}$ and $\mathcal{V}$, no one can learn any information of the sk employed by $\mathcal{P}$.

The first ZK protocol based on the SDP for a binary code was presented in [Ste93], and later generalized for q-ary ECC in [CVA10]. More recently, a ZK identification protocol based on R-SDP and R-SDP(G) problems was presented in [Bal+24b], allowing to significantly reduce the signature size.

The Fiat-Shamir (FS) framework [FS86] can create a signature scheme from a ZK protocol by making its execution non-interactive via a one-way pseudo-random function to deterministically simulate the actions between the prover $\mathcal{P}$ and the verifier $\mathcal{V}$ starting from the message to sign and all the messages exchanged in the previous steps of the ZK protocol. The transcript of the protocol execution is the resulting signature to be verified by emulating all the steps of the verifier $\mathcal{V}$. A malicious prover can repeatedly simulate the ZK protocol execution tweaking the exchanged messages to forge valid signatures, requiring on average ε^{-1} attempts, and achieving the Existential Unforgeability under Chosen Message Attacks (EUF-CMA) security. CRYSTALS-Dilithium is an example of a FS-based digital signature using the R-LWE underlying hard problem that was selected as by NIST and standardized in *FIPS 204* under the name of ML-DSA. Among the proposals in the additional NIST standardization effort of digital signatures that passed the first evaluation barrier, CROSS represents a promising FS-based schemes employing a ZK identification scheme relying on the hardness of the R-SDP or R-SDP(G), and showing a moderate signature size while having low signing and verification runtime. In comparison, the recently introduced Multy-Party Computation in-the-Head (MPCitH) technique is an alternative way to construct zero-knowledge proofs. The idea is to simulate a secure multi-party computation protocol "inside the head" of a single prover $\mathcal{P}$, without actually distributing the computation across multiple real parties holding part of the sk. The advantage lies in shorter signature sizes, although requiring a non-trivial computational overhead for both the signer and the verifier $\mathcal{V}$.

4.2 Hamming Quasi-Cyclic

Hamming Quasi-Cyclic (HQC) is a code-based candidate in the NIST PQC standardization process that employs two different linear codes: a double circulant ($s = 2$) public quasi-cyclic random code that ensures the scheme's security, and a fixed public code with high error correction capacity and an efficient decoding algorithm to encode the plaintext with the high error tolerance required by the scheme. Differently from

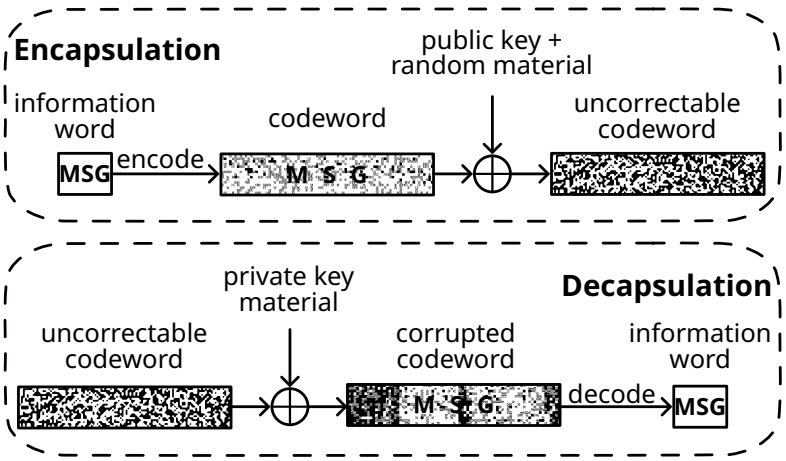


Figure 4.2: *Overview of HQC's working principle and use of the fixed public code*

McEliece and Niederreiter schemes, HQC does not need an efficient decoder for the quasi-cyclic code, and for that reason a random code is sampled directly without obfuscating a known good code.

The working principle by which HQC allows to communicate a session key between two agents A and B, in line with what was presented during the introduction to KEM in Figure 1.1, is depicted in Figure 4.2. B starts by generating a a key pair and sharing the public key with A. Having the public key of B, A encodes a fully random plaintext from which the session key will be derived, with the public and highly efficient error correcting code. Afterwards, he corrupts the resulting codeword way beyond the error-correcting capability of the fixed public code by employing both the public key and some randomness generated deterministically from the message, obtaining the ciphertext, i.e, the encapsulated key. The ciphertext is sent to B, while also deriving the private shared session key from the message. Only B, which is in possession of the private key corresponding to the public key employed by A is able to remove a large amount of the introduced corruptions of the codeword, reducing the corruption to a point where the public error correction code allows to correct the remaining errors. HQC employs the transformation described by Hofheinz-Hövelmanns-Kiltz (HHK) in [HHK17] to achieve resistance against active attackers. Informally, this requires that B should be able to validate if the recovered message is actually the one sent by A: this is done in practice through a re-encryption of the message performed by B after the decryption, and a comparison with the received ciphertext.

In the following, the structure of the HQC.KEM is detailed, i.e., the cryptographic primitive obtained applying the HHK transformation to the HQC Probabilistic Public-Key Encryption (PPKE) (HQC.PPKE).

4.2.1 Algebraic structures and parameters sets

Let $\mathbb{F}_2$ be the binary finite field, and $\mathbf{R}$ be the polynomial ring $\mathbb{F}_2[x]/\langle x^p - 1\rangle$. To thwart mathematical attacks based on the ring structure, HQC picks p as a prime number such that $p > 3$ and $\mathrm{ord}_2(p) = p - 1$, determining that $x^p - 1 \in \mathbb{F}_2[x]$ would only admit two irreducible factors $(\mathrm{mod}\ p)$.

A polynomial $a \in \mathbf{R}$ has a Hamming weight $\omega(a)$ if it has $\omega(a)$ non-zero binary coefficients. The set $\mathbf{R}_w \subset \mathbf{R}$ contains all polynomials in $\mathbf{R}$ having Hamming weight equal to w, $w \geq 1$. Operatively, each polynomial $a = a_0 + a_1 x + \ldots + a_{p-1}x^{p-1} \in \mathbf{R}$ can also be considered as a p-dimensional binary vector composed by its coefficients $\mathbf{a} = [a_0, a_1, \ldots, a_{p-1}] \in \mathbb{F}_2^p$.

HQC uses a random quasi-cyclic $[2p, p, d]$ code with a public parity-check matrix $\mathbf{H} = [\mathbf{I_p} \mid \mathrm{ROT}(h)]$, where $\mathbf{I_p}$ is the $p \times p$ identity matrix, while $\mathrm{ROT}(h)$ is the $p \times p$ matrix obtained from the juxtaposition of h and a sequence of columns, each of which is obtained through a vertical rotation of the previous one by one coefficient. The public efficiently decodable code is generated by the concatenation of a shortened RS $[n_e, k_e, d_e]$ (external) code with a duplicated Reed-Muller (RM) $[n_i, k_i, d_i]$ (internal) code, producing a concatenated code with length, dimension and minimum distance of $[n_e n_i, k_e k_i, d_e d_i]$. We will refer to the Reed-Muller/Reed-Solomon (RM/RS) generator matrix of this code as $\mathbf{G}$.

HQC employs the SHAKE256 algorithm from the SHA-3 NIST standard [PM15] to implement the CSPRNG (further details will be provided in subsection 5.2.1).

Moreover, the KEM construction of HQC makes use of two different hash functions, HASH_G and HASH_K, that for efficiency reasons are both based on SHAKE256 and differentiated by a single byte absorbed as the last element, providing a domain separation to securely employ the same cryptographic primitive for different purposes.

4.2.2 HQC PPKE

HQC.PPKE-KeyGeneration

The PPKE key generation procedure, presented in Algorithm 7, starts by sampling two random seeds γ and φ of 320 bits each. φ is used to sample the random public polynomial h from $\mathbf{R}$, that in turn generates the parity-check matrix of a random QC code $\mathbf{H} = [\mathbf{I_p}, \mathrm{ROT}(h)]$. The seed γ is used to sample the random secret polynomials $x, y \in \mathbf{R}$, both with fixed Hamming weight $\omega(x) = \omega(y) = w$. s is the syndrome polynomial of the pair (x, y) in the random quasi-cyclic code defined by $\mathbf{H}$, $s \leftarrow x + h \cdot y$, i.e., $\mathbf{s} = \mathbf{H}[\mathbf{x} \mid \mathbf{y}]^\top$. The outputs are the public key (φ, s) and the private key γ.

HQC.PPKE-Encryption

The encryption procedure (Algorithm 8) receives as input the public key (φ, s), the random message $\mathbf{m}$ to be transmitted encoded with $k \in \{128, 192, 256\}$ bits, and a 512-bit public salt θ employed to randomize the ciphertext and achieve the IND-CPA

Algorithm 7 HQC.PPKE-KEYGENERATION

Require: None
Ensure: $\mathsf{pk} = (\varphi \in \{0,1\}^{320}, s \in \mathbf{R})$
$\mathsf{sk} = \gamma \in \{0,1\}^{320}$
1: $(\gamma, \varphi) \xleftarrow{\$} \{0,1\}^{320} \times \{0,1\}^{320}$
2: $h \leftarrow \text{CSPRNG}(\varphi, \mathbf{R})$
3: $(x, y) \leftarrow \text{CSPRNG}(\gamma, \mathbf{R}_w \times \mathbf{R}_w)$
4: $s \leftarrow x + h \cdot y$
5: **return** $\mathsf{pk} = (\varphi, s)$, $\mathsf{sk} = \gamma$

Algorithm 8 HQC.PPKE-ENCRYPTION

Require: $\mathsf{pk} = (\varphi \in \{0,1\}^{320}, s \in \mathbf{R})$
$\mathbf{m} \in \{0,1\}^{k}, \theta \in \{0,1\}^{512}$
Ensure: $\mathsf{ctx} = (u \in \mathbf{R}, \mathbf{v} \in \mathbb{F}_2^{n_e n_i})$
1: $(e, r_a, r_b) \leftarrow \text{CSPRNG}(\theta, \mathbf{R}_{w_e} \times \mathbf{R}_{w_r} \times \mathbf{R}_{w_r})$
2: $h \leftarrow \text{CSPRNG}(\varphi, \mathbf{R})$
3: $u \leftarrow r_a + h \cdot r_b$
4: $\mathbf{v} \leftarrow \text{ENCODE}_{\mathbf{G}}(\mathbf{m}) + \text{TRUNC}\,(s \cdot r_b + e)$
5: **return** $\mathsf{ctx} = (u, \mathbf{v})$

property. Three random polynomials $e, r_a, r_b \in \mathbf{R}$, are randomly sampled using the seed θ such that $\omega(e) = w_e$ and $\omega(r_a) = \omega(r_b) = w_r$. The parity-check matrix $\mathbf{H}$ is expanded from the public seed φ and used to compute the syndrome polynomial u of the bit vector $[\mathbf{r}_a \mid \mathbf{r}_b]$ as $u \leftarrow r_a + h \cdot r_b \Leftrightarrow \mathbf{u} = \mathbf{H}[\mathbf{r_a} \mid \mathbf{r_b}]^\top$. The codeword resulting from the encoding of the message $\mathbf{m}$ with the public RM/RS code having generator matrix $\mathbf{G}$ is then corrupted adding the first $n_e n_i < p$ bits obtained from the $s \cdot r_b + e$ operation. The resulting vector $\mathbf{v} \in \mathbb{F}_2^{n_e n_i}$ cannot be used to obtain $\mathbf{m}$ using the public RM/RS decoding algorithm alone, since the introduced error bits are far more than its maximum correcting capability. The ciphertext is $(u, \mathbf{v})$.

HQC.PPKE-DECRYPTION

The decryption algorithm (Algorithm 9) starts by expanding the secret vector $[\mathbf{x} \mid \mathbf{y}] \in \mathbb{F}_2^{2p}$ from the secret key $\mathsf{sk} = \gamma$ and subtracts from the polynomial v, derived from the second component $\mathbf{v}$ of the ciphertext, the product of the first one u by the second element of the secret vector y. The resulting quantity, $v - u \cdot y$ can be shown to be close to a codeword of the RM/RS public code $\mathbf{G}$, recalling that $s = h \cdot y + x$ from the keygen algorithm:

$$\begin{aligned} v - u \cdot y &= \mathbf{mG} + (s \cdot r_b + e - h \cdot r_b \cdot y - r_a \cdot y) \\ &= \mathbf{mG} + (h \cdot y \cdot r_b + x \cdot r_b + e - h \cdot y \cdot r_b - r_a \cdot y) \\ &= \mathbf{mG} + (x \cdot r_b + e - r_a \cdot y) \\ &= \mathbf{mG} + \mathbf{e}' \end{aligned} \tag{4.8}$$

Algorithm 9 HQC.PPKE-DECRYPTION

Require: $\mathsf{sk} = \gamma \in \{0,1\}^{320}$, $\mathsf{ctx} = (u \in \mathbf{R}, \mathbf{v} \in \mathbb{F}_2^{n_e n_i})$
Ensure: $\mathbf{m}' \in \{0,1\}^k$
1: $(x, y) \leftarrow \text{CSPRNG}(\gamma, \mathbf{R}_w \times \mathbf{R}_w)$
2: $\mathbf{m}' \leftarrow \text{DECODE}_G(\text{TRUNC}([\mathbf{v} \| 0^{p-n_e n_i}] - u \cdot y))$
3: **return** $\mathbf{m}'$

Algorithm 10 HQC.KEM-KEYGENERATION

Require: None
Ensure: $\mathsf{pk} = (\varphi \in \{0,1\}^{320}, s \in \mathbf{R})$
$\mathsf{sk} = (\gamma \in \{0,1\}^{320}, \sigma \in \{0,1\}^k, \varphi \in \{0,1\}^{320}, s \in \mathbf{R})$
1: $\sigma \xleftarrow{\$} \{0,1\}^k$
2: $((\varphi, s), \gamma) \leftarrow$ HQC.PKE-KEYGENERATION()
3: **return** $\mathsf{pk} = (\varphi, s)$, $\mathsf{sk} = (\gamma, \sigma, \varphi, s)$

The resulting $\mathbf{e}'$ vector corrupting the codeword $\mathbf{mG}$ has a Hamming weight low enough to be successfully corrected by the fixed RM/RS decoder algorithm, retrieving the original message $\mathbf{m}$. The HQC.PPKE parameters (i.e., p, w, w_e, w_r, n_i, n_e, k_i, k_e, d_i, d_e, with $k = k_i k_e$) are tuned so that this decoding action has a negligible failure rate. In the call for post-quantum cryptographic schemes, NIST provided a security level classification to categorize each cipher. Specifically, security levels 1, 3, and 5 correspond to the lowest computational efforts needed to derive the secret key of `AES-128`, `AES-192`, `AES-256` via the best classical and quantum cryptanalytic attacks, respectively (see Table 1.1). The designers of HQC provided parameters for the cryptosystem, reported in Table 4.1, so that the decryption failures take place with a probability of $2^{-\lambda}$, where λ is the bit-length of the key of the AES cipher considered at the corresponding security level.

4.2.3 HQC KEM

The HQC.KEM is obtained wrapping the HQC.PPKE with the HHK transformation [HHK17] which, from a functional standpoint, feeds the HQC.PPKE with a random message, from which the secret to be employed as a session key K is derived, and adds to the ciphertext additional information which allows to check if a decryption error took

Table 4.1: *HQC parameter sets. Public fixed codes are specified via the $[n, k, d]$ notation. The rightmost column reports the Decoding Failure Rate (DFR) of each parameter set.*

Security level	Parameter set	Public concatenated code		Polynomial ring $\mathbf{R}$			DFR
		Reed-Solomon	Reed-Muller	p	w	w_e=w_r	
1	`hqc128`	$[46, 16, 31]$	$[384, 8, 192]$	17669	66	75	2^{-128}
3	`hqc192`	$[56, 24, 33]$	$[640, 8, 320]$	35851	100	114	2^{-192}
5	`hqc256`	$[90, 32, 59]$	$[640, 8, 320]$	57637	131	149	2^{-256}

Algorithm 11 HQC.KEM-ENCAPSULATION

Require: $\mathsf{pk}=(\varphi \in \{0,1\}^{320}, s \in \mathbf{R})$
Ensure: $\mathsf{ctx}=(u \in \mathbf{R}, \mathbf{v} \in \mathbb{F}_2^{n_e n_i}, \text{salt} \in \{0,1\}^{128}), K \in \{0,1\}^{512}$
1: $\mathbf{m} \xleftarrow{\$} \{0,1\}^k$, salt $\xleftarrow{\$} \{0,1\}^{128}$
2: $\theta \leftarrow \text{HASH}_G(\mathbf{m} \| \varphi \| s \| \text{salt})$
3: $(u, \mathbf{v}) \leftarrow \text{HQC.PKE-ENCRYPTION}((\varphi, s), \mathbf{m}, \theta)$
4: $K \leftarrow \text{HASH}_K(\mathbf{m} \| u \| \mathbf{v})$
5: **return** $\mathsf{ctx} = (u, \mathbf{v}, \text{salt}), K$

Algorithm 12 HQC.KEM-DECAPSULATION

Require: $\mathsf{ctx} = (u \in \mathbf{R}, \mathbf{v} \in \mathbb{F}_2^{n_e n_i}, \text{salt} \in \{0,1\}^{128})$
$\mathsf{sk} = (\gamma \in \{0,1\}^{320}, \sigma \in \{0,1\}^k, \varphi \in \{0,1\}^{320}, s \in \mathbf{R})$
Ensure: $K \in \{0,1\}^{512}$
1: $\mathbf{m}' \leftarrow \text{HQC.PKE-DECRYPTION}(\gamma, (u, \mathbf{v}))$
2: $\theta' \leftarrow \text{HASH}_G(\mathbf{m}' \| \varphi \| s \| \text{salt})$
3: $\mathsf{ctx}' \leftarrow \text{HQC.PKE-ENCRYPTION}((\varphi, s), \mathbf{m}', \theta')$
4: **if** $(\mathsf{ctx} \neq \mathsf{ctx}')$ $K' \leftarrow \text{HASH}_K(\sigma \| u \| \mathbf{v})$ **else** $K' \leftarrow \text{HASH}_K(\mathbf{m}' \| u \| \mathbf{v})$
5: **return** K'

place. In this case, to avoid information leakage, the construction emits a random string, deterministically derived with a CSPRNG from the ciphertext and a secret seed stored within the private key.

HQC.KEM-KEYGENERATION

The KEM key generation algorithm matches the one of HQC.PPKE-KEYGENERATION, save for the generation of the additional seed, denoted as σ, of 128, 192, or 256 bits (depending on the security level), stored together with the private seed γ.

HQC.KEM-ENCAPSULATION

The encapsulation algorithm, presented in Algorithm 11, starts by picking a uniformly distributed random message $\mathbf{m}$ encoded with 128, 192, or 256 bits, depending on the security level, and a 128-bit public salt. The two quantities are concatenated with the public key (φ, s) and hashed via the HASH_G function to get a digest θ, which is subsequently employed as the ephemeral value required as an input by the HQC.PPKE-ENCRYPTION algorithm. After calling the routine HQC.PPKE-ENCRYPTION, the shared secret session key K is derived through a different hash function, HASH_K, fed with the concatenation of the message $\mathbf{m}$ and the components $u, \mathbf{v}$ of the ciphertext.

HQC.KEM-DECAPSULATION

The decapsulation procedure (Algorithm 12) starts by computing the output value $\mathbf{m}'$ of the routine HQC.PPKE-DECRYPTION fed with the secret key seed and the received

components $u, \mathbf{v}$ of the ciphertext ctx, which should match the original confidential message $\mathbf{m}$ unless a decryption failure occurs. To distinguish between the two possible scenarios and provide security against active attackers, which may have mangled the received ciphertext, the HHK transformation followed by the decapsulation procedure mandates to compare the received ctx with the output obtained from the re-computation of the HQC.PPKE-ENCRYPTION routine, which in turn requires the re-computation of the ephemeral value θ fed to it as last input parameter. If the retrieved secret message $\mathbf{m}'$ matches the one that was actually encrypted, the outcome of such process (lines 2 and 3 of Algorithm 12) will yield a value ctx$'$ matching the received ciphertext ctx (check performed at line 4). In case the recomputed ciphertext ctx$'$ does not match the received one, the *implicit* rejection mechanism requires the computation of the session key K as the result of the HASH_K function fed with the concatenation of the received ctx and the binary string σ (included in the secret key), instead of the mangled message $\mathbf{m}'$, which may provide information to an attacker.

4.2.4 Comparison with BIKE

BIKE is a code-based post-quantum scheme whose security is guaranteed by:

Private key recovery the indistinguishability of a hidden QC Medium Density Parity Check (QC-MDPC) code from a QC random parity check code

Session key recovery the hardness of the random quasi-cyclic (QC) syndrome decoding problem

HQC on the other hand relies only on the latter for both recovery attacks.

Both cryptographic schemes operates with element in a binary polynomial ring $\mathbf{R} = \mathbb{F}_2[x]/\langle x^p - 1\rangle$, with p a large prime number ≥ 10000. Note that the polynomial ring used by BIKE has slightly smaller elements than the ones used in HQC – the value p, which also represents the bit-length of an element, is equal to 12323, 24659, 40973, as opposed to 17669, 35851, 57637, for the parameter sets defining security levels of `AES-128`, `AES-192`, and `AES-256`, respectively. This practically means that the arithmetic operations performed on the elements defined by the mid-range security level of HQC are roughly executed in the same amount of time as the BIKE's arithmetic operations defined by the highest security margin. This performance gap for the execution of arithmetic operation is similar and consistent for all the parameters sets defined by the two schemes.

To thoroughly compare the schemes, we need to consider the schedule of the operations for the three KEM primitives (key generation, encapsulation, and decapsulation), focusing primarily on the most computationally intensive operations. A summary of the operations involved in the computation of the primitives is provided in Table 4.2.

Key generation

BIKE needs to sample two polynomials with a fixed Hamming weight. In this case, the constant-time property of such operation is not mandatory, hence the sampling

Table 4.2: *Comparison of BIKE and HQC on the type and number of operations involved in the three KEM primitives* KEYGENERATION, ENCAPSULATION, *and* DECAPSULATION.

Operations	KeyGeneration		Encapsulation		Decapsulation	
	BIKE	HQC	BIKE	HQC	BIKE	HQC
Fixed-weight sample in $\mathbf{R}$	2	2	2	3	2	4
Inverse in $\mathbf{R}$	1	0	0	0	0	0
Multiplication in $\mathbf{R}$	1	1	1	2	1	3
Decoder	0	0	0	0	1	1
Encoder	0	0	0	1	0	1

algorithm specified by the authors of BIKE is more straightforward than the one in HQC. Afterwards, an inverse inverse $\mathbf{R}$ is computed and used in a multiplication between polynomials to generate the public key. Compared to HQC, the simplified sampling algorithm gives a slight benefit in performance, but it does not compensate for the large computational cost of computing the inverse element in $\mathbf{R}$ (about 10-30 multiplications in $\mathbf{R}$), as demonstrated by the latency of the key generation operation highlighted in the presented results for HQC.

Encapsulation

BIKE samples two additional random polynomials with fixed-weight, as opposed to the 3 random polynomials required by HQC. Furthermore, a singe multiplication among polynomials is needed, whereas in HQC two of such operations are carried out. Finally, HQC needs to encode a message using the Reed-Muller/Reed-Solomon concatenated code. The presented benchmarks determine a $1.6 \sim 2.1\times$ latency improvement of the HQC design in this thesis, along with a $\approx 2\times$ efficiency gain with respect with the current state-of-the-art BIKE hardware implementation [Ric+22], a remarkable result considering the higher number of slightly more complex operations carried out in HQC.

Decapsulation

BIKE performs one multiplication before decoding the resulting codeword to the final message, which is used to re-sample the same two random-polynomials computed during the encapsulation. The decoding of BIKE's Moderate-Density Parity-Check (MDPC) code is achieved with a simple iterative decoder. The recently adopted (October 2024) BIKE-FLIP decoder has a simpler logic than the Black-Gray-Flip (BGF) decoder previously used, making the memory bound nature of iterative hard decision bit flipping decoders even more evident. Furthermore, iterative decoders need to act on their entire input for the whole duration of the decoding computation. On the other hand, HQC performs substantially more operations during the decapsulation – samples 4 polynomial with fixed-weight, performs 3 multiplications among polynomials, and decodes the codeword obtained from the ciphertext to the final message, which is then re-encoded to validate the operations by the HHK transformation. HQC uses a conceptually more com-

plex algebraic decoder algorithm, which, besides being computationally bound, leaves a greater margin for engineering, as it on works with smaller elements. This gave the opportunity to efficiently extract a higher degree of computational parallelism with respect to an iterative decoder, without imposing taxing requirements on memory resources. Summing up, the presented design handles a higher number of slightly more complex operations faster ($9.1 \sim 20.7\times$) and more efficiently ($11.2 \sim 25.8\times$) than the best design for BIKE [Ric+22].

4.3 Codes and Restricted Objects Signature Scheme

The Codes and Restricted Objects Signature Scheme (CROSS) is a post-quantum Digital Signature (DS) obtained applying the Fiat-Shamir (FS) framework to the CROSS-ID Zero-Knowledge (ZK) protocol relying on the NP-complete Restricted Syndrome Decoding Problem (R-SDP) or Restricted Syndrome Decoding Problem in the subgroup G (R-SDP(G)) introduced in subsection 4.1.2, and on the hardness of finding collisions in a cryptographic hash function. The main advantage over other code-based digital signatures derives from the removal of the fixed-weight constraint in favor of the restriction of the ambient space of the error vector, meaning that its coefficients must be in the subgroups $E^n \subset \mathbb{F}_p^n$ or $G \subset E^n$ for R-SDP and R-SDP(G), respectively. In turn this allows to have shorter signatures while having simple arithmetic, particularly when it comes to computing element transformations. In fact, CROSS performs modular arithmetic in integer fields modulo a small prime, mostly a Mersenne prime to simplify the computation of the modulo remainder, and can be easily performed in a constant amount of time to thwart time side-channel attacks. The size of public and private keys are minimal, where the private key consists of just a single random seed, while the public key takes at most 121 B and 74 B for the R-SDP and R-SDP(G) variants, respectively. Some standard optimization techniques are adopted to reduce the signature size, for example deriving the seeds used in each ZK round from a binary tree, using a Merkle tree to reduce the number of commitments transmitted in the signature, and producing an intentionally unbalanced distribution of commitments.

4.3.1 Algebraic structures and parameters sets

As in the SDP, the public key consists in a parity check matrix $\mathbf{H} \in \mathbb{F}_p^{(n-k)\times n}$ of a p-ary $[n, k, d]$ random linear code deterministically generated from the output material of a CSPRNG, in particular from the SHAKE eXtendable-Output Function (XOF), starting from a small public random seed (further details will be provided in subsection 5.2.1). The error vector $\mathsf{sk} = \mathbf{e} \in \mathbb{F}_p^n$ is the private key of the scheme, similarly deterministically generated from the output material of a CSPRNG initialized by a small private random seed. Differently from the SDP, $\mathbf{e}$ does not have a fixed weight, but rather its coefficients are in the subgroup E generated by the public element g of order z: $\langle g \rangle = \{g^i \mid i \in \{1, \dots, z\}\} = E \subset \mathbb{F}_p^*$, thus $\mathbf{e} \in E^n$. The syndrome is consequently

generated as $\mathbf{s} = \mathbf{e}\mathbf{H}^\top$ and is part of the public key $\mathsf{pk} = (\mathbf{H}, \mathbf{s})$, which be reduced in size by transmitting the public seed used to deterministically generate the matrix $\mathbf{H}$ in its place. Note that both p and z are small prime numbers, so the vector coefficients are part of an algebraic field, and therefore each coefficient of $\mathbf{e}$ has also order z.

Let $(\odot)$ be the Hadamard product (component-wise multiplication), then the commutative group $(E^n, \odot)$ is isomorphic to $(\mathbb{F}_z^n, +)$. An element $\mathbf{a} \in E^n$ can be represented by a vector $\overline{\mathbf{a}} \in \mathbb{F}_z^n$ with vector coefficients in $\mathbb{F}_z$, hereafter denoted by an overline to specify that it is from the $\mathbb{F}_z^n$ vector space, such that $\left[g^{\overline{\mathbf{a}}_1}, \ldots, g^{\overline{\mathbf{a}}_n}\right]$, or $g^{\overline{\mathbf{a}}}$ for brevity. Having two values $\mathbf{a}, \mathbf{b} \in E^n$, then their component-wise product is $\mathbf{a} \odot \mathbf{b} = g^{\overline{\mathbf{a}}+\overline{\mathbf{b}}}$, and the inverse element of $\mathbf{a}$ is $\mathbf{a}^{-1} = g^{-\overline{\mathbf{a}}}$.

Considering the R-SDP(G), let G be the subgroup of E^n defined as follows:

$$G = \langle\{\mathbf{b_1}, \ldots, \mathbf{b_m}\}\rangle = \left\{ \bigodot_{i=1}^{m} \mathbf{b_i}^{u_i} \mid \mathbf{b_i} \in E^n \wedge u_i \in \mathbb{F}_z^\star \wedge m < n \right\} \subset E^n \tag{4.9}$$

Then let $\overline{\mathbf{M}} = \left[\overline{\mathbf{b_1}}, \ldots, \overline{\mathbf{b_m}}\right]^\top$ be the $\mathbb{F}_z^{m \times n}$ matrix of the exponents of g composing $\mathbf{b_1}, \ldots, \mathbf{b_m}$, having rank equal to m. An element $\mathbf{a} \in G$ can be computed as $\mathbf{a} = g^{\overline{\mathbf{a}}_G \overline{\mathbf{M}}}$ starting from an even smaller vector $\overline{\mathbf{a}}_G \in \mathbb{F}_z^m$, hereafter denoted denoted by the $_G$ subscript.

These isomorphisms are useful when considering the linear transitive maps $\mathrm{v} : E^n \mapsto E^n$ and $\mathrm{v}_G : G \mapsto G$ simply given by the Hadamard product by an element $\mathbf{v} \in E^n$ or $\mathbf{v} \in G$:

$$\mathrm{v}(\mathbf{a}) = \mathbf{v} \odot \mathbf{a} = g^{\overline{\mathbf{v}}+\overline{\mathbf{a}}} \tag{4.10}$$

As a consequence, a random linear map v can be compactly represented by any random element $\overline{\mathbf{v}}$. The same is true for a random linear map v_G that is be represented by an even more compact random element $\overline{\mathbf{v}}_G$ in the R-SDP(G) variant.

The CROSS scheme offers two variants, the more conservative one relying on the R-SDP, and the one based on the R-SDP(G) providing smaller signatures. For the first variant both $p = 127$ and $z = 7$ are Mersenne primes, while for the latter only $z = 127$ is a Mersenne prime, while $p = 509$ is a generic small prime number. For each variant there are different parametrization of the random $[n, k, d]$ random linear code guaranteeing the three security margins mandated by NIST, equivalent to the security offered by `AES-128`, `AES-192`, and `AES-256`. As a further choice offered for each parameter set, a trade-off between execution runtime and signature size is proposed by tweaking some parameters, such as the number of ZK protocol repetitions t, or the unbalanced distribution of the second CROSS-ID binary challenge having Hamming weight w, to the goal of producing a *fast* variant, one generating *small* signatures, and a *balanced* option in between the previous two choices. Overall, the CROSS specification [Bal+24a] defines 18 parameter sets, here reported in Table 4.3.

Table 4.3: *CROSS parameter sets for the three security levels mandated by NIST. Two variants are proposed for each security level, depending on the hard problem relying on, and for each one of them three optimization corners are offered producing a trade-off between sing/verification runtime and signature size. The generator element g is public and fixed for every parameter sets, and is equal to* 2 *and* 16 *for R-SDP and R-SDP(G), respectively.*

Security level	Hard problem	Trade-off variant	Parameter set	p	z	n	k	m	t	w	Size (B) sk	pk	sig
1	R-SDP	fast	CROSS-RSDP-1-f	127	7	127	76	–	157	82	32	77	18432
		balanced	CROSS-RSDP-1-b						256	215	32	77	13200
		small	CROSS-RSDP-1-s						520	488	32	77	12480
	R-SDP(G)	fast	CROSS-RSDPG-1-f	509	127	55	36	25	147	76	32	54	11980
		balanced	CROSS-RSDPG-1-b						256	220	32	54	9168
		small	CROSS-RSDPG-1-s						512	484	32	54	9008
3	R-SDP	fast	CROSS-RSDP-3-f	127	7	187	111	–	239	125	48	115	41406
		balanced	CROSS-RSDP-3-b						384	321	48	115	29925
		small	CROSS-RSDP-3-s						580	527	48	115	28463
	R-SDP(G)	fast	CROSS-RSDPG-3-f	509	127	79	48	40	224	119	48	83	26772
		balanced	CROSS-RSDPG-3-b						268	196	48	83	22536
		small	CROSS-RSDPG-3-s						512	463	48	83	20524
5	R-SDP	fast	CROSS-RSDP-5-f	127	7	251	150	–	321	167	64	153	74590
		balanced	CROSS-RSDP-5-b						512	427	64	153	53623
		small	CROSS-RSDP-5-s						832	762	64	153	50914
	R-SDP(G)	fast	CROSS-RSDPG-5-f	509	127	106	69	48	300	153	64	106	48102
		balanced	CROSS-RSDPG-5-b						356	258	64	106	40196
		small	CROSS-RSDPG-5-s						642	575	64	106	36550

4.3.2 CROSS ZK protocol

CROSS defines the CROSS-ID ZK identification protocol starting from the one in [CVA10], and using R-SDP hard problem as a foundation. Within a CROSS-ID protocol execution, the signer will prove to the verifier that the secret key, represented by the error vector $\mathbf{e} \in \mathbb{F}_p^n$, either satisfies the syndrome equation $\mathbf{s} = \mathbf{e}\mathbf{H}^\top$, or that it is from E^n (or G in case of R-SDP(G)). For the sake of simplicity, in this section it is presented the identification protocol for R-SDP(G), which is applicable also for R-SDP by considering $m = n$ and $\overline{\mathbf{M}} = \overline{\mathbf{I}}_m$, which make $G = E^n$.

The CROSS-ID is a 5-pass protocol here represented in Algorithm 13, where a prover $\mathcal{P}$ tries to convince a verifier $\mathcal{V}$ that it is in possess of a secret value $\mathbf{e}$ bound to the public key $(\mathbf{s}, \mathbf{H})$ as $\mathbf{s} = \mathbf{e}\mathbf{H}^\top$. After an initial commitment of the prover $\mathcal{P}$, the verifier $\mathcal{V}$ sends two challenges, the first one from a space of cardinality $q = p - 1$, and the second one from a space of cardinality 2. Finally, the verifier $\mathcal{V}$ checks the sent commitment and the response to the first challenge. It is therefore classified as $q2$-Identification scheme, with a soundness error of $\varepsilon = \frac{p}{2(p-1)} \approx \frac{1}{2}$.

Transformation

An initial random bit string seed $\texttt{seed}$ of size λ equal to the security margin of the parameter set in use is sampled uniformly at random. It is then used to initialize a CSPRNG to generate the transformed error $\mathbf{e}' \in G$ and the element $\mathbf{u}' \in \mathbb{F}_p^n$. A peculiarity of CROSS-ID is that the transformation v_G associated to the element $\mathbf{v}$ is computed via Equation 4.10 from the random element $\mathbf{e}' \in G$ such that $\mathrm{v}_G(\mathbf{e}) = \mathbf{e}'$: Note that if both $\mathbf{e}$ and $\mathbf{e}'$ are sampled from a uniform distribution, so it is for the transformation $\mathbf{v}$. It is imperative to not share both the transformation $\mathbf{v}$ and the random element $\mathbf{e}'$ in the same response to the challenge to avoid the verifier to recompute the secret key as $\mathbf{e} = \mathrm{v}_G(\mathbf{e}')$.

Commitment

Then the syndrome $\mathbf{s}' \in \mathbb{F}_p^{n-k}$ is computed from the transformed element $\mathbf{u} = \mathbf{v} \odot \mathbf{u}'$ using the parity check matrix $\mathbf{H}$. The two commitments $\texttt{cmt}_0$ and $\texttt{cmt}_1$ are generated as the digest of the absorption of the bit string representations of $\mathbf{s}'$ concatenated to $\mathbf{v}$, and $\mathbf{u}'$ concatenated to $\mathbf{e}'$, respectively.

First challenge

After receiving the commitments from the prover $\mathcal{P}$, the verifier $\mathcal{V}$ generates a first random challenge $\texttt{chall}_1 \in \mathbb{F}_p^*$, which is then used by the prover $\mathcal{P}$ to compute the quantity $\mathbf{y} = \mathbf{u}' + \texttt{chall}_1\mathbf{e}'$. As a response to the first challenge, the prover sends the digest $\texttt{dig}_\mathbf{y}$ of the bit string representation of $\mathbf{y}$.

Second challenge

The verifier now generates a second challenge $\texttt{chall}_2$, a single binary choice to ask the prover to communicate either the bit string representation $\texttt{resp}$ of both $\mathbf{y}$ and $\mathbf{v}$, or the randomness $\texttt{seed}$ used in the protocol execution. Note that $\overline{\mathbf{v}}_G \in \mathbb{F}_z^m$

Algorithm 13 CROSS-ID

Require: $\mathsf{sk} = e \in G \subseteq E^n$
$\mathsf{pk} = \left(\mathbf{H} \in \mathbb{F}_p^{(n-k)\times n}, \mathbf{s} = \mathbf{e}\mathbf{H}^\top \in \mathbb{F}_p^{n-k}\right)$

PROVER $\mathcal{P}$		VERIFIER $\mathcal{V}$
$\mathtt{seed} \xleftarrow{\$} \{0,1\}^\lambda$		
..... Commitment		
$\mathbf{e}', \mathbf{u}' \leftarrow \text{CSPRNG}(\mathtt{seed}, G \times \mathbb{F}_p^n)$		
$\mathbf{v} \leftarrow \mathbf{e} \odot (\mathbf{e}')^{-1}$		
$\mathbf{u} \leftarrow \mathbf{v} \odot \mathbf{u}'$		
$\mathbf{s}' \leftarrow \mathbf{u}\mathbf{H}^\top$		
$\mathtt{cmt}_0 \leftarrow \textsc{Hash}(\mathbf{s}' \Vert \mathbf{v})$		
$\mathtt{cmt}_1 \leftarrow \textsc{Hash}(\mathbf{u}' \Vert \mathbf{e}')$		
	$\xrightarrow[\text{commitment}]{\mathtt{cmt}_0, \mathtt{cmt}_1}$	
..... First challenge		
		$\mathtt{chall}_1 \xleftarrow{\$} \mathbb{F}_p^*$
	$\xleftarrow[1^{\text{st}}\text{ challenge}]{\mathtt{chall}_1}$	
$\mathbf{y} \leftarrow \mathbf{u}' + \mathtt{chall}_1 \mathbf{e}'$		
$\mathtt{dig}_\mathbf{y} \leftarrow \textsc{Hash}(\mathbf{y})$		
	$\xrightarrow[1^{\text{st}}\text{ response}]{\mathtt{dig}_\mathbf{y}}$	
..... Second challenge		
		$\mathtt{chall}_2 \xleftarrow{\$} \{0,1\}$
	$\xleftarrow[2^{\text{nd}}\text{ challenge}]{\mathtt{chall}_2}$	
if $\mathtt{chall}_2 = 0$ **then** $\mathtt{resp} \leftarrow \mathbf{y} \Vert \mathbf{v}$		
else $\mathtt{resp} \leftarrow \mathtt{seed}$		
	$\xrightarrow[2^{\text{nd}}\text{ response}]{\mathtt{resp}}$	
..... Verification		
		if $\mathtt{chall}_2 = 0$ **then**
		$\mathbf{y}' \leftarrow \mathbf{v} \odot \mathbf{y}$
		$\mathbf{s}' \leftarrow \mathbf{y}'\mathbf{H}^\top - \mathtt{chall}_1 \mathbf{s}$
		if $\textsc{Hash}(\mathbf{y}) \neq \mathtt{dig}_\mathbf{y} \vee \textsc{Hash}(\mathbf{s}' \Vert \mathbf{v}) \neq \mathtt{cmt}_0 \vee \mathbf{v} \notin G$ **then fail**
		else
		$(\mathbf{e}', \mathbf{u}') \leftarrow \text{CSPRNG}(\mathtt{seed}, G \times \mathbb{F}_p^n)$
		$\mathbf{y} \leftarrow \mathbf{u}' + \mathtt{chall}_1 \mathbf{e}'$
		if $\textsc{Hash}(\mathbf{y}) \neq \mathtt{dig}_\mathbf{y} \vee \textsc{Hash}(\mathbf{u}' \Vert \mathbf{e}') \neq \mathtt{cmt}_1$ **then fail**

is smaller than $\mathbf{v} \in \mathbb{F}_p^n$, therefore to reduce the size of the transmitted message, $\overline{\mathbf{v}}_G$ is sent in its place, and recomputed by the verifier as $\mathbf{v} = g^{\overline{\mathbf{v}}_G \overline{\mathbf{M}}}$. Moreover, consider that when $\texttt{chall}_2 = 1$, the transmitted message is just a seed, which is way smaller than the bit string representation of the elements in the $\mathbb{F}_p^n \times \mathbb{F}_z^m$ space. Therefore, an optimization to reduce the signature size makes the verifier $\mathcal{V}$ generate $\texttt{chall}_2 = 1$ more often than $\texttt{chall}_2 = 0$.

Verification

The verifier $\mathcal{V}$ now has to check the initial commitment and the response to the first challenge are correct. If the second challenge $\texttt{chall}_1$ is 0, then it received $\mathbf{y}$ and $\mathbf{v}$ from the prover $\mathcal{P}$. It initially computes $\mathbf{y}' = \mathbf{v} \odot \mathbf{y}$:

$$\mathbf{y}' = \mathbf{v} \odot \mathbf{y} = \mathbf{v} \odot (\mathbf{u}' + \texttt{chall}_1 \mathbf{e}') = \mathbf{v} \odot \mathbf{u}' + \mathbf{v} \odot (\texttt{chall}_1 \mathbf{e}') = \mathbf{u} + \texttt{chall}_1 \mathbf{e} \tag{4.11}$$

Therefore, $\mathbf{y}'$ contains the secret error vector $\mathbf{e}$ scaled by the first challenge $\texttt{chall}_2$, and masked by the unknown value $\mathbf{u}$. The quantity $\mathbf{y}'$ is then multiplied by the parity-check matrix $\mathbf{H}$ to obtain $\mathbf{s}' + \texttt{chall}_1 \mathbf{s}$:

$$\mathbf{y}' \mathbf{H}^\top = (\mathbf{u} + \texttt{chall}_1 \mathbf{e})\, \mathbf{H}^\top = \mathbf{u}\mathbf{H}^\top + \texttt{chall}_1 \left(\mathbf{e}\mathbf{H}^\top\right) = \mathbf{s}' + \texttt{chall}_1 \mathbf{s} \tag{4.12}$$

Starting from Equation 4.12, it is possible to isolate $\mathbf{s}'$ as $\mathbf{y}'\mathbf{H}^\top - \texttt{chall}_2 \mathbf{s}$. The verifier $\mathcal{V}$ can now check the commitment $\texttt{cmt}_0$, the first challenge response $\texttt{dig}_\mathbf{y}$, and that $\overline{\mathbf{v}}_G \in \mathbb{F}_z^m$.

In case the second challenge $\texttt{chall}_2$ is 1, then the prover $\mathcal{P}$ sent the seed of the protocol $\texttt{seed}$. Consequently, the verifier $\mathcal{V}$ can recompute $\mathbf{e}'$ and $\mathbf{u}'$, and verify that the first response is indeed computed as $\mathbf{y} = \mathbf{u}' + \texttt{chall}_1 \mathbf{e}'$, and that the commitment $\texttt{cmt}_1$ is correct.

In [Bal+24a] the authors give the proofs of the completeness, ZK, and soundness properties of this identification protocol, and describe the details of the efficient implementation of such protocol in terms of message sizes and computation complexity.

4.3.3 CROSS digital signature

The CROSS Digital Signature algorithm is constructed via the Fiat-Shamir transformation of t independent executions of the CROSS-ID ZK identification protocol, making their execution non-interactive using a one-way function and achieving the EUF-CMA security. The protocol executions are performed by the signer deterministically computing the challenges of the verifier $\mathcal{V}$ using the digest of a cryptographically safe hash function that absorbed all the previous transferred messages between the prover $\mathcal{P}$ and the verifier $\mathcal{V}$, the message msg to be signed, and a 2λ salt public random value unique for each signing operation. The transcript of the transferred messages from the prover $\mathcal{P}$ to the verifier $\mathcal{V}$, along with the salt used, is the requested message signature sig. The verification consists in the deterministic recreation of the two challenges used by the signer, and the execution of the final verification step of the CROSS-ID protocol.

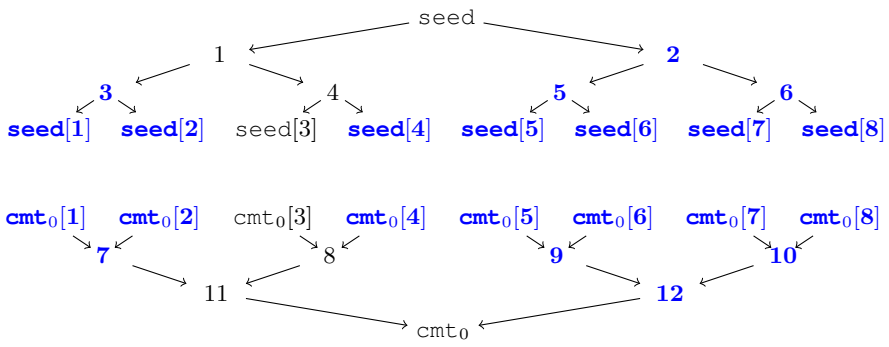


Figure 4.3: *Simplified working principle of CROSS seed (above) and commitment (below) trees. Note that the CROSS specification do not generate perfect nor complete binary trees. The signer needs to communicate to the verifier all the w out of t seeds and commitments highlighted in blue, determined by $\texttt{chall}_2[i] = 1$. However, it can just send $\texttt{path} = \mathbf{3}\|\mathbf{seed}[4]\|\mathbf{2}$, called the seed path, and let the verifier expand the missing seeds (*REBUILDLEAVES*). Similarly, the signature contains $\texttt{proof} = \mathbf{7}\|\mathbf{cmt}_0[4]\|\mathbf{12}$, called the Merkle tree proof. When the verifier will compute the missing commitment $\texttt{cmt}_0[\mathbf{3}]$, the verifier will be able to retrieve the root of the Merkle tree (*RECOMPUTEROOT*).*

To distinguish the elements and messages of the i-th protocol execution, with $0 < i \leq t$, a suffix $[i]$ is added to them. All the t CROSS-ID protocol executions are performed in parallel, synchronizing the message transfers by aggregating the t homogeneous messages in a single one (e.g., $\texttt{chall}_2 = \texttt{chall}_2[1]\| \ldots \|\texttt{chall}_2[t]$). For performance reasons, in some cases it is also possible to replace the computation of t hash functions having a single element with a single hash computation of the t concatenated elements, such as in the case of the response to the first challenge $\texttt{dig}_\mathbf{y} = \text{HASH}(\mathbf{y}[1]\| \ldots \|\mathbf{y}[t])$. Note that with this shortened notation, it is intended to produce a digest of the concatenated bit string representation of all elements $\mathbf{y}$.

To reduce the signature size, sig only contains the commitments that the verifier cannot recompute depending on the second challenge $\texttt{chall}_2$ value. The validity of all the commitments is then checked by attaching to the signature a small digest of the commitments concatenated together, so that the verifier can combine the half ones received in the signature with the other half computed during the verification, and compare the digest of their absorption. In case the second challenge is sampled from an unbalanced distribution to send more often $\texttt{chall}_2[i] = 1$ challenges, as it is the case for the *balanced* and *small* parameter sets, the number of protocol executions t needs to increase. In the end, the signature size is slightly reduced, at cost of an increased computational complexity due to the larger number of parallel CROSS-ID protocols executed.

It is possible, however, to compress the large number of transmitted $\texttt{seed}[i]$ by considering them as the leaves of a binary tree computed by expanding λ intermediate nodes in 2λ child nodes. In this way, it is possible to only communicate the ancestor nodes of the tree leafs to be revealed to the verifier. The same principle can be used for the transmission of the commitments $\texttt{cmt}_0[i]$, which are present in the signature more

often than $\texttt{cmt}_1[i]$, by using a Merkle tree structure where the commitments $\texttt{cmt}_0[i]$ are the leafs of the tree. Each intermediate node is composed by absorbing the two child nodes, up to the tree root. Consequently, in the signature are attached the common ancestors of the tree leafs to be revealed to the verifier. A simplified case is presented in Figure 4.3. Note that the *fast* parameter set does not employ any tree structure, as $w \approx t/2$, so there are no improvements in the signature size, but rather just an increased computational complexity.

For performance and memory saving reasons, the random matrices $\mathbf{H} \in \mathbb{F}_p^{(n-k)\times n}$ of rank $n-k$ and $\overline{\mathbf{M}} \in \mathbb{F}_z^{m\times n}$ of rank m are sampled and used in their systematic form $\mathbf{H} = [\mathbf{V}, \mathbf{I}_{n-k}]$ and $\overline{\mathbf{M}} = \left[\overline{\mathbf{W}}, \bar{\mathbf{I}}_m\right]$. In this way, $\mathbf{V} \in \mathbb{F}_p^{(n-k)\times k}$ and $\overline{\mathbf{W}} \in \mathbb{F}_z^{m\times(n-m)}$ are generated from the output material of a CSPRNG after the absorption of $\texttt{seed}_{\mathsf{pk}}$.

The CROSS scheme makes use of two cryptographically safe one-way functions, $\text{HASH} : \{0,1\}^* \mapsto \{0,1\}^{2\lambda}$ and $\text{CSPRNG} : \{0,1\}^* \mapsto \{0,1\}^*$. Both are implemented using the SHAKE XOF function standardized by NIST in the SHA-3 *FIPS 202* standard[PM15], to minimize the size of imported libraries in SW implementation, and reduce the area consumption in HW designs. Depending on the required security margin λ, it can either use the SHAKE128 ($\lambda = 128$) or SHAKE256 ($\lambda = 192, 256$) functions, with a performance boost derived by the slightly higher output throughput delivered by the former one. The implementation of the HASH function truncates the XOF output to 2λ. However, to avoid potential collisions in the output of the two functions CSPRNG and HASH, a domain separation mechanism is implemented by transparently appending a different 16-bit integer to the absorbed material. In particular, a value $\geq 2^{15}$ is used for defining the HASH domain, thus any value $< 2^{15}$ denotes an implementation of the CSPRNG function. It is possible to define up to 2^{15} different CSPRNG and HASH instances for further separation. Considering the i-th instance of the HASH, here denoted as HASH_i, the appended 16-bit integer will be $2^{15} + i$.

CROSS.KEYGENERATION

In the following paragraphs is presented an overview of the operations necessary to generate the key pair of the CROSS signature scheme, here also reported in Algorithm 14. Whenever there is a difference between the hard problem instance, on the left side, highlighted in orange, are reported the steps specific to instantiate the R-SDP(G) problem, while on the right side, highlighted in teal, are reported the steps specific to instantiate the R-SDP problem.

The generation of the key pair starts by securely generating a 2λ-bits long seed from a TRNG, which will be the secret key of the scheme $\texttt{seed}_{\mathsf{sk}}$. This seed is then absorbed and expanded via the CSPRNG function to generate two random bit strings, $\texttt{seed}_{\mathsf{e}}$ and $\texttt{seed}_{\mathsf{pk}}$, both 2λ-bits long.

The second one is then expanded in an arbitrary long bit string material used by a sampling algorithm to generate the matrices $\overline{\mathbf{W}} \in \mathbb{F}_z^{m\times(n-m)}$ (only for R-SDP(G) parameters) and $\mathbf{V}^\top \in \mathbb{F}_p^{k\times(n-k)}$ which, adjointed column-wise and row-wise with the

Algorithm 14 CROSS.KEYGENERATION

Require: None
Ensure: $\mathsf{sk} = \mathtt{seed}_{\mathsf{sk}} \in \{0,1\}^{2\lambda}$
$\mathsf{pk} = (\mathtt{seed}_{\mathsf{pk}} \in \{0,1\}^{2\lambda}, \mathbf{s} \in \mathbb{F}_p^{n-k})$

$\mathtt{seed}_{\mathsf{sk}} \xleftarrow{\$} \{0,1\}^{2\lambda}$
$(\mathtt{seed}_{\mathbf{e}}, \mathtt{seed}_{\mathsf{pk}}) \leftarrow \text{CSPRNG}(\mathtt{seed}_{\mathsf{sk}} \| 3t+1, \{0,1\}^{2\lambda} \times \{0,1\}^{2\lambda})$
▷ *Sampling the random matrices* $\mathbf{H}$ *and* $\overline{\mathbf{M}}$ ◁

$\overline{\mathbf{W}}, \mathbf{V}^\top \leftarrow \text{CSPRNG}_{3t+2}(\mathtt{seed}_{\mathsf{pk}}, \mathbb{F}_z^{m\times(n-m)} \times \mathbb{F}_p^{k\times(n-k)})$ $\overline{\mathbf{M}} \leftarrow [\overline{\mathbf{W}}, \mathbf{I}_m]$	$\mathbf{V}^\top \leftarrow \text{CSPRNG}_{3t+2}(\mathtt{seed}_{\mathsf{pk}}, \mathbb{F}_p^{k\times(n-k)})$

$\mathbf{H}^\top \leftarrow [\mathbf{V}^\top \mid \mathbf{I}_{n-k}]$
▷ *Computing* $\mathbf{e}$ ◁

$\overline{\mathbf{e}}_G \leftarrow \text{CSPRNG}_{3t+3}(\mathtt{seed}_{\mathbf{e}}, \mathbb{F}_z^m)$ $\overline{\mathbf{e}} \leftarrow \overline{\mathbf{e}}_G \overline{\mathbf{M}}$	$\overline{\mathbf{e}} \leftarrow \text{CSPRNG}_{3t+3}(\mathtt{seed}_{\mathbf{e}}, \mathbb{F}_z^n)$

for $j \leftarrow 0$ **to** $n-1$ **do**
  $\mathbf{e}_j \leftarrow g^{\overline{\mathbf{e}}_j}$
▷ *Computing the syndrome* $\mathbf{s}$ ◁
$\mathbf{s} \leftarrow \mathbf{e}\mathbf{H}^\top$
return $(\mathsf{sk} = \mathtt{seed}_{\mathsf{sk}}, \mathsf{pk} = (\mathtt{seed}_{\mathsf{pk}}, \mathbf{s}))$;

identity matrices $\overline{\mathbf{I}}_m$ and $\mathbf{I}_{\mathbf{n-k}}$, respectively, form the public matrix $\overline{\mathbf{M}} \in \mathbb{F}_z^{m\times n}$ and the transposed parity-check matrix $\mathbf{H}^\top \in \mathbb{F}_p^{n\times(n-k)}$ of the random code. For the transmission of such large public matrices, it is possible to share just the bit string material $\mathtt{seed}_{\mathsf{pk}}$ initially absorbed by the CSPRNG, and by using the same deterministic sampling algorithm it is possible to recreate the matrices.

Afterwards, the secret bit string $\mathtt{seed}_{\mathbf{e}}$ is absorbed by a CSPRNG and expanded in an arbitrary long bit string material used by a vector sampling algorithm to generate either $\overline{\mathbf{e}}_G \in \mathbb{F}_z^m$ or $\overline{\mathbf{e}} \in \mathbb{F}_z^n$, depending on the chosen instance of hard problem. For the R-SDP(G) variant, $\overline{\mathbf{e}} \in \mathbb{F}_z^n$ is computed by expanding $\overline{\mathbf{e}}_G$ using the public matrix $\overline{\mathbf{M}}$. This expansion can be seen as the encoding of the message $\overline{\mathbf{e}}_G$ with the random linear code having generator matrix $\overline{\mathbf{M}}$, producing the codeword $\overline{\mathbf{e}}$. The error vector $\mathbf{e} \in \mathbb{F}_p$ is obtained as $\mathbf{e} = g^{\overline{\mathbf{e}}}$, where $g \in \mathbb{F}_p^*$ is the public generator element of order z creating the multiplicative subgroup E. Operatively, this step is carried out exponentiating g by each element of $\overline{\mathbf{e}}$.

Finally, the syndrome $\mathbf{s} \in \mathbb{F}_p^{n-k}$ is obtained as usual as $\mathbf{s} = \mathbf{e}\mathbf{H}^\top$, and is transmitted as part of the public key $\mathsf{pk} = (\mathtt{seed}_{\mathsf{pk}}, \mathbf{s})$, while the private key is $\mathsf{sk} = \mathtt{seed}_{\mathsf{sk}}$.

CROSS.SIGN

In Algorithm 15 are reported the operations performed to create a valid digital signature of an input message msg using the private key $\mathsf{sk} = \mathtt{seed}_{\mathsf{sk}}$ by performing t parallel executions of the CROSS-ID identification protocol. An overview of the steps are represented in Algorithm 13 and previously described in subsection 4.3.2.

Algorithm 15 CROSS.SIGN

Require: $\mathsf{sk} = \mathtt{seed}_{\mathsf{sk}} \in \{0,1\}^{2\lambda}$, $\mathsf{msg} \in \{0,1\}^*$
Ensure: $\mathsf{sig} = (\mathtt{salt} \in \{0,1\}^{2\lambda}, \mathtt{dig}_{\mathtt{cmt}} \in \{0,1\}^{2\lambda}, \mathtt{dig}_{\mathtt{chall}_2} \in \{0,1\}^{2\lambda}, \mathtt{path}, \mathtt{proof}, \mathtt{resp})$

▷ *Expanding the secret key (steps 2-7 in* CROSS.KEYGENERATION*)* ◁
$\overline{\mathbf{e}}, \overline{\mathbf{e}}_G, \mathbf{H}^\top, \overline{\mathbf{M}} \leftarrow$ EXPANDSK($\mathtt{seed}_{\mathsf{sk}}$) | $\overline{\mathbf{e}}, \mathbf{H}^\top \leftarrow$ EXPANDSK($\mathtt{seed}_{\mathsf{sk}}$)
▷ *Computing the commitments* ◁
$\mathtt{seed} \xleftarrow{\$} \{0,1\}^\lambda$
$\mathtt{salt} \xleftarrow{\$} \{0,1\}^{2\lambda}$
$(\mathtt{seed}[1], \ldots, \mathtt{seed}[t]) \leftarrow$ SEEDLEAVES($\mathtt{seed}, \mathtt{salt}$)
for $i \leftarrow 0$ **to** $t-1$ **do**
  ▷ *Compute the transformation* $\mathbf{v}[i]$ *such that* $\mathbf{v}[i] \odot \mathbf{e}'[i] = \mathbf{e}$ ◁
  $\overline{\mathbf{e}}'_G[i], \mathbf{u}'[i] \leftarrow \mathrm{CSPRNG}_{2t+i}(\mathtt{seed}[i] \| \mathtt{salt}, \mathbb{F}_z^m \times \mathbb{F}_p^n)$
  $\overline{\mathbf{e}}'[i] \leftarrow \overline{\mathbf{e}}'_G[i]\overline{\mathbf{M}}$
  $\overline{\mathbf{v}}_G[i] \leftarrow \overline{\mathbf{e}}_G - \overline{\mathbf{e}}'_G[i]$ | $\overline{\mathbf{e}}'[i], \mathbf{u}'[i] \leftarrow \mathrm{CSPRNG}_{2t+i}(\mathtt{seed}[i] \| \mathtt{salt}, \mathbb{F}_z^n \times \mathbb{F}_p^n)$
  $\overline{\mathbf{v}}[i] \leftarrow \overline{\mathbf{e}} - \overline{\mathbf{e}}'[i]$
  for $j \leftarrow 0$ **to** $n-1$ **do**
    $\mathbf{v}[i]_j \leftarrow g^{\overline{\mathbf{v}}[i]_j}$
  $\mathbf{u}[i] \leftarrow \mathbf{v}[i] \odot \mathbf{u}'[i]$
  $\mathbf{s}'[i] \leftarrow \mathbf{u}[i]\mathbf{H}^\top$
  $\mathtt{cmt}_0[i] \leftarrow \mathrm{HASH}_{2t+i}(\mathbf{s}'[i] \| \overline{\mathbf{v}}_G[i] \| \mathtt{salt})$ | $\mathtt{cmt}_0[i] \leftarrow \mathrm{HASH}_{2t+i}(\mathbf{s}'[i] \| \overline{\mathbf{v}}[i] \| \mathtt{salt})$
  $\mathtt{cmt}_1[i] \leftarrow \mathrm{HASH}_{2t+i}(\mathtt{seed}[i] \| \mathtt{salt})$
$\mathtt{dig}_{\mathtt{cmt}_0} \leftarrow$ TREEROOT($\mathtt{cmt}_0[1] \| \ldots \| \mathtt{cmt}_0[t]$)
$\mathtt{dig}_{\mathtt{cmt}_1} \leftarrow \mathrm{HASH}_0(\mathtt{cmt}_1[1] \| \ldots \| \mathtt{cmt}_1[t])$
$\mathtt{dig}_{\mathtt{cmt}} \leftarrow \mathrm{HASH}_0(\mathtt{dig}_{\mathtt{cmt}_0} \| \mathtt{dig}_{\mathtt{cmt}_1})$
▷ *Computing the first challenge* ◁
$\mathtt{dig}_{\mathsf{msg}} \leftarrow \mathrm{HASH}_0(\mathsf{msg})$
$\mathtt{dig}_{\mathtt{chall}_1} \leftarrow \mathrm{HASH}_0(\mathtt{dig}_{\mathsf{msg}} \| \mathtt{dig}_{\mathtt{cmt}} \| \mathtt{salt})$
$\mathtt{chall}_1 \leftarrow \mathrm{CSPRNG}_{3t-1}(\mathtt{dig}_{\mathtt{chall}_1}, (\mathbb{F}_p^*)^t)$
▷ *Computing the first response* ◁
for $i \leftarrow 0$ **to** $t-1$ **do**
  for $j \leftarrow 0$ **to** $n-1$ **do**
    $\mathbf{e}'[i]_j \leftarrow g^{\overline{\mathbf{e}}'[i]_j}$
  $\mathbf{y}[i] \leftarrow \mathbf{u}'[i] + \mathtt{chall}_1[i]\mathbf{e}'[i]$
▷ *Computing the second challenge* ◁
$\mathtt{dig}_{\mathtt{chall}_2} \leftarrow \mathrm{HASH}_0(\mathbf{y}[1] \| \ldots \| \mathbf{y}[t] \| \mathtt{dig}_{\mathtt{chall}_1})$
$\mathtt{chall}_2 \leftarrow \mathrm{CSPRNG}_{3t}(\mathtt{dig}_{\mathtt{chall}_2}, \mathcal{B}_{t,w})$
▷ *Computing the second response* ◁
$\mathtt{proof} \leftarrow$ TREEPROOF($\mathtt{cmt}_0[1] \| \ldots \| \mathtt{cmt}_0[t], \mathtt{chall}_2$)
$\mathtt{path} \leftarrow$ SEEDPATH($\mathtt{seed}, \mathtt{salt}, \mathtt{chall}_2$)
for $i \leftarrow 0$ **to** $t-1$ **do**
  if $\mathtt{chall}_2[i] = 0$ **then**
    $\mathtt{resp}[i]_0 \leftarrow (\mathbf{y}[i], \overline{\mathbf{v}}_G[i])$ | $\mathtt{resp}[i]_0 \leftarrow (\mathbf{y}[i], \overline{\mathbf{v}}[i])$
    $\mathtt{resp}[i]_1 \leftarrow \mathtt{cmt}_1[i]$
return $\mathsf{sig} = (\mathtt{salt}, \mathtt{dig}_{\mathtt{cmt}}, \mathtt{dig}_{\mathtt{chall}_2}, \mathtt{path}, \mathtt{proof}, \mathtt{resp})$

The algorithm starts by expanding the public matrices $\overline{\mathbf{M}}$ (only for the R-SDP(G) variant) and $\mathbf{H}^\top$, and the secret error vector $\overline{\mathbf{e}}$ in the same way as steps 2 through 7 of the CROSS.KEYGENERATION algorithm.

Commitments Subsequently, the signer needs to generate the two commitments $\mathtt{cmt}_0$ and $\mathtt{cmt}_1$ before constructing the two challenges. It starts by drawing two random bit strings: an λ-bit long signature randomness $\mathtt{seed}$ and a 2λ-bit long $\mathtt{salt}$ tagged to the current signature. Considering the randomness $\mathtt{seed}$ as the root of a binary tree, each node of the tree consists of a λ-bit string that is expanded via the CSPRNG function into two λ-bit child nodes, until there are t leaf nodes representing the randomness seeds $\mathtt{seed}[1], \ldots, \mathtt{seed}[t]$ used in each protocol execution.

Afterwards, each i-th randomness values $\mathtt{seed}[i]$ is expanded to an arbitrary long bit string that is used by a vector sampling algorithm to generate $\mathbf{u}'[i] \in \mathbb{F}_p^n$ and the exponents of the transformed error vector $\mathbf{e}'[i]$. When instantiating the R-SDP(G) problem, the sampled vector $\overline{\mathbf{e}}'_G[i] \in \mathbb{F}_z^m$ is expanded to $\overline{\mathbf{e}}'[i] \in \mathbb{F}_z^n$ by multiplying it with the public matrix $\overline{\mathbf{M}}$, whereas in R-SDP the vector $\overline{\mathbf{e}}'[i]$ is directly sampled from the expanded seed. Each transformation element $\mathbf{v}[i] \in \mathbb{F}_p^n$ is then computed by subtracting the transformed error vector $\overline{\mathbf{e}}'[i]$ to the secret key vector $\overline{\mathbf{e}}$, and using it as the exponents of the generator element g of the restricted group E. Moreover, in R-SDP(G) the more compact transformation $\overline{\mathbf{v}}_G[i] \in \mathbb{F}_z^m$ is computed similarly starting from the elements in $\mathbb{F}_z^m$. The transformed element $\mathbf{u}[i]$ are computed applying the transformation $\mathbf{v}[i]$ to the sampled vector $\mathbf{u}'[i]$, and the round syndrome $\mathbf{s}'[i] \in \mathbb{F}_p^{n-k}$ is computed as $\mathbf{u}[i]\mathbf{H}^\top$.

Finally, the commitments $\mathtt{cmt}_0$ and $\mathtt{cmt}_1$ can be computed. Each $\mathtt{cmt}_0[i]$ is the 2λ-bits long digest resulting from the absorption of the round syndrome $\mathbf{s}'[i]$, the smallest transformation element $\overline{\mathbf{v}}_G$ or $\overline{\mathbf{v}}$ depending on the hard problem variant, and the per-signature salt. The computed values are then assigned to the leafs of a Merkle tree. Each internal node of the tree is the hash digest resulting when absorbing the values contained in its two child nodes, and the value of the resulting root is $\mathtt{dig}_{\mathtt{cmt}_0}$. Each $\mathtt{cmt}_1[i]$ instead is the digest of the absorbed round seed $\mathtt{seed}[i]$ concatenated to the per-signature salt, and all the concatenated $\mathtt{cmt}_1[i]$ are absorbed in a single hash application to produce the digest $\mathtt{dig}_{\mathtt{cmt}_1}$. To conclude, $\mathtt{dig}_{\mathtt{cmt}}$ is computed as the digest of the concatenation of $\mathtt{dig}_{\mathtt{cmt}_0}$ and $\mathtt{dig}_{\mathtt{cmt}_1}$.

First challenge The verifier $\mathcal{V}$ requires to generate the first challenge $\mathtt{chall}_1[i] \in \mathbb{F}_p^*$. In the FS framework, this choice is derived deterministically from the output of a collision-resistant one-way function having as input the message msg and the prover's commitments. CROSS uses the same instance of HASH as the one-way function, absorbing the digest of the message to be signed $\mathtt{dig}_{\mathsf{msg}}$, the digest of the commitments $\mathtt{dig}_{\mathtt{cmt}}$, and the per-signature salt, to generate the input material $\mathtt{dig}_{\mathtt{chall}_1}$ for the CSPRNG XOF used to sample the t random challenges $\mathtt{chall}_1$.

The prover $\mathcal{P}$ then expands each transformed round error $\overline{\mathbf{e}}'[i]$ into $\mathbf{e}'[i]$ by exponentiating the public generator element g by each element of $\overline{\mathbf{e}}'[i]$. Finally, each response to

the first challenge is computed as $\mathbf{y}[i] = \mathbf{u}'[i] + \texttt{chall}_1[i]\mathbf{e}'[i]$.

Second challenge Now the verifier $\mathcal{V}$ needs to generate the second challenge $\texttt{chall}_2[i] \in \{0,1\}$. The CROSS-ID is made non-interactive by determining the challenge from the one-way function output having as input the protocol transcript. Since all the previous "virtually" exchanged messages are contained in $\texttt{dig}_{\texttt{chall}_1}$, it is sufficient to absorb it concatenated to all the responses to the first challenges $\mathbf{y}[i]$ with a hash function, resulting in the digest $\texttt{dig}_{\texttt{chall}_2}$. The expansion of $\texttt{dig}_{\texttt{chall}_2}$ with the CSPRNG function is then used to sample the t second challenges $\texttt{chall}_2[i]$. Such challenge is a random t-bits long binary vector with Hamming weight w, thus $\mathcal{B}_{t,w}$ denotes the set of all the possible binary strings of length t and weight w. The second challenges are used to determine the seed path (`path`) and the Merkle proof (`proof`) as described in Figure 4.3, and to assemble the signature with the correct elements. In the CROSS *fast* variant the `path` and `proof` bit strings contain just the leafs at index i of their trees if $\texttt{chall}_2[i] = 1$, as no meaningful compression can be obtained with these data structures when $w \approx t/2$. If $\texttt{chall}_2[i] = 0$, the signature will contain the commitment $\texttt{cmt}_1[i]$ and the required elements to let the verifier compute $\texttt{cmt}_0[i]$. In case $\texttt{chall}_2[i] = 1$, then the signer can easily compute $\texttt{cmt}_1[i]$ from the seed expanded from `path`, while $\texttt{cmt}_0[i]$ is embedded in `proof`. Particular care is needed in this stage, because an incorrect assembly of the signature, possibly caused by a simple fault attack, can lead to an easy retrieval of the private key [Mon+24].

CROSS.VERIFY

In Algorithm 16 is described the algorithm checking a signature sig of a message msg using the public key $\mathsf{pk} = (\texttt{seed}_{\mathsf{pk}} \in \{0,1\}^{2\lambda}, \mathbf{s} \in \mathbb{F}_p^{n-k})$, and returning $\top$ on success, and $\perp$ on failure. The operations mimics the verifier $\mathcal{V}$ behavior in the t parallel executions of the CROSS-ID identification protocol (Algorithm 13, subsection 4.3.2).

The algorithm starts by expanding the public matrices $\mathbf{H}^\top$ and $\overline{\mathbf{M}}$, the latter only when the R-SDP(G) variant of the hard problem is used, starting from the public key seed $\texttt{seed}_{\mathsf{pk}}$. Afterwards, the two challenges $\texttt{chall}_1 \in (\mathbb{F}_p^*)^t$ and $\texttt{chall}_2 \in \mathcal{B}_{t,w}$ are computed via the digests and the salt value transmitted in the signature.

All the w out-of t seeds used in the CROSS-ID protocol executions are expanded from the seed paths, as explained in Figure 4.3, and are used to compute the first commitments $\texttt{cmt}_1[i]$ when $\texttt{chall}_2[i] = 1$. For the same challenge value, the transformed error $\mathbf{e}'[i]$ is expanded from their compressed forms, and used to compute the first response value $\mathbf{y}[i] = \mathbf{u}'[i] + \texttt{chall}_1[i]\mathbf{e}'[i]$.

In the $t - w$ cases when $\texttt{chall}_2[i] = 0$, then the transmitted transformation $\overline{\mathbf{v}}[i]$ is checked and expanded to compute $\mathbf{v}'[i] = \mathbf{v}[i] \odot \mathbf{y}[i]$, which is then used to retrieve the transformed syndrome $\mathbf{s}'[i] = \mathbf{y}'[i]\mathbf{H}^\top - \texttt{chall}_1[i]\mathbf{s}$. Finally, the verifier can recompute the missing $\texttt{cmt}_0[i]$, which is used to fill the missing Merkle tree leafs, before recomputing the root value $\texttt{dig}_{\texttt{cmt}_0}$. Similarly, the $\texttt{cmt}_1[i]$ commitments computed when $\texttt{chall}_2[i] = 1$ or received in the signature in the opposite case are concatenated

Algorithm 16 CROSS.VERIFY

Require: $\mathsf{pk} = (\mathtt{seed}_{\mathsf{pk}} \in \{0,1\}^{2\lambda}, \mathbf{s} \in \mathbb{F}_p^{n-k})$, $\mathsf{msg} \in \{0,1\}^*$,
$\mathsf{sig} = (\mathtt{salt} \in \{0,1\}^{2\lambda}, \mathtt{dig}_{\mathtt{cmt}} \in \{0,1\}^{2\lambda}, \mathtt{dig}_{\mathtt{chall}_2} \in \{0,1\}^{2\lambda}, \mathtt{path}, \mathtt{proof}, \mathtt{resp})$
Ensure: $\mathtt{valid} = \{\top, \bot\}$

▷ *Recover the public key* ◁
$\overline{\mathbf{W}}, \mathbf{V}^\top \leftarrow \text{CSPRNG}_{3t+2}(\mathtt{seed}_{\mathsf{pk}}, \mathbb{F}_z^{m\times(n-m)} \times \mathbb{F}_p^{k\times(n-k)})$ | $\mathbf{V}^\top \leftarrow \text{CSPRNG}_{3t+2}(\mathtt{seed}_{\mathsf{pk}}, \mathbb{F}_p^{k\times(n-k)})$
$\overline{\mathbf{M}} \leftarrow [\overline{\mathbf{W}}, \mathbf{I}_m]$
$\mathbf{H}^\top \leftarrow [\mathbf{V}^\top \mid \mathbf{I}_{n-k}]$
▷ *Compute the challenges* ◁
$\mathtt{dig}_{\mathsf{msg}} \leftarrow \text{HASH}_0(\mathsf{msg})$
$\mathtt{dig}_{\mathtt{chall}_1} \leftarrow \text{HASH}_0(\mathtt{dig}_{\mathsf{msg}} \| \mathtt{dig}_{\mathtt{cmt}} \| \mathtt{salt})$
$\mathtt{chall}_1 \leftarrow \text{CSPRNG}_{3t-1}(\mathtt{dig}_{\mathtt{chall}_1}, (\mathbb{F}_p^*)^t)$
$\mathtt{chall}_2 \leftarrow \text{CSPRNG}_{3t}(\mathtt{dig}_{\mathtt{chall}_2}, \mathcal{B}_{t,w})$
▷ *Compute the commitments* ◁
$(\mathtt{seed}[i])_{i:\mathtt{chall}_2[i]=1} \leftarrow \text{REBUILDLEAVES}(\mathtt{path}, \mathtt{chall}_2, \mathtt{salt})$
for $i \leftarrow 0$ **to** $t-1$ **do**
  if $\mathtt{chall}_2[i] = 1$ **then**
    $\mathtt{cmt}_1[i] \leftarrow \text{HASH}_{2t+i}(\mathtt{seed}[i] \| \mathtt{salt})$
    $\overline{\mathbf{e}}'_G[i], \mathbf{u}'[i] \leftarrow \text{CSPRNG}_{2t+i}(\mathtt{seed}[i] \| \mathtt{salt}, \mathbb{F}_z^m \times \mathbb{F}_p^n)$ | $\overline{\mathbf{e}}'[i], \mathbf{u}'[i] \leftarrow \text{CSPRNG}_{2t+i}(\mathtt{seed}[i] \| \mathtt{salt}, \mathbb{F}_z^n \times \mathbb{F}_p^n)$
    $\overline{\mathbf{e}}'[i] \leftarrow \overline{\mathbf{e}}'_G[i]\overline{\mathbf{M}}$
    for $j \leftarrow 0$ **to** $n-1$ **do**
      $\mathbf{e}'[i]_j \leftarrow g^{\overline{\mathbf{e}}[i]_j}$
    $\mathbf{y}[i] \leftarrow \mathbf{u}'[i] + \mathtt{chall}_1[i]\mathbf{e}'[i]$
  if $\mathtt{chall}_2[i] = 0$ **then**
    $\mathtt{cmt}_1[i] \leftarrow \mathtt{resp}[i]_1$
    $(\mathbf{y}[i], \overline{\mathbf{v}}_G[i]) \leftarrow \mathtt{resp}[i]_0$ | $(\mathbf{y}[i], \overline{\mathbf{v}}[i]) \leftarrow \mathtt{resp}[i]_0$
    Check if $\overline{\mathbf{v}}_G[i] \in \mathbb{F}_z^m$ | Check if $\overline{\mathbf{v}}[i] \in \mathbb{F}_z^n$
    $\overline{\mathbf{v}}[i] \leftarrow \overline{\mathbf{v}}_G[i]\overline{\mathbf{M}}$
    for $j \leftarrow 0$ **to** $n-1$ **do**
      $\mathbf{v}[i]_j \leftarrow g^{\overline{\mathbf{v}}[i]_j}$
    $\mathbf{y}'[i] \leftarrow \mathbf{v}[i] \odot \mathbf{y}[i]$
    $\mathbf{s}'[i] \leftarrow \mathbf{y}'[i]\mathbf{H}^\top - \mathtt{chall}_1[i]\mathbf{s}$
    $\mathtt{cmt}_0[i] \leftarrow \text{HASH}_{2t+i}(\mathbf{s}'[i] \| \overline{\mathbf{v}}_G[i] \| \mathtt{salt})$ | $\mathtt{cmt}_0[i] \leftarrow \text{HASH}_{2t+i}(\mathbf{s}'[i] \| \overline{\mathbf{v}}[i] \| \mathtt{salt})$
▷ *Check the digests* ◁
$\mathtt{dig}_{\mathtt{cmt}_0} \leftarrow \text{RECOMPUTEROOT}(\mathtt{cmt}_0, \mathtt{proof}, \mathtt{chall}_2)$
$\mathtt{dig}_{\mathtt{cmt}_1} \leftarrow \text{HASH}_0(\mathtt{cmt}_1[1] \| \dots \| \mathtt{cmt}_1[t])$
$\mathtt{dig}'_{\mathtt{cmt}} \leftarrow \text{HASH}_0(\mathtt{dig}_{\mathtt{cmt}_0} \| \mathtt{dig}_{\mathtt{cmt}_1})$
$\mathtt{dig}'_{\mathtt{chall}_2} \leftarrow \text{HASH}_0(\mathbf{y}[1] \| \dots \| \mathbf{y}[t] \| \mathtt{dig}_{\mathtt{chall}_1})$
if $\mathtt{dig}_{\mathtt{cmt}} \neq \mathtt{dig}'_{\mathtt{cmt}} \vee \mathtt{dig}_{\mathtt{chall}_2} \neq \mathtt{dig}'_{\mathtt{chall}_2}$ **then**
  return $\mathtt{valid} = \bot$
return $\mathtt{valid} = \top$

and used as input to the cryptographic hash function, producing the digest $\texttt{dig}_{\texttt{cmt}_1}$. The same process is repeated for $\mathbf{y}[i]$, producing the digest $\texttt{dig}_{\mathbf{y}}$. If any of these digests do not match the ones in the signature sig, the verification process fails.

CHAPTER 5

Cryptographic hash functions

Foundation for many security protocols, a cryptographic hash function, hereafter referred to as a hash function, is a cryptographic primitive that provides essential guarantees to many applications for data integrity, authentication, and commitment. Given its valuable properties, it is also a central element in all the post-quantum asymmetric protocols for both the KEM and DS schemes.

A cryptographic hash function is a non-injective map from the set of binary strings with any cardinality to the set of binary strings of fixed length $n \in \mathbb{N}$ called digests or hash values:

$$\text{HASH} : \{0,1\}^* \mapsto \{0,1\}^n \tag{5.1}$$

Given the definition of a cryptographic hash function, it is easy to see that there are plenty of collisions. The challenge in creating a cryptographic hash function is making practically unfeasible to find and generate collisions on purpose, while being deterministic and computationally efficient to apply over large amount of data. Let $\mathcal{M}$ be the set of messages (message space), and $\mathcal{D}$ the set of digests (digest space), so that $\text{HASH} : \mathcal{M} \mapsto \mathcal{D}$. A secure cryptographic hash function has the following properties:

Preimage resistance one-way property of a function, is deemed infeasible to reverse the process and retrieve the original input $\mathbf{m} \in \mathcal{M}$ starting from the hash output $\mathbf{d} = \text{HASH}(\mathbf{m})$. An attack can ideally succeed with probability $\leq 2^{-n}$.

Second preimage resistance also known as weak collision resistance, is impossible to produce a input $\mathbf{m}_2 \in \mathcal{M}$ having the same hash digest $\mathbf{d} = \text{HASH}(\mathbf{m}_1)$ of any

other known input $\mathbf{m}_1 \in \mathcal{M}$, $\mathbf{m}_1 \neq \mathbf{m}_2$. An attack can ideally succeed with probability $\leq 2^{-n}$.

Collision resistance is impractical to search for any two distinct messages $\mathbf{m}_1, \mathbf{m}_2 \in \mathcal{M}$ having the same digest $\mathbf{d} = \text{HASH}(\mathbf{m}_1) = \text{HASH}(\mathbf{m}_2)$. An attack can succeed with probability $\leq 2^{-n/2}$ due to the birthday paradox.

Avalanche effect a single bit flip in the input message $\mathbf{m} \in \mathcal{M}$ drastically changes its digest $\mathbf{d} \in \mathcal{D}$, ideally flipping each output bit with 50% probability

Random oracle for any given input, the hash function should produce a digest $\mathbf{d} \xleftarrow{\$} \mathcal{D}$ appearing as if it were picked from a uniform distribution on strings of the same length. Any successive query using the same message must return the same digest (deterministic)

5.1 Merkle-Damgård construction

Historically, most of the cryptographic hash functions are based on the Merkle-Damgård (MD) construction, depicted in Figure 5.1, which combines an injective *padder* function with a one-way collision-resistant *compressor* function, and an injective *finalizer* function:

$$\text{PADDER} : \{0,1\}^k \mapsto \{0,1\}^r \tag{5.2}$$

$$\text{COMPRESSOR} : \{0,1\}^{c+r} \mapsto \{0,1\}^c \tag{5.3}$$

$$\text{FINALIZER} : \{0,1\}^c \mapsto \{0,1\}^n \tag{5.4}$$

where $k, r, c \in \mathbb{N}$, $r > k > 0$, and r, c are constants.

The input message bit string is split in r-bits chunks $\mathbf{m} = \mathbf{m}_0 \| \mathbf{m}_1 \| \ldots \| \mathbf{m}_{s-1}$, and the last block $\mathbf{m}_{s-1}$ containing $k < r$ bits gets expanded using the padder function. The compressor functions has an inner state of c bits, initialized with a known Initialization Vector (IV), and r-bits message chunks are mixed in the inner state by the round-based compressor function. The compressor function could be for example an instance of a block cipher. After mixing all message chunks, the finalizer function produce the digest output of the correct size.

MD5 published in 1992, has $c = n = 128$, $r = 512$, and the compressor function composed by 4 rounds composed with 16 operations working on 32-bit operands. MD5 input space is limited to messages with up to $2^{64} - 1$ bits (≈ 2048 PiB) due to its padding scheme. Collisions are are widely reported and can be generated with just 2^{18} operations [XLF13], and using it for cryptographic purposes its a major security flow.

SHA-0, SHA-1 the first presented in 1993, and promptly decommissioned in 1995 in favour of the latter, have $c = n = 160$ and $r = 512$. The compressor function is composed by 4 rounds of 20 operations each, and uses 32-bit operands. SHA-0 and

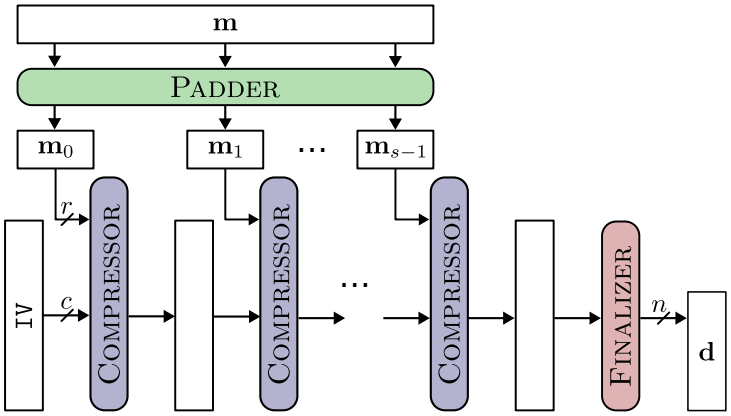


Figure 5.1: *Merkle-Damgård (MD) construction*

SHA-1 input space are limited to messages with up to $2^{64}-1$ bits. Collision can be crafted with 2^{34} and 2^{63} operations for SHA-0 and SHA-1 [Ste12], respectively, thus it is not recommended to use them in security-related contexts.

SHA-2 defines two main functions `SHA-256` ($c = 256$, $n = 256$, $r = 512$) and `SHA-512` ($c = 512$, $n = 512$, $r = 1024$). The compressor function composed by 64 (or 80) rounds operations among 32-bits (or 64-bits) operands, for `SHA-512` and `SHA-512`, respectively. There are variants with smaller digest size $n = 224$ and $n = 384$ derived from `SHA-256` and `SHA-512`, respectively. To circumvent the decrease in the second preimage resistance when long input messages are used, which is caused by the digest being the entire final state, two new variants producing $n = 256, 244$ digests using the `SHA-512` algorithm were later introduced.

MD-based hash functions proved to be weak against length-extension attack, when an attacker extends an input message with some specially crafted data to create a collision. A viable workaround requires to encode the length of the message at the beginning of the message or in the padding scheme, or not exposing the entire inner state as digest (thus having $n < c$).

5.2 Sponge construction

Given the current situation of length-extension attacks afflicting the MD, the creation of a secure Keyed Hash Algorithm based on Secure Hash Algorithm (SHA) is a delicate research question. For that reason, a call for new proposals from NIST that are not based on the MD construction led to the publication of the new standard SHA-3 in 2015. The SHA-3 winner Keccak employs a flexible and customizable sponge construction, depicted in Figure 5.2, which allows the construction of the XOF SHAKE, the Authenticated Encryption with Associated Data (AEAD) schemes Keyak and Ketje, and the tree-style hashing mode Sakura.

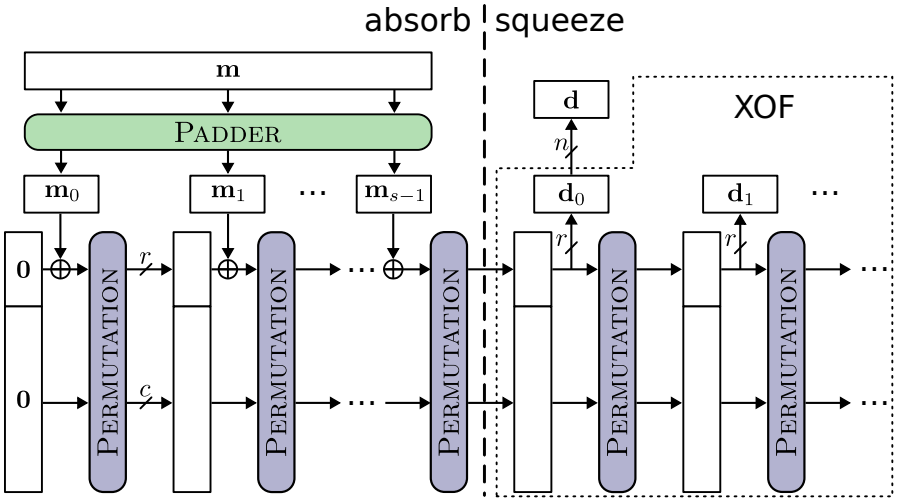


Figure 5.2: *Sponge construction*

Similarly to MD constructions (Equation 5.2), there is an injective *padder* function that splits the input message in constant-size r-bits blocks, while also providing domain separation functionality. At the core of the sponge construction there is a *permutation* function working on a $b = r + c$ bits internal state.

$$\text{PADDER} : \{0,1\}^k \mapsto \{0,1\}^r \tag{5.5}$$

$$\text{PERMUTATION} : \{0,1\}^{c+r} \mapsto \{0,1\}^{c+r} \tag{5.6}$$

where $k, r, c \in \mathbb{N}$, $r > k > 0$, and r, c are constants.

The *rate* (r) parameter defines the size of the blocks into which input and output data streams are partitioned to be processed or yielded, while the *capacity* (c) parameter denotes the portion of the state which is never output, providing its security margin. The permutation function is applied every time a r-bits input block is absorbed into the first r bits of the state via a bitwise `xor` operation. After the last input message block is absorbed, the first r bits of the state are the digest of the cryptographic hash construction.

An eXtendable-Output Function (XOF) is a map from the set of binary strings with any cardinality to a set of binary strings of length e. Contrarily to cryptographic hash functions defined as Equation 5.1, the length e is not a constant and can vary depending on the needs.

$$\text{XOF} : \{0,1\}^* \mapsto \{0,1\}^e \tag{5.7}$$

A common use case is the expansion of a small random seed in a longer random bit string maintaining the same entropy of the input data.

In case of the construction of a XOF using the sponge construction, after the initial absobption of the message, the permutation function is applied as many time as required in the squeeze phase to produce the output of the required size, reading at most r bits of

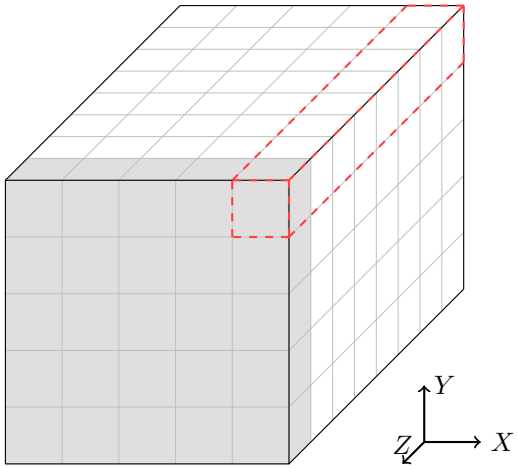


Figure 5.3: *Three-dimensional representation of the Keccak state, organized as a* 5×5 *grid of lanes with* w *length (dashed red box of* 8 *bits in the example). The sub-state of* 25 *bits with a fixed* z *coordinate is called slice, here highlighted in gray color.*

the internal state each time. If the length of the output stream is not a multiple of r, the last output block is truncated.

5.2.1 Keccak scheme

The Keccak[r,c] algorithm [Ber+11] employs the aforementioned sponge construction, and defines the round-based KECCAK-$f[r+c]$ permutation function (Equation 5.8). The internal state is logically partitioned in a three-dimensional $5 \times 5 \times w = r + c$ structure, as represented in Figure 5.3. Each one of the $12 + 2 \cdot \log_2 w$ rounds is composed by 5 steps working on the whole state:

$$\text{KECCAK-}f \triangleq (\theta \circ \rho \circ \pi \circ \chi \circ \iota)^{12+2\cdot\log_2 w} \tag{5.8}$$

θ provides a high level of diffusion, each bit at the output (input) of a round depends on (affects) 31 bits at its input (output)

ρ speeds up the diffusion between slices (z dimension)

π provides dispersion aimed at long-term diffusion, removing the exhibition of periodic trails of low weight

χ is a simple degree 2 non-linear function, providing protection against differential analysis

ι breaks the round symmetry mixing round-specific constants in the state, protecting from slide attacks

The operations used in each step are just `xor`, `and`, and `not` Boolean operations, with the only CPU word-dependent operation being the vector rotations. All the steps, with

Table 5.1: *Instances of the Keccak[r, c]($\mathbf{m}\|\texttt{dstr}$) algorithm, in terms of output size n, rate r, capacity c, domain separation string* `dstr`*, and input message* $\mathbf{m}$*, as defined in FIPS 202 standard from NIST.*

Algorithm	Output size	Rate	Capacity	Domain string
	n	r	c	`dstr`
`SHA3-224`	224	1152	448	`01`
`SHA3-256`	256	1088	512	`01`
`SHA3-384`	384	832	768	`01`
`SHA3-512`	512	576	1024	`01`
`SHAKE128`	e	1344	256	`1111`
`SHAKE256`	e	1088	512	`1111`

the exception for ι, are defined independently of the state z dimension, therefore they can be applied to different state sizes (*matryoshka* structure). The Keccak documentation of the submission to NIST SHA standardization defines $w \in \{1, 2, 4, 8, 16, 32, 64\}$, producing a state with size up to 1600 bits.

NIST standardized a subset of the Keccak scheme in FIPS 202 [PM15] under the name SHA-3. The standard permits only the instantiation of KECCAK-$f[1600]$, defines four hash functions, named as `SHA3-224`, `SHA3-256`, `SHA3-384`, and `SHA3-512`, depending on the length of the digest. Moreover, two XOFs are introduced, named as Secure Hash Algorithm Keccak (SHAKE), with either a 128-bit or a 256-bit security level against all class of attacks, and a configurable output length e. The padder function ensures that the absorbed message $\mathbf{m}$, after attaching a domain-separation string `dstr` for each different function instance, is always a multiple of r by appending the `10*1` binary string:

$$\mathbf{m}_0\|\mathbf{m}_1\|\ldots\|\mathbf{m}_{s-1} = \mathbf{m}\|\texttt{dstr}\|1\|0^j\|1, \qquad 0 \leq j < r \tag{5.9}$$

The rate, capacity, domain-separation strings, and hash output size for the SHA-3 algorithms are contained in Table 5.1, and the comparison among the previously presented cryptographic hash functions are reported in Table 5.2.

5.2.2 SHA-3 hardware designs

The authors of Keccak gathered in a report [Ber+12] all the details of the hardware and software implementations. Considering the hardware designs, three categories of implementations are defined:

High-speed each clock cycle k rounds of the KECCAK-f permutation are performed on the whole state, concluding the permutation in $^{12+2\cdot\log_2 w}/_k$ clock cycles. In case of the Keccak instance used in SHA-3, the 1600-bits state must be implemented entirely in FFs or latches, which may be prohibitive in some contexts.

Mid-range this core applies the fixed permutation steps ρ and π on the whole state, as in hardware are achieved simply via re-wiring and basically for free, and the computation steps θ, χ, and ι are applied on a group of k slices, saving gate resources.

Table 5.2: *Comparison of preimage, second preimage and collision resistance of NIST's standardized SHA from the FIPS 202 document.* $\mathrm{L}(\mathbf{m})$ *is the function* $\lceil \log_2(\textsc{len}(\mathbf{m})/b) \rceil$*, where* $b \in \mathbb{N}$ *is the block length of the function and* $\textsc{len}(\mathbf{m})$ *is the length in bit of the message* $\mathbf{m}$*.*

Algorithm	Digest size	Attack success probability (2^{-x})		
		Collision	Preimage	Second preimage
SHA-1	160	< 80	160	$160 - \mathrm{L}(\mathbf{m})$
SHA-224	224	112	224	$\min(224, 256 - \mathrm{L}(\mathbf{m}))$
SHA-512/224	224	112	224	224
SHA-256	256	128	256	$256 - \mathrm{L}(\mathbf{m})$
SHA-512/256	256	128	256	256
SHA-384	384	192	384	384
SHA-512	512	256	512	$512 - \mathrm{L}(\mathbf{m})$
SHA3-224	224	112	224	224
SHA3-256	256	128	256	256
SHA3-384	384	192	384	384
SHA3-512	512	256	512	512
SHAKE128	n	$\min(n/2, 128)$	$\geq \min(n, 128)$	$\min(n, 128)$
SHAKE256	n	$\min(n/2, 256)$	$\geq \min(n, 256)$	$\min(n, 256)$

The state is kept in 25 distinct 8×8 Random-Access Memorys (RAMs). Round steps have to be rescheduled adding an artificial first round, and some multiplexers to differentiate the first and last round steps. Additionally, a register containing the parity of the previous θ step is required. The overall latency of the permutation is $(12 + 2 \cdot \log_2 w) + (12 + 2 \cdot \log_2 w + 1)(w/k)$.

Low-area compact solution suitable for smart cards, containing the whole Keccak state in a single RAM, and working on lanes. Only few registers are required for storing temporary variables. Each KECCAK-f permutation takes thousands of clock cycles.

Considering FPGA targets, the mid-range core is an interesting solution as it uses LUT as distributed memories, taking few hundreds of LUTs to store the Keccak state, significantly improving the overall area cost of the core. However, this approach is not feasible in ASIC implementations, where multiple small memory macros needs to be compiled, characterized, and replaced in the netlist, with the associated extra cost required for implementing the memory ports addressing logic. For this target, the simplicity of the high-speed variant provides invaluable benefits, as the state can be implemented with latches to save silicon area. Furthermore, the critical path of the full KECCAK-f permutation round is only few gates deep, making the place and route step the most challenging task of the implementation.

For these reasons, I opted for the creation of a high-speed core variant, starting from [Xin18], with several improvements:

XOF mode the padding function and the control logic is updated to support the XOF mode, supporting the SHAKE algorithm. The implemented algorithm is selected

at synthesis time determining the rate/capacity ratio, simplifying the resulting design.

Stream width the streams used to write the message $\mathbf{m}$ and read the digest $\mathbf{d}$ are generalized with an arbitrary width B fixed during the synthesis. This parametrization is necessary for the following optimizations

I/O buffer introducing an separated r-sized input buffer from the $r + c$-sized state allows the parallel execution of the padder module, preparing the message block $\mathbf{m}_{i+1}$, with the absorption of the previously processed block $\mathbf{m}_i$. If $\lceil r/B \rceil \leq 12+2\cdot\log_2 w$, then the module implementing the KECCAK-f permutation is always busy absorbing the input, maximizing the efficiency of the design. Similarly, when performing the XOF SHAKE, having a r-sized input buffer from the $r + c$-sized state allows the parallel execution of the squeeze operation of the digest block $\mathbf{v}_{i+1}$, while the previously processed block $\mathbf{v}_i$ is read out. If $\lceil r/B \rceil \approx 12 + 2 \cdot \log_2 w$, the efficiency of the design is maximized. Note that the absorb and squeeze phases of the sponge construction are mutually exclusive, thus the input/output buffers are merged in a single buffer instance, which is efficiently implemented as a shift register to minimize the number of employed multiplexers.

Fast absorption quite often the absorbed inputs of the SHAKE XOF are seeds with a fixed size. Therefore, it is possible to enhance the padding module such that, for these specific input sizes, it fast-forwards the shift register buffer with the correct padded input, to immediately start the f-permutation in the following clock cycle.

Unroll factor performing k rounds of the KECCAK-f permutation in the same clock cycle clearly reduces the overall clock cycles required to complete the permutation, and the area requirement of just the permutation module increases by k times. However, the gates in the critical path increases by k, and the place and route implementation tasks becomes more difficult, causing a reduction of the maximum work frequency by more than k times. Therefore, the efficiency of the design requires wider stream widths B and may decrease after a certain unroll factor k. Considering a hardware design with a single clock domain, all the modules are driven with the same clock signal. Therefore, the module with the longest critical path is imposing the clock frequency to the other modules. Unrolling the permutation function in order to have a maximum frequency matching the one of the design allows to achieve the maximum throughput possible. The unroll factor k must divide evenly the number of the permutation rounds, i.e. $1, 2, 3, 4, 6, 8, 12, 24$ in case $r + c = 1600$.

The module implementing the padder function from Equation 5.9 works with the input/output buffer. For efficiency reasons, the buffer is a shift register with B-bits blocks matching the width of the streams. The read and write access to the buffer via multiplexers would improve the latency and efficiency in case small messages are

Table 5.3: *Synthesis results of* `SHAKE256` *module for the AMD Artix-7* `xc7a200t` *FPGA platform (results for -2 speed grade chip are denoted with *, otherwise are referred to -3 speed grade chip). The design variant defines the fold factor for mid-range core types (number of parallel slices processed each clock cycle), or the unroll factor for high-speed cores (number of* KECCAK-f *rounds performed each clock cycle). The latency, efficiency and throughput figures for* `SHA3-256` *are referred to the absorption of a* 320*-bits message to generate the* r*-bits of the digest block* $\mathbf{d}_0$.

Design			Area			Frequency	SHA3-256			
Ref.	Type	Variant	LUT	FF	eSlice	MHz	CC	μs	AT prod.	Gb/s
[Wan+20]	mid-range*	×1	811	490	203	178	2681	15.06	3.06	0.07
[Des+23]	mid-range	×1	1437	498	360	163	2408	14.77	5.32	0.07
[Wan+20]	mid-range*	×2	908	450	227	163	1353	8.30	1.88	0.13
[Des+23]	mid-range	×2	1558	466	390	167	1206	7.22	2.82	0.15
[Wan+20]	mid-range*	×4	1069	361	268	158	680	4.30	1.15	0.25
[Des+23]	mid-range	×4	1625	370	407	157	604	3.85	1.57	0.28
[Wan+20]	mid-range*	×8	1466	270	367	164	337	2.05	0.75	0.52
[Des+23]	mid-range	×8	1958	280	490	158	302	1.91	0.94	0.56
[Wan+20]	mid-range*	×16	2401	226	601	165	168	1.02	0.61	1.04
[Des+23]	mid-range	×16	2819	236	705	164	150	0.91	0.64	1.17
[Wan+20]	mid-range*	×32	4436	180	1109	161	85	0.53	0.59	2.00
[Des+23]	mid-range	×32	4797	191	1200	166	74	0.45	0.53	2.36
This work	high-speed	×1	5589	2744	1398	237	29	0.12	0.17	8.85
This work	high-speed	×2	10571	2736	2643	125	17	0.14	0.36	7.59
This work	high-speed	×3	12700	2719	3175	74	13	0.18	0.56	5.90
This work	high-speed	×4	15472	2717	3868	53	11	0.21	0.80	5.06
This work	high-speed	×6	22725	2715	5682	28	9	0.32	1.83	3.32
This work	high-speed	×8	20262	2714	5066	26	8	0.31	1.56	3.43
This work	high-speed	×12	28782	2713	7196	14	7	0.50	3.60	2.12
This work	high-speed	×24	55403	2714	13851	6	6	1.00	13.85	1.06

absorbed, but given the practical size of the rate r and the block size B, the additional area cost and increased complexity would be non-negligible.

In Table 5.3 the core designs for FPGA platforms executing the `SHAKE256` algorithm using a 320-bits seed and producing a single r-bits digest block are compared in terms of maximum working frequency, area, latency, throughput, and efficiency. All the cores in Table 5.3 are specialized at synthesis time for a specific SHA-3 scheme. The adopted methodology for the evaluation of the design are detailed in section 2.2. For the designs in the mid-range core category, the two works from [Wan+20] and [Des+23] present various instances processing multiple state slices in parallel. The results are quite similar, except for the LUT usage when few slices are processed in parallel, showing a linear increase in area requirement when processing more slices, and a slight frequency drop for the larger designs. The throughput and the efficiency indicator, computed as the `Area×Time` product, confirm the improvements when more slices are processed simultaneously.

By contrast, the results obtained from the developed high-speed core clearly indicate that processing one round per clock cycle is the best solution in terms of efficiency

and throughput, having a $\approx 4\times$ improvement with respect to the mid-range core. The achieved maximum frequency is remarkably higher ($+38\%$) and the designs require a similar amount eSlices. This is due to the proportion of LUT and FF resources composing a single Slice, which in this case the contribution is dominated by the LUT usage. The substantial difference in FF usage is due to the Keccak state being implemented entirely in FFs, instead of relying on few hundreds of LUTs used as distributed memories. Computing more KECCAK-f permutation rounds in the same clock cycle does not produce an improvement in terms of efficiency and throughput as in the case of the mid-range core, due to a substantial frequency drop caused by the routing congestion in the FPGA fabric. Nonetheless, these variants can be a useful solution in case the Keccak core is instantiated in a design with a single clock region. Notice that using the SHAKE core as a regular Extendable-Output Function (XOF) the performance is severely limited by the read speed of the sink unit reading out the stream bits from the rate section of the Keccak state. In such case, the width of the AXI Stream must be carefully considered to obtain the best efficiency and leverage the whole potential of the unrolled designs.

CHAPTER 6

Element generation

A major design challenge for cryptographic schemes is the secure generation of new elements from the prescribed sample space from some distribution. The way a cryptoscheme samples the elements plays a crucial role in the security and efficiency. Special attention is also required to correctly implement the sampling process of sensitive elements composing the private key or using private key-derived material as a source of randomness without leaking information in the form of execution time latency. This is crucial to comply with the IND-CCA2 property, and avoid that an attacker producing mangled ciphertexts is able to determine a failure due to a different the execution time, bypassing the implicit rejection mechanism. Since the presented algorithms are analyzed to be implemented securely in hardware, the time variances caused by data-dependent memory access due to the presence of data caches is omitted in this chapter.

Typical distributions used by PQC schemes are the uniform distribution, where every element from the set has equal probability of being chosen, the discrete Gaussian distribution, necessary to select elements with a probability related to their distance to another element, and the binomial distribution, used to efficiently approximate the discrete Gaussian distribution. Notably, lattice-based schemes rely on discrete Gaussian distribution (FALCON) binomial distribution (CRYSTALS-Dilithium, CRYSTALS-Kyber), or custom algorithms (NTRU) to sample errors with specific properties guaranteeing the correctness and security of the scheme. The sampling algorithms used by the cryptoschemes considered in this thesis, NTRU, HQC, and CROSS, are only generating elements from the uniform distribution from the sample space.

One of the key components in element sampling algorithms is a good quality Random Number Generator (RNG). A TRNG extracts the entropy from physical random processes, such as thermal noise, metastability, or phase noise jitter, producing a high-quality yet low throughput source of randomness. The resulting bit stream can pass through one or more post-processing stages to improve the quality of the random bits, which are processed by a health check routine to detect failures and anomalies, such as long runs of 0 or 1 bits or more sophisticated tests. This last step is critical since an attacker may be able to modify the environment conditions in which the device is running, significantly altering the physical process generating the entropy, reducing the quality, or even making it a deterministic process. Efficient techniques compromising the entropy source are lowering the environment temperature using liquid nitrogen [Sou+11], voltage regulation or spike injection on the power line [Mar+15], and locking ring oscillators injecting a high-frequency signal [MM09].

While properly designed TRNGs fit the description of an ideal source of randomness, their low throughput is a serious limitation for PQC schemes using a significant amount of randomness to sample random elements. To this end, an appropriate-sized random binary string generated from a TRNG is expanded via a deterministic PRNG algorithm, which maintains the desired statistical properties. The validation of TRNG and PRNG outputs can be performed using the NIST's special publication SP800-22 [Bas+10] statistical test suite, while NIST's recommendations are published in the special publication SP800-90C [Bar+24]. A few examples of commonly used PRNG algorithms are the Mersenne Twister, xoshiro/xoroshiro, stream ciphers (e.g., ChaCha20), block ciphers in the counter mode of operations (e.g., AES-CTR), or XOF functions (e.g., SHAKE). In particular, many PQC schemes opted for AES-CTR and SHAKE, due to the presence of many highly-optimized software libraries for the former, and the high throughput offered by the latter.

For the sake of clarity in the description of the sampling algorithms, in this chapter a polynomial $a(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1}$ having coefficients $a_i \in \mathbb{F}_q,\ i \in 0, \ldots, n-1$ is considered as a n-dimensional vector composed by its coefficients $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}] \in \mathbb{F}_q^n$. The generation of a random j-bit binary string $\texttt{str} \in \{0,1\}^j$ from a PRNG is indicated with $\texttt{str} \leftarrow \text{PRNG}(j)$. The $\texttt{str}_{i+:j}$ notation represents the j-bit long substring of $\texttt{str}$ composed by the bit elements from index i to $i+j$. The described algorithms will be evaluated on the basis of distribution conformity, quantity of used randomness, and latency variability.

6.1 Pack and unpack vectors into and from bit strings

Before starting to explore the algorithms to sample random vectors, it is necessary to know how to deal with vectors transmitted as public keys or ciphertexts. The *pack* operation consists in encoding vectors into a bit string in a compact way to store and transmit more efficiently ciphertexts and keys, while also providing a performance boost in case the packed vectors are the input messages of PRNGs. The opposite operation is

Algorithm 17 Pack vectors with elements approximately a power of two

Require: $n \in \mathbb{N}$: the dimension of the vector to be sampled
$q \in \mathbb{N}$: the order of the finite field
$\mathbf{a} \in \mathbb{F}_q^n$
Ensure: `a_str`: the binary string representing the vector
1: `a_str` $\leftarrow \emptyset$
2: **for** $i \leftarrow 0$ **to** $n-1$ **do**
3: **for** $k \leftarrow 0$ **to** $\lceil \log_2(q) \rceil - 1$ **do**
4: `a_str` $\leftarrow$ `a_str` $\|((a_i \gg k)\&1)$
5: **return a**

Algorithm 18 Unpack vectors with elements approximately a power of two

Require: $n \in \mathbb{N}$: the dimension of the vector to be sampled
$q \in \mathbb{N}$: the order of the finite field
`a_str`: the binary string representing the vector
Ensure: $\mathbf{a} \in \mathbb{F}_q^n$
1: $\mathbf{a} \leftarrow \mathbf{0}$
2: **for** $i \leftarrow 0$ **to** $n-1$ **do**
3: $a_i \leftarrow \text{UINT}(\texttt{a\_str}_{i \cdot \lceil \log_2(q) \rceil + : \lceil \log_2(q) \rceil})$
4: **return a**

the *unpack*, and decodes vectors from bit strings before starting using them in arithmetic operations. A bit string `str` of length j can be interpreted as an unsigned integer $a \in \mathbb{N}$ such that $0 \leq a < 2^j$ via the function $\text{UINT} : \{0,1\}^j \mapsto \mathbb{N}_{2^j-1}$:

$$a = \sum_{i=0}^{j} 2^i \cdot \texttt{str}_i \tag{6.1}$$

Considering a vector of dimension n with elements in $\mathbb{F}_q$ and $q \approx 2^j,\ j \in \mathbb{N}$, the pack procedure, described in Algorithm 17, appends the bit representation of the element value a_i from the Least Significant Bit (LSB) to the Most Significant Bit (MSB), and repeats the operation for all elements in the vector. The unpack operation simply calls n times the UINT function on consecutive j-bit strings representing the packed vector, as represented in Algorithm 18.

When q is not a power of two, some binary strings are invalid representation of elements in $\mathbb{F}_q$. Minimizing the number of invalid encoded values improves the compression efficiency, leading to smaller ciphertexts and keys. Taking as an example $q = 3$, the 2-bits string $\texttt{str} \in \{\texttt{00}, \texttt{01}, \texttt{10}\}$ is employed to represent an element in $\{0,1,2\}$. However, it is possible to encode 5 consecutive vector elements $a_i \in \{0,1,2\}$ in a 8-bit string `a_str` instead of a 10 bits one, using the following equation:

$$\text{UINT}(\texttt{a\_str}) = \sum_{i=0}^{4} 3^i \cdot a_i \tag{6.2}$$

Consequently, compressing and then packing a vector of dimension n leads to a binary string of size $1.6 \cdot n$ bits instead of $2 \cdot n$ bits.

In this case, the unpack operation reads 8-bit binary strings and decodes it in 5 ternary elements through a decoding table, where the address of the table corresponds to the 8-bit string and the output is 10-bits long containing the expanded encoding of 5 ternary elements. Consequently, the table has size $2^8 * 10 = 2560$ bits (320 bytes). Of those 256 table entries, only 13 are invalid encodings, which translates in almost 95 % of space efficiency, compared to the 75% space efficiency obtained without applying the compression.

Applying the same look-up table approach for the compression procedure transforming 5 ternary elements in a 8 bit string, the table size would be much different: 10 bit addresses with 8-bits long output, requiring a memory of $2^{10} * 8 = 8192$ bits (1KiB) in size, with a representation efficiency of just $(243/1024) * 100 = 23.7\%$.

Note that most memories are accessed with a byte granularity, therefore it may be necessary to pad the resulting compressed and packed string, normally with 0 bits, until the length is a multiple of 8.

6.2 Sampling random vectors

Starting from the simplest case when $q = 2$, sampling elements from the uniform distribution from $\mathbb{F}_2^n$ just requires to truncate the PRNG output to n bits. This approach is applicable also to sample random binary strings. In case of a generic power of two number $q = 2^j,\ j \in \mathbb{N}$, special attention is needed when packing coefficients in blocks of size $B \in \mathbb{N}$ if $\lceil \log_2(q) \rceil \nmid B$, as it may require the introduction of some `pad` binary string to fill the unused bits of each block. This process is required to guarantee the alignment of elements in each block to perform efficient arithmetic computations, both in SW and HW implementations. In order to properly align the elements in blocks, a Barrel shift can be employed, both for SW and HW implementations, producing the desired vector element in the lowest $\lceil \log_2(q) \rceil$ bits of the result. Considering that q is normally a fixed value in the parameters set of a cryptographic scheme, some optimizations are available for HW implementations to improve the latency or the area size of the solution.

Considering now a generic $q \in \mathbb{N}$, the rejection sampling technique interprets j random bit strings as an integer unsigned number x, and assigns it to a_i if $0 \leq x < q$, otherwise discards them and tries again with the following j bits. The algorithm, represented in Algorithm 19, is straightforward, but does not run in constant time. The time complexity is $T(n) = \Omega(n)$, but does not have an upper bound. If q is such that the probability of discarding the random integer x is minimal, then the rejection sampling algorithm on average has low latency and uses a low amount of randomness. Considering the rejection check a Bernoulli trial with success probability p, a geometric distribution models the sampling process of the vector element giving the probability of the first success in k independent trials. Let $X \sim G(p)$ be a discrete random variable

Algorithm 19 Rejection sampling

Require: $n \in \mathbb{N}$: the dimension of the vector to be sampled
$q \in \mathbb{N}$: the order of the finite field
Ensure: $\mathbf{a} \in \mathbb{F}_q^n$
1: $\mathbf{a} \leftarrow \mathbf{0}$
2: **for** $i \leftarrow 0$ **to** $n-1$ **do**
3: **repeat**
4: $x \leftarrow \text{UINT}(\text{PRNG}(\lceil \log_2(q) \rceil))$
5: **until** $x < q$
6: $a_i \leftarrow x$
7: **return** $\mathbf{a}$

Algorithm 20 Modulo remainder sampling

Require: $n \in \mathbb{N}$: the dimension of the vector to be sampled
$q \in \mathbb{N}$: the order of the finite field
$j \in \mathbb{N} \mid j \gg \lceil \log_2(q) \rceil$: the size of bit strings from the PRNG
Ensure: $\mathbf{a} \in \mathbb{F}_q^n$
1: $\mathbf{a} \leftarrow \mathbf{0}$
2: **for** $i \leftarrow 0$ **to** $n-1$ **do**
3: $x \leftarrow \text{UINT}(\text{PRNG}(j)) \bmod q$
4: $a_i \leftarrow x$
5: **return** $\mathbf{a}$

from the geometric distribution with success probability p of the Bernoulli trial. The expected number of trials required for a successful sample is $E(X) = \frac{1}{p}$, leading to an average consumption of randomness equal to $\frac{\lceil \log_2(q) \rceil}{p}$. Taking as example the sampling algorithm of the NTRU HRSS scheme having $q = 3$. Interpreting any $\lceil \log_2(3) \rceil = 2$ bit string as a random integer $x \in \{0, 1, 2, 3\}$ and rejecting the $11 = 3$ encoded value, there is a $p = 3/4$ probability of accepting the encoded value. The average randomness requirement is therefore 2.67 input random bits per vector element.

Another algorithm working for generic $q \in \mathbb{N}$ is listed in Algorithm 20. This algorithm completes the task in constant time similarly interpreting j random bit strings as an integer unsigned number x, and but computes the sampled vector coefficient as $x \bmod q$, thus having vector coefficients not sampled from a perfect uniform distribution. In [Sen21] it is proved that if $j \gg \lceil \log_2(q) \rceil$, then the distribution bias can be considered negligible and does not imply a security loss for the PQC schemes. Nonetheless, the amount of consumed randomness is greater than the rejection sampling technique. Considering the same example for the NTRU HRSS scheme, starting from a 8-bit number and taking the remainder of the modulo 3 operation to compute the vector coefficient leads to an average of $8 > 2.67$ random bits consumed. In this case, the probabilities of the outcomes $\{0, 1, 2\}$ differ by at most $1/256$, and the authors of the schemed deemed it a worthy compromise.

It is possible to combine the two approaches producing values from a perfect uniform

Algorithm 21 Rejection sampling with modulo

Require: $n \in \mathbb{N}$: the dimension of the vector to be sampled
$q \in \mathbb{N}$: the order of the finite field
$j \in \mathbb{N} \mid j \gg \lceil \log_2(q) \rceil$: the size of bit strings from the PRNG
Ensure: $\mathbf{a} \in \mathbb{F}_q^n$
1: $\mathbf{a} \leftarrow \mathbf{0}$
2: **for** $i \leftarrow 0$ **to** $n-1$ **do**
3: **repeat**
4: $x \leftarrow \text{UINT}(\text{PRNG}(j))$
5: **until** $x < \lfloor 2^j / q \rfloor \, q$
6: $a_i \leftarrow x \bmod q$
7: **return** $\mathbf{a}$

distribution with a significant lower rejection rate, although with higher consumption of randomness compared to the rejection sampling algorithm. The resulting algorithm reported in Algorithm 21 shows that there is not an upper bound for the running time and randomness usage. Note that the rejection threshold $\lfloor 2^j / q \rfloor \, q$ is a known constant value.

6.3 Sampling random vectors with fixed Hamming weight

Lattice-based and code-based PQC schemes both have in common the need of an efficient procedure to sample vectors $\mathbf{a} \in \mathbb{F}_q^n$ of dimension n and coefficients in $\mathbb{F}_q$ with a pre-defined Hamming weight $\text{HW}(\mathbf{a}) = w$. In some cases, there are further constraints in addition to the weight, such in the case of NTRU-HPS, where $q = 3$ and a specific number of -1 and 1 coefficients needs to be present in the randomly sampled vector. The creation of said sampling procedure proved to be a challenging task, especially considering the requirement of finding an algorithm having low and fixed latency, avoiding significant distribution distortions and consuming a reasonable amount of randomness. In this section, we are considering vectors in $\mathbb{F}_2^n$ to simplify the description, but the presented techniques can be adapted for generic vectors $\mathbf{a} \in \mathbb{F}_q^n$.

Considering $q = 2$, the most straightforward algorithm consists in sampling random vector positions $j \in \{0, 1, \ldots, n-1\}$ where set the coefficient bit to 1. The generation of random indexes can use either the rejection sampling (Algorithm 19) or rejection with modulo (Algorithm 21) algorithm. In Algorithm 22, if a drawn position already contains a one, the sampled number is discarded and a new one is produced, thus the algorithm does not work in constant time and uses a variable amount of randomness. This solution is viable only if the number of ones is by far smaller than the number of zeros ($w \ll n$), as if w approaches n, the probability of generating colliding vector indexes increases due to the Birthday Paradox. Modeling as a random variable X the number of calls to the PRNG function to successfully sample the random vector, and estimating its distribution, it is possible to determine the Cumulative Distribution Function (CDF) and compute the number x_λ of PRNG calls such that the the sampling process completes

Algorithm 22 Fixed-weight rejection sampling

Require: $n \in \mathbb{N}$: the dimension of the vector to be sampled
$w \in \mathbb{N} \mid w \leq n$: the Hamming weight of the vector to be sampled
Ensure: $\mathbf{a} \in \mathbb{F}_2^n$ such that $\text{HW}(\mathbf{a}) = w$

$\mathcal{S} \leftarrow \emptyset$
for $j \leftarrow 0$ **to** $w - 1$ **do**
  repeat
    idx $\leftarrow$ UINT(PRNG($\lceil \log_2(n) \rceil$))
  until idx $< n \wedge$ idx $\notin \mathcal{S}$
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{\text{idx}\}$
▷ *Compose the final vector from the list of indexes of non-zero elements* ◁
$\mathbf{a} \leftarrow \mathbf{0}$
for all $j \in \mathcal{S}$ **do**
  $a_j \leftarrow 1$
return $\mathbf{a}$

Algorithm 23 Constant time fixed-weight rejection sampling

Require: $n \in \mathbb{N}$: the dimension of the vector to be sampled
$w \in \mathbb{N} \mid w \leq n$: the Hamming weight of the vector to be sampled
$x_\lambda \in \mathbb{N} \mid x_\lambda \geq w$: the number of index sampling operations to perform
Ensure: $\mathbf{a} \in \mathbb{F}_2^n$ such that $\text{HW}(\mathbf{a}) = w$

$\mathcal{S} \leftarrow \emptyset$
for $j \leftarrow 0$ **to** $x_\lambda - 1$ **do**
  idx $\leftarrow$ UINT(PRNG($\lceil \log_2(n) \rceil$))
  if idx $< n \wedge$ idx $\notin \mathcal{S} \wedge |\mathcal{S}| < w$ **then**
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{\text{idx}\}$
▷ *Compose the final vector from the list of indexes of non-zero elements* ◁
$\mathbf{a} \leftarrow \mathbf{0}$
for all $j \in \mathcal{S}$ **do**
  $a_j \leftarrow 1$
return $\mathbf{a}$

Algorithm 24 Fixed-weight sampling via scramble

Require: $n \in \mathbb{N}$: the length of the vector to be sampled
$w \in \mathbb{N} \mid w \leq n$: the Hamming weight of the vector to be sampled
Ensure: $\mathbf{a} \in \mathbb{F}_2^n$ such that $\text{HW}(\mathbf{a}) = w$

1: ▷ *Set the first w elements in the vector* ◁
2: $\mathbf{a} \leftarrow \mathbf{0}$
3: **for** $i \leftarrow 0$ **to** $w - 1$ **do**
4: $\quad a_i \leftarrow 1$
5: ▷ *Fisher-Yates scramble* ◁
6: **for** $i \leftarrow n - 1$ **to** 1 **do**
7: $\quad$ **repeat**
8: $\quad\quad j \leftarrow \text{UINT}(\text{PRNG}(\lceil \log_2(i) \rceil))$ ▷ *Reduce the rejection rate*
9: $\quad$ **until** $j \leq i$
10: $\quad t \leftarrow a_j$
11: $\quad a_j \leftarrow a_i$
12: $\quad a_i \leftarrow t$
13: **return** $\mathbf{a}$

successfully in less than x_λ call with a probability equal to $\Pr(X \leq x_\lambda) = 1 - 2^{-\lambda}$. Therefore, setting λ to match the one of the appropriate security margin of each parameter set Table 1.1, and executing exactly x_λ loop iterations calling the PRNG function, the running time and the randomness used are constant. The Algorithm 23 is the result of this approach.

Another approach consists in preparing an initial vector with the required weight at a pre-defined location (e.g., in the first w coordinates of the vector), and then scramble the vector, as presented in Algorithm 24. The resulting vector is sampled from an ideal uniform distribution, but the consumption of randomness if not fixed. Note that most of the scrambling algorithms are not constant time in systems equipped with data cache memories, but generally HW accelerators do not make use of them. The asymptotic time complexity of the Fisher-Yates scrambling algorithm [Knu98] is $T(n) = \Theta(n)$, but the multiplicative constant hidden in the asymptotic notation reduces its utility in practical use cases. This is caused by the high latency in accessing random memory locations considering that the read-after-write data dependency imposes to flush the modified blocks after each swap. As i approaches 1, the number of rejected random indexes increases dramatically. To reduce the average rejection rate to 25%, it is sufficient to use the least amount of random bits necessary to represent the integer i, as performed in line 6 of Algorithm 24.

A similar algorithm consists in prefixing all the elements in the initial vector with 30 random bit strings, for a total of $n \cdot 30$ bits, and then sort the elements via any sorting algorithm, such as the merge sort having time complexity $\Theta(n \log n)$. At the end of the computation, the highest 30 bits of each element is discarded, leaving the desired fixed-weight vector. The result produced in a fixed amount of time is from a uniform distribution, but uses a remarkable quantity of random bits in the process.

Relaxing the requirement of an ideal uniform distribution, in [Sen21] is presented

Algorithm 25 Fixed-weight sampling via sorting

Require: $n \in \mathbb{N}$: the length of the vector to be sampled
$w \in \mathbb{N} \mid w \leq n$: the Hamming weight of the vector to be sampled
$\gamma \in \mathbb{N} \mid \gamma \gg \lceil \log_2(n) \rceil$: the size of bit strings from the PRNG
Ensure: $\mathbf{a} \in \mathbb{F}_2^n$ such that $\mathrm{HW}(\mathbf{a}) = w$

▷ *Set the first w elements in the vector* ◁
$\mathbf{a} \leftarrow \mathbf{0}$
for $i \leftarrow 0$ **to** $w - 1$ **do**
    $a_i \leftarrow 1$
for $i \leftarrow 0$ **to** $n - 1$ **do**
    $a_i \leftarrow a_i \| \mathrm{PRNG}(\gamma)$ ▷ *Append γ random most significant bits*
$\textsc{Sort}(\mathbf{a})$ ▷ *Sort the elements of the vector*
for $i \leftarrow n - 1$ **to** 1 **do**
    $a_i \leftarrow a_{i,0}$ ▷ *Remove γ most significant bits*
return $\mathbf{a}$

Algorithm 26 Non-uniform Fisher-Yates fixed-weight sampling

Require: $n \in \mathbb{N}$: the dimension of the vector to be sampled
$w \in \mathbb{N} \mid w \leq n$: the Hamming weight of the vector to be sampled
$\gamma \in \mathbb{N} \mid \gamma \gg \lceil \log_2(n) \rceil$: the size of bit strings from the PRNG
Ensure: $\mathbf{a} \in \mathbb{F}_2^n$ such that $\mathrm{HW}(\mathbf{a}) = w$

for $j \leftarrow 0$ **to** $w - 1$ **do**
    support$[j] \leftarrow j + \textsc{UInt}(\mathrm{PRNG}(\gamma)) \bmod (n - j)$
▷ *Search for duplicates* ◁
for $j \leftarrow w - 1$ **to** 0 **do**
    found $\leftarrow 0$
    for $k \leftarrow j + 1$ **to** $w - 1$ **do**
        if support$[j] ==$ support$[k]$ **then**
            found $\leftarrow 1$
    if found **then**
        support$[j] \leftarrow j$
▷ *Compose the final vector from the list of indexes of non-zero elements* ◁
$\mathbf{a} \leftarrow \mathbf{0}$
for $j \leftarrow 0$ **to** $w - 1$ **do**
    $a_{\text{support}[j]} \leftarrow 1$
return $\mathbf{a}$

Table 6.1: *Comparison of sampler algorithm of fixed-weight $\mathbb{F}_2^n$ vectors. $x_\lambda > n$ is computed from the CDF of the distribution of the discrete random variable X representing the number of calls to the* PRNG *function, such that $\Pr(X \leq x_\lambda) = 1 - 2^{-\lambda}$. There is not an upper bound running time for the rejection sampling algorithm, and the constant time rejection sampling algorithm is not guaranteed to succeed for every input random string.*

Algorithm	Distribution bias	Average randomness	Constant time and randomness	Asymptotic time complexity	Always successful
Algorithm 22	✓	–	✗	$\Omega(n)$	✗
Algorithm 23	✓	$\lceil \log_2(n) \rceil \cdot x_\lambda$	✓	$\Theta(x_\lambda)$	✗
Algorithm 24	✓	$1.25 \cdot \lceil \log_2(n) \rceil \cdot n$	✗	$\Theta(n)$	✓
Algorithm 25	✓	$\gamma \cdot n$	✓	$\Theta(n \log n)$	✓
Algorithm 26	✗	$\gamma \cdot w$	✓	$\Theta(w^2)$	✓

an algorithm, here reported in Algorithm 26, using the same concept of the Fisher-Yates scrambling, but using a fixed amount of randomness and working in constant time. The procedure samples a set of w integers from a binary string of length $\gamma \gg \lceil \log_2(n) \rceil$, requiring a total of $\gamma \cdot w$ random bits. Then it computes the modulo $n - j$ remainder for each one of them, with j being a monotonically increasing counter starting from 0 up to the required weight $w - 1$ summed to the modulo remainder to create a support vector. Afterwards, the w elements in the support vector are checked for duplicates, and in case of a collision the element with highest index gets replaced with the index itself. Given how the remainders are generated, this step guarantees that after the first check there are no duplicates in the support vector. The resulting values are the indexes of the coefficients to be set to one in the fixed-weight vector. The time complexity of this algorithm is $T(n) = \Theta(w^2)$, and depends entirely only on the weight w of the polynomials. Another important difference of this algorithm compared to the scrambling-based one, is that in this case the entirety of PRNG randomness usage is consumed at the beginning of the algorithm. Therefore, a high-throughput source of randomness is necessary to not limit the performance of this algorithm.

In Table 6.1 is reported the comparison of the presented fixed-weight random $\mathbb{F}_2^n$ vectors, summarizing their property of absence of uniform distribution distortions, average quantity of used randomness, and time and randomness usage variability, and asymptotic time complexity.

6.4 Hardware designs

In this section are reported the synthesis results of the hardware designs implementing the (un)pack, (de)compression, and sampling of random vectors/polynomials with or without fixed-weight for the NTRU, HQC, and CROSS schemes. Whenever possible, the implementations of each work are referred to the pseudo-algorithms previously described to give a general idea of its working principle, although there could be some substantial differences with the actual implemented algorithm.

6.4.1 NTRU

The elements of NTRU-HPS and NTRU-HRSS cryptoschemes are polynomials in the ring $\mathbf{R}_q$ or $\mathcal{T} \subseteq \mathbf{S}_p$, corresponding to vectors in $\mathbb{Z}_q^n$ or $\mathbb{Z}_p^{n-1}$. The coefficients are the integers modulo $q = \{2048, 2048, 4096, 8192\}$ for the parameter sets `ntruhps2048509`, `ntruhps2048677`, `ntruhps4096821`, and `ntruhrss701`, respectively, or modulo $p = 3$. In both cases, the polynomial coefficients are represented with their zero-centered equivalence classes representatives encoded in two's complement.

When storing these polynomials in the ciphertext or public/private keys, the corresponding vectors must be packed in their bit string representation, and possibly compressed. For vectors in $\mathbb{Z}_q^n$ we can efficiently use the Algorithm 17 to pack the coefficients since those are exactly power of two, while vectors in $\mathbb{Z}_p^{n-1}$ additionally require the compression function Equation 6.2 to reduce the number of invalid bit sting encodings.

Considering the Equation 6.2, it is possible to have a trade-off between logic and registers requirements, storing the constants $3^i, i \in \{0, 1, 2, 3, 4\}$ in FFs. Each clock cycle an element a_i of $\mathbf{a}$ at index i is processed. The index i selects one of the stored constants via a multiplexer, which is multiplied by two via a bit-shift operation, which is a free operation in HW. A last multiplexer select between the zero value constant, the selected constant, or the selected constant multiplied by two depending on the element a_i value. The output of the last multiplexer is then added to an 8-bits accumulator, which after five iterations contains the compressed binary string.

The reverse unpack operation is described in Algorithm 18, and the uncompression of binary strings encoding ternary polynomials is performed using the compact look-up table.

Both NTRU variants need to sample random q-ary and ternary polynomials, while NTRU-HPS also requires to generate random ternary polynomials with $q/8 - 2$ non-zero coefficients. See Table 3.1 for more details.

Random q-ary polynomials are generated from the random bit strings coming from the PRNG by interpreting the $\lceil \log_2(q) \rceil$ bits consecutive sub-strings as signed integer numbers. Being the coefficients transferred in B-sized blocks, an appropriate `pad` binary string is introduced in each block to fill the unused bits of each memory block, aligning the coefficients in each block to perform efficient arithmetic computations.

The official specification mandates the use of Algorithm 20 to generate random polynomials in $\mathcal{T}$, computing each coefficient as the remainder of the modulo operation of a random 8-bit integer divided by 3. For such dimensions, the resulting bias of the uniform distribution is deemed negligible from the security standpoint. In this work, both the mandated modulo-reduction based algorithm and the rejection sampling strategy are evaluated, as the former is has straightforward implementation, while the latter is the most efficient in terms of randomness usage, and consequently on the pressure on the PRNG. The modulo 3 reductions are performed with a dedicated unit exploiting the fast reduction algorithm for Mersenne primes, which employs only shift operations and additions. Given the size of the prime number, the resulting design is extremely small and

Table 6.2: *Performance and area figures of $\mathbb{F}_2^n$ vector samplers with dimension and weight $(n, w) \in \{(17669, 75), (35851, 114), (57637, 149)\}$, defined by the parameters sets `hqc128`, `hqc192`, and `hqc256`, respectively. Area-Time product in eSlices $\cdot$ μs. The result denoted with * runs in constant time, but does only $w < n$ swap operations, and improperly produces random indexes, consequently not generating polynomials from the uniform distribution. The result denoted with ⋆ targets an Artix-7 FPGA with a lower speed grade -1.*

Param. set	Work	Algorithm	Resources					Freq.	Latency		AT
			LUT	FF	BRAM	DSP	eSlice	MHz	CC	μs	prod.
`hqc128`	This work	Algorithm 26	1055	1189	2.0	0	688	230	2983	12.97	8.92
	[Des+23]	Algorithm 26	201	229	1.0	4	801	201	3062	15.23	12.20
	[Des+23]	Algorithm 23	316	124	2.0	0	503	223	1479	6.63	3.34
	[HTX23]	Algorithm 24*	1560	766	2.0	0	814	170	976	5.74	4.67
	[Ae22]	Algorithm 22	9942	4354	0.0	0	2486	151	2573	17.04	42.36
`hqc192`	This work	Algorithm 26	1010	1178	2.0	0	677	237	6727	28.38	19.22
	[Des+23]	Algorithm 26	211	245	1.0	5	938	200	6817	34.08	31.97
	[Des+23]	Algorithm 23	295	125	2.0	0	498	236	2226	9.43	4.70
	[HTX23]	Algorithm 24*	1553	761	3.0	0	1025	185	1636	8.84	9.06
`hqc256`	This work	Algorithm 26	1027	1172	2.0	0	681	225	11382	50.59	34.45
	[Des+23]	Algorithm 26	216	248	1.0	5	939	204	11487	56.31	52.87
	[Des+23]	Algorithm 23	314	192	2.5	0	609	242	3248	13.42	8.17
	[HTX23]	Algorithm 24*	1569	779	3.0	0	1029	181	2268	12.53	12.89

efficient.

For what concerns the random polynomials in $\mathcal{T}$ having fixed weight, the reference algorithm is based on the sorting of random values similarly to Algorithm 25, where the first $q/16 - 1$ are arbitrary set to 1, the following $q/16 - 1$ are set to -1, and the remaining coefficients are set to 0. Given the parameters n and q, the Hamming weights of the vectors associated to such polynomials are not extremely small. Instead of using Algorithm 25, we used Algorithm 24 to apply a random permutation to a fixed weight array of ternary coefficients via the Fisher-Yates shuffle. While this technique is a source of timing-side channel insecurity when employed in cache-endowed architectures, we are able to guarantee constant-time memory access in this design, and are thus immune to such concerns. This approach provides significant area gains with respect to the one of [DMG21] following the official specification. This method exhibits several key advantages: it demands substantially less randomness per coefficient (approximately 9 bits versus 30), necessitates significantly less temporary storage by storing only ternary coefficients rather than 32-bit integers, and utilizes randomness uniformly throughout the execution, eliminating the need for large upfront randomness generation.

6.4.2 HQC

In HQC the elements are polynomials with coefficients from the binary field which can be represented as a vector $\mathbf{h} \in \mathbb{F}_q^n$ with $q = 2$. Therefore, their binary string representations composing the ciphertext or the public and private key do not require

compression or packing.

The random quasi-cyclic parity-check matrix is represented by the polynomial $h \in \mathbf{R}$, thus the random vector $\mathbf{h}$ is generated straightforwardly by truncating the PRNG output to n bits. The hardware module performing such operation producing the vector with elements packed in blocks of $B = 128$ bits takes just 89 LUTs and 333 FFs, for an eSlices area indicator of 235. The maximum frequency of 357 MHz allowed to complete the operation in 314, 598, and 938 clock cycles taking 0.20, 0.40, and 0.62 μs for the three parameters sets `hqc128`, `hqc192`, and `hqc256`, respectively.

Regarding the sampling algorithms for sparse polynomials, the HQC scheme needs to generate the elements $x, y \in \mathbf{R}_w$, $e \in \mathbf{R}_{w_e}$, and $r_a, r_b \in \mathbf{R}_{w_r}$, where $w, w_e, w_r \approx \sqrt{n}$. Various approaches are reported in Table 6.2, using the highest vector weight w_r. Starting from the fourth specification document revision, the authors prescribe the use of the Algorithm 26 with $\gamma = 32$ to protect against timing side-channel attacks after that [Guo+22] demonstrated a successful key recovery attack due to timing variations of Algorithm 22 dependent on the secret key in the deterministic re-encryption executed in the HQC.KEM-DECAPSULATE primitive. Any change to the specified sampling algorithm will lead to different results from the KAT. However, if the actor executing the key generation and decapsulation algorithm is consistent in the algorithm chose to generate the polynomial y, the compatibility of the resulting design with other specification-compliant implementations is guaranteed. Given the $\gamma = 32$ parameter value, the amount of randomness used by this algorithm amounts to $32 \cdot w_r$, which for the parameters set with the highest security margin results in 4768 random bits consumed.

From an hardware design perspective, in the Algorithm 26, the computation of the remainder of a modulo operation is the most computationally expensive operation. The authors in [Des+23] opted use a pipelined Barrett reduction [Bar86] circuit where the large 32-bit integer multiplication is split in three smaller multiplications with halved bit-size via the sub-quadratic Karatsuba multiplication algorithm [Kar63] (more on them in chapter 7), and computed in the specialized DSP units. In this thesis is explored a novel approach starting from the consideration that the divisor, even though has not a fixed value, has always a 16-bit size. Therefore, a *shift-and-subtract* algorithm can be employed, completing the operation with exactly 16 shifts and subtractions without using DSP units. The design has a pipeline with k stages in which $\lceil 16/k \rceil$ shift and subtractions are performed, completing each operation in k clock cycles, but being capable of executing k modulo operations in parallel. Evaluating the maximum operating frequency of this design while varying the number of pipeline stages determined that, to prevent the divider functional unit from being the limiting factor of the other modules, each stage must compute a single shift and subtraction. The synthesis results show a reduction of area from $1.16\times$ to $1.37\times$, and a substantially improved maximum frequency reducing the latency from $1.11\times$ to $1.17\times$. Consequently, the efficiency improvement ranges from $1.36\times$ to $1.53\times$.

The same authors in [Des+23] tried another solution akin to the constant time rejection sampling in Algorithm 23 with $x_\lambda = 2 \cdot w_r$, reducing the rejection rate generating

the random indexes with Algorithm 21, and showing better figures in terms of latency, area usage, and efficiency than the designs implementing the official Algorithm 26. The amount of randomness used by this algorithm results higher than the one used in Algorithm 26 and amounts to $24 \cdot 2 \cdot w_r$, which for the parameters set with the highest security margin results in 7152 random bits consumed.

In [Ae22] the authors provide a High Level Synthesis (HLS) description of the rejection sampling-based algorithm of the initial specification just for the parameters set offering the lowest security margin. The synthesis results highlight the inefficiencies resulting from the HLS description compared to a RTL one, as the simpler rejection sampling algorithm takes more resources compared to the more refined constant time algorithms.

Lastly, in [HTX23] is presented another design implementing a scrambling technique resembling Algorithm 24, although with few key differences generating a distribution bias. Firstly, the Fisher-Yates algorithm is used to scramble a vector with w bits in the first positions of the vector, but only $w < n$ random indices are swapped. Moreover, the random indices are generated as the remainder of a modulo operation from an unspecified sized random dividend, leading to random values not sampled from the uniform distribution. The resulting component shows interesting latency and efficiency figures at the cost of a larger area occupied and a security yet-to-be-proven.

6.4.3 CROSS

Considering the CROSS scheme, the elements transmitted in the public key pk and the signature sig are bit strings ($\texttt{path}$, $\texttt{proof}$, $\texttt{salt}$, $\texttt{cmt}_1[i]$) and vectors with coefficients in $\mathbb{F}_p$ or $\mathbb{F}_z$ ($\mathbf{s}$, $\overline{\mathbf{v}}$, and $\mathbf{y}$), with p and z two prime numbers from the set $\mathcal{S} = \{7, 127, 509\}$. Being all the values in $\mathcal{S}$ approximately a power of two 8, 128, and 512, respectively, the usual Algorithm 17 and Algorithm 18 can be used to efficiently pack the vectors in bit strings, without requiring extra logic to handle some compression and decompression tasks. Note that the elements are packed before the absorption by the HASH function.

Within the CROSS.KEYGENERATION, CROSS.SIGN and CROSS.VERIFY primitives the public key seed $\texttt{seed}_{\mathsf{pk}}$ is expanded by an instance of the CSPRNG to obtain an arbitrary long bit string used to generate the matrices $\overline{\mathbf{W}} \in \mathbb{F}_z^{m \times n-m}$ and $\mathbf{V} \in \mathbb{F}_p^{(n-k) \times k}$, the former only in case a R-SDP(G) based parameter set is selected.

Similarly, in CROSS.KEYGENERATION and CROSS.SIGN the private key $\mathsf{sk} = \texttt{seed}_{\mathsf{sk}}$ is expanded via a CSPRNG in a longer bit string used to generate the secret error vector $\overline{\mathbf{e}}_G \in \mathbb{F}_z^m$ and $\overline{\mathbf{e}} \in \mathbb{F}_z^n$, respectively for R-SDP(G) and R-SDP parameters.

Moreover, in each one of the t CROSS.ID protocol executions in CROSS.SIGN is generated the random vector $\mathbf{u}'[i] \in \mathbb{F}_p^n$, and the transformed error vector $\overline{\mathbf{e}}'_G[i] \in \mathbb{F}_z^m$ or $\overline{\mathbf{e}}'[i] \in \mathbb{F}_z^n$ for R-SDP(G) and R-SDP parameters, respectively, starting from the CSPRNG output generated by the absorption of the round seed and salt values $\texttt{seed}[i] \| \texttt{salt}$.

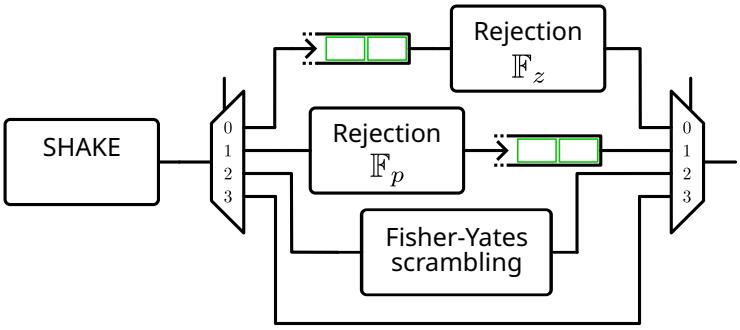


Figure 6.1: *Sampler unit of CROSS elements implementing the* CSPRNG *and* HASH *functions. Credits to Patrick Karl from the Technical University of Munich.*

Finally, the two challenges $\texttt{chall}_1 \in (\mathbb{F}_p^*)t$ and $\texttt{chall}_2 \in \mathcal{B}_t, w$ (the set of all binary strings of length t having Hamming weight w) are generated from the expansion of the hash digests $\texttt{dig}_{\texttt{chall}_1}$ and $\texttt{dig}_{\texttt{chall}_1}$, respectively.

With the exclusion of $\texttt{chall}_2$, each coefficient composing all the previously described vectors are generated via a rejection sampling as detailed in Algorithm 19, since the probability of rejection of each coefficient is low, namely $1/8$, $1/128$, and $3/256$ when sampling coefficients in $\mathbb{F}_7$, $\mathbb{F}_{127}$, and $\mathbb{F}_{509}$, respectively. The process consists in interpreting each $\lceil \log_2 p \rceil$ bit long string chunk $\texttt{a}$ as an integer element $a = \text{UINT}(\texttt{a}) \in \mathbb{Z}$, and discard it if $a \geq p$. Recalling that the rejection sampling does not exhibit the execution in a constant amount of time, the authors provided an upper bound x_λ similar to Algorithm 23 to fail the sampling process after x_λ random samplings with an extremely low probability $< 2^\lambda$, with $\lambda \in \{128, 192, 256\}$ being the security margin. The coefficients of the vectors are sampled sequentially from the lowest to the highest indexes, whereas matrices are linearized by rows, hence the sampled coefficients are filling the matrix from the top-most row to the bottom-most one, left to right.

Regarding the generation of the second challenge $\texttt{chall}_2$, the Algorithm 24 is employed to perform the scramble using a Fisher-Yates algorithm of a vector having the first w bits set to $\texttt{1}$, and the following $t - w$ bits set to $\texttt{0}$.

The designed sampling unit, depicted in Figure 6.1, generates all the elements of CROSS from the expansion of a small seed absorbed by an instance of the SHAKE XOF, either `SHAKE256` or `SHAKE128` depending on the parameter set, that generates an arbitrary long random bit string used by two rejection sampling units specialized for the generation of $\mathbb{F}_p$ and $\mathbb{F}_z$ coefficients, and a Fisher-Yates algorithm scrambling a binary vector having fixed Hamming weight w.

Each CROSS-ID parallel execution asks to generate two elements, $\overline{\mathbf{e}}'[i] \in \mathbb{F}_z^n$ (or $\overline{\mathbf{e}}'_G[i] \in \mathbb{F}_z^m$ in case of R-SDP(G) parameters) and $\mathbf{u}'[i] \in \mathbb{F}_p^n$, taking part to some arithmetic computation. However, each rejection sampler is able to produce at most one vector coefficient per clock cycle, and the official specification mandates the sequential sampling of $\overline{\mathbf{e}}'[i]$ (or $\overline{\mathbf{e}}'_G[i]$) before $\mathbf{u}'[i]$ from the expansion of the same seed. To improve the latency of such operation, the generation of the two elements can be performed in

Table 6.3: *Synthesis results for the sampler of CROSS vectors when targeting an AMD Artix-7 FPGA and using 64-bits word size. The sampler unit is agnostic to the fast, balanced, and small optimization corners. Credits to Patrick Karl from the Technical University of Munich.*

Parameter set	Resources				Freq.
	LUT	FF	BRAM	eSlice	MHz
CROSS-RSDP-1	10757	3749	2.5	3220	159
CROSS-RSDP-3	9478	3487	3.5	3112	173
CROSS-RSDP-5	9903	3579	4.5	3430	179
CROSS-RSDPG-1	9643	3824	1.5	2729	167
CROSS-RSDPG-3	9357	3427	3.5	3082	176
CROSS-RSDPG-5	8077	3453	3.5	2762	179

parallel by employing two First-In First-Out (FIFO) buffers. The first one accumulates all the XOF material necessary to sample the entire vector with coefficients in $\mathbb{F}_z$ (selection 0 of the multiplexer/demultiplexer in Figure 6.1), allowing the second rejection sampler to start as soon as enough material is present in the buffer by switching the demultiplexer signal to the selection 1. This is possible due to the constant-time sampling algorithm exactly specifying the number of coefficient samplings x_λ performed in each operation. The second FIFO buffer is used to guarantee the sequentiality of the sampled elements, by releasing to the output the vector $\mathbf{u}'[i]$ switching the multiplexer selection to 1 only after the full transmission of the vector with coefficients in $\mathbb{F}_z$.

In Table 6.3 is reported the synthesis of the sampler unit for the AMD Artix-7 FPGA specialized for every parameter set. Note that the vector and matrix shapes do not vary depending on the fast, balanced, or small optimization corners. The FIFO buffers are implemented by Vivado in a few BRAM units. The resource occupation figures include the contribution of the SHAKE module, which takes ≈ 6443 LUT and 2735 FF. This unit is optimized for the absorption of two seed sizes of length $3\lambda + 16$ and $4\lambda + 16$ by instantly padding the remaining part of the input buffer and start the f-permutation function in the following clock cycle. Note that the SHAKE module in the sampler unit is also used as the instance implementing the HASH function, therefore the output of the SHAKE unit is also directly routed to the output of the sampler to produce the hash digests via the multiplexer/demultiplexer selection 3 in Figure 6.1.

CHAPTER 7

Arithmetic

This chapter introduces to the arithmetic operations performed in lattice-based and code-based cryptographic schemes. A brief mathematical background is provided in section 2.1. Each section starts with a description of the operations for modular arithmetic and then, building on them, presents the algorithms for vector spaces and polynomial rings. Note that the described operations are always valid only if the operands are from a finite field, which guarantees the existence of the additive and multiplicative inverses for each element in the field.

7.1 Addition

A natural number $a \in \mathbb{N}$ is representable by a n-bit binary string $\mathtt{a} = \mathtt{a}_0 \| \mathtt{a}_1 \| \dots \| \mathtt{a}_{n-1}$ such that $a = \sum_{i=0}^{n} \mathtt{a}_i \cdot 2^i$. The n-bit binary string can encode all the unsigned integer numbers in the range $[0, 2^n - 1]$. Considering $a, b \in \mathbb{N}$ in their binary representation, the addition operation $(+)$ can be performed digit by digit starting from the LSB $\mathtt{a}_0$ to the MSB $\mathtt{a}_{n-1}$. A *half adder* operation produces a sum bit $\mathtt{s}_0 = \mathtt{a}_0 \oplus \mathtt{b}_0$ and a carry bit $\mathtt{c}_0 = \mathtt{a}_0 \wedge \mathtt{b}_0$, computed as the Boolean `xor` and the Boolean `and` operations of two bits of the operands, respectively. The carry bit represents the overflow in the addition computation for the previous $(i-1)$-th digit, which is then given in input to the *full adder* operation of the current i-th digit. The sum bit is similarly computed as

Part of the material presented in this chapter was originally described in [Ant+23b; Ant+23a; Ant+24a; Ant+24b; Ant+24c; ABP25].

$\mathtt{s}_i = \mathtt{a}_i \oplus \mathtt{b}_i \oplus \mathtt{c}_{i-1}$, while the produced carry is $\mathtt{c}_i = (\mathtt{a}_i \cdot \mathtt{b}_i) \oplus (\mathtt{c}_{i-1} \wedge (\mathtt{a}_i \oplus \mathtt{b}_i))$. Consequently, the addition of two n-bit numbers produces a $(n+1)$-bit result via a chain of full adders, except for the LSB being computed via a half adder. This structure is known as *ripple carry adder*, and has a critical path determined by the chain of n carry computations. There are several known algorithms trying to provide an area-time trade-off (carry look-ahead adder), or deferring the computation of the carry (carry save adder).

The two's complement representation encodes an integer number $a \in \mathbb{Z}$ in a n-bit binary string such that $a + (-a) = 0$ using the previous algorithm. A n-bit binary string can encode all the unsigned integer numbers in the range $[-2^{n-1}, 2^{n-1} - 1]$. To compute the additive inverse element $-a$, all binary digits of a are negated via a Boolean not operation, here denoted with an abuse of notation as $\neg a$, and adding 1 to the result. The subtraction operation $a - b$ therefore can be computed as $a + (-b) = a + \neg b + 1$. This means that the subtraction can be implemented via a chain of full adders similarly to the addition, but all the bits of b are negated, and the input carry c_{-1} of the LSB digit is artificially set to 1. By using some multiplexers, the same architecture employed for the subtraction can be reused to compute also the addition of two numbers.

Nowadays all the EDA tools handle the generation of such structures transparently when the addition or subtraction operations between two integer numbers are detected. FPGA vendors even provide hard IPs to minimize the latency of additions/subtractions in the designs.

7.1.1 Modular arithmetic

A prime field $\mathbb{F}_p \equiv \mathbb{Z}/p\mathbb{Z}$ is a finite field having prime order p where the set of elements $\mathcal{A}$ is the integers modulo p as the canonical representatives of the residue classes. The defined operations follow the modular arithmetic for a generic modulo q, thus given $a, b \in \mathbb{Z}_q$, their sum or difference is $a \pm b \bmod q$. Operatively, to compute the residue class representative of the sum result $\geq q$, it is necessary to subtract q to it. Conversely, to compute the residue class representative of the difference result < 0, it is necessary to add q to it. To make the operation constant time, the addition or subtraction by q is always carried out, and the correct result is picked via a selection mask in SW, or via a multiplexer in HW.

Note that when $q = 2^k$ for $k \in \mathbb{N} \setminus 0$ – clearly not the case for prime fields – the modulo remainder is equivalent to the integer encoded in the first k bits, therefore the carry bit of the addition or subtraction operation is simply ignored. The edge case is represented by $q = 2$ where the Boolean `xor` operator performs both the addition and the subtraction of elements in $\mathbb{F}_2$.

7.1.2 Vector space and polynomials

Another cryptographically-relevant class of finite fields are extension fields $\mathbb{F}_{p^m}$, which are constructed from a base field $\mathbb{F}_p$ and contain p^m elements. These fields can be

Algorithm 27 Modular addition/subtraction between vectors

Require: $n \in \mathbb{N}$: the dimension of the vectors
$q \in \mathbb{N}$: the modulo
$\mathbf{a}, \mathbf{b} \in \mathbb{Z}_q^n$: the two vector operands
Ensure: $\mathbf{c} \in \mathbb{Z}_q^n \mid \mathbf{c} = \mathbf{a} \pm \mathbf{b}$
1: **for** $i \leftarrow 0$ **to** $n - 1$ **do**
2: $\quad c_i \leftarrow (a_i \pm b_i) \bmod q$
3: **return c**

Algorithm 28 Modular addition/subtraction between vectors when one operand is sparse

Require: $n \in \mathbb{N}$: the dimension of the vectors
$q \in \mathbb{N}$: the modulo
$w \in \mathbb{N}$: the weight of polynomial $\mathbf{b} \in \mathbb{Z}_q^n$
$\mathbf{a} \in \mathbb{Z}_q^n$: the first operand in regular format
$\mathbf{b}_{\text{sparse}} \in \mathcal{S}^w$: the operand $\mathbf{b} \in \mathbb{Z}_q^n$ with w non-zero elements in sparse format
Ensure: $\mathbf{c} \in \mathbb{Z}_q^n \mid \mathbf{c} = \mathbf{a} \pm \mathbf{b}$
1: $\mathbf{c} \leftarrow \mathbf{a}$
2: **for** $(i, v) \in \mathbf{b}_{\text{sparse}}$ **do**
3: $\quad c_i \leftarrow (c_i \pm v) \bmod q$
4: **return c**

realized as quotient rings of the form $\mathbb{F}_p[x]/\langle f(x)\rangle$, where $f(x)$ is an monic irreducible polynomial of degree m over $\mathbb{F}_p$. Elements of the extension field $\mathbb{F}_{p^m}$ have multiple but equivalent forms, where the most commonly used ones are polynomials of degree less than m, or power of the root element α of $f(x)$. For more details refer to subsection 2.1.3. Using the usual representation of a polynomial $a(x) = a_0 \cdot x^0 + a_1 \cdot x^1 \cdot \ldots \cdot a_{m-1}x^{m-1}$ as a vector $\mathbf{a} \in \mathbb{F}_p^m$ of dimension m having as elements the polynomial coefficients $a_i \in \mathbb{F}_p$, the addition and subtraction operations are carried out element-wise following the modular arithmetic defined in $\mathbb{Z}_q$, as represented in Algorithm 27. The computational complexity of this algorithm is $\Theta(n)$ when considering the addition/subtraction operation in $\mathbb{Z}$ as an elementary operation. Binary extension fields $\mathbb{F}_{2^m}$ are remarkably efficient to implement in SW and HW with respect to other choices of base field order p, as the elements can be represented with binary vectors. In this case, both addition and subtraction are carried out via a bitwise Boolean `xor` operation, without considering carries or modulo remainders to compute.

A polynomial or a vector is considered *sparse* when it has significantly fewer w non-zero coefficients than its total length ($w \ll n$). Such sparse entities can be efficiently represented using a sparse format, where only the indexes i and corresponding values v of the non-zero elements are stored in a list of tuples (i, v). Let $\mathcal{S} = \{(i, v) \mid i \in \mathbb{N}, v \in \mathbb{Z}, i < n, -q/2 \leq v < q/2\}$ be the set containing the (i, v) valid pairs for a vector in $\mathbb{Z}_q^n$. The sparse representation of a vector having Hamming weight w is given by a vector of dimension w with elements in $\mathcal{S}$. For binary polynomials or vectors, since all non-zero values are inherently 1, only the indices i of these non-zero elements need to be stored. When one of the addition or subtraction operand is in the sparse form, more efficient

Algorithm 29 Barrett reduction

Require: $q \in \mathbb{N}$: the modulo value
$a \in \mathbb{Z} \mid 0 \leq a < q^2$: the input value
Ensure: $b \in \mathbb{Z} \mid 0 \leq b < q$
1: ▷ *Approximation factor* $k = \lceil \log_2(q) \rceil$ ◁
2: ▷ *Pre-computed constant* $r = \left\lfloor \frac{4^k}{q} \right\rfloor$ ◁
3: $c \leftarrow \left\lfloor \frac{a \cdot r}{4^k} \right\rfloor$ ▷ *Equivalent to* $(a \cdot r) \gg 2k$
4: $t \leftarrow a - c \cdot q$
5: **if** $t \geq q$ **then**
6: $\quad b \leftarrow t - q$
7: **else**
8: $\quad b \leftarrow t$
9: **return** b

algorithms can be devised, improving the time complexity of the solution. For example Algorithm 28 has a time complexity $\Theta(w)$ when it works in-place, thus updating the first operand without copying it to the result vector in step 1.

7.2 Multiplication

Let $a, b \in \mathbb{Z}$ two integers such that both a and b can be encoded in n-bit strings in two's complement notation. The schoolbook algorithm computes the multiplication operation $(\cdot)$ between a and b summing the integers corresponding to the extension of the bit string representation of a with i zero bits at the LSB side if $\mathtt{b}_i$ is 1. The result c is the $2n$-bit long given by Equation 7.1.

$$c = \sum_{i=0}^{n-1} (\mathtt{b}_i \wedge \mathtt{a}) \ll i \tag{7.1}$$

Most EDA tools automatically infer the optimal architecture performing this integer multiplication.

7.2.1 Modular arithmetic

In modular arithmetic, the multiplication between two elements $a, b \in \mathbb{Z}_q$ is given by $(a \cdot b) \bmod q$, but this time the computation of the equivalence class representative is more challenging with respect to the case of addition or subtraction.

The Barrett reduction [Bar86] (Algorithm 29) is an algorithm efficiently performing the modulo operation of a for a fixed divisor value q. Instead of performing a full division, the algorithm uses a multiplication with a pre-computed constant $r = \left\lfloor \frac{4^k}{q} \right\rfloor$ and bit shift to get an approximated quotient c. Multiplying that value by q and subtracting it from the input value a results in a value in the range $[0, 2q)$, which is checked and conditionally reduced in the required range $[0, q)$. Overall, the arithmetic operations used are a $2k \times k$-bit multiplication, a $2k \times 2k$-bit subtraction, a $k \times k$-bit smaller

Algorithm 30 Mersenne primes reduction

Require: $q \in \mathbb{N} \mid q = 2^k - 1 \wedge j \nmid q,\ k, j \in \mathbb{N} \wedge 2 \leq j < q$: the Mersenne prime modulo value
$a \in \mathbb{Z} \mid 0 \leq a < q^2$: the input value
Ensure: $b \in \mathbb{Z} \mid 0 \leq b < q$
1: $t \leftarrow a$
2: **for** $i \leftarrow 0$ **to** $\lceil \lceil \log_2(a) \rceil / \lceil \log_2(q) \rceil \rceil - 1$ **do**
3: $\quad \mathtt{c} \leftarrow \mathtt{t}_0 \| \mathtt{t}_1 \| \dots \| \mathtt{t}_{\lceil \log_2(q) \rceil - 1}$ ▷ *Lower* $\lceil \log_2(q) \rceil$ *bits of the binary string representation* $\mathtt{t}$ *of* t
4: $\quad \mathtt{d} \leftarrow \mathtt{t}_{\lceil \log_2(q) \rceil} \| \mathtt{t}_{\lceil \log_2(q) \rceil + 1} \| \dots \| \mathtt{t}_{\lceil \log_2(a) \rceil - 1}$ ▷ *Remaining upper bits of* $\mathtt{t}$
5: $\quad t = c + d$ ▷ *Considering the integers* c, d *from their binary string* $\mathtt{c}, \mathtt{d}$
6: **if** $t \geq q$ **then**
7: $\quad b \leftarrow t - q$
8: **else**
9: $\quad b \leftarrow t$
10: **return** b

multiplication, a $(k+1) \times k$-bit smaller subtraction, and a shift by a constant amount of bits.

Another approach that similarly trades the division by q with a cheaper division by a power-of-two consists in performing the multiplications in the Montgomery domain. The conversion of inputs to the Montgomery domain, and the reverse process for the result is expensive, thus it is considered only when the number of modular multiplication performed in a row is high, such in case of RSA or DH key exchange. A more formalized view of the integer approximation of these two reduction schemes is analyzed in [Bec+22b], where the correspondence between the Montgomery multiplication and the Barrett multiplication is detailed.

When $q = 2^k - 1$ is a prime number for some $k \in \mathbb{N}$, the modulus q is referred as *Mersenne prime*. These prime numbers cover an important role in cryptography since the extremely efficient Algorithm 30 computes the modulo operation with a few additions/subtractions.

Finally, recalling that if q is a power-of-two the modulo operation corresponds to the selection of the first $\lceil \log_2(q) \rceil$ bits, in the edge case $q = 2$ the multiplication in $\mathbb{F}_2$ simplifies to the Boolean `and` between the two operands.

A naive modular exponentiation $b^a \bmod q$ for some $a, b \in \mathbb{Z}_q$ may iterate the modular multiplication of an auxiliary variable c, initialized with the multiplicative identity element 1, by the value b for a times. This approach however has a considerable cost due to the large number of multiplications required, which depends on the value a. Moreover, there may be a problem when a is a secret element of a cryptographic algorithm, leading to a timing side-channel leakage. The *square-and-multiply* algorithm is a well-known technique to sensibly reduce the number of multiplications from $O(a)$ to $O(\log_2 a)$. Iterating on each digit of $\mathtt{a}$, the binary string representation of a, if $\mathtt{a}_i = 1$ then the auxiliary variable c is multiplied by b^{i+1}, which is simply computed by the modular squaring of the initial value b each round iteration. Particular care is still needed when a is a secret element of a cipher, as the literature offers many simple examples of

Algorithm 31 Schoolbook polynomial multiplication algorithm

Require: $n \in \mathbb{N}$: the maximum degree of polynomials
$q \in \mathbb{N}$: the coefficient modulo
$\mathbf{a}, \mathbf{b} \in \mathbb{Z}_q^n$: the two operand vectors containing the coefficients of polynomials $a(x)$ and $b(x)$
Ensure: $\mathbf{c} \in \mathbb{Z}_q^{2n-1} \mid c(x) = a(x) \cdot b(x)$
1: **for** $j \leftarrow 0$ **to** $n-1$ **do**
2: **for** $i \leftarrow 0$ **to** $n-1$ **do** ▷ *This can be parallelized in n multiply-and-accumulate units*
3: $c_{j+i} \leftarrow (c_{j+i} + a_j \cdot b_i) \bmod q$ ▷ *Multiply-and-accumulate operation*
4: **return** $\mathbf{c}$

side-channel attacks targeting this operation.

7.2.2 Vector space and polynomials

Let $\mathbf{A}$ be a matrix from the vector space $\mathbb{F}_q^{n \times m}$, and $\lambda \in \mathbb{F}_q$ a scalar. The scalar multiplication $\lambda\mathbf{A}$ produces a matrix $\mathbf{C} \in \mathbb{F}_q^{n \times m}$ where each coefficient is computed as

$$C_{i,j} \leftarrow \lambda \cdot A_{i,j} \qquad \forall i \in \{0, 1, \ldots, n-1\},\ \forall j \in \{0, 1, \ldots, m-1\} \tag{7.2}$$

Considering now two matrices $\mathbf{A}, \mathbf{B}$ from the vector space $\mathbb{F}_q^{n \times m}$, the Hadamard multiplication $\mathbf{C} = \mathbf{A} \odot \mathbf{B}$ consists in the modular multiplication of the elements at the same indexes:

$$C_{i,j} = A_{i,j} \cdot B_{i,j} \bmod q \qquad \forall i \in \{0, 1, \ldots, n-1\},\ \forall j \in \{0, 1, \ldots, m-1\} \tag{7.3}$$

Having instead another matrix $\mathbf{B}$ from the vector space $\mathbb{F}_q^{m \times r}$, the matrix multiplication $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ is defined as:

$$C_{i,j} = \sum_{k=1}^{m} A_{i,k} \cdot B_{k,j} \bmod q, \qquad \forall i \in \{0, 1, \ldots, n-1\},\ \forall j \in \{0, 1, \ldots, p-1\} \tag{7.4}$$

When dealing with polynomials, the schoolbook operand scanning algorithm performing the multiplication between two polynomials, described in Algorithm 31, is not dissimilar to the one for integers described in Equation 7.1, adding together all the results of multiplying the first polynomial by each one of the monomials composing the second polynomial. The resulting polynomial has a maximum degree of $2n-2$, and the running time of the operation has a quadratic complexity $T(n) = O(n^2)$ in the cost of modular multiplications.

The sub-quadratic methods, pioneered by Karatsuba [Kar63], provide algorithms to compute the polynomial multiplication in $\mathcal{O}(n^{\log_i(2i-1)})$ coefficient-wise multiplications, where $i \geq 2$ and $i \mid n$. In particular, Karatsuba proposed the algorithmic variant for $i = 2$, here reported in Algorithm 32, while Toom and Cook [Bod07] generalized the result for $i > 2$. Note that the level of recursions can be modified by adjusting the base case condition (line 2 in Algorithm 32) and performing such multiplication with other parametrization of i, or even other algorithms. This is particularly useful

Algorithm 32 Karatsuba polynomial multiplication algorithm

Require: $n \in \mathbb{N}$: the maximum degree of polynomials
$q \in \mathbb{N}$: the coefficient modulo
$\mathbf{a}, \mathbf{b} \in \mathbb{Z}_q^n$: the two operand vectors containing the coefficients of polynomials $a(x)$ and $b(x)$
Ensure: $\mathbf{c} \in \mathbb{Z}_q^{2n-1} \mid c(x) = a(x) \cdot b(x)$

function KARATSUBAMULTIPLY($\mathbf{a}$, $\mathbf{b}$, n)
  if $n = 1$ **then** ▷ *Base case*
    return $(a_0 \cdot b_0) \bmod q$
  $m \leftarrow \lfloor n/2 \rfloor$
  $\mathbf{a_L} \leftarrow [a_0, \ldots, a_{m-1}]$, $\mathbf{a_H} \leftarrow [a_m, \ldots, a_{n-1}]$ ▷ *Split* $\mathbf{a}$ *in low and high part*
  $\mathbf{b_L} \leftarrow [b_0, \ldots, b_{m-1}]$, $\mathbf{b_H} \leftarrow [b_m, \ldots, b_{n-1}]$ ▷ *Split* $\mathbf{b}$ *in low and high part*
  $\mathbf{c_L} \leftarrow$ KARATSUBAMULTIPLY($\mathbf{a_L}, \mathbf{b_L}, m$) ▷ *Recuirsive call with low parts*
  $\mathbf{c_H} \leftarrow$ KARATSUBAMULTIPLY($\mathbf{a_H}, \mathbf{b_H}, n-m$) ▷ *Recuirsive call with high parts*
  $\mathbf{p} \leftarrow$ KARATSUBAMULTIPLY($\mathbf{a_L} + \mathbf{a_H}, \mathbf{b_L} + \mathbf{b_H}, \max(m, n-m)$) ▷ *Rec. call with mixed parts*
  $\mathbf{c_M} \leftarrow \mathbf{p} - \mathbf{c_L} - \mathbf{c_H}$ ▷ *Avoids a fourth multiplication*
  ▷ *Compose the result by shifting vectors to the left, padding with zero coefficients* ◁
  $\mathbf{c} \leftarrow \mathbf{c_L} +$ SHIFT($\mathbf{c_M}, m$) + SHIFT($\mathbf{c_H}, 2m$)
  return $\mathbf{c}$
return KARATSUBAMULTIPLY($\mathbf{a}, \mathbf{b}, n$)

when the size of polynomials do not exactly satisfy the condition $i \mid n$. These methods are not universally used because, although they reduce the number of individual coefficient operations, they also increase the number of polynomial additions and subtractions needed. While additions and subtractions have a linear cost proportional to the degree of the polynomial n, their overhead may outweigh the multiplication savings for small n. Since the relative cost of multiplication versus addition/subtraction depends on the hardware, the optimal threshold is typically found through testing for each specific cryptographic implementation.

Leveraging the Discrete Fourier Transforms (DFT), in particular conditions the polynomial multiplication can be performed in just $O(n \log_2(n))$ time. This method exploits the equivalence between polynomial multiplication and the convolution of the sequences of coefficients of the operands. The polynomial product is obtained by computing the DFT of these sequences, performing element-wise multiplication on the transformed results, and then applying the inverse Fourier transform. The overall complexity is dominated by the Fourier transform computation, having a complexity of $O(n \log_2(n))$, plus a linear number of coefficient-wise multiplications, resulting in a total multiplication cost of $O(2(n \log_2(n)) + n)$. This technique is applied fruitfully to polynomials in a ring $\mathbb{Z}_q[x] / \langle f(x) \rangle$, provided that $\mathbb{Z}_q$ is a field and the degree of $f(x)$ is a power of two generating the $2n$-th root of unity, and goes by the name of Number-Theoretic Transform (NTT). The aforementioned prerequisites on the polynomials were not commonly employed in cryptography, and many schemes, such as CRYSTALS-Kyber, CRYSTALS-Dilithium, SABER, and NTTRU, have been developed specifically to leverage this multiplication algorithm with asymptotic optimal cost. Willing to multiply two gen-

Table 7.1: *Comparison of polynomial multiplication algorithms. The computation of the asymptotic running time complexity assumes that each modular multiplication takes a constant amount of time.*

Algorithm	Asymptotic complexity	Parameters compatibility	
		q	n
Schoolbook	n^2	any	any
Karatsuba	$n^{\log(3)/\log(2)}$	any	even
Toom-Cook i	$n^{\log(2i-1)/\log(i)}$	any	divisible by i
NTT	$n \log_2 n$	prime values	power-of-two

eral polynomials $\mathbb{Z}_q[x]$ not belonging to the described ring, any operand having up to $(\lceil n/2 \rceil - 1)$-th degree can be safely used with such fast multiplication algorithm. It is however possible to adapt the NTT algorithm for the big-number arithmetic used in the RSA scheme [Bec+22a], or for NTT-unfriendly rings [Chu+21] using any arbitrary combination of the Cooley-Tukey, Good-Thomas, Bluestein, Rader, Rader-Winograd, Binary Rader-Winograd, Bruun, and Johnson-Burrus NTT algorithms. As it is the case for the other sub-quadratic multiplication techniques, also the NTT proves to be practically worth for large values of n.

To conclude, Table 7.1 summarizes the presented techniques to perform the polynomial multiplications, comparing the asymptotic complexity, and the compatibility with the coefficient modulo q and the polynomial maximum degree n.

7.3 Arithmetic in NTRU

The parameters of the NTRU scheme, described in subsection 3.2.1, generate polynomials over the quotient rings

$$R_q \cong \mathbb{Z}_q[x]/\langle \Phi_n \Phi_1 \rangle, \qquad S_q \cong \mathbb{Z}_q[x]/\langle \Phi_n \rangle, \qquad S_p \cong \mathbb{Z}_p[x]/\langle \Phi_n \rangle \tag{7.5}$$

where $\Phi_1 = x - 1$ and $\Phi_n = \frac{x^n - 1}{x - 1} = x^{n-1} + x^{n-2} + \cdots + x + 1$ are the 1-st and the n-th irreducible cyclotomic polynomial, respectively, and $\Phi_n \Phi_1 = x^n - 1$. The modules must handle the arithmetic correctly depending on the defined polynomial rings. However, note that not all operations must be carried out in every ring in the NTRU scheme. Polynomials in $\mathbf{R}_q$ and $\mathbf{S}_q$ are represented with coefficients encoded in two's complement with $\varepsilon_q = \lceil \log_2(q) \rceil$ bits, i.e.: $\mathbb{Z}_q = \left\{ -\frac{q}{2}, -\frac{q}{2} + 1, \cdots, 0, \cdots, \frac{q}{2} - 1 \right\}$.

7.3.1 Polynomial addition/subtraction

The polynomial adder employs a coefficient-wise addition approach, regardless of the polynomial modulus generating the ring. NTRU performs only additions between polynomials in $\mathbf{R}_q$, thus taking the form of the binary operation $\mathbf{R}_q \times \mathbf{R}_q \mapsto \mathbf{R}_q$. The hardware component responsible for this operation has been enhanced to perform α additions simultaneously, allowing α coefficients to be transferred from memory as a single block and completing the task in $\lceil n/\alpha \rceil$ clock cycles. This hardware module handles both addition and subtraction of the coefficients, with a multiplexer selecting the

Algorithm 33 Comba polynomial multiplication algorithm for $\mathbb{Z}_q[x]/\langle x^n \pm 1\rangle$

Require: $n \in \mathbb{N}$: the maximum degree of polynomials
$q \in \mathbb{N}$: the coefficient modulo
$\mathbf{a}, \mathbf{b} \in \mathbb{Z}_q^n$: the two operand vectors containing the coefficients $a(x), b(x) \in \mathbb{Z}_q[x]/\langle x^n \pm 1\rangle$
Ensure: $\mathbf{c} \in \mathbb{Z}_q^n \mid c(x) = a(x) \cdot b(x) \bmod x^n \pm 1$

```
c ← 0
for j ← 0 to n − 1 do
    for i ← 0 to j do
        t ← t + a_{j−i} · b_i                    ▷ Accumulated locally
    c_j ← c_j + t
for j ← 0 to n − 2 do
    for i ← 0 to j do
        t ← t + a_{n−1−j+i} · b_{n−1−i}          ▷ Accumulated locally
    c_{n−1−j} ← c_{n−1−j} ∓ t
```

correct result based on the specified operation. Modular arithmetic in the coefficient ring is highly efficient since q is a power-of-two, retaining only the lower $\lceil \log_2(q) \rceil$ bits of the coefficient addition or subtraction.

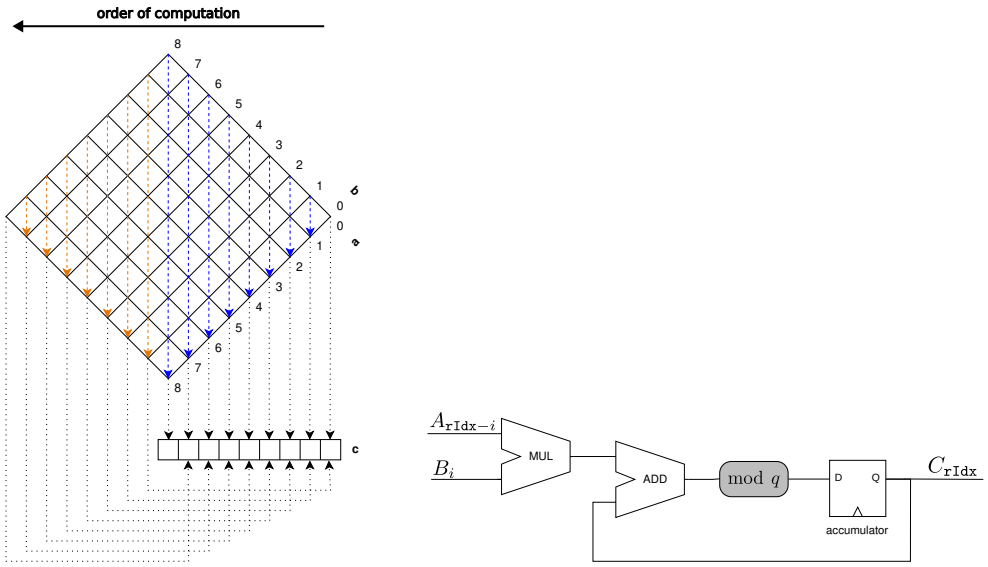
7.3.2 Polynomial multiplication

The multiplication strategy place a critical role in many post-quantum cryptographic scheme due to its extensive use and high cost compared to the other operations in the schemes, especially in NTRU were there is not a clear optimal algorithm choice for both software and hardware implementations. For this reason, this thesis proposes a study of hardware modules implementing different algorithms derived from the schoolbook method and applying different degrees of parallelization.

For the NTRU scheme, multiplications are always performed in the $\mathbf{R}_q$ ring, and the result is eventually embedded in the $\mathbf{S}_q$ or $\mathbf{S}_p$ rings. In all cases except one, the multiplication binary operation is in the form $\mathbf{S}_p \times \mathbf{R}_q \mapsto \mathbf{R}_q$, what is commonly known as *small-by-large* multiplication, which allows to apply some optimizations reducing the area of the multiplier module. Only in one instance during the decapsulation algorithm – when computing $(c - m') \cdot h_q$ – both the operands are in the $\mathbf{R}_q$ ring, requiring the regular *large-by-large* binary multiplication operation $\mathbf{R}_q \times \mathbf{R}_q \mapsto \mathbf{R}_q$.

Comba algorithm

The operand scanning algorithm, also known as the Comba's method [Com90], is a schoolbook multiplication algorithm that reduces the number of memory accesses while still performing $O(n^2)$ coefficient-wise multiplications, while still requiring minimal computational resources. Comba's method optimizes polynomial multiplication by re-arranging the order of single-coefficient multiplications. This ensures that each coefficient of the resulting polynomial is calculated and stored in its final memory location exactly once, processing either from the lowest to highest order coefficients, or vice



(a) *Scheduling of the memory accesses and MAC computation*

(b) *modular MAC unit*

Figure 7.1: *Scheduling of operations (left) and datapath (right) of the Comba multiplier for polynomials in* $\mathbb{Z}_q[x]/\langle x^n \pm 1\rangle$

versa. From an implementation perspective, this approach offers a significant advantage: it reduces memory access to just $2n-1$ read/write operations for the result, compared to the $2n^2$ read/write operations needed by the traditional schoolbook method. In software, this approach results in the smallest code size and register usage, while in hardware the multiplier has a compact area occupation and minimizes the required bandwidth towards the memory.

To achieve a multiplication in $\mathbb{Z}_q[x]/\langle x^n \pm 1\rangle$ with an operand-scanning Comba multiplier, it is sufficient to accumulate the coefficient of the result in the appropriate position, as reported in Algorithm 33 and shown in Figure 7.1a. In particular, Figure 7.1a represents the simple case of a multiplication of two polynomials $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_q[x]/\langle x^9 \pm 1\rangle$, where the intersection points in the grid consists in every single multiplication among a coefficient of $\mathbf{a}$ by a coefficient of $\mathbf{b}$. The vertical lines in the figure connect the sub-multiplication results that needs to be accumulated to produce the partial result of the Comba algorithm. In case the sum of the polynomial degrees of the two operand coefficients is $\leq n-1$, the partial result is added to the first positions of the result polynomial (lines 2–5, highlighted in blue color both in Algorithm 33 and Figure 7.1a). In the other case (lines 6–9, highlighted in green color both in Algorithm 33 and Figure 7.1a), the partial result is added or subtracted to a set of monomials of lower degrees corresponding to the non-null coefficients of the polynomial ring modulus $f(x)$ – in the case example $f(x) = x^9 \pm 1$, the resulting polynomial coefficient having degree $n-9$ is updated.

The Comba multiplier is designed with a highly compact datapath that focuses on

executing a single multiply-and-accumulate (MAC) operation between polynomial coefficients per clock cycle. The result of each operation is stored in an accumulator register. As illustrated in Figure 7.1b, the datapath consists of a multiplier, an adder, and a modulo q reducer. If q is a power of two, the modular reduction step is greatly simplified and reduces to truncating the output of the MAC operation. This design enables efficient processing of one loop iteration from Algorithm 33, specifically lines 3–5 or 9–11, during each clock cycle. The value of the intermediate variable t is preserved within the local accumulator throughout the computation. The final computed value, c_j, is written back to memory only once at the end of each iteration of the outer loops, as specified in lines 2–5 and 6–9 of Algorithm 33.

When performing a standard multiplication followed by a polynomial reduction using Comba's method, the resulting polynomial could have a degree as high as $2n - 1$, which would require twice the memory size to store it. To address this, the traditional Comba algorithm is optimized by ensuring that only the result of a single outer loop iteration from lines 6–9 of Algorithm 33 is retained, while carefully managing the effects of modular reduction. This optimization involves either adding or subtracting the coefficients of monomials with degree greater than $n - 1$ from the results of the regular multiplication, adjusting them to match the corresponding coefficients of the modular multiplication result. Implementing this approach is particularly challenging for the NTRU Prime cryptographic scheme, as the coefficients of monomials with degree greater than $n - 1$ must be added twice, each time to two consecutive coefficients in the modular multiplication result. Furthermore, before storing the sum or difference of the coefficient of the monomial with degree greater than $n - 1$ and the corresponding value in the result accumulator, a modular reduction must be performed. Given that the maximum value resulting from the accumulation is smaller than $2q - 2$, it is feasible to handle the modular reduction simultaneously with the accumulation. This is accomplished through a straightforward selection process within a small chain of adders and subtractors, even when the modulus q is not naturally reduction-friendly (i.e., not a power of two).

The two steps of the algorithm can be executed in parallel in order to halve the computation time, at cost of introducing another modular MAC unit and a slightly more complex Finite State Machine (FSM) handling the case if the two result writes collide on the same result block. It is worth noting that the parallelization of this algorithm leads to the access different parts the two operands at the same time, requiring the serialization of the requests and consequently nullifying the potential performance gains. To this end, the operands need to be prepared modifying their layout in memory blocks in order to fetch the pair of coefficients in a single fetch operation. Consequently, each memory transfer is now retrieving $2\lceil \log_2(q) \rceil < B$ bits, halving the number of operand transfer throughout the algorithm execution.

Algorithm 34 Parallelized schoolbook polynomial multiplication algorithm for $\mathbb{Z}_q[x]/\langle f(x)\rangle$

Require: $n \in \mathbb{N}$: the maximum degree of polynomials
$q \in \mathbb{N}$: the coefficient modulo
$\mathbf{a}, \mathbf{b} \in \mathbb{Z}_q^n$: the two operand vectors containing the coefficients of $a(x), b(x) \in \mathbb{Z}_q[x]/\langle f(x)\rangle$
$f(x)$: a monic polynomial generating the ring $\mathbb{Z}_q[x]/\langle f(x)\rangle$
Ensure: $\mathbf{c} \in \mathbb{Z}_q^n \mid c(x) = a(x) \cdot b(x) \bmod f(x)$

1: $\mathbf{c} \leftarrow \mathbf{0}$
2: **for** $j \leftarrow 0$ **to** $n-1$ **do**
3: $\quad \mathbf{c} \leftarrow \mathbf{c} + b_j \cdot \mathbf{a}$ ▷ *scalar multiplication with n parallel MAC units*
4: $\quad \mathbf{a} \leftarrow \mathbf{a} \cdot [0, 1, 0, \ldots, 0] \bmod f(x)$ ▷ *$a(x) \cdot x \bmod f(x)$ performed via the LFSR structure*
5: **return** $\mathbf{c}$

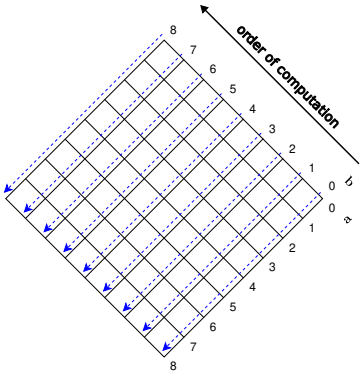


Figure 7.2: *Scheduling of operations in the parallelized schoolbook multiplier for polynomials.*

Parallelized schoolbook

Instead of focusing on algorithmic improvements like Karatsuba or DFT-based methods, another way to accelerate the multiplication is to leverage the parallel nature of the standard schoolbook multiplication Algorithm 31, as depicted in Figure 7.2. In particular, Figure 7.2 represents the simple case of a multiplication of two polynomials $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_q[x]/\langle x^9 \pm 1\rangle$, where the intersection points in the grid consists in every single multiplication among a coefficient of $\mathbf{a}$ by a coefficient of $\mathbf{b}$. With this approach, multiplying a single coefficient of one polynomial by all coefficients of the other polynomial involves independent operations, while showing a predictable and simplified memory access pattern. Each diagonal blue lines in Figure 7.2 represents the parallel sub-multiplications carried out during each iteration. This inherent parallelism allows for a linear-time multiplication algorithm by unrolling the for loop in line 2 using n processing units and $2n$ memory units (one for each coefficient of the accumulated partial result), completing the product in $\Theta(n)$ time. Moreover, it is possible to interleave the reduction by polynomial ring modulus $f(x)$ with each intermediate step of the multiplication algorithm (lines 3 and 4 of Algorithm 31), saving n memory elements required for the computation.

In [LW15] it is proposed a linear-time modular multiplication algorithm specifi-

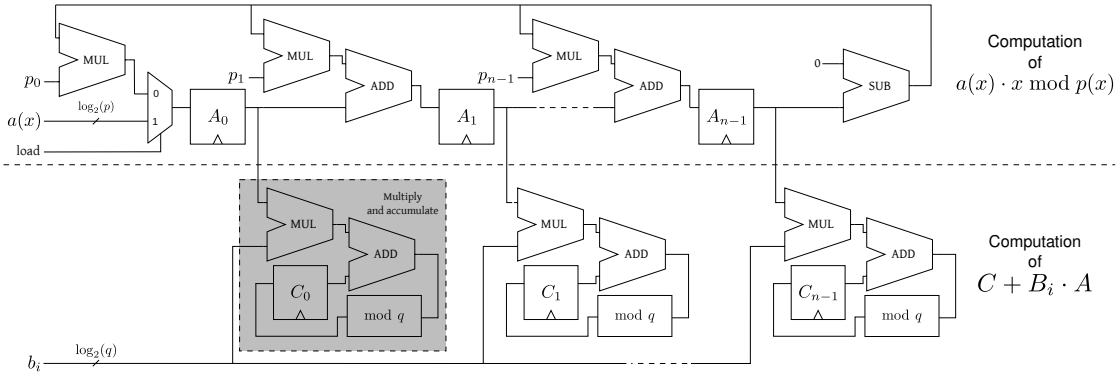


Figure 7.3: *Structure of the parallelized schoolbook multiplier x-net computing the product $r(x) = (a(x) \cdot b(x)) \bmod p(x)$. The top portion of the modular multiplier takes care of computing $x^i \cdot a(x) \bmod p(x)$ at the i-th clock cycle, while the bottom part performs the vector scalar multiplication.*

cally designed for the NTRUEncrypt polynomial ring. Their method uses n parallel MAC units to achieve multiplication in n clock cycles. To minimize the area of each MAC unit, they replaced the multiplier with a multiplexer. This multiplexer selects one of three possible multiplication outcomes, leveraging the small coefficient size of the operand in $\mathbf{R}_p$, where $p = 3$. This technique has since been adapted for the polynomial rings used in SABER, NTRU, and NTRU Prime [BR21; DMG23; Far+19]. Authors in [BR21] suggested a centralized approach to pre-compute the limited set of possible coefficient-wise multiplication results and then distribute them to the MAC units. [Pen+23] proposed delaying the modulo q coefficient reduction of the multiplication result's coefficients until the end of the multiplication when the coefficients are read-out. While this requires larger accumulators to store the resulting polynomial's coefficients, it reduces area by needing only a single modular reduction unit.

In the following, it is assumed that the polynomial modulus $f(x)$ is monic, as it is often the case in practice. The modular polynomial multiplication, $a(x) \cdot b(x) = c(x) \bmod f(x)$, is broken down into a vector-scalar multiplication and a polynomial addition. The first step involves multiplying a single coefficient of the second operand by the entire first operand (line 3), followed by multiplications by x and modular reductions of the first operand (line 4). This decomposition of the modular multiplication operation allows an efficient hardware implementation. The coefficient-wise multiplications in line 3 expose significant data parallelism, while the modular multiplication by x and the subsequent reduction can be efficiently realized using an Linear-Feedback Shift Register (LFSR) structure. The hardware structure of the generic parallelized schoolbook modular multiplier for a monic $f(x)$, hereafter named *x-net* for brevity after the multiplication by x of one operand by the LFSR structure, is depicted in Figure 7.3.

Generic hardware design The coefficient-by-polynomial multiplication (line 3 in Algorithm 31) is computed with n independent MAC elements that compute the product of

the coefficient b_i by each coefficient of polynomial $a(x)$, and add the result to the corresponding coefficient of $c(x)$. The corresponding portion of the circuit in Figure 7.3 is the bottom half, where one MAC element is highlighted in grey. A single MAC element is composed by an integer multiplier, an adder, a modular reducer $\mod q$, and a register containing the value of the coefficient $c_i, 0 \leq i < n$.

The computation of the multiplication of the first factor by x, $a(x) \leftarrow a(x) \cdot x$ is efficiently done by storing the coefficients of $a(x)$ in a shift register, as the multiplication by x acts shifting the coefficients by one position towards higher degree monomials (to the right, in Figure 7.3). Since the degree of $a(x)$ is at most $n-1$ before the multiplication by x, the modular reduction $a(x) \leftarrow a(x) \cdot x \mod f(x)$ can be efficiently computed. Indeed, since $f(x)$ is monic, computing the remainder of $a(x) \cdot x \mod f(x)$ is equivalent to the subtraction from $a(x) \cdot x$ of the polynomial $a_{n-1} \cdot (f(x) - x^n)$.

The multiplication and modular reduction are performed in the same clock cycle by the the portion of the x-net multiplier managing the operation (top portion of Figure 7.3). This circuit, structured as a shift register with feedback, performs the $a(x) \cdot x$ shifting the contents of the registers containing $(a_0, \ldots a_{n-1})$ towards right. The same circuit also subtracts $a_{n-1} \cdot (f(x) - x^n)$ from $a(x) \cdot x$ by adding the coefficients of $-a_{n-1} \cdot (f(x) - x^n)$ to the ones of $a(x) \cdot x$. This is done inserting the adders on the shift lines between any two elements of the shift-register that contains $a(x)$. This feedback network structure will thus need as many multipliers and adders as the number of non-null coefficients in $f(x)$, benefiting from values of $f(x)$ with a very small number of coefficients. Finally, note that the shift-register structure also allows to perform the loading of $a(x)$ with minimal additional hardware. Indeed, $a(x)$ in this design is loaded coefficient-wise from a_{n-1} to a_0, inserting a single mux (represented on the left in Figure 7.3).

Structural optimizations The first observation leading to an optimization is that the topmost portion of the x-net multiplier may operate entirely with values $\mod p$, leading to a significant saving in the resource consumption for the cases where $p \ll q$. The lifting required to multiply coefficient in $\mathbb{Z}_p$ by coefficients in $\mathbb{Z}_q$ is efficiently realized within the multiplier units in the MAC elements by sign-extending the two's complement representation of the $\mathbb{Z}_p$ elements.

The second observation leading to an optimization is that, in case p is very small, as it is the case in this cryptosystems, the multiplier in the MAC can be substituted by a multiplexer that selects among a small set of fixed multiples of b_i, which are in turn computed by a small number of additions. Taking as an example $p = 5$, the multiplier is substituted by a multiplexer selecting among the values $\{-2b_i, -b_i, 0, b_i, 2b_i\}$, depending on the value of the coefficient of the $a(x)$ polynomial. The values can be either pre-computed only once, and distributed, or computed within the MAC unit and selected in place. The latter approach requires a larger amount of resources for each single MAC unit, while obtaining a reduction in the wiring congestion, which may be particularly beneficial for FPGA targeted implementations.

A final point concerning the optimization of the x-net multiplier is the trade-off be-

tween performing modular reductions in the MAC complex managing the coefficients of the result, and performing the reductions upon result readout. Choosing to perform the modular reductions at readout requires wider accumulator registers for $c(x)$; in particular, their size grows from $\lceil \log_2 (q) \rceil$ to $\lceil \log_2 (npq) \rceil$ bits, as n values $\bmod q$ will be multiplied by a value $\bmod p$ and added by the x-net multiplier during its operation. This increase in area is however compensated by the removal of n modular reducers $\bmod q$ from each multiply and accumulate complex, enacting a trade-off that typically gains in area consumption, unless the reduction by q is trivial (e.g., when q is a power of two). In the former case, each accumulator register has $\log_2(q)$ bits size, and the $\bmod q$ operation is performed by conditionally applying additions and subtractions. Since the distance between each integer multiplication result and a valid $\mathbb{Z}_q$ element is at most $(q-1) \cdot \lceil (p-1)/2 \rceil$, then $\lceil (p-1)/2 \rceil$ additions and subtractions are carried out in parallel with values multiple of q and the only valid result in $\mathbb{Z}_q$ is selected. In case of the reduction operation performed during the readout, a single Barrett reduction module is used.

The described architecture uses n clock cycles to load the $a(x)$ from memory, n cycles to compute the result of the modular multiplication (potentially without coefficient-wise modular reduction), and n cycles to read out the final polynomial multiplication result and store it into the memory. This process can be sped up devising a memory bus transferring multiple polynomial coefficients at once. Transferring α, β and γ coefficients for respectively $a(x)$, $b(x)$, and $c(x)$, the overall latency of a polynomial multiplication is $\lceil n/\alpha \rceil + \lceil n/\beta \rceil + \lceil n/\gamma \rceil$. Loading α coefficients of $a(x)$ for each clock cycle is achieved transferring them in parallel from main memory, and having the shift register containing $\mathbf{a}$ rotate by α positions at each clock cycle through appropriate connections. The same approach is applied for reading out γ coefficients of the result from the accumulator registers, possibly instantiating γ parallel Barrett modules when performing the reductions-at-readout approach. To compute the multiplication of β $\mathbb{Z}_q$ coefficients in parallel, a total of $\beta \cdot n$ MACs are required. Indeed, to compute the result of β multiplication steps, β multiplications and sums need to be computed at each clock cycle, to obtain the result which is to be stored $\beta - 1$ cells to the right of each MAC unit. In this thesis the generalized architecture using $\beta \neq 1$ is denoted as x^β-net. Note that β steps of the update of $a(x)$ should be computed in a single step. This in turn may require to perform $\beta - 1$ sign flips of the $\mathbb{Z}_p$ coefficient and additional multiply and additions for specific MAC units depending on the value of the modulus polynomial $f(x)$.

Specialization for NTRU parameters Considering the parameters defined by the NTRU scheme, q is always a power-of-two, and the polynomial modulus $f(x)$ is $x^n - 1$. Consequently, the $\bmod q$ operation is equivalent to keeping the lower $\log_2 q$ bits of the MAC result and discarding the remaining one without requiring any computation to perform the modulo of coefficients, while the feedback network of the LFSR simplifies in just a rotation of the coefficients of the polynomial $a(x)$.

NTRU mainly uses $\mathbf{S}_p \times \mathbf{R}_q \mapsto \mathbf{R}_q$ multiplications, and in only one occasion a

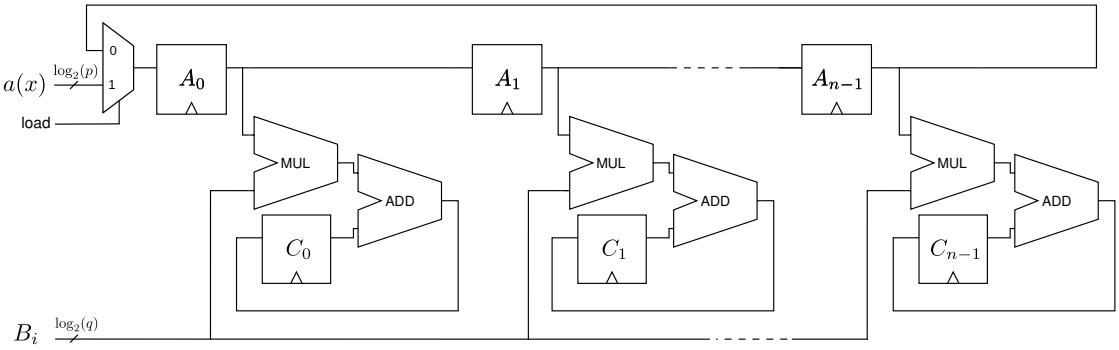


Figure 7.4: *Structure of the parallelized schoolbook multiplier (x-net) computing the product $r(x) = (a(x) \cdot b(x)) \bmod x^n - 1$ specifically for the NTRU scheme. The top portion of the modular multiplier takes care of computing $x^i \cdot a(x) \bmod x^n - 1$ at the i-th clock cycle, while the bottom part performs the vector scalar multiplication.*

$\mathbf{R}_q \times \mathbf{R}_q \mapsto \mathbf{R}_q$ multiplication during the decapsulation (when computing $((c - m') \cdot h_q)$). In the first case, the x-net multiplier can be further optimized being $p = 3$ an extremely small number, as all the possible results for the multiplication of the coefficient b_i at clock cycle i by any value of $\mathbb{Z}_p$ can be pre-computed and distributed to each MAC unit. To reduce the effect of routing congestion in the critical path due to the large x-net structure, the pre-computed results are stored in a register before being used in the MAC units, allowing the EDA tools to replicate such register throughout the silicon area and reduce the average distance between the pre-computed values and the MAC units. The result of the application of all this optimizations is represented in the Figure 7.4. During the decapsulation such optimization cannot be applied as both operands are in $\mathbf{R}_q$, thus in that case a regular multiplier is instantiated in the MAC unit.

7.3.3 Ring embed and lift

A peculiarity of the NTRU scheme is the use of three different polynomial rings throughout the entire algorithm. The process of moving an element from a smaller ring to a larger one is called *lift*, whereas the opposite process is called *embed*.

After the computation of multiplications, the result $a(x) \in \mathbf{R}_q$ may need to be transformed into an element in $\mathbf{S}_q$ or $\mathbf{S}_p$, for which two embed functions $\mathrm{E}_1 : \mathbf{R}_q \mapsto \mathbf{S}_q$ and $\mathrm{E}_2 : \mathbf{R}_q \mapsto \mathbf{S}_p$ are necessary. The latter function can be derived applying the embedding function E_1 and then applying a third embedding function $\mathrm{E}_3 : \mathbf{S}_q \mapsto \mathbf{S}_p$ such that $\mathrm{E}_2 = \mathrm{E}_1 \circ \mathrm{E}_3$.

For the specific definition of rings $\mathbf{R}_q$ and $\mathbf{S}_q$, the embed function E_1 corresponds to a division of the input polynomial $a(x)$ by $\Phi_1 = x - 1$. This operation can be efficiently carried out by subtracting the coefficient a_{n-1} having the highest degree to all the other coefficients. Regarding the embed function E_3, this operation corresponds to the computation of the coefficient modulo remainder $\bmod p$. Given the value of $p = 3$,

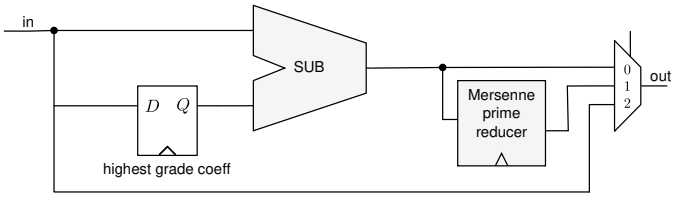


Figure 7.5: *Hardware module computing the embedding functions* $\mathrm{E}_1 : \mathbf{R}_q \mapsto \mathbf{S}_q$ *and* $\mathrm{E}_2 : \mathbf{R}_q \mapsto \mathbf{S}_p$ *defined in* NTRU *depending on the signal driving the output multiplexer*

Algorithm 35 LIFT operation in NTRU-HRSS using multiplications

Require: $a \in \mathcal{S}_p$

Ensure: $b \in \mathcal{R}_q \mid \mathtt{S_p}(b) = a$
1: **for** $i \leftarrow 0$ **to** $n-2$ **do**
2: $\quad c_i \leftarrow (1-i) \mod p$ ▷ $c \leftarrow \mathtt{S_p}(1/\Phi_1)$ *for NTRU parameters*
3: $t \leftarrow \mathtt{S_p}(a \cdot c)$ ▷ *small-by-small multiplication*
4: ▷ *multiplication by* Φ_1 *replaced by additions* ◁
5: $b_0 \leftarrow (t_{n-1} - t_0) \mod q$
6: **for** $i \leftarrow 1$ **to** $n-1$ **do**
7: $\quad b_i \leftarrow (t_{i-1} - t_i) \mod q$
8: **return** b

the best approach consists in using a Mersenne prime modulo algorithm. The embed hardware module is either attached to the output of the x-net polynomial multiplier unit, or working independently accessing the coefficients of $a(x)$ directly from a bus connected to the memory. In the first case, the coefficients must be streamed out from the multiplier unit starting from the highest degree a_{n-1} towards the one with lower degree. The embed module, depicted in Figure 7.5, is able to select which embedding function apply, but is also able to process a configurable amount of coefficients of the input polynomial $a(x)$ per clock cycle in parallel.

The LIFT operation maps elements $a(x) \in \mathcal{S}_p$ in wider rings $\mathcal{R}_q$ such that

$$\mathbf{a}' \leftarrow \mathtt{Lift}(\mathbf{a}) \Rightarrow \mathbf{a}' \bmod (p, \Phi_n) = \mathbf{a} \tag{7.6}$$

In NTRU-HPS it is just the sign extension of the coefficients, whereas NTRU-HRSS employs a non trivial lift function

$$\mathtt{Lift} : \mathbf{a} \rightarrow \Phi_1 \cdot ((\mathbf{a}/\Phi_1) \bmod (p, \Phi_n)) \tag{7.7}$$

This last operation, described in Algorithm 35, can be performed by one multiplication with $c = {}^1/_{\Phi_n}$, then followed by reduction in $\mathbf{S}_p$, and lastly multiplied by Φ_1. As the first step, the polynomial c is algorithmically generated via small decreasing counter with reset value equal to 2. Afterwards, the small-by-large multiplication is carried out using the polynomial multiplier unit. Finally, the multiplication by Φ_1 is again performed via subtractions.

Algorithm 36 LIFT operation in NTRU-HRSS without using multiplications

Require: $a \in \mathcal{S}_p$

Ensure: $b \in \mathcal{R}_q \mid \mathtt{S_p}(b) = a$

for $i \leftarrow 0$ **to** $n-2$ **do**
  $c_i \leftarrow (1-i) \mod p$ ▷ $c \leftarrow \mathtt{S_p}(1/\Phi_1)$ *for NTRU parameters*
for $i \leftarrow 0$ **to** $p-1$ **do**
  $d_i \leftarrow \langle x^i \bar{c}, a \rangle$ ▷ *inner-product of* a *with the rotated reversal map of* c
for $i \leftarrow p$ **to** $n-1$ **do**
  $d_i \leftarrow d_{i-p} - \sum_{j=0}^{p-1} a_{i-j}$
$d_0 \leftarrow d_0 - d_{n-1} \mod p$
$b_0 \leftarrow -d_0$
▷ *multiplication by* Φ_1 *replaced by additions* ◁
for $i \leftarrow 1$ **to** $n-1$ **do**
  $d_i \leftarrow d_i - d_{n-1} \mod p$
  $b_i \leftarrow d_{i-1} - d_i \mod q$
return b

However, leveraging on the structure of the polynomial rings [Hül+17] introduces a new algorithm, here reported in Algorithm 36, that only makes use of addition and subtraction operations. The algorithm produces the rotated reversal map of $\mathrm{S}_p\left(1/\Phi_1\right)$ on the fly with a similar procedure of the other algorithm. Each clock cycle p inner products are carried out simultaneously, with multiple coefficients of the input polynomial accessed sequentially. Afterwards, the input polynomial is accessed linearly maintaining the last p accessed coefficients in a buffer, and subtract their sum to the coefficient computed p cycles before. Finally, the reduction in $\mathbf{S}_p$ and multiplication by $\Phi_1 = x - 1$ are carried simultaneously by means of simple subtractions working on multiple coefficients in the same clock cycle. This algorithm has a time complexity of $\Theta(3n)$, and is composed by many elementary operations that required about 1.5% the area of a small-by-large x-net multiplier. Another advantage of this module is that can be scheduled in parallel to other multiplication operations, hiding the latency of such operation in case of the use of ax-net multiplier, or vastly speeding-up the operation in case a slow multiplier is available.

7.4 Arithmetic in HQC

HQC.PPKE primitives use multiplications and additions between polynomials in $\mathbf{R} = \mathbb{F}_2[x]/\langle x^p - 1 \rangle$, with p a prime number, and in $\mathbf{R}_w \subset \mathbf{R}$, the set of all polynomials in $\mathbf{R}$ having Hamming weight equal to w, $w \approx \sqrt{p}$. We commonly refer to a polynomial in the former set as *dense polynomial*, represented in memory as a sequence of p binary coefficients in little-endian format organized as a list of $\lceil \frac{p}{B} \rceil$ words with a configurable bit-length B (e.g., $B \in \{32, 64, 128\}$), and the polynomial in the latter class as *sparse polynomial*, stored in memory as a sequence of 16-bit unsigned integers for all parameters sets, each of which is intended as the exponent of a non-null monomial

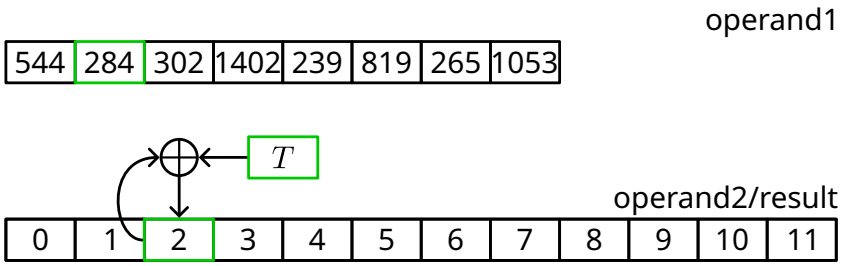


Figure 7.6: *Addition of a sparse polynomial $a \in \mathbf{R}_w$ to a dense one $b \in \mathbf{R}$. The algorithm works in-place, and the dense operand becomes the result after the computation. The word T is a B-bits word containing a single bit in a specific position determined by the processed index of the sparse operand that causes a bit-flip in b.*

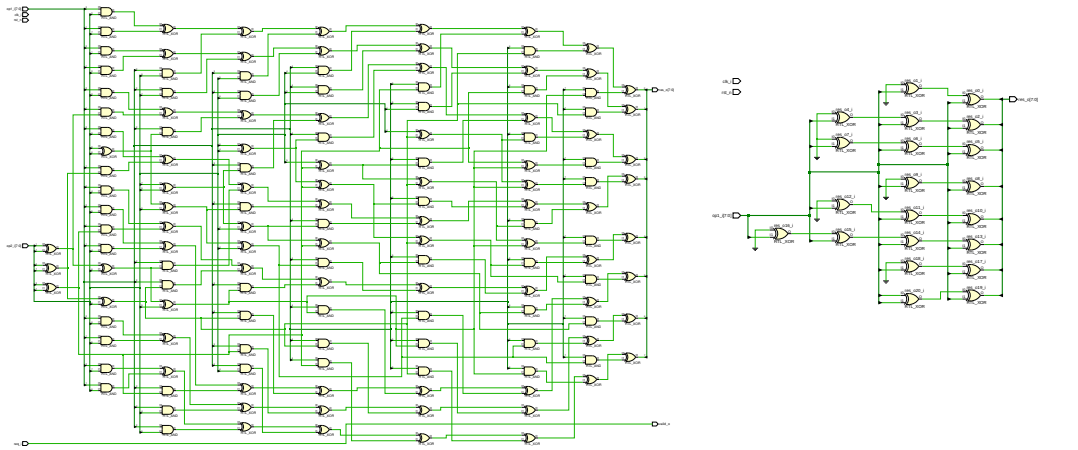
$(x^i, 0 \leq i \leq p-1)$.

Additionally, HQC employs a Reed-Solomon (RS) code in the concatenated RM/RS concatenated code which interprets the message and codeword data in m-bit long bit strings called *symbols*. Each symbol is then mapped to an element of the extension field $\mathbb{F}_{2^m}$, with $m = 8$, to perform some arithmetic operations on it. Recalling that each element in $\mathbb{F}_{2^m}$ can be considered as a polynomials of degree less than m, the arithmetic used by the symbols in the RS code in HQC is given by the polynomial field $\mathbb{F}_2[x]/\langle f(x)\rangle$, where $f(x)$ is the pentanomial $x^8 + x^4 + x^3 + x^2 + 1$ irreducible in $\mathbb{F}_2$.

7.4.1 Polynomial addition/subtraction

Addition and subtraction among two binary polynomials $a(x), b(x)$ of maximum degree n corresponds to a coefficient-wise Boolean `xor` between their vector representations (i.e., $a_i \oplus b_i$, $0 \leq i < n$). When dealing with polynomials $a, b \in \mathbf{R}$ having maximum degree $p > 10000$, their size imposes to split their binary coefficients into $\lceil \frac{p}{B} \rceil$ consecutive words, and carry out B-bit `xors` each clock cycle.

Moreover, HQC.PPKE-ENCRYPT requires two additions between a sparse and a dense polynomial, and HQC.PPKE-KEYGENERATION also performs another one. This operation is realized by flipping the bits of the dense operand in the positions indicated by the sparse one. Recalling that the indexes of the sparse polynomial are stored as 16-bits unsigned integers, we split it to determine the dense operand word address as the highest $16 - \log_2(B)$ bits of the index encoding, while the remaining bits of the index are encoding the bit position within the word to flip. An example is provided in Figure 7.6, where T denotes a B-sized word containing a single bit set in a position where a bit-flip of the result is required. Notice that this is an algorithm working in-place, thus after the computation the dense operand is lost as it is transformed into the result. Since two consecutive indexes may flip bits in the same memory word, creating a *read-after-write* data dependency, it is employed a straightforward sequential architecture that, even considering the non-negligible memory access latencies, does not penalize the overall performance thanks to the low weight of the sparse polynomials



(a) *Multiplication between any two polynomials* **(b)** *Multiplication of any polynomial by* $x^5 + 1$

Figure 7.7: *Multiplication circuits for polynomials in the field* $\mathbb{F}_2[x]/\langle x^8 + x^4 + x^3 + x^2 + 1\rangle$

(see Table 4.1). Each parameter set determines a specific number of memory words to process, which is used as initialization value of a counter determining the number of iterations in the task.

7.4.2 Polynomial multiplication

The algorithm performing polynomial multiplications are reported in Algorithm 31. However, starting from n-bits long operands, the result is up to $2n$ bits long. Working in a polynomial ring, the equivalence class representative must be computed as the remainder of the modulo operation with the modulus polynomial $f(x)$.

Multiplication in $\mathbb{F}_2[x]/\langle x^8 + x^4 + x^3 + x^2 + 1\rangle$

For this polynomial field the same parallelized schoolbook algorithm of NTRU described in subsubsection 7.3.2 is employed. The realized x-net architecture has 4 LFSR taps in positions corresponding to the non-null monomials of the modulus polynomial $x^8 + x^4 + x^3 + x^2 + 1$. Given the small size of the polynomials in the field $\mathbb{F}_2[x]/\langle x^8 + x^4 + x^3 + x^2 + 1\rangle$ and the straightforward coefficient arithmetic in $\mathbb{F}_2$, a fully unrolled x^8-net architecture is used, generating a combinatorial network computing each multiplication in a single clock cycle, here depicted in Figure 7.7a. The critical path of such combinatorial circuit has a depth of only 8 two-input logic gates, and the output fanout of the gates is well balanced, allowing to reach high working frequencies in ASIC designs. Moreover, due to the LUT combining optimization applied during the synthesis for FPGA targets, the same circuit gets synthesized and compacted in a few LUTs, and the resulting critical path is only passing through three LUTs.

The prescribed RS generator polynomial $g(x)$ is specified in the HQC specification, and has a fixed value for every instance of HQC key pairs. When one multiplication operand has a fixed value, the inputs of the `and` gates have a fixed value. By looking at the truth table of the `and` operation it is clear that if one operand has a zero value, the output of the gate is always zero, independently from the value of the second operand. Therefore, the constant zero value is propagate to the following layers of the circuit. When the fixed input of the `and` gate is 1, then the second operand is passing through the gate untouched, thus it is possible to remove that gate from the circuit. An example of such an optimized circuit is represented in Figure 7.7b when the multiplication operand $x^5 + 1 \in \mathbb{F}_2^8$ is used, however consider that the optimization level depends on the actual value of the fixed operand.

Multiplication in R

Considering the arithmetic in $\mathbf{R}$, due to the polynomial modulus $x^p - 1$, the ring $\mathbf{R}$ has a cyclic structure. Therefore, knowing that $x^p \equiv 1$, the product of two polynomials $a, b \in \mathbf{R}$, derived from Algorithm 31, in $\mathbb{F}_2[x]$ simplifies in

$$c_i = \bigoplus_{j+k \bmod p = i} (a_j \wedge b_k), \qquad i, j, k \in \{0, 1, \ldots, p-1\} \tag{7.8}$$

Considering the parameter p values for `hqc128`, `hqc192`, `hqc256` are 17669, 35851, 57637, the polynomials a and b are extremely large. A naive implementation of the schoolbook algorithm having a quadratic time complexity in p would be extremely slow, completing the task in millions of clock cycles. However, the HQC.PPKE primitives require only multiplications where one operand is a sparse and the other one is dense, hereafter referred to as *sparse-by-dense* multiplications. While sub-quadratic approaches for generic polynomial multiplication exist, in the HQC the amount of non-null coefficients of a polynomial $a \in \mathbf{R}_w$ is $w \approx \sqrt{p}$ out of p. Therefore, in case of a multiplication $\mathbf{R}_w \times \mathbf{R} \mapsto \mathbf{R}$, the schoolbook approach from Algorithm 28 has an asymptotic complexity of just $O(p^{1.5})$, which is already better than the one of the Karatsuba approach. Furthermore, the schoolbook method does not hide large constants within the asymptotic notation, and allows for a resource-sparing implementation. Therefore, in the proposed polynomial multiplier component a *shift-and-add* approach is used, where the dense operand is shifted by an amount of bits specified by each index of the sparse operand, and accumulated into the result.

The number of accumulator memory words are reduced to the minimum possible by immediately reducing modulo $x^p - 1$ the shifted polynomial to be added, therefore interleaving each shift-and-add operation with the modulo computation. Due to the structure of the polynomial ring $\mathbf{R} = \mathbb{F}_2[x] / \langle x^p - 1 \rangle$, this is possible through a simple change in the location where the coefficients of the dense polynomial are added, as the polynomial modular reduction modulo $x^p - 1$ amounts to a bit-wise `xor` of the coefficients of degrees greater or equal than p onto the coefficients of the monomials of the result having a degree lower by exactly p units.

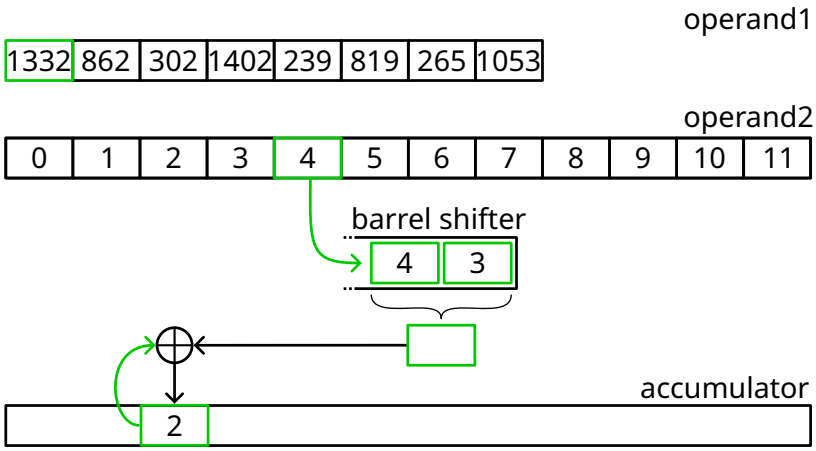


Figure 7.8: *Multiplication of a sparse polynomial $a \in \mathbf{R}_w$ by a dense polynomial $b \in \mathbf{R}$*

The word-wise shift-and-add approach rotates and accumulates a B-bit word each clock cycle, in turn taking $w \cdot \lceil \frac{p}{B} \rceil$ clock cycles and using $\lceil \frac{p}{B} \rceil$ temporary memory words for the accumulator. Each processed index of the sparse operand uniquely identifies a memory word of the dense operand starting from which all its blocks are read out in a cyclic sequence. The words of the accumulator are retrieved, updated, and written back in sequence starting from the first word to the last one. The working principle is represented in Figure 7.8. Since p is a prime value, $p \nmid B$, hence the last memory word contains some padding bits not encoding the coefficients of the dense polynomial, and some care is required in managing this memory word. Note that this algorithm runs in constant time as the memories of this RTL design do not feature caches, thus fully eliminating the timing side channel which would be present in an analogue software implementation.

Furthermore, the design improves by $l\times$ the latency of the sparse-by-dense polynomial multiplication processing l sparse indexes in parallel. To this end, l `read` memory ports accessing the dense binary polynomial operand are employed, along with a `read` and `write` memory port for the accumulated result, and a `read` memory port for the sparse polynomial. A similar, yet sub-optimal, approach would use a single `read` memory port accessing the dense binary polynomial operand, along with l `read` and `write` memory ports for the accumulated result, and a `read` memory port for the sparse polynomial. This solution was discarded because it would require l independent p-sized accumulators, and an extra round of computation merging the distinct accumulators to retrieve the final result.

Due to the potentially prominent size of the memory words B, it is employ a pipelined Barrel module to perform the shift operation, and it is parametrized in the number of pipeline stages to break the possibly long critical path and improve performance.

To accomplish the support of all parameter set, the number of indexes i of the sparse polynomial operand to process is received as input to the module upon the start of oper-

ation, masking out the accumulation resulting from the processing of invalid indexes in case l does not divide i.

In Table 7.2 are reported the area and performance figures of such devised architecture when the `hqc256` parameters set is used, and varying the word size of the transfers between memories, the number l of read memory ports, and the number of stages of the Barrel shifter. The latency of the multiplier clearly is inversely proportional to both the word size, and the number of read ports. As the word size increments, a Barrel shifter with more pipeline stages proves to be beneficial for the maximum working frequency, but has a tangible contribution in the additional FFs used by the design.

7.5 Arithmetic in CROSS

Considering the three primitives CROSS.KEYGENERATION (Algorithm 14), CROSS.SIGN (Algorithm 15) and CROSS.VERIFY (Algorithm 16) detailed in section 4.3, the main arithmetic operations involve vectors in $\mathbb{F}_p$ and $\mathbb{F}_z$. In particular, CROSS uses the subtractions and point-wise multiplications among vectors, the exponentiation of the generator element $g \in \mathbb{F}_p$ of order z by each element of a $\mathbb{F}_z$ vector, and the vector-matrix multiplications. Each sub-operation among scalar elements of the vectors is performed modulo the small prime numbers $p, z \in \{7, 127, 509\}$. Therefore, it is possible to use the optimized reduction algorithm for Mersenne primes, described in Algorithm 30, when $p, z \in \{7, 127\}$, and use the generic Barrett reduction, reported in Algorithm 29, for $p = 509$.

7.5.1 Vector addition/subtraction and point-wise multiplication

The addition/subtraction and point-wise multiplication among vectors can be performed as described in Algorithm 27, potentially leveraging multiple adders, multipliers, and modulo units to compute multiple coefficients in parallel. Considering memory words of 64 bits, up to 21, 9, and 7 parallel operations can be carried out for p, z equal to 7, 127, and 509, respectively.

In Table 7.3 are reported the results of a synthesis targeting the AMD Artix-7 FPGA of the modules implementing the addition or subtraction (upper half) and point-wise multiplication (lower half) among two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{F}_p$ for all the parameter sets defined by the CROSS specification. The modulo p is small enough that the synthesizer implement the multiplication logic in LUT. The inferred DSP units are due to the first multiplication within the 7 parallel Barrett modulo units. Note the remarkable efficiency of the computation of the modulo remainder when the modulo is a Mersenne prime, both in terms of occupied area ($\approx {}^{1}/_{6}$ considering the eSlice indicator) and working frequency.

Table 7.2: *Performance of the polynomial multipliers when using the* `hqc256` *parameter set. The resource usage for the other parameter sets do not vary significantly, and only the latency is influenced by a different parameter set choice. Area-Time (AT) product in eSlices · ms*

Word size	Read ports	Shift stages	Resources LUT	FF	BRAM	eSlice	Freq. MHz	Latency CC	μs	AT prod.
32	1	1	403	405	4.5	1055	248	236723	954	1006
		2	466	411	4.5	1071	249	236724	950	1017
		3	483	448	4.5	1075	243	236725	974	1047
		4	487	487	4.5	1076	262	236726	903	971
	4	1	1127	803	7.0	1766	210	59637	283	499
		2	1341	922	7.0	1820	238	59638	250	455
		3	1417	1065	7.0	1839	234	59639	254	467
		4	1408	1205	7.0	1836	232	59640	257	471
64	1	1	696	655	4.5	1128	227	118692	522	588
		2	848	665	4.5	1166	251	118693	472	550
		3	828	744	4.5	1161	267	118694	444	515
		4	825	808	4.5	1161	261	118695	454	527
	4	1	2615	1306	7.0	2138	183	29904	163	348
		2	2566	1529	7.0	2126	237	29905	126	267
		3	2538	1829	7.0	2119	222	29906	134	283
		4	2552	2099	7.0	2122	250	29907	119	252
128	1	1	963	1161	6.5	1619	173	59742	345	558
		2	1435	1173	6.5	1737	211	59743	283	491
		3	1442	1315	6.5	1739	254	59744	235	408
		4	1572	1464	6.5	1771	258	59745	231	409
	4	1	4962	2333	11.0	3573	155	15054	97	346
		2	4779	2747	11.0	3527	194	15055	77	271
		3	4544	3304	11.0	3468	214	15056	70	242
		4	5101	3903	11.0	3608	233	15057	64	230

Table 7.3: *Synthesis results for addition/subtraction (top) and point-wise multiplication (bottom) among* $\mathbb{F}_p^n$ *vectors when targeting an AMD Artix-7 FPGA and using* 64*-bits word sizes.* CROSS *arithmetic units are agnostic to the fast, balanced, and small optimization corners.*

Arithmetic operation	Parameter set	Resources LUT	FF	DSP	eSlice	Freq. MHz	Latency CC
+/−	`CROSS-RSDP-1`	247	304	0	62	530	33
	`CROSS-RSDP-3`						39
	`CROSS-RSDP-5`						46
	`CROSS-RSDPG-1`	448	296	0	112	544	26
	`CROSS-RSDPG-3`						30
	`CROSS-RSDPG-5`						34
$\odot$	`CROSS-RSDP-1`	809	598	0	203	415	34
	`CROSS-RSDP-3`						40
	`CROSS-RSDP-5`						47
	`CROSS-RSDPG-1`	1085	727	7	1213	205	28
	`CROSS-RSDPG-3`						32
	`CROSS-RSDPG-5`						36

Table 7.4: *Synthesis results for exponentiation of a $\mathbb{F}_z^n$ vector to a $\mathbb{F}_p^n$ one using as base the public generator element g when targeting an AMD Artix-7 FPGA and using 64-bits word sizes. CROSS arithmetic units are agnostic to the fast, balanced, and small optimization corners.*

Parameter set	Resources				Freq.	Latency
	LUT	FF	DSP	eSlice	MHz	CC
CROSS-RSDP-1						26
CROSS-RSDP-3	360	365	0	90	328	30
CROSS-RSDP-5						34
CROSS-RSDPG-1						33
CROSS-RSDPG-3	539	358	0	135	260	39
CROSS-RSDPG-5						46

7.5.2 Vector exponentiation

The modular exponentiation $g^a \bmod p$ for some $g \in \mathbb{F}_p^*$ of order z and some $a \in \mathbb{F}_z$ may use the *square-and-multiply* technique, requiring $O(\log_2 a)$ modular multiplications. However, when considering that g is an element fixed in the specification (in particular equal to 2 or 16 for R-SDP or R-SDP(G) parameter sets) and that the field $\mathbb{F}_z$ has a small number of elements, a look-up table with z entries of $\mathbb{F}_p$ elements can be used to efficiently perform such operation.

Specifically, for R-SDP and R-SDP(G) parameters a smaller $7 \times \log_2 127$ table serializable in just 49 bits, and a larger $127 \times \log_2 509$ table of ≈ 1 KiB can be employed. This technique is particularly interesting for FPGA solutions where moderately large look-up tables can be implemented in just few LUTs, particularly the aforementioned tables fit in just 7 and 18 LUTs, respectively. Similarly to the subtraction and point-wise multiplication, multiple exponentiations can be performed in parallel by replicating the look-up tables, or by allowing multiple read ports to such Read-Only Memory (ROM) memory.

However, considering a word size of 64-bits, the number $\mathbb{F}_p$ and $\mathbb{F}_z$ coefficients encoded in a single word differs, as $z < p$. Therefore, a small FIFO buffer at the input of the exponentiation unit is used to compose a smaller word having fewer $\mathbb{F}_z$ coefficients in it, matching the number of the $\mathbb{F}_p$ ones contained in a single 64-bits word.

In Table 7.4 is reported the result of a synthesis for the AMD Artix-7 FPGA when using words of 64-bits, computing $^{64}/_{\log_2 p}$ exponentiations in parallel via look-up tables every clock cycle. Due to the LUT resources in the FPGA fabric, this solution has minimal area impact, and reaches high working frequencies for both R-SDP and R-SDP(G) parameters. The difference in size of the required look-up tables for the R-SDP and R-SDP(G) parameters explains the difference in LUT usage and gap in the working frequency.

7.5.3 Vector-matrix multiplication

Regarding the vector-matrix multiplications, only two matrices $\overline{\mathbf{M}}$ and $\mathbf{H}^\top$ are involved in such operation. Recall that $\overline{\mathbf{M}} \in \mathbb{F}_z^{m \times n}$ and $\mathbf{H}^\top \in \mathbb{F}_p^{n \times (n-k)}$ are in their systematic forms $\overline{\mathbf{M}} = \left[\overline{\mathbf{W}}, \bar{\mathbf{I}}_m\right]$ and $\mathbf{H}^\top = \left[\mathbf{V}^\top \mid \mathbf{I}_{n-k}\right]$, with $\overline{\mathbf{W}} \in \mathbb{F}_z^{m \times (n-m)}$ and $\mathbf{V}^\top \in \mathbb{F}_p^{k \times (n-k)}$, here reported their partially expanded versions for a visual aid.

$$\overline{\mathbf{M}} = \begin{bmatrix} \overline{W}_{0,0} & \overline{W}_{0,1} & \cdots & \overline{W}_{0,n-m-1} & 1 & 0 & \cdots & 0 \\ \overline{W}_{1,0} & \overline{W}_{1,1} & \cdots & \overline{W}_{1,n-m-1} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \overline{W}_{m-1,0} & \overline{W}_{m-1,1} & \cdots & \overline{W}_{m-1,n-m-1} & 0 & 0 & \cdots & 1 \end{bmatrix} \tag{7.9}$$

$$\mathbf{H}^\top = \begin{bmatrix} V_{0,0}^\top & V_{0,1}^\top & \cdots & V_{0,n-k-1}^\top \\ V_{1,0}^\top & V_{1,1}^\top & \cdots & V_{1,n-k-1}^\top \\ \vdots & \vdots & \ddots & \vdots \\ V_{k-1,0}^\top & V_{k-1,1}^\top & \cdots & V_{k-1,n-k-1}^\top \\ 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \tag{7.10}$$

Furthermore, consider that the matrices $\overline{\mathbf{W}}$ and $\mathbf{V}^\top$ are serialized row-wise, hence they are received by the arithmetic modules as the concatenations:

$$\overline{W}_{0,0} \| \overline{W}_{0,1} \| ... \| \overline{W}_{0,n-m-1} \| \overline{W}_{1,0} \| \overline{W}_{1,1} \| ... \| \overline{W}_{1,n-m-1} \| \overline{W}_{m-1,0} \| \overline{W}_{m-1,1} \| ... \| \overline{W}_{m-1,n-m-1}$$

$$V_{0,0}^\top \| V_{0,1}^\top \| ... \| V_{0,n-k-1}^\top \| V_{1,0}^\top \| V_{1,1}^\top \| ... \| V_{1,n-k-1}^\top \| V_{k-1,0}^\top \| V_{k-1,1}^\top \| ... \| V_{k-1,n-k-1}^\top$$

Taking as an example the operation $\bar{\mathbf{e}} = \bar{\mathbf{e}}_G \overline{\mathbf{M}}$, it is possible to leverage the systematic form of the matrix $\overline{\mathbf{M}}$ to simplify the computation:

$$\bar{\mathbf{e}} = \bar{\mathbf{e}}_G \overline{\mathbf{M}} = \bar{\mathbf{e}}_G \left[\overline{\mathbf{W}}, \bar{\mathbf{I}}_m\right] = \left[\bar{\mathbf{e}}_G \overline{\mathbf{W}}, \bar{\mathbf{e}}_G\right] \tag{7.11}$$

Therefore, the last m elements of the resulting vector $\bar{\mathbf{e}}$ correspond to $\bar{\mathbf{e}}_G$, and a smaller vector-matrix multiplication is performed to compute the first $n - m$ elements of the result. Due to the order of the received matrix coefficients, the used algorithm slightly differs from the usual schoolbook *row-by-vector* approach where the inner product of the input vector by the transposed i-th matrix column is performed to produce the i-th result element. In this case, a scalar-vector multiplication between the i-th vector coefficient of $\bar{\mathbf{e}}_G$ and the i-th matrix row is accumulated in a $n - m$ result buffer, which yields $\bar{\mathbf{e}}_G \overline{\mathbf{W}}$ after m iterations. Therefore, this simplified vector-matrix multiplication takes only $O(m(n-m)) < O(mn)$ multiplications in $\mathbb{F}_z$.

Table 7.5: *Synthesis results for CROSS vector-matrix multiplications vectors when targeting an AMD Artix-7 FPGA and using 64-bits word sizes for the transmission of vectors and 192-bits for the transfer of the matrix rows. CROSS arithmetic units are agnostic to the fast, balanced, and small optimization corners.*

Matrix operator	Parameter set	Resources LUT	FF	DSP	eSlice	Freq. MHz	Latency CC
$\mathbf{H}^\top$	CROSS-RSDP-1	2776	1351	0	694	149	221
	CROSS-RSDP-3	2780	1541	0	695	149	427
	CROSS-RSDP-5	2985	1427	0	747	149	719
	CROSS-RSDPG-1	3515	1162	21	3704	120	74
	CROSS-RSDPG-3	3538	1352	21	3709	118	146
	CROSS-RSDPG-5	3555	1356	21	3714	119	194
$\overline{\mathbf{M}}$	CROSS-RSDPG-1	2755	1015	0	689	149	122
	CROSS-RSDPG-3	2767	1033	0	692	149	176
	CROSS-RSDPG-5	2772	1230	0	693	149	267

Considering now the operation $\mathbf{s} = \mathbf{e}\mathbf{H}^\top$, the systematic form of the matrix $\mathbf{H}^\top$ can be similarly leveraged to simplify the computation:

$$\mathbf{s} = \mathbf{e}\mathbf{H}^\top = \mathbf{e}\left[\mathbf{V} \mid \mathbf{I}_{n-k}\right] = [e_k, e_{k+1}, \ldots, e_n] + [e_0, e_1, \ldots, e_{k-1}]\,\mathbf{V}^\top \tag{7.12}$$

This time the $n-k$ result buffer is pre-initialized with the last $n-k$ coefficients of $\mathbf{e}$, and a smaller $[e_k, e_{k+1}, \ldots, e_n] + [e_0, e_1, \ldots, e_{k-1}]\,\mathbf{V}^\top$ vector-matrix multiplication is performed using the same scalar-vector multiplication approach. The resulting computational complexity is therefore decreased from $O(n(n-k))$ arithmetic operations in $\mathbb{F}_p$ to just $O(k(n-k))$.

In Table 7.5 are reported the synthesis for the AMD Artix-7 FPGA of the vector-matrix multiplications for both the matrices $\mathbf{H}^\top$ and $\overline{\mathbf{M}}$ leveraging their systematic form. The vector operator and result are transferred using 64-bits words, while the systematic part of the matrix, either $\mathbf{V}$ or $\overline{\mathbf{W}}$, are accessed using larger words of 192-bits to speed-up the computation. Consequently, $\lfloor 192/\log_2 p \rfloor$ parallel modular multiplications in $\mathbb{F}_p$ are carried out in parallel each clock cycle for the design specialized for the $\mathbf{H}^\top$ matrix, instantiating 21 or 27 multiplier units when using the R-SDP or R-SDP(G) parameters, respectively. Considering the unit specialized for the $\overline{\mathbf{M}}$ matrix operator, $\lfloor 192/\log_2 z \rfloor = 64$ parallel modular multiplications in $\mathbb{F}_z$ are performed each clock cycle. When the modulo is a Mersenne prime, the resulting designs are able to work with a clock frequency up to 150 MHz. Conversely, each Barrett reduction unit, which are compatible with a generic modulo value, make use of a DSP unit that became part of the critical path, stepping back the maximum reachable working frequency to around 120 MHz.

7.6 Arithmetic in lattice-based schemes

During the PQC standardization contest, four lattice-based KEMs stood out during the first three rounds of analysis.

The KEM candidate CRYSTALS-Kyber was picked for immediate standardization as ML-KEM scheme. This scheme is based on Module-LWR (M-LWR) , and provides three parameters sets, `kyber512`, `kyber768`, and `kyber1024`, for a security margin equivalent to the one of `AES-128`, `AES-192`, and `AES-256`, respectively. All the three parameters sets work in the same NTT friendly polynomial ring algebraic structure $\mathbf{R}_q = \mathbb{F}_{3319}/\langle x^{256}+1\rangle$, and varies the module rank to increase the size of the lattice.

NTRU was officially recommended as the fallback alternative in case patent issues cannot be solved by the end of 2023 [Ala+22a]. As a further testimony of NTRU's security and efficiency, Google LLC adopted it as the key encapsulation method of choice in its internal infrastructure [ISE; Sch]. The polynomial rings used by their parameters sets are $\mathbf{R}_q = \mathbb{Z}_{2^k}/\langle x^p-1\rangle$, for some $k \in \mathbb{N}$ and p being a prime value. The NTRU-HPS specification defines four parameters sets, `ntruhps2048509`, `ntruhps2048677`, and `ntruhps4096821` providing increasing protection equivalent to the ones of AES, while NTRU-HRSS only provides one parameters set for `ntruhrss701` having security margin equivalent to the one of `AES-192`.

NTRU Prime is an NTRU variant with conservative choices in the underlying algebraic structure, which prevent a number of attacks preemptively, and the errors are generated deterministically via rounding operations. Thanks to its conservative design choices, it has been adopted and employed by default in the hybrid mode of OpenSSH [The22], the most widely diffused implementation of the Secure SHell (SSH) protocol suite starting from the release version 9.0. The algebraic structure consists in the polynomial fields $\mathbf{R}_q = \mathbb{Z}_q/\langle x^p-x-1\rangle$, with p and q prime numbers defined by their parameters sets. Two different schemes are proposed, one based on the NTRU lattice under the name of Streamlined NTRU Prime, and another based on the R-LWR problem and named NTRU LPRime. For each scheme, five parameters sets are defined, with the first three corresponding to the security margins offered by AES: `sntrup653`,`sntrup761`, `sntrup857`, `sntrup953`, `sntrup1013`, and `sntrup1277` for the former scheme, and `ntrulpr653`,`ntrulpr761`, `ntrulpr857`, `ntrulpr953`, `ntrulpr1013`, and `ntrulpr1277` for the latter.

SABER is based on the M-LWR algebraic problem, which is at least as computationally hard as the M-LWE one of CRYSTALS-Kyber. The polynomial ring used by the scheme is $\mathbf{R}_q = \mathbb{Z}_{2^k}/\langle x^{256}+1\rangle$, for some $k \in \mathbb{N}$. The specification mandates three parameters sets, `lightsaber`, `saber`, and `firesaber`, having the same security margins of the parameters of AES.

The four lattice-based cryptosystems previously mentioned are built upon the arithmetic of polynomials with integer coefficients modulo q, where q is either a power of two or a small prime number. These computations occur within a ring defined by a polynomial modulus characterized by a limited number of terms. Additionally, each

Table 7.6: *Summary of the features of the polynomial rings* $\mathbf{R}_p = \mathbb{Z}_p[x] / \langle f(x) \rangle$ *and* $\mathbf{R}_q = \mathbb{Z}_q[x] / \langle f(x) \rangle$ *for each lattice-based KEM.*

Cryptographic Scheme	q	p	n	$\mathbf{f(x)}$
NTRU	2^i values	3	prime values	$x^n - 1$
NTRU Prime	prime values	3	prime values	$x^n - x - 1$
Kyber	prime values	5, 7	2^i 256	$x^n + 1$
SABER	2^i values	7, 9, 11	2^i 256	$x^n + 1$

scheme employs polynomials whose coefficients belong to smaller fields $\mathbb{Z}_p$, where $p \in 3, 5, 7, 9, 11$. A summary of the polynomial ring characteristics for each scheme is presented in Table 7.6.

Recalling the compatibility of the parameters with sub-quadratic multiplication techniques summarized in Table 7.1, the efficient NTT algorithm running in $\Theta(n \log_2(n)))$ sequential steps and requiring a polynomial modulus with a power-of-two degree with the coefficients being in a field, is clearly only compatible with CRYSTALS-Kyber. On the other hand, optimized versions of the schoolbook algorithm, which has a computational complexity of $O(n^2)$, such as the method introduced by Comba [Com90], are universally applicable. These approaches enable highly compact designs but come at the cost of lower throughput. Multiplication algorithms implemented via divide-and-conquer strategies, such as Karatsuba or Toom-Cook, introduce additional design complexity and larger constants hidden within the $\mathcal{O}$ notation, but they provide a consistent reduction in the complexity exponent.

In the following subsections are described several flexible hardware components capable of adapting to the different ring structures of the aforementioned lattice-based cryptographic schemes, either via a parameters selection at synthesis time for best efficiency, or even at runtime to enhance the flexibility of the hardware accelerator integrating it. The second case is particularly interesting as can enable cryptographic agility due to the compatibility with several PQC schemes without the need of replacing the hardware component. Two base designs are used, the parallelized schoolbook (x-net) multiplier algorithm for polynomials described in subsubsection 7.3.2 and generalized in Figure 7.3, and the compact Comba multiplier presented in subsubsection 7.3.2.

The CRYSTALS-Kyber specification indicates that both private and public keys are stored in the NTT-transformed domain to reduce the number of NTT computations. This approach provides a computational speed advantage but comes at the expense of cryptographic flexibility when other multiplication techniques are used. However, there is still some degree of flexibility when the key pairs are used by the same multiplication algorithm implementation, such as in the host performing the key generation and decapsulation primitives, making worth exploring different solutions. In this context, it is

Table 7.7: *Number of $\mathbf{R}_p \times \mathbf{R}_q \mapsto \mathbf{R}_q$ multiplications in key generation, encapsulation and decapsulation primitives of each cryptographic scheme. One further $\mathbf{R}_q \times \mathbf{R}_q \mapsto \mathbf{R}_q$ multiplication is performed during the decapsulation in NTRU, denoted by a * symbol. Module-based cryptographic schemes SABER and CRYSTALS-Kyber perform one $k \times k$ matrix-vector and one or two vector-vector multiplications, where the elements are built from the coefficients of polynomials in $\mathbf{R}_q$ or $\mathbf{R}_p$, during key generation, encapsulation and decapsulation, respectively.*

Cryptographic scheme	Module rank k	$\mathbf{R}_p \times \mathbf{R}_q$ Multiplications		
		KeyGen.	Encap.	Decap.
NTRU	1	5	1	$2^\star$
Streamlined NTRU Prime	1	1	1	3
NTRU LPRime	1	1	2	3
Kyber	2,3,4	k^2	k^2+k	k^2+2k
SABER	2,3,4	k^2	k^2+k	k^2+2k

assumed that the presented multipliers performing operations with CRYSTALS-Kyber polynomials receive keypairs that have already been converted back to the canonical domain during the loading process. This approach is well-suited for scenarios where long-term keypairs are employed, and cryptographic flexibility is required. Examples include smartcards, IPSec-based VPNs, instant messaging protocols, and the SSH transport layer protocol [Ylo06].

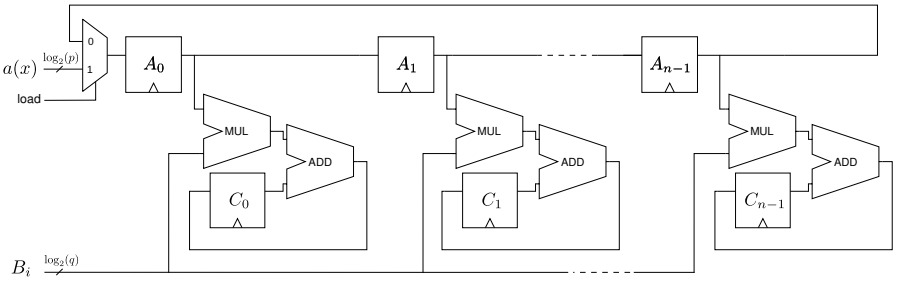
The correctness of the multiplier results was verified using testbenches derived from a synthetic computation model developed in SageMath. This model produced known answer tests aligned with the reference implementations of the cryptographic ciphers across all rings defined by the parameters sets of CRYSTALS-Kyber, NTRU, SABER, and NTRU Prime cryptographic schemes.

To report the overall latency of computations accelerated by the designs across the four cryptosystems, the number of accelerated multiplications required for each KEM primitive, namely the key generation, encapsulation, and decapsulation, are reported in Table 7.7. The latency figures are therefore based on the sequential execution of the necessary multiplication operations. Note that the Area-Time (AT) product keeps constant even in case even multiple parallel multipliers are deployed, provided the scheme allows for data parallelism.

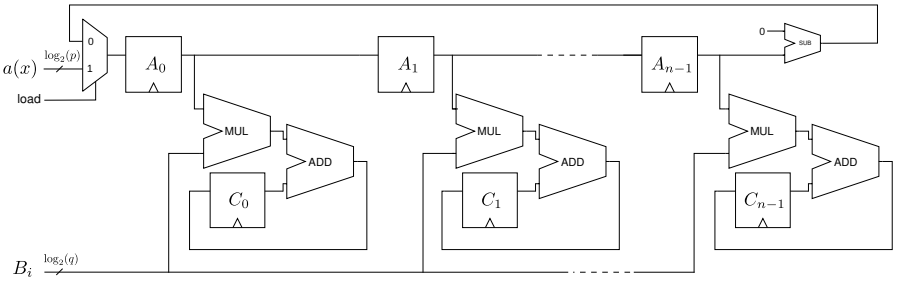
Specialized and unified parallelized schoolbook designs

While x-net based multipliers require a non negligible amount of resources, they show very good performance and flexibility. Starting from the generic architecture in Figure 7.3, the specializations of the x-net designs optimizing for the specific parameters of the polynomial ring defined by NTRU, SABER, NTRU Prime and CRYSTALS-Kyber are shown in Figure 7.9.

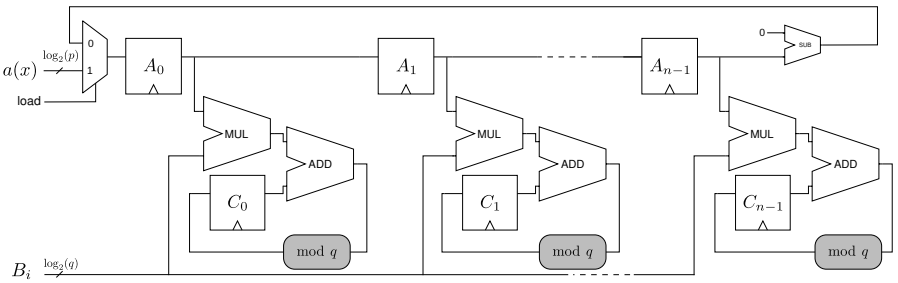
Specialized designs For the NTRU polynomial ring, the modulus q is a power of two, and the modulus polynomial is defined as $f(x) = x^n - 1$. This structure enables the



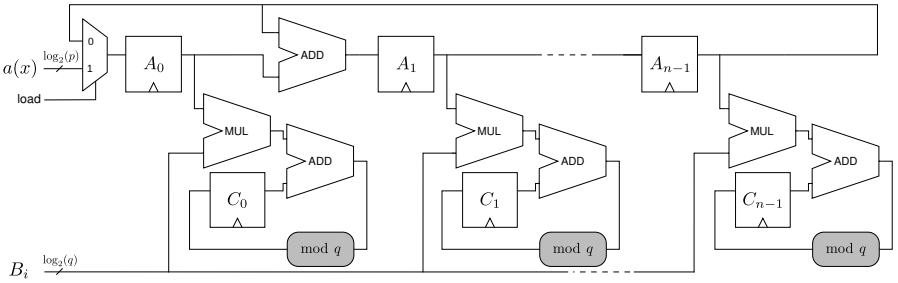
(a) *Parallelized schoolbook design tailored for NTRU polynomial ring*



(b) *Parallelized schoolbook design tailored for SABER polynomial ring*



(c) *Parallelized schoolbook design tailored for Kyber polynomial ring*



(d) *Parallelized schoolbook design tailored for NTRU Prime polynomial ring*

Figure 7.9: *Parallelized schoolbook architectures specifically tailored for each polynomial ring. The readout circuit of the accumulators is omitted for clarity. The* $\mod q$ *reducer is not present whenever the reduction is performed upon result readout.*

optimizations shown in Figure 7.9a. Modular reduction is a straightforward truncation of the first $\lceil \log_2(q) \rceil$ bits, removing the need for modulo q components. The feedback term $-a_{n-1} \cdot (f(x) - x^n)$ in the LFSR structure simplifies to adding a_{n-1} as the constant term because $f(x) - x^n = -1$. Since the product $a \cdot x$ always has $a_0 = 0$, no additional adder is required.

For the case of SABER, the modulus q remains a power-of-two, which allows for the same truncation adopted by the NTRU case. The polynomial modulus $f(x) = x^n + 1$ involves adding the $-a_{n-1} \cdot (f(x) - x^n)$ result to the first operand polynomial $a(x)$, which in this case is equivalent to subtracting a_{n-1} from the zero a_0 coefficient of $a(x) \cdot x$. As a result, a subtractor is introduced on the feedback line, where 0 is used as the minuend and a_{n-1} as the subtrahend. The resulting design is represented in Figure 7.9b.

The x-net design for CRYSTALS-Kyber, depicted in Figure 7.9c, requires a modulo q operation either in all the MAC units after the computation of the accumulation, or during the readout phase by employing larger accumulators. The polynomial modulus $f(x) = x^n + 1$ is the same as the one of SABER, so the computation of $a(x) \cdot x \bmod f(x)$ again results in $-a_{n-1}$ being added to the zero a_0 coefficient of $a(x) \cdot x$, requiring to compute the additive inverse element of a_{n-1}.

Finally, the design of the x-net multiplier for NTRU Prime, also requires to perform one or more $\bmod q$ units, depending if used in every MAC unit or in the readout circuit. The modulus for NTRU Prime is $f(x) = x^n - x - 1$, which means that adding $-a_{n-1} \cdot (f(x) - x^n)$ is equivalent to adding $-a_{n-1} \cdot (-x - 1) = a_{n-1}x + a_{n-1}$ to $a(x) \cdot x$. While adding a_{n-1} does not require an actual adder as in the case of NTRU, adding $a_{n-1}x$ does require a coefficient-wise addition of $a_{n-1} + a_1$, with a_1 being the coefficient of the x monomial in $a(x) \cdot x$, hence the the constant term of $a(x)$. Therefore, the feedback network for the x-net design of NTRU Prime, shown in Figure 7.9d, includes an adder that takes as inputs a_{n-1} and a_0 from $a(x)$.

A synthesis campaign is set for all the prescribed parameters sets of all the four cryptoschemes on a Xilinx UltraScale+ platform, gathering the Configurable Logic Blocks (CLBs) usage as the area occupation indicator, the latency taken for a single polynomial multiplication, and the maximum working frequency reached by the design. Additionally, the total latency for all $\mathbf{R}_p \times \mathbf{R}_q \mapsto \mathbf{R}_q$ multiplications involved in the key generation, encapsulation, and decapsulation of the scheme are computed considering the number of operations used in each primitive that are reported in Table 7.7. Both coefficient ring modulo strategies outlined in subsubsection 7.3.2 are assessed, either performed in every MAC unit, or deferred during the readout but requiring larger accumulators, to determine which approach is more suitable when targeting an FPGA design.

In particular, the latter strategy yields an significant performance and area gains in the designs of CRYSTALS-Kyber and NTRU Prime, although at the cost of a moderate increase in the number of needed FFs. On the other hand, the multipliers specialized for SABER and NTRU do not benefit from such strategy, as it comes at cost of an increased

Table 7.8: *Results of the synthesis targeting an Xilinx UltraScale+ ZCU106 FPGA specialized for each supported parameter sets. The design is based on the x-net algorithm, when 4 $\mathcal{R}_p$ and 1 $\mathcal{R}_q$ coefficients are loaded/read per clock cycle. One further $\mathcal{R}_q \times \mathcal{R}_q$ multiplication is performed during the decapsulation in NTRU, denoted by $\star$ symbol. Area-Time product computed as latency (ms) $\times$ CLB*

(a) *modulo reduction operation performed in each MAC unit*

Security level	Parameter set	CLB	CC	Freq. MHz	Latency (µs) Keyg.	Enc.	Dec.	AT product Keyg.	Enc.	Dec.
AES-128	kyber512	4226	583	312	7.47	11.21	14.95	31.58	47.37	63.17
	ntruhps2048509	2150	1153	638	9.04	1.81	3.61	19.42	3.88	7.77*
	sntrup653	8411	1477	275	5.37	5.37	16.11	45.17	45.17	135.52
	ntrulpr653	8411	1477	275	5.37	10.74	16.11	45.17	90.34	135.52
	lightsaber	3245	583	497	4.69	7.04	9.38	15.22	22.83	30.45
AES-192	kyber768	3202	583	328	16.00	21.33	26.66	51.22	68.29	85.37
	ntruhps2048677	2825	1531	625	12.25	2.45	4.90	34.60	6.92	13.84*
	ntruhrss701	3336	1585	600	13.21	2.64	5.28	44.06	8.81	17.62*
	sntrup761	9691	1720	325	5.29	5.29	15.88	51.28	51.28	153.86
	ntrulpr761	9691	1720	325	5.29	10.58	15.88	51.28	102.57	153.86
	saber	3019	583	553	9.49	12.65	15.81	28.64	38.19	47.74
AES-256	kyber1024	3202	583	328	28.44	35.55	42.66	91.06	113.82	136.59
	ntruhps4096821	3712	1855	562	16.50	3.30	6.60	61.26	12.25	24.50*
	sntrup857	11142	1936	312	6.21	6.21	18.62	69.13	69.13	207.41
	ntrulpr857	11142	1936	312	6.21	12.41	18.62	69.13	138.27	207.41
	firesaber	2468	583	581	16.06	20.07	24.08	39.62	49.52	59.43
above AES-256	sntrup953	12770	2152	312	6.90	6.90	20.69	88.08	88.08	264.24
	ntrulpr953	12770	2152	312	6.90	13.79	20.69	88.08	176.16	264.24
	sntrup1013	13017	2287	275	8.32	8.32	24.95	108.25	108.25	324.76
	ntrulpr1013	13017	2287	275	8.32	16.63	24.95	108.25	216.50	324.76
	sntrup1277	16686	2881	262	11.00	11.00	32.99	183.48	183.48	550.44
	ntrulpr1277	16686	2881	262	11.00	21.99	32.99	183.48	366.96	550.44

(b) *modulo reduction operation performed at readout*

Security level	Parameter set	CLB	CC	Freq. MHz	Latency (µs) Keyg.	Enc.	Dec.	AT product Keyg.	Enc.	Dec.
AES-128	kyber512	3186	585	328	7.13	10.70	14.27	22.72	34.09	45.45
	sntrup653	8138	1479	288	5.14	5.14	15.41	41.79	41.79	125.37
	ntrulpr653	8138	1479	288	5.14	10.27	15.41	41.79	83.58	125.37
AES-192	kyber768	2615	585	312	16.88	22.50	28.12	44.12	58.83	73.54
	sntrup761	9043	1722	312	5.52	5.52	16.56	49.91	49.91	149.73
	ntrulpr761	9043	1722	312	5.52	11.04	16.56	49.91	99.82	149.73
AES-256	kyber1024	2615	585	312	30.00	37.50	45.00	78.45	98.06	117.67
	sntrup857	10141	1938	312	6.21	6.21	18.63	62.99	62.99	188.97
	ntrulpr857	10141	1938	312	6.21	12.42	18.63	62.99	125.98	188.97
above AES-256	sntrup953	11073	2154	312	6.90	6.90	20.71	76.44	76.44	229.33
	ntrulpr953	11073	2154	312	6.90	13.81	20.71	76.44	152.89	229.33
	sntrup1013	12022	2289	312	7.34	7.34	22.01	88.19	88.19	264.59
	ntrulpr1013	12022	2289	312	7.34	14.67	22.01	88.19	176.39	264.59
	sntrup1277	14735	2883	325	8.87	8.87	26.61	130.71	130.71	392.13
	ntrulpr1277	14735	2883	325	8.87	17.74	26.61	130.71	261.42	392.13

Table 7.9: *Results of the synthesis targeting an Xilinx UltraScale+ ZCU106 FPGA specialized for each supported parameter sets. The design is based on the x-net algorithm, $4\,\mathcal{R}_p$ and $2\,\mathcal{R}_q$ coefficients are loaded/read per clock cycle. One further $\mathcal{R}_q \times \mathcal{R}_q$ multiplication is performed during the decapsulation in NTRU, denoted by $\star$ symbol. Area-Time product computed as latency (ms) $\times$ CLB*

(a) *modulo reduction operation performed in each MAC unit*

Security level	Parameter set	CLB	CC	Freq. MHz	Latency (µs) Keyg.	Enc.	Dec.	AT product Keyg.	Enc.	Dec.
AES-128	kyber512	6508	327	197	6.64	9.96	13.28	43.21	64.81	86.42
	ntruhps2048509	4455	645	438	7.36	1.47	2.95	32.80	6.56	13.12*
	sntrup653	13992	825	200	4.12	4.12	12.38	57.71	57.71	173.15
	ntrulpr653	13992	825	200	4.12	8.25	12.38	57.71	115.43	173.15
	lightsaber	5796	327	400	3.27	4.91	6.54	18.95	28.42	37.90
AES-192	kyber768	5579	327	206	14.29	19.05	23.81	79.70	106.27	132.83
	ntruhps2048677	6208	855	475	9.00	1.80	3.60	55.87	11.17	22.34*
	ntruhrss701	7428	885	425	10.41	2.08	4.16	77.33	15.46	30.93*
	sntrup761	16429	960	188	5.11	5.11	15.32	83.89	83.89	251.67
	ntrulpr761	16429	960	188	5.11	10.21	15.32	83.89	167.78	251.67
	saber	4993	327	425	6.92	9.23	11.54	34.57	46.10	57.62
AES-256	kyber1024	5579	327	206	25.40	31.75	38.10	141.69	177.11	212.54
	ntruhps4096821	8052	1035	438	11.82	2.36	4.73	95.13	19.02	38.05*
	sntrup857	19034	1080	188	5.74	5.74	17.23	109.34	109.34	328.03
	ntrulpr857	19034	1080	188	5.74	11.49	17.23	109.34	218.68	328.03
	firesaber	4200	327	375	13.95	17.44	20.93	58.59	73.24	87.89
above AES-256	sntrup953	21165	1200	188	6.38	6.38	19.15	135.09	135.09	405.28
	ntrulpr953	21165	1200	188	6.38	12.77	19.15	135.09	270.19	405.28
	sntrup1013	22219	1275	188	6.78	6.78	20.35	150.68	150.68	452.06
	ntrulpr1013	22219	1275	188	6.78	13.56	20.35	150.68	301.37	452.06
	sntrup1277	27283	1605	200	8.03	8.03	24.07	218.94	218.94	656.83
	ntrulpr1277	27283	1605	200	8.03	16.05	24.07	218.94	437.89	656.83

(b) *modulo reduction operation performed at readout*

Security level	Parameter set	CLB	CC	Freq. MHz	Latency (µs) Keyg.	Enc.	Dec.	AT product Keyg.	Enc.	Dec.
AES-128	kyber512	5460	329	291	4.52	6.78	9.04	24.69	37.03	49.38
	sntrup653	11867	827	238	3.47	3.47	10.42	41.23	41.23	123.70
	ntrulpr653	11867	827	238	3.47	6.95	10.42	41.23	82.47	123.70
AES-192	kyber768	4733	329	291	10.18	13.57	16.96	48.15	64.21	80.26
	sntrup761	13762	962	238	4.04	4.04	12.13	55.62	55.62	166.87
	ntrulpr761	13762	962	238	4.04	8.08	12.13	55.62	111.25	166.87
AES-256	kyber1024	4733	329	291	18.09	22.61	27.13	85.61	107.02	12.842
	sntrup857	15404	1082	238	4.55	4.55	13.64	70.02	70.02	21.008
	ntrulpr857	15404	1082	238	4.55	9.09	13.64	70.02	140.05	21.008
above AES-256	sntrup953	18111	1202	238	5.05	5.05	15.15	91.46	91.46	274.40
	ntrulpr953	18111	1202	238	5.05	10.10	15.15	91.46	182.93	274.40
	sntrup1013	19201	1277	238	5.37	5.37	16.10	103.02	103.02	309.07
	ntrulpr1013	19201	1277	238	5.37	10.73	16.10	103.02	206.04	309.07
	sntrup1277	23332	1607	238	6.75	6.75	20.26	157.54	157.54	472.62
	ntrulpr1277	23332	1607	238	6.75	13.50	20.26	157.54	315.08	472.62

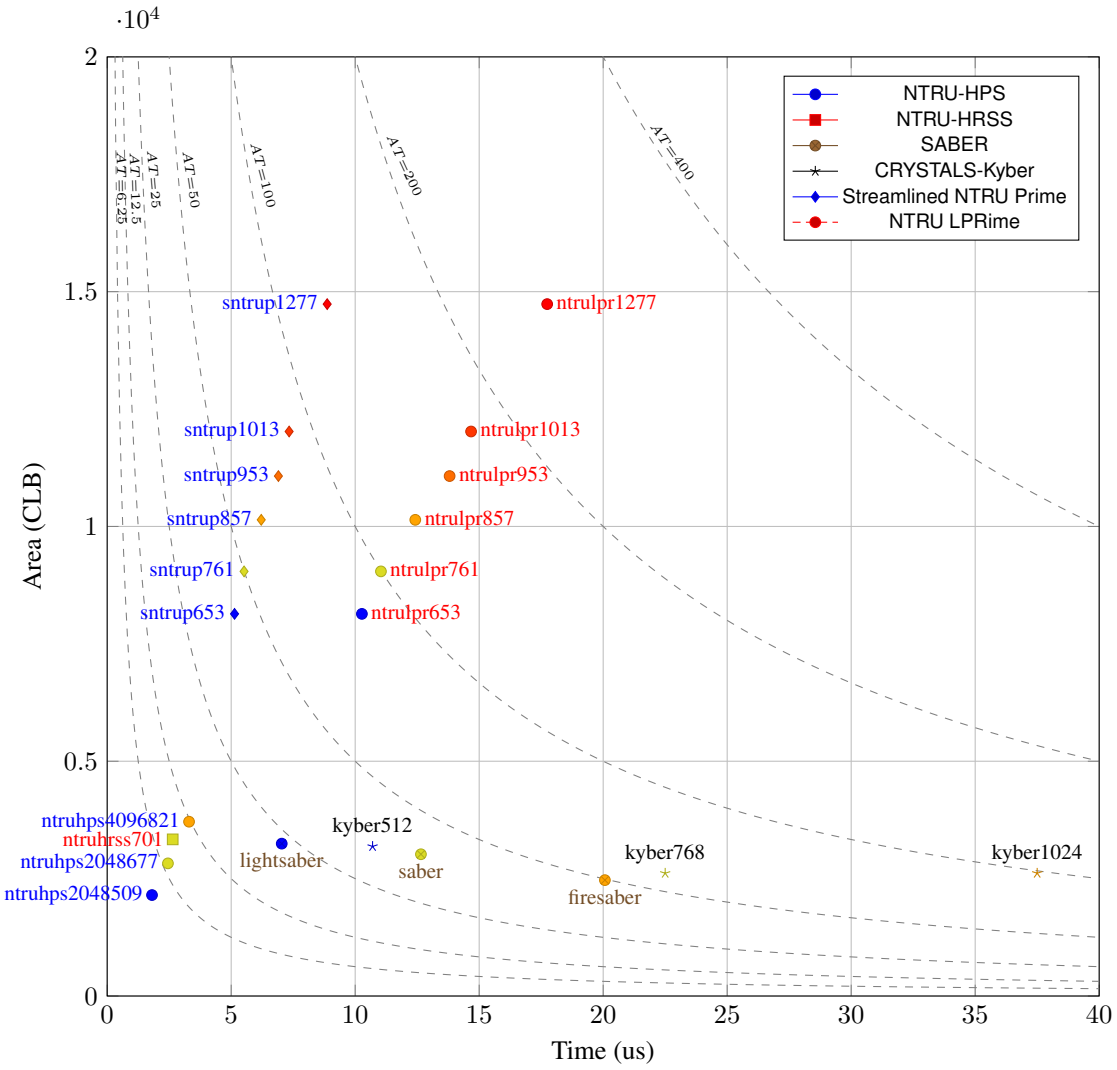


Figure 7.10: *efficiency comparison of x-net based designs. Blue, yellow and orange markers refer to parameters of security level 1, 3, and 5, respectively. Red markers denote parameters above security level 5. Dashed lines exhibit the same area-time (AT) product (lower is better).*

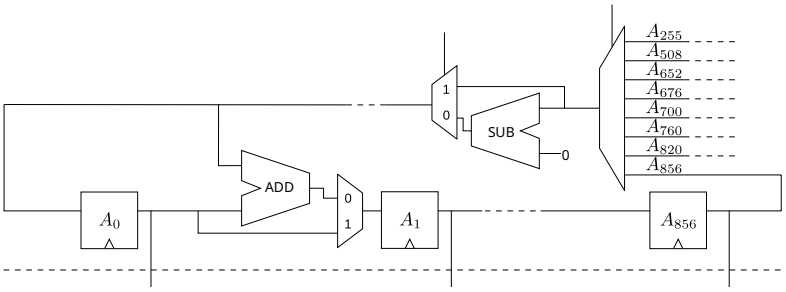


Figure 7.11: *Multiplexers introduced by the unified parallelized schoolbook multiplier*

size of the accumulator registers without any performance advantage.

Figure 7.10 allows to compare the designs of the encapsulation module of CRYSTALS-Kyber, SABER, NTRU and NTRU Prime, employing the x-net based multipliers with their most suitable reduction strategy. A blue marker denotes a design with a parameter set corresponding to NIST security level 1 (`AES-128`), a yellow marker represents security level 3 (`AES-192`), an orange marker indicates security level 5 (`AES-256`), and red markers correspond to security levels above level 5. The figure also includes dashed lines representing design space points showing equivalent AT products, which simplifies the comparison of the efficiency indicators in such chart. Figures with similar trends were also obtained for decapsulation modules and key generation modules, thus are omitted here.

By examining the designs with the same security level (markers of the same color), it is evident that the time spent on polynomial multiplications is greater in CRYSTALS-Kyber (a module RLWE scheme) and SABER (a module RLWR scheme) compared to NTRU-based schemes (the right-most values on the x axis of Figure 7.10), and such difference increases with the security level. The only exception to this trend is the key generation of NTRU-HPS and NTRU-HRSS for the parameter sets corresponding to security levels 1 and 3 due to the larger number of operations involved with respect to the other schemes. CRYSTALS-Kyber and SABER schemes show an almost constant value on the y axis of Figure 7.10 due to almost identical polynomial multiplier resulting from similar q, n, p, and $f(x)$ from the parameters sets. The latency of polynomial multiplications in NTRU Prime scales linearly with the degree n of the polynomials, and the performance degradation due to higher security margins increases at a slower rate compared to CRYSTALS-Kyber and SABER.

From Figure 7.10 it is evident that NTRU Prime has the least efficient implementations among all due to more strict security of the employed polynomial field. NTRU-based parameters exhibit a considerably lower degree of variability in terms of efficiency with benchmark points less spread than the other schemes, and is from $4\times$ to $8\times$ more efficient during encapsulations. Finally, designs for CRYSTALS-Kyber lag significantly behind the efficiency achieved with SABER, with $\approx 2\times$ worse efficiency.

Table 7.10: *Results of the synthesis targeting an Xilinx UltraScale+ ZCU106 FPGA for the unified designs compatible with the specified parameter sets. The design based on x-net is configured to transfer $4\,\mathcal{R}_p$ coefficients and $1\,\mathcal{R}_q$ coefficients per clock cycle. Each supported parameter set can be selected at runtime.*

Design	Supported ciphers	Security level	CLB	Freq.	LUT	FF	CARRY8	DSP48E2
x-net	NTRU, NTRU Prime, SABER, Kyber	AES-128	12090	272	70704	20184	2630	2
		AES-192	13935	247	83922	24237	3064	2
		AES-256	15273	241	94410	27276	3448	2
x-net	NTRU, NTRU Prime, Kyber	AES-128	8825	272	53479	18750	2624	2
		AES-192	10071	247	63718	22593	3058	2
		AES-256	11435	244	71775	25401	3442	2
Comba	NTRU, SABER Kyber	AES-256	67	328	394	142	30	0

Unified design A single unified design for all four cryptosystems, shown in Figure 7.11, was achieved by working on the following areas:

Coefficient modulo the operation $\mathrm{mod} q$ is performed during the readout phase, requiring a Barrett reduction module compatible with multiple dividend values. For this reason, the pre-computed approximated constants $1/q$ for each possible value of q are stored in small read-only memories, with the exception of $q = 2^k$ for some $k \in \mathbb{N}$ that used the regular bit trim operation for efficiency.

Accumulator the number and size of accumulation registers are carefully sized in order to fit the largest value resulting from all the supported schemes

LFSR net few multiplexers are introduced to control which a_{n-1} coefficients are fed back, depending on the n value defined by the parameters set in use, and what taps are active on the LFSR feedback network. Furthermore, and additional multiplexer determines if the selected feedback value a_{n-1} needs a sign-flip or not.

To explore potential efficiency tradeoffs, the resulting design was tested limiting the supported security levels to 1, ≤ 3, and ≤ 5.

These proposed designs offer complete runtime flexibility at the cost of approximately 50% more area resources compared to the largest tailored component. The achieved operating frequency is only from 5% to 22% slower than the slowest component it includes, with no penalty in the number of clock cycles for any of the multiplications compared to the corresponding optimized design. By removing support to SABER, the area penalty reduces to less than 12%.

Comba designs

The Comba multiplier presented in subsubsection 7.3.2 shows a remarkably small datapath because it only performs a single MAC operation between polynomial coefficients per clock cycle. The optimal memory access strategy is guaranteed due to the efficient

Table 7.11: *Results of the synthesis targeting an Xilinx UltraScale+ ZCU106 FPGA specialized for each supported parameter sets. The design is based on the Comba algorithm, and the modulo reduction operation is performed each clock cycle. Area-Time product computed as latency (ms) × CLB*

Comba algorithm | modulo each CC										
Security level	**Parameter set**	**CLB**	**kCC**	**Freq. MHz**	**Latency (µs)**			**AT product**		
					Keyg.	**Enc.**	**Dec.**	**Keyg.**	**Enc.**	**Dec.**
AES-128	ntruhps2048509	35	260.1	930	1398.4	279.6	559.3	48.9	9.7	19.5⋆
	lightsaber	36	66.1	522	506.1	759.2	1012.3	18.2	27.3	36.4
	kyber512	60	66.1	295	895.6	1343.4	1791.2	53.7	80.6	107.4
AES-192	ntruhps2048677	37	459.7	906	2536.9	507.3	1014.7	93.8	18.7	37.5⋆
	ntruhrss701	46	492.8	883	2790.5	558.1	1116.2	128.3	25.6	51.3⋆
	saber	39	66.1	522	1138.8	1518.4	1898.1	44.4	59.2	74.0
	kyber768	56	66.1	319	1863.5	2484.7	3105.9	104.3	139.1	173.9
AES-256	ntruhps4096821	43	675.7	845	3998.1	799.6	1599.2	171.9	34.3	68.7⋆
	firesaber	33	66.1	667	1584.5	1980.6	2376.7	52.2	65.3	78.4
	kyber1024	56	66.1	319	3313.0	4141.3	4969.5	185.5	231.9	278.2

use of an accumulator register t. An additional modulo q component is necessary to deal with the modular arithmetic in the coefficient ring $\mathbb{Z}_q$. As in the case for the x-net multiplier, it can be omitted in case the value of q is a power of two due to the bit trim operation performing it for free.

Performing a regular polynomial multiplication in $\mathbb{Z}_q[x]$ would produce a result with a maximum degree up to $2n-1$, and a complex polynomial modulo operation is later required. This can be avoided computing a modular addition or subtraction of each outer loop iteration results of lines 6–9 of Algorithm 33 with specific monomial of the temporary result. However, the NTRU Prime cryptographic scheme requires two additions per iteration due to the polynomial modulus giving the equation $x^n = x + 1$. Therefore, to maintain the design as compact and as efficient as possible, the support to the NTRU Prime parameters sets are dropped.

Since each coefficient multiplication in the $\mathbf{R}_p \times \mathbf{R}_q \mapsto \mathbf{R}_q$ polynomial operation can produce a relatively small maximum value of $p \cdot q - 1$, the $\mathrm{mod} q$ operation can be performed through a selection of the results of a short chain of adders and subtractors even when the modulus q is not a power-of-two.

Considering the results of the synthesis campaign of the Comba-based multipliers specialized for each parameters set of all the supported cryptoschemes, the design shows a remarkably small area requiring almost two orders of magnitude less CLBs with respect to the x-net design, leading to an operating frequency improved up to 36% when NTRU parameters sets are used.

The efficiency analysis depicted in Figure 7.12 confirms similar trends than Figure 7.10, with CRYSTALS-Kyber being the slowest cryptoscheme, NTRU the fastest and most efficient one for encapsulation and decapsulation procedures, and SABER is showing the same results for the key generation. Note that such extremely compact and slow solution still has interesting efficiency figures while requiring almost two order of

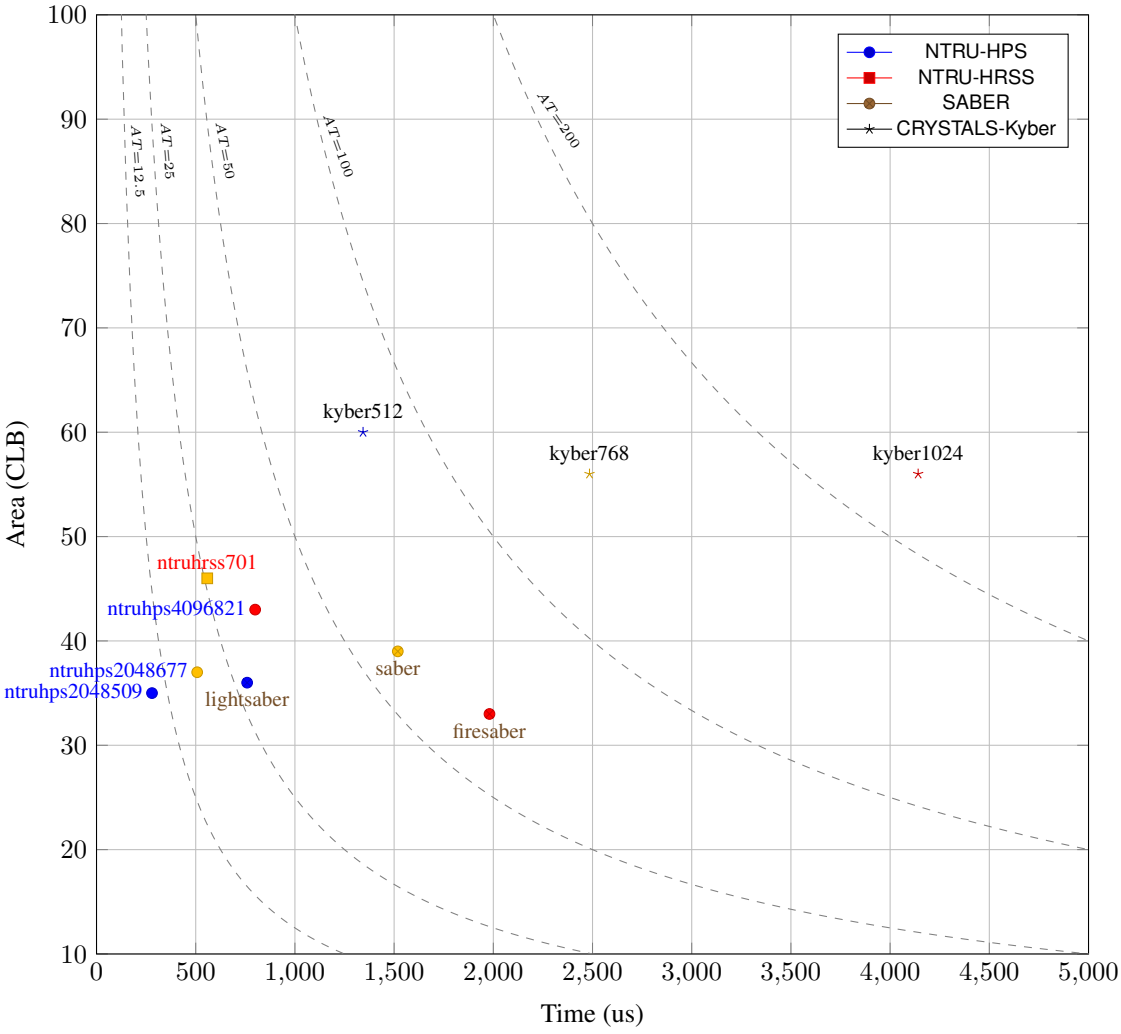


Figure 7.12: *efficiency comparison of Comba-based designs. Blue, yellow and orange markers refer to parameters of security level 1, 3, and 5, respectively. Red markers denote parameters above security level 5. Dashed lines exhibit the same area-time (AT) product (lower is better).*

magnitude less area than the low-latency solution based on x-net. Therefore, when a slow multiplier completing its task in few milliseconds is an acceptable solution, this compact design can be an appealing solution.

Ultimately, this multiplier requires only 10% more area compared to the largest tailored component, in this case resulting from the support of the `kyber512` parameters set, while does not exhibit an operating frequency degradation.

The complete dataset of this DSE is available at DOI:10.5281/zenodo.8337625.

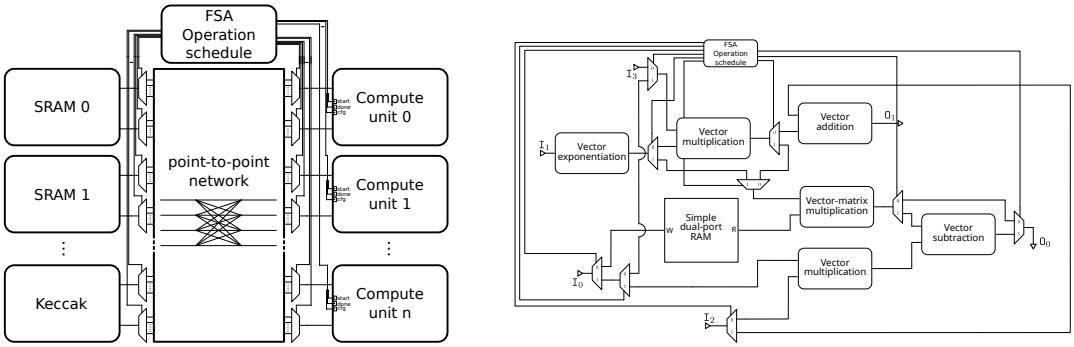
CHAPTER 8

Top-level design

Once all basic components of a cryptographic scheme are designed and validated, the final step consists in the integration of those components in a Top-Level Design (TLD) along with some RAMs and a secure TRNG, designing their interconnection, and creating the FSM logic implementing a schedule of the operations to produce a valid cryptographic primitive. Then, the TLD must be validated against test vectors from the KAT file distributed along with the specification of the cryptographic scheme, usually directly derived by the reference SW code. During this validation task, the source of randomness is not derived directly from a TRNG, but rather uses the seed contained in the KAT file, hence having a deterministic and reproducible behavior.

The ultimate goal of this thesis is to produce several reference designs for post-quantum asymmetric cryptographic algorithms which are easily portable to different targets and are functionally correct. Even though these designs are compliant with the official specification, they must be considered solely for research purposes for several reasons. Firstly, the specifications of the cryptographic algorithms are not frozen and certified by a standardization body, and it is imperative that all the parties involved in the cryptographic protocol use exactly the same version of the algorithm for a guaranteed compatibility and security assurance. Secondly, in the developed TLDs the TRNG component is missing, and the randomness seed are retrieved from the internal memory for testing purposes using the KATs. The reason is that the choice of a secure TRNG intrinsically depends on the target device implementing the design, such as the EDA

Part of the material presented in this chapter was originally described in [Ant+23a; Ant+24a; Ant+24b; ABP25].



(a) *Units interconnected via a point-to-point network*

(b) *Arithmetic modules interconnected in a partially reprogrammable chain*

Figure 8.1: *Overview of two strategies for modules instantiation in the top-level design. A centralized Finite State Automata (FSA) manages the control signals following a specific schedule of algorithm operations.*

tools employed, the product family and production revision of the FPGA chip, or the technology library, the fabrication process flow for ASIC chips, and the Process, Voltage, and Temperature (PVT) conditions. Furthermore, in addition to the obstruction of portability, a secure TRNG must be validated and certified by passing the statistical test suite for the validation of RNGs for cryptographic applications, such as [Bas+10]. A systematic review of tools and technologies employed for the assessing of RNGs is reported in [Cro+23]. Lastly, the developed solutions do not implement any countermeasure against invasive or non-invasive side-channel attacks other than timing attacks, which is achieved through the constant-time execution of algorithms with the exclusion of rejection sampling mechanisms well known not to be a source of time leakage. The security requirements, tests, and approved mitigations of non-invasive attacks for cryptographic modules are extensively normed by the NIST standard *FIPS 140-3* [JC19] and ISO/IEC standards *17825:2024* [ISO24], *19790:2025* [ISO25a], and *24759:2025* [ISO25b], identifying 11 different security areas and 4 levels of security assurance. Every cryptographic modules operated by U.S. federal departments and agencies must comply with the *FIPS 140-3* standard, but this certification is also required by several other national agencies and companies, thus is recognized as a globally relevant standard.

An important design choice to be decided during the integration of the basic components is the selection of how to interconnect the sub-modules. Generally speaking, two different strategies can be devised, as represented in Figure 8.1: either directly connect each module to the memories holding the operand values and storing the computed result, or chain them in a configurable pipe structure specifically tailored to the sequence of operations defined in the cryptoscheme. The former solution requires an interconnection network that allows a high degree of flexibility in connecting the hardware components, but also has a non-negligible cost both in terms of area and routing congestion. An example of such design is depicted in Figure 8.1a. This solution is par-

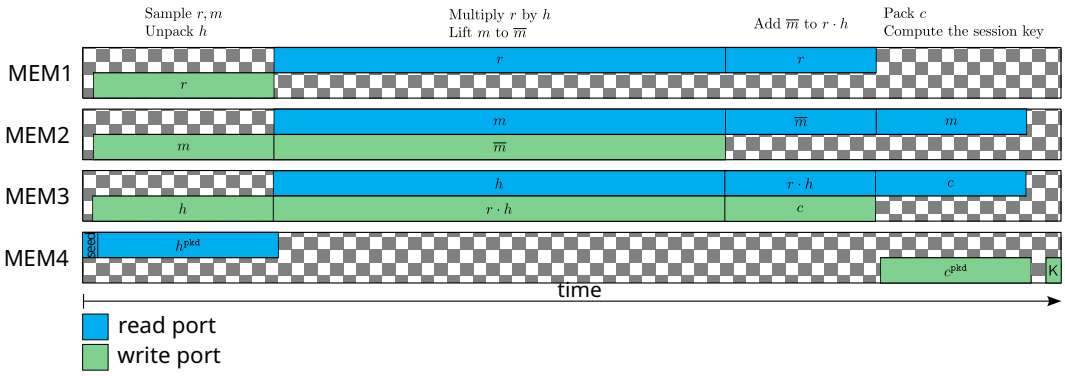


Figure 8.2: *Simplified example of a memory port binding analysis exclusively assigning a read or write port of a simple dual-port SRAM to let a computing module access an operand or write a result in a memory. This information is used by the FSA in the top-level design to configure the point-to-point connection between the memory and a computing module. Note that if more advanced contention protocols are used, such as a round-robin queue or the broadcast of shared request, this analysis must consider the newly introduced constraints.*

ticularly interesting when conducting a DSE in order to easily swap each component, minimizing the compatibility issues. The other solution requires a thorough analysis of the sequence of operations to define the structure using the least amount of multiplexers, and benefits from the partially overlapped execution of the units composing the chain. However, even small changes to the cryptoscheme specification may require the redesign of the piping connections. Moreover, incompatibilities between the units composing the chain may arise when there are strict constraints on the transmitted data, such as the absence of transfer stalls, or the shape of the interconnection. It is possible to adopt a hybrid approach between the two solutions, for example chaining only the arithmetic operations which are less prone to updates in the documentations, and leaving to a simplified interconnection network the management of point-to-point connections of the hardware modules. In Figure 8.1b is represented an example of a partially re-wireable chain connecting some arithmetic components, where $\texttt{I}_0$, $\texttt{I}_1$, $\texttt{I}_2$, $\texttt{I}_3$, $\texttt{O}_0$, and $\texttt{O}_1$ are the input/output connections of the arithmetic module.

Both integration strategies require a FSA driving the multiplexers connecting the hardware components following a schedule that reflects the data dependencies of the operations in the cryptoscheme to correctly compute the results, and satisfying the constraints imposed by the hardware resources. An example of such constraint is given by the unique binding of a memory port to a specific computing module. Considering that the number of ports offered by a memory has an increasing area and performance cost, and often is limited by the availability of the required SRAM macro, the FSA must consider such constraints when trying to schedule the execution of two or more parallel operations. An example of the port binding in a system composed of four simple dual-port RAMs is depicted in Figure 8.2. Fortunately the order of operations in KEM and DS primitives are fixed, and such schedule can be pre-computed. The size

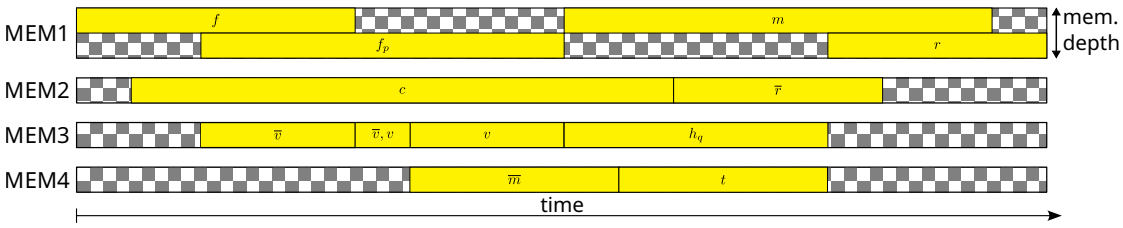


Figure 8.3: *Simplified example of variable liveness analysis throughout the execution of the cryptographic primitive having access to four memories. The yellow block corresponds to a variable starting when the element is generated and terminating after its last access. The complete analysis must also consider elements with different sizes, keeping track of their base memory addresses, which is then used by the FSA in the top-level design.*

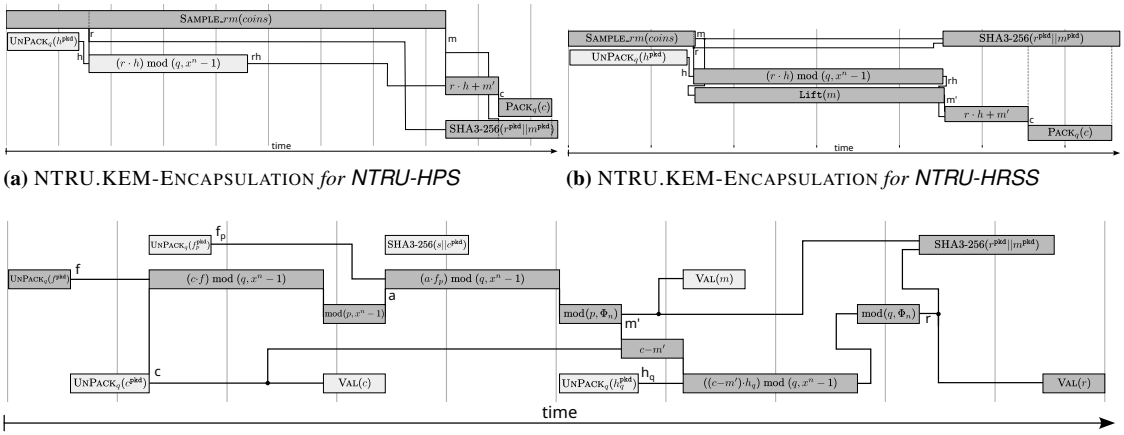
of centralized memories depends on how many and which elements needs to persist in memory at the same time. The optimal size can be determined statically by a process known as *liveness analysis* thanks to the known order of operations provided by each cryptoscheme specification. A small example is provided in Figure 8.3.

8.1 NTRU

A desired aspect of the designed hardware accelerator for the NTRU cryptoscheme consists in having all the computing modules highly decoupled in order to easily replace each component to explore and evaluate different algorithms and implementation strategies. Moreover, different strategies may require to access data with different word sizes. Such flexibility is provided by the point-to-point interconnection network linking computing modules to the centralized memories and shared components, such as the SHAKE module.

Two main components are necessary to correctly implement the network: a port arbiter unit, and a link width converter. The former handles the access to a shared source or sink of data, and in its simplest form consists in a set of multiplexers to establish a one-to-one link coherently to the transfer protocol in use depending on a selection signal. Note that more sophisticated control logic, such as a round-robin based approach or broadcasting the data to links making the same request, may allow the parallel scheduling of some operations. The link width converter manages the access to the data with different granularity either in a stateless or stateful manner via some buffers, to allow a transparent access to resources, at cost of requiring memories supporting byte-enable writes to update small parts of the data words contained in the memory.

The main developed components are the packer and unpacker of polynomials with coefficients in $\mathbb{Z}_p$ and $\mathbb{Z}_q$ (PACK$_p$, UNPACK$_p$, PACK$_q$, and UNPACK$_q$), the polynomial generator CSPRNG creating ternary polynomials in $\mathcal{T}$, $\mathcal{T}_+$, or having w fixed weight $\mathcal{T}(w)$, the polynomial multiplier and adder, and the lift (LIFT) and embed ($\mathrm{mod}(\cdot,\cdot)$) of polynomials in polynomial rings. The top-level design makes use of 4 main simple dual-port RAMs, for which one port is a read only memory port and the other is a write



(a) NTRU.KEM-ENCAPSULATION *for* NTRU-HPS

(b) NTRU.KEM-ENCAPSULATION *for* NTRU-HRSS

(c) NTRU.KEM-DECAPSULATION *for* NTRU-HPS *and* NTRU-HRSS

Figure 8.4: *Schedule of the operations for the* NTRU.KEM *scheme*

only memory port. The memories have 64-bits word size, with a 16-bits write enable chunks to support the modification of small portions of the words via the link width converters.

8.1.1 Operation scheduling

As a case study, this thesis considers the use of the NTRU scheme with long-term keypairs, thus producing hardware accelerators for NTRU.KEM-ENCAPSULATION and NTRU.KEM-DECAPSULATION algorithms only, and having the keypair pre-generated in software and securely stored in a memory accessible by the hardware components [Che+19]. Two design targets are provided, a high efficiency and low latency design using the x-net polynomial multiplier and leveraging a higher degree of parallelism for the other computing modules, or a low area solution using the Comba multiplier without leveraging any parallelism in the other modules.

The scheduling of the operations included in the NTRU.KEM-ENCAPSULATION and NTRU.KEM-DECAPSULATION primitives, shown in Figure 8.4, depends on the algorithm implementing the LIFT map, and the SAMPLE$_{rm}$ function generating the elements r and m prescribed by NTRU-HPS and NTRU-HRSS schemes.

Indeed, the NTRU-HPS requires to sample m from the space of ternary polynomials having fixed Hamming weight $\mathcal{T}(q/8 - 2)$, for which the sampling algorithms take more time compared to the sampling algorithms used by NTRU-HRSS generating them from the set of ternary polynomials of any weight. The specification mandates to sample first r and then m, thus as soon as r is retrieved and h is decoded from its compressed binary string form, the $r \cdot h$ multiplication can take place. In this way, the latency of this polynomial multiplication is hidden by the latency of sampling the polynomial m for NTRU-HPS if using the x-net multiplication architecture, as pictured in Figure 8.4a. The LIFT map in this scheme is a mere sign extension of the coefficients in the ring $\mathbb{Z}_q$,

which is implicitly performed during the final addition.

By contrast, in NTRU-HRSS the sampling of m does not represent a bottleneck, but the LIFT does require a dedicated module for its computation. We hide such computation scheduling it in parallel to the $r \cdot h$ multiplication, as represented in Figure 8.4b, thanks to the developed unit not relying on the polynomial multiplier to compute the result.

The generation of the session key K is performed immediately absorbing in the SHA3-256 module the bit string representation of r and m as soon as they are generated, and runs in parallel to the computation of the ciphertext c. Overall, the devised NTRU.KEM-Encapsulation schedule manages to have the SHA3-256/SHAKE256 module represent a significant portion of the overall computation of around 20% in NTRU-HPS and 30% in NTRU-HRSS.

Scheduling of operations in the NTRU.KEM-DECAPSULATION primitive is less influenced by the differences between NTRU-HPS and NTRU-HRSS, as there is no need to sample random ternary polynomials. The only difference relies on the implementation of the LIFT function that this time cannot be scheduled in parallel to other operations and is scheduled right before the execution of the last multiplication.

The longest read-after-write sequence of operations in the proposed schedules is highlighted with darker gray boxes in all the subfigures of Figure 8.4, and in case of Figure 8.4c it is clear that most of the operations are in this long read-after-write dependency chain. As a result, we scheduled the operations on this path to be executed as soon as possible, and parallelized all the others, preemptively starting unpacking and validation computations where possible. The only computation which cannot be parallelized with the final SHA3-256 computation is the one acting on r, as the SHA3-256 module is accessing the same memory at the same time, potentially with a different width. Since a $\mathbf{R}_q \times \mathbf{R}_q \mapsto \mathbf{R}_q$ multiplication takes place, the multiplier component in the decapsulation top-level module cannot apply the optimization leveraging the small size of the coefficients in $\mathbf{R}_p$.

8.1.2 Design synthesis and implementation

A DSE for the encapsulation and decapsulation primitives is conducted on a AMD Zynq UltraScale+ FPGA platform, varying the word size that each computing module is using to access the centralized simple dual-port RAMs, and consequently the degree of parallelization used in the implemented algorithms. The choice of an UltraScale+ FPGA platform in place of the usual AMD Artix-7 FPGA which is the reference platform selected by NIST, is due to the large number of DSP units required by the decapsulation design that could not fit in any Artix-7 chip. The minimum memory word size is set to 16 to allow the inference of RAMs having support to the byte-enable writes in FPGA targets, although synthesis results demonstrated that it was not achieved, resulting in a higher than expected BRAM count, with a consequent efficiency degradation. Only 4 independent RAMs are used in both encapsulation and decapsulation designs, with a word size up to 64 bits long. Among all the configurations, we selected for a synthe-

sis assessment only the ones which dominated their alternatives in RTL simulations in terms of latency, which led to the synthesis of 9 NTRU.KEM-ENCAPSULATION and 4 NTRU.KEM-DECAPSULATION configurations for each parameters set, reported in Table 8.1.

The target frequency was set to 400 MHz for the encapsulation, and 350 MHz for the decapsulation, with the small difference caused by routing congestion. In both cases, the critical path involved the Keccak core in the SHAKE module.

8.2a and 8.2b report the best speed oriented designs found by this exploration, highlighted in gray in 8.1a and 8.1b. The NTRU-HRSS variant is more efficient (lower AT product) than its NTRU-HPS counterpart, saving between 50% in latency at an increased $\approx 4\%$ area cost. The compact solutions using the Comba multiplier are able to achieve a 66% area reduction, but are also significantly less efficient than the designs using the x-net multiplier. However, approximately 80% of the area is constituted by the SHA-3 module, which constitutes a hard barrier to further area savings.

Building upon the outcomes of the design space exploration conducted on FPGA, we selected the configuration that exhibited the lowest latency and the most compact area. These chosen designs were then synthesized for an ASIC implementation using the *Synopsys Design Compiler Ultra* 2022.03-SP1 tool, and leveraging a high-density 40 nm industrial-grade technology library. Memories are excluded from the logic synthesis by removing them and routing the memory buses through the input/output pins. 8.3a and 8.3b report the results of the ASIC synthesis. Using the slow process corner using a 1.15V core voltage at 40°C, the architecture incorporating the Comba multiplier attains an operational frequency of 750 MHz for both encapsulation and decapsulation processes, and the design utilizing the x-net multiplier achieves 700 MHz for encapsulation and 650 MHz for decapsulation.

In the x-net based designs for encapsulation and decapsulation, the primary contributors to area requirements are the $\mathcal{R}_p \times \mathcal{R}_q$ and $\mathcal{R}_q \times \mathcal{R}_q$ multipliers, respectively. Notably, in the encapsulation module, these multipliers are between $2.3\times$ and $4.1\times$ larger than the Keccak module, making their resource usage comparable. Additionally, an $\mathcal{R}_q \times \mathcal{R}_q$ multiplier occupies only three times the area of an $\mathcal{R}_p \times \mathcal{R}_q$ multiplier. It is also worth noting that the decapsulation module for `ntruhrss701` demands more area than that of `ntruhps4096821`, even though the latter offers a higher security margin. This difference arises from the larger value of q used in `ntruhrss701`, which consequently increases the area requirements for the $\mathcal{R}_q \times \mathcal{R}_q$ multiplier. Considering the operating frequencies of the ASIC implementations, the NTRU.KEM-ENCAPSULATION operation is executed within a range of 6.2μs to 10.2μs for NTRU-HPS, depending on the security margin, and in 4.1μs for NTRU-HRSS. Similarly, NTRU.KEM-Decapsulation is completed in 7.1μs to 11.5μs for NTRU-HPS and in 11.7μs for NTRU-HRSS. Moreover, the Comba-based design occupies an area between 49.91 and $52.43 \cdot 10^3$ µm^2, with approximately $41 \cdot 10^3$ µm^2 allocated to the Keccak module. As a result, the NTRU.KEM modules require only approximately 20% more area compared to an existing SHA3-256/SHAKE256 accelerator. Meanwhile, the compact designs achieve

Table 8.1: *Design space exploration for the* NTRU.KEM *varying the multiplier architecture (Comba, x-net), the sampler algorithm (Modulo, Rejection), and the parallelized coefficient transfers (PT) for first multiplier operand (op1) and result (res), adder, session key generator (SKG), validator (Val.)*

(a) NTRU.KEM-ENCAPSULATION, 400 *MHz clock target, the first multiplication operand is in* $\mathbf{R}_p$*, AT: eSlice* $\times$ *ms*

NTRU variant	$\mathbf{R}_p \times \mathbf{R}_q$ mul. arch.	op1	res	Sampler alg.	Sampler PT	Add PT	SKG PT	Lift PT	Latency CC	Latency μs	Area LUT	Area FF	Area BRAM	Area eSlice	AT prod.
ntruhps2048509	x-net	1	1	M	2	1	4	1	4765	11.91	18186	11545	4.0	5395	64.2
	x-net	1	1	R	2	1	4	1	4787	11.97	18127	11490	4.0	5380	64.3
	x-net	4	2	M	2	2	4	2	4511	11.28	19229	11587	5.0	5868	66.1
	x-net	4	2	R	2	2	4	2	4533	11.33	19152	11524	5.0	5848	66.2
	x-net	8	4	M	4	4	4	4	4384	10.96	19379	11663	8.5	6647	72.8
	x-net	8	4	R	4	4	4	4	4394	10.98	19333	11552	8.5	6636	72.8
	x-net	8	4	M	4	8	4	4	4320	10.80	19462	11715	12.5	7516	81.1
	x-net	8	4	R	4	8	4	4	4330	10.82	19212	11597	12.5	7453	80.6
	Comba	1	1	R	1	1	1	1	261950	654.88	7750	4805	1.5	2256	1477
hps2048677	x-net	1	1	M	2	1	4	1	6435	16.09	23215	13772	4.0	6652	107
	x-net	1	1	R	2	1	4	1	6465	16.16	23161	13716	4.0	6639	107
	x-net	4	2	M	2	2	4	2	6097	15.24	23357	13824	5.0	6900	105
	x-net	4	2	R	2	2	4	2	6127	15.32	23543	13805	5.0	6946	106
	x-net	8	4	M	4	4	4	4	5928	14.82	24664	13902	8.5	7968	118
	x-net	8	4	R	4	4	4	4	5947	14.87	24562	13760	8.5	7943	118
	x-net	8	4	M	4	8	4	4	5843	14.61	24817	13923	12.5	8855	129
	x-net	8	4	R	4	8	4	4	5862	14.65	24387	13817	12.5	8747	128
	Comba	1	1	R	1	1	1	1	462079	1155.20	8309	4828	1.5	2396	2767
ntruhps4096821	x-net	1	1	M	2	1	4	1	7796	19.49	27850	16496	4.5	7917	154
	x-net	1	1	R	2	1	4	1	7780	19.45	27841	16449	4.5	7915	153
	x-net	4	2	M	2	2	4	2	7386	18.46	29543	16582	5.5	8552	157
	x-net	4	2	R	2	2	4	2	7370	18.43	29188	16462	5.5	8463	155
	x-net	8	4	M	4	4	4	4	7181	17.95	29637	16634	9.0	9318	167
	x-net	8	4	R	4	4	4	4	7171	17.93	29565	16511	9.0	9300	166
	x-net	8	4	M	4	8	4	4	7078	17.70	29806	16667	13.0	10208	180
	x-net	8	4	R	4	8	4	4	7068	17.67	29631	16555	13.0	10164	179
	Comba	1	1	R	1	1	1	1	678596	1696.49	8140	4821	2.0	2459	4171
ntruhrss701	x-net	1	1	R	2	1	4	1	4542	11.36	26255	15534	6.5	7942	90.2
	x-net	1	1	M	2	1	4	1	4542	11.36	26350	15653	6.5	7966	90.4
	x-net	4	2	M	2	2	4	2	3317	8.29	27881	15690	6.5	8349	69.2
	x-net	4	2	R	2	2	4	2	3317	8.29	27731	15634	6.5	8311	68.8
	x-net	8	4	M	4	4	4	4	2879	7.20	28561	16005	9.0	9049	65.1
	x-net	8	4	R	4	4	4	4	2879	7.20	27898	15743	9.0	8883	63.9
	x-net	8	4	M	4	8	4	4	2791	6.98	28563	15934	13.0	9897	69.0
	x-net	8	4	R	4	8	4	4	2791	6.98	28041	15689	13.0	9767	68.1
	Comba	1	1	R	1	1	1	1	495312	1238.28	8112	4917	2.0	2452	3036

(b) NTRU.KEM-DECAPSULATION, 350 *MHz clock target, both operands of the multiplier are* $\mathbf{R}_q$ *elements, AT: eSlice* $\times$ *ms*

NTRU variant	$\mathbf{R}_q \times \mathbf{R}_q$ mul. arch.	op1	res	Add PT	SKG PT	Val. PT	Lift PT	Latency CC	Latency µs	Area LUT	Area FF	Area DSP	Area BRAM	Area eSlice	AT prod.
ntruhps2048509	x-net	1	1	1	1	1	1	8110	23.17	14666	16776	509	2.0	72551	1681
	x-net	2	2	2	2	2	2	5822	16.63	19261	16209	509	3.0	73912	1229
	x-net	4	4	4	4	4	4	4678	13.37	20051	17379	509	5.5	74640	997
	Comba	1	1	1	1	1	1	785351	2243.75	8331	4641	1	2.0	2642	5927
ntruhps2048677	x-net	1	1	1	1	1	1	10729	30.65	15516	20648	677	2.5	95466	2926
	x-net	2	2	2	2	2	2	7686	21.96	22689	20059	677	3.5	97471	2140
	x-net	4	4	4	4	4	4	6163	17.61	23689	21286	677	6.0	98251	1730
	Comba	1	1	1	1	1	1	1385714	3958.98	8446	4705	1	2.5	2776	10990
ntruhps4096821	x-net	1	1	1	1	1	1	13061	37.32	18348	25482	821	4.0	115860	4323
	x-net	2	2	2	2	2	2	9366	26.76	28504	25953	821	5.0	118611	3174
	x-net	4	4	4	4	4	4	7521	21.49	29074	26474	821	6.0	118965	2556
	Comba	1	1	1	1	1	1	2035182	5814.51	8347	4668	1	4.0	3070	17850
ntruhrss701	x-net	1	1	1	1	1	1	13351	38.14	17972	24300	701	2.5	99308	3787
	x-net	2	2	2	2	2	2	9514	27.18	27343	24468	701	3.5	101863	2768
	x-net	4	4	4	4	4	4	7606	21.73	27790	24979	701	6.0	102504	2227
	Comba	1	1	1	1	1	1	1487552	4249.94	8526	4813	1	2.5	2796	11882

Table 8.2: *Results of the synthesis exploration for the* NTRU.KEM *primitive with speed objective on FPGA. AT metric computed as eSlice × ms.*

(a) NTRU.KEM-ENCAPSULATION

Sec. lvl.	NTRU Variant	Work	Freq.	Area					Latency		AT
				LUT	FF	DSP	BRAM	eSlice	CC	µs	prod.
1	hps2048509	This work	400	19379	11663	0	8.5	6647	4384	10.9	72.4
3	hps2048677	This work	400	24664	13902	0	8.5	7968	5928	14.8	117
		[DMG21]	250	26325	17568	0	5.0	7642	3687	14.8	113
	hrss701	This work	400	28396	15894	0	9.0	9007	2879	7.2	64.8
		[DMG21]	300	31494	25120	0	2.5	8404	2219	7.4	62.1
	sntrup761	[Pen+21]	289	31996	22425	6	4.5	9760	5007	17.3	168
5	hps4096821	This work	400	29637	16634	0	9.0	9318	7181	17.9	166
		[DMG21]	250	33698	30551	0	5.5	9591	4576	18.3	175

(b) NTRU.KEM-DECAPSULATION

Sec. lvl.	NTRU Variant	Work	Freq.	Area					Latency		AT
				LUT	FF	DSP	BRAM	eSlice	CC	µs	prod.
1	hps2048509	This work	350	20051	17379	509	5.5	74640	4678	13.3	992
3	hps2048677	This work	350	23689	21286	677	6.0	98251	6163	17.6	1729
		[DMG21]	300	29935	19511	45	2.5	14067	7522	25.1	353
	hrss701	This work	350	27790	24979	701	6.0	102504	7606	21.7	2224
		[DMG21]	300	37702	34441	45	2.5	16008	8826	29.4	470
	sntrup761	[Pen+21]	285	32301	22724	9	3.5	10028	10989	38.6	387
5	hps4096821	This work	350	29074	26474	821	6.0	118965	7521	21.4	2545
		[DMG21]	300	38642	33003	45	2.5	16243	10211	34.0	552

Table 8.3: *Results of the ASIC synthesis for the* NTRU.KEM *primitive reached* 750 *and* 650 *MHz for area constrained (Comba) and fast (x-net) designs, respectively, when using the slow process corner of* 1.15*V at* 40°*C from a* 40 *nm industrial-grade technology library.*

(a) NTRU.KEM-ENCAPSULATION

Mul. type	NTRU Variant	Area (10^3 µm^2) add	sample	Keccak	q pack	K gen	$\mathbf{R}_p \times \mathbf{R}_q$	q unp.	lift	Total	Latency µs
x-net	hps2048509	0.66	2.76	39.12	1.12	2.64	90.40	1.41	–	140.19	6.2
	hps2048677	0.68	2.97	39.16	1.11	2.67	120.44	1.41	–	170.44	8.4
	hps4096821	0.73	2.95	39.28	0.82	2.68	161.21	1.07	–	211.05	10.2
	hrss701	0.83	1.36	40.24	1.26	2.67	148.91	1.54	1.87	201.15	4.1
Com.	hps2048509	0.39	2.92	41.56	1.14	1.63	1.22	1.39	–	51.52	349.2
	hps2048677	0.42	3.01	40.06	1.13	1.65	1.30	1.40	–	50.30	616.1
	hps4096821	0.44	3.00	40.29	0.82	1.67	1.31	1.07	–	49.91	904.7
	hrss701	0.47	2.40	40.96	1.35	1.68	1.36	1.52	1.22	52.43	660.4

(b) NTRU.KEM-DECAPSULATION

Mul. type	NTRU Variant	Area (10^3 µm^2) add	Keccak	K_1 gen.	K_2 gen.	$\mathbf{R}_q \times \mathbf{R}_q$ mult.	unpack p	unpack q	validat.	lift	Total	Latency µs
x-net	hps2048509	0.78	40.60	0.68	2.67	268.95	1.02	1.54	0.21	–	320.06	7.1
	hps2048677	0.81	40.10	0.72	2.70	359.13	1.06	1.52	0.26	–	410.03	9.4
	hps4096821	0.81	38.96	0.71	2.66	495.69	1.07	1.17	0.28	–	517.80	11.5
	hrss701	0.91	40.71	0.72	2.71	507.23	1.05	1.66	0.25	1.72	561.02	11.7
Com.	hps2048509	0.42	41.46	0.67	1.64	1.59	1.08	1.51	0.31	–	51.45	1047.1
	hps2048677	0.45	40.70	0.71	1.68	2.38	1.05	1.48	0.29	–	50.74	1847.6
	hps4096821	0.46	40.14	0.72	1.68	2.49	1.07	1.17	0.30	–	50.08	2713.5
	hrss701	0.50	40.33	0.71	1.69	2.66	1.09	1.63	0.31	1.14	52.26	1983.4

encapsulation within a range of 0.34 to 0.9 ms and perform decapsulation in 1.04 to 2.71 ms.

8.2 HQC

Other than the polynomial multiplier and adder, the sampler of dense and sparse polynomials, and the SHAKE256/SHA3-256 unit, HQC needs the encode and decode vectors using an ECC. In the following subsection are presented the designs of the algebraic encoders and decoders for the RM/RS concatenated code, before introducing to the scheduling of the operations orchestrated by the FSA in the top-level design.

8.2.1 Encoders and decoders for Reed-Solomon and Reed-Muller codes

The HQC scheme, introduced in section 4.2, uses two public Error-Correcting Codes (ECCs) for different purposes, the first one being a random instance of a $[2p, p]$ quasi-cyclic code, and the second one being a highly correcting code fixed in the specification that is easy to decode. As the former one does not have an efficient decoder algorithm, an eavesdropper analyzing the syndromes $\mathbf{h}$ and $\mathbf{u}$ publicly exchanged during the protocol is not able to retrieve the low-weight errors encoding the secret messages transferred. Only the owner of the private key is able to remove most of the corruption added by the actor sending the first message (see Equation 4.8), and correct the remaining low-weight error $\mathbf{e}'$ using the decoding algorithm for the fixed ECC.

The $[n, k]$ fixed code is selected such that the length $n \approx p$, and that the message $\mathbf{m} \in \mathbb{F}_2^k$. Initially the specification of HQC used the tensor product of a BCH code with a repetition code due to the vast choice of parameters satisfying the aforementioned constraints, its encoding and decoding simplicity, and allowing a precise DFR estimation. Later on, the same authors proposed in [Agu+24b] a concatenated code composed by a shortened Reed-Solomon (RS) as the external code, and a duplicated Reed-Muller (RM) as the internal one, that shows a better error-correcting capability. As a consequence, this new code allowed to reduce the size of the polynomial ring p speeding-up the HQC arithmetic operations, and the latest specification adopted the Reed-Muller/Reed-Solomon (RM/RS) concatenated code. The fixed Reed-Muller/Reed-Solomon (RM/RS) concatenated code $[n_e n_i, k_e k_i, d_e d_i]$ is formed by a shortened Reed-Solomon (RS) $[n_e, k_e, d_e]$ external code, and a duplicated Reed-Muller (RM) $[n_i, k_i, d_i]$ internal code, identified by the generator matrix $\mathbf{G}$.

In 1965 Forney proved in [For65] that the concatenation of ECC with short lengths to form one with a longer length generates a code with an exponentially improved error-correcting capability at cost of a polynomial-time increase of the complexity of the decoder algorithm. For that reason, concatenated codes were quickly adopted in space communications in the 1970s, and is still used today for satellite communication, for example in the DVB-S standard.

As the internal and external codes in a concatenated code are independent, we will consider each component separately, detailing their encoding and decoding procedures in the remaining part of the subsection. We will also report some hardware design that is produced to efficiently implement such operations.

Shortened Reed-Solomon code

The HQC specification defines three different $[n_e, k_e, d_e]$ RS codes, one for each NIST security level, that are treating each 8-bits block of data as an element of $\mathbb{F}_{2^8}$ called symbol, and able to correct $t = \frac{d_e-1}{2}$ erroneous symbols: $\texttt{RS-1} = [255, 225, 31]$, $\texttt{RS-2} = [255, 223, 33]$, and $\texttt{RS-3} = [255, 197, 59]$. A shortened variant of these codes are actually used in the HQC primitives, and are simply obtained by fixing 209, 199, or 165 message symbols to 0, respectively, avoiding their transmission in the codeword: $\texttt{RS-S1} = [46, 16, 31]$, $\texttt{RS-S2} = [56, 24, 33]$, and $\texttt{RS-S3} = [90, 32, 59]$.

Each symbol is a 8-bits block of data that is interpreted as an element of $\mathbb{F}_{2^8}$ represented employing the irreducible polynomial $y^8+y^4+y^3+y^2+1 \in \mathbb{F}_2[y]$. In particular, the message to be encoded is used to build a *message polynomial* $u(x) \in \mathbb{F}_{2^8}[x]$, with degree $k_e - 1$. An n_e-symbol codeword $c(x) = u(x)g(x) \in \mathbb{F}_{2^8}[x]$ can be simply obtained multiplying the input message polynomial $u(x) \in \mathbb{F}_{2^8}[x]$ by the code *generator polynomial* $g(x) \in \mathbb{F}_{2^8}[x]$ having $d_e = n_e - k_e + 1$ degree.

The RS code generator polynomial is defined as:

$$g(x) = (x - \alpha) \cdot (x - \alpha^2) \cdot (x - \alpha^3) \cdots (x - \alpha^{d_e-1}) \tag{8.1}$$

where α is the primitive element in $\mathbb{F}_{2^8}$ having representation $\alpha = y$. As a consequence, the first d_e powers of α are roots of both $g(x)$ and any error-free codeword $c(x)$. A systematic encoding procedure requires that the resulting codeword contains the sequence of symbols of the message polynomial $u(x)$ as a prefix and the error correcting symbols as a suffix, while still being a multiple of $g(x)$. Such encoding is obtained computing

$$c(x) = x^{n_e-k_e}u(x) - \left(x^{n_e-k_e}u(x) \bmod g(x)\right) \tag{8.2}$$

Encoder algorithm In a cyclic block ECC every cyclic rotation of any vector corresponding to a valid codeword is still a valid codeword. This considerable algebraic structure allows to employ a LFSR to perform the encoding of a message to a codeword in systematic form. As the RS code is a cyclic ECC, we employed the LFSR-based circuit represented in Figure 8.5, adapted from the design in [LJ83] to work with symbols in $\mathbb{F}_2^8$, computing $x^{n_e-k_e}u(x)$ modulo $g(x)$ in exactly k_e clock cycles. The coefficients of the message polynomial $u(x)$ are streamed in from highest to lowest degree, while also being reported to the output, by selecting the input 1 of the multiplexers. After k_e clock cycles, the last input coefficient is received, and the content of the shift register is read-out in $n_e - k_e$ clock cycles by driving the multiplexer selector to 0. This circuit efficiently computes the Reed-Solomon codeword in systematic form (Equation 8.2), producing the polynomial coefficients from the highest to lowest degree.

Aiming to support all the three RS codes with a single circuit, we can construct a LFSR able to fit all the coefficients of the generator polynomial with largest degree. During the computation, the coefficients of the correct generator polynomial are selected

In the official HQC specification, the constant term of the generator polynomial for the RS-S1 Reed-Solomon code is incorrectly reported as 9, however the correct value is 89.

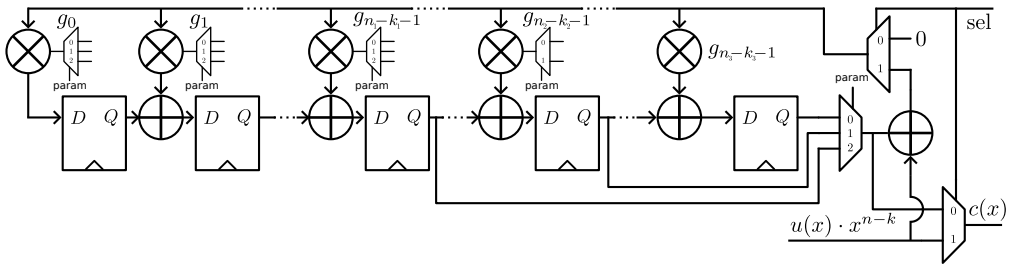


Figure 8.5: *Linear-Feedback Shift Register (LFSR) circuit computing $x^{n_e-k_e} \cdot u(x)$ modulo $g(x)$. The coefficients of the generator polynomial are one operand of the $\mathbb{F}_{2^8}$ multipliers, while the second one is a single coefficient fed-back in the network, which can be either a coefficient of the message polynomial, or the constant $\mathbf{0} \in \mathbb{F}_{2^8}$.*

from a small local ROM and used as one operand of the $\mathbb{F}_{2^8}$ multipliers. Furthermore, the appropriate symbol to report in the feedback network and to the output is chosen depending on the degree of the generator polynomial $g(x)$ of the RS code in use via an additional multiplexer. The encoding latency remains exactly the same of the circuit specifically tailored for each one of the three RS codes, but effectively improving the efficiency of the design by sharing the common resources.

In case a single RS code needs to be supported, the n_e-k_e instantiated $\mathbb{F}_{2^8}$ multipliers have a fixed operand value g_i, $0 \leq i \leq n_e-k_e-1$. Note that each of these fixed operands lies in a small subset of all possible values in $\mathbb{F}_{2^8}$. As detailed in subsection 7.4.2, the fully combinatorial network of a generic $\mathbb{F}_{2^8}$ multiplier include 8 layers composed by `and` and `xor` gates. When a constant 0 is an input of an `and` gate, it makes the output also a constant 0, and consequently such gate can be removed. Therefore, in case an operand of the multiplication is fixed, all `and` gates can be removed, reducing the worst critical path of the circuits of any $\mathbb{F}_{2^8}$ multiplication, improving the overall performance and area usage of the encoder. More refined bit-parallel designs, as the ones in [FH15], have lower performance gains in the case of HQC due to the small size of the $\mathbb{F}_{2^8}$ field.

In Table 8.4 are reported the synthesis of the RS encoder design for the Artix-7 FPGA. All the designs are extremely fast both in terms of the latency of the encoding operation and the maximum working frequency. The missed optimization opportunity when the $\mathbb{F}_{2^8}$ operand is not fixed in the unified design, resulted in a $1.95\times$ area increase compared to the RS encoder for the largest code (the one for `hqc256`). Nonetheless, this unified encoder is still useful due to the halved number of FF and a 10% area reduction compared to the sum of three separated RS encoders optimized for each parameter set.

Decoder algorithm The typical algebraic RS decoder takes as input an error-affected codeword and considers it as a polynomial $r(x) \in \mathbb{F}_{2^8}[x]$, with degree n_e, obtained from the addition of an error-free codeword $c(x) \in \mathbb{F}_{2^8}[x]$, with degree n_e, to an unknown error polynomial $e(x) \in \mathbb{F}_{2^8}[x]$ having $\nu \leq t$ terms: $e(x) = \sum_{k=1}^{\nu} e_{i_k} x^{i_k}$, where i_k identifies the error coefficient with non-zero value. The decoder computes the syndrome

Table 8.4: *Synthesis results for the Reed-Solomon encoder on the Artix-7 FPGA. The Area-Time product is computed as eSlices · μs. The first design has a single LFSR optimized for a specific parameter set. The second design combines the three optimized LFSR to support all the parameter sets in a single unit, while the third one can do the same with a single LFSR.*

Param. set	Design	Resources				Freq.	Latency		AT
		LUT	FF	BRAM	eSlice	MHz	CC	μs	prod.
hqc128	Parameter set optimized	250	299	0	63	419	52	0.12	7.56
hqc192		260	316	0	65	397	62	0.16	10.40
hqc256		446	526	0	112	396	96	0.24	26.88
hqc128	Sum of parameter set optimized	956	1141	0	240	396	52	0.13	31.20
hqc192							62	0.16	38.40
hqc256							96	0.24	57.60
hqc128	Unified design	875	527	0	219	383	52	0.14	30.66
hqc192							62	0.16	35.04
hqc256							96	0.25	54.75

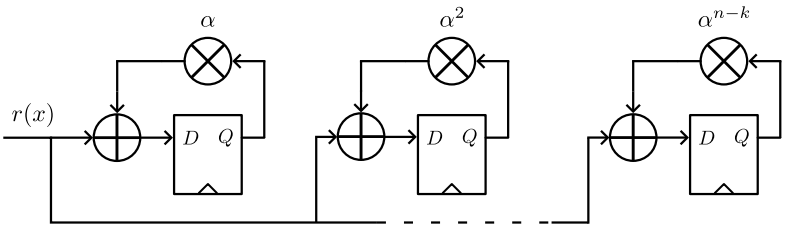


Figure 8.6: *Syndrome polynomial computation*

of the received codeword $r(x)$ and from it derives the number, the positions and the values of the coefficients of $e(x)$. Finally, the error-free codeword is computed as $c(x) = r(x) - e(x)$.

The coefficients of the *syndrome polynomial* $S(x) = S_1 + S_2x + \ldots + S_{2t}x^{2t-1} \in \mathbb{F}_{2^8}[x]$, with $2t = d_e - 1 = n_e - k_e$, associated to the received codeword $r(x)$, are derived by evaluating the latter at each root of the generator polynomial $g(x)$, i.e., $S_j = r(\alpha^j) = c(\alpha^j) + e(\alpha^j) = e(\alpha^j)$. The corresponding evaluation circuit, shown in Figure 8.6, applies Horner's method for polynomial evaluation[Pan66], processing a symbol of the input polynomial $r(x)$ at each clock cycle, from r_{n_e-1} to r_0. After processing all n_e symbols of $r(x)$, the j-th memory element, with $1 \leq j \leq 2t$, will store the j-th syndrome value, i.e., $r_0 + r_1\alpha^j + \ldots + r_{n_e-1}(\alpha^j)^{(n_e-1)} = r(\alpha^j) = e(\alpha^j) = S_j$. By construction, the generator polynomials $g(x)$ of the three codes share the same roots, therefore the $\mathbb{F}_{2^8}$ multipliers in the circuit, that use a power of α as one operand, have a fixed input for all parameters set. Starting from the fully combinatorial network of the Mastrovito bit-parallel design of binary finite fields multiplier [Mas88], which includes both `and` and `xor` gates, we specialized each multiplier instance for each distinct value of an α power, with a net advantage both in terms of critical path length and number of logic gates. The number of $\mathbb{F}_{2^8}$ multiplier instances is given by the largest number of

roots of the RS code generator polynomials defined by the parameter sets.

If all $2t$ syndrome values are zero, then $c(x) = r(x)$ and no errors have occurred. Otherwise $S_j = e(\alpha^j) = \sum_{i=0}^{n_e-1} e_i(\alpha^j)^i = \sum_{k=1}^{\nu} e_{i_k}(\alpha^j)^{i_k} = \sum_{k=1}^{\nu} Y_k X_k^j$ for each $j \in \{1, \ldots, 2t\}$, where $X_k = \alpha^{i_k}$ indicates the presence of an error in the i_k-th received symbol, and $Y_k = e_{i_k}$ is the error symbol value. A decoding algorithm attempts to solve the following set of simultaneous equations, which does not have a straightforward solution:

$$\begin{cases} Y_1X_1 + Y_2X_2 + \ldots + Y_\nu X_\nu = S_1 \\ Y_1X_1^2 + Y_2X_2^2 + \ldots + Y_\nu X_\nu^2 = S_2 \\ \qquad\qquad \vdots \\ Y_1X_1^{2t} + Y_2X_2^{2t} + \ldots + Y_\nu X_\nu^{2t} = S_{2t} \end{cases} \tag{8.3}$$

A strategy to obtain the information captured by Equation 8.3 considers the methods for solving the following equations, based on the definitions of *error locator polynomial* $\Lambda(x)$ and *error evaluator polynomial* $\Omega(x)$:

$$\Lambda(x) \triangleq \prod_{k=1}^{\nu}(1 - X_k x) = 1 + \Lambda_1 x + \ldots + \Lambda_\nu x^\nu \tag{8.4}$$

$$\begin{aligned} \Omega(x) &\triangleq \sum_{k=1}^{\nu} Y_k X_k \prod_{j=1, j\neq k}^{\nu} (1 - X_j x) \\ &= \Lambda(x)S(x) \bmod x^{2t} \\ &= \Omega_0 + \Omega_1 x + \ldots + \Omega_{\nu-1} x^{\nu-1} \end{aligned} \tag{8.5}$$

In particular, $X_k^{-1}{=}\alpha^{-i_k}$ is a root of $\Lambda(x)$ that determines an unique error location i_k. Given $\Lambda(X_k^{-1}) = 0$ for $1 \leq k \leq \nu$, by multiplying both sides by $Y_k X_k^{j+\nu}$ and expanding $\Lambda(x)$, we have that $\sum_{k=1}^{\nu}(Y_k X_k^{j+\nu} + \Lambda_1 Y_k X_k^{j+\nu-1} + \ldots + \Lambda_\nu Y_k X_k^j) = 0$ which yields

$$S_j\Lambda_\nu + S_{j+1}\Lambda_{\nu-1} + \ldots + S_{j+\nu-1}\Lambda_1 = -S_{j+\nu} \tag{8.6}$$

as $S_j = \sum_{k=1}^{\nu} Y_k X_k^j$. Having up to $2t$ syndromes, we can construct a system of t linear equations having the coefficients of Λ as unknowns:

$$\begin{bmatrix} S_1 & S_2 & \ldots & S_\nu \\ S_2 & S_3 & \ldots & S_{\nu+1} \\ \vdots & \vdots & \ddots & \vdots \\ S_\nu & S_{\nu+1} & \ldots & S_{2\nu-1} \end{bmatrix} \begin{bmatrix} \Lambda_\nu \\ \Lambda_{\nu-1} \\ \vdots \\ \Lambda_1 \end{bmatrix} = \begin{bmatrix} -S_{\nu+1} \\ -S_{\nu+2} \\ \vdots \\ -S_{2\nu} \end{bmatrix} \tag{8.7}$$

Since the number of errors $\nu \leq t$ is not known, the PGZ tries all $1 \leq \nu \leq t$ in decreasing order until a matrix with nonzero determinant is found, whose inverse is used to compute the Λ values.

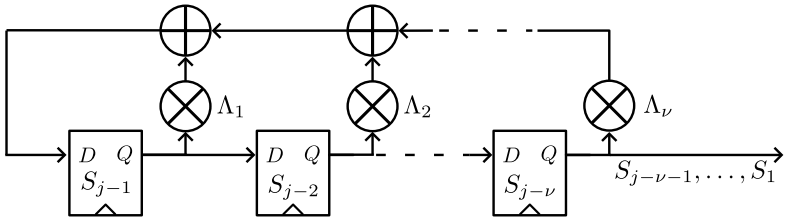


Figure 8.7: *Minimal length LFSR generated by the Berlekamp-Massey Algorithm staring from the sequence of syndromes*

A more efficient algorithm is given by the Berlekamp-Massey algorithm [Mas69]. The Equation 8.6 describes an operation of a LFSR, as depicted in Figure 8.7. Determining the coefficients of Λ is equivalent to synthesizing the minimum length LFSR generating the entire sequence of syndromes $S_1, S_2, \ldots, S_{2t}$ when the LFSR registers are initialized with $S_1, S_2, \ldots, S_\nu$.

Chien Search is a fast algorithm determining the roots of $\Lambda(x)$ with an efficient hardware architecture composed by t multipliers with a constant operand and $t-1$ adders. Given that $\Lambda(\alpha^{-i}) = \sum_{k=1}^{t} \Lambda_k \alpha^{-ik}$, the evaluation with the following power of α is $\Lambda(\alpha^{-i+1}) = \sum_{k=1}^{t} \Lambda_k \alpha^{-ik+k} = \sum_{k=1}^{t} (\Lambda_k \alpha^{-ik}) \alpha^k$, which only requires to multiply the previous result by α^k. The initial value for the k-th term is $\Lambda_k \alpha^{-n_e k}$, and the tested α power is a root for $\Lambda(x)$ if the result is 0.

Once the error locations are known, the error values Y_k can be obtained possibly solving Equation 8.3 via Gaussian elimination, or using Forney's formula, which exploits the fact that the matrix of X_k^j is a Vandermonde matrix. Consider now that the formal derivative $\Lambda'(x)$ is $\frac{d}{dx} \prod_{i=1}^{\nu}(1 - X_i x) = -\sum_{l=1}^{\nu} X_l \prod_{i=1, i\neq l}^{\nu}(1 - X_i x)$ due to Leibniz rule. Evaluated in X_k^{-1}, it yields $-X_k \prod_{j=1, j\neq k}^{\nu}(1 - X_j X_k^{-1}) = \frac{-1}{Y_k}\Omega(X_k^{-1})$ which is used to retrieve $Y_k = -\frac{\Omega(X_k^{-1})}{\Lambda'(X_k^{-1})}$. After the computation of the error polynomial, the error-free codeword can be reconstructed.

In this design we employed the Enhanced Parallel Inversionless Berlekamp-Massey Algorithm (ePIBMA) introduced in [Wu15] to derive $\Lambda(x)$. An overview of the architecture is depicted in Figure 8.8. To support all parameter sets, a number of Processing Elements (PE) equal to the maximum $2t+1$ value are instantiated, taking care to properly preset their registers m_a and m_b with the correct amount of coefficients from the previously computed syndrome polynomial $S(x)$. Eventual unused PEs are initialized with a zero symbol, and the computation is completed after just $2t-1$ clock cycles.

To find the error locations and values, we used the Enhanced Chien Search and Error Evaluation (eCSEE) architecture from [Wu15], depicted in Figure 8.9. It performs both tasks in parallel, reusing the evaluation of X_k on the even powers of the unknown of $\Lambda(x)$ for both the Chien Search and the Error Evaluation, as $\Lambda'(x) = \frac{d}{dx}(\Lambda_0 + \Lambda_1 x + \ldots + \Lambda_\nu x^\nu) = \Lambda_1 + 2\Lambda_2 x + 3\Lambda_3 x^2 + \ldots + \nu\Lambda_\nu x^{\nu-1} = \Lambda_1 + \Lambda_3 x^2 + \ldots = x^{-1}\Lambda_{\text{odd}}(x)$. Due to the small size of the finite field used by the RS code, the finite field inversion necessary to compute Y_k is obtained via a 2 KiB read-only memory. Adapting this

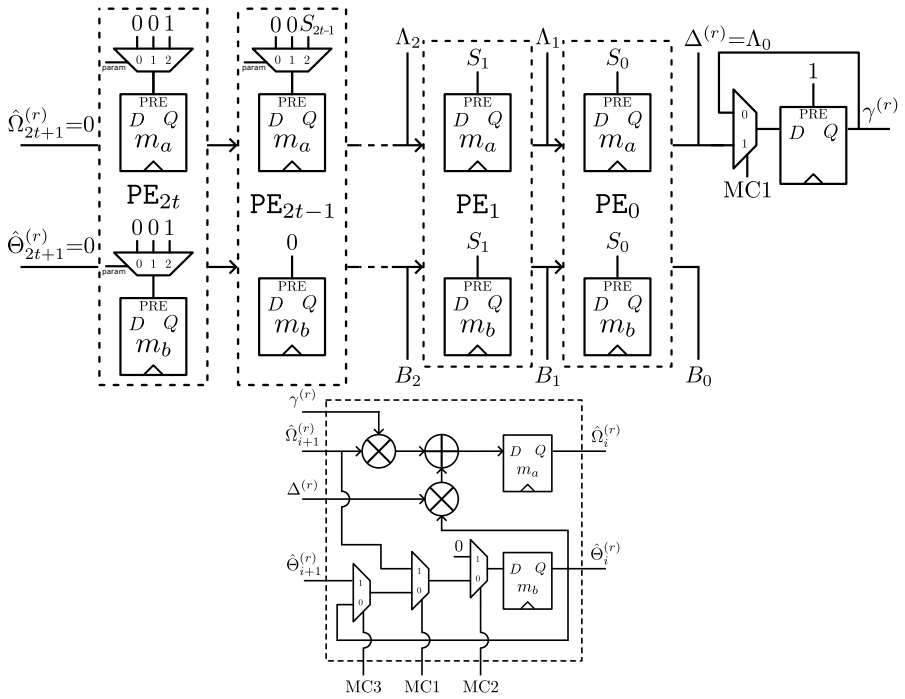


Figure 8.8: *Enhanced Parallel Inversionless Berlekamp-Massey Algorithm architecture computing the error locator polynomial $\Lambda(x)$ and an auxiliary polynomial $B(x)$ starting from the syndrome polynomial $S(x)$*

architecture to support all parameter sets required to instantiate a number of multipliers equal to the maximum length of the error locator and auxiliary polynomials. The zero symbols introduced in the ePIBMA PEs ensured the correct computation of the error vector in n_e clock cycles. To retain a constant-time behavior, the decoding procedure is always executed, even if there are no errors detected during the syndrome computation.

For the RS decoder, the ePIBMA module contributes for the vast majority to the logic resource usage ($\approx 75\%$ of LUTs) due to the large amount of general $\mathbb{F}(2^8)$ multipliers required.

The unified component proved to be extremely efficient with a $1.1\times$ area increase compared to the design optimized for the largest code (the one for `hqc256`). The occupied area is reduced by $2\times$ when compared to a component containing all the decoders specialized for the three RS codes.

Duplicated Reed-Muller code

The HQC specification describes a binary 1^{st} order RM code with parameters $[n_i = 2^l, k_i = l, d_i = 2^{l-1}] = [128, 8, 64]$, with $l = 7$, extended with a repetition code having each codeword composed by m replicas of the same RM original codeword. For the NIST security level 1, the extended RM code $[384, 8, 192]$ is obtained applying a multiplicity $m = 3$, whereas for the security levels 3 and 5, the extended RM code $[640, 8, 320]$ is

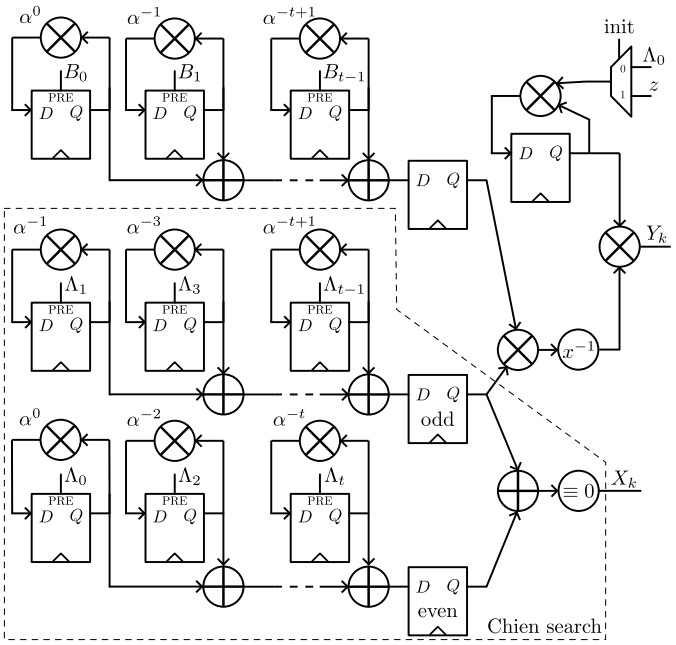


Figure 8.9: *Enhanced Chien Search and Error Evaluation architecture computing the error indexes X_k and error values Y_k starting from $\Lambda(x)$ and an auxiliary polynomial $B(x)$*

Table 8.5: *Synthesis results for the Reed-Solomon decoder on the Artix-7 FPGA. The Area-Time product is computed as eSlices · μs*

Param. set	Design	Resources				Freq.	Latency		AT
		LUT	FF	BRAM	eSlice	MHz	CC	μs	prod.
hqc128	Parameter set optimized	3169	1643	0	793	273	97	0.36	285
hqc192		3648	1652	0	912	265	117	0.44	401
hqc256		6147	2579	0	1537	249	185	0.74	1137
hqc128	Sum of parameter set optimized	12964	5874	0	3242	249	97	0.39	1264
hqc192							117	0.47	1523
hqc256							185	0.74	2399
hqc128	Unified design	6426	2576	0	1607	249	97	0.39	626
hqc192							117	0.47	755
hqc256							185	0.74	1189

Table 8.6: *Synthesis results for the Reed-Muller encoder on the Artix-7 FPGA. The Area-Time product is computed as eSlices · μs*

Param. set	Resources				Freq.	Latency		AT
	LUT	FF	BRAM	eSlice	MHz	CC	μs	prod.
hqc128	246	322	2.0	486	434	194	0.45	218
hqc192	247	329	2.0	486	396	336	0.85	413
hqc256	255	329	2.0	486	411	506	1.23	600

obtained using a multiplicity $m = 5$.

Encoder algorithm To derive the 128-bits codewords $\mathbf{c}$ corresponding to each 8-bits input message $\mathbf{m}$, the schoolbook vector-matrix multiplication encoding procedure considers the RM generator matrix $\mathbf{G}_{\texttt{RM}} \in \mathbb{F}_2^{8\times 128}$ and computes $\mathbf{c} = \mathbf{m}\mathbf{G}_{\texttt{RM}}$. The generator matrix $\mathbf{G}_{\texttt{RM}}$ defined by the authors of HQC is reported in Equation 8.8, here grouped in 32-bits chunks for easier visualization.

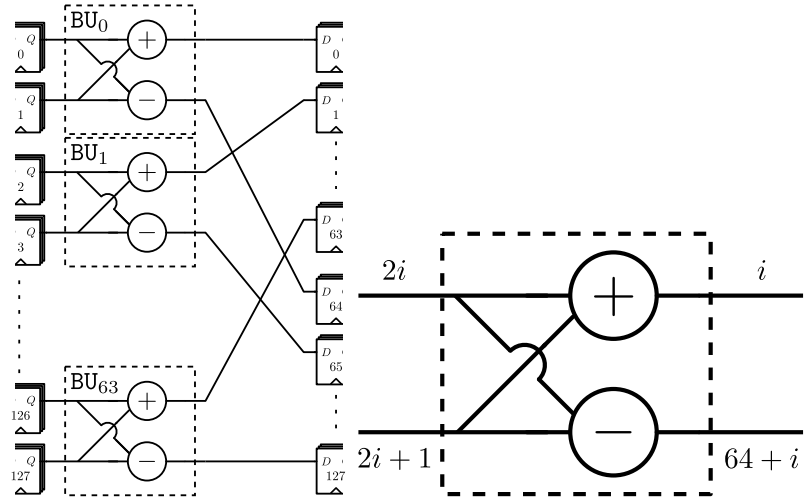
$$\mathbf{G}_{\texttt{RM}} = \begin{bmatrix} \texttt{0xAAAAAAAA} & \texttt{0xAAAAAAAA} & \texttt{0xAAAAAAAA} & \texttt{0xAAAAAAAA} \\ \texttt{0xCCCCCCCC} & \texttt{0xCCCCCCCC} & \texttt{0xCCCCCCCC} & \texttt{0xCCCCCCCC} \\ \texttt{0xF0F0F0F0} & \texttt{0xF0F0F0F0} & \texttt{0xF0F0F0F0} & \texttt{0xF0F0F0F0} \\ \texttt{0xFF00FF00} & \texttt{0xFF00FF00} & \texttt{0xFF00FF00} & \texttt{0xFF00FF00} \\ \texttt{0xFFFF0000} & \texttt{0xFFFF0000} & \texttt{0xFFFF0000} & \texttt{0xFFFF0000} \\ \mathbf{0x00000000} & \mathbf{0xFFFFFFFF} & \mathbf{0x00000000} & \mathbf{0xFFFFFFFF} \\ \mathbf{0x00000000} & \mathbf{0x00000000} & \mathbf{0xFFFFFFFF} & \mathbf{0xFFFFFFFF} \\ \texttt{0xFFFFFFFF} & \texttt{0xFFFFFFFF} & \texttt{0xFFFFFFFF} & \texttt{0xFFFFFFFF} \end{bmatrix} \tag{8.8}$$

Notice that the values in the first 32-bit wide column are repeated also on the following columns, except for the rows highlighted in boldface.

A straightforward encoding design following the schoolbook approach employs $k_i = 8$ multiplexers with two 128 input bits, where one input is the matrix row, and the other input is the constant $\mathbf{0} \in \mathbb{F}_2^{128}$. Each bit of the message $\mathbf{m}$ drives the selection signal of such multiplexers, and $k_i - 1 = 7$ `xor` gates 128-bits wide are used to sum the selected matrix rows. A binary tree structure can be used to minimize the depth of such accumulation process.

A simple and effective optimization applied in the design of the RM encoder considers each row in the generation matrix $\mathbf{G}_{\texttt{RM}}$ as a sequence of 32-bits words and leverages the presence of computations yielding the same intermediate values due to the redundancy of some columns in the matrix. As a consequence, a more compact vector-matrix multiplication can be devised using 9 multiplexers, with smaller two 32 input bits, and 9 banks of `xor` gates 32-bits wide.

Finally, the required multiplicity m, the only differentiator between the duplicated codes defined by each HQC parameter set, is implemented by sequentially storing the resulting value into m contiguous memory locations.



(a) *Hadamard transform layer*

(b) *Connections between the i-th register and each Butterfly unit*

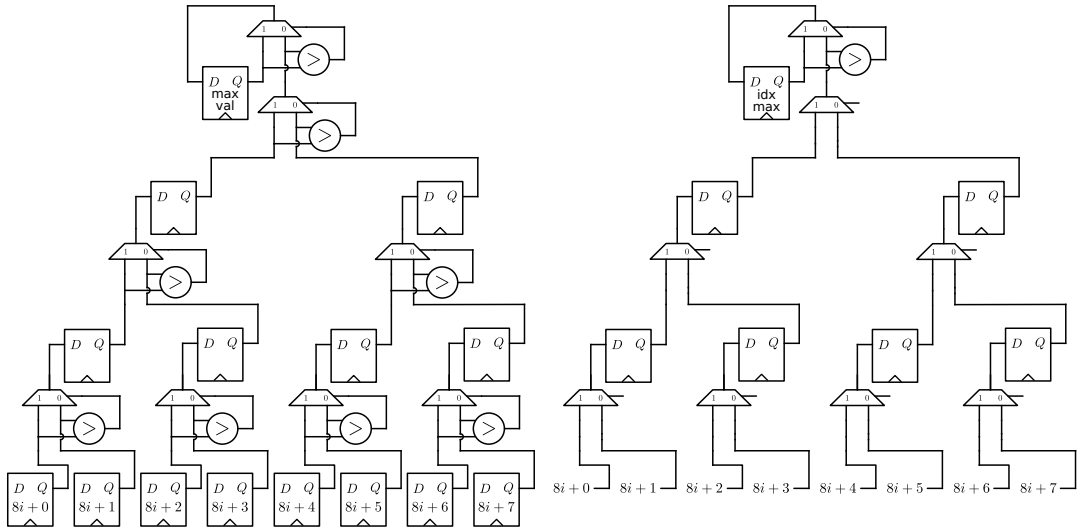
Figure 8.10: *Design computing the Hadamard transform. For graphical reasons, each one of the* 128 *registers on the left side are replicated on the right side, although being the same entity. The same layer is applied* $\log_2 128 = 7$ *times, without tweaking the connections to the registers, and at the end of the computation the transformed result is contained in the registers.*

Decoder algorithm Since the RS decoder acting as the second stage of the RM/RS concatenated decoder needs the entire output of the RM decoder, minimizing the latency of the RM decoder plays a significant role in this design. Therefore, in designing the RM decoder we prioritized performance over resource reuse.

The extended RM codeword is initially de-duplicated accumulating in a flip-flop-based buffer the bitwise sum of the m-fold 128-bit replicas, via 128 independent adders, to the end of dealing with the plain 1st order RM code $[n_i = 2^l, k_i = l, d_i = 2^{l-1}] = [128, 8, 64]$. Considering the appropriate number of message/codeword blocks and multiplicity m defined by each parameter set is enough to support all security levels without extra incurred cost.

The design of the RM decoder follows the implementation of the Maximum Likelihood (ML) decoder computing a fast Hadamard transform, firstly introduced by [BS86], which requires only $O(n_i \log(n_i))$ binary operations and $\log(n_i)$ clock cycles in contrast with the $O(n_i^2)$ binary operations and $O(n_i)$ clock cycles provided by a plain ML-based RM decoder. The transform layer and its butterfly unit are depicted in Figure 8.10. Note that the de-duplication process influenced the actual computation, and the subtraction of the value 64 to the first register is required to compensate such operation.

The maximum absolute value in the result of the Hadamard transform is found with a pipelined comparator tree computing pairwise maxima, acting on a tunable-sized input vector. An example of comparison of 8 parallel values is represented in Figure 8.11. The encoded message corresponds to the index of the register containing the maximum value, hence a side tree is built to determine the (fixed) index value using exactly the



(a) *Comparison of register content finding the maximum value* **(b)** *Selection tree of the register index with the maximum value*

Figure 8.11: *Pipelined comparison tree performing $\zeta = 8$ parallel comparisons determining the maximum value. The operation is iterated $^{128}/_{\zeta}$ to determine the maximum vector value.*

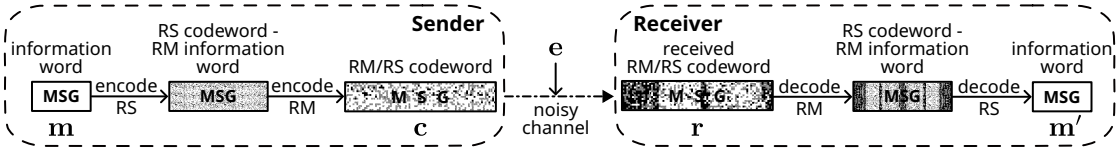


Figure 8.12: *Encoding and decoding of a Error-Correcting Code generated by the concatenation of an external RS code with an internal RM code. The execution order of the sub-codes encoding and decoding procedures is specular.*

same selection signal.

In Table 8.7 are reported the synthesis results of the RM decoder for the Artix-7 FPGA, exploring different the speed/area trade-offs given by the parallel comparison of multiple elements. As the comparison tree is fully pipelined, the number of FF used quickly rises with the size of comparison block, with the best efficiency obtained having 16 parallel comparisons at a small cost of incremented resource count.

Reed-Muller/Reed-Solomon concatenated code

Figure 8.12 represents the steps to perform the encoding and decoding of code generated by the concatenation of an external RS code with an internal RM code:

Encoding the external RS code encodes the original message $\mathbf{m}$ into a RS codeword, which is then interpreted as the message to be encoded by the internal RM code into the final RM/RS codeword

Table 8.7: *Synthesis results for the Reed-Muller decoder on the Artix-7 FPGA when comparing multiple elements elements in parallel. The Area-Time product is computed as eSlices · μs*

Parameter set	Parallel comparisons	Resources LUT	FF	BRAM	eSlice	Freq. MHz	Latency CC	μs	AT prod.
hqc128	4	3034	1392	1.0	971	226	2256	9.98	9.69
	8	3069	1486	1.0	980	210	1566	7.46	7.31
	16	2761	1642	1.0	903	219	1244	5.68	5.13
	32	3266	2006	1.0	1029	211	1106	5.24	5.39
	64	3815	2995	1.0	1166	211	1060	5.02	5.86
hqc192	4	3411	1557	1.0	1065	225	2858	12.70	13.53
	8	3432	1660	1.0	1070	216	2018	9.34	10.00
	16	3606	1850	1.0	1114	193	1626	8.42	9.39
	32	3935	2235	1.0	1196	196	1458	7.44	8.90
	64	4630	3290	1.0	1370	210	1402	6.68	9.15
hqc256	4	3347	1528	1.0	1049	187	4592	24.56	25.76
	8	3449	1659	1.0	1075	216	3242	15.01	16.13
	16	3599	1835	1.0	1112	217	2612	12.04	13.38
	32	4095	2238	1.0	1236	218	2342	10.74	13.28
	64	4631	3291	1.0	1370	205	2252	10.99	15.05

Table 8.8: *Length, dimension, and minimum distance of the $[n, k, d]$ concatenated RM/RS fixed code and its sub-codes. The rightmost column represents the failure probability of the decoding procedure when the concatenated code is used in the HQC scheme.*

Parameter set	Concatenated code Reed-Solomon	Reed-Muller	Constraints $p \approx n_e n_i$	$k_e k_i$	DFR
hqc128	$[46, 16, 31]$	$[384, 8, 192]$	$17669 \approx 17664$	128	2^{-128}
hqc192	$[56, 24, 33]$	$[640, 8, 320]$	$35851 \approx 35840$	192	2^{-192}
hqc256	$[90, 32, 59]$	$[640, 8, 320]$	$57637 \approx 57600$	256	2^{-256}

Decoding the RM/RS codeword, considered as a codeword of the internal RM code, gets decoded into its message, which is then interpreted as the codeword of the external RS code to be decoded into the message $\mathbf{m}'$

If the error $\mathbf{e}$ introduced during the transmission has a Hamming weight lower than the error-correcting capability of the concatenated code, the original message $\mathbf{m} = \mathbf{m}'$ is successfully retrieved.

While the RM and RS encoders can work in a pipelined fashion processing the original input message byte-wise, this is not possible for the RM and RS decoders, since the latter requires the whole codeword during the very first step of the decoding algorithm.

Recall that fixed RM/RS concatenated code $[n_e n_i, k_e k_i, d_e d_i]$ is formed by a shortened RS $[n_e, k_e, d_e]$ external code, and a duplicated RM $[n_i, k_i, d_i]$ internal code, identified by the generator matrix $\mathbf{G}$. Table 8.8 reports the lengths and dimensions of the RS and RM codes used by HQC that satisfy the constraints $p \approx n_e n_i$, $k_e k_i$ being the appropriate length of the message $\mathbf{m}$ secure against a bruteforce search, and guaranteeing the required DFR for each security level defined by NIST.

Table 8.9: *Synthesis results for the Reed-Muller/Reed-Solomon encoder and decoder on the Artix-7 FPGA compared to the current state-of-the-art. The RM decoder compares* 16 *elements at a time, and the Area-Time product is computed as eSlices* $\cdot$ μs

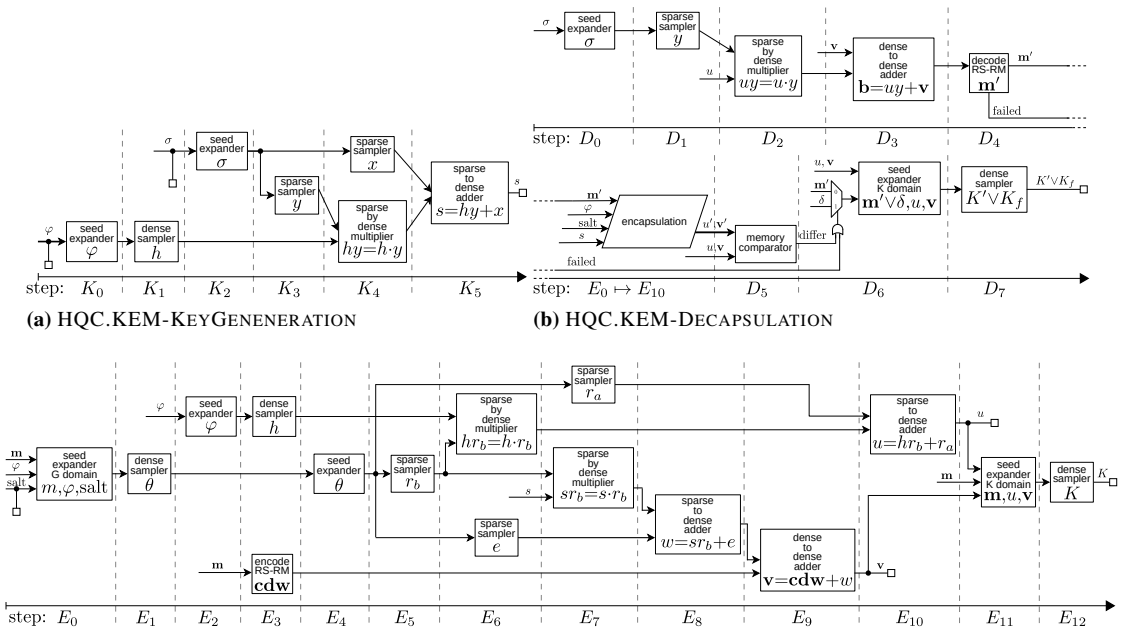
Module	Parameter set	Design	Resources				Freq.	Latency		AT
			LUT	FF	BRAM	eSlice	MHz	CC	μs	prod.
Encoder	hqc128	This work	532	620	2.0	557	374	148	0.40	222
		[Des+23]	858	922	2.0	639	270	97	0.36	230
		[Ae22]	2019	603	0.0	505	–	7244	–	–
	hqc192	This work	570	644	2.0	567	394	290	0.74	419
		[Des+23]	1011	1088	2.0	677	298	131	0.44	297
	hqc256	This work	743	854	2.0	610	386	460	1.19	725
		[Des+23]	1503	1689	2.0	800	293	189	0.65	520
Decoder	hqc128	This work	5896	3364	2.5	2004	212	1293	6.10	12224
		[Des+23]	2817	3779	2.5	1235	205	4611	22.49	27775
		[Ae22]	10154	2569	3.0	3175	–	68619	–	–
	hqc192	This work	7219	3561	2.5	2335	219	1685	7.69	17956
		[Des+23]	3257	4727	2.5	1345	212	5485	25.87	34795
	hqc256	This work	10090	4472	2.5	3053	225	2705	12.02	36697
		[Des+23]	3679	5574	2.5	1450	206	9199	44.66	64756

In Table 8.9 are reported the synthesis results of the concatenated RM/RS code encoder and decoder for the Artix-7 FPGA. The RM/RS encoder proved to be remarkably compact, fast, and efficient, although slower with respect to [Des+23]. However, the encoder is already faster than the dense polynomial sampler, which is run in parallel in the operation schedule, therefore in this case a smaller design is preferred over a faster one.

Comparing the concatenated decoder with [Des+23], this design uses from $2\times$ to $2.74\times$ more LUTs, while requiring less FFs. Nonetheless, this solution is from $3.36\times$ to $3.71\times$ faster and about twice more efficient in the Area-Time product metric.

8.2.2 Operation scheduling

Figure 8.13 illustrates the resulting operation schedule for the hqc scheme when utilizing five true dual-port RAM memories. Each column delimited by vertical dashed lines correspond to a FSM state, and the blocks are the scheme operations mapped on a specific hardware component. Furthermore, the decision to employ precisely five memories stems from an analysis of parallelizable operations, which revealed the feasibility of scheduling a polynomial multiplication alongside the sampling of a random polynomial with a fixed Hamming weight. By constraining the latency of the multiplier unit to closely match the one of the polynomial sampling unit, the required number of memory read ports accessed by the multiplier was determined to be $l = 4$, which in turn determined the amount of memories used. The HQC scheme offers limited opportunities for parallelization at the algorithmic level. One notable instance is the concurrent execution of the RM/RS concatenated encoding alongside the sampling of the public



(a) HQC.KEM-KEYGENERATION **(b)** HQC.KEM-DECAPSULATION

(c) HQC.KEM-ENCAPSULATION

Figure 8.13: *Schedule of the operations for the* HQC.KEM *scheme. A white box represents an output value*

key polynomial h. However, additional parallelization possibilities may be restricted by the presence of shared components, such as the Keccak module, which could introduce resource contention.

The choice of true dual-port RAMs over simple dual-port variants is driven by the high demand for read ports in the polynomial multiplication unit. Since the dense operand for the polynomial multiplication must be replicated across all memories that supply read access to the multiplier component, minimizing the number of copies and instantiated RAM modules led to the selection of true dual-port memories as the most efficient option for the FPGA implementation. However, in an ASIC design flow where the use of customized memories gives more design freedom, alternative and more efficient memory architectures can be explored. Additionally, due to the specific access pattern required for sampling fixed-weight vectors, two memory blocks are necessary to support byte-enable writes with 16-bit wide words. Each memory port is dedicated to a single computing module during the execution of a specific state in the global FSM, ensuring that the arbitration logic remains minimal.

To optimize performance, we adopted a flattened HDL module hierarchy, integrating the HQC.PPKE components directly within the HQC.KEM implementation. This approach leverages the reduced latency achieved through a manually routed interconnect, efficiently linking the computing modules to both the memory blocks and the SHAKE256 component. Given these constraints, the operation schedule executed by

each top-level module remained identical, irrespective of the differing parameters specified in the official HQC documentation for the NIST security levels 1, 3, and 5, which are referred to as `hqc128`, `hqc192`, and `hqc256`, respectively.

HQC.KEM-KeyGeneration Module

The schedule of the key pair generation is reported in Figure 8.13a. The seed φ is expanded into the dense public polynomial h (step K_0 and K_1), and the seed γ is expanded (step K_2) into the two private polynomials x and y.

Inverting the order of the sampled private polynomials enables a potential parallelism between the sampling of the second polynomial x and the first polynomial multiplication $h \cdot y$. The proposed variation of the HQC specification, recently accepted in the latest specification 4th October 2024, comes at no resource or security loss. Alternatively, software implementation in constraint platforms, such as low-end microcontrollers, can benefit from it deferring the expansion of x and overwriting y after the multiplication, as it is no longer used in the algorithm after that operation. Moreover, only the y polynomial is used during the decapsulation element, therefore the polynomial x is not generated using the new sampling order, avoiding to waste time discarding the first $w \cdot 32$-bits linked to x squeeze.

Consequently, the HQC.KEM-KeyGeneration schedule samples the sparse polynomial y in step K_3, and multiplies it by h in step K_4 while x is sampled in the meantime. This variation in the sampling order combined to this schedule of operation improved the overall latency of the whole key generation from 15% to 38% depending on the parameter set in use. Adding the sparse polynomial x to the result of the multiplication $h \cdot y$ generates the syndrome s in step K_5. Note that the sparse polynomial y can be sampled in parallel with h by replicating the SHAKE module if the price of the extra hardware resource is acceptable.

HQC.KEM-Encapsulation Module

The schedule of the key encapsulation, reported in Figure 8.13c, starts by absorbing the message $\mathbf{m}$, the salt, the syndrome s, and the public key seed φ via the HASH_G function (step E_0), generating the encryption seed θ (step E_1). During the expansion of the parity-check polynomial h from φ, the message $\mathbf{m}$ is encoded via the RM/RS encoder (steps E_2 and E_3). Afterwards, the sparse polynomials r_a, r_b, and e are sampled using the output of the CSPRNG having absorbed the encryption salt θ.

Here is proposed a further optimization of the generation order of the polynomials, recently included in the latest HQC specification [Agu+24a], by sampling the values r_b, e, and then r_a. This allows the parallel execution of two sparse-by-dense multiplications during the first two sampling operations (steps E_5 to E_7). Applying this variation in the sampling order of these elements improved the latency of the whole encapsulation from 15% to 35% depending on the chosen parameter set. Moreover, it enables SW implementations in constrained platforms to expand the required element right before its

use, without cluttering the system memory or expanding multiple times the encryption seed.

In step E_8, the sparse polynomial e is incorporated into the product of s and r_b by inverting the binary coefficients at the index positions indicated by e. The resulting value is then combined with the codeword to produce $\mathbf{v}$ in step E_9. Ultimately, in step E_{10}, the sparse polynomial r_a is added to the product of h and r_b, resulting in the polynomial u. Once the message $\mathbf{m}$ and the ciphertext $u\|\mathbf{v}$ are absorbed in the HASH_K domain (step E_{11}), the first 512-bits of SHAKE's internal state will hold the session key K, which is securely stored in memory.

HQC.KEM-DECAPSULATION Module

The scheduling of this module, depicted in Figure 8.13b, begins by leveraging the optimization of the HQC.KEM-KEYGENERATION procedure. This allows for the sampling of the sparse polynomial y (step D_1) without discarding any output from the SHAKE-based CSPRNG, which would have been required if the x polynomial had been sampled first. In step D_2, the sparse polynomial y is multiplied by the first portion of the ciphertext u, and in step D_3 added to the second portion $\mathbf{v}$. Following step D_3, the resulting value is a decodable message $\mathbf{m}'$, which is passed to the concatenated RM/RS decoder in step D_4, completing the HQC.PPKE decryption.

Then, the encapsulation steps E_0 through E_{10} are repeated due to the HHK framework using the decoded message $\mathbf{m}'$. If the resulting values $u'\|\mathbf{v}'$ do not match the received ciphertext $u\|\mathbf{v}$ (step D_5), or if the decoding of the message fails, the secret value σ bound to HQC's private key is used instead of $\mathbf{m}'$ to generate an invalid session key K_f through the SHAKE module. The session key $\mathsf{K}' \vee \mathsf{K}_f$ is then produced by absorbing the ciphertext $u\|\mathbf{v}$ (step D_6) and keeping the first 512-bits of the output produced by the SHAKE module (step D_7).

Note that the variation in the sampling order of y, r_a, r_b, and e also benefited the decapsulation module due to the re-encryption performed by the HHK transformation, resulting in a latency reduction between 13% and 32% depending on the employed parameter set.

8.2.3 Design synthesis and implementation

In Table 8.10 are reported the synthesis results of full HW designs of some Post-Quantum KEM schemes from NIST PQC contest targeting an Artix-7 FPGA. The lattice-based candidates Streamlined NTRU Prime and CRYSTALS-Kyber are reported in the first rows in each security category, followed by the code-based schemes HQC, BIKE and Classic McEliece. Highlighted rows distinguish unified designs that additionally are compatible with all parameter sets.

The proposed solution has lower latency and higher efficiency than [Des+23], while also supporting all HQC parameters sets, even considering the lower working frequency caused by high routing congestion and a net delay contributing by more than 83% to the

Table 8.10: *Comparison with other unified hardware accelerators for post-quantum KEM algorithms synthesized for an Artix-7 FPGA and supporting all KEM operations. The highlighted designs in addition support all parameters sets. The efficiency indicator Area × Time (AT) product is expressed in eSlices·ms. Lattice-based designs are reported in the first rows for each security category, followed by code-based designs.*

	Design			Resource					Freq.	KeyGeneration		Encapsulation		Decapsulation	
sec.	scheme	ref.	variant	LUT	FF	BR	DSP	eSlice	MHz	μs	AT	μs	AT	μs	AT
AES-128	Kyber	[DMG23]	–	9347	8186	6	4	4147	220	10	40	15	62	20	85
	Kyber	[XL21]	–	7412	4644	3	2	2758	161	24	65	32	87	42	115
	HQC	This work	–	26561	13636	28	0	12471	143	39	488	82	1027	128	1597
	HQC	[Des+23]	balanced	18662	7088	22	8	10406	164	96	1000	204	2122	294	3059
	HQC	[Des+23]	high-speed	20011	7484	24	8	11167	178	89	989	126	1407	209	2333
	HQC	[Li+23]	–	24591	11270	68	0	20564	178	112	2311	225	4621	393	8087
	BIKE	[Ric+22]	lightweight	12319	3896	9	7	5930	121	3826	22691	446	2646	6950	41216
	BIKE	[Ric+22]	trade-off	19607	5008	17	9	9717	100	1870	18171	280	2721	4210	40909
	BIKE	[Ric+22]	high-speed	25549	5462	34	13	15344	113	1681	25800	133	2037	1168	17924
	C. McEliece	[Che+22]	lightweight	23890	45658	138	5	36007	112	1161	41794	1518	54653	79286	2854841
	C. McEliece	[Che+22]	high-speed	40018	61881	178	4	48173	113	265	12789	885	42631	8584	413520
AES-192	S. NTRU P.	[Pen+23]	low area	9574	4399	8	18	6617	128	4917	32535	228	1512	669	4427
	S. NTRU P.	[Pen+23]	high-speed	41428	26381	36	31	22265	140	457	10182	36	796	78	1748
	Kyber	[DMG23]	–	10434	9473	8	6	5218	220	12	64	18	93	23	119
	Kyber	[XL21]	–	7412	4644	3	2	2758	161	39	108	49	135	62	171
	HQC	This work	-	26561	13636	28	0	12471	143	96	1200	200	2497	305	3799
	BIKE	[Ric+22]	lightweight	13850	4010	15	7	7584	116	15302	116048	1353	10265	20526	155668
	BIKE	[Ric+22]	trade-off	20049	5039	17	9	9827	100	6930	68101	800	7862	11980	117727
	BIKE	[Ric+22]	high-speed	25811	5460	34	13	15410	113	6027	92869	372	5728	5354	82505
AES-256	Kyber	[DMG23]	–	11527	10767	10	8	6184	220	16	101	22	135	27	169
	Kyber	[GLK22]	–	7900	3900	16	4	5905	159	49	290	53	312	66	390
	Kyber	[XL21]	–	7412	4644	3	2	2758	161	58	161	70	194	86	238
	HQC	This work	–	26561	13636	28	0	12471	143	181	2262	378	4718	574	7158
	BIKE	[Ric+22]	lightweight	13973	4002	34	7	11643	113	42558	495497	3035	35341	46168	537536
	BIKE	[Ric+22]	trade-off	21373	5160	34	9	13762	94	19649	270409	1851	25474	27872	383579
	BIKE	[Ric+22]	high-speed	26441	5601	34	13	15567	111	16198	252157	811	12622	11901	185261

total delay in the critical path. Similarly to [Li+23], we are not using specialized DSP units, but this design requires less than half of the BRAMs and supports all the parameters sets, showing a remarkable efficiency. CRYSTALS-Kyber has several low-latency, compact, and efficient hardware implementations: among all the other code-based candidate designs, this design for HQC is the only one showing metrics in the same order of magnitude. This further confirms that HQC is a noteworthy alternative to CRYSTALS-Kyber, particularly for the AES−128 equivalent security margin, as the penalties in latency and efficiency when using higher security guarantees grow faster in all considered code-based schemes compared to CRYSTALS-Kyber. When compared to BIKE, the high-speed variant of [Ric+22] reports a similar but higher eSlice resource usage, while being from $1.62\times$ to $89\times$ slower than our solution. Classic McEliece is unable to guarantee security margins higher than AES−128 on a low-end target as an Artix-7 due to the error correction code size not fitting in the available BRAM, and results from $6.8\times$ to $67\times$ slower than the presented HQC solution. Finally, this HQC design can compete with hardware designs for lattice-based Streamlined NTRU Prime, roughly positioning somewhere between the low area and high-performance solutions of [Pen+23]. As for any NTRU-derived scheme, the key generation is the most expensive operation which does not compare favorably with HQC.

In Table 8.11 are reported the results for the designs specialized for either the server (key generation and decapsulation KEM operations) and client (just supporting the encapsulation KEM operation). The BIKE designs in [Gal+22] maximized the parallelism of the computing modules to fill the entire FPGA fabric of an Artix-7 50T (low-cost chip variant) and an Artix-7 200T (top-of-the-line chip variant), providing top level designs specialized for the parameter sets having security margins of AES−128 and AES−192. Due to the expensive inversion operation in the key generation, the server designs are penalized both in terms of latency and, by a larger margin, in efficiency with respect to the presented solution. On the other end, the lightweight client designs show $\approx 2\times$ better latency and efficiency figures than the proposed design, partially due to the larger number of operations carried out during the HQC encapsulation. Finally, the CRYSTALS-Kyber client and server designs in [XL21] have from $2.3\times$ to $4.7\times$ better latency than this design of HQC, with an efficiency improved by $7\times$ to $21\times$.

The unified design was then evaluated using an ASIC toolchain composed of *Yosys* and *OpenROAD* for synthesis and place-and-route, respectively. The tools used the *FreePDK45* technology library to complete the open-source ASIC toolchain, and evaluated using the typical process corner of 1.1V at 25°C. Due to the current lack of maturity of the *OpenRAM* memory compiler, the memory buses are directly exposed as I/O pins of the chip. Despite this fact, the resulting design reached a maximum operating frequency of 419 MHz, occupying an area of 0.496mm^2.

To better appreciate the computational advantages provided by this HQC accelerator targeting an AMD Artix-7 FPGA, it is worth considering also the performance of a software implementation run on a CPU-based platform, which can be fit in the same cost envelop and at the same level of maturity of semiconductor fabrication technology.

Table 8.11: *Results of the synthesis exploration for the* NTRU.KEM *server/client configurations on an Artix-7 FPGA. The highlighted designs in addition support all parameters sets. The contribution of the* `SHAKE256` *of* 5520 *LUTs and* 2810 *FFs is excluded. The efficiency indicator Area-Time (AT) product is expressed in eSlices · ms.*

(a) *Comparison with other client hardware accelerators for post-quantum KEM supporting only the encapsulation primitive*

Design				Resource					Freq.	Client	
security	scheme	ref.	variant	LUT	FF	BR	DSP	eSlice	MHz	μs	AT
AES-128	BIKE	[Gal+22]	high-speed	126510	51492	357	0	107312	91	30	2438
	BIKE	[Gal+22]	lightweight	31792	17805	44	0	17170	91	30	339
	HQC	This work	–	26533	13688	28	0	12464	156	73	595
	Kyber	[XL21]	–	7412	4644	3	2	2758	161	30	79
AES-192	BIKE	[Gal+22]	high-speed	124891	53067	360	0	107543	91	60	4086
	BIKE	[Gal+22]	lightweight	31411	20181	46	0	17499	91	80	874
	HQC	This work	–	26533	13688	28	0	12464	156	175	1433
	Kyber	[XL21]	–	7412	4644	3	2	2758	161	47	123
AES-256	HQC	This work	–	26533	13688	28	0	12464	156	328	2688
	Kyber	[XL21]	–	7412	4644	3	2	2758	161	68	176

(b) *Comparison with other server hardware accelerators for post-quantum KEM supporting the key generation and decapsulation primitives*

Design				Resource (server)					Freq.	Server	
security	scheme	ref.	variant	LUT	FF	BR	DSP	eSlice	MHz	μs	AT
AES-128	BIKE	[Gal+22]	high-speed	91422	46208	276	0	81262	91	580	62241
	BIKE	[Gal+22]	lightweight	19804	11401	30	0	11311	91	5710	98041
	HQC	This work	–	15847	9335	20	0	8202	172	158	1966
	Kyber	[XL21]	–	6785	3981	3	2	2602	167	65	179
AES-192	BIKE	[Gal+22]	high-speed	72725	37795	236	0	68108	91	1710	183899
	BIKE	[Gal+22]	lightweight	19979	12282	28	0	10931	91	19270	337206
	HQC	This work	–	15847	9335	20	0	8202	172	377	4693
	Kyber	[XL21]	–	6785	3981	3	2	2602	167	102	280
AES-256	HQC	This work	–	15847	9335	20	0	8202	172	707	8813
	Kyber	[XL21]	–	6785	3981	3	2	2602	167	145	399

Specifically, we referred to the eBATS (ECRYPT Benchmarking of Asymmetric Systems) public benchmarking tool [VAM22] and identified the Rockchip RK3288 ARM Cortex-A17 CPU, running at 1800 MHz, produced with a 28nm HKMG technology, as the nearest one to a XC7A35T platform, included in the AMD Artix-7 lineup produced with a 28nm HPL technology, which can easily accomodate our design. Note that the current bulk price of the AMD Artix-7 XC7A35T chip and of the Rockchip RK3288 ARM Cortex-A17 CPU is 16 vs 19 US dollars, respectively. The benchmarked software complies to the HQC round 1 submission, and is the optimized variant provided by the authors of HQC. The runtime execution of the key generation, the encapsulation, and the decapsulation takes $8,901$, $18,430$, and $31,917$ kcycle for the three parameter sets, thus 4.94, 10.2, and 17.7 ms considering the $1,800$ MHz clock frequency. By comparison, the unified hardware accelerator takes only 0.249, 0.601, and 1.13 ms, which makes it $19.8\times$, $16.9\times$, and $15.6\times$ faster than the CPU counterpart.

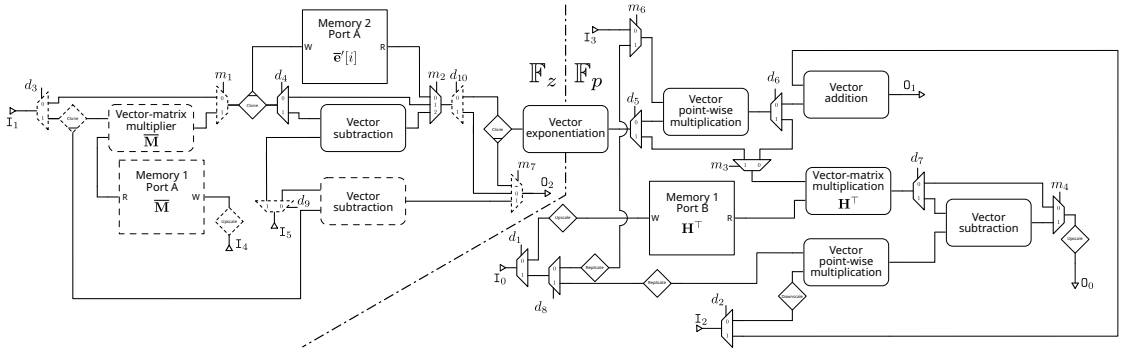


Figure 8.14: *Arithmetic unit design for CROSS supporting all the arithmetic operations of* CROSS.KEYGENERATION*,* CROSS.SIGN*, and* CROSS.VERIFY *composed as a partially reprogrammable chain of modules. The elements outlined with a dashed line are specific to R-SDP(G) parameters, and can be omitted by the designs supporting a R-SDP parameter set.*

8.3 CROSS

In this section are presented the missing components required to complete the design of the CROSS digital signature scheme, starting from the development of an arithmetic unit collecting all the arithmetic modules in a single component to simplify the logic of the top-level design FSA. Afterwards, the tree unit module is briefly presented, with some insights on the performance and the required resources. Finally, the scheduling of the operations carried out during the three DS primitives, namely the CROSS.KEYGENERATION, CROSS.SIGN, and CROSS.VERIFY, is described before presenting the results of the CROSS hardware accelerator.

8.3.1 Arithmetic unit

Analyzing the arithmetic operations carried out during the CROSS.KEYGENERATION, CROSS.SIGN and CROSS.VERIFY, such as the vector addition, subtraction and pointwise multiplication, exponentiation, and the vector-matrix multiplications, it can be noticed that the sequence of arithmetic operations is mostly fixed. Therefore, to reduce the size of the point-to-point network and the size of the FSA in the top-level design, it can be designed an arithmetic unit combining the aforementioned modules, previously described in section 7.5, with a semi-flexible chain structure depicted in Figure 8.14. Each module transfers data using a 64-bits wide Advanced eXtensible Interface (AXI) stream interface, which is part of the Advanced Microcontroller Bus Architecture (AMBA) open-standard from ARM.

Inside the arithmetic unit there are some locally memories to access the matrices $\mathbf{V}^\top$ and $\overline{\mathbf{W}}$ using large 192-bits words, which are loaded during an initialization phase receiving a single coefficient in $\mathbb{F}_z$ or $\mathbb{F}_p$ per clock cycle in parallel. Furthermore, another memory contains all the t distinct $\overline{\mathbf{e}}'[i]$ determined during the computation of the commitments $\mathtt{cmt}_0[i]$ during the CROSS.SIGN operation, used to generate the first chal-

Table 8.12: *Logic of the FSA managing the CROSS arithmetic unit, and elements transferred in the input/output stream interfaces. $a \vee b$ means that a is stream when using a R-SDP parameters set, and b in case of R-SDP(G) parameter sets. Text in red (respectively, blue) means that the element is streamed from the CSPRNG (respectively, memory).*

Stream interface	Arithmetic operation						
	Init.	**Key generation**	**Expand $\overline{\mathbf{e}}$**	**$\mathtt{cmt}_0[i]$ computation**	**$\mathtt{chall}_1[i]$ response**	**Verify $\mathtt{chall}_2[i]$=0**	**Verify $\mathtt{chall}_2[i]$=1**
$\mathtt{I}_0$	$\mathbf{V}^\top$	–	–	–	$\mathtt{chall}_1[i]$	$\mathtt{chall}_1[i]$	$\mathtt{chall}_1[i]$
$\mathtt{I}_1$	–	$\overline{\mathbf{e}} \vee \overline{\mathbf{e}}_G$	$\overline{\mathbf{e}}_G$	$\overline{\mathbf{e}}'[i] \vee \overline{\mathbf{e}}'_G[i]$	–	$\overline{\mathbf{v}}[i] \vee \overline{\mathbf{v}}_G[i]$	$\overline{\mathbf{e}}'[i] \vee \overline{\mathbf{e}}'_G[i]$
$\mathtt{I}_2$	–	–	–	–	$\mathbf{u}'[i]$	$\mathbf{s}$	$\mathbf{u}'[i]$
$\mathtt{I}_3$	–	–	–	$\mathbf{u}'[i]$	–	$\mathbf{y}[i]$	–
$\mathtt{I}_4$	$\overline{\mathbf{W}}$	–	–	–	–	–	–
$\mathtt{I}_5$	–	–	–	$\overline{\mathbf{e}} \vee (\overline{\mathbf{e}}_G, \overline{\mathbf{e}})$	–	–	–
$\mathtt{O}_0$	–	$\mathbf{s}$	–	$\mathbf{s}'[i]$	–	$\mathbf{s}'[i]$	–
$\mathtt{O}_1$	–	–	–	–	$\mathbf{y}[i]$	–	$\mathbf{y}[i]$
$\mathtt{O}_2$	–	–	$\overline{\mathbf{e}}$	$\overline{\mathbf{v}}[i] \vee \overline{\mathbf{v}}_G[i]$	–	–	–

lenge responses $\mathbf{y}[i]$ during the same operation. As a consequence, the sampling unit and the vector-matrix multiplication unit specialized for the matrix $\overline{\mathbf{M}}$ are not executed a second time, sensibly speeding-up the computation of the first challenge responses.

The streams used by the modules are 64-bits wide, allowing to compute multiple operations on $\mathbb{F}_z$ and $\mathbb{F}_p$ coefficients each clock cycle. When a R-SDP(G) parameter set is used, some parts of the arithmetic unit can be safely removed, such as the vector-matrix multiplication specialized for the matrix $\overline{\mathbf{M}}$. These modules are denoted by dashed borders in Figure 8.14. Moreover, by looking at the figure, the schematic can be partitioned in two isolated sections, separated by the vector exponentiation unit. The first section involves arithmetic in $\mathbb{F}_z$ where the vector coefficients represent the exponents of the element g generating the group E^n, while the other section involves arithmetic in $\mathbb{F}_p$ after the conversion of a vector in $\mathbb{F}_z^n$ to a vector in $\mathbb{F}_p^n$ by means of the exponentiation unit. Considering 64-bits wide AXI streams, the number $\mathbb{F}_p$ and $\mathbb{F}_z$ coefficients encoded in a single word differs, as $z < p$. Therefore, a small FIFO buffer at the input of the exponentiation unit is used to compose a smaller word having fewer $\mathbb{F}_z$ coefficients in it, matching the number of the $\mathbb{F}_p$ ones contained in a single 64-bits word.

In Table 8.12 is reported the logic of the FSA managing the semi-programmable computation chain. For each arithmetic operation is defined the expected elements received at the input stream interfaces, and the elements generated at some output stream interface.

The initialization operation loads the matrices $\mathbf{V}^\top$ and $\overline{\mathbf{W}}$ into the local memories with the data received from the CSPRNG during the CROSS.KeyGeneration, CROSS.Sign and CROSS.Verify procedures.

The key generation operation receives the private key error vectors $\overline{\mathbf{e}}$ or $\overline{\mathbf{e}}_G$, depending on the underlying hard problem, and produces the syndrome $\mathbf{s}$. This operation is clearly performed in the CROSS.KeyGeneration algorithm, and is executed only

Table 8.13: *Synthesis results for the CROSS arithmetic unit when targeting an AMD Artix-7 FPGA and using 64-bits wide AXI streams. The latencies consider the throughput of the rejection sampler unit, and exclude the contribution of the initialization phase and the one-time expansion of $\overline{\mathbf{e}}_G$. and consider the t parallel executions of the* CROSS-ID *protocol and the second challenge Hamming weight w. The area of the arithmetic unit does not vary depending on the fast, balanced, or small optimization corners, with the only exception being the BRAM units.*

Parameter set	Resources					Freq.	Latency (CC)		
	LUT	FF	BRAM	DSP	eSlice	MHz	KeyGen.	Sign	Verify
CROSS-RSDP-1-f	4715	2808	10.0	0	3299	138	355	$(360+27)t$	$298(t-w)+210w$
CROSS-RSDP-3-f	4746	2971	14.0	0	4155	136	716	$(710+33)t$	$650(t-w)+274w$
CROSS-RSDP-5-f	5019	2929	14.0	0	4223	137	1074	$(1084+40)t$	$1020(t-w)+347w$
CROSS-RSDPG-1-f	9141	4153	10.0	29	8306	109	188	$(193+21)t$	$190(t-w)+145w$
CROSS-RSDPG-3-f	9122	4479	10.0	29	8301	112	343	$(349+25)t$	$344(t-w)+248w$
CROSS-RSDPG-5-f	9158	4693	14.0	29	9158	111	482	$(488+29)t$	$485(t-w)+348w$

once.

The expand $\overline{\mathbf{e}}$ operation is only required by R-SDP(G) parameters to expand $\overline{\mathbf{e}}_G$ into the secret error exponents $\overline{\mathbf{e}}$ using the matrix $\overline{\mathbf{M}}$ at the beginning of the CROSS.SIGN algorithm, before performing the computation of the commitment $\mathtt{cmt}_0[i]$ for t times. This arithmetic operation produces the vector transformation element $\overline{\mathbf{v}}[i] \vee \overline{\mathbf{v}}_G[i]$ and the round syndrome $\mathbf{s}'[i]$ starting from $\overline{\mathbf{e}}'[i] \vee \overline{\mathbf{e}}'_G[i]$ and $\mathbf{u}'[i]$, both generated from the expansion of $\mathtt{seed}[i]$ by the CSPRNG. During the process, each transformed error vector $\overline{\mathbf{e}}'[i]$ is stored in a local memory, to avoid the expensive re-expansion of $\overline{\mathbf{e}}'_G[i]$ via the vector-matrix multiplication specialized for the matrix $\overline{\mathbf{M}}$ in the following arithmetic operation performed by CROSS.SIGN, the computation of the first challenge response. In such operation, executed t times, the first challenge $\mathtt{chall}_1[i]$ and the previously sampled element $\mathbf{u}'[i]$ are used to compute the first challenge response $\mathbf{y}[i]$.

In the CROSS.VERIFY algorithm one of the two verification routines implemented by the arithmetic unit is executed depending on the value of the second challenge $\mathtt{chall}_2[i]$. If it is equal to $\mathtt{0}$, then the arithmetic unit uses the fist challenge value $\mathtt{chall}_1[i]$, the vector transformation element $\overline{\mathbf{v}}[i] \vee \overline{\mathbf{v}}_G[i]$, the syndrome from the public key $\mathbf{s}$, and the first challenge response $\mathbf{y}[i]$ to retrieve the round syndrome $\mathbf{s}'[i]$. If $\mathtt{chall}_2[i] = \mathtt{1}$, then the transformed error vector $\overline{\mathbf{e}}'[i] \vee \overline{\mathbf{e}}'_G[i]$ and the vector $\mathbf{u}'[i]$ are expanded from the $\mathtt{seed}[i]$ by the CSPRNG and provided to the input stream interfaces, along with the first challenge value $\mathtt{chall}_1[i]$. In this case, the arithmetic unit will recompute the first challenge response and provide it to the output interface.

In Table 8.13 is reported the out-of-context synthesis of the CROSS arithmetic unit for the AMD Artix-7 FPGA platform when varying the parameter sets in use and using 64-bits wide AXI streams. In line with the synthesis results of each component presented in section 7.5, the vector-matrix multiplication units take roughly $2/3$ of the occupied area, while the point-wise multiplier units take the majority of the remaining logic area. Only the parameters instantiating the R-SDP(G) hard problem use the DSP units in the first multiplication of the Barrett reduction. The local memories account

for 10 to 14 BRAM resources when using the parameter sets from the fast optimization corner, growing linearly with the parameter t and utilizing up to 42 BRAMs when the parameter set `CROSS-RSDP-5-s` is used. Note that with the exception of BRAMs, the occupation of the remaining resources in the FPGA fabric do not vary depending on the optimization corner. By looking at the maximum working frequency of the arithmetic unit, it is easy to notice that the additional modules introduced by the R-SDP(G) parameters slightly decrease the clock frequency to ≈ 110 MHz when compared to the ≈ 135 MHz reached by designs using R-SDP parameters.

In the remaining section of Table 8.13 are reported the latencies, in clock cycles, of the arithmetic computations performed in the three DS primitives, namely the generation of the private and public keys, the singing of a message, and the verification of the signature. The latencies are expressed as formulas in t and w variables to generalize the latency for every optimization corner, and highlight the cost of each arithmetic operation performed by the arithmetic unit. These results do not consider the cost of the initialization phase when loading the two matrices $\mathbf{V}^\top$ and $\overline{\mathbf{W}}$, and the expansion of the secret error vector $\overline{\mathbf{e}}$. The reason is that the latency cost of the initialization is already accounted in the sampling operation of such elements, as the execution of the two tasks are overlapped. Regarding the latency of the expansion of $\overline{\mathbf{e}}$, such cost of a single expansion execution is only paid by R-SDP(G) parameters and is by far dominated by the t parallel executions of the CROSS-ID protocol, thus having a negligible contribution in the overall sign operation.

The generation of the key pair is extremely fast, particularly for R-SDP(G) parameters, taking from 188 to 1074 clock cycles to generate the syndrome $\mathbf{s}$. Computing each commitment $\texttt{cmt}_0[i]$ takes from 193 to 488 clock cycles using the R-SDP(G) parameters, and from 360 to 1084 clock cycles for the R-SDP ones, and this task is repeated for each one of the t CROSS-ID executions. The computation of the first challenge response $\mathbf{y}[i]$ is only taking few tens of clock cycles thanks to the relatively large size of the streams connecting the modules compared to the size of vector coefficients, and since there are no vector-matrix multiplications in this arithmetic operation. This operation is repeated t times during the signing of the message. The two arithmetic operations involved during the verification have similar latency, with the verification of $\texttt{chall}_2[i] = 1$ taking $\approx 3/4$ the clock cycles of the other verification operation. Recalling that in the CROSS.VERIFY function the second challenge has w out of t times the value `1`, with $\frac{1}{2} < w < t$, the slightly faster verification operation is executed more frequently than the other one.

8.3.2 Merkle and seed trees

An interesting aspect of CROSS is the availability of multiple optimization corners for each parameter set defined the specification, providing a trade-off between the size of the produced signatures, and the latency of the sign and verify operations. Specifically, by unbalancing the second binary challenge in order to transmit within the signature more $\texttt{seed}[i]$ and $\texttt{cmt}_0[i]$ rather than the vector tuple $(\mathbf{y}[i], \overline{\mathbf{v}}[i] \vee \overline{\mathbf{v}}_G[i])$ and $\texttt{cmt}_1[i]$, it

Table 8.14: *Synthesis results for CROSS Merkle tree of* $\texttt{cmt}_0[i]$ *and binary tree containing* `seed[i]` *as leafs when targeting an AMD Artix-7 FPGA and using* 64*-bits word sizes. The signer computes the* SEEDLEAVES, SEEDPATH, TREEROOT, *and* TREEPROOF *routines, while the verifier computes only* REBUILDLEAVES *and* RECOMPUTEROOT. *Credits to Patrick Karl from the Technical University of Munich.*

Parameter set	Resources					Freq.	Latency (CC)	
	LUT	FF	BRAM	DSP	eSlice	MHz	Sign	Verify
CROSS-RSDP-1-f	1072	266	6.5	0	1646	171	4213	2381
CROSS-RSDP-1-b	972	259	6.5	0	1621	178	10536	9649
CROSS-RSDP-1-s	1138	310	28.5	0	6327	153	21376	19099
CROSS-RSDP-3-f	1149	273	12.5	0	2938	135	8439	4943
CROSS-RSDP-3-b	1143	298	28.5	0	6328	131	20075	17141
CROSS-RSDP-3-s	1282	312	28.5	0	6363	135	30261	25120
CROSS-RSDP-5-f	1136	297	28.5	0	6326	154	14055	8402
CROSS-RSDP-5-b	1032	289	28.5	0	6300	150	32455	26303
CROSS-RSDP-5-s	1221	310	56.5	0	12284	148	52610	41117
CROSS-RSDPG-1-f	1067	266	6.5	0	1645	173	3945	2225
CROSS-RSDPG-1-b	972	259	6.5	0	1621	188	10538	9589
CROSS-RSDPG-1-s	1022	280	12.5	0	2906	164	21032	18795
CROSS-RSDPG-3-f	1092	272	12.5	0	2923	148	7937	4664
CROSS-RSDPG-3-b	1207	297	12.5	0	2952	142	14055	12312
CROSS-RSDPG-3-s	1113	293	28.5	0	6321	133	26692	22245
CROSS-RSDPG-5-f	1142	296	28.5	0	6328	154	13105	7807
CROSS-RSDPG-5-b	1148	297	28.5	0	6329	150	22746	18991
CROSS-RSDPG-5-s	1215	313	56.5	0	12282	146	40608	32103

is possible to achieve a reduced signature size since the size of $\texttt{seed}[i]$ is much smaller than $(\mathbf{y}[i], \overline{\mathbf{v}}[i] \vee \overline{\mathbf{v}}_G[i])$. Note that the security loss caused by this unbalanced choice is compensated by increasing the number of parallel executions of the ZK identification scheme CROSS-ID. With the exclusion of the fast parameter set where $w \approx {}^{t}/_{2}t$, the large amount of $\texttt{seed}[i]$ and $\texttt{cmt}_0[i]$ are efficiently transmitted by using the binary expansion tree and the Merkle tree.

Upon loading the per-signature randomness $\texttt{seed}$ at the root of the binary tree, the implemented SEEDLEAVES function computes the tree leafs corresponding to all the $\texttt{seed}[i]$. Afterwards the signer includes a subset of $w < t$ $\texttt{seed}[i]$ in the signature by only selecting the internal tree nodes that are the oldest ancestors to the transmitted $\texttt{seed}[i]$ (SEEDPATH). On the verifier side, the REBUILDLEAVES loads the internal tree nodes transmitted in the signature, and expands them to retrieve the subset of the $\texttt{seed}[i]$ sent by the signer.

The Merkle tree is used in a specular way by loading all the computed commitments $\texttt{cmt}_0[i]$ on the leafs of a binary tree, and computing the value of the parent node up to the root of the tree (TREEROOT). The signer includes in the signature only the internal tree nodes that are the the oldest ancestors of the commitments to be sent (TREEPROOF). The verifier loads the internal nodes when receives the signature, and computes the missing tree leafs, and retrieves the tree root via the RECOMPUTEROOT function.

In Table 8.14 are reported the results of the synthesis targeting the AMD Artix-7 FPGA of the trees module incorporating the designs of both the $\texttt{seed}[i]$ binary tree and the Merkle tree of commitments $\texttt{cmt}_0[i]$. The table then includes the resulting latency of the functions executing the SEEDLEAVES, SEEDPATH, TREEROOT, and TREEPROOF routines on the signer side, and only REBUILDLEAVES and RECOMPUTEROOT on the verifier side.

The trees unit occupies few resources in terms of LUT and FF, with a maximum working frequency > 130 MHz. Each tree has at most $2t-1$ nodes containing either λ or 2λ bits value. Moreover, two side trees containing the metadata for the valid tree nodes are used, where each node has a negligible size. The trees are linearized and stored in a centralized memory, which is composed by Vivado using an amount of BRAMs ranging from 12.5 to 56.5 depending on the parameter sets.

Note that for the fast optimization corner the design does not generate the Merkle tree, and the previously described routines result simplified, justifying the reduced FPGA resource usage, higher maximum working frequency, and lower latency.

8.3.3 Operation scheduling

The top level design mainly consists in three dual-port memories, the arithmetic unit, the sampler unit, and the tree unit. Furthermore, some logic is necessary to compose the signature during the sign operation, and parse it during the verification. All this units are managed by a FSA implementing a schedule of the macro functions implementing the CROSS.KEYGENERATION, CROSS.SING, and CROSS.VERIFY algorithms in a single design.

Given the consistent number of data dependencies in the CROSS scheme, there is not much parallelism to leverage, other than performing multiple CROSS-ID protocols in parallel. However, that would require the replication of the arithmetic unit, the introduction of further memories, and the addition of more SHAKE units, with a considerable increment of the occupied area and complexity of the design. For this design we decided to aim for a streamlined design with competitive performance to provide a base reference for future works.

The generation of the key pair starts by receiving the secret key randomness $\texttt{seed}_{\mathsf{sk}}$ from a TRNG. However, to properly test the designs against the official KATs, we use a deterministic value and store it in the memory. Afterwards, the seed is expanded in $\texttt{seed}_{\mathbf{e}}$ and $\texttt{seed}_{\mathsf{pk}}$ from the sampler unit directly using the output of the SHAKE module. Afterwards, the two matrices $\overline{\mathbf{M}}$ (only for R-SDP(G) parameters) and $\mathbf{V}^\top$ are generated from $\texttt{seed}_{\mathsf{pk}}$ via the sampler unit, and immediately used to initialize the memory with wide data ports contained in the arithmetic unit. The private seed $\texttt{seed}_{\mathbf{e}}$ is then used to generate the element $\overline{\mathbf{e}}_G$ or $\overline{\mathbf{e}}$, depending on the underlying hard problem generated by the parameter sets, and the public syndrome $\mathbf{s}$ can be generated by the arithmetic unit. Finally the private and public keys are read from the memories packed back to save some memory space and comply with the format mandated by the specification.

Regarding the generation of the signature, the secret key, the public matrices, the signature randomness $\texttt{seed}$, and the salt are prepared in memory. Similarly to the secret key randomness, the signature randomness and salt are deterministic values provided by the KATs. The signature randomness $\texttt{seed}$ is expanded using the seed tree via the SEEDLEAVES routine provided by the tree unit. Then the t loop iterations are executed generating the elements $\overline{\mathbf{e}}'_G[i] \vee \overline{\mathbf{e}}'[i]$ and $\mathbf{u}'[i]$ from each leaf $\texttt{seed}[i]$ of the seed tree via the sampler unit, and the arithmetic unit at the same time proceeds in parallel computing the elements $\mathbf{s}'[i]$ and $\overline{\mathbf{v}}_G[i] \vee \overline{\mathbf{v}}[i]$. Those elements are immediately absorbed by the SHAKE unit, yielding the $\texttt{cmt}_0[i]$ wrote in the leafs position of the Merkle tree. Each $\texttt{cmt}_1[i]$ is then generated by the SHAKE module in the sampler unit as the digest of $\texttt{seed}[i] \| \texttt{salt}$, and stored consecutively in memory. The root of the Merkle tree $\texttt{dig}_{\texttt{cmt}_0}$ can now be computed via the TREEROOT routine provided by the tree unit, and the concatenated $\texttt{cmt}_1[i]$ are absorbed by the SHAKE module in the sample unit, producing $\texttt{dig}_{\texttt{cmt}_1}$. The two digest are then concatenated and used to produce the digest $\texttt{dig}_{\texttt{cmt}}$. Using the digest of the message to be signed, the digest of the commitments, and the salt, the first challenges $\texttt{chall}_1[i]$ are generated by the sampler unit, and used by the arithmetic unit to compute the t responses $\mathbf{y}[i]$. Then the sampler unit generates the second challenges with fixed weight w by expanding the digest of all the first challenge responses, and the previous digests, and then the signature can be assembled by picking the internal nodes of the Merkle tree and seed tree via the TREEPROOF and SEEDPATH routines provided by the tree unit, and $w - t$ elements $\mathbf{y}[i]$ and $\overline{\mathbf{v}}_G[i] \vee \overline{\mathbf{v}}_G[i]$, and pack everything to minimize the signature size.

When considering the verification procedure, the signature is loaded and unpacked,

Table 8.15: *Synthesis results for* CROSS.KEYGENERATION, CROSS.SIGN, *and* CROSS.VERIFY *when targeting an AMD Artix-7 FPGA and using* 64*-bits word sizes. The efficiency indicator Area-Time product is expressed in eSlices · ms. Credits to Patrick Karl from the Technical University of Munich.*

Design	Parameter set	Resources					Freq.	KeyGen.		Sign		Verify	
		LUT	FF	BRAM	DSP	eSlice	MHz	μs	AT	μs	AT	μs	AT
This work	CROSS-RSDP-1-f	18852	8382	39.5	0	13087	119	35	467	764	10006	584	7653
	CROSS-RSDP-1-b	19457	8537	52.0	0	15889	113	37	596	1444	22951	1016	16157
	CROSS-RSDP-1-s	19957	8663	106.0	0	27462	115	36	1013	2858	78508	1960	53841
	CROSS-RSDP-3-f	18185	8447	86.5	0	22885	110	82	1886	1905	43609	1455	33310
	CROSS-RSDP-3-b	19002	8640	111.0	0	28283	108	84	2380	3230	91355	2067	58488
	CROSS-RSDP-3-s	19339	8690	137.5	1	34120	105	86	2949	4984	170081	3037	103653
	CROSS-RSDP-5-f	18612	8382	136.5	0	33591	105	153	5170	3762	126370	2843	95514
	CROSS-RSDP-5-b	19339	8556	187.5	1	44720	101	160	7157	6297	281646	3744	167460
	CROSS-RSDP-5-s	21500	8625	265.5	1	61796	101	160	9887	10153	627431	5564	343849
This work	CROSS-RSDPG-1-f	22886	9938	27.5	29	15452	100	10	163	601	9300	477	7376
	CROSS-RSDPG-1-b	23453	9961	36.0	29	17396	103	10	178	1216	21154	886	15421
	CROSS-RSDPG-1-s	23842	10011	62.0	29	23005	102	10	239	2433	55979	1762	40540
	CROSS-RSDPG-3-f	22714	9789	41.5	29	18377	96	22	413	1260	23162	1033	18998
	CROSS-RSDPG-3-b	23610	10104	66.0	29	23795	102	21	503	1604	38185	1201	28593
	CROSS-RSDPG-3-s	23590	10052	120.5	29	35344	99	21	770	3103	109699	2214	78267
	CROSS-RSDPG-5-f	22888	10085	85.5	29	27749	99	35	997	2223	61701	1860	51637
	CROSS-RSDPG-5-b	23653	10223	116.5	29	34512	96	37	1278	2960	102183	2222	76717
	CROSS-RSDPG-5-s	24053	10282	176.5	29	47332	98	36	1715	5133	242993	3613	171042
[BNG24]	Dilithium-II							12.2	260	98.3	2096	14.4	307
	Dilithium-III	48600	30000	22.5	32	21330	185	22.6	482	166.7	3555	25.4	541
	Dilithium-V							30.3	646	196.3	4187	33.2	708
[LSG21]	Dilithium-II	27433	10681	15.0	45	16091	163	115	1850	178/470	2864/7562	121	1947
	Dilithium-III	30900	11372	21.0	45	18230	145	228	4156	310/850	5651/15495	221	4028
	Dilithium-V	44653	13814	31.0	45	23788	140	363	8635	503/1042	11965/24787	377	8968
[BNG21]	Dilithium-V	53187	28318	29.0	16	21597	116	121	2613	2520	54424	21	453
[Ami+20]	SPHINCS+-128f-s	47991	72505	11.5	1	14571	250/500	–	–	1010	14716	16	233
	SPHINCS+-192f-s	48398	73476	17.0	1	15838	250/500	–	–	1170	18530	19	300
	SPHINCS+-256f-s	51009	74539	22.5	1	17657	250/500	–	–	2520	44495	21	370

the public matrices are generated and loaded in the arithmetic unit, and the $w - t$ elements $\mathbf{y}[i]$ and $\overline{\mathbf{v}}_G[i] \vee \overline{\mathbf{v}}_G[i]$ transmitted in the signature are unpacked. Afterwards, the two challenges $\texttt{chall}_1[i]$ and $\texttt{chall}_2[i]$ are recomputed, and the transmitted seed leafs are expanded via the REBUILDLEAVES routine of the tree unit. Depending on each value of $\texttt{chall}_2[i]$, an appropriate sub-routine of the arithmetic unit is called to recompute the missing $\texttt{cmt}_0[i]$ or $\texttt{cmt}_1[i]$, repeating the operation for each one of the t parallel CROSS-ID protocols. Finally, $\texttt{cmt}_0$ is reconstructed via the RECOMPUTEROOT routine from the tree unit, $\texttt{cmt}_1$ is produced as the digest of the concatenated $\texttt{cmt}_1[i]$, and the digest of these two is verified to match against the one in the signature.

8.3.4 Design synthesis and implementation

In Table 8.15 are reported the results of the synthesis for the AMD Artix-7 FPGA of the top-level design modules specialized for each parameter set. Considering that the SHAKE core within the sampler unit takes approximately 6500 LUT and 2700 FF, this unit accounts for $\approx 30\%$ of the LUT and FF used by the top-level design. Sim-

ilarly, the arithmetic unit contributes for 23% to 33% to the area occupation in the eSlice metric, depending on the considered parameter set. Recalling that the arithmetic unit works many times coupled with the sampler unit, the arithmetic computations during the signature generation take from 43% (`CROSS-RSDPG-1-s`) to 91% (`CROSS-RSDP-5-f`) of the overall clock cycle count of the whole primitive, and from 42% (`CROSS-RSDPG-1-s`) to 72% (`CROSS-RSDP-5-f`) of the total number of clock cycles used to verify a signature. The tree unit has a negligible cost when compared with the whole top-level design, and takes from 3 to 10 % of the clock cycle count during the sign and verification operations. Unsurprisingly for a code-based digital signature scheme, CROSS requires a considerable amount of BRAM, which limits the compatibility of this top-level design on some products of the AMD Artix-7 product family due to insufficient BRAM resources offered. For some reference, the low-budget `XC7A50T` chip could fit 7 out of 18 designs, the widely available `XC7A100T` chip cannot fit 5 design, and only the top-of-the-line `XC7A200T` chip is compatible with every solution. When compared to the designs for R-SDP parameters, the R-SDP(G) parameters yield designs with lower requirements on BRAMs, although taking $\approx 20\%$ more LUT and FF, requiring 29 DSP units, and having a lower maximum operating frequency. The key generation operation takes from 10 to 160 μs depending on the parameter set, whereas the time required to generate a signature instead ranges from 600 μs to 10 ms, with the verification of the signature procedure requiring from 500 μs to 5.5 ms.

When comparing the CROSS design with other hardware accelerators for post-quantum digital signatures such as CRYSTALS-Dilithium and SPHINCS+ (respectively standardized as ML-DSA and SLH-DSA), we can see that the designed accelerator has a conservative consumption of LUT and FF resources, particularly when considering the parameter sets guaranteeing the highest security level. Considering the eSlice area occupation metric, the CROSS designs specialized for parameter sets offering the lowest security level, with the exclusion of the small optimization corner, represent the solutions with the most compact area. However, the consistent amount of BRAM used by the proposed designs rapidly inverts the trend when considering the small optimization corners and parameters with higher security margins, leading to solutions up to $3.5\times$ larger than the CRYSTALS-Dilithium and SPHINCS+ counterparts. The key generation operation is competitive with the accelerators for CRYSTALS-Dilithium, but the sign operation is from $7\times$ to $50\times$ slower. Moreover, the CROSS signature verification is one to two orders of magnitude slower with respect to the other two post-quantum digital signatures.

All things considered, the reader should also consider that the provided designs have not leveraged yet any parallelism offered by the multiple executions of the CROSS-ID scheme. Additionally, this first analysis allowed us to determine which components to focus our future efforts on to improve latency and efficiency.

CHAPTER 9

Conclusions

This thesis presents the analysis, design, and optimization of hardware accelerators for three post-quantum cryptographic primitives: the lattice-based KEM NTRU, the code-based KEM HQC, and the code-based DS CROSS. Through the development of these hardware modules, the thesis highlights how a detailed analysis of data parallelisms offered both within the algorithms implementing the required operations and at the higher level of the scheduling of such operations in a top-level design can lead to significant performance and efficiency improvements, both in terms of latency and occupied silicon area.

The cryptosystems were introduced with a uniform notation to clearly identify the operations that make up each algorithm and determine the common dependencies, such as the Keccak family of hash functions, and identify the critical operations to develop tailored hardware modules that were optimized for the unique characteristics of each scheme. This approach ensured a systematic method for designing the hardware components and facilitated the understanding of how each step of the cryptographic process could be accelerated.

The data parallelisms in the algorithms of the operations composing the cryptoschemes were benchmarked through the use of DSE to pinpoint optimal configurations for minimizing latency and maximizing the efficiency of the hardware designs. Analyzing the possible schedules of such operations (i.e. the generation of cryptographic elements, arithmetic operations, and the computation of the hash digests) while considering the data dependencies and the constraints imposed by the type and quantity of employed

memories allowed their parallel execution, leading to a significant reduction in the overall execution time of each cryptosystem. In particular, for the code-based HQC it was presented an optimization of the schedule through the re-ordering of the generated cryptographic elements, producing a new opportunity of parallelism during the key generation and the encryption, and a cycle count reduction of the decryption without having security implications. The importance of this optimization was also recognized by the authors of the cryptographic scheme, who adopted it in the latest revision of the official specification.

In conclusion, the work presented in this thesis contributes to the ongoing effort to develop efficient and secure post-quantum cryptographic solutions. In particular, the hardware accelerators developed in this research represent a step towards making post-quantum cryptographic schemes more practical for real-world applications by focusing on both the performance and efficiency of NTRU, HQC, and CROSS through hardware acceleration. Building on the developed hardware accelerators presented in this thesis, two orthogonal research lines can further investigate strategies and optimizations for power or area efficient solutions, and determine side-channel security of the countermeasures with the lowest overhead in terms of silicon area and runtime latency.

Bibliography

[ABP25] Francesco Antognazza, Alessandro Barenghi, and Gerardo Pelosi. "An Efficient and Unified RTL Accelerator Design for HQC-128, HQC-192, and HQC-256". In: *IEEE Transactions on Computers* (2025), pp. 1–14. DOI: 10.1109/TC.2025.3558044.

[Ach+24] Rajeev Acharya et al. "Quantum error correction below the surface code threshold". en. In: *Nature* (Dec. 2024). Publisher: Nature Publishing Group, pp. 1–3. ISSN: 1476-4687. DOI: 10.1038/s41586-024-08449-y. URL: https://www.nature.com/articles/s41586-024-08449-y (visited on 01/10/2025).

[AD97] Miklós Ajtai and Cynthia Dwork. "A Public-Key Cryptosystem with Worst-Case/Average-Case Equivalence". In: *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*. Ed. by Frank Thomson Leighton and Peter W. Shor. ACM, 1997, pp. 284–293. DOI: 10.1145/258533.258604. URL: https://doi.org/10.1145/258533.258604.

[Ae22] Carlos Aguilar Melchor and et al. "Towards Automating Cryptographic Hardware Implementations: A Case Study of HQC". In: *CBCrypto 2022, Trondheim, Norway, May 29-30, 2022*. Vol. 13839. LCNS. Springer, 2022, pp. 62–76. DOI: 10.1007/978-3-031-29689-5_4. URL: https://doi.org/10.1007/978-3-031-29689-5_4.

[Agu+24a] Carlos Aguilar Melchor et al. *HQC Documentation*. [Online]. Available from: https://web.archive.org/web/20250128090526/https://pqc-hqc.org/doc/hqc-specification_2024-10-30.pdf, (Archived on 28 Jan. 2025). 2024. URL: https://pqc-hqc.org/doc/hqc-specification_2024-10-30.pdf.

[Agu+24b] Carlos Aguilar-Melchor et al. "Efficient error-correcting codes for the HQC post-quantum cryptosystem". In: *Designs, Codes and Cryptography* (Oct. 9, 2024). ISSN: 1573-7586. DOI: 10.1007/s10623-024-01507-6. URL: https://doi.org/10.1007/s10623-024-01507-6 (visited on 10/14/2024).

[Aja+19] Tutu Ajayi et al. "Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project". In: *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*. ACM, 2019, p. 76. DOI: 10.1145/3316781.3326334. URL: https://doi.org/10.1145/3316781.3326334.

[Ala+19] Gorjan Alagic et al. *Status report on the first round of the NIST post-quantum cryptography standardization process*. Jan. 2019. DOI: 10.6028/nist.ir.8240. URL: http://dx.doi.org/10.6028/NIST.IR.8240.

[Ala+22a] Gorjan Alagic et al. *Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process*. 2022. DOI: 10.6028/NIST.IR.8413-upd1. URL: https://doi.org/10.6028/NIST.IR.8413-upd1.

[Ala+22b] Gorjan Alagic et al. *Status report on the third round of the NIST Post-Quantum Cryptography Standardization process*. Sept. 2022. DOI: 10.6028/nist.ir.8413-upd1. URL: http://dx.doi.org/10.6028/NIST.IR.8413-upd1.

[Ala+25a] Gorjan Alagic et al. *Recommendations for Key-Encapsulation Mechanisms*. Jan. 2025. DOI: 10.6028/nist.sp.800-227.ipd. URL: http://dx.doi.org/10.6028/NIST.SP.800-227.ipd.

[Ala+25b] Gorjan Alagic et al. *Status Report on the Fourth Round of the NIST Post-Quantum Cryptography Standardization Process*. Mar. 2025. DOI: 10.6028/nist.ir.8545. URL: http://dx.doi.org/10.6028/NIST.IR.8545.

[Ami+20] Dorian Amiet et al. "FPGA-based SPHINCS$^+$ Implementations: Mind the Glitch". In: *23rd Euromicro Conference on Digital System Design, DSD 2020, Kranj, Slovenia, August 26-28, 2020*. IEEE, 2020, pp. 229–237. DOI: 10.1109/DSD51259.2020.00046. URL: https://doi.org/10.1109/DSD51259.2020.00046.

[ANS23] ANSSI. *ANSSI views on the Post-Quantum Cryptography transition (2023 follow up)*. [Online]. Available from: https://web.archive.org/web/20240127124757/https://cyber.gouv.fr/sites/default/files/document/follow_up_position_paper_on_post_quantum_cryptography.pdf, (Archived on 7 Aug. 2024). 2023. URL: https://cyber.gouv.fr/sites/default/files/document/follow_up_position_paper_on_post_quantum_cryptography.pdf.

[Ant+23a] Francesco Antognazza et al. "A Flexible ASIC-Oriented Design for a Full NTRU Accelerator". In: *Proceedings of the 28th Asia and South Pacific Design Automation Conference, ASPDAC 2023, Tokyo, Japan, January 16-19, 2023*. Ed. by Atsushi Takahashi. ACM, 2023, pp. 591–597. DOI: 10.1145/3566097.3567916. URL: https://doi.org/10.1145/3566097.3567916.

[Ant+23b] Francesco Antognazza et al. "An Efficient Unified Architecture for Polynomial Multiplications in Lattice-Based Cryptoschemes". In: *Proceedings of the 9th International Conference on Information Systems Security and Privacy, ICISSP 2023, Lisbon, Portugal, February 22-24, 2023*. Ed. by Paolo Mori, Gabriele Lenzini, and Steven Furnell. SciTePress, 2023, pp. 81–88. DOI: 10.5220/0011654200003405. URL: https://doi.org/10.5220/0011654200003405.

[Ant+24a] Francesco Antognazza et al. "A High Efficiency Hardware Design for the Post-Quantum KEM HQC". In: *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2024, Tysons Corner, VA, USA, May 6-9, 2024*. IEEE, 2024, pp. 431–441. DOI: 10.1109/HOST55342.2024.10545409. URL: https://doi.org/10.1109/HOST55342.2024.10545409.

[Ant+24b] Francesco Antognazza et al. “A Versatile and Unified HQC Hardware Accelerator”. In: *Applied Cryptography and Network Security Workshops - ACNS 2024 Satellite Workshops, AIBlock, AIHWS, AIoTS, SCI, AAC, SiMLA, LLE, and CIMSS, Abu Dhabi, United Arab Emirates, March 5-8, 2024, Proceedings, Part II*. Ed. by Martin Andreoni. Vol. 14587. Lecture Notes in Computer Science. Springer, 2024, pp. 214–219. DOI: 10.1007/978-3-031-61489-7_17. URL: https://doi.org/10.1007/978-3-031-61489-7_17.

[Ant+24c] Francesco Antognazza et al. “Performance and Efficiency Exploration of Hardware Polynomial Multipliers for Post-Quantum Lattice-Based Cryptosystems”. In: *SN Comput. Sci.* 5.2 (2024), p. 212. DOI: 10.1007/S42979-023-02547-W. URL: https://doi.org/10.1007/s42979-023-02547-w.

[Ara+22] Nicolas Aragon et al. *BIKE Supporting Documentation*. [Online]. Available from: https://web.archive.org/web/20231221165734/https://bikesuite.org/files/v5.0/BIKE_Spec.2022.10.10.1.pdf, (Archived on 21 Dec. 2023). 2022. URL: https://bikesuite.org/files/v5.0/BIKE_Spec.2022.10.10.1.pdf.

[Bal+18] Marco Baldi et al. “LEDAkem: A Post-quantum Key Encapsulation Mechanism Based on QC-LDPC Codes”. In: *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings*. Ed. by Tanja Lange and Rainer Steinwandt. Vol. 10786. Lecture Notes in Computer Science. Springer, 2018, pp. 3–24. DOI: 10.1007/978-3-319-79063-3_1. URL: https://doi.org/10.1007/978-3-319-79063-3_1.

[Bal+20] Marco Baldi et al. “A New Path to Code-based Signatures via Identification Schemes with Restricted Errors”. In: *CoRR* abs/2008.06403 (2020). arXiv: 2008.06403. URL: https://arxiv.org/abs/2008.06403.

[Bal+24a] Marco Baldi et al. *CROSS Documentation*. [Online]. Available from: https://web.archive.org/web/20250122122245/https://www.cross-crypto.com/CROSS_Specification_v1.2.pdf, (Archived on 22 Jan. 2025). 2024. URL: https://www.cross-crypto.com/CROSS_Specification_v1.2.pdf.

[Bal+24b] Marco Baldi et al. “Zero Knowledge Protocols and Signatures from the Restricted Syndrome Decoding Problem”. In: *Public-Key Cryptography - PKC 2024 - 27th IACR International Conference on Practice and Theory of Public-Key Cryptography, Sydney, NSW, Australia, April 15-17, 2024, Proceedings, Part II*. Ed. by Qiang Tang and Vanessa Teague. Vol. 14602. Lecture Notes in Computer Science. Springer, 2024, pp. 243–274. DOI: 10.1007/978-3-031-57722-2_8. URL: https://doi.org/10.1007/978-3-031-57722-2_8.

[Bar+24] Elaine Barker et al. *Recommendation for Random Bit Generator (RBG) Constructions*. 2024. DOI: 10.6028/nist.sp.800-90c.4pd. URL: http://dx.doi.org/10.6028/NIST.SP.800-90C.4pd.

[Bar86] Paul Barrett. “Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor”. In: *Advances in Cryptology - CRYPTO ’86, Santa Barbara, California, USA, 1986, Proceedings*. Ed. by Andrew M. Odlyzko. Vol. 263. Lecture Notes in Computer Science. Springer, 1986, pp. 311–323. DOI: 10.1007/3-540-47721-7_24. URL: https://doi.org/10.1007/3-540-47721-7_24.

[Bas+10] L E Bassham et al. *A statistical test suite for random and pseudorandom number generators for cryptographic applications*. 2010. DOI: 10.6028/nist.sp.800-22r1a. URL: http://dx.doi.org/10.6028/NIST.SP.800-22r1a.

[BCD20] Elaine Barker, Lily Chen, and Richard Davis. *Recommendation for Key-Derivation Methods in Key-Establishment Schemes*. Aug. 2020. DOI: 10.6028/nist.sp.800-56cr2. URL: http://dx.doi.org/10.6028/NIST.SP.800-56Cr2.

[Bec+22a] Hanno Becker et al. "Efficient Multiplication of Somewhat Small Integers Using Number-Theoretic Transforms". In: *Advances in Information and Computer Security - 17th International Workshop on Security, IWSEC 2022, Tokyo, Japan, August 31 - September 2, 2022, Proceedings*. Ed. by Chen-Mou Cheng and Mitsuaki Akiyama. Vol. 13504. Lecture Notes in Computer Science. Springer, 2022, pp. 3–23. DOI: 10.1007/978-3-031-15255-9_1. URL: https://doi.org/10.1007/978-3-031-15255-9_1.

[Bec+22b] Hanno Becker et al. "Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.1 (2022), pp. 221–244. DOI: 10.46586/TCHES.V2022.I1.221-244. URL: https://doi.org/10.46586/tches.v2022.i1.221-244.

[Ber+11] Guido Bertorni et al. *The Keccak reference*. [Online]. Available from: https://web.archive.org/web/20240128114601/https://keccak.team/files/Keccak-reference-3.0.pdf, (Archived on 28 Jan. 2024). 2011. URL: https://keccak.team/files/Keccak-reference-3.0.pdf.

[Ber+12] Guido Bertorni et al. *Keccak implementation overview*. [Online]. Available from: https://web.archive.org/web/20240502130759/https://keccak.team/files/Keccak-implementation-3.2.pdf, (Archived on 02 May 2024). 2012. URL: https://keccak.team/files/Keccak-implementation-3.2.pdf.

[Ber+22] Daniel J. Bernstein et al. *Classic McEliece Supporting Documentation*. [Online]. Available from: https://web.archive.org/web/20231218203726/https://classic.mceliece.org/mceliece-spec-20221023.pdf, (Archived on 18 Dec. 2023). 2022. URL: https://classic.mceliece.org/mceliece-spec-20221023.pdf.

[Ber+24] Daniel J. Bernstein et al. *Report on evaluation of KpqC Round-2 candidates*. Cryptology ePrint Archive, Paper 2024/2077. 2024. URL: https://eprint.iacr.org/2024/2077.

[Beu22] Ward Beullens. "Breaking Rainbow Takes a Weekend on a Laptop". In: *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part II*. Ed. by Yevgeniy Dodis and Thomas Shrimpton. Vol. 13508. Lecture Notes in Computer Science. Springer, 2022, pp. 464–479. DOI: 10.1007/978-3-031-15979-4_16. URL: https://doi.org/10.1007/978-3-031-15979-4_16.

[BMT78] Elwyn R. Berlekamp, Robert J. McEliece, and Henk C. A. van Tilborg. "On the inherent intractability of certain coding problems (Corresp.)" In: *IEEE Trans. Inf. Theory* 24.3 (1978), pp. 384–386. DOI: 10.1109/TIT.1978.1055873. URL: https://doi.org/10.1109/TIT.1978.1055873.

[BNG21] Luke Beckwith, Duc Tri Nguyen, and Kris Gaj. "High-Performance Hardware Implementation of CRYSTALS-Dilithium". In: *International Conference on Field-Programmable Technology, (IC)FPT 2021, Auckland, New Zealand, December 6-10, 2021*. IEEE, 2021, pp. 1–10. DOI: 10.1109/ICFPT52863.2021.9609917. URL: https://doi.org/10.1109/ICFPT52863.2021.9609917.

[BNG24] Luke Beckwith, Duc Tri Nguyen, and Kris Gaj. "Hardware Accelerators for Digital Signature Algorithms Dilithium and FALCON". In: *IEEE Des. Test* 41.5 (2024), pp. 27–35. DOI: 10.1109/MDAT.2023.3305156. URL: https://doi.org/10.1109/MDAT.2023.3305156.

[Bod07] Marco Bodrato. "Towards Optimal Toom-Cook Multiplication for Univariate and Multivariate Polynomials in Characteristic 2 and 0". In: *Arithmetic of Finite Fields, First International Workshop, WAIFI 2007, Madrid, Spain, June 21-22, 2007, Proceedings*. Ed. by Claude Carlet and Berk Sunar. Vol. 4547. Lecture Notes in Computer Science. Springer, 2007, pp. 116–133. DOI: 10.1007/978-3-540-73074-3_10. URL: https://doi.org/10.1007/978-3-540-73074-3_10.

[BR21] Andrea Basso and Sujoy Sinha Roy. "Optimized Polynomial Multiplier Architectures for Post-Quantum KEM Saber". In: *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*. IEEE, 2021, pp. 1285–1290. DOI: 10.1109/DAC18074.2021.9586219. URL: https://doi.org/10.1109/DAC18074.2021.9586219.

[BS86] Yair Be'ery and Jakov Snyders. "Optimal soft decision block decoders based on fast Hadamard transform". In: *IEEE Trans. Inf. Theory* 32.3 (1986), pp. 355–364. DOI: 10.1109/TIT.1986.1057189. URL: https://doi.org/10.1109/TIT.1986.1057189.

[BSI24] BSI. *BSI TR-02102-1: "Cryptographic Mechanisms: Recommendations and Key Lengths" Version: 2024-1*. [Online]. Available from: https://web.archive.org/web/20240807092324/https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf, (Archived on 7 Aug. 2024). 2024. URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf.

[BSI25] BSI. *Study: Status of quantum computer development V2.1*. [Online]. Available from: https://web.archive.org/web/20250103231418/https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/Quantencomputer/Entwicklungstand_QC_V_2_1.html, (Archived on 7 Jan. 2025). Jan. 2025. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/Quantencomputer/Entwicklungstand_QC_V_2_1.html.

[Car] Jose Cardona. "Fundamentals of Lattice-Based Cryptography". [Online]. Available from: https://web.archive.org/save/https://github.com/jmcardon/Lattices_Talk_LC2019/raw/master/Fun_WithLattices.pdf, (Archived on 12 Feb. 2025). URL: https://github.com/jmcardon/Lattices_Talk_LC2019/raw/master/Fun_WithLattices.pdf.

[Cas23] Davide Castelvecchi. "IBM releases first-ever 1,000-qubit quantum chip". In: *Nature* 624.7991 (2023), pp. 238–238. DOI: 10.1038/d41586-023-03854-1. URL: https://doi.org/10.1038/d41586-023-03854-1.

[CD23] Wouter Castryck and Thomas Decru. "An Efficient Key Recovery Attack on SIDH". In: *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part V*. Ed. by Carmit Hazay and Martijn Stam. Vol. 14008. Lecture Notes in Computer Science. Springer, 2023, pp. 423–447. DOI: 10.1007/978-3-031-30589-4_15. URL: https://doi.org/10.1007/978-3-031-30589-4_15.

[CGF21] Ana Covic, Fatemeh Ganji, and Domenic Forte. "Circuit Masking: From Theory to Standardization, A Comprehensive Survey for Hardware Security Researchers and Practitioners". In: *CoRR* abs/2106.12714 (2021). arXiv: 2106.12714. URL: https://arxiv.org/abs/2106.12714.

[Che+19] Cong Chen et al. *NTRU: Algorithm Specifications And Supporting Documentation*. [Online]. Available from: https://web.archive.org/web/20240913141001/https://ntru.org/f/ntru-20190330.pdf, (Archived on 13 Sep. 2024). 2019. URL: https://ntru.org/f/ntru-20190330.pdf.

[Che+22] Po-Jen Chen et al. "Complete and Improved FPGA Implementation of Classic McEliece". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.3 (2022), pp. 71–113. DOI: 10.46586/TCHES.V2022.I3.71-113. URL: https://doi.org/10.46586/tches.v2022.i3.71-113.

[Chu+21] Chi-Ming Marvin Chung et al. "NTT Multiplication for NTT-unfriendly Rings New Speed Records for Saber and NTRU on Cortex-M4 and AVX2". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.2 (2021), pp. 159–188. DOI: 10.46586/TCHES.V2021.I2.159-188. URL: https://doi.org/10.46586/tches.v2021.i2.159-188.

[Com90] P. G. Comba. "Exponentiation cryptosystems on the IBM PC". In: *IBM Systems Journal* 29.4 (1990), pp. 526–538. DOI: 10.1147/sj.294.0526.

[Coo+20] David A. Cooper et al. *Recommendation for Stateful Hash-Based Signature Schemes*. Oct. 2020. DOI: 10.6028/nist.sp.800-208. URL: http://dx.doi.org/10.6028/NIST.SP.800-208.

[Cro+23] Luca Crocetti et al. "Review of Methodologies and Metrics for Assessing the Quality of Random Number Generators". In: *Electronics* 12.3 (2023). ISSN: 2079-9292. DOI: 10.3390/electronics12030723. URL: https://www.mdpi.com/2079-9292/12/3/723.

[CVA10] Pierre-Louis Cayrel, Pascal Véron, and Sidi Mohamed El Yousfi Alaoui. "A Zero-Knowledge Identification Scheme Based on the q-ary Syndrome Decoding Problem". In: *Selected Areas in Cryptography - 17th International Workshop, SAC 2010, Waterloo, Ontario, Canada, August 12-13, 2010, Revised Selected Papers*. Ed. by Alex Biryukov, Guang Gong, and Douglas R. Stinson. Vol. 6544. Lecture Notes in Computer Science. Springer, 2010, pp. 171–186. DOI: 10.1007/978-3-642-19574-7_12. URL: https://doi.org/10.1007/978-3-642-19574-7_12.

[Dae17] Joan Daemen. "Changing of the Guards: A Simple and Efficient Method for Achieving Uniformity in Threshold Sharing". In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 137–153. DOI: 10.1007/978-3-319-66787-4_7. URL: https://doi.org/10.1007/978-3-319-66787-4%5C_7.

[Des+23] Sanjay Deshpande et al. "Fast and Efficient Hardware Implementation of HQC". In: *Selected Areas in Cryptography - SAC 2023 - 30th International Conference, Fredericton, Canada, August 14-18, 2023, Revised Selected Papers*. Ed. by Claude Carlet, Kalikinkar Mandal, and Vincent Rijmen. Vol. 14201. Lecture Notes in Computer Science. Springer, 2023, pp. 297–321. DOI: 10.1007/978-3-031-53368-6_15. URL: https://doi.org/10.1007/978-3-031-53368-6_15.

[DMG21] Viet Ba Dang, Kamyar Mohajerani, and Kris Gaj. "High-Speed Hardware Architectures and FPGA Benchmarking of CRYSTALS-Kyber, NTRU, and Saber". In: *IACR Cryptol. ePrint Arch.* (2021), p. 1508. URL: https://eprint.iacr.org/2021/1508.

[DMG23] Viet Ba Dang, Kamyar Mohajerani, and Kris Gaj. "High-Speed Hardware Architectures and FPGA Benchmarking of CRYSTALS-Kyber, NTRU, and Saber". In: *IEEE Trans. Computers* 72.2 (2023), pp. 306–320. DOI: 10.1109/TC.2022.3222954. URL: https://doi.org/10.1109/TC.2022.3222954.

[EBB23] Donald L. Evans, Phillip J. Bond, and Karen H. Brown. *Advanced Encryption Standard (AES)*. NIST standard FIPS 197. May 2023. DOI: 10.6028/nist.fips.197-upd1. URL: http://dx.doi.org/10.6028/NIST.FIPS.197-upd1.

[ETS17] ETSI. *GR QSC 006 Quantum-Safe Cryptography (QSC); Limits to Quantum Computing applied to symmetric key sizes*. [Online]. Available from: hhttps://web.archive.org/web/20241009041101/https://www.etsi.org/deliver/etsi_gr/QSC/001_099/006/01.01.01_60/gr_QSC006v010101p.pdf, (Archived on 13 Feb. 2025). 2017. URL: https://portal.etsi.org/webapp/WorkProgram/Report_WorkItem.asp?WKI_ID=49740.

[ETS20] ETSI. *TS 103 744 Quantum-safe Hybrid Key Exchanges*. [Online]. Available from: https://web.archive.org/web/20240702014511/https://www.etsi.org/deliver/etsi_ts/103700_103799/103744/01.01.01_60/ts_103744v010101p.pdf, (Archived on 7 Aug. 2024). 2020. URL: https://www.etsi.org/deliver/etsi_ts/103700_103799/103744/01.01.01_60/ts_103744v010101p.pdf.

[Far+19] Farnoud Farahmand et al. "Evaluating the Potential for Hardware Acceleration of Four NTRU-Based Key Encapsulation Mechanisms Using Software/Hardware Codesign". In: *Post-Quantum Cryptography - 10th International Conference, PQCrypto 2019, Chongqing, China, May 8-10, 2019 Revised Selected Papers*. Ed. by Jintai Ding and Rainer Steinwandt. Vol. 11505. Lecture Notes in Computer Science. Springer, 2019, pp. 23–43. DOI: 10.1007/978-3-030-25510-7_2. URL: https://doi.org/10.1007/978-3-030-25510-7_2.

[FH15] Haining Fan and M. Anwar Hasan. "A survey of some recent bit-parallel GF(2^n) multipliers". In: *Finite Fields Their Appl.* 32 (2015), pp. 5–43. DOI: 10.1016/j.ffa.2014.10.008. URL: https://doi.org/10.1016/j.ffa.2014.10.008.

[For65] George David Forney. "Concatenated codes." eng. Thesis. Massachusetts Institute of Technology, 1965. DOI: 1721.1/13449. URL: https://doi.org/1721.1/13449 (visited on 10/16/2024).

[FOS19] FOSSi Foundation. *cocotb - coroutine-based cosimulation testbench environment for verifying VHDL and SystemVerilog RTL using Python*. [Online]. Available from: https://web.archive.org/web/20250130130459/https://www.cocotb.org/, (Archived on 30 Jan. 2025). 2019. URL: https://www.cocotb.org.

[FS86] Amos Fiat and Adi Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*. Ed. by Andrew M. Odlyzko. Vol. 263. Lecture Notes in Computer Science. Springer, 1986, pp. 186–194. DOI: 10.1007/3-540-47721-7_12. URL: https://doi.org/10.1007/3-540-47721-7_12.

[Gab05] Philippe Gaborit. "Shorter keys for code based cryptography". In: *Proceedings of the 2005 International Workshop on Coding and Cryptography (WCC 2005)*. 2005, pp. 81–91.

[Gal+22] Andrea Galimberti et al. “FPGA implementation of BIKE for quantum-resistant TLS”. In: *25th Euromicro Conference on Digital System Design (DSD 2022), Maspalomas, Spain, August 31 - Sept. 2, 2022*. IEEE, 2022, pp. 539–547. DOI: 10.1109/DSD57027.2022.00078. URL: https://doi.org/10.1109/DSD57027.2022.00078.

[GE21] Craig Gidney and Martin Ekerå. “How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits”. In: *Quantum* 5 (2021), p. 433. DOI: 10.22331/Q-2021-04-15-433. URL: https://doi.org/10.22331/q-2021-04-15-433.

[GGH97] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. “Public-Key Cryptosystems from Lattice Reduction Problems”. In: *Advances in Cryptology - CRYPTO ’97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*. Ed. by Burton S. Kaliski Jr. Vol. 1294. Lecture Notes in Computer Science. Springer, 1997, pp. 112–131. DOI: 10.1007/BFB0052231. URL: https://doi.org/10.1007/BFb0052231.

[GLK22] Wenbo Guo, Shuguo Li, and Liang Kong. “An Efficient Implementation of KYBER”. In: *IEEE Trans. Circuits Syst. II Express Briefs* 69.3 (2022), pp. 1562–1566. DOI: 10.1109/TCSII.2021.3103184. URL: https://doi.org/10.1109/TCSII.2021.3103184.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. “Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order”. In: *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*. Ed. by Begül Bilgin, Svetla Nikova, and Vincent Rijmen. ACM, 2016, p. 3. DOI: 10.1145/2996366.2996426. URL: https://doi.org/10.1145/2996366.2996426.

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. “Trapdoors for hard lattices and new cryptographic constructions”. In: *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*. Ed. by Cynthia Dwork. ACM, 2008, pp. 197–206. DOI: 10.1145/1374376.1374407. URL: https://doi.org/10.1145/1374376.1374407.

[Gro96] Lov K. Grover. “A Fast Quantum Mechanical Algorithm for Database Search”. In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*. Ed. by Gary L. Miller. ACM, 1996, pp. 212–219. DOI: 10.1145/237814.237866. URL: https://doi.org/10.1145/237814.237866.

[Guo+22] Qian Guo et al. “Don’t Reject This: Key-Recovery Timing Attacks Due to Rejection-Sampling in HQC and BIKE”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.3 (2022), pp. 223–263. DOI: 10.46586/tches.v2022.i3.223-263. URL: https://doi.org/10.46586/tches.v2022.i3.223-263.

[Gut+16] Matthew R. Guthaus et al. “OpenRAM: an open-source memory compiler”. In: *Proceedings of the 35th International Conference on Computer-Aided Design, ICCAD 2016, Austin, TX, USA, November 7-10, 2016*. Ed. by Frank Liu. ACM, 2016, p. 93. DOI: 10.1145/2966986.2980098. URL: https://doi.org/10.1145/2966986.2980098.

[Har15] Michael Hartmann. “The Ajtai-Dwork Cryptosystem and Other Cryptosystems Based on Lattices”. Master’s thesis. University of Zurich, 2015.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. "A Modular Analysis of the Fujisaki-Okamoto Transformation". In: *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I*. Ed. by Yael Kalai and Leonid Reyzin. Vol. 10677. Lecture Notes in Computer Science. Springer, 2017, pp. 341–371. DOI: 10.1007/978-3-319-70500-2_12. URL: https://doi.org/10.1007/978-3-319-70500-2_12.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. "NTRU: A Ring-Based Public Key Cryptosystem". In: *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings*. Ed. by Joe Buhler. Vol. 1423. Lecture Notes in Computer Science. Springer, 1998, pp. 267–288. DOI: 10.1007/BFB0054868. URL: https://doi.org/10.1007/BFb0054868.

[HTX23] Pengzhou He, Yazheng Tu, and Jiafeng Xie. "LOCS: LOw-Latency and ConStant-Timing Implementation of Fixed-Weight Sampler for HQC". In: *IEEE International Symposium on Circuits and Systems, ISCAS 2023, Monterey, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 1–5. DOI: 10.1109/ISCAS46773.2023.10181319. URL: https://doi.org/10.1109/ISCAS46773.2023.10181319.

[Hül+17] Andreas Hülsing et al. "High-Speed Key Encapsulation from NTRU". In: *CHES 2017*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. LNCS. 2017, pp. 232–252. DOI: 10.1007/978-3-319-66787-4_12. URL: https://doi.org/10.1007/978-3-319-66787-4_12.

[IEE20] IEEE. "IEEE Standard for Universal Verification Methodology Language Reference Manual". In: *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)* (2020), pp. 1–458. DOI: 10.1109/IEEESTD.2020.9195920.

[ISE] ISE Crypto PQC working group. *Securing tomorrow today: Why Google now protects its internal communications from quantum threats*. [Online]. Available from: https://web.archive.org/web/20250203232759/https://cloud.google.com/blog/products/identity-security/why-google-now-uses-post-quantum-cryptography-for-internal-comms, (Archived on 3 Feb. 2025). URL: https://cloud.google.com/blog/products/identity-security/why-google-now-uses-post-quantum-cryptography-for-internal-comms.

[ISO24] ISO Central Secretary. *Information technology – Security techniques – Testing methods for the mitigation of non-invasive attack classes against cryptographic modules*. Standard ISO/IEC 17825:2024. Geneva, CH: International Organization for Standardization, Jan. 2024. URL: https://www.iso.org/standard/82422.html.

[ISO25a] ISO Central Secretary. *Information security, cybersecurity and privacy protection – Security requirements for cryptographic modules*. Standard ISO/IEC 19790:2025. Geneva, CH: International Organization for Standardization, Feb. 2025. URL: https://www.iso.org/standard/82423.html.

[ISO25b] ISO Central Secretary. *Information security, cybersecurity and privacy protection – Test requirements for cryptographic modules*. Standard ISO/IEC 24759:2025. Geneva, CH: International Organization for Standardization, Feb. 2025. URL: https://www.iso.org/standard/82424.html.

[JC19] Wilbur L. Ross Jr. and Walter Copan. *Security requirements for cryptographic modules*. NIST standard FIPS 140-3. Mar. 2019. DOI: 10.6028/nist.fips.140-3. URL: http://dx.doi.org/10.6028/NIST.FIPS.140-3.

[JR23] Samuel Jaques and Arthur G Rattew. "Qram: A survey and critique". In: *arXiv preprint arXiv:2305.10310* (2023). DOI: 10.48550/arXiv.2305.10310. URL: https://doi.org/10.48550/arXiv.2305.10310.

[Kar63] Anatolii Karatsuba. "Multiplication of multidigit numbers on automata". In: *Soviet physics doklady*. Vol. 7. 1963, pp. 595–596.

[Knu98] Donald E. Knuth. *The art of computer programming*. Addison-Wesley, 1998.

[Li+23] Chen Li et al. "An Efficient Hardware Design for Fast Implementation of HQC". In: *36th IEEE International System-on-Chip Conference, SOCC 2023, Santa Clara, CA, USA, September 5-8, 2023*. Ed. by Jürgen Becker et al. IEEE, 2023, pp. 1–6. DOI: 10.1109/SOCC58585.2023.10257054. URL: https://doi.org/10.1109/SOCC58585.2023.10257054.

[LJ83] Shu Lin and Daniel J. Costello Jr. *Error control coding - fundamentals and applications*. Prentice Hall computer applications in electrical engineering series. Prentice Hall, 1983. ISBN: 978-0-13-283796-5.

[LSG21] Georg Land, Pascal Sasdrich, and Tim Güneysu. "A Hard Crystal - Implementing Dilithium on Reconfigurable Hardware". In: *Smart Card Research and Advanced Applications - 20th International Conference, CARDIS 2021, Lübeck, Germany, November 11-12, 2021, Revised Selected Papers*. Ed. by Vincent Grosso and Thomas Pöppelmann. Vol. 13173. Lecture Notes in Computer Science. Springer, 2021, pp. 210–230. DOI: 10.1007/978-3-030-97348-3_12. URL: https://doi.org/10.1007/978-3-030-97348-3_12.

[LW15] Bingxin Liu and Huapeng Wu. "Efficient architecture and implementation for NTRUEncrypt system". In: *IEEE 58th International Midwest Symposium on Circuits and Systems, MWSCAS 2015, Fort Collins, CO, USA, August 2-5, 2015*. IEEE, 2015, pp. 1–4. DOI: 10.1109/MWSCAS.2015.7282143. URL: https://doi.org/10.1109/MWSCAS.2015.7282143.

[Mar+15] Honorio Martín et al. "Fault Attacks on STRNGs: Impact of Glitches, Temperature, and Underpowering on Randomness". In: *IEEE Trans. Inf. Forensics Secur.* 10.2 (2015), pp. 266–277. DOI: 10.1109/TIFS.2014.2374072. URL: https://doi.org/10.1109/TIFS.2014.2374072.

[Mas69] James L. Massey. "Shift-register synthesis and BCH decoding". In: *IEEE Trans. Inf. Theory* 15.1 (1969), pp. 122–127. DOI: 10.1109/TIT.1969.1054260. URL: https://doi.org/10.1109/TIT.1969.1054260.

[Mas88] Edoardo D. Mastrovito. "VLSI Designs for Multiplication over Finite Fields *GF* (2^m)". In: *6th International Conference on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, (AAECC-6), Rome, Italy, July 4-8, 1988, Proceedings*. Ed. by Teo Mora. Vol. 357. Lecture Notes in Computer Science. Springer, 1988, pp. 297–309. DOI: 10.1007/3-540-51083-4_67. URL: https://doi.org/10.1007/3-540-51083-4_67.

[Mil+24] Carl Miller et al. *Status Report on the First Round of Additional Digital Signature Schemes for Post-Quantum Cryptography*. Oct. 2024. DOI: 10.6028/nist.ir.8528. URL: http://dx.doi.org/10.6028/NIST.IR.8528.

[MM09] A. Theodore Markettos and Simon W. Moore. "The Frequency Injection Attack on Ring-Oscillator-Based True Random Number Generators". In: *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*. Ed. by Christophe Clavier and Kris Gaj. Vol. 5747. Lecture Notes in Computer Science. Springer, 2009, pp. 317–331. DOI: 10.1007/978-3-642-04138-9_23. URL: https://doi.org/10.1007/978-3-642-04138-9_23.

[Mon+24] Puja Mondal et al. "ZKFault: Fault Attack Analysis on Zero-Knowledge Based Post-quantum Digital Signature Schemes". In: *Advances in Cryptology - ASIACRYPT 2024 - 30th International Conference on the Theory and Application of Cryptology and Information Security, Kolkata, India, December 9-13, 2024, Proceedings, Part VIII*. Ed. by Kai-Min Chung and Yu Sasaki. Vol. 15491. Lecture Notes in Computer Science. Springer, 2024, pp. 132–167. DOI: 10.1007/978-981-96-0944-4_5. URL: https://doi.org/10.1007/978-981-96-0944-4_5.

[Moo+20] Dustin Moody et al. *Status report on the second round of the NIST post-quantum cryptography standardization process*. July 2020. DOI: 10.6028/nist.ir.8309. URL: http://dx.doi.org/10.6028/NIST.IR.8309.

[Moo+24] Dustin Moody et al. *Transition to Post-Quantum Cryptography Standards*. Nov. 2024. DOI: 10.6028/nist.ir.8547.ipd. URL: http://dx.doi.org/10.6028/NIST.IR.8547.ipd.

[Ney+24] Samuel Neyens et al. "Probing single electrons across 300-mm spin qubit wafers". In: *Nature* 629.8010 (2024), pp. 80–85. DOI: 10.1038/s41586-024-07275-6. URL: https://doi.org/10.1038/s41586-024-07275-6.

[NRS08] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. "Secure Hardware Implementation of Non-linear Functions in the Presence of Glitches". In: *Information Security and Cryptology - ICISC 2008, 11th International Conference, Seoul, Korea, December 3-5, 2008, Revised Selected Papers*. Ed. by Pil Joong Lee and Jung Hee Cheon. Vol. 5461. Lecture Notes in Computer Science. Springer, 2008, pp. 218–234. DOI: 10.1007/978-3-642-00730-9_14. URL: https://doi.org/10.1007/978-3-642-00730-9%5C_14.

[ORM22] Andreas Olofsson, William Ransohoff, and Noah Moroze. "A Distributed Approach to Silicon Compilation: Invited". In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. San Francisco, California, 2022, pp. 1343–1346.

[Pan66] V Ya Pan. "Methods of computing values of polynomials". In: *Russian Mathematical Surveys* 21.1 (1966), pp. 105–136. URL: https://api.semanticscholar.org/CorpusID:250869179.

[Pen+21] Bo-Yuan Peng et al. "Streamlined NTRU Prime on FPGA". In: *IACR Cryptol. ePrint Arch.* (2021), p. 1444. URL: https://eprint.iacr.org/2021/1444.

[Pen+23] Bo-Yuan Peng et al. "Streamlined NTRU Prime on FPGA". In: *J. Cryptogr. Eng.* 13.2 (2023), pp. 167–186. DOI: 10.1007/S13389-022-00303-Z. URL: https://doi.org/10.1007/s13389-022-00303-z.

[PM15] Penny Pritzker and Willie May. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. NIST standard FIPS 202. 2015. DOI: 10.6028/NIST.FIPS.202. URL: https://doi.org/10.6028/NIST.FIPS.202.

[Ric+22] Jan Richter-Brockmann et al. "Racing BIKE: Improved Polynomial Multiplication and Inversion in Hardware". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.1 (2022), pp. 557–588. DOI: 10.46586/TCHES.V2022.I1.557-588. URL: https://doi.org/10.46586/tches.v2022.i1.557-588.

[RL23a] Gina M. Raimondo and Laurie E. Locascio. *Digital Signature Standard (DSS)*. NIST standard FIPS 186. Feb. 2023. DOI: 10.6028/nist.fips.186-5. URL: http://dx.doi.org/10.6028/NIST.FIPS.186-5.

[RL23b] Gina M. Raimondo and Laurie E. Locascio. *Module-Lattice-Based Digital Signature Standard*. RFC NIST standard FIPS 204. Aug. 2023. DOI: 10.6028/nist.fips.204.ipd. URL: http://dx.doi.org/10.6028/NIST.FIPS.204.ipd.

[RL23c] Gina M. Raimondo and Laurie E. Locascio. *Module-Lattice-Based Key-Encapsulation Mechanism Standard*. RFC NIST standard FIPS 203. Aug. 2023. DOI: 10.6028/nist.fips.203.ipd. URL: http://dx.doi.org/10.6028/NIST.FIPS.203.ipd.

[RL23d] Gina M. Raimondo and Laurie E. Locascio. *Stateless Hash-Based Digital Signature Standard*. RFC NIST standard FIPS 205. Aug. 2023. DOI: 10.6028/nist.fips.205.ipd. URL: http://dx.doi.org/10.6028/NIST.FIPS.205.ipd.

[Saa23] Markku-Juhani O. Saarinen. *Introduction to Side-Channel Security of NIST PQC Standards*. Invited talk at PQC Seminars, NIST 2023 [Online]. Available from: https://web.archive.org/web/20250319051738/https://csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/pqc-seminars/presentations/2-side-channel-security-saarinen-04042023.pdf, (Archived on 19 Mar. 2025). 2023. URL: https://csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/pqc-seminars/presentations/2-side-channel-security-saarinen-04042023.pdf.

[Sch] Sophie Schmieg. *PQC at Google*. Invited talk at The 14th International Conference on Post-Quantum Cryptography, PQCrypto 2023 [Online]. Available from: https://web.archive.org/web/20240420093321/https://pqcrypto2023.umiacs.io/slides/Invited.3.pdf, (Archived on 20 Apr. 2024). URL: https://pqcrypto2023.umiacs.io/slides/Invited.3.pdf.

[Sen21] Nicolas Sendrier. "Secure Sampling of Constant-Weight Words – Application to BIKE". In: *IACR Cryptol. ePrint Arch.* (2021), p. 1631. URL: https://eprint.iacr.org/2021/1631.

[Sho94] Peter W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700. URL: https://doi.org/10.1109/SFCS.1994.365700.

[Sil] Joseph H Silverman. "NTRU and Lattice-Based Crypto: Past, Present, and Future". [Online]. Available from: https://web.archive.org/web/20240814062435/http://archive.dimacs.rutgers.edu/Workshops/Post-Quantum/Slides/Silverman.pdf, (Archived on 14 Aug. 2024). URL: http://archive.dimacs.rutgers.edu/Workshops/Post-Quantum/Slides/Silverman.pdf.

[Sny+] Wilson Snyder et al. *Verilator*. [Online]. Available from: https://web.archive.org/web/20250109003634/https://verilator.org, (Archived on 9 Jan. 2025). URL: https://verilator.org.

[Sou+11] Mathilde Soucarros et al. “Influence of the temperature on true random number generators”. In: *HOST 2011, Proceedings of the 2011 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), 5-6 June 2011, San Diego, California, USA*. IEEE Computer Society, 2011, pp. 24–27. DOI: 10.1109/HST.2011.5954990. URL: https://doi.org/10.1109/HST.2011.5954990.

[SP24] D Sarah and C Peter. *On the Practical cost of Grover for AES Key Recovery*. [Online]. Available from: https://web.archive.org/web/20240420015549/https://csrc.nist.gov/csrc/media/Events/2024/fifth-pqc-standardization-conference/documents/papers/on-practical-cost-of-grover.pdf, (Archived on 24 Apr. 2024). 2024. URL: https://csrc.nist.gov/csrc/media/Events/2024/fifth-pqc-standardization-conference/documents/papers/on-practical-cost-of-grover.pdf.

[SS18] Ray Salemi and Siemens EDA. *pyuvm - Universal Verification Methodology based on the IEEE 1800.2 specification implemented in Python*. [Online]. Available from: https://web.archive.org/web/20250116095223/https://github.com/pyuvm/pyuvm, (Archived on 16 Jan. 2025). 2018. URL: https://github.com/pyuvm/pyuvm.

[Ste12] Marc Martinus Jacobus Stevens. “Attacks on hash functions and applications”. Doctoral thesis. Leiden University, 2012. URL: https://hdl.handle.net/1887/19093.

[Ste93] Jacques Stern. “A New Identification Scheme Based on Syndrome Decoding”. In: *Advances in Cryptology - CRYPTO ’93, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, 1993, Proceedings*. Ed. by Douglas R. Stinson. Vol. 773. Lecture Notes in Computer Science. Springer, 1993, pp. 13–21. DOI: 10.1007/3-540-48329-2_2. URL: https://doi.org/10.1007/3-540-48329-2_2.

[Sti+07] James E. Stine et al. “FreePDK: An Open-Source Variation-Aware Design Kit”. In: *IEEE International Conference on Microelectronic Systems Education, MSE ’07, San Diego, CA, USA, June 3-4, 2007*. IEEE Computer Society, 2007, pp. 173–174. DOI: 10.1109/MSE.2007.44. URL: https://doi.org/10.1109/MSE.2007.44.

[SXY18] Tsunekazu Saito, Keita Xagawa, and Takashi Yamakawa. “Tightly-Secure Key-Encapsulation Mechanism in the Quantum Random Oracle Model”. In: *EUROCRYPT 2018*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10822. LNCS. Springer, 2018, pp. 520–551. DOI: 10.1007/978-3-319-78372-7_17. URL: https://doi.org/10.1007/978-3-319-78372-7_17.

[The22] The OpenSSH Team. *OpenSSH Changelog for version 9.0*. [Online]. Available from: https://web.archive.org/web/20250203084110/https://www.openssh.com/txt/release-9.0, (Archived on 3 Feb. 2025). 2022. URL: https://www.openssh.com/txt/release-9.0.

[The25a] The OpenSSH Team. *OpenSSH Changelog for version 10.0*. [Online]. Available from: https://web.archive.org/web/20250411145536/https://www.openssh.com/txt/release-10.0, (Archived on 11 Apr. 2025). 2025. URL: https://www.openssh.com/txt/release-10.0.

[The25b] The OpenSSL Team. *OpenSSL Changelog for version 3.5.0*. [Online]. Available from: https://web.archive.org/web/20250410160838/https://github.com/openssl/openssl/releases/tag/openssl-3.5.0, (Archived on 10 Apr. 2025). 2025. URL: https://github.com/openssl/openssl/releases/tag/openssl-3.5.0.

[VAM22] VAMPIRE lab. *SUPERCOP: System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives*. [Online]. Available from: https://web.archive.org/web/20250115002136/https://bench.cr.yp.to/results-kem.html, (Archived on 15 Jan. 2025). 2022. URL: https://bench.cr.yp.to/results-kem.html.

[Wan+20] Wen Wang et al. "Parameterized Hardware Accelerators for Lattice-Based Cryptography and Their Application to the HW/SW Co-Design of qTESLA". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.3 (2020), pp. 269–306. DOI: 10.13154/TCHES.V2020.I3.269-306. URL: https://doi.org/10.13154/tches.v2020.i3.269-306.

[Web+22] Mark Webber et al. "The impact of hardware specifications on reaching quantum advantage in the fault tolerant regime". In: *AVS Quantum Science* 4.1 (2022). DOI: 10.1116/5.0073075. URL: https://doi.org/10.1116/5.0073075.

[Weg+24] Violetta Weger et al. "On the hardness of the Lee syndrome decoding problem". In: *Adv. Math. Commun.* 18.1 (2024), pp. 233–266. DOI: 10.3934/AMC.2022029. URL: https://doi.org/10.3934/amc.2022029.

[Wu15] Yingquan Wu. "New Scalable Decoder Architectures for Reed-Solomon Codes". In: *IEEE Trans. Commun.* 63.8 (2015), pp. 2741–2761. DOI: 10.1109/TCOMM.2015.2445759. URL: https://doi.org/10.1109/TCOMM.2015.2445759.

[Xin18] Homer Xing. *Keccak core*. [Online]. Available from: https://web.archive.org/web/20240804051140/https://opencores.org/projects/sha3, (Archived on 1 Aug. 2024). 2018. URL: https://opencores.org/projects/sha3.

[XL21] Yufei Xing and Shuguo Li. "A Compact Hardware Implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.2 (2021), pp. 328–356. DOI: 10.46586/TCHES.V2021.I2.328-356. URL: https://doi.org/10.46586/tches.v2021.i2.328-356.

[XLF13] Tao Xie, Fanbao Liu, and Dengguo Feng. "Fast Collision Attack on MD5". In: *IACR Cryptol. ePrint Arch.* (2013), p. 170. URL: http://eprint.iacr.org/2013/170.

[Ylo06] Tatu Ylonen. *IETF RFC 4252 - The Secure Shell (SSH) Authentication Protocol*. [Online]. Available from: https://web.archive.org/web/20250126230533/https://www.rfc-editor.org/rfc/rfc4252, (Archived on 26 Jan. 2025). 2006. URL: https://www.rfc-editor.org/rfc/rfc4252.

Acronyms

μCVP Approximate Closest Vector Problem.

μSVP Approximate Shortest Vector Problem.

CROSS Codes and Restricted Objects Signature Scheme.

FN-DSA Fast-Fourier Transform over NTRU-Lattice-Based Digital Signature Algorithm.

HQC Hamming Quasi-Cyclic.

LMS Leighton-Micali Signature.

ML-DSA Module-Lattice-Based Digital Signature Standard.

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

NTRU N-th degree Truncated polynomial Ring Units.

RSA Rivest-Shamir-Adleman.

SHAKE Secure Hash Algorithm Keccak.

SLH-DSA Stateless Hash-Based Digital Signature Standard.

XMSS eXtended Merkle Signature Scheme.

AEAD Authenticated Encryption with Associated Data.

AES Advanced Encryption Standard.

AMBA Advanced Microcontroller Bus Architecture.

ANNSI Agence nationale de la sécurité des systèmes d'information (French Cybersecurity Agency).

API Application Programming Interface.

ASIC Application Specific Integrated Circuit.

AT Area-Time.

AXI Advanced eXtensible Interface.

BCH Bose-Chaudhuri-Hocquenghem.

BGF Black-Gray-Flip.

BKZ Block Korkine-Zolotarev.

BM Berlekamp-Massey.

BRAM Block RAM.
BSI Bundesamt für Sicherheit in der Informationstechnik (German Federal Office for Information Security).

CDF Cumulative Distribution Function.
CIV Continuous Integration for Verification.
CLB Configurable Logic Block.
CPA Correplation Power Analysis.
CPU Centra Processor Unit.
CSPRNG Cryptographically Secure Random Number Generator.
CVP Closest Vector Problem.

DFR Decoding Failure Rate.
DFT Discrete Fourier Transforms.
DOM Domain-Oriented Masking.
DPA Differential Power Analysis.
DPKE Deterministic Public-Key Encryption.
DRC Design Rule Check.
DS Digital Signature.
DSE Design Space Exploration.
DSL Digital Subscriber Line.
DSP Digital Signal Processor.
DVB Digital Video Broadcasting.

EC Elliptic Curve.
ECC Error-Correcting Code.
ECDLP Elliptic Curve Discrete logarithm Problem.
eCSEE Enhanced Chien Search and Error Evaluation.
EDA Electronic Design Automation.
EM electromagnetic.
ePIBMA Enhanced Parallel Inversionless Berlekamp-Massey Algorithm.
ETSI European Telecommunications Standards Institute.
EUF-CMA Existential Unforgeability under Chosen Message Attacks.

FF Flip-Flop.
FIFO First-In First-Out.
FIPS Federal Information Processing Standard.
FPGA Field Programmable Gate Array.
FS Fiat-Shamir.
FSA Finite State Automata.
FSM Finite State Machine.

GCM Galois Counter Mode.
GDP Generic Decoding Problem.

GE Gate Equivalent.

GPV Gentry-Peikert-Vaikuntanathan.

GV Gilbert-Varshamov.

HHK Hofheinz-Hövelmanns-Kiltz.

HLS High Level Synthesis.

HMAC Hash Message Authentication Code.

HSM Hardware Security Modules.

HSP Hidden Subgroup Problem.

HW Hardware.

IEEE Institute of Electrical and Electronics Engineers.

IFP Integer Factorization Problem.

IND-CCA2 INDistinguishability under adaptive Chosen Ciphertext Attack.

IP Intellectual Property.

IR Interagency Report.

ISA Instruction Set Architecture.

IV Initialization Vector.

KAT Known Answer Test.

KDF Key Derivation Function.

KEM Key Establishment Mechanism.

KiB kibibyte, kilo binary bytes (1 KiB = 2^{10} bytes, 2.4% more than 1 kilo bytes (KB) = 10^3 bytes).

LFSR Linear-Feedback Shift Register.

LLL Lenstra-Lenstra-Lovász.

LSB Least Significant Bit.

LUT Look-Up Table.

LVS Layout Versus Schematic.

LWE Learning With Errors.

LWR Learning With Rounding.

M-LWE Module-LWE.

M-LWR Module-LWR.

MAC multiply-and-accumulate.

MD Merkle-Damgård.

MDPC Moderate-Density Parity-Check.

MiB Mebibyte, mega binary bytes (1 MiB = 2^{20} bytes, 4.8% more than 1 mega bytes (MB) = 10^6 bytes).

MPCitH Multy-Party Computation in-the-Head.

MSB Most Significant Bit.

NIC Network Interface Card.

NIST National Institute of Standards and Technology.

NTT Number-Theoretic Transform.

PDK Process Design Kit.
PGP Pretty Good Privacy.
PGZ Petterson-Gorenstein-Zierler.
PiB Pebibyte, tera binary bytes (1 PiB = 2^{50} bytes, 1024 tebibytes).
PKC Public-Key Cryptography.
PPKE Probabilistic Public-Key Encryption.
PQC Post-Quantum Cryptography.
PRNG Pseudo-Random Number Generator.
PVT Process, Voltage, and Temperature.

QC Quasi-Cyclic.
QPU Quantum Processor Unit.
QRAM Quantum Random-Access Memory.

R-LWE Ring-LWE.
R-LWR Ring-LWR.
R-SDP Restricted Syndrome Decoding Problem.
R-SDP(G) Restricted Syndrome Decoding Problem in the subgroup G.
RAM Random-Access Memory.
REST REpresentational State Transfer.
RFC Request For Comments.
RM Reed-Muller.
RM/RS Reed-Muller/Reed-Solomon.
RNG Random Number Generator.
ROM Read-Only Memory.
RS Reed-Solomon.
RTL Register Transfer Level.

SCA Side-Channel Attack.
SDP Syndrome Decoding Problem.
SHA Secure Hash Algorithm.
SIMD Single Instruction Multiple Data.
SPA Simple Power Analysis.
SRAM Static RAM.
SSH Secure SHell.
SSP Subset Sum Problem.
SV SystemVerilog.
SVP Shortest Vector Problem.
SW Software.

TI Threashold Implementations.

TLD Top-Level Design.

TLS Transport Layer Security.

TRNG True-Random Number Generator.

UART Universal Asynchronous Receiver-Transmitter.

UVM Universal Verification Methodology.

VPN Virtual Private Network.

XOF eXtendable-Output Function.

ZK Zero-Knowledge.