



PiQASO



PiQASO D3.2 PiQASO SDK and PQC as-a-service – Initial Version

Grant Agreement	101190366
Project Title	Post-Quantum Cryptography As-a-Service for Common Transmission Systems and Infrastructures
Project acronym	PiQASO
Project Start Date	01 January 2025
Number of Deliverable	D3.2
Report Title	PiQASO SDK and PQC as-a-service – Initial Version
Related Work Package	WP3
Related Task	T3.1, T3.2, T3.3, T3.4
Lead Organisation	UBITECH
Submission Date	03.08.2026
Last Change Date	31.07.2026
Dissemination Level	Public

Abstract

D3.2 marks the transition of the PiQASO PQC Ensemble from design to implementation, delivering the first operational software artifacts building upon the theoretical and architectural foundations established in D3.1. This deliverable presents the first version of the PiQASO SDK, a portable and modular software library equipping far-edge devices with standalone implementations of ML-KEM, ML-DSA, One-Time Signatures, HQC, and XMSS/LMS, together with a first realization of PQ-TLS 1.3. Complementing the SDK, the first version of PQ- CaaS is introduced, extending quantum-secure capabilities to legacy and resource-constrained devices through support for mixed TLS certificates. D3.2 further provides concrete implementations of the Forward Secure Cloud Storage Scheme based on Updatable Public Key Encryption, and of Homomorphic Functional Encryption. Finally, the deliverable details PiQASO's SW/HW co-design approach to acceleration, presenting hardware accelerators for NTT and HQC arithmetic, as well as LWE decomposition, a new construction enabling field-like operations over LWE secret keys to accelerate advanced protocols at the protocol level.

Authoring & approval

Author(s) of the document

Name (Organisation name)	Date
Dimitris Karras (UBI)	20.07.26
Thanasis Giannetsos (UBI)	31.07.26
Stylianos Kazazis (QUBI)	31.07.26
Dimitris Katsinis (QUBI)	31.07.26
Niovi Sassalou (QUBI)	31.07.26
Alexandros Tavernarakis (QUBI)	31.07.26
Angelina Katsifarakis (NCIS)	17.07.26
Olga Politi (NCIS)	17.07.26
Mark Manulis (UniBwM)	15.07.26
Sohyun Jeon (UniBwM)	15.07.26
Daniel Slamanig (UniBwM)	15.07.26
Antonis Michalas (TAU)	15.07.26
Mélina Hadjeres (TAU)	15.07.26
Camille Nuoskala (TAU)	15.07.26
Hossein Abdinasibfar (TAU)	15.07.26
Konstantin Klein (BIBA)	18.06.26
Dimitris Dimitrakoudis (BYTE)	20.07.26

Reviewed by

Name (Organisation name)	Date
Anastasios Tzoumpas (BYTE)	23.07.26
Isaac Dangana (RAL)	31.07.26

Approved for submission

Name (Organisation name)	Date
Savvoula Oikonomou, UBITECH	03.08.2026

Document history

Version	Date	Contributor/Organisation	Additional information
0.1	21.05.26	UBI	V01 created, ToC prepared
0.2	03.06.25	TAU, UniBwM	V02 initial input on Chapters 4 and 5. Weekly/biweekly calls with technical partners for feedback and development.
0.3	15.06.26	UniBwM, UBI	V03 with further input on Chapter 4 and figures added, initial input on Chapter 1. Weekly/biweekly calls with technical partners for feedback and development.
0.4	18.06.26	UBI, UniBwM, TAU	V04 with further input on Chapters 3, 4 and 5. Weekly/biweekly calls with technical partners for feedback and development.
0.5	15.07.26	UBI, QUBI, UniBwM, TAU,	V05 with finalisation of Chapters 3, 4, 5. Proof-reading and review conducted.
0.6	17.07.26	UBI, QUBI, NCIS, BYTE	V06 finalisation of content of Chapters 1, 2, 6, 7, 8, 9 and 10. Weekly/biweekly calls with technical partners for feedback and development.
0.7	20.07.26	UBI, QUBI	V07 review ready version.
1.0	31.07.26	UBI, QUBI	V1 ready for submission with review comments addressed.

Copyright statement © 2025 – PiQASO Consortium
All rights reserved. Licensed to PiQASO under conditions.

PiQASO

Post-Quantum Cryptography As-a-Service for Common Transmission Systems and Infrastructures

The project funded under Grant Agreement No. 101190366 is supported by the European Cybersecurity Competence Centre



Co-funded by
the European Union



ECCC
EUROPEAN CYBERSECURITY
COMPETENCE CENTRE

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or European Cybersecurity Competence Centre. Neither the European Union nor the granting authority can be held responsible for them."



Co-funded by
the European Union



ECCC
EUROPEAN CYBERSECURITY
COMPETENCE CENTRE

Executive Summary

Considering the advent of quantum computers, it is imperative to ensure that all existing information security systems are able to cope with attackers and malicious entities who possess quantum computing capabilities. However, one core challenge in this regard is that existing service graph chains (SGCs) may consist of heterogeneous devices with varying computational capabilities, including legacy devices with limited ability to support advanced cryptography. Building on the theoretical and architectural foundations established in **D3.1**, **D3.2** marks the transition of the PiQASO PQC Ensemble from design to implementation, delivering the *first operational software artifacts* required for the post-quantum transition of such SGCs, namely the first version of the **PiQASO SDK** and the first version of **PQCaaS**.

Chapter 2 provides a concise roadmap situating the contributions of D3.2 within the trajectory established in D3.1 and outlining the work to be completed in D3.3, clarifying which cryptographic primitives have transitioned from theoretical specification to concrete software implementation. Chapter 3 then details the first version of the **PiQASO SDK**, a portable and modular software library equipping far-edge devices with standalone implementations of **ML-KEM** and **ML-DSA** (the NIST-standardized lattice-based KEM and signature scheme), **One-Time Signatures** for blockchain-anchored data-sharing auditability, **HQC** as a code-based KEM diversifying the SDK's hardness assumptions, and the stateful hash-based **XMSS/LMS** signature schemes, together with a first realization of **PQ-TLS 1.3** combining ML-KEM key establishment with ML-DSA signatures. Complementing the SDK, the first version of **PQCaaS** is released as an open-source, containerized software stack, offering the same PQ-TLS 1.3 protocol, hybrid and mixed TLS certificates, the first version of **Updatable Public Key Encryption**, and **Functional Encryption** capabilities, enabling legacy and resource-constrained devices to obtain quantum-secure authentication *without needing to execute complex PQC operations locally*.

One important aspect of securing modern computing infrastructures is the use of cloud storage. In this regard, Chapter 4 advances the **forward-secure cloud storage scheme** of PiQASO, whose mathematical foundations and security model were established in D3.1, towards practical deployment. Specifically, we provide a concrete implementation of the scheme under a *trusted Key Provider* infrastructure, realizing key management, encapsulation, and decapsulation through ML-KEM and AES-GCM, and further introduce an advanced variant designed for the *untrusted Key Provider* setting, in which clients generate and update their own key pairs and verify the validity of public-key updates performed by the cloud server, thus removing the strict trust dependency on the Key Provider.

Chapter 5 introduces **Homomorphic Functional Encryption (HFE)**, a scheme-agnostic construction that combines Homomorphic Encryption and Functional Encryption to address the reliance of traditional Functional Encryption schemes on a fully trusted authority. In HFE, the trusted authority is replaced by a **key curator** whose role is restricted to generating and distributing encrypted functional decryption keys, without ever learning the function being requested, a property referred to as *function blindness*. This construction is particularly relevant to privacy-preserving healthcare analytics, such as the **TELLU and PARTICULE** use cases, where the function requested over encrypted patient data (e.g., a diagnostic computation) may itself reveal sensitive information about a patient or the specialization of the requesting physician.

One core target of PiQASO is the ability to provide post-quantum cryptographic capabilities to the far

edge, including embedded devices subject to strict latency, memory, and power constraints. To this end, Chapters 6 and 7 detail PiQASO's hybrid **Software/Hardware co-design** approach to accelerating the two core arithmetic operations identified in D3.1, namely the **Number-Theoretic Transform (NTT)** underlying lattice-based constructions and the **Karatsuba**-based binary polynomial multiplication underlying HQC. Chapter 6 reports the hardware-based acceleration of the NTT operation, complementing the software-level, **FFT-based** algorithmic optimization roadmap set out in Chapter 3. Chapter 7 presents a dedicated **hardware accelerator for HQC** targeting its dominant computational bottleneck, constant-time binary polynomial multiplication, which accounts for approximately **96%** of the total execution time of the standard reference implementation across all HQC security levels; the accelerator implements a flattened, non-recursive variant of the Karatsuba algorithm on the **Xilinx Zynq UltraScale+ ZCU104** platform.

Chapter 8 details the extension of **PQCaaS** to support **mixed TLS certificates**, describing a custom **NGINX**-based PQ-TLS termination and mediation stack developed by NCIS. This architecture enables a sender to authenticate via typical, classical TLS, while PQCaaS **automatically** and transparently mediates a PQ-TLS session with the receiving point on its behalf, proxying through FastAPI to the PiQASO client, across hybrid, PQ-only, and PQ-signed certificate endpoints, with certificate issuance managed through an EJBCA bridge. ***This capability is comparable to that exposed by commercial providers such as Cloudflare, replicated by PiQASO through an open-source, containerized software stack,*** and will initially be validated in the context of the Transportation use case (UC9).

Finally, Chapter 9 introduces **LWE decomposition**, a new construction that augments Learning With Errors (LWE) secret keys with invertibility, expressed via polynomial matrices. Unlike the arithmetic-level accelerations described above, which reduce the cost of operations that a protocol already performs, LWE decomposition accelerates advanced protocols, such as *threshold decryption, multi-party computation, and Attribute-Based Encryption*, **at the level of the protocol itself**, by eliminating the trapdoors and gadget matrices otherwise required to obtain field-like operations (multiplication and inversion) over lattice-based secret keys, while retaining the error-based hardness of LWE. We formalize this construction, prove its reduction to standard and module-LWE, illustrate its use through a two-party decryption scheme, and validate its practicality through a prototype integrated with the PiQASO SDK's ML-KEM implementation, exhibiting performance comparable to the baseline implementation.

Compounding all the above, D3.2 is the second deliverable in a series of PiQASO deliverables aimed towards the design and implementation of the PiQASO PQC Ensemble. The first operational artifacts delivered here, the **PiQASO SDK** and **PQCaaS**, set the foundation for their integration and evaluation in the context of the PiQASO use case demonstrators, as planned in D2.1 and to be reported in **D4.1**, while their final versions, together with further acceleration capabilities and the **QR Trusted Computing Base**, will be reported in **D3.3**.

Contents

1	Introduction	18
1.1	Scope and Purpose	18
1.2	Relation to other WPs and Deliverables	19
1.3	Deliverable Structure	20
2	Updates in the Crypto Designs of PiQASO	21
2.1	PiQASO Software Development Kit (SDK)	23
2.2	PQC-as-a-Service (PQCaaS)	24
2.3	Roadmap for Next Actions	24
2.3.1	Evaluation of PiQASO Functionalities in the Use Cases	25
2.4	Summary	26
3	SW-Based Implementation and Library	27
3.1	PiQASO SDK Interfaces	30
3.1.1	AES-GCM Encryption API	30
3.1.2	LMS/HSS Digital Signature Functions	31
3.1.3	XMSS/XMSS Digital Signature Functions	32
3.1.4	ML-KEM Key Encapsulation Functions	34
3.1.5	ML-DSA Digital Signature Functions	35
3.1.6	Post-Quantum TLS 1.3 Functions	37
3.2	Unit Tests	39
3.3	Building Quantum-Safe Infrastructures: Desiderata of the PiQASO PQCaaS Solution	40
3.3.1	Architectural Paradigm of the PQCaaS Gateway and Client	41
3.3.2	Technical Implementation and Cryptographic Parity	42
3.3.3	Integration of Mixed TLS Certificates and NGINX Proxy Architecture	45
3.4	PiQASO SDK Performance Benchmarks and Architectural Critique on the ZCU 104 Platform	46
3.5	Algorithmic Optimizations of Lattice-Based Algorithms	48
3.5.1	Context and Mathematical Foundations	51
3.5.2	Standard Approach vs. Computational Bottlenecks	51
3.5.3	Proposed Optimization: Complex Domain Mapping	51
3.5.4	Precision and Correctness Strategy	52
3.5.5	Generalization of LWE Problem for Increased Efficiency	52

4	Forward Secure Cloud Storage Scheme	53
4.1	Implementation of the Cloud Storage Scheme with Trusted Key Provider	54
4.2	Specification of the Cloud Storage Scheme with Untrusted Key Provider	56
4.2.1	Background and Building Blocks	58
4.2.2	Updatable Cloud Storage Scheme with Untrusted Key Provider	64
5	Homomorphic Functional Encryption	69
5.1	Motivation	69
5.2	Introduction	70
5.2.1	Roadmap	70
5.3	High-Level Overview	70
5.4	Background	71
5.4.1	Encryption Schemes	71
5.5	Homomorphic Evaluation of FE	73
5.5.1	Evaluation of FE Functions	73
5.5.2	Representation of Functions	75
5.6	Homomorphic Functional Encryption (HFE)	76
5.6.1	Function Blindness	76
5.7	Real-World Deployment Scenario for HFE	78
5.8	Semantic Security	80
5.8.1	Indistinguishability-based Security	80
5.8.2	Security of HFE	82
5.9	Experiments	85
5.9.1	Benchmarks	85
5.10	Conclusions and Future Directions	87
6	PiQASO NTT Accelerator for Lattice-based Schemes	89
6.1	Profiling of ML-KEM; ML-DSA & Performance Bottlenecks	89
6.2	NTT/INTT Hardware Accelerator: Architecture and Evaluation	91
6.2.1	Fundamental Concepts	92
6.2.2	Overview	92
6.2.3	Configuration and Control Interface	93
6.2.4	Twiddle-Factor Generation	95
6.2.5	Datapath and Supported Features	95
6.2.6	Core Latency	97
6.3	PiQASO NTT/INTT Hardware Accelerator Integrated Design	99
6.4	Evaluation	100
6.4.1	Results	101
6.4.2	Discussion & Critique	102
6.5	Limitations and Future Work	103

7	HQC Acceleration by Speeding Up Binary Polynomial Multiplications (BPM)	105
7.1	Current PQC Landscape and the HQC Bottleneck	105
7.2	Preliminaries	106
7.3	PiQASO's Accelerator Architecture	107
7.3.1	System Integration and Scalability	108
7.4	Evaluation	109
7.5	Latency and Resource Footprint	109
7.6	Quantitative and Metric-Driven Comparative Evaluation	110
7.7	Discussion and Critique	111
8	PQCaaS Extension to Support Mixed TLS Certificates	112
9	LWE Decomposition for Accelerated Post-Quantum Protocols	115
9.1	Preliminaries	117
9.1.1	Learning With Errors (LWE)	117
9.1.2	Module Learning With Errors (Module-LWE)	117
9.1.3	ML-KEM (Kyber)	118
9.1.4	Rings vs. Fields: A Cryptographic Perspective	118
9.2	Construction	120
9.2.1	Key Generation	120
9.2.2	Encapsulation	121
9.2.3	Decapsulation	122
9.2.4	Two-Party Kyber via Secret Splitting	123
9.3	Algorithmic Specifications and Parameter Selection	124
9.3.1	Algorithmic Primitives and Implementation Choices	124
9.3.2	Concrete Parameter Instantiation	125
9.3.3	Error Accumulation and Correctness	126
9.4	Security Analysis	126
9.4.1	Invertibility Analysis of the Secret Key	126
9.4.2	Reduction to Module-LWE	127
9.5	Implementation and Performance Evaluation	130
10	Conclusions	132
	Bibliography	134

List of Figures

1.1	Relation of D3.2 to other PiQASO WPs and Deliverables	19
4.1	System representation under distinct security paradigms	55
4.2	The variables and the functions for the updatable cloud storage	57
4.3	The flow between the server and clients	58
5.1	HFE: Protocol Flow Example.	79
5.2	Oracles in the IND-FE-CPA game for an FE scheme FE and PPT adversary $\mathcal{A}$: the encryption oracle ENCRYPT and the key derivation oracle KEYDER.	81
5.3	Oracles in the IND-HE-CPA game for an HE scheme HE and PPT adversary $\mathcal{A}$: the encryption oracle ENCRYPT and the evaluation oracle EVAL.	82
5.4	Oracles in the IND-HFE-CPA game for a HFE scheme HFE and PPT adversary $\mathcal{A}$: The encryption oracle ENCRYPT, the key derivation oracle KEYDER, the evaluation oracle EVAL and a decryption oracle DECRYPT.	83
6.1	Software profiling of ML-KEM on a 32-bit platform: share of the execution time spent in hashing, polynomial computation and encoding for key generation, encapsulation and decapsulation.	90
6.2	Software profiling of ML-DSA on a 32-bit platform: share of the execution time spent in hashing, polynomial computation and encoding for key generation, signing and verification.	91
6.3	Top-level architecture of the NTT/INTT accelerator. Coefficients enter through the input permutation and a ping-pong coefficient memory. A single radix-2 butterfly unit, fed by the on-the-fly twiddle generator, runs the transform stage by stage between the two banks. The output path applies the N^{-1} scaling, used for the INTT only, and buffers the result before the AXI-Stream master.	94
6.4	On-the-fly twiddle-factor generator. Four interleaved recurrence streams, seeded from the headstart memory and advanced by ψ_{base}^4 , fill the four-cycle Montgomery multiplier. This produces one valid factor per clock.	96
6.5	Unified Cooley–Tukey / Gentleman–Sande radix-2 butterfly unit. The transform direction selects between the two dataflows. The shared modular multiply-and-reduce path sets the pipeline depth.	97
6.6	Modular addition primitive: add, then conditionally subtract q	98
6.7	Modular subtraction primitive: subtract, then conditionally add q	98
6.8	Montgomery reduction unit (radix $R = 2^{16}$), in three pipeline stages: the multiply that forms m , then the multiply-add-shift that forms u , then the conditional subtraction.	99

6.9	Architecture of the hardware accelerator for the ML-KEM NTT.	100
7.1	Block diagram of the accelerator for both variants.	108
7.2	PS-PL integration of the Accelerator.	109
8.1	PiQASO PQ-TLS as part of PQCaaS component	113

List of Tables

2.1	Cryptographic functionalities common across all PiQASO use cases, as identified in D2.1.	22
2.2	Roadmap of PiQASO cryptographic design and implementation deliverables.	26
3.1	Overview of PiQASO SDK Functionalities	29
3.2	AES-GCM Encryption Function	30
3.3	AES-GCM Encryption Input Structure	30
3.4	AES-GCM Encryption Output Structure	30
3.5	AES-GCM Encryption Memory Cleanup	31
3.6	LMS/HSS API Functions	31
3.7	Supported LMS/HSS Parameter Sets	31
3.8	LMS Key Generation	31
3.9	LMS Signing	32
3.10	LMS Verification	32
3.11	LMS Memory Management Functions	32
3.12	XMSS/XMSS API Functions	32
3.13	Supported XMSS and XMSS Parameter Sets	33
3.14	XMSS Key Generation	33
3.15	XMSS Signing	33
3.16	XMSS Verification	34
3.17	XMSS Memory Management Functions	34
3.18	ML-KEM API Functions	34
3.19	Supported ML-KEM Security Levels	34
3.20	ML-KEM Key Generation	34
3.21	ML-KEM Encapsulation	35
3.22	ML-KEM Decapsulation	35
3.23	ML-KEM Memory Management Functions	35
3.24	ML-DSA API Functions	36
3.25	Supported ML-DSA Security Levels	36
3.26	ML-DSA Key Generation	36
3.27	ML-DSA Signing	36

3.28 ML-DSA Verification	36
3.29 ML-DSA Memory Management Functions	37
3.30 TLS 1.3 Configuration	37
3.31 TLS API Functions	37
3.32 TLS Context Creation	38
3.33 TLS Connection Lifecycle	38
3.34 TLS Application Data Functions	38
3.35 Custom Transport Callback Interface	38
3.36 Self-Signed Certificate Generation	38
3.37 Execution time (ms) for ML-KEM operations. Values are reported as mean $\pm$ standard deviation.	46
3.38 Execution time (ms) for ML-DSA operations. Values are reported as mean $\pm$ standard deviation.	46
3.39 Execution time (ms) for AES encryption and decryption operations. Values are reported as mean $\pm$ standard deviation.	47
3.40 Execution time (ms) for stateful hash-based signature operations. Values are reported as mean $\pm$ standard deviation.	47
3.41 Algorithmic and Coefficient-Level Arithmetic Characteristics of PQC Schemes	49
4.1 The new data type for the updatable cloud storage. The arrays for public keys and secret keys are indexed by a global variable epoch of the integer type. ‘ClientKemKeys’ is declared by a client (the receiver in Figure 4.3) and the components are generated by the server(KeyGen).	56
4.2 Description of the notation for UPKE/UKEM and forward secure cloud storage.	59
5.1 Parameters for Test and Benchmark	86
5.2 Running Time and Memory costs of RLWE-based IPFE	87
5.3 Running Time and Memory costs of HFE	87
5.4 Storage Costs of RLWE-based IPFE	88
5.5 Storage Costs of HFE	88
6.1 Arithmetic context written by the PS over AXI4-Lite before a run.	94
6.2 Validated parameter assignments. A negacyclic size- N transform requires $2N \mid (q - 1)$	95
6.3 Core latency: closed-form model vs. RTL simulation. All values are core-clock cycles at 200 MHz. The model equals the measurement exactly at every size.	99
6.4 End-to-end hardware vs. scalar-software latency at $q = 12289$. Times are the minimum over 30 iterations. Cycles are APU cycles at 1.2GHz ($1 \mu s = 1200$ cyc). Speedup = $SW \mu s / HW \mu s$	101
6.5 Hardware per-call breakdown, in APU cycles at 1.2GHz. Twiddles are pre-loaded, so they are excluded. Here <code>hw_wait</code> is DMA plus core plus poll. The DMA-issue phase is not shown; it is the difference between these three columns and the totals of Table 6.4.	101

6.6	Simulated core-only latency vs. on-board <code>hw_wait</code> . The gap of about $10.8\ \mu\text{s}$ is the fixed cost of starting the transfer and detecting completion. It shows no growth with N	102
7.1	Profile of the HQC reference software (PQClean C, -O3, no NEON) on Cortex-A53. Times are mean per-call wall-clock in μs ; each KEM round-trip issues six <code>vect_mul</code> calls. <code>vect_mul</code> accounts for about 96% of total runtime.	106
7.2	Accelerator per-call latency: cycles and time at f_{max} (v1 270 MHz, v2 200 MHz).	110
7.3	Whole-KEM speed-up over each design's own no-NEON reference: PiQASO (v1 narrower, v2 wider), and PHOENIX (PHX).	110
7.4	Resources comparison at HQC-256. PHOENIX [50] is built on a 28 nm Zynq-7020, PiQASO's Accelerator on a 16 nm XCZU7EV. SEC is slice-equivalent cost $\lfloor \text{LUT}/4 \rfloor + \lfloor \text{FF}/8 \rfloor + 200 \text{ BRAM} + 100 \text{ DSP}$. t_{mul} is the pure multiplication latency and the throughput density is the operand length n over t_{mul}	111
9.1	Notation and Mathematical Structures	119
9.2	Performance comparison: PiQASO construction vs. ML-KEM-768 (ms).	131

List of Abbreviations

Abbreviation	Translation
AB	Advisory Board
AoS	Ace of SPADE
ABE	Attribute-Based Encryption
AKE	Authenticated Key Exchange
AQAP	Allied Quality Assurance Publications
ARKG	Asynchronous Remote Key Generation
Adv	Adversary
B/FV	Brakerski/Fan-Vercauteren
CAI	Crypto Asset Inventory
CBOM	Cryptographic Bill Of Materials
CISA	Cybersecurity and Infrastructure Security Agency
CKKS	Cheon-Kim-Kim-Song
CM	Countermeasure
COP	Common Operational Picture
CRM	Continuous Risk Management
CU	Ciphertext Updatability
DA	Data Analyst
DL	Deliverable Leader
DO	Data Owners
DoA	Description of Application
DoS	Denial of Service
DT	Deliverable Team
EC/EDA	European Commission/European Defence Agency
ECC	Elliptic Curve Cryptography
ENISA	European Union Agency for Cybersecurity
ESPAS	European Strategy and Policy Analysis System
EUCC	European Union Cybersecurity Certification
FE	Functional Encryption
FHE	Fully Homomorphic Encryption
FIA	Fault Injection Attack
FIPS	Federal Information Processing Standards
FMN	Federated Mission Networking
FRI	Fast Reed-Solomon Interactive Oracle Proof of Proximity
HE	Homomorphic Encryption
HTFE	Homomorphic Transciphering Functional Encryption
GA	Grant Agreement
GDPR	General Data Protection Regulation

GIS	Geographic Information System
HE	Homomorphic Encryption
HQ	Headquarters
HSM	Hardware Security Modules
HTFE	Homomorphic Transciphering Functional Encryption
IAM	Identity and Access Management
IETF	Internet Engineering Task Force
IKE	Internet Key Exchange
IND-CPA	Indistinguishability under Chosen-Plaintext Attack
IPFE	Inner-Product Functional Encryption
IPR	Intellectual Property Rights
IPSEC	Internet Protocol Security
IR	Internal Reviewers
KAT	Known Answer Test
KC	Key Curator
KEK	Key Encryption Key
KEM	Key Encapsulation Mechanism
KPI	Key Performance Indicator
LACI	Lightweight Asymmetric Cryptography Implementations
LCA	Loosely Coupled Acceleration
LWE	Learning With Errors
MCFE	Multi-Client Functional Encryption
MitM	Man-in-the-Middle
MLWE	Module Learning With Errors
MVP	Minimum Viable Product
NIST	National Institute of Standards and Technology
NSA	National Security Agency
NTT	Number Theoretic Transform
OEM	Original Equipment Manufacturer
PHE	Partially Homomorphic Encryption
PKE	Public Key Encryption
PKI	Public Key Infrastructure
PPT	Probabilistic Polynomial Time
PQC	Post-Quantum Cryptography
PQCaaS	Post-Quantum-Cryptography-as-a-Service
PK	Public Key
QLDPC	Quantum Low Density Parity Check
QR	Quantum Resistant
RLWE	Ring Learning With Errors
SCA	Side-Channel Attack
SHE	Somewhat Homomorphic Encryption
SK	Secret Key
SL	Subjective Logic
SOC	Security Operation Center
SSI	Self-Sovereign Identity
SSL	Secure Sockets Layer
TA	Trusted Authority

TAF	Trust Assessment Framework
TCA	Tightly Coupled Acceleration
TCB	Trusted Computing Base
TEE	Trusted Execution Environment
TFHE	Torus FHE
TLS	Transport Layer Security
TOC	Tactical Operation Center
U	Users
UAV	Unmanned Aerial Vehicle
UC	Universal Composability
UPKE	Updatable Key Encryption
VoIP	Voice over IP
VPN	Virtual Private Network
ZKP	Zero-Knowledge Proof
ZTA	Zero Trust Architecture

Chapter 1

Introduction

1.1 Scope and Purpose

Cryptographic services constitute the backbone of trust in modern service graph chain infrastructures, enabling secure communications, authenticated command and control, controlled information sharing, and trustworthy platform operation across heterogeneous systems ranging from embedded endpoints to backend infrastructures. The advent of quantum computing challenges this trust model by undermining the hardness assumptions of widely deployed public-key cryptography, thus threatening the confidentiality, integrity, and authenticity guarantees that these infrastructures rely upon. **D3.1** framed this challenge and positioned the transition to Post-Quantum Cryptography (PQC) as a system-level transformation affecting protocols, PKI ecosystems, and deployability constraints, while establishing the theoretical and algorithmic foundations of the **PiQASO PQC Ensemble**.

In this context, the scope of **D3.2** is to mark the transition from design to implementation, delivering the *first operational software artifacts* of the PiQASO PQC Ensemble. The deliverable pursues four main objectives. First, it delivers the **first version of the PiQASO SDK** (Chapter 3), a portable and modular software library providing far-edge devices with standalone implementations of ML-KEM, ML-DSA, One-Time Signatures, HQC, and XMSS/LMS, together with a first realization of **PQ-TLS 1.3**. Second, it delivers the **first version of PQCaaS**, PiQASO's Post-Quantum Cryptography as-a-Service offering, which extends quantum-secure capabilities to legacy and resource-constrained devices unable to execute PQC primitives locally, including support for *mixed TLS certificates* (Chapter 8). Third, it provides concrete implementations of the advanced cryptographic functionalities whose mathematical foundations were established in D3.1 [46], namely the **forward-secure cloud storage scheme** based on Updatable Public Key Encryption (Chapter 4) and **Homomorphic Functional Encryption** (Chapter 5). Fourth, it details PiQASO's progress on **algorithmic optimization and HW-based acceleration**, covering the hardware realization of NTT acceleration for lattice-based constructions (Chapter 6), a dedicated HQC hardware accelerator (Chapter 7), and **LWE decomposition** (Chapter 9), a new algebraic construction that accelerates advanced protocols at the protocol level by equipping LWE secret keys with field-like operational semantics.

Thus, compounding all the above, D3.2 is the second deliverable in a series of technical PiQASO deliverables dedicated to the design and implementation of the PiQASO PQC Ensemble, offering the first concrete functionalities and primitives to be integrated and tested in the context of the PiQASO use case demonstrators. As detailed in Chapter 2, the scope of this deliverable is directly informed by the functionalities identified in D2.1 [45] as common to all PiQASO use cases, ensuring that the SDK and PQCaaS artifacts delivered here provide the shared cryptographic substrate upon which all use case demonstrators will build their quantum-secure capabilities.

1.2 Relation to other WPs and Deliverables

As illustrated in Figure 1.1, D3.2 is shaped by, and in turn shapes, the activities of several PiQASO work packages. **WP1** provides the *rules and regulations, and PiQASO framework requirements* that constrain and guide the design of the SDK and PQCaaS artifacts delivered in this deliverable, ensuring their compliance with the overarching PiQASO framework. **WP2**, through D2.1 [45], provides the *use case requirements and needs identified for the PQ transition*. As detailed in Chapter 2, the scope of D3.2 is directly informed by the cryptographic functionalities identified in D2.1 as common to all PiQASO use cases, ensuring that the software artifacts delivered here address concrete, use-case-driven requirements rather than a purely theoretical agenda.

Within **WP3** itself, D3.2 builds directly on the roadmap, mathematical foundations, and architectural positioning established in **D3.1** [46], and its own initial definitions of the PiQASO PQC enablers will, in turn, *guide the development of the final versions of these components, as well as arithmetic and hardware-based acceleration capabilities*, to be reported in **D3.3** [48]. Looking outward, D3.2 provides the software and service artifacts that **WP4** will integrate and evaluate in the context of the use cases. Specifically, the *evaluation of PiQASO modalities in the context of the use cases*, according to the plan set forth in D2.1, will leverage the PiQASO capabilities delivered here, including the **PQ Software SDK**, **PQCaaS**, and **PQ-TLS**, with this evaluation to be reported in **D4.1** [47]. The outputs of D3.2 further provide input to **WP5**, where the *dissemination of PQC knowledge and training* of all concerned stakeholders will be facilitated through the construction of the PiQASO Academy. Finally, in the context of **WP6**, *dissemination and exploitation* activities will be carried out to facilitate the adoption of the PiQASO artifacts delivered in this deliverable in real-world organizations.

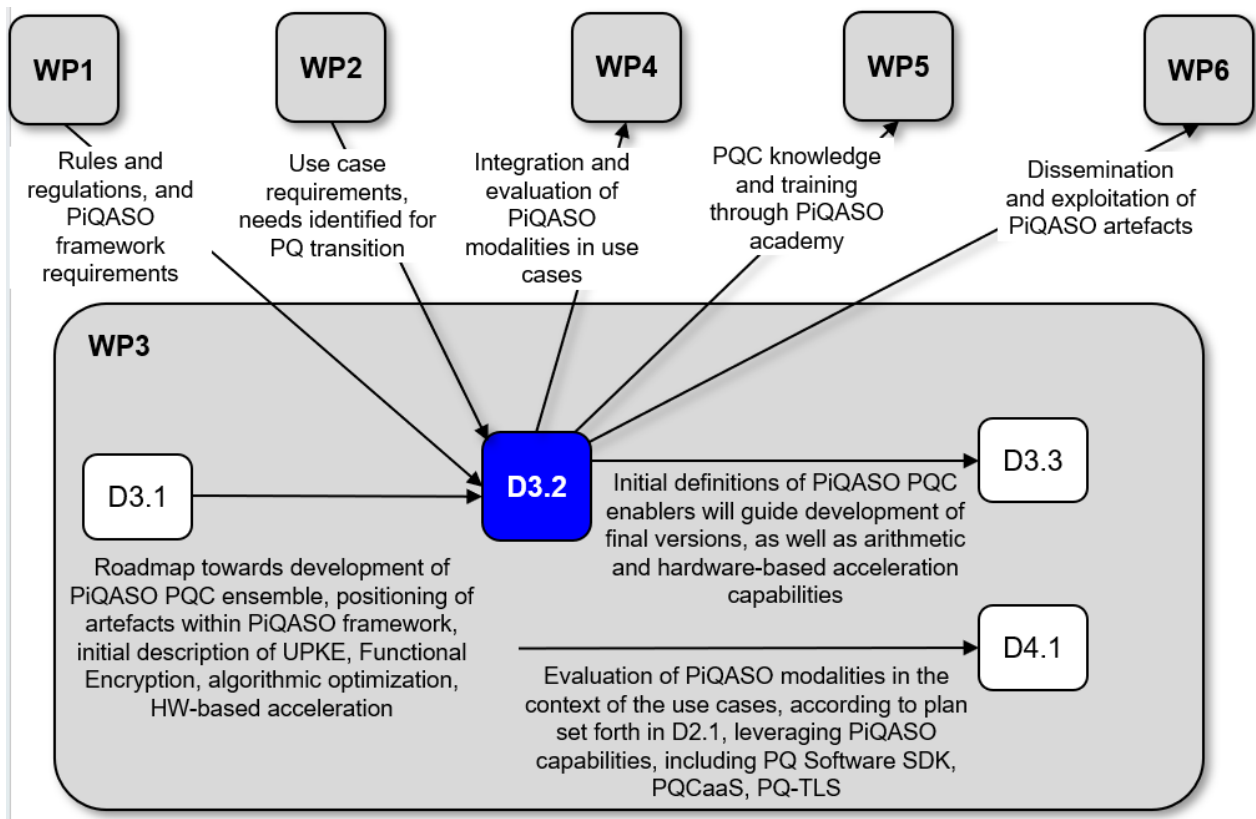


Figure 1.1: Relation of D3.2 to other PiQASO WPs and Deliverables

1.3 Deliverable Structure

The remainder of this deliverable is structured as follows:

Chapter 2 provides a concise roadmap of the cryptographic design and implementation efforts of PiQASO, situating the contributions of D3.2 within the broader trajectory established in D3.1 [46] and outlining the work to be completed in D3.3 [48].

Chapter 3 details the **first version of the PiQASO SDK**, covering its software interfaces for AES-GCM, LMS/HSS and XMSS/XMSS^{MT} signatures, ML-KEM, ML-DSA, and PQ-TLS 1.3, together with unit testing, performance benchmarking, and the roadmap for algorithmic optimization of lattice-based algorithms.

Chapter 4 presents the **Forward Secure Cloud Storage Scheme**, providing a concrete implementation of the cloud storage scheme with a trusted Key Provider, as well as the specification of an advanced variant for the untrusted Key Provider setting.

Chapter 5 introduces **Homomorphic Functional Encryption (HFE)**, a scheme-agnostic construction combining Homomorphic Encryption and Functional Encryption to replace the trusted authority of traditional FE with a key curator that cannot learn the function being evaluated, with direct applicability to the TELLU and PARTICULE healthcare use cases.

Chapter 6 details the hardware-based acceleration of the **Number-Theoretic Transform (NTT)** for lattice-based constructions, complementing the algorithmic, software-level optimization roadmap introduced in Chapter 3.

Chapter 7 presents a dedicated **hardware accelerator for HQC**, targeting the binary polynomial multiplication bottleneck that dominates the execution time of the constant-time HQC reference implementation.

Chapter 8 describes the extension of PQCaaS to support **mixed TLS certificates**, detailing the NGINX-based PQ-TLS termination and mediation architecture that enables far-edge devices to authenticate via classical TLS while PQCaaS transparently mediates PQ-TLS communication with the receiving point.

Chapter 9 introduces **LWE decomposition**, a new construction that augments LWE secret keys with invertibility via polynomial matrices, enabling field-like operations that accelerate advanced protocols such as threshold decryption and Attribute-Based Encryption at the protocol level.

Chapter 10 concludes the deliverable.

Chapter 2

Updates in the Crypto Designs of PiQASO

This chapter provides a concise roadmap of the cryptographic design and implementation efforts of PiQASO, situating the contributions of the present deliverable (D3.2) within the broader trajectory established in D3.1 and outlining the work to be completed in D3.3. The purpose of this roadmap is twofold: first, to clarify which cryptographic primitives and constructions have transitioned from theoretical specification to concrete software implementation as part of the **PiQASO SDK**, and second, to set clear expectations regarding the remaining milestones towards the **QR Trusted Computing Base (QR-TCB)**, additional cryptographic functionalities, and the evaluation of these components across the PiQASO use cases.

As established in D3.1 [46], the PiQASO PQC Ensemble is designed to provide post-quantum security guarantees across the entire compute continuum, from cloud-based back-ends down to far-edge embedded devices. D3.1 laid the theoretical and architectural foundations for this ensemble, covering the PQC migration roadmap (crypto agility, hybridization, and interoperability with legacy infrastructures), a state-of-the-art analysis of the targeted PQC constructions (Key Encapsulation Mechanisms, Digital Signatures, QR Trusted Computing architectures, and lattice trapdoors and gadgets), the mathematical foundations of Updatable Public Key Encryption (UPKE) for forward security, the Functional Encryption capabilities underpinning the ACE of SPADE (AoS) scheme, and the initial software/hardware co-design approach for accelerating lattice-based arithmetic on resource-constrained far-edge devices.

Building on these foundations, D3.2 marks the transition from design to implementation, delivering the first operational software artifacts of the PiQASO Ensemble, namely the first version of the **PiQASO SDK** and the first version of **PQCaaS**. Together with the **QR-TCB**, these constitute the three complementary variants through which PiQASO's PQC primitives are made available across the layers of the entire compute continuum, ranging from the cloud to the far edge.

The scope of D3.2 is directly informed by the definition of the functionalities that need to be tested in the context of all use cases as part of the first round of experimentation as described in D2.1. As a result of this analysis, we identified the cryptographic functionalities which are common to *all* PiQASO use cases and can be seen as the common baseline for achieving post-quantum security, regardless of application or industrial domain. Across the use cases, **the PiQASO SW SDK is used to enable far-edge devices to run PQC primitives locally**, so as to perform key establishment and authentication through PQ-TLS, as well as authentication through message integrity. Conversely, for edge devices which are not capable of supporting the execution of the complex mathematical operations required in the context of PQC schemes, the PiQASO **PQCaaS** functionality enables offloading such operations to the server so that far-edge devices are able to manifest quantum-secure authentication, and it can also act as an intermediate server executing PQ-TLS on their behalf (see Chapter 8 for the architecture of PQCaaS as a PQ-TLS intermediate).

Based on this, benchmarking of the overhead introduced by PQ-TLS will be performed against the TLS 1.3 protocol currently used for authentication across all use cases. As described in D2.1 [45] and D3.1 [46], PQCaaS supports the provision of **mixed TLS certificates**, allowing a far-edge device to authenticate via TLS through PQCaaS, while PQCaaS in turn handles communication and authentication establishment with the other party. This will be further analyzed in Chapter 8 and is **one of the core innovations of PiQASO for providing authentication capabilities belonging to any application or industrial domain, as it is comparable to that exposed by commercial providers such as Cloudflare, which PiQASO replicates through an open-source, containerized PQCaaS software stack.**

Jointly, the SW SDK and PQCaaS mediate communication for the efficient execution of quantum-secure **data sharing agreements**. In practice, this means PQCaaS retains the logic required to perform automated re-encryption of data exchanged from the far edge (via traditional TLS) to the receiver (via PQ-TLS), together with the key management required for more complex data sharing agreements. The latter requires the establishment of two types of key chains, specifically **user-specific** key chains and **attribute-policy** key chains. All of the functionalities described above are common to and will be tested in all PiQASO use cases.

The elevation of PiQASO's PQC primitives to trusted applications, i.e., their integration into the **QR-TCB**, together with new acceleration and optimization features, is deferred to the second reporting period (RP2). *At the time of writing this deliverable, however, the SW SDK has already been instantiated in the context of a QR-TCB in isolation, and is already running as a trusted application. This work will be documented in further detail in the context of D4.1 [47].* Benchmarking of this trusted-application instantiation is already underway, and a comparative evaluation of the SDK running as a trusted versus an untrusted application will be reported as part of D4.1, alongside further experimentation of the SW SDK and PQCaaS in the context of the PiQASO use cases.

Table 2.1 summarizes the cryptographic functionalities identified in D2.1 as common to all PiQASO use cases.

Table 2.1: Cryptographic functionalities common across all PiQASO use cases, as identified in D2.1.

Functionality	Description
PiQASO Software SDK	The PiQASO SW SDK, providing far-edge devices with local, standalone software implementations of PQC primitives for key establishment and authentication without reliance on backend connectivity for every operation.
PQ-TLS Authentication	PQ-TLS-based authentication, combining ML-KEM key establishment with ML-DSA signatures, used to establish trust between far-edge devices and use case backends. This can either be performed through the PQ-TLS modality provided by the PiQASO SDK for devices which are able to support the required underlying mathematical operations, or by using the PQCaaS Server as an intermediary where the PQ-TLS logic is offloaded.
Mixed TLS Certificates	Certificates combining traditional (e.g., x509) and post-quantum public keys/signatures, enabling PQCaaS to support an authentication scheme that combines classical and post-quantum cryptography for the authentication of devices which do not possess the capability to support the mathematical operations required for executing PQC (see Chapter 8). This entails the support of hybrid certificates combining traditional and ML-KEM-based public keys within a single credential, supported by PQCaaS to enable a smooth, backward-compatible migration path toward full PQC adoption.
PiQASO QR-TCB	The QR Trusted Computing Base is used for secure key storage, and elevates PiQASO's PQC primitives to trusted applications for isolated, hardware-protected execution of security-critical operations. PiQASO will investigate which are the sensitive operations that need to run in isolation (trusted world), and which can run outside of the enclave in the user space (untrusted world).

2.1 PiQASO Software Development Kit (SDK)

The central contribution of D3.2 is the release of the **first version of the PiQASO SDK**, described in detail in Chapter 3. The SDK provides software implementations of lightweight asymmetric cryptographic primitives, specifically:

- **ML-KEM**, the standardized lattice-based Key Encapsulation Mechanism, providing quantum-resistant key establishment. It is also used in the context of PQ-TLS scheme, for augmenting the handshake phase with post-quantum security;
- **ML-DSA**, the standardized lattice-based Digital Signature Algorithm, providing quantum-resistant authentication and integrity guarantees;
- **One-Time Signatures (OTS)**, offering a lightweight, hash-based alternative for signature operations in constrained settings where minimizing computational and communication overhead is critical, and used to maintain and safeguard the auditability of data sharing transactions by storing these signatures, in association with the corresponding data sharing agreements, on the PiQASO quantum-resistant **Blockchain** infrastructure. This approach also allows PiQASO to assess the impact of PQ-enabled signatures on the management of on-chain data;
- **HQC**, a standardized code-based Key Encapsulation Mechanism used to construct hybrid KEM configurations, diversifying the hardness assumptions underlying the SDK's key establishment capabilities; and
- **XMSS/LMS**, two stateful hash-based cryptographic signature schemes approved by NIST and the NSA (under CNSA 2.0) to secure critical infrastructure against quantum computing attacks. XMSS uses WOTS+ (Winternitz One-Time Signature) which includes bitmasks for stronger security bounds, and L-trees for key compression. LMS typically yields better performance (fewer clock cycles) for key generation, signing, and verification, while XMSS generally requires more memory.

These primitives aim to enable the execution of post-quantum cryptographic operations on far-edge and edge devices, i.e., the tier of the compute continuum with the most constrained computational, memory, and energy budgets. This is complemented by the **PiQASO PQCaaS**, in the sense that while the SDK equips devices at the edge with the standalone software capability to perform quantum-secure operations locally, without reliance on backend connectivity for every cryptographic operation, the PQCaaS is able to provide PQC capabilities to devices which are not able to support such advanced crypto, by offloading the corresponding mathematical operations.

The first version of the SDK is also the first PiQASO component to provide the high-assurance **PQ-TLS 1.3** protocol. The PiQASO implementation of PQ-TLS 1.3 is based on lattice-based cryptography, employing ML-KEM for key establishment, and has already been enhanced with ML-DSA signatures, thus unlocking the use of quantum-resistant signatures both for the verification of the long-term identities of communicating parties and for the authentication of ephemeral keys.

As already indicated in D2.1, the SDK constitutes the basis for *all cryptographic operations across the PiQASO framework*. Consequently, it is the component that will be evaluated in the context of **all PiQASO use cases**, serving as the common cryptographic substrate upon which the use case demonstrators will build their quantum-secure functionalities. This evaluation activity is expected to be reported progressively as the use case testbeds mature.

2.2 PQC-as-a-Service (PQCaaS)

Complementing the SDK, D3.2 also delivers the first version of **PQCaaS**, PiQASO's Post-Quantum Cryptography as-a-Service offering, already released as an open-source, containerized software stack. PQCaaS provides **PQ-TLS 1.3**, the same protocol variant offered by the SDK (Section 2.1), following the same set of **harmonized interfaces exposed by conventional TLS 1.3 implementations**. This allows communicating parties to leverage the established session key to subsequently execute **AES-GCM encryption and decryption**.

The current version of PQCaaS already provides and supports **hybrid TLS certificates**, alongside both traditional x509 certificates and ML-KEM-based x509 certificates (see Chapter 8). This first version of PQCaaS also supports the first version of **Updatable Public Key Encryption (UPKE)** (see Chapter 4), and supports the construction and maintenance of the two key chain types introduced above, namely **user-specific** and **attribute-policy-based key chains**. User-specific key chains are currently supported through an updatable hybrid encryption scheme. Finally, this first version of PQCaaS exposes the **Functional Encryption** capabilities described in D3.1 [46].

*The current focus and immediate next step for PQCaaS is the enrichment of PiQASO's interoperability, by enabling devices that rely on different PQC artifacts to communicate with one another, and with the PiQASO crypto agility layer, so as to seamlessly and efficiently leverage any of the variants introduced above (SDK, PQCaaS, QR-TCB). In this regard, PiQASO aims to provide an **efficient encoding/decoding schema capable of transforming the output of any cryptographic operation (ciphertext, signature) into a form that can be correctly interpreted by the receiving entity**. This addresses a gap identified during the implementation activities performed in the context of the PiQASO use cases, namely that the SDK and PQCaaS were not yet harmonized, and achieving this harmonization to enable full interoperability between them is a priority going forward.*

2.3 Roadmap for Next Actions

While D3.2 delivers the SDK as a software-only implementation, D3.3 will extend this work along three complementary directions, namely the **construction of the QR-TCB**, the **integration of additional cryptographic functionalities**, and **continued use case-driven evaluation**. The SDK delivered in D3.2 forms the direct basis for the construction of the **QR-TCB**, to be presented in D3.3. This transition will be carried out in two steps:

1. **Enclave placement.** The SDK, as implemented in D3.2, will first be placed in its entirety into a trusted execution enclave, establishing an initial isolated execution environment for all cryptographic operations.
2. **Profiling and partitioning.** The constituent algorithms of the SDK will then be profiled in detail in order to identify which operations are sensitive and must execute in isolation, and which can safely continue to execute in user space (i.e., the untrusted world). As a guiding principle, operations involving the management of, or access to, cryptographic key material will be confined to the isolated environment, while operations such as authentication and the handling of public key certificates (which do not require access to sensitive key material) will be permitted to execute in the untrusted world.

This two-step approach allows the QR-TCB to minimize the attack surface of the isolated environment, restricting it to only those operations that strictly require protection, while retaining performance and flexibility for the remaining operations.

In parallel to the QR-TCB construction, D3.3 will extend the SDK with additional cryptographic functionalities:

- **Updatable Public Key Encryption (UPKE):** The UPKE scheme, whose mathematical foundations and forward-security guarantees were established in D3.1 [46], will be integrated into the SDK, providing key-evolving encryption capabilities to complement the KEM, signature, and OTS primitives already available.
- **Lattice-based Attribute-Based Encryption (ABE):** PiQASO will also investigate the possibility for the integration of newly designed schemes, in particular lattice-based ABE, as candidates for inclusion in the SDK. The outcome of this investigation, and the extent of any resulting implementation, will be reported in D3.3.
- **Additional signature primitives:** The SDK will be further enriched with additional post-quantum signature schemes, namely **FN-DSA** and **SPHINCS+**, broadening the range of signature primitives available to PiQASO use cases.
- **Full UPKE in PQCaaS:** Building on the first version delivered in D3.2 (Section 2.2), PQCaaS will provide a full-blown version of UPKE, as described in the corresponding chapter.
- **Extended Functional Encryption:** The next version of PQCaaS's Functional Encryption capability will embed homomorphic encryption in order to transcipher the function being invoked. This requirement emerged directly from the use cases. For example, in the eHealth use case, when a user (e.g., a hospital) issues a query over text data, the query itself must also be transciphered.

In addition, D3.3 will include a detailed **profiling of performance and computational resource usage** when the SDK operates as a trusted versus an untrusted application, examining the composition of operations executed inside and outside the trusted environment. This will be complemented by a benchmarking exercise against existing solutions (e.g., **PQClean**), comparing the overhead introduced by PiQASO's approach against these baselines.

2.3.1 Evaluation of PiQASO Functionalities in the Use Cases

As noted above, the SDK (and, by extension, its evolution into the QR-TCB and its extended functionality set) underpins the cryptographic capabilities exercised by all PiQASO use case demonstrators. D3.3 will therefore continue to report on the integration and evaluation of these components within the use case testbeds, building on the evaluation activities initiated following the D3.2 SDK release.

Regarding the functionalities to be tested in the context of each PiQASO use case, as detailed in D2.1, each use case maps a subset of the project's cryptographic functionalities onto its own operational architecture, with a phased rollout across the first and second reporting periods (RP1, RP2). Across all use cases, a common experimentation direction entails the evaluation of PQ-TLS (typically combining ML-KEM for key encapsulation with ML-DSA-based authentication) and PQ Signatures in RP1 to secure control-plane communications and establish authenticated channels, while the QR-TCB is evaluated in essentially every use case, spanning both RP1 and RP2, to provide secure key storage and a trusted execution environment for PiQASO's cryptographic enablers. Hardware Acceleration, typically used to offload computationally intensive ML-KEM/ML-DSA operations and preserve real-time performance in latency-sensitive settings such as V2X communications (Automotive, UC1), robot fleets (Automation, UC2), SCADA telemetry (Energy, UC4), and UAV control links (UC8), is a subject of RP2.

Beyond this shared baseline, functionalities are allocated according to each domain's specific needs. **PQCaaS** is going to be leveraged in the context of all use cases and is in particular focus in use cases where legacy or resource-constrained components cannot host the full SDK locally. For example, this includes offloading certificate management for legacy SCADA components in the Energy use case, mediating CMS authentication for constrained clients in Online Media (UC7), and supporting centralized key governance for multi-ground-station UAV scenarios (UC8). **AKE (Authenticated Key Exchange)** is planned for RP2 in scenarios requiring ephemeral session establishment, such as robot-to-robot interactions in Automation (UC2) and SUT-to-sensor sessions in the Aerospace use case (UC6.1). **UPKE** and **ABE** are planned to be evaluated in RP2 for data-at-rest protection and fine-grained, policy-based access control, for instance, encrypting AMR telemetry and restricting rosbag/map access by role in Automation (UC2), protecting IP in aerospace test logs in the Aerospace use case (UC6.1), securing MORE's energy production data in Energy (UC4), and enabling policy-based article sharing in the Online Media use case (UC7), where both UPKE variants (trusted-server and fully lattice-based) are explored. The eHealth use case (UC5.1) additionally evaluates Functional Encryption for performing queries and operations on encrypted numerical patient data, and Over-The-Air (OTA) update signing with ML-DSA/FN-DSA, while the Aerospace (UC6) and Automotive (UC1) use cases also assess alternative signature primitives (FN-DSA) alongside the ML-DSA baseline. The Smart Ambulance (UC5.2) and Transportation (UC9) use cases employ a more compact subset of functionalities including PQ-TLS, signatures, QR-TCB, and hardware acceleration, reflecting their more constrained mobile and edge deployment contexts.

It should be highlighted that, since the PiQASO SDK constitutes the foundation for all cryptographic operations performed across the PiQASO Ensemble, it is not confined to any single domain-specific table, but underlies the functionalities evaluated in every one of the eleven use cases. Each use case's selection of PQ-TLS, PQ Signatures, QR-TCB, PQCaaS, AKE, UPKE, and ABE functionalities therefore represents a **domain-specific** instantiation of capabilities ultimately rooted in the shared SDK, making its evaluation a thread running through all use case demonstrators.

2.4 Summary

Table 2.2 summarizes the roadmap of cryptographic deliverables described in this chapter.

Table 2.2: Roadmap of PiQASO cryptographic design and implementation deliverables.

Deliverable	Focus	Key Outputs
D3.1 [46]	Theoretical and architectural foundations	PQC migration roadmap; SOTA on KEMs, signatures, QR-TCB, trapdoors; UPKE and AoS foundations; SW/HW co-design basis
D3.2	First software implementation	PiQASO SDK (ML-KEM, ML-DSA, OTS, HQC, XMSS/LMS, PQ-TLS 1.3) for far-edge/edge devices; first version of PQCaaS (PQ-TLS 1.3, hybrid/ML-KEM certificates, UPKE, Functional Encryption)
D3.3 [48], D4.1 [47]	Trusted execution and functionality extension	QR-TCB (enclave placement and profiling/partitioning); UPKE integration; investigation of lattice-based ABE; continued use case evaluation

Chapter 3

SW-Based Implementation and Library

The **PiQASO SDK** was established to provide **Post-Quantum Cryptography (PQC)** capabilities to far-edge devices, such as *Key Encapsulation Mechanisms (KEMs)* and *digital signature schemes*, across multiple hardware architectures, as detailed throughout this chapter. The SDK is built around several characteristics that make it well suited to such heterogeneous deployment scenarios. One of the most prominent such characteristics is that the **PiQASO SDK is portable and modular**, irrespective of the underlying architecture on which it is deployed. It should be noted that the specific target architecture becomes relevant primarily in the context of **optimization and acceleration**. Specifically, as described later in this chapter, current acceleration capabilities developed in the context of PiQASO are specific to the *Zynq UltraScale+* board. Independently of these architecture-specific optimizations, however, the SDK itself remains portable and modular by design. This modularity means that *the PiQASO SDK can operate in tandem with any cryptographic library that exposes PQC primitives, provided the library conforms to the same set of interfaces exposed to the application layer*. To maximize compatibility, PiQASO adopts the same interface conventions that are already well established in the literature and used by reference libraries, such as **PQClean**. As a consequence of this design choice, any conforming library can be integrated with the SDK, as long as it exposes the same interfaces.

This portability is leveraged to enable a detailed evaluation and comparison between well-established PQC libraries and PiQASO's own implementation, which is fully optimized and accelerated. To this end, the **wolfSSL** library, which provides ML-KEM and ML-DSA and is used by both the *PQCaaS* and *SW Library* components of PiQASO, is integrated in addition to PiQASO's own reference implementation building upon **PQClean**. PiQASO builds upon and extends PQClean, which is used as the baseline reference implementation, since it currently includes no algorithmic optimization or compiler-level optimization (e.g., -O3 and SIMD optimization). This effort will be completed in the second reporting period, further enhanced with the algorithmic optimization and acceleration capabilities of PiQASO through (**SW-HW co-design**), as well as with -O3 compiler optimization. Considering also the integration of **wolfSSL** and **OpenSSL**, **this puts PiQASO in a unique position to perform a comparative evaluation of the leading commodity PQC libraries against its own optimized and accelerated implementation.**

At its core, the PiQASO SDK is a highly optimized C-based software library engineered to provide robust, long-term quantum resistance across the entire computing continuum, bridging cloud infrastructures and resource-constrained edge environments through a unified security framework. A fundamental design objective of the PiQASO SDK is **interoperability across heterogeneous computing environments**. Modern deployments span large-scale cloud data centers, edge gateways, Internet of Things devices, and embedded microcontrollers operating on diverse architectures, including x86_64, ARM Cortex-A, ARM Cortex-M, and RISC-V platforms. To ensure portability without sacrificing performance, the SDK adopts a *native ISO C implementation* that provides deterministic memory management, zero-overhead execution, and universal compiler support. This architecture enables identical cryptographic functionality

to execute efficiently across a wide range of hardware platforms while minimizing computational, memory, and power consumption requirements, which are particularly critical for *edge deployments*.

To ensure cryptographic robustness and avoid the risks associated with implementing complex post-quantum algorithms from scratch, the PiQASO SDK integrates the latest version of the **wolfSSL** cryptographic library as its foundational security engine. The library wolfSSL was selected due to its *lightweight architecture*, extensive security auditing, and optimization for both embedded systems and high-throughput enterprise environments. Leveraging wolfSSL, the SDK inherits mature implementations of symmetric cryptography, standardized hashing algorithms, and cryptographically secure random number generation, providing a **side-channel-resistant** and compliance-oriented foundation upon which post-quantum functionality is built.

For deployment flexibility and maintainability, the complete cryptographic stack is compiled into a **unified shared object library**. This architecture establishes a stable *Application Binary Interface (ABI)* that isolates application code from the underlying cryptographic implementation, enabling updates, security patches, and performance optimizations to be deployed independently of dependent applications. Furthermore, the shared library model reduces system-wide memory consumption by allowing multiple processes to map the same executable code segments into memory simultaneously, thereby improving efficiency in both cloud and edge environments.

Recognizing the diverse software ecosystems utilized across industry, the PiQASO SDK incorporates the **Simplified Wrapper and Interface Generator (SWIG)** to provide automated cross-language interoperability. Rather than requiring manually developed bindings for each target language, SWIG automatically parses the SDK's native C interfaces and generates secure, optimized bindings for languages including *Python, Java, Kotlin, and Node.js*. This approach eliminates many common sources of memory management errors, type-safety violations, and pointer-related vulnerabilities while simplifying integration into data science workflows, enterprise applications, mobile platforms, and web services. Consequently, developers can access the full functionality of the SDK through familiar language-native interfaces without requiring expertise in low-level systems' programming.

The cryptographic capabilities of the PiQASO SDK are aligned with emerging post-quantum standards and support multiple **NIST security levels ranging from Level 1 to Level 3** (and, for ML-KEM and ML-DSA specifically, up to Level 5). For quantum-resistant key establishment and public-key encryption, the SDK implements the **Module-Lattice-Based Key Encapsulation Mechanism (ML-KEM)**, derived from the CRYSTALS-Kyber framework. ML-KEM offers strong security guarantees alongside efficient execution and compact key sizes, making it suitable for a broad range of communication scenarios. For authentication, integrity verification, and non-repudiation, the SDK provides the **Module-Lattice-Based Digital Signature Algorithm (ML-DSA)**, derived from CRYSTALS-Dilithium, enabling secure digital signatures for enterprise transactions, software distribution, and identity management. In addition to lattice-based cryptography, the SDK supports *stateful hash-based signature schemes*, including the eXtended Merkle Signature Scheme (XMSS) and the Leighton-Micali Signature Scheme (LMS). These schemes rely solely on the security properties of cryptographic hash functions and therefore provide an alternative security model that remains resilient against potential future advances in lattice cryptanalysis. The SDK natively manages the state-tracking requirements associated with XMSS and LMS, preventing signature reuse and supporting long-term security-critical applications such as secure boot infrastructures, firmware update mechanisms, and distributed ledger technologies.

Beyond securing individual cryptographic operations, the PiQASO SDK extends quantum-resistant protections to network communications through direct integration with the **TLS 1.3** protocol. This integration addresses the growing threat of *harvest-now-decrypt-later* attacks, in which adversaries collect encrypted communications today with the intention of decrypting them once practical quantum computers become available. By incorporating ML-KEM into the TLS 1.3 key exchange process and EC256P into endpoint authentication mechanisms, the SDK establishes quantum-resistant session keys and identity verifica-

tion while preserving the performance, privacy protections, and **perfect forward secrecy** characteristics of the TLS 1.3 standard. As a result, organizations can secure data both at rest and in transit using cryptographic mechanisms specifically designed to withstand future quantum attacks.

Collectively, these architectural and cryptographic capabilities position the PiQASO SDK as a **comprehensive, production-ready solution for post-quantum security**. Through its optimized C foundation, wolfSSL integration, shared object deployment model, automated SWIG-based language bindings, support for standardized post-quantum algorithms, and TLS 1.3 integration, the SDK provides organizations with a scalable and future-proof framework for protecting digital infrastructure across cloud, edge, and embedded computing environments in the post-quantum era.

Table 3.1 summarizes, at a glance, the complete set of functionalities provided by the PiQASO SDK and detailed throughout this chapter, together with a brief description of each and, where applicable, the **NIST security level(s)** that each functionality addresses.

Table 3.1: Overview of PiQASO SDK Functionalities

Functionalities	Description
AES-GCM Authenticated Encryption	Symmetric authenticated encryption primitive (FIPS-197 AES in GCM mode) used for confidentiality and integrity across communication and storage interfaces. Supports 128-, 192-, and 256-bit keys, defaulting to AES-256, and is the symmetric scheme under which the establishment of secure and authenticated communication channels is enabled in PQ-TLS 1.3.
LMS/HSS Hash-Based Signatures	Stateful, hash-based digital signature scheme (NIST SP 800-208) offering long-term quantum resistance derived solely from hash-function security. The SDK exposes parameter sets aligned with the approved parameter set endorsed by NIST based on either using SHA-256 or SHAKE-256 with 192- or 256-bit outputs . This NIST profiling requires that key and signature generation is performed by a secure module. <i>In the context of PiQASO, this will be performed during the second reporting period through the QR-TCB.</i>
XMSS/XMSS ^{MT} Hash-Based Signatures	Stateful hash-based signature scheme (RFC 8391, NIST SP 800-208) providing single-tree and multi-tree (XMSS ^{MT}) variants for extended signing capacity. Security relies exclusively on hash-function properties.
ML-KEM Key Encapsulation	Module-lattice-based KEM (FIPS 203, derived from CRYSTALS-Kyber) used for quantum-resistant key establishment. Supported at NIST Security Levels 1, 3, and 5 (ML-KEM-512/768/1024), enabling shared-secret derivation for symmetric protocols such as AES-GCM and TLS 1.3.
ML-DSA Digital Signatures	Module-lattice-based digital signature scheme (FIPS 204, derived from CRYSTALS-Dilithium) providing stateless authentication, integrity, and non-repudiation. Supported at NIST Security Levels 2, 3, and 5 (ML-DSA-44/65/87), suitable for enterprise transactions and software-distribution signing.
Post-Quantum TLS 1.3 Integration	TLS 1.3 handshake employing ML-KEM-based key exchange. The current configuration operates at NIST Security Level 1 (ML-KEM-512) with ECDSA P-256 endpoint authentication, and supports both socket-based and custom transport callbacks.
Algorithmic Optimization of binary polynomial multiplication (Lattice-Based, HQC)	Ongoing algorithmic optimization that replaces NTT-based polynomial multiplication with an FFT-based, complex-domain convolution technique using fixed-point arithmetic, aiming to reduce modular-reduction overhead in ML-KEM, ML-DSA, and HQC while preserving constant-time execution across all supported security levels.
Generalization of the LWE Problem	New construction that augments the Learning With Errors (LWE) problem with invertible secret keys expressed as polynomial matrices, enabling secret-key-composable operations and more efficient key encapsulation and key-distribution schemes, applicable across the NIST security levels supported by the SDK's lattice-based primitives.

3.1 PiQASO SDK Interfaces

3.1.1 AES-GCM Encryption API

The PiQASO SDK integrates the **Advanced Encryption Standard (AES)** as a core symmetric encryption primitive for providing confidentiality across supported communication and storage interfaces. AES is a standardized block cipher defined in *FIPS-197* and is widely adopted due to its strong security guarantees and high performance in both software and hardware implementations. Within the SDK, AES is primarily used in **authenticated encryption** workflows, ensuring both confidentiality and integrity when combined with secure modes of operation such as Galois/Counter Mode (GCM). AES operates on fixed-size 128-bit blocks and supports key sizes of 128, 192, and 256 bits, with the SDK defaulting to **AES-256** to maximize security margins in long-term and adversarial threat models. Table 3.2 summarizes the functionality provided by the AES-GCM encryption interface.

Table 3.2: AES-GCM Encryption Function

Item	Description
Function	<code>aesgcm_encrypt()</code>
Purpose	Performs authenticated encryption using AES-GCM, providing confidentiality, integrity, and authenticity guarantees.
Prototype	<code>int aesgcm_encrypt(const struct aesgcm_encrypt_in *in, struct aesgcm_encrypt_out **out);</code>
Input Structure	<code>struct aesgcm_encrypt_in</code>
Output Structure	<code>struct aesgcm_encrypt_out</code>
Supported Key Sizes	128-bit (16 bytes), 192-bit (24 bytes), 256-bit (32 bytes)
Recommended Key Size	AES-256 (32 bytes)
IV Handling	Caller may provide a 12-byte nonce or pass NULL to request automatic cryptographically secure nonce generation.
AAD Support	Optional Additional Authenticated Data (AAD) can be supplied and authenticated but not encrypted.
Ciphertext Length	Equal to plaintext length.
Authentication Tag	16-byte GCM authentication tag.
Memory Allocation	Output structures and buffers are allocated by the SDK.
Memory Release	<code>free_aesgcm_encrypt_out()</code>
Success Return	PIQ_SUCCESS
Failure Return	Negative PiQASO error code.

Table 3.3: AES-GCM Encryption Input Structure

Field	Type	Required	Description
key	<code>struct piq_buffer *</code>	Yes	AES encryption key (16, 24, or 32 bytes).
iv	<code>struct piq_buffer *</code>	No	12-byte GCM nonce. If NULL, the SDK automatically generates a secure random nonce.
plaintext	<code>struct piq_buffer *</code>	Yes	Data to be encrypted.
aad	<code>struct piq_buffer *</code>	No	Additional Authenticated Data that is authenticated but not encrypted.

Table 3.4: AES-GCM Encryption Output Structure

Field	Type	Description
iv	<code>struct piq_buffer *</code>	IV/nonce used during encryption (generated or caller supplied).
ciphertext	<code>struct piq_buffer *</code>	Encrypted plaintext output.

tag	struct piq_buffer *	16-byte authentication tag generated during encryption.
-----	---------------------	---

Table 3.5: AES-GCM Encryption Memory Cleanup

Function	Description
free_aesgcm_encrypt_out()	Releases all memory allocated by aesgcm_encrypt(), including the IV, ciphertext, authentication tag, and output structure.

3.1.2 LMS/HSS Digital Signature Functions

The PiQASO SDK provides a complete implementation of the **Leighton-Micali Signature (LMS)** scheme and **Hierarchical Signature System (HSS)** as specified in NIST SP 800-208. LMS is a *stateful* hash-based digital signature algorithm designed to provide long-term quantum-resistant authentication. Unlike lattice-based signature schemes, LMS security relies solely on the collision resistance and preimage resistance of cryptographic hash functions. Because LMS is stateful, every signature operation consumes a **one-time signing key** and advances the internal private key state. Consequently, applications must persist the updated private key returned by the SDK after every signing operation to prevent key reuse and maintain cryptographic security.

Table 3.6: LMS/HSS API Functions

Function	Description
lms_keygen()	Generates an LMS/HSS public-private key pair according to the selected parameter set.
free_lms_keygen_out()	Releases memory allocated during key generation.
lms_sign()	Generates a detached LMS signature and returns the updated private key state.
free_lms_sign_out()	Releases memory allocated during the signing operation.
lms_verify()	Verifies an LMS signature using the corresponding public key.

Table 3.7: Supported LMS/HSS Parameter Sets

Parameter Set	Levels	Tree Height	Winternitz	Approximate Signature Capacity
LMS_PARM_L1_H5_W2	1	5	2	32 signatures
LMS_PARM_L1_H10_W2	1	10	2	1,024 signatures
LMS_PARM_L2_H5_W2	2	5	2	1,024 signatures
LMS_PARM_L2_H10_W2	2	10	2	Approximately 1 million signatures

Table 3.8: LMS Key Generation

Property	Description
Function	lms_keygen()
Purpose	Generates a new LMS/HSS public-private key pair.
Input Structure	struct lms_keygen_in
Output Structure	struct lms_keygen_out
Input Parameter	Selected LMS parameter set (LMS_PARM_*).
Output	Public key and initial private key state.
Important Note	The generated private key represents the initial signing state and must be securely persisted before any signing operation.
Success Return	PIQ_SUCCESS

Failure Return	Negative PiQASO error code.
----------------	-----------------------------

Table 3.9: LMS Signing

Property	Description
Function	<code>lms_sign()</code>
Purpose	Generates a detached LMS signature for a message.
Input Structure	<code>struct lms_sign_in</code>
Output Structure	<code>struct lms_sign_out</code>
Input Parameters	Parameter set, current private key state, and message.
Output	Detached signature and updated private key state.
State Management	Every signature consumes a one-time signing key and advances the private key state.
Security Requirement	The caller must immediately persist <code>updated_private_key</code> and use it for all future signing operations. Reusing an old key state results in double-signing and compromises security.
Success Return	<code>PIQ_SUCCESS</code>
Failure Return	Negative PiQASO error code.

Table 3.10: LMS Verification

Property	Description
Function	<code>lms_verify()</code>
Purpose	Verifies the authenticity and integrity of an LMS signature.
Input Structure	<code>struct lms_verify_in</code>
Required Inputs	Parameter set, public key, original message, and signature.
Success Return	<code>PIQ_SUCCESS</code>
Verification Failure	<code>PIQ_VERIFY_FAILED</code>
Other Failures	Negative PiQASO error code.
Output Structure	None. Verification result is returned through the function return code.

Table 3.11: LMS Memory Management Functions

Function	Description
<code>free_lms_keygen_out()</code>	Releases memory allocated by <code>lms_keygen()</code> , including public and private key buffers.
<code>free_lms_sign_out()</code>	Releases memory allocated by <code>lms_sign()</code> , including signature and updated private key buffers.

3.1.3 XMSS/XMSS Digital Signature Functions

The PiQASO SDK provides a complete implementation of the **eXtended Merkle Signature Scheme (XMSS)** and **XMSS^{MT}** as standardized in RFC 8391 and NIST SP 800-208. XMSS is a *stateful* hash-based digital signature algorithm that derives its security exclusively from the properties of cryptographic hash functions, making it resistant to both classical and quantum attacks. Similar to LMS, each signature operation consumes a one-time signing key and advances the internal private key state. Consequently, applications must securely persist the updated private key returned after every signing operation. **Reusing an older private key state results in one-time key reuse and constitutes a catastrophic security failure.**

Table 3.12: XMSS/XMSS API Functions

Function	Description
<code>xmss_keygen()</code>	Generates an XMSS or XMSS public-private key pair using the selected parameter set.
<code>free_xmss_keygen_out()</code>	Releases memory allocated during key generation.
<code>xmss_sign()</code>	Generates a detached XMSS signature and returns the updated private key state.
<code>free_xmss_sign_out()</code>	Releases memory allocated during the signing operation.
<code>xmss_verify()</code>	Verifies an XMSS signature using the corresponding public key.

Table 3.13: Supported XMSS and XMSS Parameter Sets

Parameter Set	Description
XMSS-SHA2_10_256	Single-tree XMSS with height 10, supporting 1,024 signatures.
XMSS-SHA2_16_256	Single-tree XMSS with height 16, supporting 65,536 signatures.
XMSS-SHA2_20_256	Single-tree XMSS with height 20, supporting approximately 1 million signatures.
XMSSMT-SHA2_20/2_256	Multi-tree XMSS with total height 20 and 2 layers.
XMSSMT-SHA2_20/4_256	Multi-tree XMSS with total height 20 and 4 layers.
XMSSMT-SHA2_40/2_256	Multi-tree XMSS with total height 40 and 2 layers.
XMSSMT-SHA2_40/4_256	Multi-tree XMSS with total height 40 and 4 layers.
XMSSMT-SHA2_40/8_256	Multi-tree XMSS with total height 40 and 8 layers.
XMSSMT-SHA2_60/3_256	Multi-tree XMSS with total height 60 and 3 layers.
XMSSMT-SHA2_60/6_256	Multi-tree XMSS with total height 60 and 6 layers.
XMSSMT-SHA2_60/12_256	Multi-tree XMSS with total height 60 and 12 layers.

Table 3.14: XMSS Key Generation

Property	Description
Function	<code>xmss_keygen()</code>
Purpose	Generates a new XMSS or XMSS public-private key pair.
Input Structure	<code>struct xmss_keygen_in</code>
Output Structure	<code>struct xmss_keygen_out</code>
Input Parameter	XMSS parameter string specifying the desired signature scheme configuration.
Output	Public key and initial private key state.
Important Note	The generated private key represents the initial one-time signature tree state and must be securely persisted before any signing operation.
Success Return	PIQ_SUCCESS
Failure Return	Negative PiQASO error code.

Table 3.15: XMSS Signing

Property	Description
Function	<code>xmss_sign()</code>
Purpose	Generates a detached XMSS signature for a message.
Input Structure	<code>struct xmss_sign_in</code>
Output Structure	<code>struct xmss_sign_out</code>
Input Parameters	Parameter string, current private key state, and message.
Output	Detached signature and updated private key state.
State Management	Every signature operation consumes a one-time signing key and advances the internal XMSS state.
Security Requirement	The caller must immediately persist <code>updated_private_key</code> and use it in all future signing operations. Reusing an older state results in one-time key reuse and compromises security.
Success Return	PIQ_SUCCESS
Failure Return	Negative PiQASO error code.

Table 3.16: XMSS Verification

Property	Description
Function	<code>xmss_verify()</code>
Purpose	Verifies the authenticity and integrity of an XMSS signature.
Input Structure	<code>struct xmss_verify_in</code>
Required Inputs	Parameter string, public key, original message, and signature.
Success Return	PIQ_SUCCESS
Verification Failure	PIQ_VERIFY_FAILED
Other Failures	Negative PiQASO error code.
Output Structure	None. Verification result is returned through the function return code.

Table 3.17: XMSS Memory Management Functions

Function	Description
<code>free_xmss_keygen_out()</code>	Releases memory allocated by <code>xmss_keygen()</code> , including public and private key buffers.
<code>free_xmss_sign_out()</code>	Releases memory allocated by <code>xmss_sign()</code> , including signature and updated private key buffers.

3.1.4 ML-KEM Key Encapsulation Functions

The PiQASO SDK provides a complete implementation of the **Module-Lattice-Based Key Encapsulation Mechanism (ML-KEM)** as standardized in FIPS 203. ML-KEM is a lattice-based public-key cryptographic primitive derived from the *CRYSTALS-Kyber* algorithm and is designed to provide quantum-resistant key establishment. Unlike traditional public-key encryption schemes, ML-KEM is a **Key Encapsulation Mechanism (KEM)** that enables two parties to establish a shared secret over an untrusted network. The shared secret generated during encapsulation and decapsulation can subsequently be used to derive symmetric encryption keys for protocols such as TLS 1.3 or AES-GCM.

Table 3.18: ML-KEM API Functions

Function	Description
<code>mlkem_keygen()</code>	Generates an ML-KEM public-private key pair.
<code>free_mlkem_keygen_out()</code>	Releases memory allocated during key generation.
<code>mlkem_encaps()</code>	Generates a ciphertext and shared secret using a public key.
<code>free_mlkem_encaps_out()</code>	Releases memory allocated during encapsulation.
<code>mlkem_decaps()</code>	Recovers the shared secret from a ciphertext using the corresponding private key.
<code>free_mlkem_decaps_out()</code>	Releases memory allocated during decapsulation.

Table 3.19: Supported ML-KEM Security Levels

Security Level	Description
MLKEM_LEVEL1	ML-KEM-512 (CRYSTALS-Kyber512). Provides NIST Security Level 1.
MLKEM_LEVEL3	ML-KEM-768 (CRYSTALS-Kyber768). Provides NIST Security Level 3.
MLKEM_LEVEL5	ML-KEM-1024 (CRYSTALS-Kyber1024). Provides NIST Security Level 5.

Table 3.20: ML-KEM Key Generation

Property	Description
Function	<code>mlkem_keygen()</code>
Purpose	Generates a quantum-resistant ML-KEM public-private key pair.
Input Structure	<code>struct mlkem_keygen_in</code>

Output Structure	struct mlkem_keygen_out
Input Parameter	Security level (MLKEM_LEVEL1, MLKEM_LEVEL3, or MLKEM_LEVEL5).
Output	Public encapsulation key and private decapsulation key.
Success Return	PIQ_SUCCESS
Failure Return	Negative PiQASO error code.

Table 3.21: ML-KEM Encapsulation

Property	Description
Function	mlkem_encaps()
Purpose	Generates a ciphertext and a cryptographically random shared secret using a recipient's public key.
Input Structure	struct mlkem_encaps_in
Output Structure	struct mlkem_encaps_out
Input Parameters	Security level and recipient public key.
Output	Ciphertext and 32-byte shared secret.
Ciphertext Usage	The generated ciphertext must be transmitted to the holder of the corresponding private key.
Shared Secret Usage	Typically used as input to a key derivation function (KDF) or directly as a symmetric session key.
Success Return	PIQ_SUCCESS
Failure Return	Negative PiQASO error code.

Table 3.22: ML-KEM Decapsulation

Property	Description
Function	mlkem_decaps()
Purpose	Recovers the shared secret from an ML-KEM ciphertext using the corresponding private key.
Input Structure	struct mlkem_decaps_in
Output Structure	struct mlkem_decaps_out
Input Parameters	Security level, private key, and ciphertext.
Output	Recovered 32-byte shared secret.
Shared Secret Property	If decapsulation succeeds, the recovered secret is identical to the secret generated during encapsulation.
Success Return	PIQ_SUCCESS
Failure Return	Negative PiQASO error code.

Table 3.23: ML-KEM Memory Management Functions

Function	Description
free_mlkem_keygen_out()	Releases memory allocated by mlkem_keygen(), including public and private key buffers.
free_mlkem_encaps_out()	Releases memory allocated by mlkem_encaps(), including ciphertext and shared secret buffers.
free_mlkem_decaps_out()	Releases memory allocated by mlkem_decaps(), including the recovered shared secret buffer.

3.1.5 ML-DSA Digital Signature Functions

The PiQASO SDK provides a complete implementation of the **Module-Lattice-Based Digital Signature Algorithm (ML-DSA)** as standardized in FIPS 204. ML-DSA is a lattice-based digital signature scheme derived from the *CRYSTALS-Dilithium* algorithm and is designed to provide authentication, integrity protection, and non-repudiation in the presence of both classical and quantum adversaries. Unlike stateful

hash-based signature schemes such as XMSS and LMS, ML-DSA is **stateless** and does not require private key updates after each signature operation. This property simplifies deployment while maintaining strong post-quantum security guarantees suitable for software signing, certificate infrastructures, secure communications, and enterprise authentication systems.

Table 3.24: ML-DSA API Functions

Function	Description
<code>mldsa_keygen()</code>	Generates an ML-DSA public-private key pair.
<code>free_mldsa_keygen_out()</code>	Releases memory allocated during key generation.
<code>mldsa_sign()</code>	Generates a digital signature using an ML-DSA private key.
<code>free_mldsa_sign_out()</code>	Releases memory allocated during signature generation.
<code>mldsa_verify()</code>	Verifies an ML-DSA signature using the corresponding public key.

Table 3.25: Supported ML-DSA Security Levels

Security Level	Description
<code>MLDSA_LEVEL2</code>	ML-DSA-44 (CRYSTALS-Dilithium2). Provides NIST Security Level 2.
<code>MLDSA_LEVEL3</code>	ML-DSA-65 (CRYSTALS-Dilithium3). Provides NIST Security Level 3.
<code>MLDSA_LEVEL5</code>	ML-DSA-87 (CRYSTALS-Dilithium5). Provides NIST Security Level 5.

Table 3.26: ML-DSA Key Generation

Property	Description
Function	<code>mldsa_keygen()</code>
Purpose	Generates a quantum-resistant ML-DSA public-private key pair.
Input Structure	<code>struct mldsa_keygen_in</code>
Output Structure	<code>struct mldsa_keygen_out</code>
Input Parameter	Security level (<code>MLDSA_LEVEL2</code> , <code>MLDSA_LEVEL3</code> , or <code>MLDSA_LEVEL5</code>).
Output	Public verification key and private signing key.
Success Return	<code>PIQ_SUCCESS</code>
Failure Return	Negative PiQASO error code.

Table 3.27: ML-DSA Signing

Property	Description
Function	<code>mldsa_sign()</code>
Purpose	Generates a detached digital signature for a message using an ML-DSA private key.
Input Structure	<code>struct mldsa_sign_in</code>
Output Structure	<code>struct mldsa_sign_out</code>
Input Parameters	Security level, private key, and message.
Output	Detached ML-DSA signature.
Key Management	ML-DSA is stateless; the private key does not change after signing and may be reused safely for subsequent signature operations.
Success Return	<code>PIQ_SUCCESS</code>
Failure Return	Negative PiQASO error code.

Table 3.28: ML-DSA Verification

Property	Description
Function	<code>mldsa_verify()</code>
Purpose	Verifies the authenticity and integrity of an ML-DSA signature.
Input Structure	<code>struct mldsa_verify_in</code>
Required Inputs	Security level, public key, original message, and signature.

Success Return	PIQ_SUCCESS
Verification Failure	PIQ_VERIFY_FAILED
Other Failures	Negative PiQASO error code.
Output Structure	None. Verification result is returned through the function return code.

Table 3.29: ML-DSA Memory Management Functions

Function	Description
<code>free_mldsa_keygen_out()</code>	Releases memory allocated by <code>mldsa_keygen()</code> , including public and private key buffers.
<code>free_mldsa_sign_out()</code>	Releases memory allocated by <code>mldsa_sign()</code> , including the generated signature buffer.

3.1.6 Post-Quantum TLS 1.3 Functions

The PiQASO SDK provides a transport-agnostic wrapper around TLS 1.3 that integrates post-quantum key establishment using **ML-KEM-512**. The implementation is designed to secure communications across cloud, edge, embedded, and trusted execution environments while maintaining compatibility with standard TLS 1.3 application workflows. The current implementation is configured for **NIST Security Level 1** and utilizes ML-KEM-512 for key exchange, TLS_AES_128_GCM_SHA256 as the cipher suite, SHA-256 for hashing, and ECDSA P-256 certificates for endpoint authentication. The library supports both socket-based communication and user-defined transport callbacks, enabling deployment in conventional operating systems as well as environments such as *OP-TEE* where direct network access may not be available.

Table 3.30: TLS 1.3 Configuration

Component	Description
TLS Version	TLS 1.3 only (no fallback to earlier TLS versions).
Key Exchange	ML-KEM-512 (CRYSTALS-Kyber512).
NIST Security Level	Level 1.
Cipher Suite	TLS_AES_128_GCM_SHA256.
Hash Function	SHA-256.
Peer Authentication	ECDSA using the NIST P-256 elliptic curve.
Certificate Format	DER encoded X.509 certificates.
Private Key Format	DER encoded key material.
Supported KEM Group	PIQ_TLS_KEM_ML_KEM_512.

Table 3.31: TLS API Functions

Function	Description
<code>piq_tls_server_ctx_create()</code>	Creates a reusable TLS server context.
<code>piq_tls_client_ctx_create()</code>	Creates a reusable TLS client context.
<code>piq_tls_ctx_free()</code>	Releases a TLS context.
<code>piq_tls_conn_create()</code>	Creates a new TLS connection instance from a context.
<code>piq_tls_conn_set_io()</code>	Registers custom transport callbacks.
<code>piq_tls_conn_set_fd()</code>	Associates a socket file descriptor with a connection (host environments only).
<code>piq_tls_handshake()</code>	Performs the TLS 1.3 handshake.
<code>piq_tls_read()</code>	Receives application data over an encrypted TLS session.
<code>piq_tls_write()</code>	Transmits application data over an encrypted TLS session.
<code>piq_tls_shutdown()</code>	Performs a TLS close-notify and terminates the session.
<code>piq_tls_conn_free()</code>	Releases connection resources.

<code>piq_tls_generate_self_signed()</code>	Generates a self-signed ECDSA P-256 certificate and private key.
<code>free_piq_tls_self_signed_out()</code>	Releases memory allocated during certificate generation.

Table 3.32: TLS Context Creation

Property	Description
Server Context Function	<code>piq_tls_server_ctx_create()</code>
Client Context Function	<code>piq_tls_client_ctx_create()</code>
Purpose	Creates reusable TLS configuration objects that can be shared across multiple TLS sessions.
Server Inputs	Certificate, private key, optional CA certificate, optional client certificate requirement, and KEM group.
Client Inputs	Optional CA certificate, optional client certificate, optional private key, hostname, and KEM group.
Mutual TLS Support	Optional through client certificates and trusted CA configuration.
Output	Initialized TLS context object.
Success Return	<code>PIQ_SUCCESS</code> .
Failure Return	Negative PiQASO error code.

Table 3.33: TLS Connection Lifecycle

Function	Description
<code>piq_tls_conn_create()</code>	Allocates a per-session TLS connection object.
<code>piq_tls_conn_set_io()</code>	Installs application-defined receive and send callbacks.
<code>piq_tls_conn_set_fd()</code>	Associates a socket descriptor with the TLS connection.
<code>piq_tls_handshake()</code>	Establishes cryptographic session keys and authenticates peers.
<code>piq_tls_shutdown()</code>	Sends a TLS close-notify alert and closes the session gracefully.
<code>piq_tls_conn_free()</code>	Releases all connection-related resources.

Table 3.34: TLS Application Data Functions

Function	Description
<code>piq_tls_read()</code>	Reads decrypted application data from an established TLS session.
<code>piq_tls_write()</code>	Encrypts and transmits application data over an established TLS session.
Successful Return	Number of bytes transferred.
Connection Closed	Returns 0 when the peer cleanly closes the connection.
Error Handling	Returns a negative PiQASO error code on failure.
Partial Transfers	Supported; callers should continue reading or writing until all data is processed.

Table 3.35: Custom Transport Callback Interface

Callback	Description
<code>piq_tls_recv_cb</code>	Receives raw TLS record data from the underlying transport.
<code>piq_tls_send_cb</code>	Transmits raw TLS record data to the underlying transport.
Positive Return Value	Number of bytes transferred.
Return Value 0	Peer has cleanly closed the connection.
<code>PIQ_TLS_IO_WANT_READ</code>	Non-blocking transport would block on read.
<code>PIQ_TLS_IO_WANT_WRITE</code>	Non-blocking transport would block on write.
<code>PIQ_TLS_IO_ERROR</code>	Fatal transport-level error.

Table 3.36: Self-Signed Certificate Generation

Property	Description
----------	-------------

Function	<code>piq_tls_generate_self_signed()</code>
Purpose	Generates a self-signed ECDSA P-256 certificate and private key.
Input Structure	<code>struct piq_tls_self_signed_in</code>
Output Structure	<code>struct piq_tls_self_signed_out</code>
Input Parameters	Common Name (CN) and certificate validity period in days.
Output	DER-encoded X.509 certificate and DER-encoded EC private key.
Intended Usage	Testing, development, and demonstration environments.
Production Recommendation	Use certificates issued by a trusted Certificate Authority.
Memory Cleanup	<code>free_piq_tls_self_signed_out()</code>

3.2 Unit Tests

To achieve a high level of security assurance and software quality, the PiQASO SDK has been **rigorously tested** throughout its development lifecycle. The testing strategy was designed to cover multiple layers of the system, ranging from user-facing APIs to internal cryptographic primitives and low-level system interactions. This multi-layered approach ensures both functional correctness and robustness against potential misuse or exploitation scenarios.

The SDK relies on the `wolfSSL` library as its underlying cryptographic engine. As a result, the implementation benefits from a well-established and widely audited cryptographic foundation. This integration ensures compliance with industry-grade cryptographic standards and reduces the risk of implementation-level vulnerabilities in core cryptographic operations such as encryption, decryption, and key management.

To validate correctness across all supported interfaces, the SDK was subjected to extensive automated testing, executing **more than 10,000 test iterations per core module**. These tests were designed to verify expected behavior under both normal and edge-case conditions. The test suite covered API usage scenarios, error handling paths, and boundary conditions to ensure deterministic and reliable behavior across all supported functionalities.

Memory management correctness was evaluated using **AddressSanitizer (ASan)**, enabling detection of memory leaks, buffer overflows, and use-after-free vulnerabilities during runtime execution. All major SDK components were exercised under ASan instrumentation, ensuring that *no memory leaks or undefined memory behaviors* were observed throughout the test cycles. This validation is particularly critical given the SDK's use in security-sensitive environments.

In addition to correctness and safety, the performance characteristics of the SDK were evaluated across all supported language interfaces. Benchmarking was performed to assess overhead introduced by language bindings and interoperability layers. The results confirmed **consistent and predictable performance behavior**, with no significant degradation observed across different language environments.

Beyond functional validation, the SDK was also subjected to system-level testing scenarios aimed at simulating misuse and potential exploitation attempts. These tests included malformed input handling, stress testing under high load, and robustness checks against unexpected execution flows. The goal was to ensure that the SDK maintains stability and secure behavior even under adversarial or unpredictable conditions.

The remainder of this section provides further details on the PQCaaS Gateway and Client, as well as Technical Implementation and Cryptographic Parity, and aspects pertaining to the integration of Mixed TLS Certificates and the NGINX Proxy Architecture.

3.3 Building Quantum-Safe Infrastructures: Desiderata of the PiQASO PQCaaS Solution

Having established the PiQASO PQC SDK as the foundational cryptographic engine for the project throughout this Chapter, the **PQC-as-a-Service (PQCaaS)** component constitutes the *second core building block* in the composition of a comprehensive quantum-safe infrastructure. While the SDK provides a highly optimized library for organizations and devices capable of directly embedding post-quantum primitives into their own software stacks, this model implicitly assumes a baseline of computational capability and engineering effort that is not universally available. The PQCaaS paradigm is designed to remove this assumption entirely, and in doing so, resolves two of the most persistent obstacles to the post-quantum transition. First, it allows **resource-constrained far-edge devices**, which may lack the processing power, memory, or energy budget required to execute post-quantum cryptographic operations natively, to *offload these operations to a dedicated server component*. Second, it eliminates the need for **legacy organizations** to undertake a costly and disruptive overhaul of their existing infrastructure in order to accommodate PQC. Rather than requiring such organizations to re-engineer their systems from the ground up, PQCaaS enables them to *seamlessly integrate post-quantum protections on top of what they already operate*.

Built on the **OpenSSL** framework, PQCaaS extends this capability to any far-edge device, allowing it to benefit from PQC protection with *zero modifications* required on the device side, while addressing issues present in cloud-based security offerings which typically introduce non-negligible latency overheads and raise concerns regarding **data sovereignty**, as sensitive key material and workloads are handed over to third-party infrastructure outside the direct control of the data owner. To address this, the PiQASO PQCaaS supports *multiple deployment modes* that allow data sovereignty to be preserved and governed entirely under the control of the consuming organization. As detailed in Deliverable D3.1 [46], this is reflected in the two supported modes of operation for the **Updatable Public Key Encryption (UPKE) scheme underpinning PQCaaS**: (i) in the first, the **PQCaaS component itself is trusted** and assumes responsibility for the generation and management of cryptographic keys on behalf of the parties involved, and (ii) in the second, **the PQCaaS component is not trusted with key material**, which instead remains under the exclusive management of the data owners themselves, thereby achieving full data sovereignty at the cost of a more decentralized operational model.

The **design philosophy** underlying PQCaaS offers several advantages of gateway-based approaches to quantum-safe infrastructure, wherein a centralized service acts as the primary point of cryptographic enforcement, which is the role played by the PQCaaS server within PiQASO. In line with this philosophy, PQCaaS is conceived to deliver *immediate protection* through its provision as a set of **containerized services**, allowing it to be deployed by any organization, regardless of its existing infrastructure, including legacy environments, in order to raise the overall security posture of that infrastructure with minimal integration effort. This containerized delivery model is further complemented by *centralized management* capabilities that can be configured to adhere to different trust models, giving adopting organizations the flexibility to select, according to their own risk appetite and regulatory context, where the management of cryptographic keys is ultimately performed. From a performance standpoint, PQCaaS is built around OpenSSL as its default cryptographic engine, but its architecture is deliberately *modular*, such that any other suitably compliant library, including the PiQASO PQC SDK itself, may be substituted as the foundational cryptographic engine without requiring changes to the surrounding service logic. Finally, because the PQCaaS is delivered as a containerized service, it can be deployed through commodity orchestration platforms such as **Kubernetes**, and allows consistent deployment across heterogeneous cloud and on-premises environments.

3.3.1 Architectural Paradigm of the PQCaaS Gateway and Client

The Post-Quantum Cryptography as a Service (**PQCaaS**) solution is designed to provide post-quantum cryptographic protection to existing networked applications while *minimizing the changes required to those applications*. Rather than requiring direct implementation and integration with post-quantum algorithms, the PQCaaS architecture introduces an **intermediate cryptographic layer** that transparently secures network traffic.

The architecture follows a **dual-node model** consisting of two primary components: the **PQC Client** and the **PQC Gateway (Server)**. These two components cooperate to establish and maintain a secure communication channel that protects data exchanged between applications using post-quantum cryptographic mechanisms.

From an architectural perspective, the PQCaaS solution acts as a *cryptographic proxy* positioned between communicating endpoints. Instead of applications directly managing post-quantum key establishment, encryption, and cryptographic protocol negotiation, these responsibilities are **delegated to the PQCaaS infrastructure**. This separation allows cryptographic functionality to evolve independently of the applications themselves, simplifying deployment and reducing the engineering effort required to adopt post-quantum security.

3.3.1.1 Transparent Integration with Legacy Applications

A primary objective of the PQCaaS architecture is to enable the adoption of post-quantum cryptography **without requiring modifications to existing software systems**. Many legacy applications were designed around classical cryptographic algorithms and standard networking libraries, making direct integration of post-quantum algorithms both complex and costly.

To address this challenge, the PQC Client operates at the *network boundary*, where it intercepts outgoing application traffic before it leaves the host system. Because the application continues to communicate using its existing networking interface, **no changes to application logic or communication workflows are necessary**. The application remains unaware that its traffic is subsequently processed using post-quantum cryptographic mechanisms.

This transparent integration significantly reduces migration complexity by allowing organizations to introduce quantum-resistant communication while *preserving existing software deployments and operational procedures*.

3.3.1.2 PQC Client Functionality

The **PQC Client** serves as the entry point into the PQCaaS infrastructure. Its primary responsibility is to receive network traffic generated by local applications and prepare it for secure transmission through the post-quantum communication channel.

Upon intercepting outgoing traffic, the client **encapsulates** the original network payload within a secure tunnel protected using post-quantum cryptographic techniques. Encapsulation preserves the original application data while adding the necessary cryptographic protection required for secure transport between the client and the gateway.

Once encapsulated and protected, the resulting data is transmitted over the network to the PQC Gateway. Throughout this process, the client operates *independently* of the application generating the traffic, allowing existing software to continue functioning without modification.

3.3.1.3 PQC Gateway Functionality

The **PQC Gateway (Server)** functions as the corresponding endpoint of the secure communication tunnel established by the PQC Client. It receives the protected traffic transmitted by the client and performs the cryptographic processing necessary to recover the original application data.

After successfully processing the protected communication, the gateway **extracts the original payload** from the secure tunnel. The recovered data is then forwarded to its intended destination using the standard networking mechanisms expected by the destination system.

By performing these operations, the gateway acts as *the bridge* between the post-quantum protected communication channel and conventional network services, enabling interoperability between quantum-resistant communications and existing network infrastructures.

3.3.1.4 Proxy-Based Communication Model

The PQCaaS architecture adopts a **proxy-based communication model**, where the client and gateway collectively mediate communication between applications and their intended destinations.

Instead of applications establishing secure communication channels directly, *all traffic passes through the PQCaaS components*. This intermediary layer is responsible for applying and removing the post-quantum cryptographic protection while preserving the original application communication semantics.

Because the proxy layer operates independently of application functionality, existing communication protocols can continue to operate without modification. The applications interact with the proxy using conventional networking interfaces, while the proxy manages the additional cryptographic operations required for post-quantum protection.

This architectural separation provides a clear distinction between application functionality and cryptographic services, allowing *each to evolve independently*.

3.3.1.5 Enabling Quantum-Resistant Communication

The overall objective of the PQCaaS architecture is to provide **quantum-resistant protection** for network communications while maintaining compatibility with existing systems.

By intercepting classical network traffic, encapsulating it within a post-quantum protected communication channel, transmitting it securely between the PQC Client and PQC Gateway, and finally restoring the original traffic before forwarding it to its destination, the architecture introduces post-quantum cryptographic protection **without altering application behavior or communication logic**.

As a result, existing networked applications continue to operate using their standard protocols and interfaces, while the communication path between the client and gateway benefits from the additional security provided by post-quantum cryptographic mechanisms. This enables organizations to *incrementally adopt* post-quantum security while preserving compatibility with current software and network infrastructures.

3.3.2 Technical Implementation and Cryptographic Parity

The PQCaaS solution is built upon the **same algorithmic foundation** as the PiQASO PQC SDK. While they utilize different underlying C libraries to suit their distinct deployment environments, the PQCaaS architecture enforces **strict cryptographic parity** with the SDK, ensuring consistency in the standardized algorithms and security mechanisms employed by both solutions.

This shared algorithmic foundation establishes *cryptographic parity* between the standalone SDK and the PQCaaS deployment model. Regardless of whether applications directly integrate the SDK or communicate through the PQCaaS proxy infrastructure, the underlying post-quantum cryptographic operations adhere to the *exact same standards*. Consequently, both deployment approaches provide **equivalent cryptographic capabilities** while differing only in how those capabilities are delivered to applications.

3.3.2.1 Shared Cryptographic Capabilities

The PQCaaS solution exposes the same collection of **NIST-standardized** post-quantum cryptographic algorithms available within the PiQASO PQC SDK. These algorithms are responsible for performing the fundamental cryptographic operations required to establish secure communications and authenticate communicating entities.

Specifically, the implementation includes **ML-KEM** (Module-Lattice Key Encapsulation Mechanism) for post-quantum key establishment and **ML-DSA** (Module-Lattice Digital Signature Algorithm) for digital signature operations. By supporting the exact same standardized algorithms as the SDK, the PQCaaS solution maintains **absolute cryptographic compatibility** across the PiQASO platform.

This standardization ensures that the security properties, supported algorithms, and cryptographic behavior remain consistent regardless of whether post-quantum protection is accessed through direct library integration or through the proxy-based service.

3.3.2.2 Cryptographic Parity Between Deployment Models

Although the PiQASO PQC SDK and the PQCaaS solution differ in their deployment architecture and underlying C libraries, they maintain **complete cryptographic equivalence**. This means that the same standardized post-quantum cryptographic primitives are used in both environments *without modification* to their mathematical specifications.

The primary distinction between the two solutions lies in *how* applications access the cryptographic functionality. Applications integrating the SDK invoke cryptographic operations directly through library interfaces, whereas applications using PQCaaS obtain the same cryptographic protection indirectly through the client and gateway proxy components.

Maintaining this parity simplifies adoption by allowing organizations to select the deployment model that best matches their infrastructure requirements **without compromising or altering the underlying cryptographic defense mechanisms**.

3.3.2.3 Deployment Model of the PiQASO PQC SDK vs PQCaaS

The architectural strategy of the PiQASO ecosystem bifurcates into two distinct deployment paradigms to accommodate diverse integration requirements: the embedded **Software Development Kit (SDK)** and the standalone **Post-Quantum Cryptography as a Service (PQCaaS)** infrastructure. The standalone PiQASO PQC SDK is engineered for *direct, native integration* into software applications. In this model, developers embed the SDK directly within their codebase, invoking its advanced cryptographic functionalities through dedicated programming language interfaces. To bridge the gap across highly heterogeneous development environments, the SDK relies on the **Simplified Wrapper and Interface Generator (SWIG)** framework to produce robust language bindings. These bindings expose the underlying, highly optimized C-based cryptographic library to a wide spectrum of modern programming languages, explicitly including **Python, Java, and Node.js**. Consequently, cryptographic operations are executed natively within the

host application's own execution context. This embedded approach is highly suitable for newly developed software or existing applications that possess the flexibility to be actively modified and recompiled to natively incorporate post-quantum cryptographic primitives.

Conversely, the PQCaaS infrastructure introduces a fundamentally **decoupled deployment model**, operating as an independent network service rather than an embedded application library. Functioning as a standalone network daemon, the PQCaaS proxy is solely responsible for intercepting, cryptographically processing, and securely forwarding protected network traffic. Because the quantum-resistant functionality is offloaded entirely to this external proxy infrastructure, host applications are **completely absolved** from the operational burden of integrating complex cryptographic libraries or directly invoking post-quantum algorithms. Instead, these applications continue to transmit data using standard, conventional networking mechanisms, while the PQCaaS layer *transparently* handles all post-quantum encapsulation and decapsulation on their behalf. This strict architectural separation ensures that post-quantum cryptographic defenses can be deployed **rapidly and entirely independent of the application logic**. Ultimately, this enables legacy systems and immutable enterprise infrastructures to achieve robust, quantum-resistant communication channels *without necessitating the direct source code integration and software modification* demanded by the SDK deployment model.

3.3.2.4 High-Throughput Network Operation

As a network-based service, the PQCaaS implementation is designed to **efficiently process large volumes of network traffic**. Unlike the standalone SDK, which performs cryptographic operations only when invoked by an application, the PQCaaS daemon continuously manages communication between multiple clients and destinations.

To support numerous simultaneous communication sessions, the service utilizes **asynchronous, event-driven networking techniques**. This execution model allows the daemon to manage many concurrent proxy tunnels *without dedicating a separate execution thread to each network connection*. Instead, network events are processed as they occur, enabling efficient utilization of system resources while maintaining responsiveness under increasing connection loads.

The use of asynchronous network processing is particularly well suited to proxy services, where communication involves handling multiple independent network streams concurrently.

3.3.2.5 Integration with the Underlying Cryptographic Library

While the PQCaaS solution introduces an independent network layer, the cryptographic operations themselves rely on a distinct foundational framework compared to the embedded SDK. Contrary to the standalone PiQASO PQC SDK, which utilizes **wolfSSL**, the PQCaaS infrastructure is explicitly developed using the **OpenSSL C** implementation. The PQCaaS proxy invokes these robust OpenSSL routines during the secure protection and recovery of network traffic exchanged between the client and the gateway. Specifically, this OpenSSL-based cryptographic library performs the intensive post-quantum operations required as classical traffic is encapsulated before transmission across the quantum-safe tunnel, and subsequently decapsulated after reception. By building the network service around the highly mature OpenSSL ecosystem, the PQCaaS architecture leverages **industry-standard asynchronous network handling** and extensive protocol support. Although it employs a different underlying C library than the wolfSSL-based SDK, it strictly maintains **cryptographic consistency across the ecosystem** executing the exact same standardized post-quantum algorithms and ensuring that the network-based deployment model delivers the same robust quantum resistance as the embedded components.

3.3.2.6 Efficient Traffic Encapsulation and Decapsulation

An essential responsibility of the PQCaaS infrastructure is the secure processing of network traffic exchanged between the client and gateway. Before outgoing application data is transmitted, it is **encapsulated** within the protected communication channel established by the PQCaaS components. Upon arrival at the gateway, the protected communication is processed to recover the original application payload before forwarding it to its intended destination.

These encapsulation and decapsulation operations are integrated with the underlying post-quantum cryptographic processing provided by the OpenSSL-based library. The network implementation is designed to perform these operations *efficiently* so that the additional cryptographic protection introduces **minimal processing overhead** while preserving the transparency of the proxy-based communication model.

3.3.3 Integration of Mixed TLS Certificates and NGINX Proxy Architecture

Building upon the OpenSSL-backed network layer established in the PQCaaS deployment model, the solution provides comprehensive support for **mixed TLS certificates**, as detailed in Chapter 8 of the architectural specification. This capability enables the proxy infrastructure to operate seamlessly in environments where both classical and post-quantum cryptographic mechanisms coexist, allowing organizations to introduce quantum-resistant security **without necessitating an immediate, disruptive replacement** of their existing Public Key Infrastructure (PKI). The integration of mixed TLS certificates directly addresses one of the primary operational challenges of the quantum migration: maintaining strict interoperability with legacy systems that continue to rely on conventional cryptographic algorithms while simultaneously introducing support for standardized post-quantum cryptography.

Through its **hybrid certificate support**, the PQCaaS infrastructure can actively participate in TLS communications that involve both classical cryptographic algorithms, such as RSA or Elliptic Curve Cryptography (ECC), and post-quantum algorithms, such as ML-DSA. Rather than restricting communication to a single algorithmic family, the proxy negotiates secure TLS connections utilizing certificate chains that incorporate both components. During the TLS handshake, the proxy utilizes its configured certificate infrastructure to authenticate endpoints and establish secure communication channels. This enables the TLS negotiation process to natively accommodate infrastructures that are transitioning toward post-quantum cryptography, allowing the proxy to serve as a *transparent intermediary* that integrates quantum-resistant mechanisms without fundamentally altering the communication model utilized by the client applications.

A significant objective of this hybrid approach is to **preserve backward compatibility** with existing enterprise deployments. Because many production environments will depend on classical public key algorithms for the foreseeable future, an immediate, wholesale migration to exclusively post-quantum certificates is highly impractical. By supporting dual configurations, the PQCaaS solution *minimizes disruption* to deployed services, empowering organizations to execute incremental migration strategies rather than simultaneous, system-wide upgrades. Furthermore, this deliberate inclusion of post-quantum authentication mechanisms critically strengthens the long-term security of encrypted communications against future threats posed by large-scale quantum computers. Specifically, it mitigates the pervasive “**harvest now, decrypt later**” threat model, ensuring that communications secured today remain impenetrable even as cryptographic standards and quantum capabilities evolve.

Within the practical PQCaaS deployment, this support for mixed TLS certificates is deeply integrated with **NGINX**, which serves as the primary TLS termination and proxy component. NGINX is explicitly configured to utilize certificate chains that include both classical and post-quantum cryptographic material. In this orchestrated architecture, NGINX handles the high-level TLS connection establishment and certificate presentation, while the underlying PQCaaS infrastructure supplies the complex post-quantum

cryptographic execution. This architectural separation allows the proxy service to leverage NGINX's mature, highly optimized TLS infrastructure while seamlessly extending it with the post-quantum algorithmic capabilities standardized across the PiQASO ecosystem, ultimately facilitating *flawless interoperability* across both legacy and next-generation environments.

3.4 PiQASO SDK Performance Benchmarks and Architectural Critique on the ZCU 104 Platform

To rigorously validate the computational viability of the PiQASO SDK across the computing continuum, comprehensive performance profiling was conducted on a **Xilinx ZCU 104** evaluation board. This ARM-based platform serves as an ideal representation of a robust edge-computing gateway, demanding highly efficient cryptographic execution. The performance metrics, quantified in milliseconds, capture the mean execution time and standard deviation across four distinct execution environments: the native C/C++ implementation, and the SWIG-generated language bindings for Java, Node.js, and Python. Integrating these empirical findings into the broader PiQASO architectural framework provides critical insights into *algorithmic scaling*, *language-wrapper overhead*, and the practical deployment realities of post-quantum cryptography at the edge.

Table 3.37: Execution time (ms) for ML-KEM operations. Values are reported as mean $\pm$ standard deviation.

Operation	Native	Python
ML-KEM-512 KeyGen (NIST Level 1)	2.152 $\pm$ 0.022	2.157 $\pm$ 0.022
ML-KEM-512 Encaps (NIST Level 1)	2.092 $\pm$ 0.017	2.091 $\pm$ 0.018
ML-KEM-512 Decaps (NIST Level 1)	1.438 $\pm$ 0.013	1.437 $\pm$ 0.013
ML-KEM-768 KeyGen (NIST Level 3)	2.631 $\pm$ 0.015	2.632 $\pm$ 0.016
ML-KEM-768 Encaps (NIST Level 3)	2.707 $\pm$ 0.017	2.707 $\pm$ 0.016
ML-KEM-768 Decaps (NIST Level 3)	2.217 $\pm$ 0.019	2.216 $\pm$ 0.018
ML-KEM-1024 KeyGen (NIST Level 5)	3.446 $\pm$ 0.020	3.447 $\pm$ 0.019
ML-KEM-1024 Encaps (NIST Level 5)	3.546 $\pm$ 0.019	3.546 $\pm$ 0.018
ML-KEM-1024 Decaps (NIST Level 5)	3.206 $\pm$ 0.018	3.207 $\pm$ 0.018

Table 3.38: Execution time (ms) for ML-DSA operations. Values are reported as mean $\pm$ standard deviation.

Operation	Native	Python
ML-DSA-44 KeyGen (NIST Level 2)	3.980 $\pm$ 0.152	3.977 $\pm$ 0.150
ML-DSA-44 Sign (NIST Level 2)	11.788 $\pm$ 6.754	12.065 $\pm$ 6.693
ML-DSA-44 Verify (NIST Level 2)	3.034 $\pm$ 0.018	3.033 $\pm$ 0.017

ML-DSA-65 KeyGen (NIST Level 3)	5.691 ± 0.018	5.688 ± 0.020
ML-DSA-65 Sign (NIST Level 3)	18.251 ± 10.284	18.270 ± 11.554
ML-DSA-65 Verify (NIST Level 3)	4.876 ± 0.016	4.875 ± 0.018
ML-DSA-87 KeyGen (NIST Level 5)	8.743 ± 0.020	8.737 ± 0.020
ML-DSA-87 Sign (NIST Level 5)	21.774 ± 11.070	22.554 ± 11.369
ML-DSA-87 Verify (NIST Level 5)	8.006 ± 0.018	8.002 ± 0.016

Table 3.39: Execution time (ms) for AES encryption and decryption operations. Values are reported as mean ± standard deviation.

Operation	Native	Python
AES-GCM-128 Encrypt	1.174 ± 0.027	1.176 ± 0.024
AES-GCM-128 Decrypt	0.079 ± 0.006	0.079 ± 0.006
AES-GCM-192 Encrypt	1.102 ± 0.025	1.104 ± 0.025
AES-GCM-192 Decrypt	0.078 ± 0.005	0.079 ± 0.010
AES-GCM-256 Encrypt	1.101 ± 0.022	1.104 ± 0.023
AES-GCM-256 Decrypt	0.081 ± 0.005	0.081 ± 0.006
AES-CBC-128 Encrypt	1.043 ± 0.020	1.045 ± 0.021
AES-CBC-128 Decrypt	0.045 ± 0.004	0.045 ± 0.005
AES-CBC-256 Encrypt	1.046 ± 0.023	1.048 ± 0.022
AES-CBC-256 Decrypt	0.038 ± 0.004	0.038 ± 0.004

Table 3.40: Execution time (ms) for stateful hash-based signature operations. Values are reported as mean ± standard deviation.

Operation	Native	Python
LMS (L1, H=5, W=2) KeyGen	380.617 ± 0.083	380.571 ± 0.069
LMS (L1, H=5, W=2) Sign	386.107 ± 0.274	386.043 ± 0.270
LMS (L1, H=5, W=2) Verify	5.520 ± 0.259	5.559 ± 0.259
XMSS-SHA2.10.256 KeyGen	93960.635 ± 251.351	93797.991 ± 40.046
XMSS-SHA2.10.256 Sign	135.719 ± 3.277	135.424 ± 3.214
XMSS-SHA2.10.256 Verify	50.834 ± 3.211	50.813 ± 3.144

An analysis of the benchmarking dataset reveals several pronounced architectural behaviors central to the design of the PiQASO SDK. Foremost is the **near-perfect parity** observed between the Native C/C++ execution times and the Python environment. Across all cryptographic primitives—from the lightweight AES-GCM operations to the heavily compute-bound XMSS key generation—the Python wrapper demonstrates *practically zero overhead*. For instance, ML-KEM Level 1 Key generation requires 2.152 milliseconds natively and 2.157 milliseconds in Python. This consistency validates the strategic choice of utilizing the **SWIG** framework; it successfully bypasses the performance penalties traditionally associated with dynamic language runtimes by maintaining execution entirely within the pre-compiled C shared object, utilizing the Python environment exclusively as a thin orchestrator.

Beyond the Native and Python cross-environment dynamics, the results emphasize the distinct performance profiles between *lattice-based asymmetric cryptography* and traditional symmetric operations. While the wolfSSL-backed AES algorithms process blocks in fractions of a millisecond, the post-quantum ML-KEM and ML-DSA implementations exhibit expected, yet manageable, multi-millisecond latencies.

Furthermore, the data vividly illustrates the operational constraints of stateful hash-based signatures. The XMSS key generation operation is **computationally exorbitant**, requiring significant processor time (nearly 94 seconds natively) to compute the foundational Merkle tree structure. This metric reinforces the architectural doctrine established earlier in the PiQASO documentation: algorithms like LMS and XMSS must be strictly reserved for *infrequent, high-value operations* such as secure boot verifications and immutable firmware signing, leaving the significantly faster lattice-based ML-DSA to handle dynamic, day-to-day transactional authentication.

3.5 Algorithmic Optimizations of Lattice-Based Algorithms

Throughout this chapter, PiQASO has provided extensive **benchmarking results** characterizing the performance of the SDK's supported primitives across various security parameter sets, for native C/C++ and Python-based environments. Building on these results, in order to achieve performance improvements, PiQASO has conducted an exercise for the identification of the most computationally intensive mathematical operations acting as bottlenecks in various families of post-quantum cryptographic schemes, the results of which are provided in Table 3.41, which lists the identified characteristic of low-level arithmetic employed in PQC schemes. Specifically, three operands were identified as the common base capturing the specificities of the multi-precision arithmetic for all examined families: (i) **integers of a length of several 100 bits** (for curve-based pre-quantum and isogeny-based post-quantum cryptography), (ii) **polynomials of a degree of between 256- and 1024-bit** with coefficients in $(\mathbb{Z}_q)$ where q is a “medium-size” prime below 32 bits (for lattice-based cryptosystems like ML-KEM, ML-DSA, FN-DSA), and (iii) **binary polynomials of a very high degree** of above 10000 (for code-based cryptosystems like HQC). Based on the benchmarking results, as well as the output of this investigation, PiQASO will design and implement targeted **optimizations** aimed at increasing the efficiency and performance of the operations detailed above, to be continued and concluded in the *second reporting period*. Specifically, PiQASO will pursue two complementary types of optimization.

The first is to convert the SDK to its NEON-optimized equivalent, i.e., an **optimized software implementation** through the manual integration of **SIMD instructions**, targeting *polynomial multiplication*, which is the fundamental bottleneck common to all major lattice- and code-based cryptographic operations. To accelerate HQC, PiQASO innovates by **enhancing the Karatsuba algorithm for binary polynomial multiplication**, by converting to a recursive to a static schedule, which translates to flattening the recursive nature of Karatsuba. This allowed PiQASO to upper bound the acceleration of the polynomial multiplication (Chapter 7)¹. As part of the investigations carried out by PiQASO in the context of **HQC acceleration**, the PQClean reference implementation of HQC was manually optimized by introducing SIMD instructions and the carry-less PMULL instruction to accelerate its binary polynomial multiplication routine, which dominates HQC's runtime across key generation, encapsulation, and decapsulation. This vectorized, NEON/PMULL-based approach yields substantial speedups beyond compiler-level (-O3) optimization alone, and a complementary hardware accelerator for HQC's polynomial multiplication is also being developed. The purpose of this parallel effort is to compare the speedups achieved through non-compiler and compiler-based software acceleration against those achieved through dedicated hardware acceleration. Applying this same SIMD-based methodology to the SDK's lattice-based primitives (ML-

¹At the moment of writing, the NEON-optimized implementation of HQC with flattened Karatsuba has been fully benchmarked, but results will be placed in D3.3 [48] since this work is under submission for publication.

KEM and ML-DSA) is planned for the second reporting period.

The second is an **algorithmic optimization** that targets the fact that all lattice-based schemes considered here rely on hardness assumptions rooted in the **Learning With Errors (LWE) problem**, whose main computational bottleneck is the **Number Theoretic Transform (NTT)** mathematical operation. To address this bottleneck, PiQASO proposes an optimization that shifts the arithmetic domain. Specifically, rather than performing polynomial multiplication via the NTT, PiQASO employs an **FFT-based convolution technique**. This approach was outlined as an initial draft in D3.1, and will be further analyzed and benchmarked in D3.3. **The purpose is to evaluate the benefits and speedups provided by compiler and non-compiler accelerations, compared to hardware-based accelerations.**

Table 3.41: Algorithmic and Coefficient-Level Arithmetic Characteristics of PQC Schemes

Family	Crypto Scheme	Algorithmic Operations	Coefficient-Level Operations
Lattice-Based	ML-KEM	- Polynomials with degree 256 - Defined in $\mathbb{Z}_q[X]/(X^n + 1)$ - NTT/INTT (one polynomial at a time) - PWM (two polynomials at a time) - Add/Sub (two polynomials at a time)	12-bit carry-propagating multiplication modulo 3329 12-bit carry-propagating additions/subtractions modulo 3329
Lattice-Based	ML-DSA	- Polynomials with degree 256 - Defined in $\mathbb{Z}_q[X]/(X^n + 1)$ - NTT/INTT (one polynomial at a time) - PWM (two polynomials at a time) - Add/Sub (two polynomials at a time)	23-bit carry-propagating multiplication modulo 8380417 23-bit carry-propagating additions/subtractions modulo 8380417
Code-Based	HQC	- Polynomials with degree: 17,669 / 35,851 / 57,637 - Defined in $\mathbb{F}_2[X]/(X^n - 1)$ - Karatsuba multiplication between two polynomials - Alternative implementation via Additive FFT (NTT-like)	- Carry-less multiplications of degree 17,669, 35,851, or 57,637 - With Karatsuba or other methods (Toom-Cook), large multiplications can be broken into smaller multiplications; the size can be chosen - The same applies if AFFT is adopted
Isogeny-Based	SCALLOP-HD, PEARL-SCALLOP	- Extension field of degree 2 arithmetic: multiplication, squaring, addition/subtraction - Elliptic curve arithmetic: point doubling/addition/scalar multiplication - Isogeny computation and evaluation: - Isogenies of dimension 1 (elliptic curves) - Isogenies of dimension 2 (higher-dimensional abelian varieties) – for SCALLOP-HD - Setup phase: orientation computation, starting curve construction	- Prime p size: 1050, 1137, 1296, 1501, 1654, 1909 (SCALLOP-HD); 865, 1161, 1410 (PEARL-SCALLOP) - Multiplication, squaring, addition/subtraction modulo p

Family	Crypto Scheme	Algorithmic Operations	Coefficient-Level Operations
Isogeny-Based	Clapoti, KLaPoTi, Qlapoti	- Extension field of degree 2 arithmetic - Elliptic curve arithmetic: point doubling/addition/scalar multiplication - Pairings – for KLaPoTi: pairings on supersingular elliptic curves - Isogeny computation and evaluation: - Isogenies of dimension 1 (elliptic curves) – for KLaPoTi - Isogenies of dimension 2 (higher-dimensional abelian varieties) – for KLaPoTi and Qlapoti - Isogenies of dimensions 4, 8 (higher-dimensional abelian varieties) – for Clapoti - Ideal to isogeny translation	- Prime p size: 250, 425, 896 (KLaPoTi); 251, 383, 505 (Qlapoti) - Multiplication, squaring, addition/subtraction modulo p
Isogeny-Based	PEGASIS, Qt-PEGASIS	- Extension field of degree 2 arithmetic – for Qt-PEGASIS - Elliptic curve arithmetic operations: point doubling/addition/scalar multiplication - Pairings – for Qt-PEGASIS: pairings on supersingular elliptic curves - Isogeny computation and evaluation: - Isogenies of dimension 1 (elliptic curves) - Isogenies of dimension 4 (higher-dimensional abelian varieties) - Ideal to isogeny translation	- Prime p size: 508, 1008, 1554, 2031, 4089 - Multiplication, squaring, addition/subtraction modulo p
Isogeny-Based	CSIDH (Key Exchange)	- Elliptic curve arithmetic operations: point doubling/addition/scalar multiplication - Isogeny computation and evaluation: 1-dimensional abelian varieties (elliptic curves) - Public key validation	- Prime p size: 511, 1020, 1792 - Prime p size: 4096, 5120, 6144, 8192, 9216 - Multiplication, squaring, addition/subtraction modulo p
Isogeny-Based	CSI-FiSh (Digital Signature)	- Elliptic curve arithmetic operations: point doubling/addition/scalar multiplication - Isogeny computation and evaluation: 1-dimensional abelian varieties (elliptic curves)	- Prime p size: 511 - Multiplication, squaring, addition/subtraction modulo p
Isogeny-Based	SQLSign (Digital Signature)	- Extension field of degree 2 arithmetic - Elliptic curve arithmetic operations: point doubling/addition/scalar multiplication - Pairings - Isogeny computation and evaluation: - Isogenies of dimension 1 (elliptic curves) - Isogenies of dimension 2 (higher-dimensional abelian varieties) - Ideal to isogeny translation	- Prime p size: 251, 383, 505 - Multiplication, squaring, addition/subtraction modulo p
Traditional Crypto	ECC, X25519, and NIST P-256	- Scalar multiplication - Point addition - Point doubling	- Rely on 256-bit carry-propagating multiplications 256-bit carry-propagating additions/subtractions - Modular inversion over 256-bit

The remainder of this section is dedicated to this latter (algorithmic) optimization. The practical deployment of lattice-based Post-Quantum Cryptography (PQC) requires not only theoretical security guarantees, but also **highly optimized implementations capable of meeting stringent latency and**

resource constraints. As these primitives rely heavily on polynomial arithmetic over finite fields, the efficiency of the underlying multiplication algorithms is the primary driver of overall system performance. While standard techniques such as the NTT provide asymptotic efficiency, they often incur **significant concrete overhead due to the dense frequency of modular reduction operations required in software**. In this section, we propose an algorithmic optimization that mitigates this bottleneck by transposing the arithmetic burden from the modular domain to the complex domain. By leveraging fixed-point arithmetic, we achieve better performance while preserving the constant-time execution, which is necessary for security against timing side-channel attacks.

3.5.1 Context and Mathematical Foundations

Lattice-based PQC primitives rely on the hardness of problems over structured lattices to achieve an optimal balance between **security**, **computational performance**, and **transmission bandwidth** (ciphertext and key sizes). These primitives typically operate over polynomial rings, where elements are represented as polynomials—or matrices of polynomials—residing in quotient rings of the form:

$$\mathcal{R}_q = \mathbb{Z}_q[X]/(X^N + 1), \quad (3.1)$$

where N is a power of two (typically 256 or 512) defining the lattice dimension, and q is a prime modulus. The efficiency of polynomial operations within this ring structure is a critical determinant of the primitive's overall performance. The most computationally intensive operation is polynomial multiplication. Given two polynomials $a(x) = \sum_{i=0}^{N-1} a_i x^i$ and $b(x) = \sum_{i=0}^{N-1} b_i x^i$, their product $c(x) = a(x) \cdot b(x) \bmod (X^N + 1)$ is defined as a negacyclic convolution. The coefficients c_k of the resulting polynomial are given by:

$$c_k = \sum_{i=0}^k a_i b_{k-i} - \sum_{i=k+1}^{N-1} a_i b_{N+k-i} \quad \text{for } k = 0, \dots, N-1. \quad (3.2)$$

A naive implementation of this convolution requires $O(N^2)$ scalar operations, which is prohibitively slow for the large values of N required for high-security applications.

3.5.2 Standard Approach vs. Computational Bottlenecks

To mitigate the $O(N^2)$ complexity, parameters N and q are typically selected to support the **NTT** [52]. The NTT is an analogue of the Fast Fourier Transform (FFT) over finite fields, allowing multiplication to be performed in the frequency domain. By transforming the polynomials into their NTT representations, point-wise multiplication can be applied, reducing the asymptotic complexity to $O(N \log_2 N)$. However, while the NTT reduces the operation count, the standard implementation introduces significant overhead due to the nature of modular arithmetic. *Modular reductions* are required after every addition and multiplication to keep coefficients within the field $\mathbb{Z}_q$. These modular reduction operations hinder the performance.

3.5.3 Proposed Optimization: Complex Domain Mapping

To address these bottlenecks, we propose an algorithmic optimization that **shifts the arithmetic domain**. Instead of performing operations directly in $\mathbb{Z}_q$, we map the original polynomials to appropriate polynomials with complex coefficients. This allows us to leverage standard **FFT-based convolution** techniques rather than the modular NTT. The core of this strategy involves implementing *fixed-point arithmetic* to retain constant-time execution—a mandatory requirement for preventing timing side-channel attacks. In

this optimized flow, all intermediate operations (FFT, point-wise multiplication, inverse FFT) take place using standard integer instructions without intermediate modular reductions. The costly modular reduction modulo q is deferred until the very end of the polynomial multiplication process.

3.5.4 Precision and Correctness Strategy

The viability of this approach hinges on managing **numerical precision** to prevent data loss. Since fixed-point arithmetic is an approximation of the continuous complex domain, strict error analysis is required. We ensure correctness through a dual-selection strategy:

1. **Integer Width Selection:** We carefully select the appropriate container size (e.g., int32 vs int64) to accommodate the dynamic range of the values during the accumulation steps of the convolution, ensuring that the maximum possible value does not exceed the container's capacity (overflow protection).
2. **Fractional Bit Allocation:** We determine the optimal number of bits dedicated to the fractional part of the fixed-point representation. This minimizes the quantization noise introduced during the FFT/IFFT stages.

By rigorously configuring these parameters, we maintain sufficient precision to ensure that the accumulated error remains strictly below the rounding threshold. Consequently, the final fixed-point result rounds correctly to the exact integer expected from the theoretical convolution, guaranteeing **mathematical equivalence** to the standard NTT approach while reducing instruction overhead.

3.5.5 Generalization of LWE Problem for Increased Efficiency

As part of PiQASO, an additional optimization has already been carried out beyond the FFT-based convolution technique described above: the **generalization of the Learning With Errors (LWE) problem** and of its application, so as to enable new, more efficient cryptographic schemes that support all core key-management operations – including *key encapsulation* and *key distribution* – in a more efficient manner.

In this context, PiQASO has designed a new construction that **augments LWE with invertible secret keys**. This construction supports *secret-key-composable operations*, namely multiplication via polynomial matrices, and inherits the security properties of standard LWE-based keys while expressing them as polynomial matrices. This generalization enables more flexible and efficient key-management workflows, making secret keys composable under multiplication and providing a foundation for further efficiency gains in the lattice-based primitives supported by the PiQASO SDK.

Chapter 4

Forward Secure Cloud Storage Scheme

Cloud storage services offer high scalability but expose cryptographic keys to long-term risks. If a secret key is compromised, an attacker can access not only current data but also historically archived information. To mitigate this critical threat, **a cloud storage scheme must guarantee forward security**. By periodically evolving keys and erasing obsolete key material, forward security ensures that exposing a secret key at a given time does not compromise data protected in the past.

While prior works have already established variants of forward-secure schemes, real-world deployment necessitates forward-secure schemes that ensure practical implementations. In Deliverable 3.1 [46] we presented a solution that bridges this gap by realizing a basic, forward-secure scheme for cloud-based data management services, under a trusted infrastructure setting. However, this construction relies on the assumption that the underlying Common Transmission System (CTS) is trusted. To remove this strict dependency, **we extend the framework introduced in Deliverable 3.1 [46] by presenting an enhanced forward-secure scheme designed for an untrusted cloud storage server model**.

To describe the contributions of this chapter, we first recall the components of the CTS from Deliverable 3.1 [46, Section 5.2]. The CTS comprises three main entities, the **Identity Provider (IDP)**, the **Key Provider (KP)**, and the **PQ Data Server (DS)**. Each entity is assigned the following distinct responsibilities:

Identity Provider (IDP): The IDP validates public key certificates to verify end-user identities. It issues and securely transmits access tokens containing arbitrary data, expiration times, attributes, and permissions, which are subsequently used for user authentication and authorization across the scheme.

Key Provider (KP): In the policy-based mode, it is responsible for the generation of all required key material and the secure transfer of these keys to authorized entities (end-users, IDP, and PQ DS).

PQ Data Server (DS): The PQ DS acts as the central storage and access control entity that protects and stores the data payloads. Utilizing the PQC Ensemble SDK, it authenticates end-users via access token verification, using either challenge-response protocols (e.g., ISO/IEC 9798), PQ-TLS (ML-KEM/ML-DSA), or KEMTLS. Additionally, it ensures data confidentiality in-motion via post-quantum encryption and guarantees long-term data confidentiality at-rest using advanced updatable public key encryption (UPKE).

We assume that a user, identified by ID_U obtained from IDP in the registration phase, has already authenticated with the CTS. Consequently, following Deliverable 3.1 [46], we exclude the authentication phase and the interaction with the IDP from the description of our model. In Deliverable 3.1, we established the forward security of our simplified cloud storage scheme in PiQASO under the assumption that the

CTS is trusted, implying that the KP and DS are trusted [46]. Departing from our previous construction, we strengthen the security model by relaxing the trust assumptions on the CTS. In particular, **our new cloud-based data storage solution does not require the KP and DS to be trusted, resulting in a more realistic construction that better reflects cloud storage deployment scenarios while maintaining the envisioned strong security guarantees**. In the present deliverable, we describe the two approaches to secure cloud storage, distinguished by their trust assumptions regarding the CTS components (KP and DS). The first approach assumes that the KP and DS are trusted, while the second relaxes this assumption by considering a setting where the KP and DS are untrusted. Hence, the main contribution of this chapter is twofold:

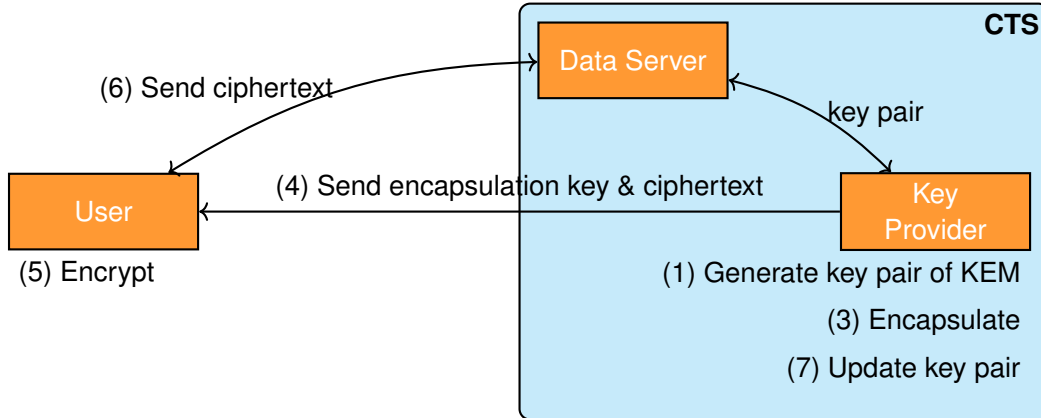
- **Implementation of the Cloud Storage Scheme with Trusted Key Provider.** In Deliverable 3.1, the KP and the DS are assumed to be trusted so that the KP generates KEM key pairs for users and the DS stores the key pairs in the database DB_K along with the ciphertexts. In our implementation, we assume the KP is trusted and assigned with the task of generating and updating keys, as well as with the task of executing encapsulation/decapsulation operations. This implementation focuses on the aspect of key management. Hence, a dedicated data server is not included. Therefore, the sender who wants to share its data directly sends a ciphertext to the receiver. The workflow between the involved parties is illustrated in Figure 4.3. The storage of ciphertexts remains flexible and open to various application contexts. The separated DS can be integrated later depending on the use case, allowing clients to store and access encrypted files in the DS. To support functionality for implementation, we define new data types and implement the encryption/decryption functions with ML-KEM and AES-GCM.
- **Specification of the Cloud Storage Scheme with Untrusted Key Provider.** Moving beyond the basic cloud-based storage scheme with the trusted KP, we consider the untrusted KP. In this model, the KP is treated as trusted only with respect to integrity of clients' public keys, and is required to prove the validity of updates. Therefore, the clients generate their own KEM key pairs and store only public keys in the cloud. Hence, **the KP updates only the public key component and each client updates its secret key**. To achieve this functionality, we need to split the update algorithm into two parts; the public key update algorithm, and the secret key update algorithm. For any public key update, the clients should be able to verify whether their public key updates are valid. The definition of forward secure cloud storage scheme with untrusted KP is provided in Section 4.2.

Figure 4.1 illustrates the simplified cloud-based storage scheme analogous to Figure 5.1 in Deliverable 3.1 [46], but with more detailed description of the cryptographic operations and the interaction between the user and each entity in the CTS. The main difference is who executes the key generation, encapsulation, and secret key updates. Since decryption and decapsulation correspond to encryption and encapsulation, we omit them in the figure for convenience.

4.1 Implementation of the Cloud Storage Scheme with Trusted Key Provider

To support the key update functionality of the PiQASO PQCaaS solution, we will implement the updatable cloud storage, focusing initially on the basic construction, which updates keys by generating a fresh key. Since we need an IND-CCA secure KEM and a symmetric key encryption with IND-CPA security and ciphertext integrity for a secure updatable cloud storage, we will implement with ML-KEM and AES-GCM. In this section, the "server" implicitly denotes the KP.

1. With Trusted Server (KP & DS)



2. With Untrusted Server (KP & DS)

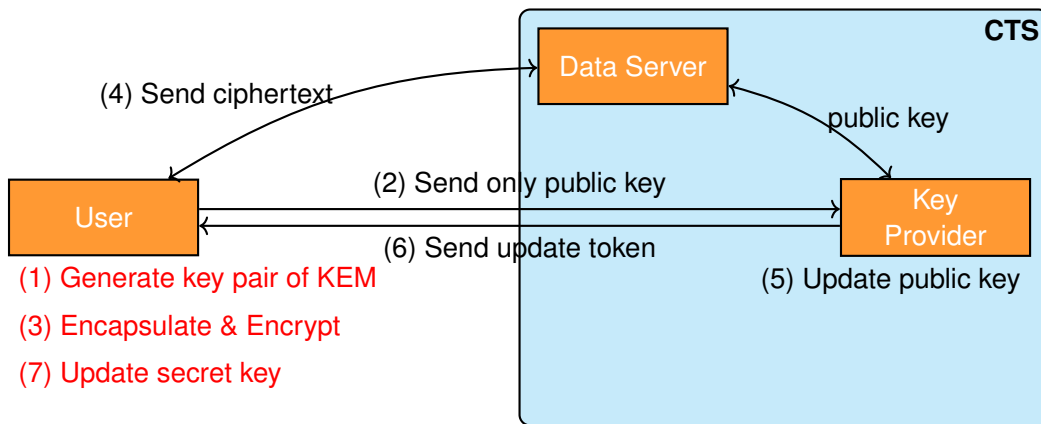


Figure 4.1: System representation under distinct security paradigms

New Data Types. For the key updatability, we introduce new data types in Table 4.1. Since a client's KEM key pairs are updated, the variable for key pairs is an array. We define this data type 'ClientKemKeys'. Another client, who is actually a sender, needs the public key of the receiver not knowing the secret key. Therefore, we define the data type 'ClientKemPK' which only stores the public key of the receiver (if the server (KP) can send only public key part of data type 'ClientKemKeys' such that $pk[epoch]$, we don't need 'ClientKemPK'). To encrypt data, the sender needs a symmetric key which is an encapsulation of the receiver's public key. For this, we define 'ServerEncaps' for the server to store an encapsulation ciphertext in 'kem_ct' and an encapsulated key in 'enc_key'. Since keys are updated, we require an indicator of the epoch that dictates when data is encrypted so the sender also stores 'epoch' in 'create_epoch', where 'epoch' is the global parameter. When the sender encrypts data, it sends a ciphertext to the receiver. Since the cloud storage is based on the hybrid encryption, a ciphertext consists of ML-KEM ciphertext and AES-GCM ciphertext. In addition, to know the epoch when the ciphertext is generated, we also need to store the epoch. Therefore, we define the data structure for a ciphertext 'Ciphertext' which contains 'kem_ct', 'aes_ct', and 'create_epoch'. All these data structure can be seen in Table 4.1.

Functions. With the above data types, the sender and receiver can share data via the cloud storage to execute ML-KEM. Figure 4.3 presents the flow between clients and the server of the updatable cloud

Data Type	Fields		Created by	Passed to
ClientKemKeys	pk[] sk[]	array of public keys array of secret keys	server(KeyGen)	receiver
ClientKemPK	pk		server(KeyGen)	sender
ServerEncaps	kem_ct enc_key create_epoch	ML-KEM ciphertext ML-KEM symmetric key epoch of encapsulation	server(Encaps)	sender
Ciphertext	kem_ct aes_ct create_epoch	ML-KEM ciphertext AES-GCM ciphertext epoch of symmetric key	sender(Enc)	receiver

Table 4.1: The new data type for the updatable cloud storage. The arrays for public keys and secret keys are indexed by a global variable epoch of the integer type. ‘ClientKemKeys’ is declared by a client (the receiver in Figure 4.3) and the components are generated by the server(KeyGen).

storage scheme, while Figure 4.2 presents a detailed description of the cryptographic functions that are executed during the data sharing procedure. Recall that in this instance, we assume that the server is trusted so that it can generate clients’ key pairs and store both the public key and secret key. In addition, the server executes encapsulation and decapsulation instead of the clients.

In order to share data via the cloud storage, a client needs its keys. Actually, the sender doesn’t need its key pair, so we describe that only the receiver requests its key pair in Figure 4.3. The receiver declares its variable ‘receiver’ of the data type ‘ClientKemKeys’, and requests its ML-KEM key pair by sending its variable ‘receiver’ for keys to the server. Then the server generates a key pair by executing the algorithm KeyGen, and send it to the receiver by storing it in ‘receiver’. For the sender to encrypt data, the server also sends the receiver’s public key to the sender using another data structure ‘ClientKemPK’.

After setup, a sender wants to send encrypted data to designated receivers. For convenience, we assume a single receiver. Therefore, the sender requests encapsulation of the receiver’s public key by sending ‘receiver_pk’ to the server. Then, the server executes the algorithm Encaps which runs ML-KEM’s encapsulation algorithm, and sends the variable ‘server’ of the data type ‘ServerEncaps’, which contains the encapsulation key and ciphertext. Now, the sender can encrypt data with the encapsulated key ‘server.enc_key’ by executing the algorithm Enc. Then, the sender sends a ciphertext ‘ct’ of data type ‘Ciphertext’, which consists of the encapsulation ciphertext, the AES-GCM ciphertext, and the epoch.

For data retrieval, the receiver needs the symmetric key to decrypt the ciphertext. Therefore, the receiver sends ‘receiver’ (containing its key pair), ‘ct.kem_ct’, and ‘ct.create_epoch’ to the server to request the decapsulation of the ML-KEM ciphertext ‘ct.kem_ct’. Then, the server runs the algorithm Decaps and sends the symmetric key K to the receiver. Now, the receiver can gain the data by executing the algorithm Decwith ‘ct.aes_ct’ (the AES-GCM ciphertext) and the symmetric key K from the server.

The keys are updated by the server. After a fixed amount of time, the server executes the algorithm UpdateEpoch to increase the global variable ‘epoch’. Then, the server generates a fresh key pair and stores it in the array for keys.

4.2 Specification of the Cloud Storage Scheme with Untrusted Key Provider

Up to this point, we have only considered the scenario in which the servers including the KP and DS are fully trusted, so the KP generates key pairs of users. Therefore, the KP can obtain a data file f by decrypting the ciphertext. When the KP and DS are not fully trusted, however, each user keeps their own

Variable Declaration

```
int epoch = 0;
ClientKemKeys receiver; //Declared by the receiver
```

Variable Declaration in Functions

```
ClientKemPK receiver_pk; //Declared by server in KeyGen
ServerEncaps server; //Declared by the server in Encaps
Ciphertext ct; //Declared by the sender in Enc
```

KeyGen(receiver)

```
ClientKemPK receiver_pk;
(pk, sk)  $\leftarrow$  ML-KEM.KeyGen( $1^\lambda$ )
receiver.pk[epoch] = pk
receiver.sk[epoch] = sk
receiver_pk.pk = pk
return receiver_pk
Encaps(receiver_pk)
```

ServerEncaps server;

```
pk = receiver_pk.pk
( $K, c$ )  $\leftarrow$  ML-KEM.Encaps(pk)
server.enc_key =  $K$ 
server.kem_ct =  $c$ 
server.create_epoch = epoch
return server
Decaps(receiver, ct.kem_ct, ct.create_epoch)
sk = receiver.sk[ct.create_epoch]
 $c = ct.kem\_ct$ 
 $K \leftarrow$  ML-KEM.Decaps(sk,  $c$ )
return  $K$ 
```

UpdateEpoch()

```
epoch ++
UpdateKeys(receiver, receiver_pk)
(pk', sk')  $\leftarrow$  ML-KEM.KeyGen( $1^\lambda$ )
receiver.pk[epoch] = pk'
receiver.sk[epoch] = sk'
receiver_pk.pk = pk'
Enc(server,  $f$ )
Ciphertext ct;
 $K = server.enc\_key$ 
 $C \leftarrow$  AES-GCM.Enc( $K, f$ )
ct.kem_ct = server.kem_ct
ct.aes_ct =  $C$ 
ct.create_epoch = server.create_epoch
return ct
Dec( $K, ct.aes\_ct$ )
 $C = ct.aes\_ct$ 
 $f \leftarrow$  AES-GCM.Dec( $K, C$ )
return  $f$ 
```

Figure 4.2: The variables and the functions for the updatable cloud storage

secret key locally (i.e., it is never stored in the database), while their public key is stored in the cloud. Moreover, in this setting, clients do not trust the updates performed by the KP. In such an environment, updatable public-key encryption provides exactly the functionality we require, namely:

- The possibility for the KP to update the public keys of all users without necessarily knowing the related secret key;
- The possibility, for everyone, to check that a new updated public key is legitimate and has not been set maliciously, for example, by generating a new secret-public key couple and setting it as the new key-pair.

In the following, we are going to define the notion of updatable key encapsulation mechanism. This will be used to construct the updatable version of a hybrid encryption scheme. In the end, we give the new construction for an updatable cloud storage scheme. Therefore, before defining the updatable cloud storage scheme, we give a detailed description of the required building blocks.

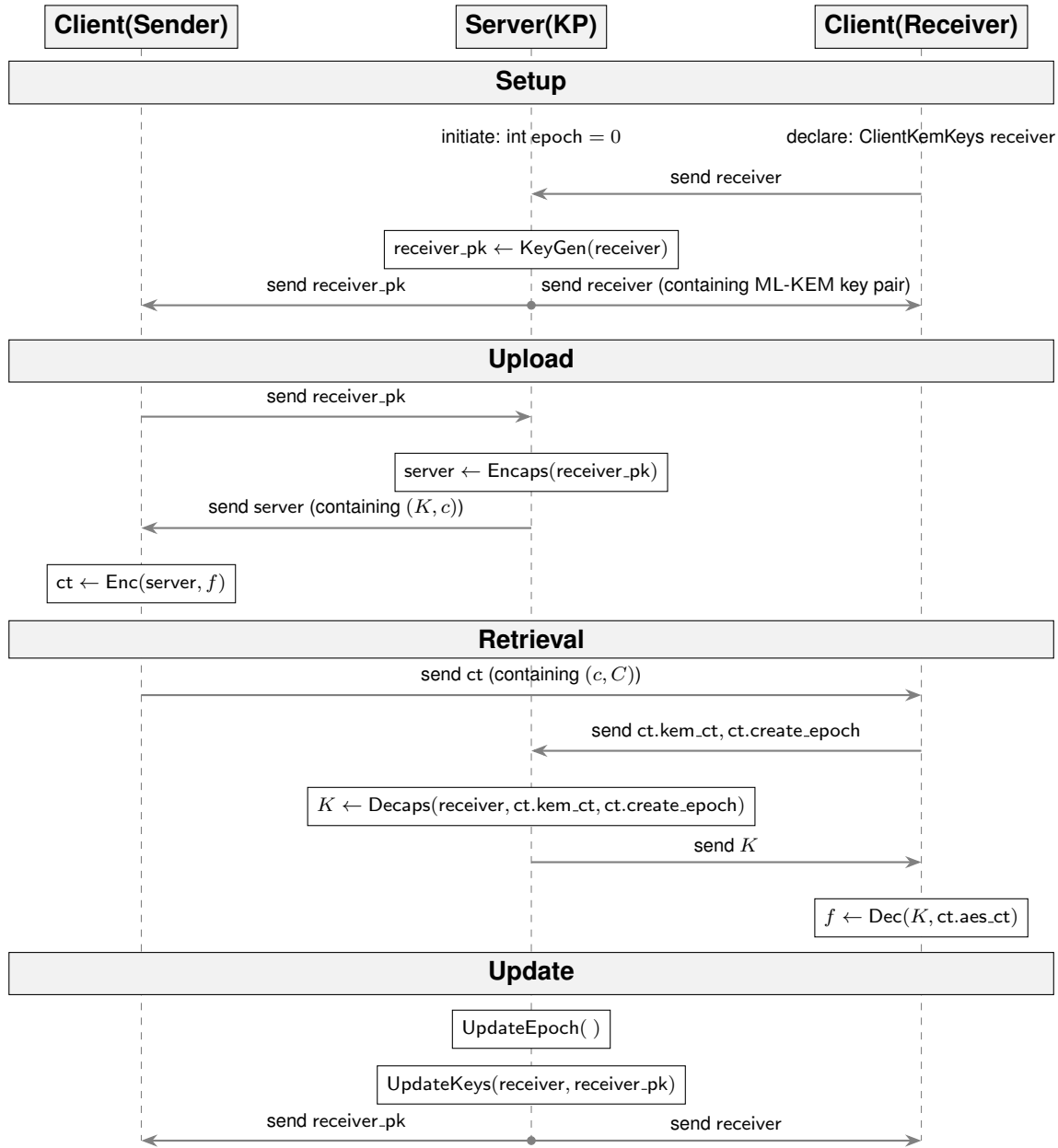


Figure 4.3: The flow between the server and clients

4.2.1 Background and Building Blocks

We recall the notations from Deliverable 3.1. For a vector $\mathbf{v}$ of dimension n and values in a set S we use the notation $\mathbf{v}[1, \dots, n] = s$, for a $s \in S$, to indicate that we initialize all the elements of the vector $\mathbf{v}$ to the value s . Also, by $\mathbf{v} \stackrel{\cup}{\leftarrow} s$ we mean that s is appended to $\mathbf{v}$, resulting in a vector with dimension $n + 1$ and last component s . For a database containing n key couples DB_K , we write $\text{DB}_K.\text{sk}$ and mean $\{\text{DB}_K[i].\text{sk}\}_{i=1}^n$, and the same for $\text{DB}_K.\text{pk}$. In Table 4.2, we describe the notation for UPKE/UKEM and cloud storage.

4.2.1.1 Updatable Key Encapsulation Mechanism (UKEM)

Definition 1 (Updatable Key Encapsulation Mechanism). *An Updatable Key Encapsulation Mechanism (UKEM) is a KEM extended with the functionality to update keys. The scheme UKEM consists of the*

Notation	Description
t	the parameter to describe the epoch
(pk^t, sk^t)	the public/secret key pair at the epoch t
up	the update value which is a randomness to update a public key
Δ	the update token to update a secret key
DB_K, DB_f, DB_P, DB_t	the databases for keys, files, policies, and epoch respectively

Table 4.2: Description of the notation for UPKE/UKEM and forward secure cloud storage.

algorithms

(KeyGen, Encaps, Decaps, UpdatePk, UpdateSk, VerifyUpdate)

- $UKEM.KeyGen(1^\lambda) \rightarrow (pk, sk)$ takes as input the security parameter λ and outputs the public-private key pair.
- $UKEM.Encaps(pk) \rightarrow (c, K)$ takes as input a public key pk and outputs an encapsulation c and a symmetric key K .
- $UKEM.Decaps(sk, c) \rightarrow K$ takes as input a secret key sk and an encapsulation c and outputs a symmetric key K .
- $UKEM.UpdatePk(pk^t; up) \rightarrow (pk^{t+1}, \Delta)$ takes as input a public key pk^t and outputs an updated public key pk^{t+1} and an update token Δ .
- $UKEM.UpdateSk(sk^t, \Delta) \rightarrow sk^{t+1}$ takes as input a secret key sk^t and an update token Δ . Firstly, it verifies the update by running $UKEM.VerifyUpdate(pk^t, (pk^{t+1}, \Delta))$. If the update is valid, it outputs an updated secret key sk^{t+1} .
- $UKEM.VerifyUpdate(pk^t, (pk^{t+1}, \Delta)) \rightarrow 0/1$ takes as input the current public key pk and a proposed update (pk^{t+1}, up) and returns a boolean, verifying the update from pk^t to pk^{t+1} using the update token Δ .

(ℓ, δ) -Correctness. Let $(pk^0, sk^0) \leftarrow UKEM.KeyGen(1^\lambda)$ be a key pair and $\ell > 0$ be an integer. For $t < \ell$, define $(pk^{t+1}, \Delta^{t+1}) \leftarrow UKEM.UpdatePk(pk^t)$ and $sk^{t+1} \leftarrow UKEM.UpdateSk(sk^t, \Delta^{t+1})$. The UKEM scheme is (ℓ, δ) -correct for $\delta > 0$, if for all $t \leq \ell$

$$\mathbb{P}[UKEM.Decaps(sk^t, c) = K | (c, K) \leftarrow UKEM.Encaps(pk^t)] \geq 1 - \delta,$$

Security. There are many definitions of security regarding updatable public key encryption schemes (and related models for updatable key encapsulation mechanisms). In our analysis, we follow the definition proposed in [3], in particular IND-CU-CCA, as it fits better what we want to catch.

Definition 2 (Secure UKEM). An Updatable Key Encapsulation Mechanism (UKEM) is said to be confidential (or, IND-CU-CCA secure) if, for any probabilistic polynomial time algorithm $\mathcal{A}$, we have that

$$\left| \mathbb{P}[\text{Exp}_{\mathcal{A}}^{CU-CCA}(1^\lambda, \ell) = 1] - \frac{1}{2} \right| \leq \text{negl}(\lambda),$$

for a negligible function negl .

$\text{Exp}_{\mathcal{A}}^{\text{CU-CCA}}(1^\lambda, \ell) :$

```

1 :  $\text{sk}^0, \text{pk}^0 \leftarrow \text{KeyGen}(1^\lambda)$ 
2 :  $t = 0$ 
3 :  $b \leftarrow \$ \{0, 1\}$ 
4 :  $st \leftarrow \mathcal{A}^{\mathcal{O}_{\text{Decaps}}, \mathcal{O}_{\text{Update}}}(\text{pk}^0)$ 
5 :  $k_0 \leftarrow \$ \mathcal{K}$ 
6 :  $c^*, k_1 \leftarrow \text{Encaps}(\text{pk}^t)$ 
7 :  $\text{pk}^{\text{chall}} = \text{pk}^t$ 
8 :  $st \leftarrow \mathcal{A}^{\mathcal{O}_{\text{Decaps}}, \mathcal{O}_{\text{Update}}}(c^*, k_b^*, st, \text{pk}^0)$ 
9 :  $up^* \leftarrow \$ \mathcal{R}$ 
10 :  $(\Delta^*, \text{pk}^*) \leftarrow \text{UpdatePk}(\text{pk}^t, up^*)$ 
11 :  $\text{sk}^* \leftarrow \text{UpdateSk}(\text{sk}^t, \Delta^*)$ 
12 :  $b' \leftarrow \mathcal{A}(\text{pk}^*, \text{sk}^*, \Delta^*, c^*, k_b^*)$ 
13 : return  $b \stackrel{?}{=} b'$ 

```

$\mathcal{O}_{\text{Update}}(\text{pk}', \Delta) :$

```

1 : if  $\text{VerifyUpdate}(\text{pk}^t, (\text{pk}', \Delta)) = \perp$  or  $t > \ell$  then return  $\perp$ 
2 :  $\text{pk}^{t+1} \leftarrow \text{pk}'$ 
3 :  $\text{sk}^{t+1} \leftarrow \text{UpdateSk}(\text{sk}^t, \Delta)$ 
4 :  $t \leftarrow t + 1$ 

```

$\mathcal{O}_{\text{Decaps}}(c) :$

```

1 : if  $(\text{pk}^t = \text{pk}^{\text{chall}} \wedge c = c^*)$  then return  $\perp$ 
2 : return  $\text{Decaps}(\text{sk}^t, c)$ 

```

The experiment $\text{Exp}_{\mathcal{A}}^{\text{CU-CCA}}$ models security of a UKEM against an adversary with chosen-update and chosen-ciphertext capabilities. The challenger generates an initial key pair $(\text{pk}^0, \text{sk}^0)$ and gives pk^0 to the adversary, who may adaptively query an update oracle and a decapsulation oracle across epochs. At some challenge epoch t , the challenger encapsulates a key under the current public key pk^t , producing a ciphertext c^* and a real key k_1 , and gives the adversary either k_1 or an independent random key depending on a hidden bit b . The adversary continues to interact with the oracles, after which the challenger performs one final honest update using fresh randomness unknown to the adversary and reveals the resulting public key, secret key, and update information. The adversary wins if it correctly guesses b , capturing the intuition that past encapsulated keys remain indistinguishable from random even under malicious updates, adaptive decapsulation queries, and eventual secret-key exposure, provided that at least one honest update occurs after the challenge.

The update oracle models adversarial control over the key-evolution process. On input a proposed update (pk', Δ) for the current epoch key pk^t , the oracle first checks validity using the deterministic predicate VerifyUpdate . If the update is invalid, the oracle returns $\perp$, and the state remains unchanged. Otherwise, the oracle advances the epoch counter, sets the public key $\text{pk}^{t+1} = \text{pk}'$, and updates the secret key by running $\text{UpdateSk}(\text{sk}^t, \Delta)$. This oracle captures chosen-update power, allowing the adversary to submit arbitrary (possibly malicious) updates, while correctness of the state evolution is enforced by VerifyUpdate . The update operation is limited and can be performed at most ℓ times, according to the correctness of the scheme. After that, an error symbol is returned, as correctness is not guaranteed anymore.

The decapsulation oracle models adaptive chosen-ciphertext access. Given a ciphertext c , the oracle

returns the result of decapsulation under the current secret key sk^t , except in the trivial case where the adversary queries the challenge ciphertext c^* at the challenge public key pk^{chall} . Such queries are disallowed to prevent trivial wins.

4.2.1.2 Updatable Hybrid Encryption Scheme

It would be ideal, for efficiency reasons, to use hybrid encryption for our cloud storage scheme. To support key updatability within this framework, update algorithms for the hybrid encryption are required. To this end, we introduce **updatable hybrid encryption (UHE)**, constructed by replacing the standard KEM with a UKEM within the hybrid encryption framework. Analogous to a standard hybrid encryption defined in Section 5.3.1 of Deliverable 3.1 [46], a UHE scheme consists of the algorithms (KeyGen, Enc, Dec), together with two additional algorithms UpdatePk and UpdateSk that enable the key update functionality.

Definition 3 (UHE). *An Updatable Hybrid Encryption (UHE) scheme consists of the algorithms*

(UHE.KeyGen, UHE.Enc, UHE.Dec, UHE.UpdatePk, UHE.UpdateSk).

The first three algorithms form the standard (non-updatable) HE scheme, while the remaining algorithms (UpdatePk, UpdateSk) enable the key update functionality.

- $\text{UHE.KeyGen}(1^\lambda) \rightarrow (pk, sk)$: *this function takes as input the security parameter λ and outputs the public-private key pair from the underlying UKEM.*
- $\text{UHE.Enc}(pk^t, m) \rightarrow C$: *first, the encapsulation algorithm is executed, producing a ciphertext c_1 and a key k . Then, k is used as the key of the symmetric key encryption scheme to encrypt m , producing c_2 . The output ciphertext is $C = (c_1, c_2)$.*
- $\text{UHE.Dec}(sk^t, C) \rightarrow m$: *this function decapsulates c_1 with a suitable secret key, obtaining a symmetric key k , and uses k as input of the symmetric encryption scheme to decrypt c_2 . The result of this last decryption is returned.*
- $\text{UHE.UpdatePk}(pk^t; up) \rightarrow (pk^{t+1}, \Delta)$: *takes as input a public key pk^t and outputs an updated public key pk^{t+1} and an update token Δ by executing UKEM.UpdatePk .*
- $\text{UHE.UpdateSk}(sk^t, \Delta) \rightarrow sk^{t+1}$: *takes as input a secret key sk^t and an update token Δ , and outputs an updated secret key sk^{t+1} by executing UKEM.UpdateSk .*

$\text{UHE.KeyGen}(1^\lambda)$:

1 : $(sk, pk) \leftarrow \text{UKEM.KeyGen}(1^\lambda)$

$\text{UHE.Enc}(pk^t, m)$:

1 : $c_1, k \leftarrow \text{UKEM.Encaps}(pk^t)$
 2 : $c_2 \leftarrow \text{SE.Enc}(k, m)$
 3 : **return** (c_1, c_2)

$\text{UHE.Dec}(sk^t, C = (c_1, c_2))$:

1 : $k \leftarrow \text{UKEM.Decaps}(sk^t, c_1)$
 2 : **if** $k \neq \perp$ **then**
 3 : **return** $\text{SE.Dec}(k, c_2)$
 4 : **return** $\perp$

UHE.UpdatePk($\text{pk}^t; up$)

1 : **return** $(\text{pk}^{t+1}, \Delta) \leftarrow \text{UKEM.UpdatePk}(\text{pk}^t; up)$

UHE.UpdateSk(sk^t, Δ)

1 : **return** $\text{sk}^{t+1} \leftarrow \text{UKEM.UpdateSk}(\text{sk}^t, \Delta)$

The HE scheme is a form of a PKE. Therefore, if an UHE scheme satisfies forward security of the UPKE, it is a forward secure UHE. We recall the the definition of δ -correctness and IND-CU-CCA security for an UPKE from Deliverable 3.1 [46, Definition 8].

(ℓ, δ) -Correctness: Let $(\text{pk}_0, \text{sk}_0) \leftarrow \text{UHE.KeyGen}(1^\lambda)$ be a key pair and $\ell > 0$ be an integer. For $t < \ell$, define $(\text{pk}_{t+1}, \Delta_{t+1}) \leftarrow \text{UHE.UpdatePk}(\text{pk}_t)$ and $\text{sk}_{t+1} \leftarrow \text{UHE.UpdateSk}(\text{sk}_t, \Delta_{t+1})$. The UHE scheme is (ℓ, δ) -correct for $\delta > 0$, if for all $t \leq \ell$

$$\mathbb{P}[\text{UHE.Dec}(\text{sk}_t, c) = m | c \leftarrow \text{UHE.Enc}(\text{pk}_t, m)] \geq 1 - \delta,$$

Definition 4 (Forward Secure UHE). *An Updatable Hybrid Encryption Scheme is said to be forward secure (or, IND-CU-CCA secure) if, for any probabilistic polynomial time algorithm $\mathcal{A}$, we have that*

$$\left| \mathbb{P}[\text{Exp}_{\mathcal{A}}^{\text{CU-CCA}}(1^\lambda, \ell) = 1] - \frac{1}{2} \right| \leq \text{negl}(\lambda),$$

for a negligible function negl .

As it happens in the non-updatable framework, where the hybrid encryption scheme resulting from the standard KEM/DEM construction inherits the IND-CCA2 property from the underlying KEM, as far as the DEM scheme SE is IND-CPA and INT-CTXT, the same construction, in the updatable framework, inherits the IND-CU-CCA property from the underlying UKEM.

$\text{Exp}_{\mathcal{A}}^{\text{CU-CCA}}(1^\lambda) :$

1 : $\text{sk}^0, \text{pk}^0 \leftarrow \text{UHE.KeyGen}(1^\lambda)$
2 : $t = 0$
3 : $b \leftarrow \$ \{0, 1\}$
4 : $(m_0, m_1, st) \leftarrow \mathcal{A}^{\mathcal{O}_{\text{Dec}}, \mathcal{O}_{\text{Update}}}(\text{pk}^0)$
5 : $c^* \leftarrow \text{UHE.Enc}(\text{pk}^t, m_b)$
6 : $\text{pk}^{\text{chall}} = \text{pk}^t$
7 : $st \leftarrow \mathcal{A}^{\mathcal{O}_{\text{Dec}}, \mathcal{O}_{\text{Update}}}(c^*, st)$
8 : $up^* \leftarrow \$ \mathcal{R}$
9 : $(\Delta^*, \text{pk}^*) \leftarrow \text{UHE.UpdatePk}(\text{pk}^t, up^*)$
10 : $\text{sk}^* \leftarrow \text{UHE.UpdateSk}(\text{sk}^t, \Delta^*)$
11 : $b' \leftarrow \mathcal{A}(\text{pk}^*, \text{sk}^*, \Delta^*, c^*, k_b^*)$
12 : **return** $b \stackrel{?}{=} b'$

$\mathcal{O}_{\text{Update}}(\text{pk}', \Delta) :$

1 : **if** $\text{VerifyUpdate}(\text{pk}^t, (\text{pk}', \Delta)) = \perp$ **or** $t > \ell$ **then return** $\perp$
2 : $\text{pk}^{t+1} \leftarrow \text{pk}'$
3 : $\text{sk}^{t+1} \leftarrow \text{UHE.UpdateSk}(\text{sk}^t, \Delta)$
4 : $t \leftarrow t + 1$

$\mathcal{O}_{\text{Dec}}(c) :$

1 : **if** $(\text{pk}^t = \text{pk}^{\text{chall}} \wedge c = c^*)$ **then return** $\perp$
2 : **return** $\text{Dec}(\text{sk}^t, c)$

Theorem 1 (IND-CU-CCA Security of UHE). *Let UKEM be an IND-CU-CCA updatable key encapsulation mechanism, and SE an IND-CPA and INT-CTXT secure symmetric key encryption scheme. Then, the updatable public key encryption scheme (or updatable hybrid encryption scheme) UHE obtained in the previous construction is IND-CU-CCA.*

Proof. We prove the statement with a sequence of games.

Game 0. The original forward security (or IND-CU-CCA) game for UHE.

$\text{Game}_0(1^\lambda, \ell) :$

```

1 :  $sk^0, pk^0 \leftarrow \text{KeyGen}(1^\lambda)$ 
2 :  $t = 0$ 
3 :  $b \leftarrow \$ \{0, 1\}$ 
4 :  $(m_0, m_1, st) \leftarrow \mathcal{A}^{\mathcal{O}_{\text{Dec}}, \mathcal{O}_{\text{Update}}}(pk^0)$ 
5 :  $(c_1^*, k^*) \leftarrow \text{UKEM.Encaps}(pk^t)$ 
6 :  $c_2^* \leftarrow \text{SE.Enc}(k^*, m_b)$ 
7 :  $pk^{\text{chall}} = pk^t$ 
8 :  $st \leftarrow \mathcal{A}^{\mathcal{O}_{\text{Dec}}, \mathcal{O}_{\text{Update}}}((c_1^*, c_2^*), st)$ 
9 :  $up^* \leftarrow \$ \mathcal{R}$ 
10 :  $(\Delta^*, pk^*) \leftarrow \text{UKEM.UpdatePk}(pk^t, up^*)$ 
11 :  $sk^* \leftarrow \text{UKEM.UpdateSk}(sk^t, \Delta^*)$ 
12 :  $b' \leftarrow \mathcal{A}(pk^*, sk^*, \Delta^*, (c_1^*, c_2^*), k_b^*)$ 
13 : return  $b \stackrel{?}{=} b'$ 

```

Game 1. In this game, we replace the symmetric key k^* obtained by the encapsulation with a random key k from the set of symmetric keys. Let brk be the event in which an adversary can compute a valid ciphertext for the secret key encryption scheme without having computed it with the encryption algorithm. We show that, under the condition that this event does not occur, the two games are indistinguishable. In particular, a distinguisher for the two games would imply the existence of an adversary breaking the IND-CU-CCA security of UKEM.

The reduction forwards the initial key-pair to the adversary, and handles the oracle queries before the challenge in the following way: it replies to update queries faithfully, and for decryption queries of (c_1, c_2) , the reduction queries for decapsulation of c_1 and forwards $\text{SE.Dec}(k, c_2)$. Then, upon receiving (m_0, m_1) , $\mathcal{B}$ sends the challenge request to $\mathcal{C}$ and receives back the ciphertext and key $\hat{c}, \hat{k}$. The reduction replies with the challenge $C = (\hat{c}_1, \hat{c}_2) := (\hat{c}, \text{SE.Enc}(\hat{k}, m_b))$. After this, the reduction still has to manage Oracle queries. Update queries are handled as before. For decryption queries, it works exactly as before every time the reduction is queried with (c_1, c_2) such that $c_1 \neq \hat{c}_1$. Whenever $c_1 = \hat{c}_1$, the reduction replies with $\perp$. The simulation is perfect as long as the adversary does not query the reduction with $\hat{c}_1, c_2$ with $c_2 \neq \hat{c}_2$ and such that c_2 is a valid ciphertext, that is, as long as the event brk does not happen.

In the end, the reduction forwards the new updated public and secret keys, as well as the update token, waits for the response of the adversary, and forwards the opposite bit to the challenger. Therefore, we have that

$$|\mathbb{P}[\text{Game}_0 = 1 | \neg \text{brk}] - \mathbb{P}[\text{Game}_1 = 1 | \neg \text{brk}]| \leq \text{Adv}^{\text{IND-CU-CCA}_{\text{UKEM}}}.$$

Game 2. In this game, we replace c_2^* with the symmetric encryption of a random ciphertext. We show that a distinguisher between these two games implies an adversary against the IND-CPA property of SE. The reduction simulates honestly update and decryption oracles since it generates the asymmetric public and secret keys. When it receives the two messages (m_0, m_1) , for encryption, it computes a random

string m_r of the same length as the two received, and sets $m'_0 = m_0, m'_1 = m_r$ and forwards m'_0, m'_1 to the challenger. The challenger then replies with $c^* = \text{SE.Enc}(k, m'_b)$ for a secret key k . The reduction, then, sends back $\hat{c}_1$ as normally computed in the game and $\hat{c}_2 = c^*$. After the queries, all the oracles are simulated as before. At the end of the game, the adversary provides the answer and the reduction forwards the bit to the challenger. So we have

$$|\mathbb{P}[\text{Game}_1 = 1 | \neg \text{brk}] - \mathbb{P}[\text{Game}_2 = 1 | \neg \text{brk}]| \leq \text{Adv}^{\text{IND-CPA}_{\text{SE}}}.$$

In the end, the view of the adversary is independent of the bit choice, so

$$|\mathbb{P}[\text{Game}_2 = 1 | \neg \text{brk}] = \frac{1}{2}.$$

To end the proof, we show that the event brk happens with negligible probability. In particular, an adversary that triggers the event brk can be used to attack the INT-CTXT property of SE. Indeed, the reduction can simply ask the challenger for an SE encryption and forward it to the adversary as second component of the challenge ciphertext. Then, the adversary submits a new ciphertext with different second component, and this is forwarded by the reduction to the challenger. $\square$

4.2.2 Updatable Cloud Storage Scheme with Untrusted Key Provider

The syntax includes, as before, a setup, an upload, an update, and a retrieval algorithm. For an easier analysis, this time we suppose that none of the parties taking part in the protocol have their secret key stored on the cloud.

Each user generates its own public/secret key pair $(pk_i, sk_i) \leftarrow \text{KeyGen}(1^\lambda)$ then stores only the public key pk_i to the cloud. After receiving public keys from all users, the DS sets up the databases by initializing $\text{DB}_f, \text{DB}_P, \text{DB}_t$ and storing pk_i 's in DB_K .

A user want to upload a file which can be accessed by users who satisfy the specific policy. Since the database doesn't have any secret key, the user encrypts the file f by itself with the associated public keys of users (recipients) whom the user (sender) wants to give the access to the file f . Then, the user sends the ciphertexts $\{C_i\}_{i \in P}$ of the file f and the associated policy P to the DS, and the DS stores the given ciphertexts in $\text{DB}_f[\text{ID}_f]$ and the policy in DB_P where ID_f is the string identifying the file f and outputs ID_f . Also, the DS stores the epoch when the file f is uploaded in $\text{DB}_t[\text{ID}_f]$.

A user, who want to retrieve the file f , sends its ID ID_U and the file ID ID_f to the DS. Then, the DS checks the user's ID ID_U is included in the policy P which is stored in $\text{DB}_P[\text{ID}_f]$. If it is, the DS sends the ciphertext C_{ID_U} , which is stored in $\text{DB}_f[\text{ID}_f]$, to the user ID_U . Then, the user can retrieve the file f by decrypting C_{ID_U} with its own secret key at the epoch $\text{DB}_t[\text{ID}_f]$.

At update time, the KP updates all the public keys and sends the update tokens to all the users. Each user, then, updates their secret key privately. Therefore, there are two update algorithms, one for public keys and one for private keys. The detailed description of the algorithms are as follows:

- Key Generation ($\text{KeyGen}(1^\lambda)$) takes as input a security parameter λ , and returns a public and private key pair (pk, sk) .
- Setup ($\text{Setup}(1^\lambda, \{pk_i^0\}_{i \in [n]})$) takes as input a security parameter λ and public keys of n users, and returns three initialized databases (lists), namely a database for files DB_f , a database for public keys $\text{DB}_K.pk$, a database for policies DB_P , and a database for epoch DB_t . In particular, DB_f and DB_P are empty in the beginning, while $\text{DB}_K.pk$ contains n public keys. The databases DB_P, DB_f and $\text{DB}_K.pk$ are publicly accessible after generation;

- Data Upload (Upl) consists of two algorithm $\text{Upl} = (\text{Upl}_u, \text{Upl}_s)$. The data encrypt algorithm $\text{Upl}_u(f, P)$ is executed by an user, and the upload request algorithm $\text{Upl}_s(\{C_j\}_{j \in P}, P)$ is executed by the DS.
 - $\text{Upl}_u(f, P, \{\text{pk}_j^t\}_{j \in P})$ takes as input a file $f \in F$, a policy P , and public keys $\{\text{pk}_j^t\}_{j \in P}$, encrypts the file f with all public keys pk_j^t which are corresponding public keys of users $j \in P$ at the epoch t , then returns encrypted files $\{C_j\}_{j \in P}$.
 - $\text{Upl}_s(\{C_j\}_{j \in P}, P)$ takes as input a file $f \in F$ and a policy $P \in 2^{[n]}$, possibly changes the state of DB_f and DB_P and returns a file identification string ID_f , or a special error symbol $\perp$;
- Data Retrieval (Ret) also consists of two algorithms $\text{Ret} = (\text{Ret}_s, \text{Ret}_u)$. The data request algorithm $\text{Ret}_s(\text{ID}_U, \text{ID}_f)$ is executed by the DS, and the data decrypt algorithm $\text{Ret}_u(\text{sk}_{\text{ID}_U}^t, C)$ is executed by a user ID_U .
 - $\text{Ret}_s(\text{ID}_U, \text{ID}_f)$ takes as input a user identification string ID_U , and a file identification string ID_f and outputs either an encrypted file $\text{DB}_f[\text{ID}_f][2]$ if $\text{ID}_U \in \text{DB}_P[\text{ID}_f]$ or a special error symbol $\perp$.
 - $\text{Ret}_u(\text{sk}_{\text{ID}_U}^{\text{DB}_f[\text{ID}_f]}, C)$ takes as input a secret key $\text{sk}_{\text{ID}_U}^{\text{DB}_f[\text{ID}_f]}$ and an encrypted file C which is a output of Ret_s , decrypts C , and returns a recovered file f .
- Public-key Update ($\text{UpdatePk}(up_1, \dots, up_n)$) takes as input n update values $up_1, \dots, up_n$, uses the public keys stored in the database, and computes n new public keys which are stored in the database DB_K , and outputs n update tokens $\Delta_1, \dots, \Delta_n$
- Secret-key Update ($\text{UpdateSk}(\text{sk}_i^{t-1}, \Delta_i)$) takes as input a secret key sk_i and update tokens Δ_i , and checks the validity of the update token using the algorithm VerifyUpdate . If it is valid, it outputs the updated secret key sk_i^t
- Verification $\text{VerifyUpdate}(\text{pk}_i^{t-1}, (\text{pk}_i^t, \Delta_i))$ takes as input a public key pk and a proposed update $(\text{pk}_i^t, \Delta_i)$, and returns a boolean value. The verification algorithm checks validity of an update.

(ℓ, δ) -Correctness: Let $(\text{pk}^0, \text{sk}^0) \leftarrow \text{KeyGen}(1^\lambda)$ be a key pair and $\ell > 0$ be an integer. For $t < \ell$, define $(\text{pk}_i^{t+1}, \Delta_i) \leftarrow \text{UpdatePk}(\text{pk}_i^t)$ and $\text{sk}_i^{t+1} \leftarrow \text{UpdateSk}(\text{sk}_i^t, \Delta_i)$. The cloud storage scheme is (ℓ, δ) -correct for $\delta > 0$, if for all $t \leq \ell$

$$\mathbb{P} \left[\text{Ret}(\text{sk}_{\text{ID}_U}^t, C) = f \mid \begin{array}{l} \{C_j\}_{j \in P} \leftarrow \text{Upl}_u(f, P) \\ \text{ID}_f \leftarrow \text{Upl}_s(\{C_j\}_{j \in P}, P) \\ C \leftarrow \text{Ret}_s(\text{ID}_U, \text{ID}_f) \end{array} \right] \geq 1 - \delta$$

Security. The trusted KP generates a key pair for a user and stores both public and secret key in the database DB_K . However, the untrusted KP can not store secret keys. In the experiment, the challenger needs to store secret keys to answer to queries to the oracle $\mathcal{O}_{\text{comp}}$. Therefore, the challenger stores both public and secret key in DB_K and sends only the public key part of the database $\text{DB}_K.\text{pk}$

Definition 5 (Forward Secure Cloud Storage). *A Cloud Storage Scheme is said to be forward secure in the untrusted server model if, for any probabilistic polynomial time algorithm $\mathcal{A}$, we have that*

$$\left| \mathbb{P}[\text{Exp}_{\mathcal{A}}^{\text{forwSec2}}(1^\lambda, \ell, n) = 1] - \frac{1}{2} \right| \leq \text{negl}(\lambda),$$

for a negligible function negl .

$\text{Exp}_{\mathcal{A}}^{\text{forwSec2}}(1^\lambda, n, \ell) :$

```

1 :   $t = 0; \text{corr}[1, \dots, n] = \text{false}$ 
2 :  for  $i \in [n]$  do :
3 :     $(\text{pk}_i^t, \text{sk}_i^t) \leftarrow \text{KeyGen}(1^\lambda)$ 
4 :     $\text{DB}_K.\text{sk}[i][t] \leftarrow \text{sk}_i^t$ 
5 :   $\text{DB}_f, \text{DB}_P, \text{DB}_K.\text{pk}, \text{DB}_t \leftarrow \text{Setup}(1^\lambda, \{\text{pk}_i^0\}_{i \in [n]})$ 
6 :   $(f_0, f_1, P^*) \leftarrow \mathcal{A}^{\mathcal{O}_{\text{Upl}}, \mathcal{O}_{\text{Ret}}, \mathcal{O}_{\text{comp}}, \mathcal{O}_{\text{Update}}}(\text{DB}_f, \text{DB}_P, \text{DB}_K.\text{pk})$ 
7 :   $b \leftarrow \$ \{0, 1\}$ 
8 :   $st \leftarrow \mathcal{A}^{\mathcal{O}_{\text{Upl}}, \mathcal{O}_{\text{Ret}}, \mathcal{O}_{\text{comp}}, \mathcal{O}_{\text{Update}}}(\text{DB}_f, \text{DB}_P, \text{DB}_K.\text{pk})$ 
9 :   $(up_1, \dots, up_n) \leftarrow \$ \mathcal{R}^n$ 
10 :  $\{C_j\}_{j \in P} \leftarrow \text{Upl}_u(f_b, P^*)$ 
11 :  $\text{ID}_f^* \leftarrow \text{Upl}_s(\{C_j\}_{j \in P}, P^*)$ 
12 :  $\Delta_1, \dots, \Delta_n \leftarrow \text{UpdatePk}(up_1, \dots, up_n)$ 
13 : for  $i$  in  $1 \dots n$  do :
14 :    $\text{sk}_i^t \leftarrow \text{UpdateSk}(\text{sk}_i^{t-1}, \Delta_i)$ 
15 :    $b' \leftarrow \mathcal{A}(\text{ID}_f^*, \text{DB}_f, \text{DB}_P, \text{DB}_K.\text{pk}, \text{DB}_K.\text{sk}^t, \{\Delta_i\}_{i \in [n]})$ 
16 : return  $b == b' \wedge \forall i \in P^*. \neg \text{corr}[i]$ 

```

$\mathcal{O}_{\text{Upl}}(f, P)$ <pre> 1 : $\{C_j\} \leftarrow \text{Upl}_u(f, P)$ 2 : return $\text{ID}_f \leftarrow \text{Upl}_s(\{C_j\}_{j \in P}, P)$ </pre>	$\mathcal{O}_{\text{Ret}}(\text{ID}_U, \text{ID}_f)$ <pre> 1 : if $\text{ID}_U \in P^* \wedge \text{ID}_f = \text{ID}_f^*$ then return $\perp$ 2 : $C \leftarrow \text{Ret}_s(\text{ID}_U, \text{ID}_f)$ 3 : if $C = \perp$ then return $\perp$ 4 : return $f \leftarrow \text{Ret}_u(\text{sk}_{\text{ID}_U}^t, C)$ </pre>
---	--

$\mathcal{O}_{\text{comp}}(i)$ <pre> 1 : if $\text{corr}[i]$ then return $\perp$ 2 : $\text{sk}_i^t \leftarrow \text{DB}_K.\text{sk}[i][t]$ 3 : $\text{corr}[i] = \text{true}$ 4 : return sk_i^t </pre>	$\mathcal{O}_{\text{Update}}((\text{pk}'_1, \dots, \text{pk}'_n), (\Delta_1, \dots, \Delta_n))$ <pre> 1 : for $i \in [n]$ 2 : if $\text{VerifyUpdate}(\text{DB}_K.\text{pk}[i][t], (\text{pk}'_i, \Delta_i)) = \perp$ then return $\perp$ 3 : for $i \in [n]$ 4 : $\text{pk}_i^{t+1} \leftarrow \text{pk}'_i$ 5 : $\text{sk}_i^{t+1} \leftarrow \text{UpdateSk}(\text{sk}_i^t, \Delta_i)$ 6 : $t++$ </pre>
--	---

The difference is that, this time, the KP is supposed to be trusted only with respect to integrity of clients' public keys, and can be malicious with respect to key updates so it is required to prove the validity of updates. In particular, the KP could try to sample a key pair and pretend the resulting public key is the result of the update algorithm on the previous one. More precisely, the update oracle consists exactly in calling n times the update oracle described for UKEM, i.e., it takes as input n public keys $\text{pk}'_1, \dots, \text{pk}'_n$ and n update values $up_1, \dots, up_n$ and executes the update oracle of UKEM with (pk'_i, up_i) for $i = 1, \dots, n$. If any of the operations return an error, then the algorithm also returns an error.

4.2.2.1 Instantiation with UHE

The instantiation with UHE structurally follows the instantiation with standard hybrid encryption. Aside from updating keys, the other algorithms work the same way, using UHE for file encryption and decryption. The difference regarding the update algorithms is that the KP can now only update the public keys, and has to issue update tokens so that clients can update their own secret key. Moreover, the KP needs to

update public keys of all n users. This is simply realized with n calls to the UpdatePk algorithm of UHE. On the client side, the UpdateSk is realized by calling the UpdateSk algorithm of UHE.

KeyGen(1^λ)

1 : **return** $(sk_i^0, pk_i^0) \leftarrow \text{UHE.KeyGen}(1^\lambda)$

Setup($1^\lambda, \{pk_i^0\}_{i \in [n]}$)

1 : **global** $\text{count}_f \leftarrow 0, t \leftarrow 0, \text{DB}_f, \text{DB}_P, \text{DB}_t, \text{DB}_K.pk[1, \dots, n][0] \leftarrow \epsilon$
2 : **for** $i = 1$ **to** n **do**
3 : $\text{DB}_K.pk[i][0] \leftarrow pk_i^0$
4 : **return** $(\text{DB}_f, \text{DB}_P, \text{DB}_K, \text{DB}_t)$

Upl_u($f, P, \{pk_j^t\}_{j \in P}$)

1 : **if** $P = \emptyset$ **then**
2 : **return** $\perp$
3 : **for** $j \in P$
4 : $C_j \leftarrow \text{UHE.Enc}(pk_j^t, f)$
5 : **return** $\{C_j\}_{j \in P}$

Upl_s($\{C_j\}_{j \in P}, P$)

1 : $\text{ID}_f \leftarrow \text{count}_f++$
2 : **for** $j \in P$
3 : $\text{DB}_f[\text{ID}_f] \leftarrow^{\cup} (j, C)$
4 : $\text{DB}_P[\text{ID}_f] \leftarrow P$
5 : $\text{DB}_t[\text{ID}_f] \leftarrow t$
6 : **return** ID_f

Ret_s(ID_U, ID_f) :

1 : **if** $\text{ID}_f > |\text{DB}_f|$ **then return** $\perp$
2 : $P \leftarrow \text{DB}_P[\text{ID}_f]$
3 : $t \leftarrow \text{DB}_t[\text{ID}_f]$
4 : **if** $\text{ID}_U \notin P$ **then return** $\perp$
5 : **for** $\text{tuple} \in \text{DB}_f[\text{ID}_f]$
6 : **if** $\text{tuple}[1] == \text{ID}_U$ **then**
7 : **return** $C = \text{tuple}[2]$
8 : **return** $\perp$

Ret_u($sk_{\text{ID}_U}^{\text{DB}_t[\text{ID}_f]}, C$)

1 : **return** $f \leftarrow \text{UHE.Dec}(sk_{\text{ID}_U}^{\text{DB}_t[\text{ID}_f]}, C)$

UpdatePk($up_1, \dots, up_n$) :

1 : $t++$
2 : **for** $i = 1$ **to** n **do**
3 : $pk_i^t, \Delta_i \leftarrow \text{UHE.UpdatePk}(\text{DB}_K[i][t-1].pk, up_i)$
4 : $\text{DB}_K[i][t] \leftarrow pk_i^t$
5 : **return** $\Delta_1, \dots, \Delta_n$

UpdateSk(sk_i^{t-1}, Δ_i)

1 : **return** $sk_i^t \leftarrow \text{UHE.UpdateSk}(sk_i^{t-1}, \Delta_i)$

VerifyUpdate($pk_i^{t-1}, (pk_i^t, \Delta_i)$)

1 : **return** $\text{UHE.VerifyUpdate}(pk_i^{t-1}, (pk_i^t, \Delta_i))$

Security Proof. We will prove the forward security of the cloud storage based on the updatable hybrid encryption. It is implied by the forward security(IND-CU-CCA) of the updatable hybrid encryption scheme.

Theorem 2 (Forward Security of Cloud Storage Scheme). *If the UHE is IND-CU-CCA secure, then the cloud storage scheme defined above is forward secure in the untrusted server model.*

Proof. We prove the forward security via the reduction. Suppose that an adversary $\mathcal{A}$ breaks the forward security of the cloud storage scheme with a non-negligible probability $p(\lambda)$. Then we can build an adver-

sary $\mathcal{B}$ who breaks the IND-CU-CCA security of UHE. As before, we assume that the adversary $\mathcal{A}$ does not corrupt any parties in the challenge policy, and remove the final check that enforces this condition. Then, removing this check does not change the adversary $\mathcal{A}$'s winning probability.

Given pk , $\mathcal{B}$ runs the forward security experiment. $\mathcal{B}$ randomly chooses $I \leftarrow \$ [n]$ and sets $\text{pk}_I^0 := \text{pk}$. $\mathcal{B}$ generates keys of the other users by running key generation algorithm of UHE, i.e., $(\text{pk}_i^0, \text{sk}_i^0) \leftarrow \text{UHE.KeyGen}(1^\lambda)$. Then, $\mathcal{B}$ runs Setup to initialize the databases and store pk_i^0 's. Additionally, $\mathcal{B}$ stores sk_i^0 's in the database $\text{DB}_K.\text{sk}$, i.e., $\text{DB}_K.\text{sk}[i][0] \leftarrow \text{sk}_i^0$ for all $i \in [n] \setminus \{I\}$ but doesn't send it to $\mathcal{A}$. $\mathcal{B}$ will simulate all the oracles in the natural way except for:

- $\mathcal{O}_{\text{Ret}}(\text{ID}_U, \text{ID}_f)$: Since $\mathcal{B}$ generates keys of all users except for the user I , $\mathcal{B}$ can answer queries in the natural way by decrypting $\text{DB}_f[i][2]$ when $\text{ID}_U \neq I$. When $\text{ID}_U = I$, $\mathcal{B}$ gets f by querying to $\mathcal{O}_{\text{UHE.Dec}}(\text{DB}_f[\text{ID}_f][2])$ and returns it.
- $\mathcal{O}_{\text{Update}}((\text{pk}'_1, \dots, \text{pk}'_n), (\Delta_1, \dots, \Delta_n))$: For I , $\mathcal{B}$ queries to $\mathcal{O}_{\text{UHE.Update}}(\text{pk}'_I, \Delta_i)$ to update the secret key of the user I . If $\mathcal{B}$ gets $\perp$, it also answer $\perp$ to $\mathcal{A}$. Otherwise, $\mathcal{B}$ stores $\text{DB}_K.\text{pk}[I][t+1] \leftarrow \text{pk}'_I$. For the other index $i \neq I$, $\mathcal{B}$ updates keys as the $\mathcal{O}_{\text{Update}}$ in the definition.

When $\mathcal{A}$ outputs the challenge files and policy (f_0, f_1, P^*) , if $I \notin P^*$, $\mathcal{B}$ aborts. Otherwise, $\mathcal{B}$ outputs (f_0, f_1) as the challenge messages. Given the last update $(\text{pk}^*, \text{sk}^*, \Delta^*)$, $\mathcal{B}$ sets it as the user I 's last update, that is, $\Delta_I := \Delta^*$, $\text{DB}_f[I][t] = \text{pk}_I^t = \text{pk}^*$, and $\text{DB}_K.\text{sk}[I][t] = \text{sk}_I^t = \text{sk}^*$. Then, $\mathcal{B}$ updates the other users' keys and sends $(\text{ID}_f^*, \text{DB}_f, \text{DB}_P, \text{DB}_K.\text{pk}, \text{DB}_K.\text{sk}^t, \{\Delta_i\}_{i \in [n]})$ to $\mathcal{A}$ where $\text{DB}_K.\text{sk}^t$ is the set of secret keys in the current epoch.

At the end of the experiment, $\mathcal{A}$ outputs b , then $\mathcal{B}$ also outputs b . If the $\mathcal{A}$ can distinguish which file is encrypted, $\mathcal{B}$ also can distinguish. $\mathcal{B}$ aborts the experiment when $I \notin P^*$. Therefore, the probability that $\mathcal{B}$ doesn't abort and wins is $p'(\lambda) = \frac{|P^*|}{n}p(\lambda)$. Since the parameters $|P^*|$ and n are the polynomial in λ and $p(\lambda)$ is non-negligible, the probability $p'(\lambda)$ is also non-negligible. By the contradiction, the cloud storage scheme is forward secure in the untrusted server model. $\square$

Chapter 5

Homomorphic Functional Encryption

5.1 Motivation

The massive increase in the use of sensitive data, i.e., healthcare, financial data, etc., in modern computation infrastructures has raised the importance of security and privacy-preserving mechanisms in real-world applications more than ever before. While traditional cryptography addresses the security of data both at rest and in transit, it does not satisfy the full needs of modern applications, especially those that outsource data to service providers offering better computational resources. Modern cryptography offers promising approaches, namely Homomorphic Encryption (HE) and Functional Encryption (FE), that enable computation on encrypted data. While HE enables the execution of arbitrary operations on encrypted data, it has two important drawbacks: (1) the concept of *all-or-nothing*, similar to other public-key encryption schemes, means that the holder of the secret key can access *all* of the encrypted data, and therefore there is no proper access control over the computation results; (2) HE introduces high resource usage per computation, even for simple operations, which is not acceptable in many real-world applications.

On the other hand, FE enables fine-grained access control over computation on encrypted data by generating specific keys, known as functional decryption keys, tied to unique functions. Although this is a very interesting feature, almost all FE schemes rely on a *trusted* authority, who holds the master secret key required to generate functional decryption keys. Moreover, the trusted authority always has access to the plaintext function, which is often a sensitive asset for the function owner. Altogether, FE security highly relies on *trust* in the third entity, which is not satisfactory in most real-world applications.

To address the need for a trusted authority and to secure not only the input data but also the function in the case of even an honest but curious entity, we combine the strengths of HE and FE and propose a Homomorphic Functional Encryption (HFE) scheme. Our construction is scheme-agnostic and can be used as a general design that adapts to any FE and HE schemes, based on the application's data types and operations, as long as the intermediate plaintext and ciphertext spaces match.

Use-Cases TELLU (UC5.1) and PARTICULE (UC5.2) A particularly relevant application of our Homomorphic Functional Encryption (HFE) scheme is privacy-preserving healthcare analytics, such as the TELLU and PARTICULE use cases considered in the PiQASO project. In these scenarios, encrypted patient medical records are shared and made accessible to multiple categories of healthcare professionals. For instance, a general practitioner, a cardiologist, or an oncology specialist may each be permitted to evaluate different diagnostic functions over the same encrypted dataset, e.g., heart-rate or ECG-related metrics.

In such settings, revealing the requested function itself may leak sensitive patient details, the intent of

the medical analysis, or even the specialization of the physician performing the query. By leveraging the function-hiding property of our HFE scheme, we protect not only the confidentiality of the underlying patient data but also the nature of the medical analysis being conducted. HFE provides the necessary framework by enabling a key curator, i.e., the **PQCaaS key provider**, to generate functional decryption material without learning which specific medical computation is requested, thereby significantly reducing both information leakage and trust assumptions in collaborative healthcare infrastructures.

5.2 Introduction

HFE addresses the limitations of traditional FE schemes by combining functional encryption with homomorphic encryption. The main idea of HFE is to homomorphically evaluate the functional key derivation algorithm over encrypted material, thereby hiding the requested function while allowing the remaining parts of the FE scheme to operate as in the traditional model. In HFE, the trusted authority is replaced by a key curator, whose role is restricted to generating and distributing encrypted functional decryption keys requested by a data analyst. This design introduces an additional privacy characteristic, namely function blindness, where the requested function remains encrypted so that the key curator can assist in the computation without learning which function is being requested.

Compared to approaches such as Multi-Party Computation (MPC), HFE follows a different trust model by relying on homomorphic encryption rather than distributing trust among several parties. However, this stronger privacy model comes at the cost of high computational overhead during the functional key derivation phase, since homomorphic evaluation of FE key derivation algorithms remains expensive. The security of the scheme also depends on a non-collusion assumption between the key curator and the data analyst, as collusion between these entities could reveal FE secret information.

5.2.1 Roadmap

This chapter is organized as follows. Sect. 5.4 provides the necessary background (i.e., notations and definitions) needed to understand this work. Sect. 5.5 describes how one can homomorphically evaluate the functional circuits to build a correct transciphering framework. Sect. 5.6 provides a formal definition of the encryption scheme obtained by evaluating the FE algorithms homomorphically. In Sect. 5.7, we demonstrate how our HFE scheme can be utilized in real-world applications. Sect. 5.8 provides a detailed and formal definition of the semantic security required for an HFE scheme. Sect. 5.9 reports the benchmarking results for the HFE protocol. Finally, Sect. 5.10 concludes this chapter.

5.3 High-Level Overview

To initiate our HFE scheme, the parties (i.e., the user, curator, and analyst) agree on an HE and an FE scheme. The user generates locally the master keys msk , mpk while the analyst has his own unique keys sk and pk . The encryption of the data is done as in a classic FE setup: The user encrypts his data using the master public key mpk . The main difference lies in the way functional decryption keys are generated. Instead of running the key derivation algorithm using msk and a function f , the curator only receives their encryption under the HE scheme, that is c_{msk} and c_f . More precisely, the user who owns msk , encrypts it homomorphically with pk and sends the result c_{msk} to the curator. At a later point, when an analyst wishes to calculate a function f on a set of data (ciphertexts), she encrypts the function f homomorphically with pk and sends the result c_f to the curator. Following, the curator runs the homomorphic *evaluation*

algorithm on the key derivation circuit. This technique is known as *transcipherring*. The output of this computation is a homomorphic encryption of the decryption key dk_f , which can only be decrypted by the party holding the corresponding secret key sk , for example, a data analyst. The final result is obtained by running the classic FE decryption algorithm using the decryption key dk_f on encrypted data given by the user.

5.4 Background

This section provides the necessary mathematical background and definitions for understanding this chapter.

Notation: For $n \in \mathbb{N}^*$, $[n]$ is the set $\{1, \dots, n\}$. Vectors are written in bold lowercase letters, and ones denotes $\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^n x_i y_i$, the inner product of two vectors $\mathbf{x} = (x_1, \dots, x_n)$ and $\mathbf{y} = (y_1, \dots, y_n)$. For a set $\mathcal{Y}$, one uses $x \leftarrow_s \mathcal{Y}$ if x is sampled at random from $\mathcal{Y}$. Generally, we write $x \stackrel{\chi}{\leftarrow} (\mathcal{Y})$ if x is sampled from $\mathcal{Y}$ following a distribution χ . The image, or range, of a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ is denoted $\text{Im}(f) \subseteq \mathcal{Y}$. The probability of an event $A \in \Omega$ is denoted $\mathbb{P}(A)$, where $\mathbb{P} : \Omega \rightarrow [0, 1]$ is a probability measure on a set Ω usually omitted for brevity. A (probabilistic) polynomial time algorithm $\mathcal{A}$, or (P)PT, is a (randomized) algorithm for which there exists a polynomial $P(x)$ s.t. for all input x , the running time of $\mathcal{A}(x)$ is bounded by $|P(x)|$. A function $f : \mathbb{N} \mapsto \mathbb{R}$ is said negligible if $\forall \delta \in \mathbb{N}, \exists x_0 \in \mathbb{N}$ such that $\forall x \geq x_0, f(x) < x^{-\delta}$. One denotes $\text{negl}(\cdot)$ an arbitrary negligible function.

Throughout the chapter, we often refer to the following parties: user (**U**), key curator (**KC**), and data analyst (**DA**). To avoid any misunderstanding while referring to them in the text, we use the pronouns *he* for a user, *it* for the curator, and *she* for the analyst.

5.4.1 Encryption Schemes

Definition 6 (Public-Key Encryption (PKE)). *A Public-Key Encryption scheme for message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$ consists of three polynomial time algorithms KGen, Enc, Dec defined as follows:*

- $\text{KGen}(1^\lambda)$. *The key generation algorithm is a probabilistic algorithm. On the input of a security parameter λ , it outputs a pair of keys (sk, pk) , respectively the secret and public keys;*
- $\text{Enc}(pk, x)$. *The encryption algorithm is a probabilistic algorithm. On the input of a public key pk and a message m in the message space $\mathcal{M}$, it outputs a ciphertext c in the ciphertext space $\mathcal{C}$. One also writes $c \leftarrow_s \text{Enc}_{pk}(x)$;*
- $\text{Dec}(sk, c)$. *The decryption algorithm is a deterministic algorithm. On the input of a secret key sk and a ciphertext c , it outputs a message x . One also writes $\text{Dec}_{sk}(c) \rightarrow x$.*

Correctness: *For all parameters $\lambda \in \mathbb{N}$, $(sk, pk) \leftarrow_s \text{KGen}(1^\lambda)$ and ciphertext $c \leftarrow_s \text{Enc}_{pk}(x)$, $\mathbb{P}[\text{Dec}_{sk}(c) \neq x] = \text{negl}(\lambda)$.*

The security of a PKE scheme relies on the fact that the encryption algorithm Enc is probabilistic. For well-chosen parameters, it is unlikely for two ciphertexts $c \leftarrow_s \text{Enc}_{pk}(x)$ and $c' \leftarrow_s \text{Enc}_{pk}(x)$ of the same plaintext to be equal. However, these are both valid encryptions of x and hence can be correctly decrypted. We call the set of such ciphertexts the *image* of $\text{Enc}_{pk}(x)$ and denote it as $\text{Im}(\text{Enc}_{pk}(x))$. More formally,

$$\text{Im}(\text{Enc}_{pk}(x)) = \{c \in \mathcal{C} \mid \text{Dec}_{sk}(c) = x\}$$

(Approximate) Homomorphic Encryption: Following, we provide the definition of HE. Since our transpiling solution is parametrized by an arbitrary HE scheme, we decide to use a global definition that includes both *approximate* and *exact* HE schemes. We discuss the difference between these two paradigms later on.

Definition 7 (Homomorphic Encryption). *A Homomorphic Encryption scheme HE for message space $\mathcal{M}_{\text{HE}}$, ciphertext space $\mathcal{C}_{\text{HE}}$ and class of functions $\mathcal{F}_{\text{HE}} : \mathcal{M}_{\text{HE}}^n \rightarrow \mathcal{M}_{\text{HE}}$, $n \in \mathbb{N}^*$, consists of four polynomial time algorithms HE.KGen, HE.Enc, HE.Eval, HE.Dec defined as follows:*

- HE.KGen (1^λ). *The key generation algorithm is a probabilistic algorithm. On the input of a security parameter λ , it outputs a tuple of keys $(\text{sk}, \text{pk}, \text{evk})$, respectively the secret, public and evaluation keys;*
- HE.Enc (pk, x, Δ). *The encryption algorithm is a probabilistic algorithm. On the input of a public key pk , a message x in the message space $\mathcal{M}_{\text{HE}}$ and a scaling factor $\Delta \in \mathbb{N}$, it outputs a ciphertext c in the ciphertext space $\mathcal{C}_{\text{HE}}$. One also writes $c \leftarrow_{\$} \text{HE.Enc}_{\text{pk}}(x, \Delta)$;*
- HE.Eval ($\text{evk}, f, (c_1, \dots, c_n)$). *The evaluation algorithm is a deterministic algorithm that, on input the evaluation key evk , a function $f \in \mathcal{F}_{\text{HE}}$ and n ciphertexts $c_1, \dots, c_n$, outputs a unique ciphertext c . One can also write $\text{HE.Eval}_{\text{evk}}(f, (c_1, \dots, c_n)) \rightarrow c$;*
- HE.Dec (sk, c, Δ). *The decryption algorithm is a deterministic algorithm that, on input the secret key sk and a ciphertext c , outputs a message x . One also writes $\text{HE.Dec}_{\text{sk}}(c) \rightarrow x$.*

Correctness: *For all parameters $\lambda \in \mathbb{N}$, $n \in \mathbb{N}^*$, $(\text{sk}, \text{pk}, \text{evk}) \leftarrow_{\$} \text{HE.KGen}(1^\lambda)$, $f \in \mathcal{F}_{\text{HE}}$ and ciphertexts $c_i \leftarrow_{\$} \text{HE.Enc}_{\text{pk}}(x_i, \Delta)$, $x_i \in \mathcal{M}_{\text{HE}}$, $i \in [n]$,*

$$\mathbb{P}[\text{HE.Dec}_{\text{sk}}(\text{HE.Eval}_{\text{evk}}(f, (c_1, \dots, c_n)), \Delta) \neq f(x_1, \dots, x_n) + \delta] = \text{negl}(\lambda)$$

where δ is a small value (or noise) that depends on the scaling factor Δ . If HE is such that δ always vanishes, HE is said to be exact. Otherwise, it is called approximate.

Remark 1 (Evaluation Algorithm). *The name homomorphic encryption comes from the notion of group homomorphism, widely used in mathematics. In an HE scheme, the encryption algorithm is indeed a group homomorphism between the group $\mathcal{M}_{\text{HE}}$ and the group $\mathcal{C}_{\text{HE}}$, with their respective operations. The particularity of a homomorphism is that it preserves these operations. More precisely, an operation in $\mathcal{C}_{\text{HE}}$ is associated to an operation in $\mathcal{M}_{\text{HE}}$. This is what the evaluation algorithm is: $\text{HE.Eval}_{\text{evk}}(f, \cdot)$ is the corresponding function operating on $\mathcal{C}_{\text{HE}}^n$ to a function $f(\cdot)$ that operates on $\mathcal{M}_{\text{HE}}^n$. Informally, one could write $\text{HE.Eval}_{\text{evk}}(f, (c_1, \dots, c_n)) = \text{HE.Enc}_{\text{pk}}(f(x_1, \dots, x_n))$. As the function $f : \mathcal{M}_{\text{HE}}^n \rightarrow \mathcal{M}_{\text{HE}}$ returns a single plaintext $x = f(x_1, \dots, x_n)$, the corresponding function returns a single ciphertext $c = \text{HE.Enc}_{\text{pk}}(f(x_1, \dots, x_n))$ that can be decrypted using the decryption algorithm $\text{HE.Dec}_{\text{sk}}(c)$ to retrieve $x = f(x_1, \dots, x_n)$.*

Most commonly used HE schemes are exact, including BGV [15], B/FV [14, 27] and TFHE [21] for example, and cover most application cases using integer data. However, to perform homomorphic operations on *real* data, or more accurately floating-point arithmetic, it is useful to consider approximate HE. The first and most popular example of such scheme is CKKS [20]. In theory, the only difference is the addition of a small noise in the final result. This noise is implicitly introduced in the encryption algorithm as a *scaling factor*, which determines the number of digits, or *most significant bits*, contained in the results.

(Noisy) Functional Encryption: Now that we have defined the two classes of HE schemes that are relevant in the scope of our chapter, we move on with FE, the second encryption technique on which our construction relies. Similarly, we provide a comprehensive definition that includes the novel notion of *noisy* functional encryption [60].

Definition 8 (Functional Encryption). *A functional encryption scheme FE for message space $\mathcal{M}_{FE}$, ciphertext space $\mathcal{C}_{FE}$ and class of functions $\mathcal{F}_{FE}$, consists of four PT algorithms FE.Setup, FE.KDer, FE.Enc, FE.Dec defined as follows:*

- FE.Setup (1^λ). *The setup algorithm is a probabilistic algorithm. On the input of a security parameter λ it outputs a (master) secret key msk and a (master) public key mpk;*
- FE.Enc (mpk, x). *The encryption algorithm is a probabilistic algorithm. On the input of public key mpk and a message x in the message space $\mathcal{M}_{FE}$, it outputs a ciphertext c in the ciphertext space $\mathcal{C}_{FE}$. One also writes $c \leftarrow \$ \text{FE.Enc}_{\text{mpk}}(x)$;*
- FE.KDer (msk, f, χ). *The key derivation algorithm is a deterministic algorithm. On the input of the secret key msk, a function $f \in \mathcal{F}_{FE}$ and a probability distribution χ , it outputs a decryption key dk_f . One also writes $\text{FE.KDer}_{\text{msk}}(f, \chi) \rightarrow \text{dk}_f$;*
- FE.Dec (dk_f, c). *The decryption algorithm is a deterministic algorithm that, on the input of a decryption key dk_f and a ciphertext c , outputs a result y . One also writes $\text{FE.Dec}(\text{dk}_f, c) \rightarrow y$.*

Correctness: *For all parameters $\lambda \in \mathbb{N}$, $(\text{msk}, \text{mpk}) \leftarrow \$ \text{FE.Setup}(1^\lambda)$, $f \in \mathcal{F}_{FE}$, $\text{dk}_f \leftarrow \$ \text{FE.KDer}_{\text{msk}}(f, \chi)$ and $x \in \mathcal{M}_{FE}$,*

$$\mathbb{P} [\text{FE.Dec}_{\text{dk}_f}(\text{FE.Enc}_{\text{mpk}}(x)) \neq f(x) + \delta] = \text{negl}(\lambda)$$

where $\text{Im}(f) \ni \delta \leftarrow \chi$ is a (small) noise sampled during the key derivation FE.KDer. If χ vanishes everywhere, FE is called exact. Otherwise, FE is said to be noisy, and we write nFE. In this latter case, the key derivation algorithm is probabilistic, and we write $\text{dk}_f \leftarrow \$ \text{FE.KDer}_{\text{msk}}(f, \chi)$.

Remark 2 (Functional Decryption Algorithm). *Unlike traditional encryption schemes, the decryption algorithm does not necessarily output an element of the plaintext space, but the result of a function that takes a value into an arbitrary space. This is one of the advantages of FE over HE, since the latter exclusively supports the function defined on the message space.*

5.5 Homomorphic Evaluation of FE

This section describes how one can homomorphically evaluate the functional circuits to build a correct transciphering framework. In particular, it indicates the condition under which to design such a scheme.

5.5.1 Evaluation of FE Functions

This section studies the conditions for homomorphic evaluation of the FE key generation and decryption circuits. We consider an arbitrary FE scheme as defined in Definition 8. We first describe how to homomorphically evaluate the functional key derivation algorithm with encrypted functionality and master secret key. This theorem is fundamental to the establishment of a secure and function-blind transciphering encryption scheme in Sect. 5.6.

Theorem 3 (Homomorphic Functional Key Derivation). *Let HE be an HE scheme and FE be an FE scheme. Let $\text{FE.KDer} : (\text{msk}, f) \mapsto \text{FE.KDer}(\text{msk}, f)$ be the 2-ary key derivation function. We make the following assumptions:*

- *For any parameter $\lambda \in \mathbb{N}$ and pair of keys $(\text{msk}, \text{mpk}) \leftarrow \$ \text{FE.Setup}(1^\lambda)$, msk lies in the message space $\mathcal{M}_{HE}$ of HE;*

- FE.KDer is in the class of functions $\mathcal{F}_{\text{HE}}$;
- There exists a representation of f such that f is in the plaintext space $\mathcal{M}_{\text{HE}}$.

Under these assumptions, the key derivation algorithm can be run homomorphically, that is for all parameters $(\lambda_1, \lambda_2) \in \mathbb{N}^2$, $(\text{sk}, \text{pk}, \text{evk}) \leftarrow \$ \text{HE.KGen}(1^{\lambda_1})$, $(\text{msk}, \text{mpk}) \leftarrow \$ \text{FE.Setup}(1^{\lambda_2})$, $c_{\text{msk}} := \text{HE.Enc}_{\text{pk}}(\text{msk})$, and $f \in \mathcal{F}_{\text{FE}}$:

$$\text{HE.Eval}_{\text{evk}}(\text{FE.KDer}, c_{\text{msk}}, \text{HE.Enc}_{\text{pk}}(f)) \in \text{Im}(\text{HE.Enc}_{\text{pk}}(\text{dk}_f)) \quad (5.1)$$

with overwhelming probability in λ_1, λ_2 .

Proof. Let $(\lambda_1, \lambda_2) \in \mathbb{N}^2$ be arbitrary security parameters, for HE and FE respectively. Let $f \in \mathcal{F}_{\text{FE}}$ be an arbitrary function in the functionality space of FE. Run $(\text{sk}, \text{pk}, \text{evk}) \leftarrow \$ \text{HE.Setup}(1^{\lambda_1})$ and $(\text{msk}, \text{mpk}) \leftarrow \$ \text{FE.Setup}(1^{\lambda_2})$. For the sake of brevity and readability, we use the following notations: $c_{\text{msk}} := \text{HE.Enc}_{\text{pk}}(\text{msk})$, $c_f := \text{HE.Enc}_{\text{pk}}(f)$, and $g := \text{FE.KDer}$. The first member of Equation (5.1) becomes $\text{HE.Eval}_{\text{evk}}(g, c_{\text{msk}})$, where $g \in \mathcal{F}_{\text{HE}}$ by assumption. It follows:

$$\text{HE.Dec}_{\text{sk}}(\text{HE.Eval}_{\text{evk}}(g, c_{\text{msk}}, c_f)) = g(\text{HE.Dec}_{\text{sk}}(c_{\text{msk}}), \text{HE.Dec}_{\text{sk}}(c_f))$$

(with overwhelming probability) because HE is correct. Hence,

$$\text{HE.Dec}_{\text{sk}}(\text{HE.Eval}_{\text{evk}}(g, c_{\text{msk}}, c_f)) = \text{FE.KDer}(\text{msk}, f) = \text{dk}_f$$

In other words, $\text{HE.Eval}_{\text{evk}}(g, c_{\text{msk}})$ is a valid encryption of dk_f , that is

$$\text{HE.Eval}_{\text{evk}}(g, c_{\text{msk}}, c_f) \in \text{Im}(\text{HE.Enc}_{\text{pk}}(\text{dk}_f))$$

with overwhelming probability in λ_1, λ_2 . □

The main goal of this chapter is to address the privacy leakages that the generation of the decryption key causes. In section 5.7, we utilize theorem 3 to build a protocol that both narrows the trust in the key curator and achieves function-blindness. However, the possibilities offered by HE are not limited to evaluating the key derivation algorithm, and our result can be extended to the decryption algorithm, as shown below.

Theorem 4 (Homomorphic Functional Decryption). *Let HE be an HE scheme and let $(\lambda_1, \lambda_2) \in \mathbb{N}^2$ be their respective security parameters. Let $\text{FE.Dec}(\cdot, c) : \text{dk}_f \mapsto \text{FE.Dec}(\text{dk}_f, c)$ be the univariate function associated to FE.Dec obtained by fixing c . We make the following assumptions:*

- For any parameter $\lambda \in \mathbb{N}$, pair of keys $(\text{msk}, \text{mpk}) \leftarrow \text{FE.Setup}(1^{\lambda_2})$ and function $f \in \mathcal{F}_{\text{FE}}$, $\text{FE.KDer}_{\text{msk}}(f) \rightarrow \text{dk}_f$ lies in the message space $\mathcal{M}_{\text{HE}}$ of HE;
- For any ciphertext $c \leftarrow \$ \text{FE.Enc}_{\text{mpk}}(x)$, $x \in \mathcal{M}_{\text{FE}}$, $\text{FE.Dec}(\cdot, c)$ is in the class of functions $\mathcal{F}_{\text{HE}}$.

Under these assumptions, the functional decryption algorithm can be run homomorphically, that is, for any tuple of keys $(\text{sk}, \text{pk}, \text{evk}) \leftarrow \$ \text{HE.KGen}(1^{\lambda_1})$ and encryption $c \leftarrow \$ \text{FE.Enc}_{\text{mpk}}(x)$,

$$\mathbb{P}[\text{HE.Eval}_{\text{evk}}(\text{FE.Dec}(\cdot, c), \text{HE.Enc}_{\text{pk}}(\text{dk}_f)) \notin \text{Im}(\text{HE.Enc}_{\text{pk}}(f(x)))] \quad (5.2)$$

is negligible in λ_1 and λ_2 .

Proof. Let $(\lambda_1, \lambda_2) \in \mathbb{N}^2$ be arbitrary security parameters, for HE and FE respectively. Let $f \in \mathcal{F}_{\text{FE}}$ be an arbitrary function and $m \in \mathcal{M}_{\text{FE}}$ a random message. Run $(\text{sk}, \text{pk}, \text{evk}) \leftarrow_{\$} \text{HE.KGen}(1^{\lambda_1})$, $(\text{msk}, \text{mpk}) \leftarrow_{\$} \text{FE.Setup}(1^{\lambda_2})$ and $c \leftarrow_{\$} \text{FE.Enc}_{\text{mpk}}(x)$. For the sake of brevity and readability, we use the following notations: $c_{\text{dk}_f} := \text{HE.Enc}_{\text{pk}}(\text{dk}_f)$ and $g := \text{FE.Dec}(\cdot, c)$. The first member of Equation (5.2) becomes $\text{HE.Dec}_{\text{sk}}(\text{HE.Eval}_{\text{evk}}(g, c_{\text{dk}_f}))$, where $g \in \mathcal{F}_{\text{HE}}$ by assumption. The homomorphic properties of HE guarantee the following equality:

$$p_1 := \mathbb{P}[\text{HE.Dec}_{\text{sk}}(\text{HE.Eval}_{\text{evk}}(g, c_{\text{dk}_f})) \neq g(\text{HE.Dec}_{\text{sk}}(c_{\text{dk}_f}))] = \text{negl}(\lambda_1) \quad (5.3)$$

by correctness of HE. Beside,

$$p_2 := \mathbb{P}[g(\text{HE.Dec}_{\text{sk}}(c_{\text{dk}_f})) \neq g(\text{dk}_f)] = \text{negl}(\lambda_1) \quad (5.4)$$

because g is deterministic and HE is correct in the sense of PKE. Finally, as $g(\text{dk}_f) = \text{FE.Dec}(\cdot, c)(\text{dk}_f) = \text{FE.Dec}(\text{dk}_f, c)$ by definition, we have:

$$p_3 := \mathbb{P}[g(\text{dk}_f) \neq f(x)] = \text{negl}(\lambda_2) \quad (5.5)$$

because FE is correct. Denote p_0 the probability given in Equation (5.2). Then $p_0 = p_1 + p_2 + p_3 = 2 \cdot \text{negl}(\lambda_1) + \text{negl}(\lambda_2) = \text{negl}(\lambda_1, \lambda_2)$. $\square$

5.5.2 Representation of Functions

Theorem 3 and theorem 4 are valid in theory, but when it comes to practice, the condition $f \in \mathcal{M}_{\text{HE}}$ stated in theorem 3 can be hard to capture.

One of the most popular and notable classes of FE schemes is *inner product functional encryption* (IPFE) [22, 42, 1, 23, 2, 17], which allows the evaluation of the inner product over encrypted data. In this scenario, the function $f(\mathbf{x}) = \langle \mathbf{x}, \mathbf{y} \rangle$ can be represented by the vector $\mathbf{y}$, which is sent by the analyst to the curator. In this case, it is easy to choose $\mathbf{y}$ and HE such that $\mathbf{y} \in \mathcal{M}_{\text{HE}}$.

In the case of Quadratic Functional Encryption (QFE) [29, 11, 55], the function is a quadratic polynomial $P = \sum a_{ij} X^i Y^j$. As for IPFE, this function is entirely determined by a finite set of coefficients $\{a_{ij}\}$ that can be represented as a vector in $\mathcal{M}_{\text{HE}}$. Finally, we mention the case of SPADE [43], an FE scheme in which the function is an equality test for a queried value v . This particular FE function is entirely determined by v and therefore has a straightforward representation as an element of $\mathcal{M}_{\text{HE}}$. In the general case, we define a function as follows:

Definition 9 (Representation of a function). *Let $f : \mathcal{X} \rightarrow \mathcal{Y}$ be an arbitrary function for sets $\mathcal{X}$ and $\mathcal{Y}$. Then f is entirely determined by the image $f(x) \in f(\mathcal{X}) \subseteq \mathcal{Y}$ of each element $x \in \mathcal{X}$. In practice, $f := \{(x, f(x)) \in \mathcal{X} \times f(\mathcal{X})\}$. Moreover, if f is linear and $\mathcal{X}$ is finitely generated, $f := \{(e_i, f(e_i))\}$, for all generators e_i of $\mathcal{X}$.*

In the context of FE, $f := ((x_1, f(x_1)), \dots, (x_n, f(x_n)))$, where $x_i \in \mathcal{M}$ is an element of the FE message space and $f(x_i)$ its evaluation by $f \in f(\mathcal{M})$. With a proper encoding, each pair can be represented as a plaintext of HE hence allowing the practical encryption of f .

Example: Let $\mathcal{X} := \mathbb{Z}^n$, $\mathcal{Y} := \mathbb{Z}$ and $f_{\mathbf{y}} : \mathbb{Z}^n \rightarrow \mathbb{Z}; \mathbf{x} \mapsto \langle \mathbf{x}, \mathbf{y} \rangle$, for $\mathbf{y} \in \mathbb{Z}^n$. $f_{\mathbf{y}}$ is naturally determined by the set $\{(\mathbf{x}, \langle \mathbf{x}, \mathbf{y} \rangle)\}_{\mathbf{x} \in \mathbb{Z}^n}$. However, since $f_{\mathbf{y}}$ is a group morphism and $\mathbb{Z}^n$ is an additive group generated by the canonical basis $(e_1, \dots, e_n)$, it is sufficient to consider the representation $\{(e_i, \langle e_i, \mathbf{y} \rangle)\}_{i \in n}$, that is $\{(e_i, y_i)\}$, that is $\mathbf{y} = (y_1, \dots, y_n)$, which is exactly what we do in practice.

5.6 Homomorphic Functional Encryption (HFE)

This section provides a formal definition of the encryption scheme obtained by evaluating the FE algorithms homomorphically. The semantic security of our construction is presented in Sect. 5.8.

Definition 10 (Homomorphic Transciphering Functional Encryption Scheme). *Let $HE := (KGen, Enc, Eval, Dec)$ be an exact homomorphic encryption scheme and $FE := (Setup, Enc, KDer, Dec)$ be a non-noisy functional encryption scheme for message space $\mathcal{M}$ and functionality space $\mathcal{F}$, such that theorem 3 is applicable. An HFE scheme HFE for message space $\mathcal{M}$ and class of functions $\mathcal{F}$ consists of five PT algorithms HFE.Setup, HFE.Encap, HFE.Enc, HFE.KDer, HFE.Dec defined as follows:*

- $HFE.Setup(1^\lambda)$. *The setup algorithm is a probabilistic algorithm. On input it takes a security parameter λ ; computes $(sk, pk, evk) \leftarrow \$ HE.KGen(1^\lambda)$ and outputs the secret key sk , the public key pk and the evaluation key evk ;*
- $HFE.Encap(1^\mu, pk)$. *The encapsulation algorithm is a probabilistic algorithm that takes on input a security parameter μ and the public key pk . It generates the master secret and public keys $(msk, mpk) \leftarrow \$ FE.Setup(1^\mu)$, encrypts $c_{msk} \leftarrow \$ HE.Enc_{pk}(msk)$ and outputs the master public key mpk and the homomorphically encrypted master secret key c_{msk} ;*
- $HFE.Enc(mpk, x)$. *The encryption algorithm is a probabilistic algorithm. On the input of public key mpk and a message x in the message space $\mathcal{M}$, outputs a ciphertext $c \leftarrow \$ FE.Enc_{mpk}(x)$;*
- $HFE.KDer(c_{msk}, evk, f)$. *The key derivation algorithm is a deterministic algorithm. On input it takes the homomorphically encrypted key c_{msk} , the evaluation key evk and a function $f \in \mathcal{F}$. It encrypts homomorphically the function f into $c_f \leftarrow \$ HE.Enc_{pk}(f)$ and outputs a homomorphically encrypted decryption key $HE.Eval_{evk}(FE.KDer, (c_{msk}, c_f)) \rightarrow c_{dk_f}$;*
- $HFE.Dec(sk, c_{dk_f}, c)$. *The decryption algorithm is a deterministic algorithm. On input it takes the secret homomorphic key sk , a homomorphically encrypted decryption key c_{dk_f} and a ciphertext c ; computes $HE.Dec_{sk}(c_{dk_f}) \rightarrow dk_f$ and outputs a result $FE.Dec(dk_f, c) \rightarrow res$.*

Correctness: *For all parameters $\lambda, \mu \in \mathbb{N}$, $f \in \mathcal{F}$, $(sk, pk, evk) \leftarrow \$ HFE.Setup(1^\lambda)$, $(mpk, c_{msk}) \leftarrow \$ HFE.Encap(1^\mu, pk)$, $HFE.KDer(c_{msk}, evk, f) \rightarrow c_{dk_f}$ and $x \in \mathcal{M}$,*

$$\mathbb{P}[HFE.Dec(sk, c_{dk_f}, HFE.Enc(mpk, x)) \neq f(x)] = \text{negl}(\lambda, \mu)$$

Remark 3 (Correctness). *HFE is correct by construction: Let HFE be a HFE scheme constructed after an HE scheme HE and an FE scheme FE. Let $\lambda, \mu \in \mathbb{N}$, $(sk, pk, evk) \leftarrow \$ HFE.Setup(1^\lambda)$, $(mpk, c_{msk}) \leftarrow \$ HFE.Encap(1^\mu, pk)$, $f \in \mathcal{F}$, $HFE.KDer(c_{msk}, evk, f) \rightarrow c_{dk_f}$ and $x \in \mathcal{M}$, $c \leftarrow \$ HFE.Enc(mpk, x)$,*

$$\begin{aligned} \mathbb{P}[HFE.Dec(sk, c_{dk_f}, c) \neq f(x)] &= \mathbb{P}[FE.Dec(HE.Dec(sk, c_{dk_f}), c) \neq f(x)] \\ &\leq \mathbb{P}[(FE.Dec(dk_f, c) \neq f(x)) \cup (HE.Dec(sk, c_{dk_f}) \neq dk_f)] \\ &\leq \mathbb{P}[FE.Dec(dk_f, c) \neq f(x)] + \mathbb{P}[HE.Dec(sk, c_{dk_f}) \neq dk_f] \\ &\leq \text{negl}(\lambda) + \text{negl}(\mu) = \text{negl}(\lambda, \mu) \end{aligned}$$

5.6.1 Function Blindness

This section does not aim at providing a formal security proof, but rather to give a grasp of how HFE achieves the security level we claim. Its main asset being the *obfuscation* of FE circuits using HE, we give an intuition of how HFE ensures function blindness.

Definition 11 (Function Blindness). *An FE scheme is said to be function-blind if no PPT adversary, given two ciphertexts $c_1 \in \text{Im}(\text{Enc}(\text{pk}, x_1))$ and $c_2 \in \text{Im}(\text{Enc}(\text{pk}, x_2))$, decryption key requests req_{f_1} , req_{f_2} and a key generation algorithm $\text{KDer} : \text{req}_f \mapsto \text{dk}_f$, can't distinguish between req_{f_1} and req_{f_2} or c_1 and c_2 with non-negligible probability.*

We propose a systematic method to achieve function blindness. This technique relies on the homomorphic encryption of f , prior to derivation. We present algorithm 1, which involves two parties: **U** (user) and **KC** (key curator). In this scenario, **U** sends a request to **KC** to obtain a decryption key for a function f , while ensuring that f remains hidden from **KC**.

Algorithm 1 Function-Blind Key Derivation

Input: HE keys $(\text{sk}, \text{pk}, \text{evk})$, the master secret key msk and a function $f \in \mathcal{F}$

Output: the decryption key dk_f

- 1: **U** encrypts $c_f \leftarrow \text{HE.Enc}_{\text{pk}}(f)$ and sends it to **KC**
 - 2: **KC** encrypts $c_{\text{msk}} \leftarrow \text{HE.Enc}_{\text{pk}}(\text{msk})$
 - 3: **KC** runs $c \leftarrow \text{HE.Eval}_{\text{evk}}(\text{FE.KDer}(c_{\text{msk}}, \cdot), c_f)$ and sends it to **U**
 - 4: **U** decrypts $\text{dk}_f \leftarrow \text{HE.Dec}_{\text{sk}}(c)$
-

Correctness: The output dk_f of algorithm 1 is indeed the decryption key for function f .

Proof. This is straightforward as it relies on the correctness of the underlying HE and FE schemes. Therefore, we provide a short and intuitive proof.

$$\begin{aligned}
 \text{HE.Dec}_{\text{sk}}(c) &= \text{HE.Dec}_{\text{sk}}(\text{HE.Eval}_{\text{evk}}(\text{FE.KDer}_{\text{msk}}(\cdot), c_f)) \\
 &= \text{FE.KDer}_{\text{msk}}(\text{HE.Dec}_{\text{sk}}(c_f)) \\
 &= \text{FE.KDer}_{\text{msk}}(\text{HE.Dec}_{\text{sk}}(\text{HE.Enc}_{\text{pk}}(f))) \\
 &= \text{FE.KDer}_{\text{msk}}(f) = \text{dk}_f
 \end{aligned}$$

□

Function-blindness: Assuming that **KC** knows the functional secret key msk , a homomorphic encryption of the functional secret key c_{msk} and the public homomorphic keys (pk, evk) , algorithm 1 is function-blind. In other words, **KC** learns only negligible information about f during the key generation.

Intuition. We assume that **KC** has access to (pk, evk) and c_{msk} , and first assume that **KC** is a passive eavesdropper. The HE scheme HE is IND-HE-CPA secure, as defined in Definition 13. This means that without the knowledge of sk , **KC** can not distinguish between c_f and a random element of $\mathcal{C}$ with a non-negligible advantage. Moreover, for any function (or circuit) g supported by HE, the homomorphic evaluation of g on a finite number of ciphertext gives a negligible advantage to **KC** to distinguish the said ciphertexts. Since FE.KDer is, as stated in subsection 5.5.1, a function supported by HE, the advantage of **KC** as a passive eavesdropper is negligible.

However, the problem with function-blindness in a public key setup is that **KC** can evaluate the function f in finitely many points, hence gathering non-negligible information on f . If we consider **KC** as an active eavesdropper, we need to guarantee that it can not conduct this type of attack with c_f only. We assume that **KC** can adaptively choose a finite number of elements $x_1, \dots, x_n$ in the plaintext space of FE and get their encryption $\mu_1, \dots, \mu_n$ under mpk . Following, **KC** can re-encrypt these ciphertexts into homomorphic ciphertexts, that is $c_i \leftarrow \text{HE.Enc}_{\text{pk}}(\mu_i)$, and run the functional decryption algorithm,

as described in subsection 5.5.1. More formally, given c a homomorphic encryption of dk_f and c_i a homomorphic ciphertext for which it knows the underlying plaintext, **KC** can compute:

$$\text{HE.Eval}_{\text{evk}}(\text{FE.Dec}(c, c_i)) = c_{f(x_i)} \in \text{Im}(\text{HE.Enc}(f(x_i))) \quad (5.6)$$

by correctness of HFE. The major result here is that **KC** performs only computations on homomorphic ciphertexts. The attack aforementioned only returns an encryption of the result $f(x_i)$ and therefore leaks no information about the function f as HE is IND-HE-CPA. $\square$

To conclude, the HFE construction defined in 10 is function-blind.

5.7 Real-World Deployment Scenario for HFE

In this section, we present a three-party toy protocol, demonstrating how our HFE scheme can be utilized in real-world applications. To map this general protocol to the PiQASO project infrastructure and use-cases, the abstract roles are instantiated as follows: the Key Curator (**KC**) represents the cloud-based **PQC-as-a-Service (PQCaaS)** module, the User (**U**) corresponds to the resource-constrained **edge devices** (e.g., patient monitoring sensors), and the Data Analyst (**DA**) represents the data-consuming entity, such as the **hospital** or medical analytics platform. To preserve alignment with the underlying mathematical security models, the abstract notation and labels (**KC**, **U**, **DA**) are kept throughout this section. Consider a scenario in which the data analyst **DA** (hospital) wants to perform selective computations on sensitive data x held by the user **U** (edge device). To do so, they rely on a third-party **KC** (PQCaaS module), that should not learn either x or the computations performed by **DA**.

Overview of the protocol: **DA** generates the homomorphic keys (sk, pk, evk) on one side, and the data owner generates the functional keys (msk, mpk) $\leftarrow_{\$} \text{FE.Setup}(1^\lambda)$ on the other side. The latter encrypts msk into a ciphertext c_{msk} with the analyst's homomorphic key pk and encrypts x into a ciphertext c under his functional public key mpk . The analyst, who has access to c , wants to calculate $f(x)$. She first encrypts f under pk into c_f and sends a request to **KC** to obtain (an encryption of) the functional decryption key c_{dk_f} . The curator, given the encrypted function c_f , the encrypted master secret key c_{msk} , and the evaluation key evk , runs homomorphically the functional key generation algorithm to output the encrypted decryption key c_{dk_f} . It then sends it back to the analyst who uses the homomorphic secret key sk to retrieve dk_f and finally $f(x)$.

System model: The model we consider consists of three entities: (i) Key Curator (**KC**), (ii) User (**U**) and (iii) Data Analyst (**DA**). In the context of the PiQASO infrastructure, these abstract entities are mapped to the PQCaaS module, edge devices, and the hospital, respectively.

- **Key Curator (KC / PQCaaS module):** The curator is an authority responsible for the generation of decryption keys. Upon request of **DA** for a function f , **KC** is responsible for the generation of the corresponding decryption key dk_f ;
- **User (U / Edge device):** The user holds the plain data x (e.g., patient vital signs). He sets the system up by generating the functional parameters (msk, mpk) $\leftarrow_{\$} \text{FE.Setup}(1^\lambda)$ and sending $c_{msk} \leftarrow_{\$} \text{HE.Enc}_{pk}(msk)$ to **KC**, where pk is the public key of **DA** for the HE scheme.
- **Data Analyst (DA / Hospital):** The data analyst is an external party who wishes to learn the evaluation of a certain function f over data x provided by the user. With exclusive access to this as a ciphertext, **DA** sends a request for f to **KC**, which provides the corresponding decryption key dk_f , through which the result of the computation $f(x)$ can be obtained.

Threat model: Concerning the *semantic* security (IND-CPA) and function-blindness, we rely on the analyses we provide in other sections, namely theorem 5 and Sect. 5.6.1 respectively. Concerning the

security of the protocol, we trust the user **U** with the information he sends, and assume that he provides genuine data, performs the encapsulation using the right homomorphic key pk , and encrypts the data using the public key mpk . We do *not* require the key curator to be fully trusted. However, we assume that **KC** and **DA** *do not collude*, as the knowledge of both c_{msk} and sk is sufficient to retrieve msk and hence decrypt every ciphertext produced by **U**. This last assumption is mandatory in most FE construction relying on a curator, trusted or not. Indeed, the curator can generate a decryption key, and the data analyst can obtain valuable information from it. When implemented in practice, one assumes that **KC** will generate a reasonable amount of decryption keys – usually one. However, if it provides a larger number of decryption keys, **DA** can bypass the limited information she is originally supposed to get.

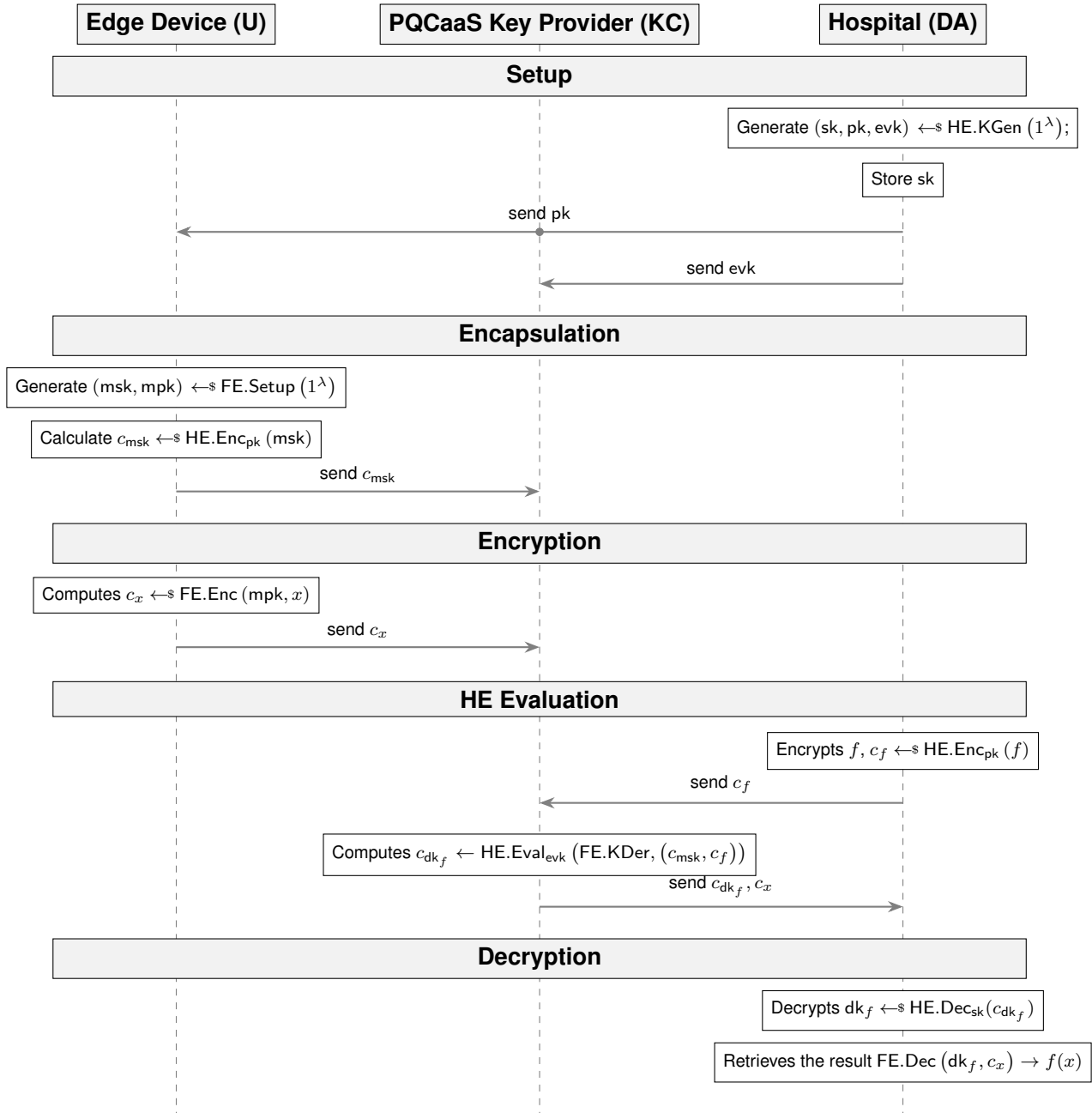


Figure 5.1: HFE: Protocol Flow Example.

In HFE protocol, we consider two arbitrary schemes: HE and FE for a space of the function $\mathcal{F}$. To give a more global impact to our chapter, we do not specify any scheme in particular, and assume that $FE.KDer$ is a valid input of $HE.Eval$, and that we can express msk and $f \in \mathcal{F}$ as elements of the plaintext space of

HE.

As shown in Fig. 5.1, the setup consists of providing **KC** with all the information it requires to run the subsequent key derivation algorithm. On one side, **DA** generates the homomorphic parameters sk, pk , and evk . She stores the secret key sk for further decryption, sends pk to **U** and evk to **KC**. On the other side, the user generates the functional parameters msk, mpk . He encrypts homomorphically the master secret key into $c_{msk} \leftarrow^s HE.Enc_{pk}(msk)$ and sends it to **KC**.

After the system is set up, **U** can encrypt his data x . Using the public key mpk previously generated, he produces an FE ciphertext c_x and sends it to an external entity that can store it and forward it to **DA**. For the sake of simplicity, we consider that **KC** itself stores the ciphertexts, as it does not impact the practicability of our protocol.

The main component of HFE is the generation of the decryption key. Having an encryption c_{msk} of the master secret key and an encryption c_f of the function f , **KC** is capable of generating an encryption of the functional decryption key. To do that, it runs homomorphically (through the Eval algorithm) the decryption key generation, that is $HE.Eval_{evk}(FE.KDer, (c_{msk}, c_f))$. As explained in remark 2, this algorithm outputs a ciphertext c that lies in the image $Im(HE.Enc_{pk}(dk_f))$, that is the subset of $\mathcal{C}$ of the correct homomorphic encryptions of dk_f .

Using the secret key sk , **DA** can successfully retrieve dk_f . Following, **DA** runs the functional decryption algorithm as usual on the ciphertext c_x and finally gets the desired result $f(x)$.

Bound on the decryption key generation: As mentioned in the threat model, the security of FE always relies on a bound on the number of generated decryption keys. Regardless of the level of trust in it, **KC** has to keep records of the decryption keys it provides to **DA** and, in our model, of the decryption keys she could obtain with the evaluation algorithm. The *good practices* for the key generation can not be systematized, as the leakage produced by a decryption key depends on the FE scheme used.

5.8 Semantic Security

This section provides a detailed and formal definition of the semantic security required for an HFE scheme. In addition, we formally demonstrate the semantic security of HFE with respect to our definition. This definition relies on the traditional notion of IND-CPA security for HE and FE, since it is the most commonly used paradigm. We provide theorem 5, which ensures the theoretical security of HFE, given that HE and FE are IND-CPA secure.

5.8.1 Indistinguishability-based Security

While the security of PKE is well-studied, it remains challenging to define a universal security definition for HE and FE. The reason is two-fold: firstly, the capacity to make computations on the ciphertexts clearly impacts the notion of *indistinguishability*. This is particularly true in FE, where the decryption algorithm acts as a *trapdoor*, hence challenging traditional security definitions. Secondly, the application case and the considered system model significantly impact the security expectations. If these concerns are usually delegated to the security of the protocol, the large panel of use-cases offered by both HE and FE requires the user to ask what oracle can reasonably be provided to an attacker, even at the scheme's semantic security level. A good example of this problem is the recent introduction by Li and Micciancio [35] of $IND-CPA^D$. In short, it consists of providing the attacker conditional access to the decryption oracle. This seems like a reasonable assumption, since HE can provide the decryption of particular results to a not-fully-trusted entity. However, there is yet no consensus on this level of security [19, 18], which is highly influenced by the considered use case.

FE Indistinguishability under Chosen Plaintext Attack: This security notion is usually formalized through a security game. The game starts with a challenger flipping a truly random coin $\beta \in \{0, 1\}$, generating the master secret and public keys, and sending the master public key to the adversary $\mathcal{A}$. The latter subsequently requests functional decryption keys for functions of its choice. At some point, $\mathcal{A}$ outputs two messages x_0, x_1 to the challenger, which replies with c_β . The scheme is IND-FE-CPA secure if $\mathcal{A}$ cannot guess the value of β with probability greater than $1/2 + \text{negl}$. More formally, the security game is described below:

Definition 12 (IND-CPA Security for FE (IND-FE-CPA)). *We say that an FE scheme FE is IND-FE-CPA secure, if for all PPT adversary $\mathcal{A}$, it holds that the advantage of $\mathcal{A}$:*

$$\text{Adv}_{\text{FE}, \mathcal{A}}^{\text{ind-cpa}}(1^\lambda) = \left| \Pr \left[\text{Exp}_{\text{FE}, \mathcal{A}}^{\text{ind-cpa}}(1^\lambda) \rightarrow \text{true} \right] - \frac{1}{2} \right|$$

is negligible in λ , where the experiment is defined as follows in Figure 5.2.

$\text{Exp}_{\text{FE}, \mathcal{A}}^{\text{ind-cpa}}(1^\lambda) :$
 $\beta \leftarrow \{0, 1\}; L_K, L_M \leftarrow \emptyset$
 $(\text{msk}, \text{mpk}) \leftarrow \text{Setup}(1^\lambda)$
 $\beta' \leftarrow \mathcal{A}(1^\lambda, \text{mpk} : \mathcal{O}_{\text{ind-FE}})$
Return $(\beta = \beta')$

Oracles $\mathcal{O}_{\text{ind-FE}}$

ENCRYPT (mpk, x_0, x_1)

If $x_0, x_1 \notin \mathcal{M}$:

return $\perp$

If $\exists (f_0, f_1) \in L_F ; f_0(x_0) \neq f_1(x_1)$:

return $\perp$

$c_\beta \leftarrow \text{Enc}(\text{mpk}, x_\beta)$

$L_M \leftarrow L_M \cup \{(x_0, x_1, c_\beta)\}$

return c_β

KEYDER (f_0, f_1)

If $f_0, f_1 \notin \mathcal{F}$:

return $\perp$

If $\exists (x_0, x_1) \in L_M ; f_0(x_0) \neq f_1(x_1)$:

return $\perp$

$\text{dk}_{f_\beta} \leftarrow \text{KDer}(\text{msk}, f_\beta, D)$

$L_F \leftarrow L_F \cup \{(f_0, f_1, \text{dk}_{f_\beta})\}$

return dk_{f_β}

Figure 5.2: Oracles in the IND-FE-CPA game for an FE scheme FE and PPT adversary $\mathcal{A}$: the encryption oracle ENCRYPT and the key derivation oracle KEYDER.

The game records the queries made by $\mathcal{A}$ through a list L_M storing encryption queries and a list L_F storing functional queries. Each request is made under the condition that $f_0(x_0) = f_1(x_1)$, for any $(f_0, f_1) \in L_F$ and $(x_0, x_1) \in L_M$. This condition is mandatory, as violating it gives $\mathcal{A}$ an overwhelming advantage to distinguish $f_0(x_0)$ from $f_1(x_1)$.

HE Indistinguishability under Chosen Plaintext Attack: The security notion of IND-HE-CPA security is usually formalized through a security game. The game starts with a challenger $\mathcal{C}$ flipping a truly random coin $\beta \in \{0, 1\}$, generating the public, secret and evaluation keys, and sending the public key to the adversary $\mathcal{A}$. The latter has access to an encryption oracle, that on the input of two messages x_0, x_1 outputs the ciphertext c_β . Subsequently, $\mathcal{A}$ requests evaluations for functions of its choice. At some point, $\mathcal{A}$ outputs a guess β' on the value of β . The scheme is IND-HE-CPA secure if $\mathcal{A}$ cannot guess the value of β with probability greater than $1/2 + \text{negl}$. More formally, the security game is described below:

Definition 13 (IND-CPA Security for HE (IND-HE-CPA)). *Let $\text{HE} = (\text{KGen}, \text{Enc}, \text{Eval}, \text{Dec})$ be an HE scheme for message space $\mathcal{M}$ and functionality space $\mathcal{F} : \mathcal{M}^n \rightarrow \mathcal{M}$. We say that HE is IND-HE-CPA secure if for all PPT adversary $\mathcal{A}$, it holds that the advantage of $\mathcal{A}$ given by $\text{Adv}_{\text{HE}, \mathcal{A}}^{\text{ind-cpa}}(\lambda) = \left| \Pr \left[\text{Exp}_{\text{HE}, \mathcal{A}}^{\text{ind-cpa}}(\lambda) \rightarrow \text{true} \right] - \frac{1}{2} \right|$ is negligible in λ , where the experiment is defined as follows:*

$\text{Exp}_{\text{HE}, \mathcal{A}}^{\text{ind-cpa}}(\lambda):$
 $\beta \leftarrow_{\$} \{0, 1\}; L \leftarrow \emptyset$
 $(\text{sk}, \text{pk}, \text{evk}) \leftarrow_{\$} \text{KGen}(1^\lambda)$
 $\beta' \leftarrow_{\$} \mathcal{A}(1^\lambda, \text{pk}, \text{evk}, : \mathcal{O}_{\text{ind-HE}})$
return $(\beta = \beta')$

Oracles $\mathcal{O}_{\text{ind-HE}}$

ENCRYPT $(\text{pk}, x_0, x_1):$

If $x_0, x_1 \notin \mathcal{M} : \text{return } \perp$
 $c_\beta \leftarrow_{\$} \text{Enc}(\text{pk}, x_\beta)$
 $L \leftarrow L \cup \{c_\beta\}$
return c_β

EVAL $(\text{evk}, f, (c_1, \dots, c_n)):$

If $f \notin \mathcal{F} : \text{return } \perp$
For $i \in [1, n]$
 If $c_i \notin L : \text{return } \perp$
 $c \leftarrow \text{Eval}(\text{evk}, f, (c_1, \dots, c_n))$
 $L \leftarrow L \cup \{c\}$
return c

Figure 5.3: Oracles in the IND-HE-CPA game for an HE scheme HE and PPT adversary $\mathcal{A}$: the encryption oracle ENCRYPT and the evaluation oracle EVAL.

The game records the queries made by $\mathcal{A}$ through a list L storing all the ciphertexts it received. This includes encryption of a plain message sent by $\mathcal{A}$ as well as ciphertexts resulting from the evaluation of a function f .

5.8.2 Security of HFE

Similarly to the security of FE defined in Definition 12, we formalize IND-CPA through a security game. The challenger picks randomly $\beta \in \{0, 1\}$, generates the keys, encapsulates the master secret key, and sends the public parameters to $\mathcal{A}$; that is the master public key mpk , the evaluation key evk and the encapsulation c_{msk} . The adversary subsequently requests functional decryption keys for functions of its choice. $\mathcal{A}$ eventually sends two messages x_0, x_1 to the challenger, which replies with c_β . The scheme is IND-CPA secure if $\mathcal{A}$ cannot guess the value of β with probability greater than $1/2 + \text{negl}$. More formally, the security game is described below:

Definition 14 (IND-CPA Security for HFE (IND-HFE-CPA)). *Let HFE be an HFE scheme for message space $\mathcal{M}$ and functionality space $\mathcal{F}$. We say that HFE is IND-HFE-CPA if for all PPT adversary $\mathcal{A}$, it holds that the advantage of $\mathcal{A}$:*

$$\text{Adv}_{\text{HFE}, \mathcal{A}}^{\text{ind-cpa}}(\lambda, \mu) = \left| \mathbb{P} \left[\text{Exp}_{\text{HFE}, \mathcal{A}}^{\text{ind-cpa}}(\lambda, \mu) \rightarrow \text{true} \right] - \frac{1}{2} \right|$$

is negligible in λ and μ , where the experiment is defined as follows:

$\text{Exp}_{\text{HFE}, \mathcal{A}}^{\text{ind-cpa}}(\lambda, \mu):$
 $\beta \leftarrow_{\$} \{0, 1\}; L_M \leftarrow \emptyset; L_F \leftarrow \emptyset$
 $(\text{sk}, \text{pk}, \text{evk}) \leftarrow_{\$} \text{Setup}(1^\lambda)$
 $(\text{mpk}, c_{\text{msk}}) \leftarrow_{\$} \text{Encap}(1^\mu, \text{pk})$
 $\beta' \leftarrow_{\$} \mathcal{A}(1^\lambda, 1^\mu, \text{mpk}, \text{pk}, \text{evk} : \mathcal{O}_{\text{ind-HFE}})$
return $(\beta == \beta')$

Oracles $\mathcal{O}_{\text{ind-HFE}}$

ENCRYPT (mpk, x_0, x_1)

If $x_0, x_1 \notin \mathcal{M}$: **return** $\perp$
 If $\exists (f_0, f_1, c_{\text{dk}_{f_\beta}}) \in L_F$;
 $f_0(x_0) \neq f_1(x_1)$: **return** $\perp$
 $c_\beta \leftarrow \text{Enc}(\text{mpk}, x_\beta)$
 $L_M \leftarrow L_M \cup \{(x_0, x_1, c_\beta)\}$
return c_β

EVAL (evk, $g, (c_{\text{dk}_{f_1}}, \dots, c_{\text{dk}_{f_n}})$) :

If $g \notin \mathcal{F}_{\text{HE}}$: **return** $\perp$
For $i \in [1, n]$:
 If $c_{\text{dk}_{f_i}} \notin L_F$: **return** $\perp$
 $f_0 \leftarrow g(\{f_{i0}\}_i), f_1 \leftarrow g(\{f_{i1}\}_i)$
 $c_{\text{dk}_{f_\beta}} \leftarrow \text{Eval}_{\text{evk}}(g, (c_{\text{dk}_{f_1}}, \dots, c_{\text{dk}_{f_n}}))$
 $L_F \leftarrow L_F \cup \{(f_0, f_1, c_{\text{dk}_{f_\beta}})\}$
return c

KEYDER (f_0, f_1)

If $f_0, f_1 \notin \mathcal{F}$: **return** $\perp$
 If $\exists (x_0, x_1) \in L_M ; f_0(x_0) \neq f_1(x_1)$:
return $\perp$
 $c_{\text{dk}_{f_\beta}} \leftarrow \text{KDer}(c_{\text{msk}}, f_\beta)$
 $L_F \leftarrow L_F \cup \{(f_0, f_1, c_{\text{dk}_{f_\beta}})\}$
return $c_{\text{dk}_{f_\beta}}$

DECRYPT ($c_{\text{dk}_{f_\beta}}$)

If $c_{\text{dk}_{f_\beta}} \notin L_F$: **return** $\perp$
 $\text{Dec}(\text{sk}, c_{\text{dk}_{f_\beta}}) \rightarrow \text{dk}_{f_\beta}$
return dk_{f_β}

Figure 5.4: Oracles in the IND-HFE-CPA game for a HFE scheme HFE and PPT adversary $\mathcal{A}$: The encryption oracle ENCRYPT, the key derivation oracle KEYDER, the evaluation oracle EVAL and a decryption oracle DECRYPT.

As for FE, the game keeps record of the queries made by $\mathcal{A}$ through a list L_M storing all the encryption queries and a list L_F storing each functional queries, represented as $(f_0, f_1, c_{\text{dk}_{f_\beta}})$, where f_0, f_1 are the queried functions and $c_{\text{dk}_{f_\beta}}$ is the output of the HFE key derivation algorithm. Each request is made under the condition that $f_0(x_0) = f_1(x_1)$, for any $(f_0, f_1, c_{\text{dk}_{f_\beta}}) \in L_F$ and $(x_0, x_1, c_\beta) \in L_M$. This condition is mandatory, as violating it gives $\mathcal{A}$ an overwhelming advantage to distinguish $f_0(x_0)$ from $f_1(x_1)$. Note that the EVAL algorithm works slightly differently than in a classical HE scheme. Even though the output of the evaluation for a function g is a valid homomorphic ciphertext, it is not necessarily the encryption of a valid functional decryption key for, e.g. dk_g . However, $\mathcal{A}$ can use this oracle to try to get valuable information for $\text{dk}_{f_1}, \dots, \text{dk}_{f_n}$. Finally, a decryption oracle DECRYPT outputs the functional decryption key $f(\text{dk}_{f_\beta})$ as in a traditional FE scheme.

Theorem 5. Let HFE be an HFE scheme built from an HE scheme HE and an FE scheme FE. If HE and FE are IND-HE-CPA and IND-FE-CPA secure respectively, then HFE is IND-HFE-CPA secure.

Proof. We prove theorem 5 through a sequence of games, starting with the genuine HFE game and ending with a game where the advantage of $\mathcal{A}$ is negligible. We denote ϵ_i the advantage of $\mathcal{A}$ for game i and $\epsilon(\lambda)$ (resp. $\epsilon(\mu)$) its advantage for the HE (resp. FE) game.

Game 0: This is the initial genuine HFE game defined in Definition 14. The challenger initializes the game by flipping at random a bit $\beta \leftarrow \{0, 1\}$. Then, the key generation goes as follows: on the one hand, a PPT adversary $\mathcal{A}$ samples the homomorphic keys $(\text{pk}, \text{sk}, \text{evk}) \leftarrow \text{Setup}(1^\lambda)$ and forwards pk to $\mathcal{C}$ who runs the encapsulation $(\text{msk}, c_{\text{msk}}) \leftarrow \text{Encap}(1^\mu, \text{pk})$. $\mathcal{A}$ has access to the oracles ENCRYPT, EVAL, KEYDER and EVAL. Eventually, $\mathcal{A}$ outputs a guess β' on the value of β with an advantage ϵ_0 .

Game 1: This game is similar to the previous one, except that the oracle DECRYPT is replaced by an oracle DECRYPT_{G1} defined as follows:

$$\begin{array}{l} \text{DECRYPT}_{G1}(c_{dk_{f_\beta}}) : \\ \text{If } c_{dk_{f_\beta}} \notin L_F : \text{return } \perp \\ dk \leftarrow \$ \text{Im}(\text{FE.KDer}_{\text{msk}}) \\ \text{return } dk \end{array}$$

In short, when $\mathcal{A}$ calls the DECRYPT oracle, $\mathcal{C}$ runs the DECRYPT_{G1} oracle instead, which returns a random functional decryption key dk as plain, as in the classical FE game. We prove thereunder that a PPT adversary cannot distinguish this game from the previous one (with a non-negligible advantage).

Claim 1. $|\epsilon_0 - \epsilon_1| \leq \epsilon(\mu)$, where $\epsilon(\mu)$ is the advantage of an efficient adversary that breaks the FE game.

To prove this claim, we consider a PPT adversary $\mathcal{B}$ that breaks the IND-FE-CPA game with an advantage $\epsilon(\mu)$. $\mathcal{A}$ constructs an oracle KEYDER_{G1} defined as follows:

$$\begin{array}{l} \text{KEYDER}_{G1}(f_0, f_1) \\ c_{dk_{f_\beta}} \leftarrow \$ \text{KEYDER}(f_0, f_1) \\ dk_{f_\beta} \leftarrow \$ \text{DECRYPT}(sk, c_{dk_{f_\beta}}) \\ \text{return } dk_{f_\beta} \end{array}$$

Note that the oracle KEYDER_{G1} is well-defined, as it only calls oracle accessible to $\mathcal{A}$. One also notices that KEYDER_{G1} corresponds to the classical key derivation oracle KEYDER defined in Figure 5.2. Therefore, $\mathcal{A}$ provides to $\mathcal{B}$ the master public key mpk and access to the oracles ENCRYPT and KEYDER . $\mathcal{B}$ plays the IND-FE-CPA game using these data and outputs to $\mathcal{A}$ a guess β'' on the value β with an advantage $\epsilon(\mu)$, which concludes.

Game 2: This game proceeds as the previous one, except that we replace the KEYDER oracle with a new oracle KEYDER_{G2} , defined as follows:

$$\begin{array}{l} \text{KEYDER}_{G2}(f_0, f_1) \\ \text{If } f_0, f_1 \notin \mathcal{F} : \text{return } \perp \\ c \leftarrow \$ \text{Im}(\text{HE.Enc}) \\ L_F \leftarrow L_F \cup \{(f_0, f_1, c)\} \\ \text{return } c \end{array}$$

In this game, when $\mathcal{A}$ calls the KEYDER oracle, $\mathcal{C}$ runs the output of KEYDER_{G2} instead, and hence substitutes a random element of the ciphertext space to the encryption of f_β . We prove that a PPT adversary cannot distinguish this game from the previous one (with a non-negligible advantage).

Claim 2. $|\epsilon_1 - \epsilon_2| \leq \epsilon(\lambda)$, where $\epsilon(\mu)$ is the advantage of an efficient adversary that breaks the HE game.

To prove this claim, we reduce Game 2 to the IND-FE-CPA game defined in Definition 13. We introduce an oracle ENCRYPT_{G2} defined as follows:

$$\begin{array}{l} \text{ENCRYPT}_{G2}(x_0, x_1) \\ c_\beta \leftarrow \$ \text{KEYDER}(x_0, x_1) \\ \text{return } c_\beta \end{array}$$

As before, we consider a second PPT adversary $\mathcal{B}$ that has an advantage $\epsilon(\lambda)$ in the IND-HE-CPA game. The oracle ENCRYPT_{G2} is well-defined, as it only calls oracle accessible to $\mathcal{A}$, and corresponds to the classical homomorphic encryption oracle ENCRYPT defined in Figure 5.3. During the IND-HFE-CPA game, $\mathcal{A}$ can forward (pk, evk) to $\mathcal{B}$. With access to both oracles EVAL and ENCRYPT_{G2} , $\mathcal{B}$ outputs a guess β'' on the value β with an advantage $\epsilon(\lambda)$, which concludes.

Game 3: This game proceeds as the previous one, except that we replace the ENCRYPT oracle with a new oracle $\text{ENCRYPT}_{\mathbf{G}_3}$, defined as follows:

$$\begin{aligned} & \text{ENCRYPT}_{\mathbf{G}_3}(\text{mpk}, x_0, x_1) \\ & \quad \text{If } x_0, x_1 \notin \mathcal{M} : \text{return } \perp \\ & \quad c \leftarrow \$ \text{Im}(\text{Enc}) \\ & \quad \text{return } c \end{aligned}$$

In this game, when $\mathcal{A}$ calls the ENCRYPT oracle, $\mathcal{C}$ runs the output of $\text{ENCRYPT}_{\mathbf{G}_3}$ instead, and hence substitutes a random element of the ciphertext space to the encryption of x_β . We prove that a PPT adversary cannot distinguish this game from the previous one (with a non-negligible advantage).

Claim 3. $|\epsilon_3 - \epsilon_2| \leq \epsilon'(\mu)$, where $\epsilon'(\mu)$ is the advantage of an efficient adversary that breaks the FE game in the sense of PKE.

The key argument to prove this claim is that the functional encryption c_β is not used in any oracle but the encryption, as KEYDER has been replaced by random oracles in the previous games. Hence, distinguishing the output c_β of the algorithm ENCRYPT from a random element $c \leftarrow \$ \text{Im}(\text{FE.Enc})$ is reduced to a traditional IND-CPA game in the sense of PKE. The advantage $\epsilon'(\mu)$ of an efficient adversary $\mathcal{A}$ in such a game is negligible.

Conclusion: Every oracle of Game 3 is independent from β , hence $\mathcal{A}$ has a nil advantage in Game 3. The overall advantage ϵ_{HFE} of $\mathcal{A}$ in the HFE game defined in Definition 14 is $\epsilon_{\text{HFE}} \leq \epsilon(\mu) + \epsilon(\lambda) + \epsilon(\mu')$. As a finite sum of negligible elements, ϵ_{HFE} is negligible, which concludes. $\square$

5.9 Experiments

To evaluate the practicality of our construction, HFE, we implemented a proof-of-concept based on our real-world scenario shown in Fig. 5.1. We chose the IPFE scheme of Mera *et al.* [37] and the Brakerski-Gentry-Vaikuntanathan (BGV) scheme [16] as the underlying FE and HE schemes for HFE. Our implementation uses the *Python* programming language. For testing and benchmarking, we followed best practices by using the *pytest* and *pytest-benchmark* libraries. Additionally, we measured memory usage with the *memory-profiler* library. Our experiments were conducted in a single-threaded environment on a PC with an Intel Core i7-8700 CPU @ 3.20GHz and 64GB of memory.

To ensure a solid comparison, we present benchmark results of RLWE-based IPFE [37] and HFE. The results include execution time, memory usage of various components, and storage costs associated with the multiple elements used during execution. Our comparison demonstrates that constructing a functional encryption scheme without relying on trusted entities is not only feasible but also comes with a reasonable overhead. We hope this deliverable encourages further research and development toward trustless FE schemes.

Open Science: To support open science and reproducible research, we made our implementation source code available online¹ (anonymized version).

5.9.1 Benchmarks

To benchmark RLWE-based IPFE and HFE, we follow the notation and algorithms in Sect. 5.7 using a modular approach for implementing schemes, where $\text{FE} = \{\text{Setup}, \text{KDer}, \text{Enc}, \text{Dec}\}$ and $\text{HE} =$

¹<https://anonymous.4open.science/r/HFE-EEFA>

$\{\text{KGen, Enc, Eval, Dec}\}$. For simplicity, we denote the transciphering process as Transcipher. As shown in Tab. 5.1, we define three parameter sets—Params = $\{\mathbf{S}, \mathbf{M}, \mathbf{L}\}$ —to evaluate performance across different system scales. The small set (**S**) uses input vectors m and functions f of length 50 with a polynomial ring degree of $\log_2 N = 5$. The medium set (**M**) extends this to a length of 500 with $\log_2 N = 6$, while the large set (**L**) increases it to 1000 with $\log_2 N = 7$. Additionally, Tab. 5.1 provides the ciphertext modulus, plaintext modulus, and bounds on the input and function vectors, denoted as $\log_2 q, \log_2 p, B_x, B_y$.

Table 5.1: Parameters for Test and Benchmark

Params	Vector Size	$\log_2 N$	$\log_2 q$	$\log_2 p$	B_x	B_y
Small	50	5	64	16	16	8
Medium	500	6	86	17	32	16
Large	1000	7	128	18	64	32

Running time and memory usage: As shown in Tab. 5.2, the running time of IPFE functions increases with the change of parameters. For the smallest benchmark (**S**), where vectors have a length of 50, the Setup phase takes $22.2ms$, while Enc, KDer, and Dec take $24.3ms$, $4.6ms$, and $5ms$, respectively. For the most significant benchmark (**L**), with vectors of length 1000, Setup extends to $2,988ms$, and Enc, KDer, and Dec require $3,098ms$, $243.7ms$, and $231.6ms$, respectively. However, as shown in Tab. 5.2, the memory usage of IPFE functions does not increase gradually. Even with larger vector lengths and input bounds, memory usage remains stable, averaging around $65MB$.

Similarly, as presented in Tab. 5.3, the running time of HFE functions increases as the parameters change. As expected, the execution times of the underlying FE functions Setup, Enc, Dec within HFE remain nearly identical to RLWE-based IPFE, with minor variations due to benchmarking conditions that are negligible. Since HFE employs BGV for transciphering, its running time is also reported. For the smallest benchmark (**S**), with vector length 50, HE.KGen takes $21.3ms$, while HE.Enc and HE.Dec take $47.8ms$ and $0.4ms$, respectively. In the largest benchmark (**L**), with vector length 1000, HE.KGen extends to $258ms$, and HE.Enc and HE.Dec require $6,192ms$ and $3.1ms$, respectively. Most importantly, in HFE, the running time of Transcipher increases significantly, from $1054.7ms$ in benchmark (**S**) to $335,695.6ms$ in benchmark (**L**). However, as shown in Tab. 5.3, the memory usage of HFE functions does not increase gradually. Even with larger vector lengths and input bounds, memory usage remains stable, averaging around $84MB$.

Storage costs: As shown in Tab. 5.4, the storage costs for various data in RLWE-based IPFE, including the message vector x , function f , master secret key msk , master public key mpk , FE-encrypted ciphertext c , functional decryption key dk_f , and the function result $f(x)$, are provided for three parameter sets. Note that the sizes of the message and function vectors are equal. As expected, increasing the vector length from 50 to 1000 and enlarging the plaintext and ciphertext modulus leads to higher storage costs. However, the rate of increase is more pronounced for master keys and ciphertexts due to the significant difference between the ciphertext and plaintext modulus. For example, comparing benchmark (**S**) and (**L**), the size of msk grows from $7.73KB$ to $414.39KB$, mpk increases from $24.83KB$ to $2,665.92KB$, and c expands from $24.83KB$ to $2,665.94KB$. Additionally, the ciphertext expansion rate ($\frac{c}{m}$) is approximately $225\times$ and $1353\times$ for benchmark (**S**) and (**L**), respectively. Overall, HFE is $21.8\times$ slower than RLWE-based IPFE for benchmark (**S**), $32.8\times$ slower for benchmark (**M**), and $54.1\times$ slower for benchmark (**L**), while requiring $2.11\times$ more memory for benchmark (**S**), $2.12\times$ for benchmark (**M**), and $2.41\times$ for benchmark (**L**).

Similarly, as presented in Tab. 5.5, the storage costs for HFE include the same RLWE-based IPFE data as well as additional BGV components: secret key sk , public key pk , evaluation keys evk , homomorphically encrypted master secret key c_{msk} , homomorphically encrypted message c_x , plaintext function p_f , homomorphically encrypted function c_f , and homomorphically encrypted functional decryption key c_{dk_f} . These costs are provided for three parameter sets. As expected, the storage costs for x, f, msk, mpk ,

Table 5.2: Running Time and Memory costs of Table 5.3: Running Time and Memory costs of HFE RLWE-based IPFE

Bench	Params	Time (ms)	Memory (MB)
Setup	S	22.2	55.99
	M	499.2	59.31
	L	2988	67.62
Enc	S	24.3	57.41
	M	521.1	61.56
	L	3098	76.99
KDer	S	4.6	67.51
	M	69.6	67.85
	L	243.7	70.69
Dec	S	5	67.06
	M	67.8	68.46
	L	231.6	76.50

Bench	Params	Time (ms)	Memory (MB)
FE.Setup	S	22.2	56.19
	M	494.5	59.20
	L	2967.5	67.73
HE.KGen	S	21.3	57.39
	M	62	57.44
	L	258	57.74
HE.Enc _{pk} (msk)	S	47.8	57.61
	M	1048.4	64.11
	L	6192	84.77
FE.Enc(mpk, x)	S	23.9	62.18
	M	520.9	63.78
	L	3096	77.26
HE.Enc _{pk} (f)	S	48.2	68.87
	M	1050.3	69.56
	L	6208.2	76.32
Transcipherer	S	1054.7	68.68
	M	35720.6	72.43
	L	335695.6	112.72
HE.Dec _{sk} (c _{dk})	S	0.4	75.46
	M	1	79.22
	L	3.1	113.12
FE.Dec(dk _f , c _x)	S	5	77.98
	M	68.3	81.35
	L	247.8	112.96

* Transcipherer : HE.Eval_{evk}(FE.KDer, (c_{msk}, c_f))

and $f(x)$ in HFE are nearly identical to RLWE-based IPFE, with minor variations due to benchmarking conditions that are negligible.

When examining the additional storage costs introduced by the BGV scheme, for benchmark (**S**), the total storage for sk and pk is less than 2KB, while *evk* requires 26.44KB. The storage costs for c_{msk} , c_x , c_f , and c_{dk_f} are 47.07KB, 24.26KB, 47.06KB, and 1.32KB, respectively. For benchmark (**L**), the total storage for sk and pk is less than 6.5KB, while *evk* requires 291.63KB. The storage costs for c_{msk} , c_x , c_f , and c_{dk_f} are 5, 203.88KB, 2, 602.64KB, 5, 203.82KB, and 5.58KB, respectively. Similar to RLWE-based IPFE, the results show that the rate of size increase is higher for master keys and ciphertexts due to the significant difference between the ciphertext and plaintext modulus. Overall, HFE requires $3.11\times$ more storage than RLWE-based IPFE for benchmark (**S**), $2.80\times$ for benchmark (**M**), and $2.84\times$ for benchmark (**L**), with the ciphertext expansion rate ($\frac{c_x}{x}$) reaching $220\times$ for small vectors and $1321\times$ for large vectors.

The comparison between the RLWE-based IPFE scheme and HFE highlights important trade-offs in security, performance, and resource consumption. While HFE introduces additional overheads, these costs are justified by its ability to eliminate the need for a trusted entity, providing a fully trustless and secure solution.

5.10 Conclusions and Future Directions

As this chapter combines two major modern encryption schemes, namely HE and FE, we provided an in-depth analysis of their definitions, constructions, and related security notions. This extensive background not only gives all the tools to fully comprehend the field but also highlights the significance of this work.

In the introduction, we identified two issues in the classical FE scenario. The first one, the problem of trust in an authority, was the central question and the motivation of our work. To address it, we developed a transciphering technique that takes advantage of the security provided by HE to ensure privacy in an untrusted environment. We showcase the validity of our approach by providing a formal definition of our construction, an analysis of its security, and a protocol supported by practical experiments. The second

Table 5.4: Storage Costs of RLWE-based IPFE

Name	S	Size (KB)	
		M	L
m	0.11	0.99	1.97
f	0.11	0.99	1.97
msk	7.73	118.68	414.39
mpk	24.83	529.48	2665.92
c	24.83	529.54	2665.94
dk_f	0.45	0.65	0.98
$f(m)$	0.01	0.01	0.02

Table 5.5: Storage Costs of HFE

Name	S	Size (KB)	
		M	L
m	0.11	0.99	1.97
f	0.11	0.99	1.97
msk	7.71	118.26	414.04
mpk	24.26	515.67	2602.61
sk	0.44	0.59	0.88
pk	1.32	2.46	5.59
evk	26.44	78.63	291.63
c_{msk}	47.07	1032.79	5203.88
c_m	24.26	515.66	2602.64
c_f	47.06	1032.77	5203.82
c_{dk_f}	1.32	2.45	5.58
dk_f	0.45	0.64	1.05
$f(m)$	0.01	0.01	0.02

problem was the question of function privacy, or how to ensure confidentiality of the functional request. The HE-FE transciphering technique we developed guarantees this feature naturally, without requiring additional computations or assumptions. This underlines the possibilities offered by the combination of HE and FE. Below, we propose future directions to extend the capabilities of our construction.

Future Direction. The protocol we designed eliminates the need for a trusted authority while allowing function blindness. However, the HFE scheme has the potential to support an even wider range of features. In particular, we believe it can serve as a differentially private (DP) FE scheme, ensuring the privacy of individuals in scenarios where a large volume of data is collected from a plethora of users. Originally formalized by Dwork *et al.* [26], it has been naturally extended to the realm of FE in 2022 by Bakas *et al.* [10] who proposed the first FE scheme achieving DP. Following, Zalonis *et al.* [60] introduced *noisy* FE – a specific FE paradigm which underpins a systematic DP construction for most FE schemes. This notion of noisy FE seems to be compatible with our construction, and could benefit from a homomorphic evaluation using an *approximate* HE scheme, such as CKKS [20].

Chapter 6

PiQASO NTT Accelerator for Lattice-based Schemes

This chapter presents the first version of **PiQASO's SW/HW co-design approach** to accelerating lattice-based cryptography, complementing the algorithmic optimization roadmap for **ML-KEM** and **ML-DSA** introduced in Chapter 3 and the broader cryptographic roadmap of Chapter 2. As established in **D3.1** [46], and summarized in Section 6.1, the dominant computational bottleneck of lattice-based constructions is *polynomial multiplication*, and specifically the **Number Theoretic Transform (NTT)** used to reduce this operation from quadratic to quasi-linear complexity. In Chapter 3, we discussed this bottleneck and the capability for addressing it in an algorithmic manner, by replacing the modular-reduction-heavy NTT with an **FFT-based, complex-domain convolution** technique that preserves constant-time execution. In this Chapter, we detail the *hardware-based* approach of PiQASO, which entails the use of a dedicated NTT/INTT accelerator, realized as a loosely-coupled coprocessor for far-edge and edge deployments.

The purpose of this hardware accelerator is twofold within PiQASO. First, it establishes a concrete HW-based acceleration capability for the NTT primitive that underlies **ML-KEM**, **ML-DSA**, and can be extended to also support **FN-DSA**, three of the lattice-based schemes targeted by the SDK (Chapter 3), as well as homomorphic encryption constructions, detailed in Chapter 5, including also numerical evaluations and bounds of the potential acceleration it can provide. Second, by validating a full hardware/software offload pipeline, this accelerator provides the architectural blueprint and the AXI control interface that will be reused and extended for the remaining PQC primitives targeted for hardware acceleration in **D3.3** [48] - *hardware acceleration of ML-KEM/ML-DSA operations* is identified as a priority for latency-sensitive PiQASO use cases such as V2X communications (Automotive, UC1), robot fleets (Automation, UC2), SCADA telemetry (Energy, UC4), and UAV control links (UC8).

The remainder of this chapter is organized as follows. Section 6.2.1 introduces the fundamentals of the NTT and the design goals of the accelerator. Sections 6.2.2 to 6.2.5 detail the architecture, from the top-level datapath down to the twiddle-factor generator and the control finite-state machine. Section 6.2.6 derives a closed-form latency model for the core, validated exactly against RTL simulation. Section 6.4 then evaluates the accelerator end-to-end on the ZCU104 platform against a scalar software baseline, quantifying the speedups relevant to PiQASO's target deployment scenarios. Finally, Section 6.5 discusses the accelerator's current limitations and the extensions planned as this work continues into **D3.3**.

6.1 Profiling of ML-KEM; ML-DSA & Performance Bottlenecks

The lattice-based standards ML-KEM (FIPS 203) and ML-DSA (FIPS 204) share a common computational profile that makes them costly to execute in software on embedded and edge platforms. Two

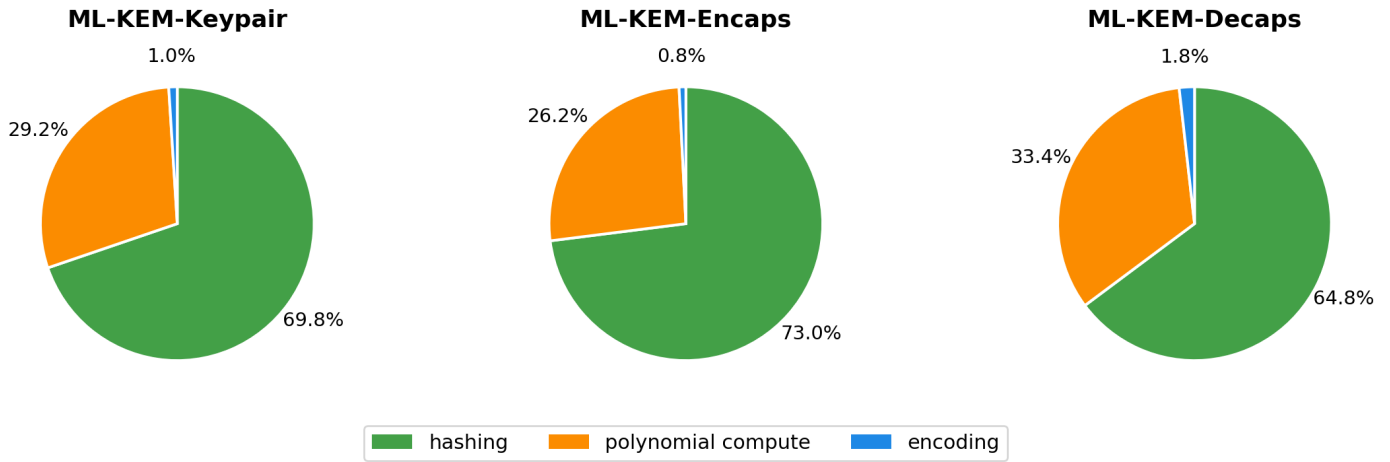


Figure 6.1: Software profiling of ML-KEM on a 32-bit platform: share of the execution time spent in hashing, polynomial computation and encoding for key generation, encapsulation and decapsulation.

kernels dominate their runtime: The first is polynomial multiplication over the ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$: although the Number Theoretic Transform (NTT) reduces its complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$, its butterfly-based dataflow induces irregular memory-access patterns that map poorly onto general-purpose processors. The second is pseudo-random polynomial generation, which relies on the SHA-3 / SHAKE functions (FIPS 202); since most embedded microprocessors lack native SHA-3 support, these functions account for a dominant fraction of the total execution time.

This dominance is confirmed by software profiling. Figures 6.1 and 6.2 report the breakdown of the execution time of ML-KEM and ML-DSA, respectively, on a 32-bit platform, separating the contributions of hashing, polynomial computation and encoding. Across every operation the hashing functions are the single largest contributor, reaching up to 73.0% of the runtime for ML-KEM (in encapsulation) and up to 74.9% for ML-DSA (in key generation). The share taken by hashing is smaller on a 64-bit processor, because the 64-bit lanes of Keccak are handled more efficiently there, but it remains one of the dominant contributions in both schemes.

Dedicated hardware acceleration of both kernels is therefore essential to make PQC practical under tight area, latency and energy budgets. A further requirement stems from physical security: because embedded devices are exposed to side-channel attacks, the countermeasures applied to sensitive operations must extend to the hash layer, so the SHA-3 accelerator is required to support not only the plain algorithms but also their masked variants (as will be elaborated further in D3.3 with the PiQASO masked implementation of both these algorithms).

In what follows, we document PiQASO’s hardware acceleration for the core mathematical operations of ML-KEM and ML-DSA: The purpose is to offload the polynomial arithmetic from the host processor. It supports the forward NTT, the inverse NTT (INTT) and the Coefficient-Wise Multiplication (CWM) over the ML-KEM ring $R_q = \mathbb{Z}_q[x]/(x^{256} + 1)$, together with modular polynomial addition and subtraction in hardware, so that the accumulation steps of the matrix-vector product need not fall back to software. It orchestrates the full sequence of forward NTTs, coefficient-wise multiplications, accumulations and inverse NTTs required for the parameter $k \in \{2, 3, 4\}$, and exposes a memory-mapped interface compatible with 32- and 64-bit buses. Currently the design is configured for one butterfly unit with the optimal mapping in memory organisation; the instance described here is also evaluated in detail for performance requirements and hardware area footprint (including number of memory-banks required).

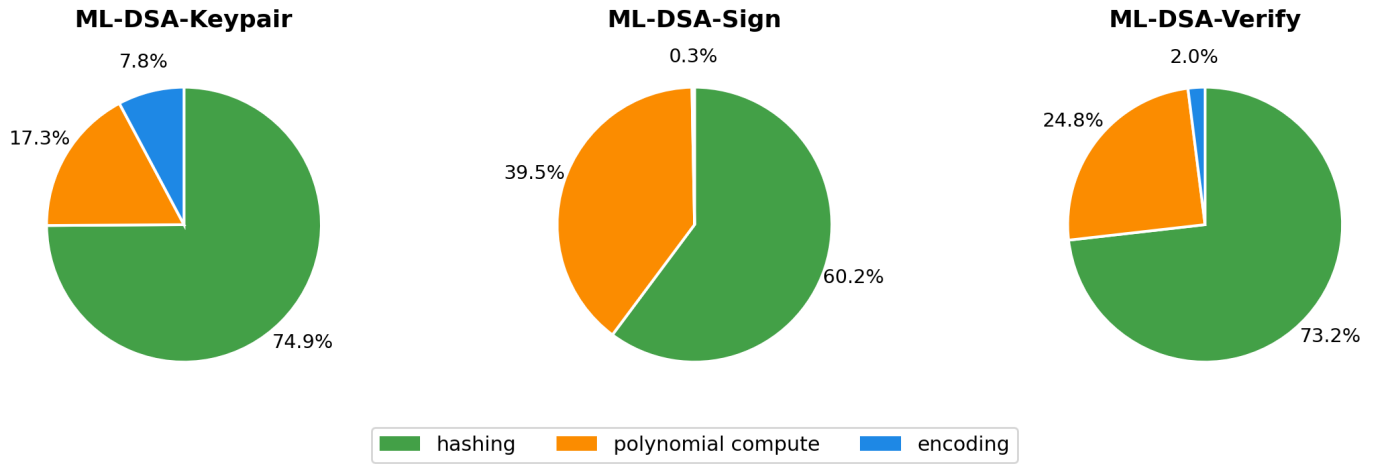


Figure 6.2: Software profiling of ML-DSA on a 32-bit platform: share of the execution time spent in hashing, polynomial computation and encoding for key generation, signing and verification.

6.2 NTT/INTT Hardware Accelerator: Architecture and Evaluation

Before describing the accelerator itself, it is useful to recall, at a high level, what the NTT and INTT actually compute and how they are used in the context of post-quantum cryptography. Lattice-based schemes such as ML-KEM and ML-DSA represent both keys and messages as *polynomials*, and their core operation is *polynomial multiplication*. Multiplying two polynomials of N coefficients directly costs $O(N^2)$ coefficient-wise multiply-and-add operations, which quickly becomes the dominant cost as N grows into the hundreds or thousands used by these schemes. The **Number Theoretic Transform (NTT)** avoids this quadratic cost. It re-expresses a polynomial in an alternative representation, over the finite field $\mathbb{Z}_q$, in which multiplying two polynomials reduces to multiplying their transformed coefficients *one position at a time* (a pointwise product), instead of the full cross-multiplication that schoolbook multiplication requires.

The **Inverse NTT (INTT)** performs the complementary step, converting the pointwise product back into a normal polynomial. Because computing the NTT and INTT themselves costs only $O(N \log N)$ rather than $O(N^2)$, this transform-multiply-inverse-transform pipeline is what makes polynomial multiplication practical at the sizes lattice-based PQC requires. In consequence, the NTT and INTT are invoked repeatedly inside essentially every operation of ML-KEM (key generation, encapsulation, decapsulation) and ML-DSA (key generation, signing, verification), and profiling of these schemes' reference software consistently shows that polynomial arithmetic, and specifically the NTT/INTT and the modular reductions that accompany them, is a leading contributor to their execution time. ***This is the bottleneck addressed by the PiQASO acceleration capabilities detailed throughout this Chapter. Moving the NTT/INTT into dedicated hardware therefore targets one of the dominant costs of every lattice-based operation in the SDK (Chapter 3).***

The two transforms are the main kernels behind *negacyclic* polynomial multiplication, meaning multiplication modulo $X^N + 1$, which is the form the schemes above use. Why a hardware implementation helps is worth stating up front. On a general-purpose CPU, most of the time goes into the modular reductions that follow every butterfly. A dedicated datapath, in contrast, can finish one butterfly every clock cycle. To make this practical, the core keeps all arithmetic in the **Montgomery domain** with radix $R = 2^{16}$. Modular reduction then needs only a multiply, an add and a conditional subtract, and no division on the critical path. The datapath follows the structure of a reference C implementation of the forward and inverse transforms. That same C code also serves as the software golden model in Section 6.4.

6.2.1 Fundamental Concepts

This section introduces, in plain terms, the mathematical notions needed to follow the rest of the chapter, namely the **FFT**, the **twiddle factors** it relies on, and the **decomposition strategy** that both the FFT and the NTT use to reach their speed.

Fast Fourier Transform (FFT). The FFT is a fast algorithm for computing the Discrete Fourier Transform, i.e., for re-expressing a sequence of N values as a sum of N oscillating (sinusoidal) components. Its significance here is that computing the transform directly, term by term, costs $O(N^2)$ operations, but the FFT computes the *same* result in only $O(N \log N)$ operations, by recursively splitting the problem into smaller ones (see **decomposition**, below). The NTT is the same fast algorithm, but carried out over a finite field $\mathbb{Z}_q$ instead of over complex numbers.

Twiddle factor. A twiddle factor is one of the fixed, precomputable multiplicative constants that the FFT/NTT combines with the data at each computation step. In the classical FFT these are powers of the complex root of unity $e^{-2\pi i/N}$, but in the NTT, they are the corresponding powers of a *root of unity modulo q* (an element $\psi \in \mathbb{Z}_q$ such that $\psi^{2N} \equiv 1 \pmod{q}$ and no smaller power does). Because the set of twiddle factors needed for a given transform size N is fixed and known in advance, it can be derived from a handful of constants instead of being stored in full. How the core supplies one such factor per cycle is the design problem solved in Section 6.2.4.

Decomposition. Decomposition is the “divide and conquer” step that gives the FFT/NTT its speed, specifically, a transform of size N is broken down into smaller transforms of size $N/2$, which are in turn broken down further, until each remaining piece is trivially small. The FFT and NTT specifically use a *radix-2* decomposition, meaning each split halves the problem in two, so that a size- N transform decomposes into $\log_2 N$ successive stages, each consisting of $N/2$ elementary two-input, two-output operations called **butterflies** (defined below). This essentially turns an $O(N^2)$ computation into an $O(N \log N)$ one, as there are $\log_2 N$ stages, but each performs $O(N)$ work.

With these three notions in hand, the operation this accelerator performs can be stated concretely. Given a length- N vector of polynomial coefficients, the **NTT evaluates that polynomial at N fixed points**, namely the odd powers of ψ , a primitive $2N$ -th root of unity in the finite field $\mathbb{Z}_q$, producing a new length- N vector in this “transformed” representation. Every intermediate and final value is therefore an exact integer residue in $\mathbb{Z}_q$, and not an approximate floating-point number, which is what makes the transform usable for cryptography. The same **radix-2 decomposition** still applies, and its basic operation is the **radix-2 butterfly**. Each butterfly takes two coefficients and one twiddle factor, and it returns two coefficients. A full size- N transform is therefore a fixed network of exactly $\frac{N}{2} \log_2 N$ butterflies. The butterfly is the smallest unit of work, so throughput depends on how often the core can finish one. The whole accelerator is built around a single goal, i.e., to *finish one butterfly every clock cycle*. Every mechanism described below serves that goal. Twiddle supply, operand fetch, coefficient storage and result write-back all exist to keep the arithmetic pipeline busy, with no stalls.

6.2.2 Overview

Before detailing the interfaces and offload sequence, it is worth clarifying what *loosely-coupled* acceleration means and why this is the integration model chosen here. A hardware accelerator can be attached to a host processor in essentially two ways. A **tightly-coupled** accelerator sits inside the processor’s own pipeline or instruction set (e.g., as a custom instruction or a functional unit sharing the processor’s registers), so that the host and the accelerator advance in lockstep, with minimal handoff overhead, but

with a design that is specific to one processor architecture. A **loosely-coupled** accelerator, by contrast, sits outside the processor pipeline, as an independent block in the programmable logic, and communicates with the host over a standard system interconnect (here, AXI), exchanging whole batches of data (an entire transform's worth of coefficients) per invocation rather than individual operands. This costs some fixed per-call overhead (measured in Section 6.4.1), since every invocation requires setting up and tearing down a data transfer, but it decouples the accelerator's internal design from the specific host core in use, keeps the accelerator reusable across different boards and host processors, and matches the offload pattern PiQASO needs. Specifically, the SDK issues one NTT/INTT call per polynomial operation, waits for it to complete, and continues, rather than requiring per-cycle interaction with the host pipeline. This is the same integration pattern reused for the HQC accelerator of Chapter 7.

The core is a **loosely-coupled accelerator** in the programmable logic (PL) of the **Xilinx Zynq Ultra-Scale+** platform, on the **ZCU104 evaluation board**, the same target platform used for the **HQC accelerator** of Chapter 7 and for the SDK's software benchmarking in Chapter 3, ensuring that all of PiQASO's hardware-acceleration results share a common, directly comparable baseline. It is not built into the processor pipeline. Instead, the Processing System (PS) drives it over the AXI interconnect and hands off one complete transform at a time. Three interfaces cross the PS/PL boundary:

- **AXI4-Lite (slave)**: carries the configuration registers (modulus q , transform size, Montgomery constants, twiddle memories) and the control and status handshake (start, busy, done). It runs in the slow control clock domain, on a **100 MHz** control clock.
- **AXI4-Stream (slave), driven by AXI DMA**: the input channel. It streams the input coefficients into the core, one 16-bit coefficient per beat, in the fast core clock domain at **200 MHz**.
- **AXI4-Stream (master), driven by AXI DMA**: the output channel. It streams the processed coefficients back to memory.

The two clock domains separate the control logic from the computation core, and the core runs at the higher rate. A full offload then proceeds in a few steps. First, the PS writes the parameters and pre-loads the twiddle-factor memories over AXI4-Lite. Next, the DMA streams the coefficient vector into the core. Inside the core, an input permutation and a ping-pong coefficient memory feed the butterfly unit, and the transform runs stage by stage between the two banks. After that, the result is streamed back out through the DMA. Finally, the PS checks for completion by polling the status register. Figure 6.3 shows the overall datapath.

6.2.3 Configuration and Control Interface

The accelerator is driven entirely from the PS. Before a transform can start, the core needs the transform size, the modulus, the two Montgomery constants and the transform direction. All of these are written over AXI4-Lite. None of them is used until START is asserted, and the core then samples them all at once, so the order of the writes does not matter. Since the parameters are supplied at run time rather than fixed in the RTL, one bitstream serves both transform directions and every (N, q) pair the datapath supports.

Table 6.1 lists the arithmetic context. The PS writes the size exponent $m = \log_2 N$ instead of N , and the core forms $N = 2^m$ with a single left shift, which keeps the address generators dependent on a small field. The two Montgomery constants $q' = -q^{-1} \bmod R$ and $R^2 \bmod q$ let the core enter the Montgomery domain and stay in it for the whole transform. The last register holds N^{-1} in Montgomery form and is read only by the inverse transform. Two further offsets, R and R_INV, are decoded by the register file but are unused in the current build.

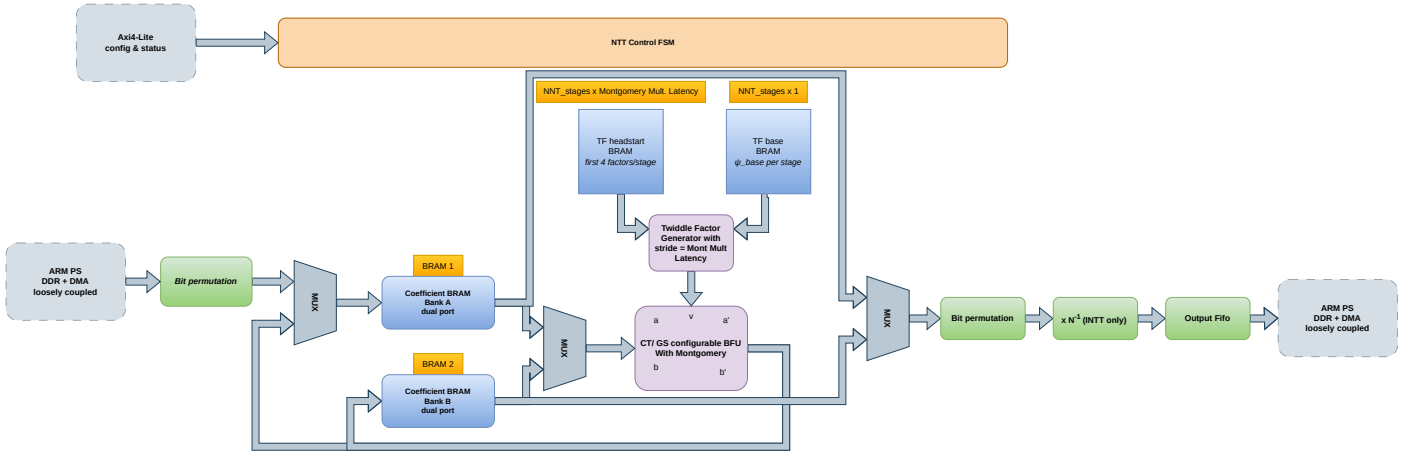


Figure 6.3: Top-level architecture of the NTT/INTT accelerator. Coefficients enter through the input permutation and a ping-pong coefficient memory. A single radix-2 butterfly unit, fed by the on-the-fly twiddle generator, runs the transform stage by stage between the two banks. The output path applies the N^{-1} scaling, used for the INTT only, and buffers the result before the AXI-Stream master.

Table 6.1: Arithmetic context written by the PS over AXI4-Lite before a run.

Register	Meaning	Used by
$\log_2 N$	transform size exponent ($N = 2^{\log_2 N}$)	FWD, INV
q	modulus	FWD, INV
$q' = -q^{-1} \bmod R$	Montgomery reduction constant	FWD, INV
$R^2 \bmod q$	Montgomery domain conversion	FWD, INV
$N^{-1} \text{ (Mont.)}$	inverse-size scaling factor	INV only

A single control and status register carries eight one-bit fields in three groups. `START`, `DONE_CLEAR` and `SOFT_RESET` are commands, written by the PS and acted on by the core. `BUSY` and `DONE` are status flags, driven by the core and read by the PS. `FWD_INV`, `INPUT_PERMUTE` and `OUTPUT_PERMUTE` are mode bits, and they are sampled at the `START` edge together with the arithmetic context. A forward transform is run with `INPUT_PERMUTE` set, so that the bit-reversal is folded into the load addressing. An inverse transform is run with `FWD_INV` and `OUTPUT_PERMUTE` set, where `FWD_INV` both selects the Gentleman–Sande dataflow in the butterfly and enables the N^{-1} scaler on the output path (Section 6.2.5).

The full sequence a host must follow is therefore the following.

1. Write the arithmetic context of Table 6.1 over AXI4-Lite.
2. Load the twiddle seeds into the two AXI4-Lite mapped memories, namely four headstart values and one stride constant per stage (Section 6.2.4). These depend only on (N, q, ψ) , so this step is repeated per parameter set and not per transform.
3. Arm the two DMA channels, one carrying the coefficients into the core and one carrying the results back.
4. Write the mode bits and assert `START`. The core latches the whole configuration on this edge, raises `BUSY`, and begins accepting the input stream.
5. Poll `DONE`, then clear it with `DONE_CLEAR` so that a stale flag cannot terminate the next run early.

The core is a fixed radix-2 datapath. For a correct result, the chosen pair (N, q) must allow a negacyclic transform. In other words, a primitive $2N$ -th root of unity must exist modulo q , and this requires $2N \mid$

$(q - 1)$. Table 6.2 lists the parameter sets used in this evaluation. All of them use the Falcon prime $q = 12289 = 3 \cdot 2^{12} + 1$, so $q - 1 = 2^{12} \cdot 3$.

Table 6.2: Validated parameter assignments. A negacyclic size- N transform requires $2N \mid (q - 1)$.

N	q	$2N$	$2N \mid (q - 1)$
256	12289	512	yes
512	12289	1024	yes
1024	12289	2048	yes
2048	12289	4096	yes

6.2.4 Twiddle-Factor Generation

As established in Section 6.2.1, every butterfly needs its own twiddle factor, and the accelerator’s one-butterfly-per-cycle goal therefore implies a matching requirement, that a fresh, correct twiddle factor must be ready at the butterfly unit *every single cycle*, with no exceptions. The most direct way to meet this would be to precompute all N twiddle factors in software and store them in on-chip memory, so that the core simply reads the next value each cycle. This section explains why the core does not take that route, and how it generates the twiddle factors on the fly instead, without ever stalling the butterfly pipeline.

A stored table has to be written into the core over AXI4-Lite before the first transform, and that cost is N writes per parameter set. It therefore grows with the transform size. The core instead **generates the factors on the fly**, using the recurrence $W_{i+1} = W_i \cdot \psi_{\text{base}}$. The host then writes fewer than eighty constants in total, namely four headstart seeds for each stage and one stride constant per stage, and this count barely moves with the transform size. The block RAM a full table would occupy is saved as well. The difficulty here is latency. The Montgomery multiplier that advances the recurrence takes four cycles, but the butterfly unit needs one twiddle every cycle. A simple single-element feedback loop would therefore stall the datapath for three cycles out of every four.

The core hides this latency with a headstart-plus-stride-4 scheme. A small headstart memory holds the first four factors of every stage. The feedback loop then multiplies by $\psi_{\text{stride}} = \psi_{\text{base}}^4$ instead of ψ_{base} . As a result, four interleaved recurrence streams fill the four pipeline slots of the multiplier, and one valid twiddle comes out every cycle. Because the four seeds are precomputed, there is no start-up stall. The first three stages ($s = 0, 1, 2$) need only $2^s \in \{1, 2, 4\}$ distinct factors, which is at most the stride. In these stages the generator simply reads from the headstart memory. It turns on the feedback path only from stage $s = 3$ onward, where the number of factors first passes four. The host computes both the four seeds per stage and the stride base ψ_{base}^4 once, in Montgomery form, and writes them over AXI4-Lite. Figure 6.4 shows the generator.

6.2.5 Datapath and Supported Features

This section follows a coefficient through the datapath, in the order it is handled. It arrives from the host and is written into the coefficient memory in bit-reversed order (**input permutation**). From there it is read as one half of an operand pair and passed to the **butterfly unit**, which is shared by both transform directions. The result is written into the second coefficient memory, and the two banks swap roles at every stage (**ping-pong**). On the way out, the inverse transform needs one step that the forward transform does not, namely an **output permutation** and a scaling by N^{-1} . A small **output FIFO** then holds the results while the AXI-Stream master drains them. These blocks, together with the control logic that sequences them, are what let the core sustain the one-butterfly-per-cycle target of Section 6.2.1.

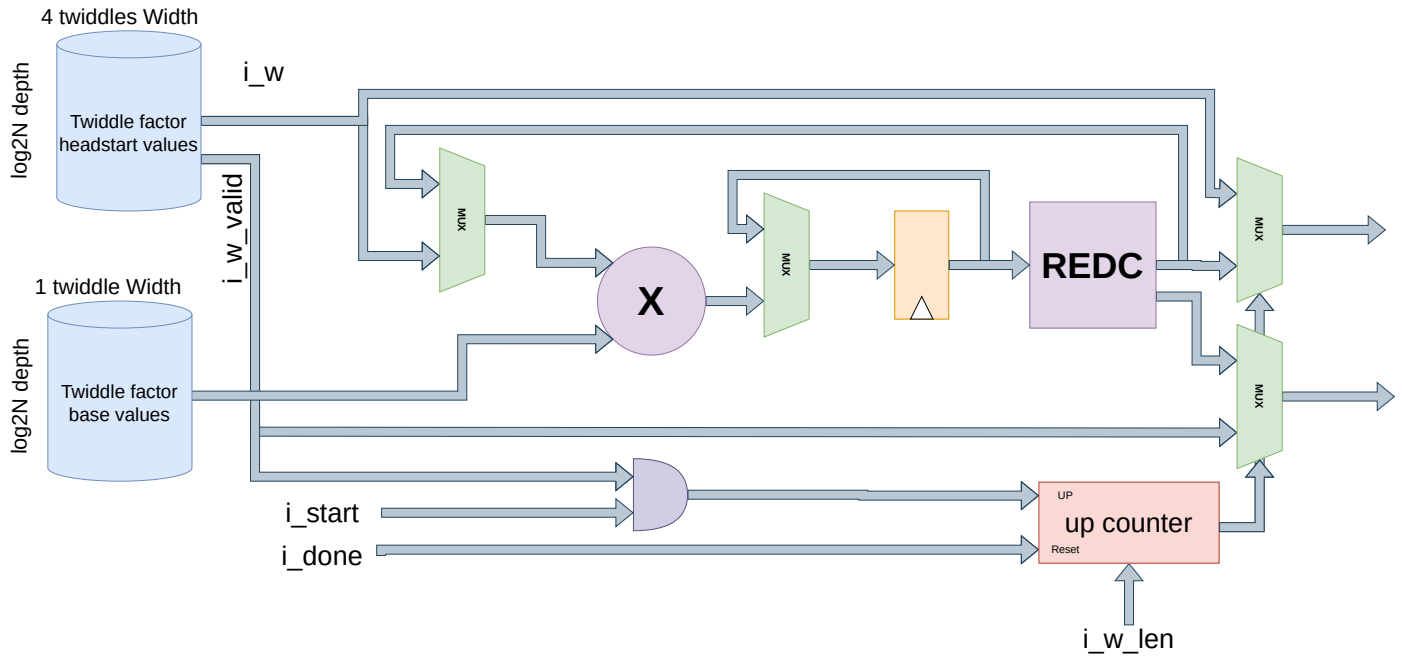


Figure 6.4: On-the-fly twiddle-factor generator. Four interleaved recurrence streams, seeded from the headstart memory and advanced by ψ_{base}^4 , fill the four-cycle Montgomery multiplier. This produces one valid factor per clock.

Input permutation. The Cooley–Tukey decimation-in-time schedule needs the input in bit-reversed order, so that the output comes out in natural order. The core does not use a separate reordering pass. Instead, it folds the bit-reversal into the write addressing while the coefficients stream into the coefficient memory. This costs no extra cycles. A separate pass would add another N elements, and the I/O cost would then be $3N$ instead of $2N$ (see Section 6.2.6).

Butterfly unit. A single unified radix-2 butterfly does the computation, one butterfly per clock. It supports both dataflows, and the transform direction selects between them:

$$\begin{aligned} \text{Forward (Cooley–Tukey): } & U' = U + V \cdot W, & V' = U - V \cdot W, \\ \text{Inverse (Gentleman–Sande): } & U' = U + V, & V' = (U - V) \cdot W. \end{aligned}$$

The butterfly is built from three primitives: (i) **modular addition**, (ii) **modular subtraction** and (iii) **Montgomery reduction** (Figures 6.5 to 6.8). Of the three primitives, the Montgomery reduction is the deepest, and it is split over three pipeline stages. The first forms $m = tq' \bmod R$. The second multiplies m by q , adds t and shifts the result down by R . The third applies the conditional subtraction. These three stages are the largest single contribution to the nine-cycle depth of the butterfly. That depth is a pipeline delay and not a throughput limit, since a new operand pair still enters the unit on every clock cycle (Section 6.2.6).

Ping-pong coefficient memory. Two coefficient memories work in a ping-pong arrangement. While the butterfly reads its operand pairs from one bank, it writes results into the other. The two banks then swap roles at each stage boundary. In this way the core reads and writes at the same time without conflict, and the butterfly stays fed every cycle.

Output permutation and inverse scaling. On the output side, a bit-reversal on the read address restores natural order for the inverse transform. This is needed because Gentleman–Sande produces

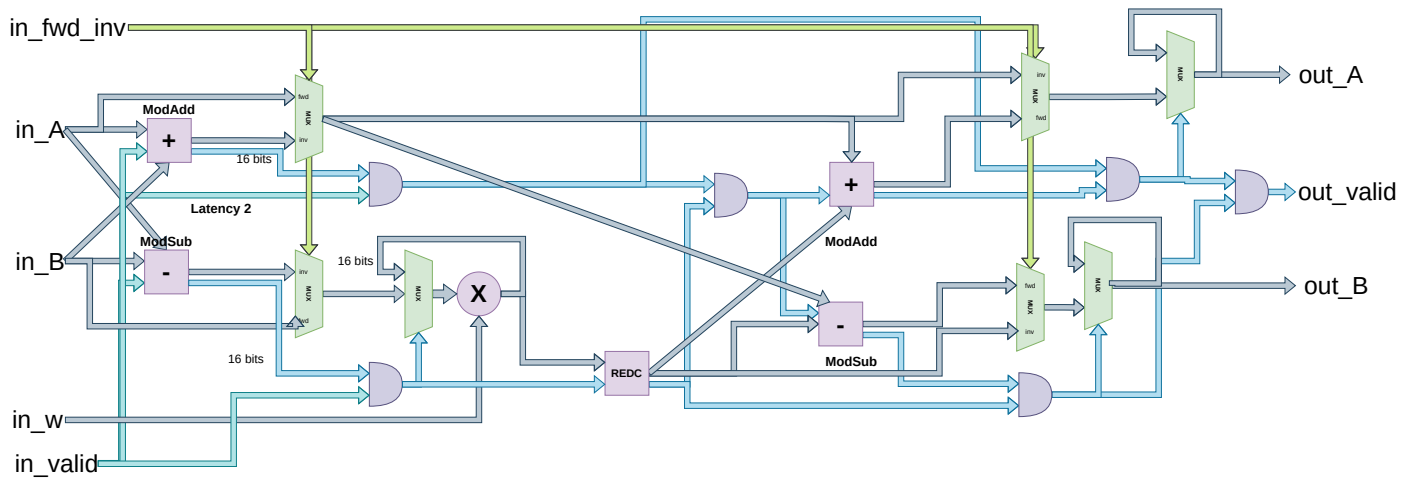


Figure 6.5: Unified Cooley–Tukey / Gentleman–Sande radix-2 butterfly unit. The transform direction selects between the two dataflows. The shared modular multiply-and-reduce path sets the pipeline depth.

bit-reversed output from natural input. The inverse transform also has to scale every coefficient by N^{-1} . The core does this as a streaming Montgomery multiply on the output path, not as a separate memory pass. The scaling is pipelined into the outgoing stream, so it adds no extra latency. As a result, the forward and inverse transforms take the same number of cycles (Section 6.2.6).

Output buffering and backpressure. The output path ends in a small output FIFO. It separates the core's steady output rate from the rate at which the AXI-Stream master and its DMA drain the data. Once the unload starts, the core sends out one coefficient per cycle, whereas the downstream channel can stall for short periods under interconnect contention. The FIFO absorbs those stalls. It takes the core's steady stream on one side, and releases the data under the stream master's ready/valid handshake on the other. The unload also runs after the last butterfly of the transform, so a stall on the AXI side cannot reach back into the arithmetic pipeline. The core signals DONE as the last coefficient enters the FIFO, so the drain to the AXI master falls outside the core latency of Equation (6.1).

Control and address generation. A single state machine in the core clock domain sequences the whole transform. It keeps counters for the stage, the group and the butterfly index, and from these it produces the read addresses, the twiddle index and the write addresses on every cycle. Nothing is held in an address table, so the sequencing depends only on the $\log_2 N$ value the host wrote. A read address is not discarded once the memory has been accessed. It is delayed by thirteen cycles in a chain of registers, which is exactly the time the data needs to pass through the memory read, the butterfly and the output registers. The address therefore reappears on the same cycle as its own result, and is applied directly to the write port. The same machine drives the coefficient memories through load, compute and unload, and swaps the two banks at each stage boundary.

6.2.6 Core Latency

Having described the datapath and the control logic that sequences it (Section 6.2.5), we now ask a practical question: *exactly how many clock cycles does a transform of a given size N take?* Answering this precisely, as a closed-form expression rather than only as a simulation result, matters for two reasons. First, it lets a system integrator **predict the accelerator's contribution to end-to-end latency for any N without having to re-run RTL simulation**, which is useful both for the evaluation later in this chapter

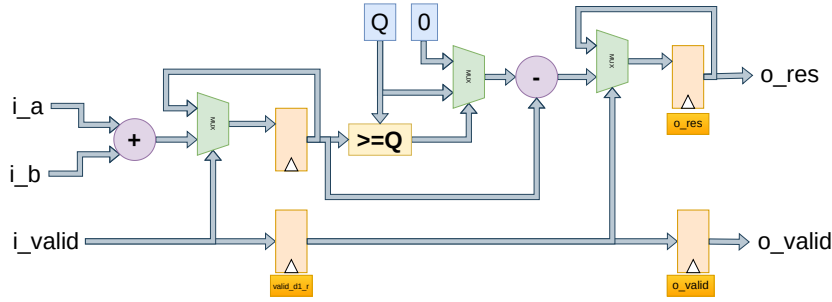


Figure 6.6: Modular addition primitive: add, then conditionally subtract q .

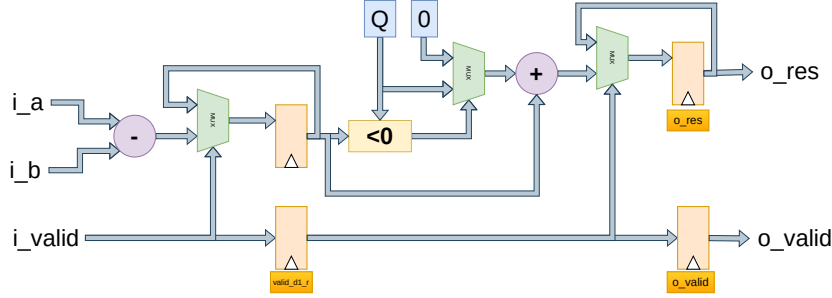


Figure 6.7: Modular subtraction primitive: subtract, then conditionally add q .

(Section 6.4) and for future integration into the SDK (Chapter 3). Second, deriving the formula term by term maps each term onto a specific structural feature of the design, namely the butterfly work, the streaming I/O, the per-stage pipeline drain and a fixed setup cost.

The core-only latency is the interval from the internal start pulse to the done pulse. It is measured with a free-running counter in the core clock domain, during RTL simulation. Across all supported sizes, one closed form describes it exactly:

$$\text{cycles}(N) = \underbrace{\frac{N}{2} \log_2 N}_{\text{butterfly work}} + \underbrace{2N}_{\text{stream I/O}} + \underbrace{13 \log_2 N}_{\text{per-stage drain}} + \underbrace{12}_{\text{one-time setup}}. \quad (6.1)$$

Each term maps onto a structural feature of the datapath.

- $\frac{N}{2} \log_2 N$ is the butterfly work itself. A single unit finishes one of the $N/2$ butterflies per cycle, across $\log_2 N$ stages. This term is the largest one, and it grows fastest with N . At $N = 2048$ it is about 73% of the total.
- $2N$ is the streaming cost. The input is streamed in and permuted (N beats), and the result is streamed out (N beats), at one coefficient per cycle each. The bit-reversal is folded into the load addressing, so there is no third N -beat pass.
- $13 \log_2 N$ is the per-stage pipeline drain and bank changeover. Stage $s+1$ reads the results written by stage s . The pipeline must therefore drain once per stage, before the banks swap. The per-stage constant is 13, which is the write-back latency the control logic counts, namely two cycles of memory read, nine cycles of butterfly and two further register stages.
- 12 is a one-time cost, paid once per transform instead of once per stage. It is the depth of the output path, four cycles to read a coefficient out of the memory and four more through the N^{-1} scaler, plus four cycles of start-up before the first read address is issued.

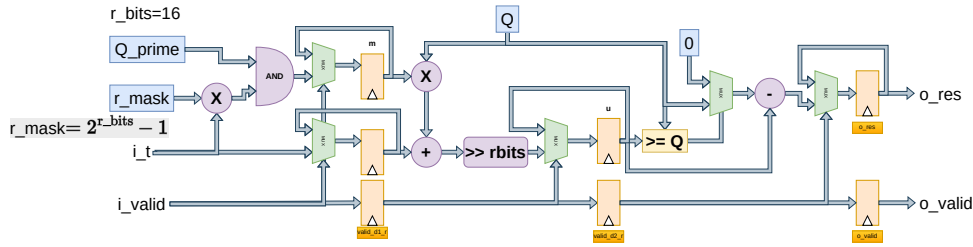


Figure 6.8: Montgomery reduction unit (radix $R = 2^{16}$), in three pipeline stages: the multiply that forms m , then the multiply-add-shift that forms u , then the conditional subtraction.

The nine-cycle butterfly depth breaks down as follows: two cycles of modular add and subtract, one multiply cycle, three cycles of Montgomery reduction, two more modular add and subtract cycles, and one output-register cycle. The four-cycle twiddle latency is the depth of the same Montgomery multiply-and-reduce path, reused in the generator's recurrence.

Table 6.3 validates the Equation (6.1) against the cycle count needed for its execution, as measured in RTL simulation. The model matches the measured value exactly at every size. Its two constants are taken from the design rather than from the measurements, 13 being the write-back latency the control logic counts and 12 the fixed depth of the output path and the schedule start-up. Two things follow from this. First, the latency is a pure function of N . It does not depend on q , on ψ , or on the data values, because the schedule is fixed. A run can therefore be measured at any modulus. Second, the forward and inverse transforms match to the exact cycle. This confirms that the inverse N^{-1} scaling is fully hidden in the output stream.

Table 6.3: Core latency: closed-form model vs. RTL simulation. All values are core-clock cycles at 200 MHz. The model equals the measurement exactly at every size.

N	$\log_2 N$	$\frac{N}{2} \log_2 N$	$2N$	$13 \log_2 N$	setup	model	measured
256	8	1024	512	104	12	1652	1652
512	9	2304	1024	117	12	3457	3457
1024	10	5120	2048	130	12	7310	7310
2048	11	11264	4096	143	12	15515	15515

6.3 PiQASO NTT/INTT Hardware Accelerator Integrated Design

Figure 6.9 shows the integrated architecture described before instantiated on the ZCU104 Board. It is organised around a Polynomial Unit containing the butterfly datapath and the modular add/sub logic, a twiddle-factor ROM, two memory banks, and a control subsystem, all connected to the host through the bus interface.

Polynomial Unit. Hosts the butterfly unit and implements NTT, INTT and CWM, as well as modular polynomial addition and subtraction. The datapath supports the Cooley–Tukey configuration for the forward transform and the Gentleman–Sande configuration for the inverse transform. Supporting add/sub in hardware keeps the accumulation of the matrix-vector product inside the accelerator.

Twiddle-factor ROM. Stores the precomputed powers of the primitive root of unity for the ML-KEM ring; it is addressed by the control unit according to the current transform stage.

Memory banks. Two independent sets of banks are provided, each composed of a Primary and a Secondary memory. Operations acting on a single polynomial (NTT, INTT) use only a Primary

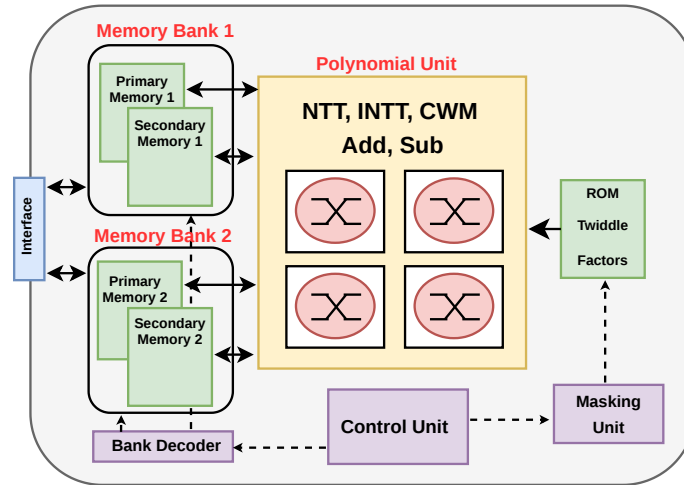


Figure 6.9: Architecture of the hardware accelerator for the ML-KEM NTT.

memory, whereas operations requiring two polynomials at once (CWM, addition, subtraction) use a Primary and a Secondary memory concurrently. The second set enables overlap: while one set feeds the computation, the other is accessed externally, which is the key to an efficient matrix-vector multiplication. On the bus, two 13-bit ML-KEM coefficients are packed into a single 32-bit word, halving the number of transfers.

Control subsystem. A Control Unit sequences the overall operation flow (transform stages, CWM, accumulation and the composed matrix-vector product), while a Bank Decoder generates the memory addresses that implement the conflict-free access pattern of the butterfly stages.

6.4 Evaluation

The accelerator is evaluated on the ZCU104 board. The PL is programmed from Vivado, and the application is attached to the running system. The application processing unit (APU, Cortex-A53) runs at 1.2 GHz, and the PL core runs at 200 MHz. Note that 200 MHz is the clock frequency of the current build. The core was not pushed to find its maximum operating frequency. Timing uses the APU cycle counter (PMCCNTR_ELO), and cycle counts are converted at $1 \mu s = 1200$ APU cycles. Each data point is the minimum over 30 timed iterations, after 5 untimed warm-up iterations. The minimum is reported rather than the mean, because it is the value least affected by interrupts and other one-off disturbances. Note that the tables in this section report the forward NTT only. The inverse transform runs on the same datapath, takes the same number of core cycles (Section 6.2.6), and was measured on the board in the same way. The two sets of numbers therefore coincide, and only one is presented here.

The **hardware** time reported here is the cost of a complete transform, not only the interval the core spends computing. It includes the cache flush of the transmit buffer and the invalidation of the receive buffer, the DMA transfers in and out, the core computation, the completion poll, and a second invalidation of the receive buffer once the transfer has finished. The accelerator writes its result straight to memory and does not pass through the CPU cache. The final invalidation is therefore needed so that the CPU reads the new result rather than an older copy it may still hold. The twiddle memories are written once per parameter set, not once per transform, so their cost sits outside the timed window.

The **software** baseline is a scalar radix-2 Montgomery NTT. It runs the same full negacyclic transform as the hardware. This includes the bit-reversal permutation, which the hardware does in the pipeline. It also

includes a copy of the input into a scratch buffer, so that each iteration starts from clean data. That copy mirrors the hardware DMA load. The baseline is compiled with `aarch64-none-elf-gcc 13.3`, using `-O3 -mcpu=cortex-a53 -funroll-loops`. The same baseline methodology is used for the HQC accelerator of Chapter 7. The implementation itself is not new here; it was released as part of earlier published work [57, 58]. It is a straightforward implementation rather than a tuned one, in that it makes an explicit bit-reversal pass and recomputes the twiddle factors of every stage. A comparison against a reference that keeps its twiddle factors in a precomputed table is planned for **D3.3**, and it would tighten the ratios reported below. Both sides report the same end-to-end quantity, which keeps the comparison fair.

6.4.1 Results

The **latencies** compared in this section are the *full, end-to-end, per-call* costs defined in the preceding paragraphs, not the core-only cycle count of Equation (6.1). Table 6.4 reports the hardware and software latencies, together with the resulting speedup, over the validated range. At every size the hardware result matches the software one (`errs = 0`). The accelerator is faster than the scalar software at every size. Moreover, the advantage grows with N , from $5.9\times$ at $N = 256$ to $14.1\times$ at $N = 2048$.

Table 6.4: End-to-end hardware vs. scalar-software latency at $q = 12289$. Times are the minimum over 30 iterations. Cycles are APU cycles at 1.2 GHz ($1\ \mu\text{s} = 1200\ \text{cyc}$). Speedup = $\text{SW}\ \mu\text{s} / \text{HW}\ \mu\text{s}$.

N	HW (μs)	HW (cyc)	SW (μs)	SW (cyc)	Speedup	Errs
256	19.94	23,928	117.99	141,596	$5.92\times$	0
512	29.00	34,799	265.96	319,154	$9.17\times$	0
1024	48.93	58,718	583.20	699,836	$11.92\times$	0
2048	90.01	108,007	1270.41	1,524,490	$14.11\times$	0

The rising speedup has two causes that add up. First, the software cost scales as $\Theta(N \log N)$. Its measured ratio per doubling is 2.25, 2.19 and 2.18, close to the $2(\log N + 1)/\log N$ expected of an $N \log N$ kernel. Second, the hardware cost grows more slowly per doubling, at 1.45, 1.69 and 1.84. This is not because the core scales better than the software, since it carries the same $N \log N$ factor and its own cycle count grows by 2.09, 2.11 and 2.12 per doubling (Table 6.3). What holds the hardware below two is the fixed per-call overhead, measured below, which is spread over a larger transform as N grows. Each doubling of N therefore costs the software more than a factor of two and the hardware less than a factor of two, so the distance between them widens at every step. The gap between the two curves is therefore built in, and not an artefact of a single size.

Table 6.5 breaks the hardware call into its parts and shows where the time is spent. The cache flush and the invalidation together stay below $1.1\ \mu\text{s}$ across the whole range, and they grow far more slowly than the transform. Issuing the two DMA transfers and starting the core accounts for the rest of the call. That part is not tabulated separately, so the three columns shown here do not add up to the totals of Table 6.4. Their difference is 548, 517, 530 and 529 cycles for the four sizes. The four values stay close together and do not grow with N , because this cost is paid once per call whatever amount of data follows. The remainder falls almost entirely in `hw_wait`, which covers the transfers, the core computation and the completion poll. Its share of the call rises from about 95% at $N = 256$ to 98% at $N = 2048$. The CPU-side work is therefore never the limiting cost, and the accelerator is faster than the software even at the smallest size.

Table 6.5: Hardware per-call breakdown, in APU cycles at 1.2 GHz. Twiddles are pre-loaded, so they are excluded. Here `hw_wait` is DMA plus core plus poll. The DMA-issue phase is not shown; it is the difference between these three columns and the totals of Table 6.4.

N	flush	hw_wait	invalidate
256	622	22,659	99
512	679	33,446	157
1024	753	57,204	231
2048	902	106,185	391

Finally, Table 6.6 compares the simulated core-only latency (Section 6.2.6, converted to time at 200 MHz) with the on-board `hw_wait`. The difference is about $10.8 \mu\text{s}$, and it stays between 10.6 and $11.1 \mu\text{s}$ while N grows eightfold. A gap that stays flat while the transform grows is the mark of a fixed cost. Issuing the transfers is not part of it, because that happens before `hw_wait` begins and was counted separately above. What is left is the start-up and completion latency of the DMA engine itself, together with the AXI-Lite reads of the polling loop. The per-element streaming is not in the gap either, since it already sits inside the core's $2N$ term.

That residual is the key observation, because Equation (6.1) already accounts for everything inside the core that depends on N . A cost which stays constant while N grows eightfold cannot be coming from the datapath. It belongs to the offload path, where the transfers are started and completion is detected. **Which of the two limits a call therefore depends on the size.** At $N = 256$ the fixed cost is more than half of `hw_wait`, so the offload sets the pace. At $N = 2048$ the core itself accounts for 88% of it, and the datapath sets the pace. Shortening the offload is a question of how the transfers are issued and how completion is signalled, and it helps most at the small sizes. Section 6.5 covers this together with the addition of further butterfly units, which is what the large sizes need.

Table 6.6: Simulated core-only latency vs. on-board `hw_wait`. The gap of about $10.8 \mu\text{s}$ is the fixed cost of starting the transfer and detecting completion. It shows no growth with N .

N	core μs (sim, 200 MHz)	hw_wait μs (board)	gap μs
256	8.26	18.88	10.6
512	17.29	27.87	10.6
1024	36.55	47.67	11.1
2048	77.58	88.49	10.9

6.4.2 Discussion & Critique

Two points put these numbers in context. The first is what the speedups actually compare. They compare implementations and not a normalised algorithm, so the accelerated path is $5.9\times$ to $14.1\times$ faster than this particular scalar software, and not than the NTT as such. The hardware side also carries DMA and cache costs that the software side does not, and any system-level claim remains bounded by the share of the end-to-end work the NTT takes. The second is where the remaining headroom lies. It is divided between the butterfly term at large N and the fixed offload cost at small N , and the two call for different fixes. Both are engineering limits rather than fundamental ones, and Section 6.5 sets out how each can be reduced.

These results are directly relevant to the target parameter sets of the PiQASO SDK's lattice-based primitives (Chapter 3), since **ML-KEM** and **ML-DSA** both operate on polynomials of 256 coefficients (with ML-DSA additionally requiring multiple such polynomials per matrix/vector operation), which falls inside the validated transform sizes of Table 6.4. The moduli are a separate matter. ML-KEM's $q = 3329$ does not admit a full negacyclic transform at $N = 256$, and ML-DSA's 23-bit modulus is wider than the present datapath. The measured $5.9\times$ – $14.1\times$ speedup is a kernel-level figure, and what a full scheme operation gains from it depends on how much of its runtime the NTT takes. Quantifying that is part of the SDK integration planned for **D3.3** [48]. Moreover, N , q and the transform direction are register writes and not synthesis constants (Section 6.2.3), so one instance can in principle serve *several* parameter sets in turn, provided each stays inside the arithmetic limits of the datapath. This is directly aligned

with PiQASO's crypto-agility objective of **dynamic hardware resource sharing across cryptographic algorithms** (Chapter 2), rather than requiring one accelerator instance per scheme.

6.5 Limitations and Future Work

The results above are validated up to $N = 2048$ at $q = 12289$, the 14-bit prime used by FN-DSA (Falcon). Two separate limits produce that ceiling. The first is the modulus itself. Since $q - 1 = 2^{12} \cdot 3$, the largest $2N$ that divides it is 4096, so no negacyclic transform exists beyond $N = 2048$ and a larger size needs a larger prime. The second limit is what stops a larger prime from being used. The arithmetic of the present build stays correct only up to about $q = 40500$, which is the widest modulus whose intermediate Montgomery value still fits the datapath. What blocks a larger transform is therefore the width of the arithmetic and not the throughput of the core. On this basis we identify four areas for future work.

- **Wider arithmetic datapath.** A valid negacyclic transform at $N = 4096$ needs $q \equiv 1 \pmod{8192}$. The smallest such prime is 40961, a full 16-bit modulus. The present Montgomery reduction forms the intermediate value $T + m q$ on a 32-bit path, and that path overflows for such a modulus. Widening the reduction datapath to at least 33 bits, and checking the butterfly for the same assumption, would open up the full 16-bit prime range and a correct $N = 4096$.
- **More butterfly units.** The dominant $\frac{N}{2} \log_2 N$ term becomes $\frac{N}{2P} \log_2 N$ for a core with P butterflies. Replicating the unit is therefore the most direct way to cut this latency.
- **Higher-radix butterflies.** A radix- 2^2 or radix-4 butterfly halves the number of stages. Since the pipeline drains once per stage, this halves the total drain as well, although a radix-4 butterfly is itself deeper than a radix-2 one.
- **Leaner offload.** The busy-poll and single-shot DMA descriptors could be replaced by interrupt-driven completion, descriptor chaining, or a wider AXI-Stream. The first two would reduce the constant per-call floor of about $10.8 \mu\text{s}$, which currently dominates at small N . A wider stream would instead shorten the $2N$ streaming term inside the core.

Beyond these four directions, the natural next step within PiQASO is the **end-to-end integration of the accelerator into the SDK**, so that ML-KEM and ML-DSA operations can transparently offload their polynomial-multiplication step to the PL, mirroring the *loosely-coupled* integration pattern already validated here and reused for the HQC accelerator of Chapter 7. **FN-DSA (Falcon)** already falls inside the validated range, since its parameter sets are $N = 512$ and $N = 1024$ at $q = 12289$, both measured above with $\text{errs} = 0$. In parallel, extending the arithmetic datapath to the full 16-bit modulus range would take the core beyond that, to $N = 4096$ and the larger primes such a transform requires. Both extensions are planned for **D3.3** [48], together with a joint evaluation against the algorithmic, FFT-based optimization track of Chapter 3, so as to quantify, on a common platform, the relative benefits of *compiler-level*, *algorithmic*, and *hardware-based* acceleration of lattice-based arithmetic, a comparison that directly informs which combination of these techniques PiQASO's use case demonstrators should adopt for their latency and resource budgets (Chapter 2).

Finally, another dimension to be explored by PiQASO is the **hardware implementation of ML-KEM and ML-DSA with side-channel protection**. As it has been highlighted, the acceleration of the core bottlenecks for lattice-based algorithms has been the focal point of industry and academic research, most of which are optimised for performance or area but provide no protection against physical attacks. Masking is the standard countermeasure against side-channel attacks on lattice-based schemes, but

masking ML-KEM is not straightforward, because its decapsulation mixes operations of different algebraic nature. The Fujisaki–Okamoto (FO) transform requires the whole decrypt–re-encrypt–compare chain to be protected. The polynomial arithmetic (NTT, pointwise multiplication and additions) can be masked with arithmetic shares modulo q , whereas the symmetric primitives (SHA-3/SHAKE) and the compression and comparison steps require Boolean masking. As a result, most of the cost of masked ML-KEM comes from the masked Keccak permutation and the A2B/B2A conversions, rather than from the polynomial arithmetic itself. [13] gives the first complete masked implementation of Kyber (the scheme later standardised as ML-KEM) at any order. It splits decapsulation into maskable parts, shows that the linear NTT and polynomial arithmetic are cheap to mask while the non-linear steps are expensive, and provides the key gadgets for the latter: a masked binomial sampler, the A2B/B2A conversions, and a masked ciphertext comparison. It is the algorithmic reference for how masking should be structured in ML-KEM. [33] specialises this to the first order on the ARM Cortex-M4, with a heavily optimised implementation whose critical routines (NTT, conversions, Keccak, sampling, comparison) are written in assembly, and validates it on the device with a Test Vector Leakage Assessment (TVLA) that detects no first-order leakage; it has become a common software baseline for masked ML-KEM. [28] is the most relevant mixed hardware/software design: it extends a RISC-V core with instruction-set extensions and masked accelerators for exactly the operations that dominate software—the **Keccak permutation, the mask conversions, the binomial sampling, the NTT and the masked comparison**—and thereby runs masked lattice KEMs at a fraction of the software cost. *This is the central idea of the mixed HW/SW approach that will be investigated in PiQASO: the primitives that are at once the most expensive and the most sensitive (the masked SHA-3, the NTT and the A2B/B2A conversions) are the ones worth moving into hardware.*

Chapter 7

HQC Acceleration by Speeding Up Binary Polynomial Multiplications (BPM)

Having established the PiQASO PQC SDK as the software foundation for post-quantum cryptography (Chapter 3) and the PQCaaS component as a means of extending this protection to organizations and devices that cannot embed PQC directly, this chapter addresses another core pillar of PiQASO's approach to building quantum-safe infrastructures as detailed in Chapter 2, namely **hardware acceleration**. While the SDK and PQCaaS cover the software-level integration paths for the post-quantum transition, a purely software-based approach remains insufficient for the latency, throughput, and energy constraints encountered in resource-constrained edge deployments. As first identified in Deliverable D2.1 [45], and further delineated in Table 3.41, PiQASO's acceleration approach is centred around *the identification of the most computationally intensive mathematical and arithmetic operations, which constitute bottlenecks towards improving the performance of the examined PQC primitives*.

In this regard, this chapter focuses on identifying and addressing the performance bottlenecks in **Hamming Quasi-Cyclic (HQC)** scheme, which has been selected and standardized by NIST to complement lattice-based ML-KEM and provide cryptographic diversity. As will be detailed in the following, the core operation identified as the dominant bottleneck is **dense-dense binary polynomial multiplication**. However, in contrast to other NIST-standardized PQC primitives, HQC has received comparatively little hardware acceleration attention to date, and the aforementioned bottleneck has grown more computationally expensive following recent constant-time hardening of the specification.

Compounding all the above, building on the architectural direction outlined in D2.1 and the underlying algorithmic foundations of D3.1 [46], this Chapter presents a high-level overview of **the accelerator developed within PiQASO for Binary Polynomial Multiplication to establish a definitive hardware performance baseline** for this operation, isolating the polynomial multiplication kernel. *Note that, while in this deliverable we provide a high-level component synthesis of binary polynomial acceleration and detailed evaluation results, technical specifications and implementation details will be provided in D3.3 [48], as this work is currently under review for publication.*

7.1 Current PQC Landscape and the HQC Bottleneck

With the global mandate to transition to Post-Quantum Cryptography (PQC) by 2035, NIST has selected the code-based Hamming Quasi-Cyclic (HQC) scheme to complement lattice-based ML-KEM. Building on the historical security of the McEliece cryptosystem, HQC offers vital cryptographic diversity. However, a gap remains between theoretical maturity and practical deployments. The latter, especially when it comes to resource-constrained devices, requires a certain level of hardware agility. Ideally, resources

should be dynamically shared between multiple algorithms. Yet, rushing to multi-scheme architectures is premature without first understanding *the absolute limits of individual primitives*. Isolating a specific operation reveals where performance saturates and how bottlenecks shift. This requires a detailed profiling to identify how specific operations (such as Hashing, Polynomial Multiplication, etc) contribute to protocol-level latency.

While optimization for ML-KEM is well-explored, **HQC hardware acceleration is in its infancy**. The primary performance bottleneck is binary polynomial multiplication (BPM) in $\mathbb{F}_2[x]$. To mitigate side-channel vulnerabilities, the latest HQC specification abandoned non-constant-time sparse-dense multiplication in favor of constant-time dense-dense multiplication. This security fix came at a steep computational cost: **dense-dense multiplication now accounts for 96% of total execution time** across all security levels and KEM operations (key generation, encapsulation, and decapsulation) when running the standard PQ-Clean reference implementation on an ARM Cortex-A53. Table 7.1 contains the profiling measurements.

Table 7.1: Profile of the HQC reference software (PQClean C, -O3, no NEON) on Cortex-A53. Times are mean per-call wall-clock in μs ; each KEM round-trip issues six `vect_mul` calls. `vect_mul` accounts for about 96% of total runtime.

Operation	calls	HQC-128			HQC-192			HQC-256		
		vect_mul = 12 068 $\mu\text{s}/\text{call}$			vect_mul = 37 101 $\mu\text{s}/\text{call}$			vect_mul = 67 814 $\mu\text{s}/\text{call}$		
		vect_mul	rest	mul %	vect_mul	rest	mul %	vect_mul	rest	mul %
Key generation	1	12068	442	96.5	37101	1175	96.9	67814	2341	96.7
Encapsulation	2	24135	943	96.2	74202	2376	96.9	135628	4584	96.7
Decapsulation	3	36203	1704	95.5	111303	3923	96.6	203441	7867	96.3
Round-trip	6	72406	3091	95.9	222606	7476	96.8	406883	14795	96.5

In order to address the bottleneck outlined above, **PiQASO constructs a highly optimized hardware accelerator for the BPM of HQC**. Our architecture aims to establish a definitive hardware baseline for the specific operation. Contrary to traditional approaches that rely on Custom RISC-V Extensions (e.g. [25] and [53]), or more recent ones that implement Frobenius Additive FFT (PHOENIX [50] and ANDROMEDA [49]), **we use a flattened, non-recursive variant of the Karatsuba algorithm**.

7.2 Preliminaries

Hamming Quasi-Cyclic (HQC) is a code-based, IND-CCA2 secure key-encapsulation mechanism (KEM) whose foundational security relies on the hardness of decoding random quasi-cyclic codes over $\mathbb{F}_2$. All core mathematical operations within HQC are evaluated over the polynomial ring $R = \mathbb{F}_2[X]/(X^n - 1)$, with $n = 17669$, 35851 , and 57637 corresponding to the HQC-128, HQC-192, and HQC-256 that target NIST security levels 1, 3, and 5, respectively. The KEM consists of three primary routines: KeyGen, which samples low-weight secret vectors $x, y \in R$ to compute a public key $s = x + y \cdot h$; Encaps, which generates a ciphertext (u, v) via random low-weight errors and derives a shared key using a SHAKE extendable-output function; and Decaps, which utilizes a concatenated Reed-Muller and Reed-Solomon decoder to recover the message before performing a re-encryption consistency check. Across all three routines, the primary computational bottleneck is the cyclic binary-polynomial product (`vect_mul`), consisting of a large multiplication in $\mathbb{F}_2[x]$ and a subsequent modular reduction, whereas the remaining hashing, sampling, and decoding components remain comparatively lightweight.

The operands for this critical polynomial product are dense bit-vectors packed into 64-bit words, where the modular reduction is a computationally cheap cyclic convolution, leaving the polynomial multiplication to dominate the overall execution profile. To achieve *sub-quadratic complexity* while maintaining a

constant-time execution profile to mitigate side-channel risks, HQC avoids data-dependent sparse multipliers in favor of dense-dense Karatsuba multiplication, reducing the algorithmic complexity to $O(n^{\log_2 3}) \approx O(n^{1.585})$. By splitting polynomials into lower and higher halves such that $A = A_L + \epsilon A_H$ and $B = B_L + \epsilon B_H$ with $\epsilon = x^{\lceil \frac{n+1}{2} \rceil}$, the product is reconstructed via the relation

$$C = \left[C_H^{(1)} \epsilon + \left(C_M^{(1)} \oplus C_H^{(1)} \oplus C_L^{(1)} \oplus C_H^{(0)} \right) \right] \epsilon^2 + \left[C_L^{(0)} + \epsilon (C_M^{(0)} \oplus C_L^{(0)} \oplus C_L^{(1)} \oplus C_H^{(0)}) \right], \quad (7.1)$$

where $C_L = A_L B_L$, $C_H = A_H B_H$, and $C_M = (A_L + A_H)(B_L + B_H)$. This divide-and-conquer strategy is exceptionally well-suited for hardware implementation because the carry-less additions map directly to shallow, low-latency XOR trees, and the fixed split yields a static, data-independent execution flow. **The accelerator builds on this by implementing a flattened Karatsuba layout**, expanding the recursive decomposition into a hardware-friendly flow that processes all leaf multiplications through a single, shared carry-less multiplier engine and folds the results back at statically predetermined memory addresses.

To establish a concrete performance and architectural baseline, all hardware and software evaluations are targeted at the Xilinx Zynq UltraScale+ MPSoC ZCU104 platform, which features a tightly integrated Processing System (PS) and Programmable Logic (PL) on a single chip. The PS side contains a quad-core Arm Cortex-A53 processor operating at 1.2 GHz paired with 2GB of 64-bit DDR4 RAM, serving as the host environment for executing the non-accelerated portable software baseline. The accelerator is synthesized within the 16nm PL fabric, utilizing its available resources of 230,400 look-up tables (LUTs), 460,800 flip-flops, 312 block-RAM tiles (36 Kb each), 96 UltraRAM tiles (288 Kb each), and 1,728 DSP slices. High-speed communication and data movement between the software stack running on the ARM core and the accelerator in the programmable fabric are managed via configurable Advanced eXtensible Interface (AXI) ports, supporting data widths ranging from 32 to 128 bits to optimize system-level throughput.

7.3 PiQASO's Accelerator Architecture

Our hardware design is based on a **flattened variant of the Karatsuba multiplication**. The algorithm has three phases Expansion, BCM (Base-Case Multiplication) and Compression. By transitioning from traditional recursive execution to a flattened Karatsuba architecture, the design eliminates the need for complex, dynamic memory orchestration and heavy control logic overhead. Instead, this unrolled layout allows the hardware to utilize a highly deterministic datapath governed entirely by simple finite-state machines, hardware counters, and static address generators.

At its arithmetic core, **the PiQASO accelerator trades logic area for temporal reuse** by instantiating a single, spatial Karatsuba engine that is time-multiplexed over successive clock cycles. Rather than dedicating separate hardware to every leaf node in the Karatsuba tree (which would be prohibitively expensive) the architecture processes partial polynomial products serially. The working data is maintained entirely within localized, high-speed on-chip memory primitives configured to match the datapath width, enabling single-access processing without the need for multi-cycle width-matching overhead.

The PiQASO accelerator is designed to realize the dense-dense polynomial multiplication required by HQC through a spatially optimized, flattened Karatsuba architecture. By completely unrolling the traditional recursive structure into a deterministic sequence of execution steps, the hardware bypasses the need for dynamic runtime stack management or complex memory orchestration logic. The entire execution workflow is governed by static finite-state machines (FSMs) and predictable address generators. Because every processing step represents a linear sweep over memory determined solely by the targeted HQC parameter set, the execution schedule is entirely fixed at design time. **This data-independent**

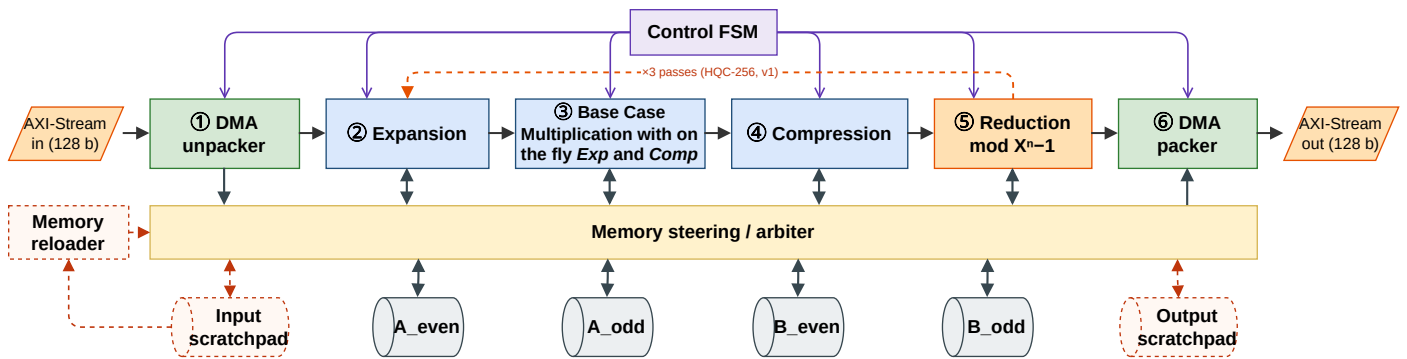


Figure 7.1: Block diagram of the accelerator for both variants.

execution flow guarantees a constant-time profile, mitigating timing-based side-channel vulnerabilities. The developed accelerator is instantiated in two distinct hardware configurations representing two variants of the accelerator that use different datapath width, as illustrated in Figure 7.1:

1. **Variant 1 (v1)**, which features a narrower datapath.
2. **Variant 2 (v2)**, which employs a wide, high-throughput datapath.

Both variants implement a three-stage flattened Karatsuba execution workflow using an identical chain of functional modules. They differ primarily in their memory and datapath bus widths, as well as in the operand routing logic fed into the central multiplication processing element.

Having detailed the architecture of the accelerator, we next focus on the **execution sequence**, which corresponds to a pipeline consisting of six sequential modules: the **Direct Memory Access (DMA) unpacker**, **expansion**, **base-case multiplication**, **compression**, **modular reduction**, and the **DMA packer**. Arithmetic computation is carried out by the four intermediate stages, which execute sequentially over shared memory banks. Within this sequence, the reduction stage applies a cyclic convolution modulo $X^n - 1$ to fold the unreduced double-length product. This standard pipeline execution is uniform across parameter sets, with one notable exception: processing HQC-256 on the compact v1 engine requires an iterative, depth-first execution mode. To support this depth-first scheme without expanding the physical memory width, v1 incorporates auxiliary passes, processing logic, and scratchpad memories (highlighted with dotted outlines in Figure 7.1).

Figure 7.1 presents the block diagram of the accelerator. In the narrower variant, exclusively for HQC-256, we use three depth-first calls in order to reduce the required memory footprint.

7.3.1 System Integration and Scalability

To seamlessly interface with the host processor, the accelerator incorporates dedicated Direct Memory Access (DMA) buffering modules that operate over a standard high-throughput AXI-Stream interface. These modules handle the packaging and unpacking of data between the system's memory bus and the local accelerator fabric. Figure 7.2 shows the integration of the acceleration to the Processing System of the development board. To maintain a uniform, single-primitive-deep memory geometry across all configurations, the architecture scales gracefully across various security levels. For memory-constrained configurations or high-security parameter sets like HQC-256, the design elegantly switches to an iterative, depth-first execution mode. This configuration partitions the top-level operands and routes them through successive passes of the same hardware chain to compute the final, reduced product without expanding the physical silicon footprint.

7.4 Evaluation

We evaluate the proposed accelerator integrated as a loosely coupled coprocessor within a full System-on-Chip (SoC) environment, measuring its *hardware footprint*, *standalone unit performance*, and *end-to-end impact* within the HQC Key Encapsulation Mechanism (KEM).

The accelerator interfaces with the Processing System (PS) of the target board via two dedicated AXI channels, as depicted in Figure 7.2. A 32-bit master port (HPM0), driven by the PS, manages the configuration registers of both the AXI DMA engine and the accelerator’s control unit. High-throughput data movement between the PS DDR memory and the Programmable Logic (PL) is routed through a 128-bit slave port (HPO). The host ARM Cortex-A53 core operates at 1.2 GHz. Within the PL, the narrower-bus variant achieves a maximum clock frequency of $f_{\max} = 270$ MHz, whereas the wider-bus variant operates at $f_{\max} = 200$ MHz.

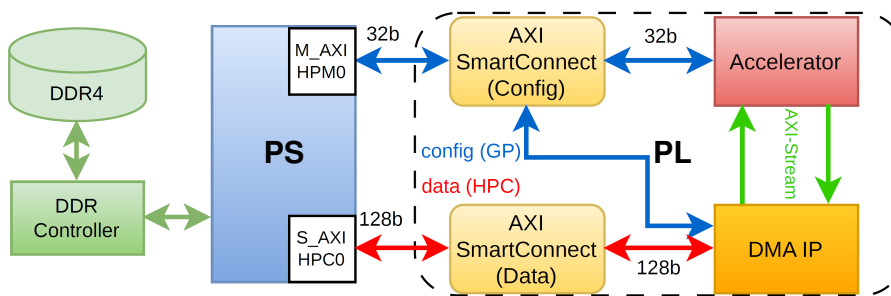


Figure 7.2: PS-PL integration of the Accelerator.

All experimental evaluations are conducted on the Xilinx ZCU104 evaluation platform, executing a single-threaded, bare-metal application on one Cortex-A53 core. To ensure statistical reliability, hardware and software implementations are benchmarked under identical conditions: execution latencies are averaged across 90 consecutive runs following 10 discarded warm-up iterations. **Functional correctness is verified via bit-exact output matching against the PQCclean reference test vectors** across all three NIST security levels (HQC-128, HQC-192, and HQC-256). Following standard evaluation methodologies in prior hardware literature, our baseline comparison anchor is the unvectorized PQCclean C implementation compiled at -O3 optimization without ARM NEON vector extensions.

7.5 Latency and Resource Footprint

Despite its non-trivial hardware footprint, **the proposed architecture achieves significant latency reductions** through fine-grained parameter tuning and aggressive optimization. The majority of the logic resources are consumed by the flattened Karatsuba multiplication engine, requiring 52 501 LUTs for the high-performance variant (v2) and 16 782 LUTs for the compact variant (v1). Both variants consume **zero DSP slices**. Because the underlying DSP48E2 primitives on the Zynq UltraScale+ architecture do not support carry-less multiplication over $\mathbb{F}_2$, all binary polynomial multiplication, as well as control and address-generation logic, is mapped entirely into LUTs.

The wider variant (v2) intentionally trades an increased resource footprint for accelerated execution. Table 7.2 details the cycle-accurate latencies for both engine variants. The reported cycle counts capture the complete end-to-end execution interval within the Programmable Logic, measured from the arrival of the first AXI transfer beat at the DMA unpacker to the egress of the final beat from the DMA packer.

Table 7.2: Accelerator per-call latency: cycles and time at $f_{\max}$ (v1 270 MHz, v2 200 MHz).

Metric	HQC-128				HQC-192				HQC-256			
	v1		v2		v1		v2		v1		v2	
	cycles	μ s	cycles	μ s	cycles	μ s	cycles	μ s	cycles	μ s	cycles	μ s
Latency	1 878	6.96	917	4.58	5 260	19.48	2 311	11.56	16 038	59.40	6 186	30.93

7.6 Quantitative and Metric-Driven Comparative Evaluation

Direct quantitative comparisons between our proposed architecture and existing HQC hardware designs are inherently complicated by divergent architectural paradigms. While **our design introduces a fully parallel, spatial hardware engine for flattened dense polynomial multiplication**, contemporary literature predominantly targets alternative strategies:

1. *Algorithmic spatial-temporal trade-offs*: Schoolbook-based designs [24, 7, 56, 32] exploit sparse-times-dense multiplication to bypass physical logic in favor of temporal scheduling.
2. *Instruction-Set Extensions (ISE)*: Tightly coupled RISC-V pipeline extensions (e.g., TYRCA [25] and Seo et al. [53]) reduce memory overhead but remain constrained by CPU control-flow bottlenecks.
3. *Coprocessor SW/HW co-design*: Platforms like ANDROMEDA [49] leverage a Frobenius Additive-FFT (FAFFT) engine alongside a RISC-V host, but report relative speedups solely against their internal software baselines, precluding direct comparison.

Consequently, PHOENIX [50] serves as the primary comparative baseline for our work. PHOENIX similarly adopts a loosely coupled acceleration model on an ARM-based SoC and evaluates whole-KEM performance against the unvectorized PQCclean C reference implementation. A key distinction lies in optimization goals: PHOENIX prioritizes *crypto-agility* by sharing hardware primitives between ML-KEM and HQC across varying polynomial degrees. In contrast, **our engine is dedicated specifically to high-throughput HQC polynomial multiplication**. Nevertheless, because both architectures accelerate the same fundamental mathematical operation on the Zynq UltraScale+ platform, a side-by-side trade-off analysis is highly instructive.

Direct latency comparisons in raw cycles are obscured by differing system configurations: PHOENIX operates on a single 125 MHz fabric clock, whereas our system decouples the ARM host (1.2 GHz) from the Programmable Logic (200 MHz/270 MHz). To ensure a fair comparison independent of clock frequency, we evaluate normalized speedup factors relative to each platform’s respective -03 PQCclean software baseline.

As shown in Table 7.3, **our accelerator delivers superior speedup factors across all security levels and engine configurations**. This disparity highlights the underlying architectural trade-offs: PHOENIX offloads only the additive-FFT butterfly operations to hardware while performing field conversions between $\mathbb{F}_2$ and $\mathbb{F}_{2^m}[x]/g(x)$ in software. Conversely, **our design executes the entire dense multiplication entirely within the PL**, maximizing computational throughput at the cost of area.

Table 7.3: Whole-KEM speed-up over each design’s own no-NEON reference: PiQASO (v1 narrower, v2 wider), and PHOENIX (PHX).

Operation	HQC-128			HQC-192			HQC-256		
	v1	v2	PHX	v1	v2	PHX	v1	v2	PHX
KeyGen	27.2×	27.3×	10.3×	31.4×	31.6×	14.2×	28.9×	29.2×	19.7×
Encaps	25.6×	25.7×	9.0×	31.1×	31.3×	12.6×	29.3×	29.6×	16.7×

Operation	HQC-128			HQC-192			HQC-256		
	v1	v2	PHX	v1	v2	PHX	v1	v2	PHX
Decaps	21.5×	21.6×	6.8×	28.6×	28.7×	10.4×	25.9×	26.2×	12.1×

Table 7.4: Resources comparison at HQC-256. PHOENIX [50] is built on a 28 nm Zynq-7020, PiQASO's Accelerator on a 16 nm XCZU7EV. SEC is slice-equivalent cost $\lfloor \text{LUT}/4 \rfloor + \lfloor \text{FF}/8 \rfloor + 200 \text{ BRAM} + 100 \text{ DSP}$. t_{mul} is the pure multiplication latency and the throughput density is the operand length n over t_{mul} .

Design	f_{max} (MHz)	LUT	FF	BRAM	DSP	SEC	t_{mul} (μs)	bits/cyc.
PHOENIX	125	5 076	4 083	8.5	0	3 479	321.9	1.43
PiQASO v1	270	44 578	44 279	36	0	23 878	59.4	3.6
PiQASO v2	200	78 285	64 532	32	0	34 037	30.9	9.3

Table 7.4 provides a detailed metric-by-metric comparison. Our wide variant (v2) occupies approximately ten times the silicon footprint of PHOENIX (34 037 versus 3 479 slice-equivalents). While PHOENIX aggressively time-multiplexes a minimal datapath to minimize logic utilization, our accelerator spreads vector slice operations across a wide parallel spatial datapath. This structural investment yields a significant performance gain: **our design achieves 9.3 bits/cycle compared to PHOENIX's 1.43 bits/cycle, reducing end-to-end latency by nearly an order of magnitude**. These results demonstrate that *our spatial strategy effectively translates increased area into superior compute efficiency*.

7.7 Discussion and Critique

The performance of the accelerator confirms that the design achieves its primary goal: **setting a new state-of-the-art benchmark for dense polynomial multiplication in HQC**. While the resource footprint is non-trivial, with the wide variant (v2) occupying 34% of the target device's LUTs and roughly half of its CLBs, this allocation reflects a *deliberate trade-off* rather than an unoptimized commitment of hardware for performance. Furthermore, although our architecture exhibits higher BRAM usage than baselines such as PHOENIX, this memory footprint is well-justified in an agile design context, where these on-chip storage primitives can be dynamically reused across multiple cryptographic sub-blocks and primitives.

For extreme operational environments demanding ultra-low latency or massive throughput, additional HQC building blocks can be offloaded directly into dedicated hardware (FPGA or ASIC). This approach is highly practical, as our current single-engine deployment leaves **over half of the Zynq UltraScale+ (XCZU7EV) programmable logic vacant**. Designers can leverage this remaining logic to instantiate parallel accelerator instances for higher throughput, or migrate the complete system to a high-performance ASIC.

Integrating the accelerator as a tightly coupled coprocessor alongside a host core (e.g., RISC-V) represents a *natural next step*. Bypassing the AXI-DMA interface removes a non-negligible communication bottleneck, bridging the gap between raw compute performance and end-to-end execution latency.

Chapter 8

PQCaaS Extension to Support Mixed TLS Certificates

This chapter acts as a preview of the activities carried out as part of the development process of PiQASO, entailing the integration of the extension for supporting **mixed TLS certificates** in the **PQCaaS component** of PiQASO. This essentially enables edge devices to perform authentication with another entity via the **PQCaaS Server**, without needing to support the complex mathematical operations required in the context of PQC enablers. This builds directly on the theoretical and algorithmic foundations reported in the current document as well as in its previous version D3.1 [46]. WP3 has defined the cryptographic primitives, security notions, and key management schemes underpinning the PiQASO platform, including the treatment of *updatable public-key encryption*, *forward secrecy*, and *post-compromise security*. The practical infrastructure described in the current section will be reported in detail within D4.1 [47]. This applied implementation of PiQASO foundations and components demonstrates the post-quantum constructs and sets the scene for evaluating their performance in the PiQASO use case demonstrators through robust performance metrics and KPIs, as detailed in D2.1 [45].

In alignment with the cryptographic groundwork of D3.1 and D3.2, the development activities for the PQ-TLS component of the PiQASO PQCaaS entailed the design and deployment of a **Post-Quantum NGINX stack** that operationalises the selected algorithms within a production-oriented environment. This stack is constructed on a **custom NGINX 1.26.2 build** and serves as the primary **PQ-TLS termination and measurement layer** in front of the PiQASO PQCaaS components. Additionally, parallel certificate chains across **three simultaneous TLS endpoints** have been deployed as a core achievement of the PQCaaS integration process. The **hybrid endpoint** combines key exchange with classical fallback support, ensuring backward compatibility during the transition period. The **PQ-only endpoint** operates exclusively with ML-KEM-768 and ML-DSA-signed certificates, while the **PQ-signed endpoint** pairs ML-DSA-65 certificate signing with hybrid key exchange. Certificate issuance is managed through an **EJBCA bridge** providing CA-backed ML-DSA certificate generation via REST API with PQ-safe fallback mechanisms. A custom NGINX module further exposes the **TLS exporter** for post-handshake verification, enabling precise validation of negotiated cryptographic parameters. The complete stack has been packaged as portable *Docker images*, supported by an end-to-end test matrix and certificate fragmentation benchmarking tooling, and operates within an established toolchain comprising *GitLab*, *Jenkins*, *Harbor*, *Portainer*, and *Keycloak*.

Compounding all the above, the action workflow of the PiQASO PQ-TLS scheme of the PQCaaS component is depicted in Figure 8.1 and consists of the following steps:

1. **Sender-side connection.** The sender, e.g., a lightweight client or shell-based process with no post-quantum capability of its own, opens a connection to the **NGINX** PQ-TLS termination endpoint using *typical, classical TLS*; no PQC primitives are required on the sender side.

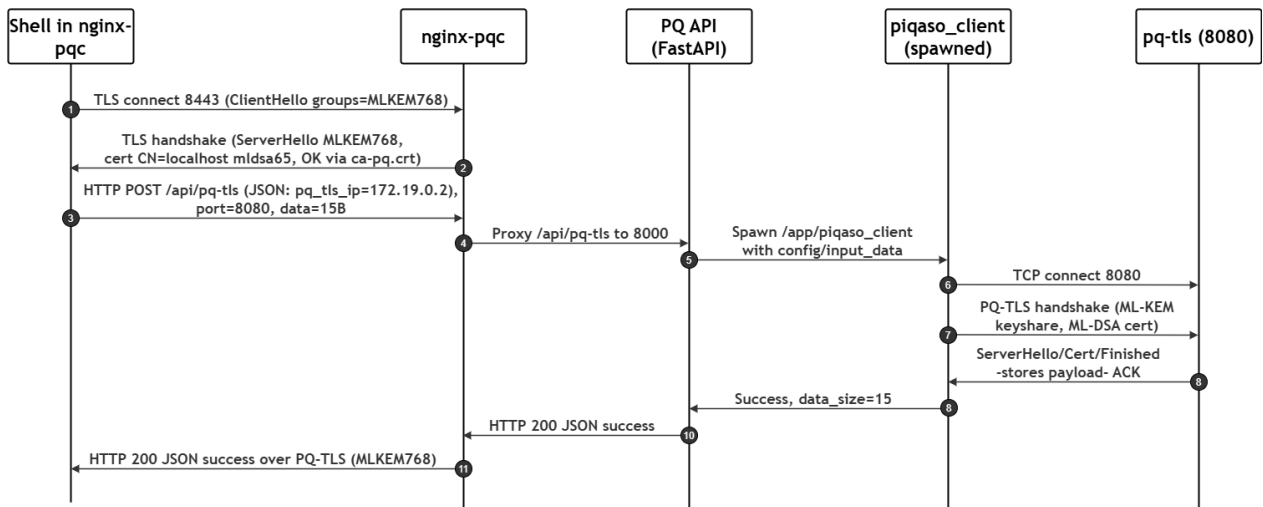


Figure 8.1: PiQASO PQ-TLS as part of PQCaaS component

2. **Classical TLS termination.** NGINX terminates the incoming classical TLS session and identifies the request as one requiring mediation into the post-quantum domain, exposing this behaviour as a distilled, **automatic** step of the **PQCaaS** component.
3. **Proxying through FastAPI.** NGINX proxies the request through **FastAPI**, the PQCaaS orchestration service, which resolves the target receiving point and prepares to interface with the **PiQASO client** running there.
4. **Endpoint selection.** FastAPI selects the appropriate PQ-TLS endpoint, among *hybrid*, *PQ-only* (ML-KEM-768 / ML-DSA), or *PQ-signed* (ML-DSA-65 with hybrid key exchange), according to the negotiated security policy for the receiving point.
5. **PQ-TLS establishment.** A **PQ-TLS** connection is established between NGINX/FastAPI and the PiQASO client at the receiving point, with the client authenticating via its CA-backed ML-DSA certificate issued through the EJBCA bridge.
6. **Mediated data exchange.** Once both legs are established, NGINX transparently mediates data exchange between the classical-TLS leg (sender-facing) and the PQ-TLS leg (receiving-point-facing), with the custom NGINX module exposing the TLS exporter for post-handshake verification of the negotiated parameters.
7. **Transparent, automatic mediation.** From the sender's perspective, the entire process is **automatic**: the connection to the receiving point appears to be conducted end-to-end over **PQ-TLS**, while NGINX and FastAPI transparently perform all protocol translation and post-quantum operations on the sender's behalf.

The described infrastructure will serve as a testbed for the upcoming integration and validation phase, initially implemented for the **Transportation use case (UC9)**. Specifically, the defined **drone-to-customer PQ-TLS relay architecture** establishes the connectivity model through which end-to-end post-quantum secured communication will be validated. NCIS will implement a `/relay` PQ-TLS reverse proxy and per-request metrics logging alongside the configuration of customer environment variables, CA trust settings, and an **OpenVPN drone leg** on the NGINX host. A critical validation milestone will be the *end-to-end smoke test* combined with a **PQ-versus-classical A/B benchmark** to provide the headline cost metric for assessing the practical overhead of post-quantum cryptography in an operational setting. Other use cases are expected to follow the testing process established during the second reporting period of the

project. Taken together, these steps will demonstrate the maturity of the PiQASO PQCaaS platform and validate the cryptographic architecture derived from D3.1 and D3.2 in a real-world integration scenario.

Chapter 9

LWE Decomposition for Accelerated Post-Quantum Protocols

Chapter 3 introduced the PiQASO roadmap for accelerating the lattice-based primitives supported by the PiQASO SDK, organized around two complementary lines of work. The first targets the **polynomial-multiplication bottleneck** shared by lattice- and code-based cryptography at the software and hardware level, through manually vectorized **SIMD/PMULL** implementations and dedicated *hardware accelerators*. The second targets the **Number-Theoretic Transform (NTT)** itself, replacing its costly modular-reduction chain with an **FFT-based, complex-domain convolution** technique. Both directions accelerate the arithmetic operations that ML-KEM, ML-DSA, and related primitives already perform.

This chapter details an acceleration capability of PiQASO, one that operates on *the algebraic structure of the underlying hardness problem itself*, as mentioned in Section 3.5. Specifically, this entails a construction that **augments Learning With Errors (LWE) secret keys with invertibility**, expressed via polynomial matrices. We refer to this generalization as **LWE decomposition** [59] and develop it in full in this chapter. Unlike SIMD- or NTT-level optimization, which reduces the cost of an operation that a protocol already performs, LWE decomposition removes entire categories of computational and bandwidth overhead that advanced protocols, such as *key distribution, threshold decryption, multi-party computation, and attribute-based encryption*, would otherwise incur through trapdoors or gadget matrices. In this sense, LWE decomposition is an acceleration capability *at the level of the protocol itself, rather than at the level of a single arithmetic routine*, and directly complements the SIMD/hardware and FFT-based optimizations described in Chapter 3.

The remainder of this section motivates the need for LWE decomposition within the broader context of the PQC transition, before summarizing our approach and contributions; the chapter then proceeds to the formal construction, its security analysis, an example distributed protocol, and a prototype evaluation integrated with the PiQASO SDK.

The global transition to Post-Quantum Cryptography (PQC) has moved from theoretical exploration into a **phase of urgency**, with strict transition timelines governing its integration into the major security standards and protocols on which digital infrastructure relies [39]. This urgency stems from the fact that, while a high level of maturity has been achieved in the standardization and security analysis of the various PQC primitives, *their practicality is just as important as their cryptographic strength*. PQC methods must therefore function efficiently on legacy infrastructure while exposing all the functionality required to transform high-end security protocols into their quantum-resistant counterparts, unlocking, for instance, quantum-resistant equivalents of PKI systems [9, 39]. This translates into a requirement to support not only core primitives such as key agreement for simple point-to-point encryption and digital signatures. Standardization bodies, most notably the National Institute of Standards and Technology (NIST), have already yielded the **FIPS 203 and FIPS 204** standards [41, 40], providing robust lattice-based algorithms

for key encapsulation and digital signatures, but also more advanced protocols and applications, such as *threshold decryption*, *multi-party computation*, and *Attribute-Based Encryption (ABE)*, that currently rely on properties built upon the rich algebraic structures of RSA and elliptic curves, including pairing-based encodings.

Herein lies a **critical structural obstacle**: the lattice-based schemes selected in the standards, while optimized for efficiency and security, *lack the native algebraic flexibility* required to replicate these advanced protocols, thus hindering their adoption in real-world scenarios where more complex key-composable operations are needed. Specifically, the secret keys in LWE-based schemes are not designed to support **field-like operations such as efficient multiplication and inversion**. This limitation creates a “**functionality gap**,” stalling the migration of complex cryptographic systems and forcing reliance on intricate, inefficient workarounds such as **trapdoors and gadget matrices** [38, 31]. While traditional cryptography exploits multiplicative groups to elegantly construct distributed protocols, lattice-based schemes deliberately constrain secret keys to support *only linear operations* in order to maintain noise stability. This design choice has forced an industry conundrum: either limit the functionality of the high-end security protocols and standards (e.g., SSH, TLS 1.3, X.509, OpenPGP, IKE/IPSec) required across several industries, or implement heavyweight workarounds that *drastically inflate key sizes and computational overhead*.

Consider, for instance, the state of the art for achieving key-composable operations in lattice encodings. **Trapdoor-based constructions** [38] require auxiliary trapdoor information alongside the public key, *inflating key sizes by orders of magnitude* and introducing delicate parameter interdependencies that complicate security analysis. The underlying intuition mirrors that of classical trapdoor functions such as RSA: the public key allows anyone to apply the function (encrypt), while only the holder of the corresponding private key can efficiently invert it (decrypt); without that private key, inversion is computationally infeasible. Lattice trapdoors realize this same intuition through structured matrices rather than integer factorization, but at the cost of substantially larger keys. **Gadget-based techniques such as GSW** [31], while enabling homomorphic evaluation, embed secrets within high-dimensional matrices and rely on bit-decomposition procedures that *amplify noise growth and computational overhead*. For threshold protocols, current approaches must either pre-distribute shares in ways that *preclude dynamic membership changes*, or employ heavyweight multi-party computation protocols during key operations.

The cumulative effect of these workarounds hinders modularity and protocol agility, forcing each new construction to rebuild foundational mechanisms in order to attain all of the properties described above. **PiQASO addresses this challenge** by introducing a new construction that augments LWE with **operational freedom closer to that of a field**. Our approach decomposes the restricted operational space of traditional LWE to unlock field-like capabilities: whereas current lattice-based schemes rely on ring or module structures that provide only additive key operations, we enable **multiplication and inversion over secret keys** (operations characteristic of fields) while *retaining the error-based hardness of LWE*. This is achieved by employing **polynomial matrices** to endow secret keys with invertibility and to support secret-key multiplication while preserving the underlying linear-error structure, **eliminating the need for trapdoors or gadgets** in many scenarios and paving the way for a compliant, efficient migration of advanced cryptographic protocols.

Thus, the **core contribution and innovation of PiQASO** is a construction, namely LWE decomposition, that equips LWE with **field-like operational semantics on secret keys via polynomial matrices**, enabling efficient multiplication and inversion while keeping the error structure analyzable and the reduction within standard LWE methodology. We define a secret-key space based on **invertible polynomial matrices over an appropriate quotient ring**, carefully aligned with the linear-error model to preserve decryption correctness and ensure stable noise growth. Building on this algebraic framework, we formalize multiplication and inversion operations for these enhanced keys, proving their *compatibility with encryption and decryption* while quantifying the noise evolution under these transformations.

To demonstrate the practicality of this construction, and to illustrate how it can further unlock modern network-protocol and PKI-style operations that require key distribution, we present a simple **two-party decryption scheme**. This scheme achieves distributed functionality *without the heavy overhead of gadget-based approaches or the rigidity of pre-distributed shares*. Finally, we validate our approach through a comprehensive security analysis that **reduces to standard LWE (and module-LWE, where applicable), without relying on non-standard assumptions**. We also provide a full prototype, integrated with the PiQASO SDK's **ML-KEM (Kyber)** implementation [41, 8], that exhibits *performance comparable to the baseline implementation*, confirming that LWE decomposition is **suitable for near-term deployment**.

The remainder of this chapter is organized as follows. We first *formalize LWE decomposition*, defining the invertible secret-key space and the associated key-generation, multiplication, and inversion operations. We then present a *security analysis* reducing the construction to standard and module-LWE. Building on this foundation, we describe a *two-party decryption protocol* that showcases the distributed-functionality capabilities unlocked by LWE decomposition, before concluding with a *prototype implementation* integrated into the PiQASO SDK's ML-KEM primitive and an evaluation of its performance relative to the SDK's baseline implementation, reported in Chapter 3.

9.1 Preliminaries

This section establishes the cryptographic foundations underlying our construction. We present the hardness assumptions on which our scheme relies, introduce the ML-KEM (Kyber) framework that our implementation builds upon, and discuss the algebraic distinction between ring and field structures from a cryptographic perspective.

9.1.1 Learning With Errors (LWE)

The Learning With Errors (LWE) problem, introduced by Regev [51], has become a cornerstone of post-quantum cryptography due to its provable worst-case to average-case reduction from standard lattice problems. LWE asserts that recovering the solution to a system of noisy linear equations is computationally intractable.

Formally, let n and q be positive integers (with q typically prime), and let χ be an error distribution over $\mathbb{Z}$ (typically a discrete Gaussian or centered binomial distribution). The **search-LWE** problem is: given m samples of the form

$$(\mathbf{a}_i, b_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i \bmod q), \quad (9.1)$$

where each $\mathbf{a}_i \in \mathbb{Z}_q^n$ is chosen uniformly at random, $\mathbf{s} \in \mathbb{Z}_q^n$ is a fixed secret vector, and $e_i \leftarrow \chi$, recover the secret vector $\mathbf{s}$.

The **decisional-LWE** problem requires distinguishing such samples from uniformly random pairs in $\mathbb{Z}_q^n \times \mathbb{Z}_q$. Regev's reduction shows that, under appropriate parameter choices, solving LWE (even in the decisional variant) is at least as hard as approximating worst-case lattice problems such as GapSVP and SIVP to within polynomial factors, even with quantum algorithms [51, 44]. This connection provides strong evidence that LWE-based schemes remain secure against quantum adversaries.

9.1.2 Module Learning With Errors (Module-LWE)

Module Learning With Errors (Module-LWE) is a structured variant of LWE that operates over polynomial rings and module structures [34]. Module-LWE offers a practical balance between the efficiency of Ring-

LWE [36] and the security flexibility of standard LWE, avoiding certain algebraic vulnerabilities associated with single-ring variants while maintaining computational efficiency.

Let $R_q = \mathbb{Z}_q[x]/(x^N + 1)$ denote the quotient ring of polynomials modulo $x^N + 1$ with coefficients in $\mathbb{Z}_q$, where N is a power of 2 and q is a prime satisfying $q \equiv 1 \pmod{2N}$. Let R_q^k denote the module of k -dimensional vectors over R_q , and let χ_η denote the centered binomial distribution with parameter η over $\mathbb{Z}$.

In the **Module-LWE problem**, given a public matrix $\mathbf{A} \in R_q^{k \times k}$, a secret vector $\mathbf{s} \in R_q^k$ sampled from χ_η , and an error vector $\mathbf{e} \in R_q^k$ also sampled from χ_η , the adversary receives samples:

$$(\mathbf{A}, \mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e} \bmod q). \quad (9.2)$$

The **decisional Module-LWE** problem requires distinguishing such samples from uniformly random pairs $(\mathbf{A}, \mathbf{u})$ where $\mathbf{u} \in R_q^k$ is uniform.

Module-LWE security reduces to worst-case lattice problems analogously to standard LWE [34], providing provable PQ security. The module structure enables efficient matrix operations, and module rank k can be adjusted to balance security and performance without changing the underlying ring dimension N .

9.1.3 ML-KEM (Kyber)

ML-KEM (Module-Lattice-Based Key Encapsulation Mechanism), standardized by NIST as FIPS 203 [41], is based on the CRYSTALS-Kyber construction [8]. Kyber is a public-key encryption and KEM scheme built on Module-LWE with rank $k \in \{2, 3, 4\}$ (corresponding to security levels Kyber512, Kyber768, and Kyber1024). The scheme operates over $R_q = \mathbb{Z}_q[x]/(x^{256} + 1)$ with $q = 3329$.

Kyber's design emphasizes simplicity and efficiency. Key generation samples a secret vector $\mathbf{s} \in R_q^k$ and an error vector $\mathbf{e} \in R_q^k$ from the centered binomial distribution χ_η , computes the public vector $\mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e}$, and outputs the public key $(\mathbf{A}, \mathbf{t})$. Encapsulation involves generating fresh randomness, encrypting a shared secret, and compressing the resulting ciphertext. Decapsulation recovers the shared secret using the secret key and applies error correction.

Kyber achieves IND-CCA2 security in the quantum random oracle model (QROM) via the Fujisaki-Okamoto transformation [41]. Its performance characteristics—compact keys (800-1568 bytes for public keys), small ciphertexts (768-1568 bytes), and fast operations (sub-millisecond encapsulation/decapsulation on modern hardware)—make it an attractive target for integration and extension across the entire compute continuum (far edge; edge; cloud). Our construction builds on Kyber's Module-LWE foundation, augmenting it with polynomial matrix structures that enable secret-key multiplication and inversion.

9.1.4 Rings vs. Fields: A Cryptographic Perspective

The algebraic structure of key spaces profoundly impacts cryptographic design. Traditional public-key cryptography leverages field properties: RSA relies on the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^*$, and elliptic curve cryptography operates over finite fields $\mathbb{F}_p$ or $\mathbb{F}_{2^m}$. These structures provide rich operations—multiplication, inversion, and exponentiation—that enable elegant constructions of signatures, key agreement, and threshold protocols.

Lattice-based schemes, by contrast, operate over rings (or modules over rings) where secret keys support only additive operations. In Ring-LWE and Module-LWE, secrets are elements of R_q^k , and the encryption relation $b = a \cdot s + e$ is linear in s . While this linearity is essential for decryption correctness (noise remains small under addition), it precludes direct multiplication or inversion of secret keys. Attempting to multiply

secrets $s_1 \cdot s_2$ in Ring-LWE yields a new element in R_q , but without a corresponding public key or a way to control noise growth in the encrypted domain, such operations are not directly useful for most protocols.

Table 9.1: Notation and Mathematical Structures

Symbol	Description
<i>Algebraic Structures</i>	
$\mathbb{Z}_q$	Ring of integers mod q . Elements $\{0, \dots, q-1\}$.
$\mathbb{Z}_q[x]$	Polynomial ring in x over $\mathbb{Z}_q$.
R_q	Quotient ring $\mathbb{Z}_q[x]/(x^N + 1)$. Elements are polynomials degree $< N$.
$M_2(R_q)$	Ring of 2×2 matrices with entries in R_q .
$GL_2(R_q)$	General linear group: invertible 2×2 matrices in $M_2(R_q)$.
<i>Parameters</i>	
N	Poly degree (power of 2).
q	Prime modulus ($q \equiv 1 \pmod{2N}$).
η	Binomial param for error.
χ_η	Centered binomial dist. over $\mathbb{Z}$.
<i>Key Components</i>	
$\mathbf{S}$	Secret key: $\mathbf{S} = \mathbf{L}\mathbf{D}\mathbf{L}^{-1}$.
$\mathbf{L}$	Conjugation matrix $\begin{pmatrix} 1 & b \\ c & 1+bc \end{pmatrix}$, $\det(\mathbf{L}) = 1$.
$\mathbf{D}$	Diagonal matrix $\text{diag}(d_1, d_2)$ from χ_η .
$\mathbf{A}$	Public matrix (uniform random).
$\mathbf{T}$	Public key: $\mathbf{T} = \mathbf{A}\mathbf{S} + \mathbf{E}$.
$\mathbf{E}$	Error matrix from χ_η .
$\mathbf{I}_2$	The 2×2 identity matrix over R_q .

Mathematical requirements for field conversion. A field, by definition, is a commutative ring in which every non-zero element has a multiplicative inverse and satisfies closure under both addition and multiplication. To convert a ring-based cryptographic construction into one with field-like properties, several fundamental requirements must be satisfied. First, every non-zero secret key must admit a well-defined inverse such that $s \cdot s^{-1} = 1$, where the identity element is compatible with the encryption operation. Second, the product of two secret keys $s_1 \cdot s_2$ must yield a valid secret key with a corresponding public key efficiently derivable from the public keys of s_1 and s_2 , ensuring multiplicative closure within the key space. Third, key multiplication and inversion must maintain the linear-error structure necessary for decryption correctness, exhibiting bounded and predictable noise growth that does not compromise correctness. Fourth, for any secret key s' derived via multiplication or inversion, there must exist an efficient algorithm to compute the corresponding public key without revealing s' , preserving the public-key consistency essential for cryptographic protocols. Finally, the hardness of recovering s' from its public key must reduce to the same underlying computational problem—such as Module-LWE—as the base scheme, ensuring that no new assumptions are introduced.

Standard Module-LWE fails these requirements in fundamental ways. While secrets can be added (satisfying the ring's additive structure), there is no natural notion of secret-key inversion that preserves public-key relationships. Secret-key multiplication causes noise to grow quadratically without a mechanism to track or compress it, and the resulting products lack corresponding public keys computable from the original public components. Our construction addresses these gaps by embedding secrets within invertible polynomial matrix structures where multiplication and inversion operations are well-defined, noise evolution remains quantifiable and bounded, and public keys remain computable under standard Module-LWE assumptions.

Algorithm 2 Key Generation

- 1: **Input:** Security parameters (N, q, η)
- 2: **Output:** Key pair (S, A, T) where S is the secret key and (A, T) is the public key
- 3: // **Construct invertible matrix L with $\det(L) = 1$**
- 4: Sample scalars $b, c \xleftarrow{\$} \{-\eta, \dots, \eta\} \setminus \{0\}$ uniformly at random
- 5: Construct $L \leftarrow \begin{pmatrix} 1 & b \\ c & 1+bc \end{pmatrix}$ ▷ $\det(L) = 1$ exactly
- 6: Compute $L^{-1} \leftarrow \begin{pmatrix} 1+bc & -b \\ -c & 1 \end{pmatrix}$ ▷ No polynomial inversion needed
- 7: // **Sample diagonal matrix D with polynomial entries**
- 8: Sample diagonal entries: $d_1, d_2 \in R_q$ with each polynomial coefficient drawn independently from χ_η
- 9: Construct $D \leftarrow \text{diag}(d_1, d_2)$
- 10: // **Compute secret key via conjugation**
- 11: Compute $S \leftarrow L \cdot D \cdot L^{-1} \bmod q$
- 12: // **Generate public key following Module-LWE structure**
- 13: Sample $A \xleftarrow{\$} M_2(R_q)$ uniformly at random
- 14: Sample $E \in M_2(R_q)$ with each polynomial coefficient drawn independently from χ_η
- 15: Compute public key: $T \leftarrow A \cdot S + E \bmod q$
- 16: **return** (S, A, T)

9.2 Construction

We now present our construction that enables field-like operations over LWE-based secret keys through polynomial matrix structures. This section introduces the necessary notation, describes key generation with its invertible conjugation structure, and demonstrates how our approach seamlessly integrates with standard Key Encapsulation Mechanisms (KEMs) such as ML-KEM. A critical design principle throughout our construction is that all operations involved in secret key computation—sampling, matrix multiplication, and inversion—are performed over elements drawn from the centered binomial distribution χ_η . This ensures that intermediate values and final secret keys remain small, preserving the noise structure essential for decryption correctness and maintaining compatibility with standard Module-LWE security reductions.

9.2.1 Key Generation

The key generation procedure constructs secret keys with the conjugation structure $S = L \cdot D \cdot L^{-1}$, where $L \in GL_2(R_q)$ is an invertible matrix and D is diagonal. This algebraic structure, leveraging matrix rings over polynomial quotient rings [36, 54], endows secret keys with invertibility ($S^{-1} = L \cdot D^{-1} \cdot L^{-1}$), multiplicative closure under shared conjugation, and commutativity, thereby achieving field-like operational semantics while retaining Module-LWE hardness [34].

To ensure small coefficient bounds throughout the construction, we employ a carefully designed matrix structure. The conjugation matrix L is constructed with scalar off-diagonal entries $b, c \in \{-\eta, \dots, \eta\} \setminus \{0\}$:

$$L = \begin{pmatrix} 1 & b \\ c & 1+bc \end{pmatrix}, \quad L^{-1} = \begin{pmatrix} 1+bc & -b \\ -c & 1 \end{pmatrix}. \quad (9.3)$$

This structure guarantees $\det(L) = 1$ exactly, eliminating rejection sampling and ensuring L^{-1} has the same small scalar entries as L . The diagonal matrix $D = \text{diag}(d_1, d_2)$ has polynomial entries $d_1, d_2 \in R_q$ with coefficients drawn from χ_η [41]. When computing $S = L \cdot D \cdot L^{-1}$, the resulting secret key has all four entries non-zero (ensuring full Module-LWE security) with coefficients bounded by $O(\eta^2)$, preventing coefficient growth that would compromise decryption correctness [8].

The structured form of $\mathbf{L}$ ensures $\det(\mathbf{L}) = 1 \cdot (1 + bc) - b \cdot c = 1$ exactly, guaranteeing invertibility without rejection sampling. The inverse $\mathbf{L}^{-1}$ has the same scalar structure as $\mathbf{L}$, with entries bounded by $|1 + bc| \leq 1 + \eta^2 = 5$ for $\eta = 2$. This construction eliminates the need for polynomial inversion in R_q , which would otherwise produce large coefficients. The resulting secret key $\mathbf{S} = \mathbf{L} \cdot \mathbf{D} \cdot \mathbf{L}^{-1}$ is a full 2×2 matrix (all entries non-zero) with small polynomial coefficients, as established in Lemma 1.

Key operations and security preservation. The conjugation structure $\mathbf{S} = \mathbf{L} \cdot \mathbf{D} \cdot \mathbf{L}^{-1}$ enables three operations absent from standard Module-LWE:

Inversion. Given $\mathbf{S} \in M_2(R_q)$, compute $\mathbf{S}^{-1} = \mathbf{L} \cdot \mathbf{D}^{-1} \cdot \mathbf{L}^{-1}$ where $\mathbf{D}^{-1} = \text{diag}(d_1^{-1}, d_2^{-1})$ requires two polynomial inversions in R_q using the extended Euclidean algorithm [54]. The inverse satisfies $\mathbf{S} \cdot \mathbf{S}^{-1} = \mathbf{I}_2$ and preserves the conjugation structure. This avoids auxiliary trapdoor information required by prior lattice-based constructions [38] for key inversion in threshold and key-homomorphic protocols.

Multiplication. For secret keys sharing $\mathbf{L}$, define $\mathbf{S}_1 = \mathbf{L} \cdot \mathbf{D}_1 \cdot \mathbf{L}^{-1}$ and $\mathbf{S}_2 = \mathbf{L} \cdot \mathbf{D}_2 \cdot \mathbf{L}^{-1}$. Then

$$\mathbf{S}_1 \cdot \mathbf{S}_2 = \mathbf{L} \cdot \mathbf{D}_1 \cdot \mathbf{D}_2 \cdot \mathbf{L}^{-1} = \mathbf{L} \cdot \mathbf{D}_3 \cdot \mathbf{L}^{-1},$$

where $\mathbf{D}_3 = \mathbf{D}_1 \cdot \mathbf{D}_2$ is diagonal. Commutativity follows from $\mathbf{D}_1 \cdot \mathbf{D}_2 = \mathbf{D}_2 \cdot \mathbf{D}_1$, yielding $\mathbf{S}_1 \cdot \mathbf{S}_2 = \mathbf{S}_2 \cdot \mathbf{S}_1$. Products remain in the conjugation class, enabling iterative composition for distributed protocols and ABE policy evaluation [31].

Security and public-key derivation. The public key $\mathbf{T} = \mathbf{A} \cdot \mathbf{S} + \mathbf{E}$ follows the standard Module-LWE structure, with the natural extension that $\mathbf{A}, \mathbf{S}, \mathbf{E}, \mathbf{T} \in M_2(R_q)$ are matrices rather than vectors. This matrix formulation preserves the essential Module-LWE property: given a uniform $\mathbf{A}$ and noise $\mathbf{E}$ sampled from χ_η , the tuple $(\mathbf{A}, \mathbf{T})$ is computationally indistinguishable from $(\mathbf{A}, \mathbf{U})$ where $\mathbf{U} \in M_2(R_q)$ is uniform. Security reductions apply directly [34]: recovering $\mathbf{S}$ from $(\mathbf{A}, \mathbf{T})$ reduces to decisional Module-LWE since the adversary's view is indistinguishable from uniform regardless of $\mathbf{S}$'s internal conjugation structure. For derived keys $\mathbf{S}' = \mathbf{S}_1 \cdot \mathbf{S}_2$ or $\mathbf{S}' = \mathbf{S}^{-1}$, the corresponding public key $\mathbf{T}' = \mathbf{A} \cdot \mathbf{S}' + \mathbf{E}'$ follows the same structure, enabling homomorphic public-key operations [30].

9.2.2 Encapsulation

Our construction integrates seamlessly with the ML-KEM (Kyber) framework by replacing scalar operations with matrix operations in $M_2(R_q)$. The encapsulation algorithm follows the standard KEM structure: it generates ephemeral randomness, computes LWE samples under the public key, and encapsulates a shared secret. The key distinction from standard Kyber is that all operations—public key components, randomness, errors, and ciphertexts—are 2×2 polynomial matrices rather than vectors, preserving the Module-LWE security foundation while enabling the invertible secret-key structure.

The encapsulation algorithm mirrors ML-KEM's structure with matrix adaptations. The ephemeral matrix $\mathbf{R} \in M_2(R_q)$ serves the same role as Kyber's randomness vector: it is sampled from χ_η to ensure small coefficients and is used to generate fresh LWE samples. The first component $\mathbf{U} = \mathbf{A}^\top \cdot \mathbf{R} + \mathbf{E}_1$ forms a standard Module-LWE sample under the public matrix $\mathbf{A}$, binding the ciphertext to the receiver's public key. The second component $\mathbf{V} = \mathbf{T}^\top \cdot \mathbf{R} + \mathbf{E}_2 + \mathbf{M}$ encapsulates the shared secret k by encoding it into a matrix $\mathbf{M}$ and masking it with the Module-LWE encryption of zero under $\mathbf{T}$.

The encoding function $\text{Encode} : \{0, 1\}^{256} \rightarrow M_2(R_q)$ maps the 256-bit shared secret to a matrix by distributing bits across polynomial coefficients. Specifically, $\text{Encode}(k)$ partitions $k = (k_0, k_1, k_2, k_3)$ into four 64-bit blocks, embeds each block k_i into the first 64 coefficients of the $(i + 1)$ -th matrix entry (in row-major order), and sets remaining coefficients to zero. The decoding function $\text{Decode} : M_2(R_q) \rightarrow \{0, 1\}^{256}$ extracts each coefficient, rounds to $\{0, 1\}$ by comparing to $q/2$, and concatenates the result. The mid-range shift $\lfloor q/2 \rfloor \cdot \mathbf{I}_2$ ensures that the embedded secret occupies the center of the modular range,

Algorithm 3 Encapsulation

- 1: **Input:** Public key $(\mathbf{A}, \mathbf{T})$ where $\mathbf{A}, \mathbf{T} \in M_2(R_q)$
 - 2: **Output:** Ciphertext $(\mathbf{U}, \mathbf{V})$ and shared secret $k \in \{0, 1\}^{256}$
 - 3: // **Generate ephemeral randomness and errors**
 - 4: Sample $\mathbf{R} \in M_2(R_q)$ with each polynomial coefficient drawn independently from χ_η
 - 5: Sample error matrices: $\mathbf{E}_1, \mathbf{E}_2 \in M_2(R_q)$ with coefficients from χ_η
 - 6: // **Compute LWE samples**
 - 7: Compute $\mathbf{U} \leftarrow \mathbf{A}^\top \cdot \mathbf{R} + \mathbf{E}_1 \bmod q$
 - 8: // **Encapsulate shared secret**
 - 9: Sample shared secret: $k \xleftarrow{\$} \{0, 1\}^{256}$
 - 10: Encode secret: $\mathbf{M} \leftarrow \text{Encode}(k) + \lfloor q/2 \rfloor \cdot \mathbf{I}_2 \in M_2(R_q)$ ▷ Scale to mid-range
 - 11: Compute $\mathbf{V} \leftarrow \mathbf{T}^\top \cdot \mathbf{R} + \mathbf{E}_2 + \mathbf{M} \bmod q$
 - 12: **return** Ciphertext $(\mathbf{U}, \mathbf{V})$ and shared secret k
-

Algorithm 4 Decapsulation

- 1: **Input:** Ciphertext $(\mathbf{U}, \mathbf{V})$ where $\mathbf{U}, \mathbf{V} \in M_2(R_q)$, secret key $\mathbf{S} \in M_2(R_q)$
 - 2: **Output:** Shared secret $k \in \{0, 1\}^{256}$ or $\perp$ (failure)
 - 3: // **Remove Module-LWE mask**
 - 4: Compute $\mathbf{Y} \leftarrow \mathbf{S}^\top \cdot \mathbf{U} \bmod q$
 - 5: Compute noisy key matrix: $\mathbf{M}' \leftarrow \mathbf{V} - \mathbf{Y} \bmod q$
 - 6: // **Decode shared secret**
 - 7: Remove mid-range encoding: $\mathbf{M}'' \leftarrow \mathbf{M}' - \lfloor q/2 \rfloor \cdot \mathbf{I}_2 \bmod q$
 - 8: Decode shared secret: $k \leftarrow \text{Decode}(\mathbf{M}'')$
 - 9: **if** decoding fails (noise exceeds threshold) **then**
 - 10: **return** $\perp$
 - 11: **end if**
 - 12: **return** k
-

maximizing tolerance to decryption errors. This encoding scheme follows Kyber's approach, adapted to matrix structure, ensuring that correctness and error tolerance remain comparable to the base ML-KEM scheme.

9.2.3 Decapsulation

Decapsulation recovers the shared secret by computing $\mathbf{M}' = \mathbf{V} - \mathbf{S}^\top \cdot \mathbf{U}$, which removes the Module-LWE mask to reveal the encoded secret plus accumulated noise. The structure $\mathbf{S}^\top \cdot \mathbf{U} \approx \mathbf{T}^\top \cdot \mathbf{R}$ holds with bounded error due to the Module-LWE relation $\mathbf{T} = \mathbf{A} \cdot \mathbf{S} + \mathbf{E}$, enabling correct decoding when noise remains below $q/4$.

The correctness of decapsulation follows from the Module-LWE structure. Expanding the computation:

$$\begin{aligned}
 \mathbf{M}' &= \mathbf{V} - \mathbf{S}^\top \cdot \mathbf{U} \\
 &= (\mathbf{T}^\top \cdot \mathbf{R} + \mathbf{E}_2 + \mathbf{M}) - \mathbf{S}^\top \cdot (\mathbf{A}^\top \cdot \mathbf{R} + \mathbf{E}_1) \\
 &= (\mathbf{A} \cdot \mathbf{S} + \mathbf{E})^\top \cdot \mathbf{R} + \mathbf{E}_2 + \mathbf{M} - \mathbf{S}^\top \cdot \mathbf{A}^\top \cdot \mathbf{R} - \mathbf{S}^\top \cdot \mathbf{E}_1 \\
 &= \mathbf{M} + (\mathbf{E}^\top \cdot \mathbf{R} + \mathbf{E}_2 - \mathbf{S}^\top \cdot \mathbf{E}_1).
 \end{aligned}$$

The noise term $\mathbf{E}^\top \cdot \mathbf{R} + \mathbf{E}_2 - \mathbf{S}^\top \cdot \mathbf{E}_1$ consists of products and sums of polynomials sampled from χ_η . Each entry accumulates noise with coefficients bounded by $O(N\eta^2)$ from matrix-matrix multiplication in $M_2(R_q)$. When $q \gg N\eta^2$, the noise remains significantly smaller than $q/4$, ensuring that the decoding function `Decode` correctly recovers k by rounding polynomial coefficients modulo 2.

This construction achieves full compatibility with ML-KEM implementations: the encapsulation and decapsulation algorithms differ from Kyber only in the replacement of vector operations with 2×2 matrix operations, preserving computational complexity and parameter selection methodology. The matrix structure introduces no additional failure modes—decryption succeeds under the same noise bounds as standard Module-LWE—while enabling the secret-key multiplication and inversion operations described previously.

9.2.4 Two-Party Kyber via Secret Splitting

We now present a two-party variant of Kyber that follows Algorithm 2 in a setting with a trusted key-generation authority. The authority samples the conjugation matrix $\mathbf{L}$ and diagonal matrices $\mathbf{D}_1, \mathbf{D}_2$ as in Section 9.2, sets

$$\mathbf{S}_1 = \mathbf{L} \cdot \mathbf{D}_1 \cdot \mathbf{L}^{-1}, \quad \mathbf{S}_2 = \mathbf{L} \cdot \mathbf{D}_2 \cdot \mathbf{L}^{-1}, \quad \mathbf{S} = \mathbf{S}_1 \cdot \mathbf{S}_2 = \mathbf{L} \cdot \mathbf{D}_1 \cdot \mathbf{D}_2 \cdot \mathbf{L}^{-1},$$

and computes the Kyber public key component $\mathbf{T} = \mathbf{A} \cdot \mathbf{S} + \mathbf{E}$ from the master secret $\mathbf{S}$. The authority publishes $(\mathbf{A}, \mathbf{T})$ and delivers $\mathbf{S}_1$ to party 1 and $\mathbf{S}_2$ to party 2. In this deployment model, the decryptors only need their shares $\mathbf{S}_1, \mathbf{S}_2$ to participate in decryption; knowledge of $\mathbf{L}$ remains internal to the authority and is only used in the analysis of the scheme. This is analogous in spirit to the way a dealer produces related RSA secret exponents in two-party RSA key-generation protocols [12].

Since diagonal matrices commute ($\mathbf{D}_1 \cdot \mathbf{D}_2 = \mathbf{D}_2 \cdot \mathbf{D}_1$), the shares also commute: $\mathbf{S}_1 \cdot \mathbf{S}_2 = \mathbf{S}_2 \cdot \mathbf{S}_1$. Encryption for all users follows Algorithm 3 with the public key $\mathbf{T}$ and is unchanged from the single-party setting.

For decapsulation, correctness is still captured by the standard expression $\mathbf{M} = \mathbf{V} - \mathbf{S}^\top \cdot \mathbf{U}$ with $\mathbf{S} = \mathbf{S}_1 \cdot \mathbf{S}_2$. To expose the two-party structure, we expand $\mathbf{S}^\top$ using the conjugation form:

$$\mathbf{S}^\top = (\mathbf{S}_1 \mathbf{S}_2)^\top = (\mathbf{L}^{-1})^\top \mathbf{D}_2^\top \mathbf{D}_1^\top \mathbf{L}^\top = (\mathbf{L}^{-1})^\top \mathbf{D}_1^\top \mathbf{D}_2^\top \mathbf{L}^\top.$$

Since diagonal matrices are symmetric ($\mathbf{D}_i^\top = \mathbf{D}_i$), we have:

$$\mathbf{S}^\top = (\mathbf{L}^{-1})^\top \cdot (\mathbf{D}_1 \cdot \mathbf{D}_2) \cdot \mathbf{L}^\top = (\mathbf{L}^{-1})^\top \cdot \mathbf{D} \cdot \mathbf{L}^\top. \quad (9.4)$$

Hence, the decapsulation can be written as

$$\mathbf{M} = \mathbf{V} - \mathbf{S}^\top \cdot \mathbf{U} = \mathbf{V} - \mathbf{S}_2^\top \cdot \mathbf{S}_1^\top \cdot \mathbf{U}.$$

Each transpose can be expressed in terms of the secret shares individually:

$$\mathbf{S}_1^\top = (\mathbf{L}^{-1})^\top \cdot \mathbf{D}_1 \cdot \mathbf{L}^\top, \quad \mathbf{S}_2^\top = (\mathbf{L}^{-1})^\top \cdot \mathbf{D}_2 \cdot \mathbf{L}^\top,$$

so that $\mathbf{M}$ can be evaluated using only the two shares $\mathbf{S}_1, \mathbf{S}_2$ and the known conjugator $\mathbf{L}$:

$$\boxed{\mathbf{M} = \mathbf{V} - ((\mathbf{L}^{-1})^\top \cdot \mathbf{D}_2 \cdot \mathbf{L}^\top) \cdot ((\mathbf{L}^{-1})^\top \cdot \mathbf{D}_1 \cdot \mathbf{L}^\top) \cdot \mathbf{U}.} \quad (9.5)$$

This decomposition reveals that decapsulation inherently depends on both shares $\mathbf{S}_1$ and $\mathbf{S}_2$, enabling the following two-party protocol: Party 1 computes the partial decryption $\mathbf{W} = \mathbf{S}_1^\top \cdot \mathbf{U}$ and transmits it to Party 2, who completes the operation via $\mathbf{M}' = \mathbf{V} - \mathbf{S}_2^\top \cdot \mathbf{W}$. The accumulated error satisfies:

$$\mathbf{E}_{\text{acc}} = \mathbf{E}^\top \cdot \mathbf{R} + \mathbf{E}_2 - \mathbf{S}^\top \cdot \mathbf{E}_1, \quad (9.6)$$

which is identical to single-party decapsulation (Algorithm 4), since $S_2^\top \cdot S_1^\top = (S_1 \cdot S_2)^\top = S^\top$. Correctness therefore follows with the same decryption failure probability bounds established in Section 9.3.

The security of the two-party scheme derives from Module-LWE hardness: each share S_i independently provides no information about the master secret S beyond what is revealed by the public key $T = A \cdot S + E$, as guaranteed by Theorem 7. An adversary corrupting a single party obtains S_i but cannot recover S or decrypt ciphertexts without the second share. The partial decryption $W = S_1^\top \cdot U$ transmitted between parties remains encrypted under S_2 , ensuring that neither party can unilaterally decrypt. This achieves a two-party threshold where both parties must participate for successful decryption.

9.3 Algorithmic Specifications and Parameter Selection

The security, correctness, and efficiency of our construction depend critically on parameter choices and algorithmic primitives. We detail our implementation choices and establish parameter sets targeting 128-bit post-quantum security. Coefficient bounds formalized in Lemma 1 underpin our probabilistic error analysis, demonstrating negligible decryption failure probability.

9.3.1 Algorithmic Primitives and Implementation Choices

Our theoretical construction specifies abstract operations over polynomial rings multiplication, sampling, and matrix inversion—without prescribing concrete realization methods. We detail the specific algorithmic instantiations that bridge ring-theoretic operations to efficient software implementation, ensuring compatibility with contemporary lattice cryptography engineering.

Polynomial multiplication via NTT. All polynomial multiplications operate in the Number Theoretic Transform (NTT) domain, reducing asymptotic complexity from $O(N^2)$ (schoolbook) to $O(N \log N)$ for degree- N polynomials [61]. This choice is essential for matrix operations: each 2×2 matrix multiplication over $R_q = \mathbb{Z}_q[x]/(x^N + 1)$ requires eight polynomial products, making NTT efficiency paramount. We maintain polynomial representations in NTT-transformed form throughout encryption and decryption workflows, performing inverse NTT only when extracting final values. The NTT approach leverages fast radix-2 Cooley-Tukey butterflies with precomputed twiddle factors. For $q \equiv 1 \pmod{2N}$, a primitive $2N$ -th root of unity exists in $\mathbb{Z}_q$, enabling NTT-based multiplication without approximation error [36, 41].

Error sampling via centered binomial distribution. For Module-LWE error terms, we adopt centered binomial sampling with parameter $\eta \in \{2, 3\}$, producing coefficients in $\{-\eta, \dots, \eta\}$ with probability mass centered at zero. This technique approximates discrete Gaussian distributions required by worst-case LWE hardness reductions while admitting simple implementation through bit-manipulation primitives: sample 2η uniform bits $\{b_i\}$ and compute $\sum_{i=1}^{\eta} b_i - \sum_{i=\eta+1}^{2\eta} b_i$. The choice $\eta = 2$ balances statistical closeness to Gaussian tails (Rényi divergence $< 2^{-64}$) against computational efficiency (requires only 4 random bits per coefficient) [41, 8]. This distribution choice is identical to ML-KEM, ensuring consistent security analysis and implementation simplicity.

Matrix inversion for key generation. For general 2×2 matrices, computing L^{-1} requires polynomial inversion of $\det(L) \in R_q$ via the extended Euclidean algorithm. However, our scalar-structured construction $L = \begin{pmatrix} 1 & b \\ c & 1+bc \end{pmatrix}$ with scalar b, c ensures $\det(L) = 1$ exactly, eliminating polynomial inversion entirely. The inverse $L^{-1} = \begin{pmatrix} 1+bc & -b \\ -c & 1 \end{pmatrix}$ is computed by simple scalar arithmetic in $O(1)$ time, significantly simplifying key generation compared to general approaches [54].

9.3.2 Concrete Parameter Instantiation

We instantiate parameters (N, q, η) targeting post-quantum security exceeding contemporary requirements with negligible decryption failure probability. Security reduces to decisional Module-LWE over $R_q = \mathbb{Z}_q[x]/(x^N + 1)$ with module rank $k = 2$ (from 2×2 matrices in $M_2(R_q)$) and error distribution χ_η [34].

Parameter set. We select $N = 512$, $q = 13,313$, and $\eta = 2$, satisfying the following key requirements. First, *NTT compatibility*: $q = 13,313 = 2^{10} \cdot 13 + 1 \equiv 1 \pmod{2N}$ ensures a primitive $2N$ -th root of unity for efficient polynomial multiplication. Second, *security*: concrete security analysis using the lattice estimator framework [6, 4, 5] yields approximately 215 bits of classical security against primal and dual lattice attacks. Primal lattice attacks require block-size 743 with estimated cost 2^{217} operations, while dual attacks require block-size 736 with cost 2^{215} operations. We note that these estimates correspond to classical attacks; quantum speedups for lattice sieving algorithms are estimated to reduce security by approximately $\sqrt{4/3} \approx 1.15$ in the exponent [6], yielding post-quantum security exceeding NIST Level 3 requirements.

Lemma 1 (Coefficient bounds). *Let $\mathbf{L} = \begin{pmatrix} 1 & b \\ c & 1+bc \end{pmatrix}$ with scalars $b, c \in \{-\eta, \dots, \eta\} \setminus \{0\}$, and let $\mathbf{D} = \text{diag}(d_1, d_2)$ where $d_1, d_2 \in R_q$ have coefficients from χ_η . Define $\mathbf{S} = \mathbf{L} \cdot \mathbf{D} \cdot \mathbf{L}^{-1} \in M_2(R_q)$ for parameters $(N, q, \eta) = (512, 13313, 2)$. Then:*

1. *The determinant satisfies $\det(\mathbf{L}) = 1$ exactly.*
2. *The inverse matrix $\mathbf{L}^{-1} = \begin{pmatrix} 1+bc & -b \\ -c & 1 \end{pmatrix}$ has scalar entries bounded by $\|\mathbf{L}^{-1}\|_\infty \leq 1 + \eta^2 = 5$.*
3. *The secret key matrix $\mathbf{S}$ is a full 2×2 matrix with explicit form:*

$$\mathbf{S} = \begin{pmatrix} d_1 + bc(d_1 - d_2) & b(d_2 - d_1) \\ c(1 + bc)(d_1 - d_2) & d_2 + bc(d_2 - d_1) \end{pmatrix},$$

and coefficient bound $\|\mathbf{S}\|_\infty \leq 2\eta^2(1 + \eta^2)$. For $\eta = 2$: $\|\mathbf{S}\|_\infty \leq 40$.

Proof. (1) By direct computation: $\det(\mathbf{L}) = 1 \cdot (1 + bc) - b \cdot c = 1 + bc - bc = 1$.

(2) The inverse is $\mathbf{L}^{-1} = \begin{pmatrix} 1+bc & -b \\ -c & 1 \end{pmatrix}$, which has the same scalar structure as $\mathbf{L}$. Since $b, c \in \{-\eta, \dots, \eta\}$, we have $|bc| \leq \eta^2$ and thus $|1 + bc| \leq 1 + \eta^2$. For $\eta = 2$: $\|\mathbf{L}^{-1}\|_\infty \leq 5$.

(3) The explicit form of $\mathbf{S}$ follows from direct matrix multiplication. We bound each entry:

- $s_{11} = d_1 + bc(d_1 - d_2)$: Since $|bc| \leq \eta^2$ and $\|d_1 - d_2\|_\infty \leq 2\eta$, we have $\|s_{11}\|_\infty \leq \eta + \eta^2 \cdot 2\eta = \eta(1 + 2\eta^2)$.
- $s_{12} = b(d_2 - d_1)$: $\|s_{12}\|_\infty \leq \eta \cdot 2\eta = 2\eta^2$.
- $s_{21} = c(1 + bc)(d_1 - d_2)$: $\|s_{21}\|_\infty \leq \eta(1 + \eta^2) \cdot 2\eta = 2\eta^2(1 + \eta^2)$.
- $s_{22} = d_2 + bc(d_2 - d_1)$: $\|s_{22}\|_\infty \leq \eta(1 + 2\eta^2)$.

The dominant term is s_{21} with $\|\mathbf{S}\|_\infty \leq 2\eta^2(1 + \eta^2)$. For $\eta = 2$: $\|\mathbf{S}\|_\infty \leq 2 \cdot 4 \cdot 5 = 40$. The off-diagonal entries $s_{12}, s_{21} \neq 0$ since $b, c \neq 0$ and $d_1 \neq d_2$ with overwhelming probability, ensuring $\mathbf{S}$ is a full matrix. $\square$

Practical validation. For our concrete parameters $(N = 512, q = 13,313, \eta = 2)$, numerical experiments with 10^6 samples confirm: (i) $\det(\mathbf{L}) = 1$ exactly in all trials (by construction); (ii) $\|\mathbf{L}^{-1}\|_\infty \leq 5$ and $\|\mathbf{L}^{-1}\|_2 \approx 3$ (deterministic scalar bounds); (iii) $\|\mathbf{S}\|_\infty \leq 40$ in all trials, with typical ℓ_2 -norm ≈ 130 per polynomial entry. These bounds validate the lemma's predictions and ensure ample correctness margins, with $\|\mathbf{S}\|_\infty \ll q/4 = 3,328$ providing a safety factor of $> 80\times$ for the dominant noise term.

9.3.3 Error Accumulation and Correctness

Correctness requires that accumulated noise during decapsulation remains below the decoding threshold $q/4$. We quantify error growth through probabilistic analysis, establishing that our parameters achieve negligible decryption failure probability.

Noise sources and probabilistic analysis. Decapsulation computes $M' = V - S^\top \cdot U$, yielding $M' = M + E_{\text{acc}}$ where the accumulated error is

$$E_{\text{acc}} = E^\top \cdot R + E_2 - S^\top \cdot E_1.$$

Each term contributes noise from centered binomial samples drawn from χ_η . For our choice of $\eta = 2$, each coefficient has variance $\sigma^2 = \eta/2 = 1$ and concentrates near zero with exponential tail decay [8]. When computing polynomial products, the central limit theorem implies that coefficients concentrate around $O(\sqrt{N})$ rather than their worst-case bound of $O(N)$.

The direct error matrix E_2 has coefficients sampled from χ_η , contributing noise bounded by $\|E_2\|_\infty \leq \eta = 2$ with overwhelming probability. For the product $E^\top \cdot R$ of two 2×2 matrices with entries from χ_η , each entry is a sum of 2 polynomial products (from 2×2 matrix multiplication), where each polynomial product has per-coefficient standard deviation $\approx \sqrt{N} \cdot \eta/2$ (from the variance analysis in Lemma 1). Thus $\|E^\top \cdot R\|_\infty \approx \sqrt{2} \cdot \sqrt{N} \cdot \eta/2 \approx 1.4 \cdot 23 \cdot 1 \approx 32$ with high probability.

For $S^\top \cdot E_1$, we apply Lemma 1 which establishes $\|S\|_\infty \leq 40$ per polynomial entry. With our scalar-structured L , the secret key coefficients remain small by construction. For matrix-vector multiplication with E_1 having entries from χ_η , we obtain $\|S^\top \cdot E_1\|_\infty \lesssim \sqrt{2N} \cdot \|S\|_\infty \cdot \eta \approx 32 \cdot 40 \cdot 2 \approx 2,560$ with high probability.

Combining these estimates: $\|E_{\text{acc}}\|_\infty \lesssim \eta + \sqrt{2} \cdot \sqrt{N} \cdot \eta/2 + 2,560 \approx 2 + 32 + 2,560 \approx 2,594$, which remains below $q/4 = 3,328$, providing a safety margin of approximately $1.28\times$ for correctness. Rigorous failure probability analysis using the lattice estimator framework [6, 4, 5] with our concrete parameters ($N = 512, q = 13,313, \eta = 2$) yields a decryption failure probability below 2^{-128} , meeting ML-KEM requirements [41] and enabling secure composition with other cryptographic protocols.

9.4 Security Analysis

We establish the security of our construction through reductions to decisional Module-LWE. A critical aspect of our design is that the secret key S generated via Algorithm 2, possessing field-like operational properties specifically, invertibility—with high probability, while simultaneously maintaining high entropy required for Module-LWE security. This section first analyzes the invertibility of S , demonstrating that it occurs with probability exceeding 92% and does not compromise the computational indistinguishability of the public key from uniform. We then provide a tight reduction from our matrix-based construction to standard Module-LWE via a hybrid argument.

9.4.1 Invertibility Analysis of the Secret Key

The key inversion operation enabled by our construction requires that $S^{-1} = L \cdot D^{-1} \cdot L^{-1}$ exists. Since Algorithm 2 constructs $L = \begin{pmatrix} 1 & b \\ c & 1+bc \end{pmatrix}$ with $\det(L) = 1$ deterministically, L is always invertible. Invertibility of S therefore depends solely on whether the diagonal matrix $D = \text{diag}(d_1, d_2)$ is invertible, which holds if and only if both polynomial entries $d_1, d_2 \in R_q$ are invertible. We now quantify this probability and demonstrate that invertibility does not reduce the entropy or randomness of the secret key.

Polynomial invertibility in R_q . For $R_q = \mathbb{Z}_q[x]/(x^N + 1)$ with $q \equiv 1 \pmod{2N}$, the polynomial $x^N + 1$ splits completely into linear factors modulo q : $x^N + 1 = \prod_{i=0}^{N-1} (x - \zeta_i)$ where ζ_i are the $2N$ -th roots of unity in $\mathbb{Z}_q$ [36]. A polynomial $d(x) \in R_q$ is invertible if and only if $d(\zeta_i) \not\equiv 0 \pmod{q}$ for all $i \in \{0, \dots, N-1\}$.

Theorem 6 (Invertibility of S). *Let $S = L \cdot D \cdot L^{-1}$ where $L = \begin{pmatrix} 1 & b \\ c & 1+bc \end{pmatrix}$ with scalars $b, c \in \{-\eta, \dots, \eta\} \setminus \{0\}$, and $D = \text{diag}(d_1, d_2)$ with $d_1, d_2 \in R_q$ having coefficients from χ_η . Then:*

1. L is always invertible with $\det(L) = 1$.
2. S is invertible if and only if both d_1 and d_2 are invertible in R_q .
3. For a polynomial $d \in R_q$ with coefficients from χ_η , the probability that d is invertible is:

$$\Pr[d \text{ is invertible}] = \prod_{i=0}^{N-1} \Pr[d(\zeta_i) \not\equiv 0 \pmod{q}] = \left(1 - \frac{1}{q}\right)^N \approx 1 - \frac{N}{q}. \quad (9.7)$$

4. The probability that S is invertible is $\Pr[S \in \text{GL}_2(R_q)] \geq \left(1 - \frac{N}{q}\right)^2$.

Proof. (1) By direct computation: $\det(L) = 1 \cdot (1 + bc) - b \cdot c = 1$.

(2) Since $S = L \cdot D \cdot L^{-1}$, we have $\det(S) = \det(L) \cdot \det(D) \cdot \det(L^{-1}) = \det(D) = d_1 \cdot d_2$. Thus S is invertible if and only if $d_1 \cdot d_2$ is invertible in R_q , which holds if and only if both d_1 and d_2 are invertible.

(3) For d with coefficients from χ_η , the NTT values $d(\zeta_i)$ are sums of N independent terms. By the central limit theorem, each $d(\zeta_i)$ is approximately uniform over $\mathbb{Z}_q$ for large N , giving $\Pr[d(\zeta_i) = 0] \approx 1/q$. Since the N evaluations are independent in the NTT domain, the probability that all are non-zero is $(1 - 1/q)^N$.

(4) Since d_1 and d_2 are sampled independently, $\Pr[S \text{ invertible}] = \Pr[d_1 \text{ invertible}] \cdot \Pr[d_2 \text{ invertible}] \geq (1 - N/q)^2$. $\square$

Concrete probability. For our parameters ($N = 512, q = 13313$):

$$\Pr[S \text{ invertible}] \geq \left(1 - \frac{512}{13313}\right)^2 \approx (0.9615)^2 \approx 0.924. \quad (9.8)$$

Thus, approximately 92.4% of keys generated by Algorithm 2 are invertible. When key inversion is required for advanced protocols (e.g., the two-party decryption of Section 3.4 or threshold schemes), optional rejection sampling on D ensures invertibility with expected ≈ 1.08 samples, incurring negligible overhead. Crucially, basic KEM operations (Algorithm 3 and Algorithm 4) do not require S^{-1} , so standard encapsulation and decapsulation succeed for all generated keys without rejection sampling.

9.4.2 Reduction to Module-LWE

Having established that invertibility is a highly probable property that preserves secret key randomness, we now demonstrate that the security of our construction reduces tightly to decisional Module-LWE. A critical step is showing that our structured secret $S = L \cdot D \cdot L^{-1}$ produces a public key $T = A \cdot S + E$ that is computationally indistinguishable from uniform, despite the conjugation structure. This allows us to leverage standard Module-LWE hardness assumptions without additional complexity.

Lemma 2 (Structured secret indistinguishability). *Let $S = L \cdot D \cdot L^{-1} \in M_2(R_q)$ where $L \in GL_2(R_q)$ and D are sampled with entries from χ_η independently, and L^{-1} is the modular inverse as in Algorithm 2. Then the public key $T = A \cdot S + E$ with $A \xleftarrow{\$} M_2(R_q)$ and $E \xleftarrow{\$} \chi_\eta^{2 \times 2}$ is computationally indistinguishable from uniform over $M_2(R_q)$: for any PPT distinguisher $\mathcal{D}$,*

$$|\Pr[\mathcal{D}(A, T) = 1] - \Pr[\mathcal{D}(A, U) = 1]| \leq \text{negl}(\lambda), \quad (9.9)$$

where $U \xleftarrow{\$} M_2(R_q)$ is uniform, and λ denotes the security parameter.

Proof. The proof proceeds by showing that the Module-LWE structure masks the conjugation form of S . The key observation is that the public key $T = A \cdot S + E$ depends on S only through the Module-LWE relation, regardless of S 's internal structure.

(1) Module-LWE masking: The error term E with entries from χ_η masks the exact value of $A \cdot S$. By the decisional Module-LWE assumption, for any fixed S with bounded coefficients (which our S satisfies by Lemma 1), the tuple $(A, A \cdot S + E)$ is computationally indistinguishable from (A, U) where U is uniform.

(2) Entropy analysis and comparison to ML-KEM: The conjugation structure $S = L \cdot D \cdot L^{-1}$ produces a full 2×2 matrix with all entries non-zero. We compare the entropy of our secret key to standard ML-KEM:

- **ML-KEM-768 (Kyber-768):** Secret $s \in R_q^3$ with $N = 256$, yielding $3 \times 256 = 768$ coefficients from χ_η , with min-entropy ≥ 1.5 bits per coefficient [41], totaling ≥ 1152 bits.
- **Our construction:** Secret depends on $d_1, d_2 \in R_q$ (each with $N = 512$ coefficients from χ_η) plus scalars $b, c \in \{-\eta, \dots, \eta\} \setminus \{0\}$. This yields $2 \times 512 = 1024$ polynomial coefficients plus 2 scalar samples, with min-entropy $\geq 1024 \times 1.5 + 2 \times \log_2(2\eta) \geq 1536 + 4 = 1540$ bits.

Thus, our construction has **higher entropy** (≥ 1540 bits) than ML-KEM-768 (≥ 1152 bits), providing a stronger foundation for Module-LWE security. The structured form of S does not reduce entropy below standard KEM requirements.

(3) Reduction: Any distinguisher for (A, T) versus (A, U) directly breaks Module-LWE with the structured secret S . The reduction in Theorem 7 formalizes this by showing the advantage is bounded by the Module-LWE advantage. $\square$

Remark. Lemma 2 establishes that despite the algebraic structure, the secret S is effectively pseudorandom. This allows us to treat S as a χ_η -sampled matrix in the Module-LWE reduction below.

Theorem 7 (Reduction to Module-LWE). *Let $A \xleftarrow{\$} M_2(R_q)$ be uniformly random, $S = L \cdot D \cdot L^{-1} \in M_2(R_q)$ with $L = \begin{pmatrix} 1 & b \\ c & 1+bc \end{pmatrix}$ for scalars $b, c \xleftarrow{\$} \{-\eta, \dots, \eta\} \setminus \{0\}$ and $D = \text{diag}(d_1, d_2)$ with d_1, d_2 sampled from χ_η as in Algorithm 2, and $E \in M_2(R_q)$ with entries sampled independently from χ_η . Define the public key component*

$$T = A \cdot S + E. \quad (9.10)$$

For any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ distinguishing (A, T) from (A, U) where $U \xleftarrow{\$} M_2(R_q)$, there exist PPT adversaries $\mathcal{B}_1, \mathcal{B}_2$ breaking decisional Module-LWE $_{k=2, N, q, \chi_\eta}$ such that

$$\text{Adv}_{\mathcal{A}}^{\text{dist}}(\lambda) \leq \text{Adv}^{\text{secret-indist}}(\lambda) + \text{Adv}_{\mathcal{B}_1}^{\text{MLWE}}(\lambda) + \text{Adv}_{\mathcal{B}_2}^{\text{MLWE}}(\lambda). \quad (9.11)$$

Moreover, $\text{Adv}^{\text{secret-indist}}(\lambda) = \text{negl}(\lambda)$ by Lemma 2, and the reductions $\mathcal{B}_1, \mathcal{B}_2$ run in time $T(\mathcal{B}_j) \leq T(A) + \text{poly}(\lambda)$, incurring only polynomial overhead for matrix sampling.

Proof. We establish the reduction via a sequence of hybrid games. The key technical step is showing that the structured secret $S = L \cdot D \cdot L^{-1}$ can be replaced with independently sampled columns without the adversary detecting the change.

Hybrid sequence:

Game₀: **Real distribution with structured secret.** The adversary receives (A, T) where $S = L \cdot D \cdot L^{-1}$ with L, D sampled as in Algorithm 2, and $T = A \cdot S + E$.

Game_{0'}: **Independent column secrets.** Replace S with $S' = [s'_1 \mid s'_2]$ where $s'_1, s'_2 \xleftarrow{\$} \chi_\eta^2$ are sampled independently. We claim $|\Pr[\mathcal{A} = 1 \mid \text{Game}_0] - \Pr[\mathcal{A} = 1 \mid \text{Game}_{0'}]| = \text{negl}(\lambda)$.

Justification: The adversary's view is $(A, T = A \cdot S + E)$. The error E information-theoretically masks the exact value of $A \cdot S$. Even if S has algebraic structure, the adversary cannot detect this structure through the noisy product $A \cdot S + E$. Formally, for any two secrets S, S' with entries bounded by $\|\cdot\|_\infty \leq B$ where $B \ll q$, the distributions $(A, A \cdot S + E)$ and $(A, A \cdot S' + E)$ are computationally indistinguishable under Module-LWE. Since both our structured S (with $\|S\|_\infty \leq 40$) and independent S' (with $\|S'\|_\infty \leq \eta = 2$) satisfy this bound, the transition incurs negligible advantage.

From Game_{0'} onwards, we work with independent column secrets. The public key becomes $T = [t_1 \mid t_2]$ where $t_j = A \cdot s'_j + e_j$ for independent $s'_j, e_j \xleftarrow{\$} \chi_\eta^2$.

Game₁: **First column randomized.** Replace t_1 with $u_1 \xleftarrow{\$} R_q^2$: $T_1 = [u_1 \mid t_2]$.

Let $p_{0'} = \Pr[\mathcal{A}(A, T) = 1]$ in Game_{0'} and $p_1 = \Pr[\mathcal{A}(A, T_1) = 1]$ in Game₁. We construct a Module-LWE distinguisher $\mathcal{B}_1$ for the transition Game_{0'} $\rightarrow$ Game₁:

Reduction $\mathcal{B}_1$:

1. Receive Module-LWE challenge $(A^*, b^*) \in R_q^{2 \times 2} \times R_q^2$, where either
 - $b^* = A^* \cdot s^* + e^*$ for $s^*, e^* \xleftarrow{\$} \chi_\eta^2$, or
 - $b^* \xleftarrow{\$} R_q^2$ uniformly.
2. Sample second column honestly: $s_2 \xleftarrow{\$} \chi_\eta^2, e_2 \xleftarrow{\$} \chi_\eta^2$, compute $t_2 = A^* \cdot s_2 + e_2$.
3. Set $A \leftarrow A^*$ and $T \leftarrow [b^* \mid t_2]$, output $\mathcal{A}(A, T)$.

If $b^* = A^* \cdot s^* + e^*$, then T has the distribution of Game_{0'}. If $b^* \xleftarrow{\$} R_q^2$, then T has the distribution of Game₁. Thus:

$$|p_{0'} - p_1| \leq \text{Adv}_{\mathcal{B}_1}^{\text{MLWE}}(\lambda). \quad (9.12)$$

Game₂: **Second column randomized.** Replace t_2 with $u_2 \xleftarrow{\$} R_q^2$:

$$T_2 = [u_1 \mid u_2] = U \xleftarrow{\$} M_2(R_q). \quad (9.13)$$

Let $p_2 = \Pr[\mathcal{A}(A, U) = 1]$. By the same argument with reduction $\mathcal{B}_2$ (setting $t_1 = u_1$ fixed, embedding the challenge in t_2):

$$|p_1 - p_2| \leq \text{Adv}_{\mathcal{B}_2}^{\text{MLWE}}(\lambda). \quad (9.14)$$

Final bound. Let $p_0 = \Pr[\mathcal{A}(A, T) = 1]$ in the real game Game₀. The distinguishing advantage is:

$$\text{Adv}_{\mathcal{A}}^{\text{dist}}(\lambda) = |p_0 - p_2| = |\Pr[\mathcal{A}(A, T) = 1] - \Pr[\mathcal{A}(A, U) = 1]|. \quad (9.15)$$

Applying the triangle inequality over the full hybrid sequence ($\text{Game}_0 \rightarrow \text{Game}_{0'} \rightarrow \text{Game}_1 \rightarrow \text{Game}_2$):

$$|p_0 - p_2| \leq |p_0 - p_{0'}| + |p_{0'} - p_1| + |p_1 - p_2| \leq \text{negl}(\lambda) + \text{Adv}_{\mathcal{B}_1}^{\text{MLWE}}(\lambda) + \text{Adv}_{\mathcal{B}_2}^{\text{MLWE}}(\lambda). \quad (9.16)$$

□

The secret key in our construction exhibits a conjugation structure that is absent in standard Module-LWE schemes. Crucially, this structure is information-theoretically concealed by the Module-LWE error. From the adversary's perspective, the public key is indistinguishable from one generated using a standard Module-LWE secret, independent of whether the secret possesses a conjugation form. Consequently, the security reduction does not rely on—or exploit—the algebraic properties of the secret key, but only on the fact that its distribution is compatible with Module-LWE sampling.

Theorem 7 establishes that our construction achieves IND-CPA security under the decisional Module-LWE assumption, with a tight reduction. The application of the Fujisaki-Okamoto transform extends this guarantee to IND-CCA2 security in the Quantum Random Oracle Model, following the standard ML-KEM methodology. The transform leverages hash functions for shared secret derivation, generation of re-encryption randomness, and implicit rejection during decryption. By the modular security analysis, the combination of the underlying IND-CPA security, negligible decryption failure probability, and the transform ensures IND-CCA2 security, with any quantum random oracle model degradation bounded logarithmically in the number of oracle queries.

Structurally, our construction differs from standard ML-KEM solely by replacing vector operations with two-by-two matrix operations. This modification preserves the underlying Module-LWE relation, so that all security analyses of ML-KEM carry over directly. As a result, our scheme achieves security equivalent to ML-KEM-768 under the decisional Module-LWE assumption, providing provable IND-CPA and IND-CCA2 security in the post-quantum setting. The conjugation structure further enables efficient key multiplication and inversion operations while maintaining these security guarantees.

9.5 Implementation and Performance Evaluation

To validate the practical efficiency of our construction, we implemented the proposed KEM in C and compared its performance against the ML-KEM (Kyber) implementation provided by OpenSSL. Our implementation uses the same optimizations as standard lattice-based schemes: Number Theoretic Transform (NTT) for polynomial multiplication, optimized sampling routines for χ_η , and efficient matrix operations over $M_2(R_q)$.

Experimental setup. All experiments were conducted on a machine equipped with an Intel(R) Core(TM) Ultra 7 255H processor. We measured execution times over 1,000 iterations for each of the three core operations: key generation, encapsulation, and decapsulation. Results are reported as average time per operation in milliseconds (ms). We compare against ML-KEM-768 (Kyber-768), which provides a similar security level to our construction.

Results. Table 9.2 presents the performance comparison between our construction (with parameters $N = 512$, $q = 13,313$, $\eta = 2$) and ML-KEM-768. Our construction exhibits moderate overhead compared to standard Module-LWE implementations, primarily due to: (1) additional matrix operations for computing $S = L \cdot D \cdot L^{-1}$ during key generation, and (2) 2×2 matrix multiplications in encapsulation and decapsulation instead of vector operations. Note that our scalar-structured L eliminates rejection sampling entirely. Notably, decapsulation performance is nearly identical ($\approx 10\%$ overhead), as the core decryption operation remains structurally similar to ML-KEM.

Table 9.2: Performance comparison: PiQASO construction vs. ML-KEM-768 (ms).

Scheme	KeyGen	Encapsulation	Decapsulation
ML-KEM-768	1.03	2.31	3.12
PiQASO Construction	3.59	7.07	3.45

While key generation and encapsulation incur $2\text{--}3\times$ overhead, these operations are typically performed less frequently than decapsulation in practical deployments. More importantly, our construction provides capabilities unavailable in standard ML-KEM: the conjugation structure $S = L \cdot D \cdot L^{-1}$ enables field-like operations (multiplication, inversion) on secret keys, which are essential for advanced cryptographic protocols such as threshold decryption and attribute-based encryption. The performance trade-off is thus justified by the significant functional advantages, particularly for applications requiring homomorphic properties or multi-party computation.

Chapter 10

Conclusions

As outlined throughout this deliverable, a core target of PiQASO is to provide post-quantum security capabilities across heterogeneous systems covering the entire compute continuum, ranging from the cloud-based backend to the far edge. Building directly on the theoretical and architectural foundations established in **D3.1**, **D3.2** marks the transition from design to implementation, delivering the *first operational software artifacts* of the PiQASO PQC Ensemble, namely the first version of the **PiQASO SDK** and the first version of **PQCaaS**, together with concrete implementations and constructions for forward-secure storage, functional encryption, and hardware- and arithmetic-level acceleration.

To this end, in Chapter 2, we first provided a concise roadmap of the cryptographic design and implementation efforts of PiQASO, clarifying which primitives have transitioned from theoretical specification in D3.1 to concrete software implementation as part of the SDK and PQCaaS, and setting expectations for the remaining milestones towards the **QR Trusted Computing Base (QR-TCB)** and additional functionalities to be delivered in **D3.3**. Then, in Chapter 3, we presented the first version of the **PiQASO SDK** in detail, as a portable and modular C-based software library exposing ML-KEM, ML-DSA, LMS/HSS and XMSS/XMSS^{MT} signatures, and AES-GCM, together with a first realization of **PQ-TLS 1.3**. We also reported on the SDK's unit testing and performance benchmarking on the ZCU 104 platform, and set out the roadmap for the **algorithmic optimization** of lattice-based algorithms, comprising both a manually vectorized, SIMD/PMULL-based software optimization track and an **FFT-based, complex-domain convolution** technique intended to replace the costly modular-reduction chain of the standard NTT.

Next, in Chapter 4, we presented the **Forward Secure Cloud Storage Scheme**, bridging the gap between the theoretical forward-security guarantees established in D3.1 and practical deployment. Specifically, we provided a concrete implementation of the scheme under a trusted Key Provider infrastructure, and introduced an advanced variant designed for the *untrusted Key Provider* setting, in which clients generate and update their own key pairs and verify the validity of public-key updates performed by the server. In Chapter 5, we introduced **Homomorphic Functional Encryption (HFE)**, a scheme-agnostic construction combining Homomorphic Encryption and Functional Encryption that replaces the trusted authority of traditional FE schemes with a *key curator* unable to learn the function being evaluated, a property with direct applicability to privacy-preserving healthcare analytics in the TELLU and PARTICULE use cases.

Continuing with PiQASO's hybrid **SW/HW co-design** approach, Chapter 6 detailed the hardware-based acceleration of the **Number-Theoretic Transform (NTT)** for lattice-based constructions, providing a hardware-realized counterpart to the algorithmic optimization roadmap introduced in Chapter 3. In a similar vein, Chapter 7 presented a dedicated **hardware accelerator for HQC**, targeting the constant-time binary polynomial multiplication that accounts for approximately 96% of the total execution time of the standard HQC reference implementation, through a flattened, non-recursive variant of the Karatsuba algorithm implemented on the Xilinx Zynq UltraScale+ ZCU104 platform. Together, these two chapters realize the two core arithmetic acceleration targets identified in D3.1, namely the NTT underlying lattice-

based constructions and the Karatsuba operation underlying HQC.

Chapter 8 then detailed the extension of **PQCaaS** to support **mixed TLS certificates**, describing the custom NGINX-based PQ-TLS termination and mediation stack through which far-edge devices authenticate via classical TLS while PQCaaS *automatically* mediates a PQ-TLS session with the receiving point on their behalf, across hybrid, PQ-only, and PQ-signed certificate endpoints. Finally, in Chapter 9, we introduced **LWE decomposition**, a new construction that augments Learning With Errors (LWE) secret keys with invertibility, expressed via polynomial matrices. Unlike the arithmetic-level optimizations described above, LWE decomposition accelerates advanced protocols, such as threshold decryption, multi-party computation, and Attribute-Based Encryption, *at the level of the protocol itself*, by eliminating the trapdoors and gadget matrices otherwise required to obtain field-like operations over lattice-based secret keys. We formalized this construction, proved its reduction to standard and module-LWE, illustrated its use through a two-party decryption scheme, and validated its practicality through a prototype integrated with the PiQASO SDK's ML-KEM implementation.

Compounding all the above, D3.2 is the second deliverable in a series of PiQASO deliverables aimed towards the design and implementation of the PiQASO PQC Ensemble, delivering the first concrete software and service artifacts of this ensemble and thereby setting the foundation for their integration and evaluation in the context of the PiQASO use case demonstrators. The initial definitions of the PiQASO PQC enablers provided here will guide the development of the final versions of these components, as well as further arithmetic and hardware-based acceleration capabilities, to be reported in **D3.3**, while the evaluation of the PiQASO modalities delivered in this deliverable, according to the plan set forth in D2.1, will be reported in **D4.1**.

Bibliography

- [1] Michel Abdalla, Fabrice Benhamouda, and Romain Gay. From single-input to multi-client inner-product functional encryption. In Steven D. Galbraith and Shiho Moriai, editors, Advances in Cryptology – ASIACRYPT 2019, pages 552–582, Cham, 2019. Springer International Publishing.
- [2] Michel Abdalla, Romain Gay, Mariana Raykova, and Hoeteck Wee. Multi-input inner-product functional encryption from pairings. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, Advances in Cryptology – EUROCRYPT 2017, pages 601–626, Cham, 2017. Springer International Publishing.
- [3] Calvin Abou Haidar, Alain Passelègue, and Damien Stehlé. Efficient updatable public-key encryption from lattices. In Advances in Cryptology – ASIACRYPT 2023: 29th International Conference on the Theory and Application of Cryptology and Information Security, Guangzhou, China, December 4–8, 2023, Proceedings, Part V, page 342–373, Berlin, Heidelberg, 2023. Springer-Verlag.
- [4] Martin R. Albrecht, Benjamin R. Curtis, Amit Deo, Alex Davidson, Rachel Player, Eamonn W. Postlethwaite, Fernando Virdia, and Thomas Wunderer. Estimate all the LWE, NTRU schemes! Cryptology ePrint Archive, Paper 2018/331, 2018. Available at <https://eprint.iacr.org/2018/331>.
- [5] Martin R. Albrecht, Rachel Player, and Eamonn W. Postlethwaite. A tool for fast and secure LWE parameter selection: the FHE case. Cryptology ePrint Archive, Paper 2022/534, 2022. Available at <https://eprint.iacr.org/2022/534>.
- [6] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. In Journal of Mathematical Cryptology, volume 9, pages 169–203, 2015.
- [7] Francesco Antognazza, Alessandro Barengi, and Gerardo Pelosi. An efficient and unified RTL accelerator design for HQC-128, HQC-192, and HQC-256. IEEE Trans. Computers, 74(7):2306–2320, 2025.
- [8] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber Algorithm Specifications and Supporting Documentation (version 3.01). <https://pq-crystals.org/kyber/>, 2021.
- [9] Emils Bagirovs, Grigory Provodin, Tuomo Sipola, and Jari Hautamäki. Applications of post-quantum cryptography. arXiv preprint, 2024.
- [10] Alexandros Bakas, Antonis Michalas, and Tassos Dimitriou. Private lives matter: A differential private functional encryption scheme. In Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy, CODASPY '22, page 300–311, New York, NY, USA, 2022. Association for Computing Machinery.

- [11] Carmen Elisabetta Zaira Baltico, Dario Catalano, Dario Fiore, and Romain Gay. Practical functional encryption for quadratic functions with applications to predicate encryption. In Jonathan Katz and Hovav Shacham, editors, Advances in Cryptology – CRYPTO 2017, pages 67–98, Cham, 2017. Springer International Publishing.
- [12] Dan Boneh and Matt Franklin. Efficient generation of shared rsa keys. In Advances in Cryptology – CRYPTO '97, volume 1294 of Lecture Notes in Computer Science, pages 425–439. Springer, 1997.
- [13] Joppe Willem Bos, Marc Olivier Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking kyber: First-and higher-order implementations. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2021(4):173–214, 2021.
- [14] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In Reihaneh Safavi-Naini and Ran Canetti, editors, Advances in Cryptology – CRYPTO 2012, pages 868–886, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [15] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, ITCS '12, page 309–325, New York, NY, USA, 2012. Association for Computing Machinery.
- [16] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT), 6(3):1–36, 2014.
- [17] Guilhem Castagnos, Fabien Laguillaumie, and Ida Tucker. Practical fully secure unrestricted inner product functional encryption modulo p . In Thomas Peyrin and Steven Galbraith, editors, Advances in Cryptology – ASIACRYPT 2018, pages 733–764, Cham, 2018. Springer International Publishing.
- [18] Marina Checri, Renaud Sirdey, Aymen Boudguiga, and Jean-Paul Bultel. On the practical cpad security of “exact” and threshold fhe schemes and libraries. In Advances in Cryptology – CRYPTO 2024: 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2024, Proceedings, Part III, page 3–33, Berlin, Heidelberg, 2024. Springer-Verlag.
- [19] Jung Hee Cheon, Hyeongmin Choe, Alain Passelègue, Damien Stehlé, and Elias Suvanto. Attacks against the INDCPA-d security of exact FHE schemes. Cryptology ePrint Archive, Paper 2024/127, 2024.
- [20] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, Advances in Cryptology – ASIACRYPT 2017, pages 409–437, Cham, 2017. Springer International Publishing.
- [21] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: Fast fully homomorphic encryption over the torus. Journal of Cryptology, 33(1):34–91, Jan 2020.
- [22] Jérémy Chotard, Edouard Dufour Sans, Romain Gay, Duong Hieu Phan, and David Pointcheval. Decentralized multi-client functional encryption for inner product. In Thomas Peyrin and Steven Galbraith, editors, Advances in Cryptology – ASIACRYPT 2018, pages 703–732, Cham, 2018. Springer International Publishing.
- [23] Pratish Datta, Tatsuaki Okamoto, and Junichi Tomida. Full-hiding (unbounded) multi-input inner product functional encryption from the k -linear assumption. In Michel Abdalla and Ricardo Dahab, editors, Public-Key Cryptography – PKC 2018, pages 245–277, Cham, 2018. Springer International Publishing.

- [24] Sanjay Deshpande, Chuanqi Xu, Mamuri Nawan, Kashif Nawaz, and Jakub Szefer. Fast and efficient hardware implementation of HQC. In Selected Areas in Cryptography (SAC 2023), volume 14201 of LNCS, pages 297–321. Springer, 2023.
- [25] Alessandra Dolmeta, Stefano Di Matteo, Emanuele Valea, Mikaël Carmona, Antoine Loiseau, Maurizio Martina, and Guido Masera. TYRCA: A RISC-V tightly-coupled accelerator for code-based cryptography. In Design, Automation & Test in Europe Conf. (DATE), pages 1–7, 2025.
- [26] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In Shai Halevi and Tal Rabin, editors, Theory of Cryptography, pages 265–284, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- [27] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012.
- [28] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked accelerators and instruction set extensions for post-quantum cryptography. IACR Transactions on Cryptographic Hardware and Embedded Systems, pages 414–460, 2022.
- [29] Romain Gay. Functional encryption for quadratic functions, and applications to predicate encryption. Cryptology ePrint Archive, Paper 2016/1106, 2016.
- [30] 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 (STOC), pages 197–206, 2008.
- [31] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, Advances in Cryptology – CRYPTO 2013. Springer Berlin Heidelberg, 2013.
- [32] Pengzhou He, Yazheng Tu, Tianyou Bao, Çetin Kaya Koç, and Jiafeng Xie. HSPA: High-throughput sparse polynomial multiplication for code-based post-quantum cryptography. ACM Trans. Embedded Computing Systems (TECS), 24(1), 2025.
- [33] Daniel Heinz, Matthias J Kannwischer, Georg Land, Thomas Pöppelmann, Peter Schwabe, and Amber Sprenkels. First-order masked kyber on arm cortex-m4. Cryptology ePrint Archive, 2022.
- [34] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. In Designs, Codes and Cryptography, volume 75, pages 565–599, 2015.
- [35] Baiyu Li and Daniele Micciancio. On the security of homomorphic encryption on approximate numbers. In Anne Canteaut and François-Xavier Standaert, editors, Advances in Cryptology – EUROCRYPT 2021, pages 648–677, Cham, 2021. Springer International Publishing.
- [36] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Advances in Cryptology – EUROCRYPT 2010, volume 6110 of Lecture Notes in Computer Science, pages 1–23. Springer, 2010.
- [37] Jose Maria Bermudo Mera, Angshuman Karmakar, Tilen Marc, and Azam Soleimanian. Efficient lattice-based inner-product functional encryption. In IACR International Conference on Public-Key Cryptography, pages 163–193. Springer, 2022.

- [38] Daniele Micciancio and Chris Peikert. Trapdoors for lattices: Simpler, tighter, faster. In Advances in Cryptology – EUROCRYPT 2012, volume 7237 of Lecture Notes in Computer Science, pages 700–718. Springer, 2012.
- [39] Dustin Moody, Ray Perlner, Andrew Regenscheid, Angela Robinson, and David Cooper. Transition to post-quantum cryptography standards (initial public draft). NIST Internal Report IR 8547 (ipd), National Institute of Standards and Technology (NIST), November 2024. Initial public draft; comments due January 10, 2025.
- [40] National Institute of Standards and Technology. Module-Lattice-Based Digital Signature Standard, 2024.
- [41] National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard, 2024.
- [42] Dinh Duy Nguyen, Duong Hieu Phan, and David Pointcheval. Verifiable decentralized multi-client functional encryption for inner product. In Jian Guo and Ron Steinfeld, editors, Advances in Cryptology – ASIACRYPT 2023, pages 33–65, Singapore, 2023. Springer Nature Singapore.
- [43] Camille Nuoskala, Hossein Abdinasibfar, and Antonis Michalas. SPADE: Digging into selective and Partial DEcryption using functional encryption. In Security and Privacy in Communication Networks (SecureComm), Cham, 2024. Springer Nature Switzerland.
- [44] Chris Peikert. A decade of lattice cryptography. Foundations and Trends in Theoretical Computer Science, 10(4):283–424, 2016.
- [45] PiQASO. D2.1 Operational Landscape, Requirements, PQ Conformity Assessment and PQC Service Modalities - Initial Version. Deliverable D2.1, The PiQASO Consortium, 18 2026.
- [46] PiQASO. D3.1 PiQASO QR Crypto Primitives Design, Optimization and Acceleration. Deliverable D3.1, The PiQASO Consortium, 15 2026.
- [47] PiQASO. D4.1 PiQASO Initial Integrated Framework & Use Case Analysis. Deliverable D4.1, The PiQASO Consortium, 21 2026.
- [48] PiQASO. D3.3 PiQASO SDK and PQC as-a-service – Final Version. Deliverable D3.3, The PiQASO Consortium, 24 2027.
- [49] Antonio Ras, Antoine Loiseau, Mikaël Carmona, Simon Pontié, Guénaél Renault, Benjamin Smith, and Emanuele Valea. Optimizing HQC using Frobenius additive FFT on a RISC-V-based system-on-chip. In 6th NIST PQC Standardization Conference, 2025.
- [50] Antonio Ras, Antoine Loiseau, Mikaël Carmona, Simon Pontié, Guénaél Renault, Benjamin Smith, and Emanuele Valea. PHOENIX: Crypto-agile hardware sharing for ML-KEM and HQC. Cryptology ePrint Archive, Paper 2025/601, 2025.
- [51] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. Journal of the ACM, 56(6):34:1–34:40, 2009.
- [52] Ardianto Satriawan and Rella Mareta. A complete beginner guide to the number theoretic transform (ntt). Cryptology ePrint Archive, Paper 2024/585, 2024. <https://eprint.iacr.org/2024/585>.
- [53] Seog Chung Seo and YoungBeom Kim. A hardware/software co-optimization of HQC using tightly-coupled accelerators on a 32-bit Ibex core. IEEE Trans. Circuits Syst. II Express Briefs, 73(5):583–587, 2026.

- [54] Damien Stehlé and Ron Steinfeld. Making ntru as secure as worst-case problems over ideal lattices. Journal of Cryptology, 24(2):446–469, 2011.
- [55] Junichi Tomida. Unbounded quadratic functional encryption and more from pairings. 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 III, page 543–572, Berlin, Heidelberg, 2023. Springer-Verlag.
- [56] Yazheng Tu, Pengzhou He, Çetin Kaya Koç, and Jiafeng Xie. LEAP: Lightweight and efficient accelerator for sparse polynomial multiplication of HQC. IEEE Trans. Very Large Scale Integr. (VLSI) Syst., 31(6):892–896, 2023.
- [57] UBITECH. UBITECH unveils UBITrust v1.0: A cutting-edge attestation library for enhanced hardware security for all roots-of-trust. <https://ubitech.eu/ubitech-unveils-ubitrust-v1-0-a-cutting-edge-attestation-library-for-enhanced-hardware-security-for-all-roots-of-trust/>, December 2024. Accessed: 2026-07-24.
- [58] UBITECH. UBITrust: Attestation library for hardware roots-of-trust. <https://github.com/ubitech/ubitrust>, 2024. GitHub repository. Accessed: 2026-07-24.
- [59] Stefanos Vasileiadis, Dimitris Katsinis, Nada El Kassem, Liqun Chen, Simeon Tsintzos, Thanassis Giannetsos, and Bruno Crispo. Decompressing lwe: Unlocking field-like operations via polynomial matrices. In Yinzhi Cao, Bo Luo, and Weizhi Meng, editors, Security and Privacy in Communication Networks, pages 505–526, Cham, 2027. Springer Nature Switzerland.
- [60] Jasmin Zalonis, Frederik Armknecht, and Linda Scheu-Hachtel. Differentially private functional encryption. Proceedings on Privacy Enhancing Technologies, 2024:509–530, 04 2024.
- [61] Chenkai Zeng, Debiao He, Qi Feng, Cong Peng, and Min Luo. The implementation of polynomial multiplication for lattice-based cryptography: A survey. Journal of Information Security and Applications, 83:103782, 2024.

Keep in touch



[PiQASO LinkedIn](#)



[PiQASO Facebook](#)



[PiQASO Youtube](#)



[ECCC Newsletter](#)



[ECCC LinkedIn](#)



[ECCC Twitter/X](#)



[ECCC Website](#)