École Nationale Supérieure d'Arts et Métiers
Casablanca

Master Big Data & Internet of Things

Post-Quantum Cryptography for Lightweight IoT Devices

A Comprehensive Reference for Academic Researchers

Author:
Abdessamad JAOUAD
M2 Big Data & IoT

Supervisor:
Prof. Ibrahim GUELZIM
ENSAM Casablanca

Academic Year 2024–2025

Version 1.0 – December 2024

Abstract

The advent of large-scale quantum computers poses an existential threat to the cryptographic foundations of modern digital infrastructure. Current public-key cryptographic systems—including RSA and Elliptic Curve Cryptography—rely on mathematical problems that quantum computers can solve efficiently using Shor's algorithm. This vulnerability extends to the billions of Internet of Things (IoT) devices that form the backbone of smart cities, industrial systems, and critical infrastructure.

This comprehensive reference document surveys the emerging field of Post-Quantum Cryptography (PQC) with a specific focus on implementations suitable for resource-constrained IoT devices. We provide an in-depth analysis of the algorithms standardized by the National Institute of Standards and Technology (NIST) in August 2024: ML-KEM (Kyber) for key encapsulation, ML-DSA (Dilithium) and SLH-DSA (SPHINCS+) for digital signatures, along with the forthcoming Falcon signature scheme.

The document offers detailed benchmark data from real hardware implementations, particularly on ARM Cortex-M series microcontrollers commonly found in IoT applications. We catalog available software libraries, analyze memory and computational requirements across different IoT device categories, and provide practical guidance for migration from classical to post-quantum cryptographic systems.

This work serves as a definitive reference for academic researchers, graduate students, and practitioners seeking to understand and implement post-quantum cryptography in constrained environments.

Keywords: Post-Quantum Cryptography, Internet of Things, Lattice-Based Cryptography, Key Encapsulation, Digital Signatures, ARM Cortex-M, Embedded Systems, NIST Standards, ML-KEM, ML-DSA

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Professor Ibrahim GUELZIM, for his invaluable guidance, continuous support, and expert insights throughout this research project. His deep knowledge of cryptography and IoT systems has been instrumental in shaping this work.

I am grateful to the faculty and staff of ENSAM Casablanca, particularly the Big Data & IoT program, for providing an excellent academic environment and the resources necessary to conduct this research.

Special thanks to the open-source community, particularly the developers of liboqs, PQClean, and pqm4 projects, whose implementations and benchmark data have been invaluable resources for this survey.

Finally, I thank my family and friends for their unwavering support and encouragement throughout my academic journey.

Abdessamad JAOUAD
Casablanca, December 2024

Contents

Abstract **2**

Acknowledgments **3**

List of Figures **13**

List of Tables **17**

List of Algorithms **18**

List of Acronyms **18**

1 Introduction **19**
1.1 The Quantum Threat to Modern Cryptography 19
1.1.1 Current State of Quantum Computing 20
1.1.2 Impact on IoT Security 20
1.2 Post-Quantum Cryptography: An Overview 21
1.2.1 Mathematical Foundations 21
1.2.2 Comparison of PQC Families 23
1.3 NIST Post-Quantum Standardization 24
1.3.1 Timeline and Process 24
1.3.2 Standardized Algorithms (August 2024) 24
1.3.3 Algorithms Under Standardization 25
1.4 Scope and Organization of This Document 25
1.5 How to Use This Reference 26
1.6 Chapter Summary 26

2 Mathematical Foundations **27**
2.1 Algebraic Preliminaries 27
2.1.1 Groups, Rings, and Fields 27
2.1.2 Polynomial Rings 28
2.1.3 The Number Theoretic Transform (NTT) 29
2.2 Lattice Theory 30

2.2.1 Basic Definitions . . . 30
2.2.2 Successive Minima and the Gaussian Heuristic . . . 31
2.2.3 Important Lattice Constructions . . . 32
2.3 Computational Lattice Problems . . . 32
2.3.1 The Shortest Vector Problem (SVP) . . . 32
2.3.2 The Closest Vector Problem (CVP) . . . 33
2.3.3 Learning With Errors (LWE) . . . 33
2.3.4 Ring-LWE and Module-LWE . . . 34
2.3.5 Short Integer Solution (SIS) . . . 35
2.4 Algorithms for Lattice Problems . . . 35
2.4.1 Lattice Reduction . . . 35
2.4.2 Sieving Algorithms . . . 36
2.4.3 Security Estimation . . . 36
2.5 Hash Functions in PQC . . . 36
2.5.1 SHA-3 and SHAKE . . . 36
2.5.2 Hashing in Embedded Systems . . . 37
2.6 Chapter Summary . . . 37

3 Key Encapsulation Mechanisms: ML-KEM (Kyber) 39
3.1 Introduction to Key Encapsulation . . . 39
3.1.1 From Key Exchange to Key Encapsulation . . . 39
3.1.2 KEMs vs. Public-Key Encryption . . . 40
3.2 ML-KEM Overview . . . 40
3.2.1 Design Philosophy . . . 40
3.2.2 Parameter Sets . . . 41
3.3 ML-KEM Algorithms in Detail . . . 41
3.3.1 Key Generation . . . 41
3.3.2 Encapsulation . . . 42
3.3.3 Decapsulation . . . 43
3.3.4 Core Operations . . . 43
3.4 Security Analysis . . . 44
3.4.1 Hardness Assumptions . . . 44
3.4.2 Concrete Security Estimates . . . 45
3.4.3 Known Attack Vectors . . . 45
3.5 Implementation Considerations . . . 45
3.5.1 Memory Layout . . . 45
3.5.2 Optimization Strategies . . . 46
3.5.3 Constant-Time Implementation . . . 47
3.6 Performance Benchmarks . . . 47

3.6.1 ARM Cortex-M4 Benchmarks . . . 47
3.6.2 Comparison with Classical Algorithms . . . 48
3.6.3 Desktop/Server Benchmarks . . . 49
3.6.4 Hashing Overhead . . . 49
3.7 Code Examples . . . 49
3.7.1 Using liboqs (C) . . . 49
3.7.2 Using pqcrypto (Rust) . . . 50
3.7.3 Using liboqs-python . . . 51
3.8 Hybrid Key Exchange . . . 52
3.8.1 X25519-ML-KEM Hybrid . . . 52
3.9 IoT Deployment Considerations . . . 53
3.9.1 Device Class Recommendations . . . 53
3.9.2 Bandwidth Impact . . . 53
3.9.3 Energy Consumption . . . 54
3.10 Chapter Summary . . . 54

4 Digital Signature Schemes **55**
4.1 Introduction to Digital Signatures . . . 55
4.1.1 Signatures in IoT . . . 55
4.1.2 Comparison Overview . . . 56
4.2 ML-DSA (Dilithium) . . . 56
4.2.1 Design Overview . . . 56
4.2.2 Parameters . . . 57
4.2.3 Algorithms . . . 58
4.2.4 Performance . . . 60
4.3 SLH-DSA (SPHINCS+) . . . 61
4.3.1 Design Philosophy . . . 61
4.3.2 Parameters . . . 61
4.3.3 Algorithm Sketch . . . 62
4.3.4 Performance . . . 63
4.3.5 Use Cases . . . 63
4.4 Falcon . . . 63
4.4.1 Design Overview . . . 64
4.4.2 Parameters . . . 64
4.4.3 Algorithm Highlights . . . 64
4.4.4 Performance . . . 65
4.4.5 Falcon vs. ML-DSA . . . 65
4.5 Comparative Analysis . . . 66
4.5.1 Size Comparison . . . 66

4.5.2 Performance Comparison . . . 67
4.5.3 Selection Guidelines . . . 67
4.6 Code Examples . . . 67
4.6.1 ML-DSA with liboqs . . . 67
4.6.2 SPHINCS+ with PQClean . . . 68
4.6.3 Python Example . . . 69
4.7 Implementation Considerations . . . 70
4.7.1 Deterministic vs. Randomized Signing . . . 70
4.7.2 Batch Verification . . . 70
4.7.3 Side-Channel Considerations . . . 71
4.8 Chapter Summary . . . 71

5 Alternative and Emerging Schemes 72
5.1 Code-Based Cryptography . . . 72
5.1.1 Theoretical Foundation . . . 72
5.1.2 HQC (Hamming Quasi-Cyclic) . . . 73
5.1.3 BIKE (Bit Flipping Key Encapsulation) . . . 74
5.1.4 Classic McEliece . . . 75
5.1.5 Code-Based Schemes Summary . . . 76
5.2 Isogeny-Based Cryptography . . . 76
5.2.1 Mathematical Background . . . 76
5.2.2 SIDH/SIKE: A Cautionary Tale . . . 77
5.2.3 Post-SIKE Isogeny Research . . . 77
5.3 Multivariate Cryptography . . . 77
5.3.1 Theoretical Foundation . . . 78
5.3.2 Signature Schemes . . . 78
5.3.3 Multivariate for IoT . . . 79
5.4 Stateful Hash-Based Signatures . . . 79
5.4.1 LMS (Leighton-Micali Signatures) . . . 79
5.4.2 XMSS (eXtended Merkle Signature Scheme) . . . 80
5.4.3 State Management Challenges . . . 80
5.4.4 Stateful vs. Stateless Comparison . . . 81
5.5 Hybrid Schemes . . . 81
5.5.1 Motivation . . . 81
5.5.2 Hybrid KEM Construction . . . 82
5.5.3 Deployed Hybrid Schemes . . . 82
5.5.4 Hybrid Signatures . . . 82
5.6 Comparative Analysis . . . 83
5.6.1 Algorithm Family Summary . . . 83

5.6.2 IoT Suitability Matrix 84
5.7 Chapter Summary 84

6 Implementation for IoT Devices 86
6.1 IoT Hardware Landscape 86
6.1.1 Processor Architectures 86
6.1.2 ARM Cortex-M Series Deep Dive 86
6.1.3 Memory Hierarchy 88
6.2 Memory Optimization 88
6.2.1 Stack Usage Analysis 88
6.2.2 Stack Reduction Techniques 89
6.2.3 Code Size Optimization 91
6.3 Performance Optimization 92
6.3.1 NTT Optimization 92
6.3.2 Hash Function Optimization 94
6.3.3 Parallelism and Pipelining 95
6.4 Side-Channel Countermeasures 96
6.4.1 Timing Attacks 96
6.4.2 Power Analysis 97
6.4.3 Fault Attacks 98
6.5 Hardware Acceleration 99
6.5.1 Cryptographic Accelerators 99
6.5.2 FPGA and Custom Hardware 101
6.6 Random Number Generation 101
6.6.1 Importance in PQC 101
6.6.2 Hardware RNG 102
6.6.3 DRBG Seeding 103
6.7 Testing and Validation 104
6.7.1 Known Answer Tests (KAT) 104
6.7.2 Benchmark Framework 105
6.8 Chapter Summary 107

7 Libraries and Tools 108
7.1 Overview of PQC Libraries 108
7.2 liboqs (Open Quantum Safe) 108
7.2.1 Features 108
7.2.2 Installation 109
7.2.3 API Reference 109
7.2.4 Python Bindings 111
7.3 PQClean 112

7.3.1 Design Philosophy . . . 113
7.3.2 Structure . . . 113
7.3.3 API . . . 114
7.3.4 Integration Example . . . 114
7.4 pqm4: PQC for ARM Cortex-M4 . . . 115
7.4.1 Supported Platforms . . . 116
7.4.2 Building and Running . . . 116
7.4.3 Implementation Variants . . . 117
7.4.4 Benchmark Results . . . 117
7.4.5 Using pqm4 in Your Project . . . 117
7.5 wolfSSL / wolfCrypt . . . 118
7.5.1 Features . . . 118
7.5.2 PQC Algorithms . . . 119
7.5.3 Building with PQC . . . 119
7.5.4 API Usage . . . 119
7.5.5 TLS 1.3 with Hybrid PQC . . . 121
7.6 Rust Libraries . . . 122
7.6.1 pqcrypto Crate . . . 122
7.6.2 RustCrypto PQC . . . 124
7.7 Development Tools . . . 125
7.7.1 Testing Tools . . . 125
7.7.2 Side-Channel Analysis . . . 125
7.7.3 Formal Verification . . . 125
7.8 Library Selection Guide . . . 126
7.8.1 Decision Matrix . . . 126
7.8.2 Recommendations by Platform . . . 126
7.9 Chapter Summary . . . 126

8 Protocol Integration **128**
8.1 TLS 1.3 with Post-Quantum Cryptography . . . 128
8.1.1 TLS 1.3 Handshake Overview . . . 128
8.1.2 Key Exchange Integration . . . 129
8.1.3 Authentication Integration . . . 130
8.1.4 Handshake Size Analysis . . . 131
8.1.5 Performance Impact . . . 131
8.1.6 Implementation with wolfSSL . . . 131
8.2 DTLS 1.3 for Constrained IoT . . . 133
8.2.1 DTLS vs TLS for IoT . . . 134
8.2.2 DTLS Fragmentation with PQC . . . 134

8.2.3 Connection ID for IoT . . . 134
8.3 MQTT with PQC . . . 135
8.3.1 MQTT Architecture . . . 135
8.3.2 MQTT over TLS with PQC . . . 135
8.3.3 MQTT-SN for Constrained Devices . . . 137
8.4 CoAP with PQC . . . 137
8.4.1 CoAP Security Options . . . 138
8.4.2 OSCORE: Object Security for CoAP . . . 138
8.4.3 EDHOC: Lightweight Key Exchange . . . 139
8.5 Matter Protocol . . . 140
8.5.1 Matter Security Architecture . . . 140
8.5.2 Matter Commissioning with PQC . . . 141
8.6 LoRaWAN Security . . . 142
8.6.1 LoRaWAN Constraints . . . 142
8.6.2 PQC Strategies for LoRaWAN . . . 143
8.7 Zigbee and Thread . . . 144
8.7.1 Zigbee Security . . . 144
8.7.2 Thread Security . . . 144
8.8 Protocol Migration Strategy . . . 145
8.8.1 Phased Approach . . . 145
8.8.2 Backward Compatibility . . . 145
8.9 Chapter Summary . . . 146

9 Security Analysis and Best Practices 148
9.1 Threat Modeling for PQC-IoT Systems . . . 148
9.1.1 The STRIDE-PQ Framework . . . 148
9.1.2 Harvest-Now-Decrypt-Later (HNDL) Risk Assessment . . . 149
9.1.3 IoT-Specific Attack Surfaces . . . 149
9.1.4 Threat Actor Profiling . . . 151
9.2 Side-Channel Vulnerability Analysis . . . 151
9.2.1 Algorithm-Specific Vulnerabilities . . . 151
9.2.2 Power Analysis Countermeasures . . . 158
9.2.3 Fault Injection Countermeasures . . . 162
9.3 Security Parameter Selection . . . 165
9.3.1 NIST Security Levels Explained . . . 165
9.3.2 Parameter Selection by Application Domain . . . 165
9.3.3 Conservative vs. Aggressive Parameter Choices . . . 173
9.4 Key Management Best Practices . . . 174
9.4.1 Key Lifecycle Management . . . 174

9.4.2 Key Generation Requirements . . . 174
9.4.3 Secure Key Storage . . . 179
9.4.4 Key Rotation Strategies . . . 182
9.4.5 Key Distribution for IoT Networks . . . 187
9.5 Secure Boot with Post-Quantum Signatures . . . 187
9.5.1 Secure Boot Architecture . . . 189
9.5.2 Algorithm Selection for Secure Boot . . . 189
9.5.3 PQC Secure Boot Implementation . . . 189
9.5.4 Boot Time Analysis . . . 196
9.5.5 Hardware Root of Trust Considerations . . . 197
9.6 Secure Firmware Updates with PQC . . . 200
9.6.1 OTA Update Architecture . . . 200
9.6.2 Update Package Format . . . 200
9.6.3 Bandwidth-Efficient Update Strategies . . . 203
9.6.4 A/B Update Scheme with Rollback . . . 207
9.6.5 Update Verification Pipeline . . . 213
9.7 Certificate Lifecycle Management . . . 216
9.7.1 PQC Certificate Formats . . . 216
9.7.2 Certificate Provisioning Strategies . . . 216
9.7.3 Certificate Renewal and Rotation . . . 221
9.7.4 Certificate Revocation for IoT . . . 226
9.7.5 Certificate Chain Optimization . . . 230
9.8 Compliance Frameworks and Standards . . . 235
9.8.1 CNSA 2.0 (Commercial National Security Algorithm Suite) . . . 236
9.8.2 NIST Post-Quantum Cryptography Standards . . . 240
9.8.3 Industry-Specific Compliance . . . 240
9.8.4 FIPS 140-3 Validation . . . 245
9.8.5 European Regulations (eIDAS 2.0, Cyber Resilience Act) . . . 249
9.8.6 Compliance Implementation Checklist . . . 249
9.9 Chapter Summary . . . 249

10 Migration Roadmaps and Cost Analysis 253
10.1 Migration Timeline Overview . . . 253
10.1.1 Industry Consensus Timeline . . . 253
10.1.2 IoT-Specific Migration Challenges . . . 254
10.2 Migration Strategies . . . 254
10.2.1 Strategy 1: Crypto-Agility First . . . 254
10.2.2 Strategy 2: Phased Hybrid Deployment . . . 259
10.2.3 Strategy 3: Risk-Based Prioritization . . . 259

10.3 Cost Analysis Framework . . . 266
10.3.1 Total Cost of Migration . . . 266
10.3.2 Cost-Benefit Analysis . . . 275
10.4 Implementation Checklist . . . 276
10.5 Lessons Learned and Recommendations . . . 276
10.5.1 Key Success Factors . . . 276
10.5.2 Common Pitfalls to Avoid . . . 277
10.5.3 Final Recommendations . . . 277
10.6 Chapter Summary . . . 278

Conclusion **280**

References **282**

A Algorithm Parameter Reference **285**
A.1 ML-KEM Parameters . . . 285
A.2 ML-DSA Parameters . . . 285
A.3 SLH-DSA Parameters . . . 285

B Benchmark Data **287**
B.1 pqm4 Benchmarks (STM32F4 @ 168 MHz) . . . 287

C Code Examples Index **288**

D Glossary of Terms **289**

List of Figures

3.1 ML-KEM construction from K-PKE via Fujisaki-Okamoto transform . 40

4.1 ML-DSA construction: security from MLWE and MSIS 57

8.1 TLS 1.3 handshake with PQC integration points 128
8.2 MQTT with PQC-secured TLS connections 135

9.1 HNDL Timeline Risk Visualization 149
9.2 PQC Key Lifecycle Stages 174
9.3 Multi-Stage Secure Boot Chain with PQC Signatures 188
9.4 PQC-Secured OTA Update Architecture 201

10.1 Post-Quantum Cryptography Migration Timeline 253

List of Tables

1.1 Complexity comparison: Classical vs. Quantum algorithms 19
1.2 IoT device categories and their characteristics 21
1.3 Comparison of post-quantum cryptography families 23
1.4 NIST PQC Standardization timeline 24

2.1 Complexity comparison of polynomial multiplication methods 29
2.2 LWE parameters in NIST standards 34
2.3 Comparison of lattice reduction algorithms 36
2.4 Estimated security levels for NIST PQC standards 36
2.5 Hash functions used in NIST PQC standards 37
2.6 Keccak/SHA-3 performance on ARM Cortex-M4 (cycles per byte) . . . 37

3.1 ML-KEM parameter sets (FIPS 203) 41
3.2 ML-KEM security estimates (Core-SVP model) 45
3.3 ML-KEM-768 memory requirements 46
3.4 Implementation variants and their tradeoffs 47
3.5 ML-KEM performance on ARM Cortex-M4 (cycles) 48
3.6 ML-KEM stack usage on ARM Cortex-M4 (bytes) 48
3.7 ML-KEM vs. classical algorithms on Cortex-M4 48
3.8 ML-KEM performance on x86-64 with AVX2 (cycles) 49
3.9 Percentage of cycles spent in SHA-3/SHAKE (Cortex-M4) 49
3.10 Hybrid scheme sizes 53
3.11 ML-KEM suitability by IoT device class 53
3.12 Transmission time comparison (LoRaWAN SF7, 125 kHz) 54
3.13 Estimated energy consumption on Cortex-M4 @ 3.3V, 80 MHz 54

4.1 NIST post-quantum signature schemes comparison 56
4.2 ML-DSA parameter sets (FIPS 204) 57
4.3 ML-DSA performance on ARM Cortex-M4 (cycles) 60
4.4 ML-DSA stack usage on ARM Cortex-M4 (bytes) 60
4.5 Selected SLH-DSA parameter sets (FIPS 205) 62
4.6 SLH-DSA performance on ARM Cortex-M4 (cycles, millions) 63

4.7 SLH-DSA vs. ML-DSA: Size and speed tradeoffs (Level 1) 63
4.8 Falcon parameter sets 64
4.9 Falcon performance on ARM Cortex-M4 (cycles) 65
4.10 Falcon RAM requirements 65
4.11 Falcon vs. ML-DSA comparison 65
4.12 Complete size comparison of PQ signature schemes 66
4.13 Signature scheme performance summary (Cortex-M4, Level 1) 67
4.14 Signature scheme selection by use case 67
4.15 Deterministic vs. randomized signing 70
4.16 Side-channel vulnerability by scheme 71

5.1 HQC parameter sets 73
5.2 HQC performance on Intel Core i7-11850H (AVX2) 73
5.3 HQC vs. ML-KEM comparison (Level 1) 74
5.4 BIKE parameter sets 74
5.5 BIKE decoding failure rates 75
5.6 Classic McEliece parameter sets 75
5.7 Code-based schemes comparison (Level 1) 76
5.8 SIKE parameters (before break) 77
5.9 Rainbow parameters (before break) 78
5.10 Multivariate schemes IoT assessment 79
5.11 LMS parameter examples 79
5.12 XMSS vs. LMS comparison 80
5.13 State management strategies for IoT 81
5.14 Stateful (LMS) vs. Stateless (SLH-DSA) signatures 81
5.15 Hybrid schemes in deployment 82
5.16 PQC algorithm families comparison 83
5.17 Scheme suitability by IoT device class 84

6.1 Common IoT processor families 86
6.2 ARM Cortex-M feature comparison for cryptography 87
6.3 Typical memory organization in IoT MCUs 88
6.4 Detailed stack usage on ARM Cortex-M4 (bytes) 89
6.5 Code size (.text section) on ARM Cortex-M4 (bytes) 91
6.6 SHA-3 Keccak-f[1600] performance on various platforms 94
6.7 Power analysis countermeasures 98
6.8 Hardware crypto support in popular IoT MCUs 100
6.9 PQC on FPGA: Performance examples 101

7.1 Major PQC libraries comparison 108

7.2 pqm4 supported development boards . . . 116
7.3 pqm4 implementation variants . . . 117
7.4 Complete pqm4 benchmarks (STM32L4R5ZI @ 80 MHz) . . . 117
7.5 wolfSSL PQC support . . . 119
7.6 PQC testing and analysis tools . . . 125
7.7 Side-channel analysis tools . . . 125
7.8 Formal verification for cryptography . . . 125
7.9 Library selection by use case . . . 126
7.10 Recommended libraries by platform . . . 126

8.1 TLS named groups for PQC (draft registrations) . . . 129
8.2 Certificate chain size comparison . . . 130
8.3 TLS 1.3 handshake size comparison . . . 131
8.4 TLS 1.3 handshake time on ARM Cortex-M4 @ 80 MHz . . . 131
8.5 DTLS vs TLS comparison for IoT . . . 134
8.6 DTLS fragmentation impact . . . 134
8.7 MQTT vs MQTT-SN for PQC . . . 137
8.8 CoAP security modes . . . 138
8.9 EDHOC vs DTLS handshake comparison . . . 139
8.10 Matter security layers . . . 140
8.11 LoRaWAN data rate limitations . . . 142
8.12 Thread security with PQC considerations . . . 144
8.13 Recommended PQC protocol migration phases . . . 145

9.1 STRIDE-PQ: Extended Threat Model for Post-Quantum IoT . . . 148
9.2 Threat Actor Capabilities for PQC-IoT Attacks . . . 151
9.3 ML-KEM Side-Channel Vulnerability Assessment . . . 152
9.4 Falcon Floating-Point Side-Channel Risks . . . 156
9.5 NIST Security Levels and Equivalent Strength . . . 166
9.6 Recommended Security Levels by IoT Application . . . 166
9.7 Parameter Selection Philosophy Comparison . . . 174
9.8 Key Storage Options for IoT Devices . . . 179
9.9 Key Distribution Methods for PQC IoT . . . 188
9.10 PQC Signature Algorithms for Secure Boot . . . 189
9.11 Secure Boot Time Breakdown (Cortex-M4 @ 168 MHz) . . . 197
9.12 Update Overhead Comparison . . . 204
9.13 Update Security Checklist . . . 216
9.14 X.509 Certificate Size Comparison . . . 217
9.15 Revocation Methods for IoT Devices . . . 227
9.16 Certificate Lifecycle Best Practices Summary . . . 235

9.17 CNSA 2.0 Algorithm Requirements 236
9.18 NIST PQC Standards (August 2024) 241
9.19 PQC Considerations for Healthcare IoT 241
9.20 Automotive Cybersecurity PQC Requirements 245
9.21 FIPS 140-3 PQC Validation Status (December 2024) 245
9.22 European PQC Regulatory Landscape 249
9.23 PQC Compliance Implementation Checklist 250

10.1 Migration Phase Definitions 254
10.2 IoT Migration Challenges vs. Enterprise IT 254
10.3 Phased Hybrid Deployment Plan 259
10.4 PQC Migration Cost Categories 266
10.5 PQC Migration Cost vs. Risk of Inaction 275
10.6 PQC Migration Implementation Checklist 276
10.7 Common PQC Migration Pitfalls 277

A.1 Complete ML-KEM Parameter Sets (FIPS 203) 285
A.2 Complete ML-DSA Parameter Sets (FIPS 204) 286
A.3 SLH-DSA Parameter Sets (FIPS 205) — SHA-2 Variants 286

B.1 Complete pqm4 Benchmark Results (CPU Cycles) 287

C.1 Code Examples by Chapter 288

List of Algorithms

1 Cooley-Tukey NTT (In-place, Iterative) 30
2 ML-KEM.KeyGen() 41
3 ML-KEM.Encaps(pk) 42
4 ML-KEM.Decaps(sk, ct) 43
5 CBD_η(bytes) 44
6 Hybrid X25519-ML-KEM Key Exchange 53
7 ML-DSA.KeyGen() 58
8 ML-DSA.Sign(sk, M) 59
9 ML-DSA.Verify(pk, M, σ) 60
10 Generic Hybrid KEM 82

Chapter 1

Introduction

1.1 The Quantum Threat to Modern Cryptography

Modern digital security rests upon the presumed computational difficulty of certain mathematical problems. The Rivest-Shamir-Adleman (RSA) cryptosystem, introduced in 1977, derives its security from the hardness of factoring large integers. Elliptic Curve Cryptography (ECC), which has largely supplanted RSA in many applications due to its efficiency, relies on the discrete logarithm problem over elliptic curves. For decades, these mathematical foundations have provided robust security guarantees, with the best-known classical algorithms requiring exponential time to break properly-sized keys.

Definition 1.1 (Integer Factorization Problem)**.** Given a composite integer $N = p \cdot q$ where p and q are large primes, find the factors p and q.

Definition 1.2 (Elliptic Curve Discrete Logarithm Problem)**.** Given points P and $Q = kP$ on an elliptic curve E over a finite field $\mathbb{F}_q$, find the scalar k.

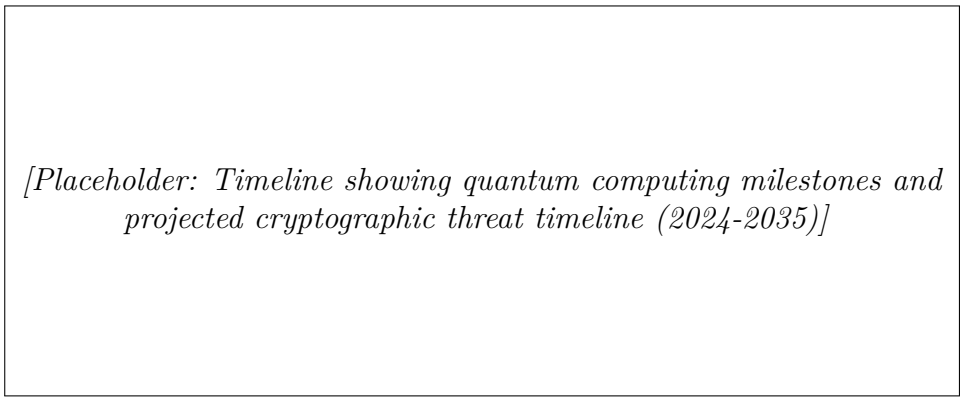
The security landscape changed fundamentally in 1994 when Peter Shor demonstrated that a sufficiently powerful quantum computer could solve both integer factorization and discrete logarithm problems in polynomial time [**?**]. Shor's algorithm achieves exponential speedup over the best-known classical algorithms:

Table 1.1: Complexity comparison: Classical vs. Quantum algorithms

Problem	**Classical**	**Quantum (Shor)**	**Speedup**
Integer Factorization	$O\left(e^{1.9 \cdot n^{1/3} \cdot (\ln n)^{2/3}}\right)$	$O(n^3)$	Exponential
Discrete Logarithm	$O(\sqrt{q})$	$O((\log q)^3)$	Exponential
ECDLP (ECC)	$O(\sqrt{q})$	$O((\log q)^3)$	Exponential

1.1.1 Current State of Quantum Computing

As of December 2024, quantum computers have made remarkable progress but have not yet achieved the scale necessary to threaten current cryptographic systems. Breaking RSA-2048 would require approximately 4,000 logical qubits with full error correction, translating to millions of physical qubits with current error rates. The largest quantum computers today operate with hundreds to low thousands of physical qubits, with limited coherence times and high error rates.



However, the cryptographic community operates under the assumption that cryptographically-relevant quantum computers (CRQCs) will eventually be built. This assumption is motivated by several factors:

1. **Continuous Progress**: Qubit counts and error correction capabilities improve yearly
2. **Major Investment**: Governments and corporations invest billions in quantum computing research
3. **Harvest Now, Decrypt Later**: Adversaries can store encrypted data today to decrypt when quantum computers become available
4. **Migration Time**: Transitioning cryptographic infrastructure takes 10–20 years

Warning: The "harvest now, decrypt later" threat means that data encrypted today with classical cryptography may be compromised in the future. This is particularly concerning for data with long-term confidentiality requirements.

1.1.2 Impact on IoT Security

The Internet of Things encompasses billions of connected devices, from industrial sensors and smart meters to medical implants and autonomous vehicles. These devices face unique challenges in the post-quantum transition:

- **Resource Constraints**: Many IoT devices have limited computational power (MHz-class processors), memory (kilobytes of RAM), and energy budgets
- **Long Deployment Lifetimes**: Industrial IoT devices may remain operational for 15–20 years, spanning the pre- and post-quantum eras
- **Difficult Updates**: Many deployed devices lack secure update mechanisms or may be physically inaccessible
- **Real-Time Requirements**: Safety-critical applications demand predictable latency
- **Bandwidth Limitations**: Low-power wide-area networks (LPWAN) have severe bandwidth constraints

Table 1.2: IoT device categories and their characteristics

Category	Processor	RAM	Flash	Example
Class 0 (Constrained)	8-bit MCU	<10 KB	<100 KB	Sensor nodes
Class 1 (Limited)	16-bit MCU	∼10 KB	∼100 KB	Smart meters
Class 2 (Capable)	32-bit ARM	∼50 KB	∼250 KB	Wearables
Class 3 (Advanced)	ARM Cortex-M4/M7	>256 KB	>1 MB	Gateways

1.2 Post-Quantum Cryptography: An Overview

Post-Quantum Cryptography (PQC) refers to cryptographic algorithms designed to resist attacks by both classical and quantum computers. Unlike quantum cryptography (which uses quantum mechanical phenomena for key distribution), PQC algorithms run on conventional hardware using classical computation.

1.2.1 Mathematical Foundations

Post-quantum cryptographic schemes are built upon mathematical problems believed to be resistant to quantum attacks. The main families include:

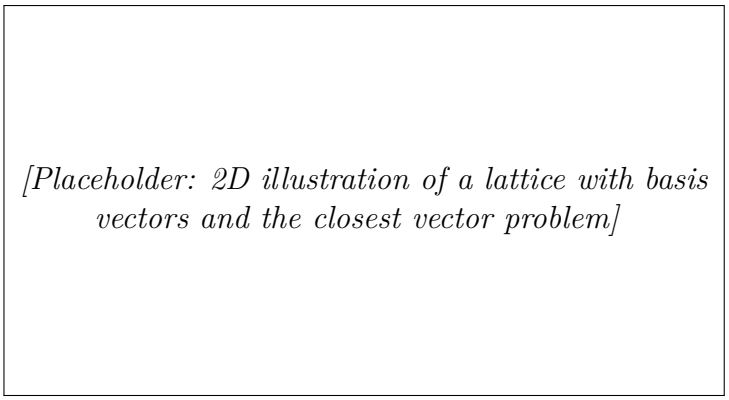
Lattice-Based Cryptography

Lattice-based schemes derive security from the hardness of problems involving high-dimensional geometric structures called lattices.

Definition 1.3 (Lattice)**.** A *lattice* Λ in $\mathbb{R}^n$ is the set of all integer linear combinations of a set of linearly independent vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{R}^n$:

$$\Lambda = \mathcal{L}(\mathbf{B}) = \left\{ \sum_{i=1}^{n} x_i \mathbf{b}_i : x_i \in \mathbb{Z} \right\}$$

where $\mathbf{B} = [\mathbf{b}_1 | \cdots | \mathbf{b}_n]$ is the *basis* of the lattice.



The security of lattice-based cryptography relies on two fundamental problems:

Definition 1.4 (Learning With Errors (LWE))**.** Given a matrix $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$, a secret vector $\mathbf{s} \in \mathbb{Z}_q^n$, and a noise vector $\mathbf{e}$ sampled from an error distribution χ, the LWE problem asks to find $\mathbf{s}$ given $(\mathbf{A}, \mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e} \mod q)$.

Definition 1.5 (Short Integer Solution (SIS))**.** Given a matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ and a bound β, find a non-zero vector $\mathbf{z} \in \mathbb{Z}^m$ such that $\mathbf{A}\mathbf{z} = \mathbf{0} \mod q$ and $\|\mathbf{z}\| \leq \beta$.

Modern schemes use *structured* variants of these problems (Ring-LWE, Module-LWE) that operate over polynomial rings, enabling more efficient implementations:

Definition 1.6 (Module-LWE)**.** The Module-LWE problem is defined over the ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ where operations involve matrices and vectors of polynomials. For rank k, given $\mathbf{A} \in R_q^{k \times k}$ and $\mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e}$, find $\mathbf{s}$.

Hash-Based Signatures

Hash-based signature schemes derive security solely from the properties of cryptographic hash functions, making them the most conservative choice from a security perspective.

Definition 1.7 (Hash Function Security Properties)**.** A cryptographic hash function $H : \{0,1\}^* \to \{0,1\}^n$ should satisfy:

- **Preimage Resistance**: Given h, hard to find m such that $H(m) = h$

- **Second Preimage Resistance**: Given m_1, hard to find $m_2 \neq m_1$ with $H(m_1) = H(m_2)$
- **Collision Resistance**: Hard to find any $m_1 \neq m_2$ with $H(m_1) = H(m_2)$

Hash-based signatures use Merkle tree constructions to authenticate many one-time signature keys under a single public key.

Code-Based Cryptography

Code-based schemes rely on the difficulty of decoding random linear codes.

Definition 1.8 (Syndrome Decoding Problem)**.** Given a parity-check matrix $\mathbf{H} \in \mathbb{F}_2^{(n-k)\times n}$, a syndrome $\mathbf{s} \in \mathbb{F}_2^{n-k}$, and a weight bound t, find an error vector $\mathbf{e} \in \mathbb{F}_2^n$ with Hamming weight at most t such that $\mathbf{He}^T = \mathbf{s}$.

Multivariate Cryptography

Multivariate schemes are based on the difficulty of solving systems of multivariate polynomial equations over finite fields.

Definition 1.9 (MQ Problem)**.** Given a system of m quadratic polynomials $p_1, \ldots, p_m$ in n variables over a finite field $\mathbb{F}_q$, find a solution $(x_1, \ldots, x_n) \in \mathbb{F}_q^n$ such that $p_i(x_1, \ldots, x_n) = 0$ for all i.

Isogeny-Based Cryptography

Isogeny-based schemes use the mathematical structure of elliptic curve isogenies.

Definition 1.10 (Isogeny)**.** An isogeny between elliptic curves E_1 and E_2 is a non-constant rational map $\phi : E_1 \to E_2$ that is also a group homomorphism.

Note: The SIDH/SIKE scheme was broken in 2022 by Castryck and Decru, demonstrating the importance of cryptanalytic scrutiny before standardization.

1.2.2 Comparison of PQC Families

Table 1.3: Comparison of post-quantum cryptography families

Family	Key Size	Sig/CT Size	Speed	Maturity	IoT Suitability
Lattice-based	Medium	Medium	Fast	High	Good
Hash-based	Small/Large	Large	Medium	Very High	Limited
Code-based	Large	Small	Fast	High	Limited
Multivariate	Large	Small	Variable	Medium	Poor
Isogeny-based	Small	Small	Slow	Low*	Poor

*Isogeny-based schemes have suffered significant cryptanalytic setbacks.

1.3 NIST Post-Quantum Standardization

The National Institute of Standards and Technology initiated its Post-Quantum Cryptography Standardization project in 2016, recognizing the urgent need to prepare for the quantum computing era. The process followed NIST's established approach for cryptographic standards, emphasizing open competition, public analysis, and iterative refinement.

1.3.1 Timeline and Process

Table 1.4: NIST PQC Standardization timeline

Date	Milestone
December 2016	Call for proposals issued
November 2017	69 submissions received (Round 1)
January 2019	26 candidates advance to Round 2
July 2020	7 finalists + 8 alternates (Round 3)
July 2022	Selection of first algorithms for standardization
August 2024	**FIPS 203, 204, 205 published (ML-KEM, ML-DSA, SLH-DSA)**
2024–2025	Round 4 (additional signatures) and HQC standardization

1.3.2 Standardized Algorithms (August 2024)

On August 13, 2024, NIST published three Federal Information Processing Standards:

1. **FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM)**
    - Based on the CRYSTALS-Kyber submission
    - Security from Module-LWE problem
    - Three parameter sets: ML-KEM-512, ML-KEM-768, ML-KEM-1024
2. **FIPS 204: Module-Lattice-Based Digital Signature Algorithm (ML-DSA)**
    - Based on the CRYSTALS-Dilithium submission
    - Security from Module-LWE and Module-SIS problems
    - Three parameter sets: ML-DSA-44, ML-DSA-65, ML-DSA-87
3. **FIPS 205: Stateless Hash-Based Digital Signature Algorithm (SLH-DSA)**
    - Based on the SPHINCS+ submission

- Security from hash function properties only
- Multiple parameter sets with size/speed tradeoffs

1.3.3 Algorithms Under Standardization

Additional algorithms are progressing through standardization:

- **Falcon**: NTRU-lattice-based signature scheme, selected for standardization but requiring additional specification work due to implementation complexity
- **HQC**: Code-based KEM selected for Round 4 standardization as a backup to ML-KEM
- **BIKE**: Code-based KEM under continued evaluation

1.4 Scope and Organization of This Document

This reference document provides a comprehensive survey of post-quantum cryptography for IoT applications, organized as follows:

Chapter 2: Mathematical Foundations Detailed treatment of lattice theory, the LWE/SIS problems, and the mathematical structures underlying PQC schemes.

Chapter 3: Key Encapsulation Mechanisms In-depth analysis of ML-KEM (Kyber), including algorithms, parameters, security analysis, and implementation considerations.

Chapter 4: Digital Signature Schemes Comprehensive coverage of ML-DSA (Dilithium), SLH-DSA (SPHINCS+), and Falcon, with comparative analysis.

Chapter 5: Alternative and Emerging Schemes Survey of code-based (HQC, BIKE, Classic McEliece), multivariate, and other PQC approaches.

Chapter 6: Implementation for IoT Practical guidance on implementing PQC in resource-constrained environments, with benchmark data and optimization strategies.

Chapter 7: Libraries and Tools Catalog of available software libraries, with feature comparisons and usage examples.

Chapter 8: Protocol Integration Integration of PQC into IoT protocols including TLS, DTLS, MQTT, and CoAP.

Chapter 9: Migration Strategies Roadmaps and best practices for transitioning from classical to post-quantum cryptography.

Chapter 10: Future Directions Emerging research areas, open problems, and the evolving PQC landscape.

1.5 How to Use This Reference

This document is designed to serve multiple audiences:

- **Graduate Students**: Start with Chapter 1–2 for foundational understanding, then proceed to specific algorithm chapters based on research interests.
- **Researchers**: Use the benchmark tables and comparative analyses to inform experimental design and publication references.
- **Practitioners/Engineers**: Focus on Chapters 6–9 for implementation guidance, library selection, and migration planning.
- **Security Architects**: Chapters 8–9 provide protocol integration and migration strategies for system design.

Note: Throughout this document, placeholder images indicate where readers may insert relevant diagrams, charts, or photographs. Image specifications are provided in the placeholder descriptions.

1.6 Chapter Summary

This introductory chapter established the context and motivation for post-quantum cryptography in IoT environments:

- Shor's algorithm renders current public-key cryptography vulnerable to quantum attacks
- The "harvest now, decrypt later" threat creates urgency for immediate action
- IoT devices face unique challenges due to resource constraints and long deployment lifetimes
- NIST published the first PQC standards (FIPS 203, 204, 205) in August 2024
- Lattice-based cryptography offers the best balance of security and efficiency for IoT
- This document provides comprehensive coverage for researchers and practitioners

Chapter 2

Mathematical Foundations

This chapter provides the mathematical background necessary to understand post-quantum cryptographic schemes. We focus on lattice theory and the computational problems that underpin the security of modern PQC algorithms, presented at a level accessible to graduate students with basic knowledge of linear algebra and abstract algebra.

2.1 Algebraic Preliminaries

2.1.1 Groups, Rings, and Fields

Definition 2.1 (Group)**.** A *group* $(G, \cdot)$ is a set G equipped with a binary operation $\cdot$ satisfying:

1. **Closure**: For all $a, b \in G$, we have $a \cdot b \in G$
2. **Associativity**: For all $a, b, c \in G$, we have $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
3. **Identity**: There exists $e \in G$ such that $e \cdot a = a \cdot e = a$ for all $a \in G$
4. **Inverse**: For each $a \in G$, there exists $a^{-1} \in G$ such that $a \cdot a^{-1} = a^{-1} \cdot a = e$

If additionally $a \cdot b = b \cdot a$ for all $a, b \in G$, the group is called *abelian*.

Definition 2.2 (Ring)**.** A *ring* $(R, +, \cdot)$ is a set R equipped with two binary operations such that:

1. $(R, +)$ is an abelian group
2. $\cdot$ is associative
3. Distributive laws hold: $a \cdot (b + c) = a \cdot b + a \cdot c$ and $(a + b) \cdot c = a \cdot c + b \cdot c$

If $\cdot$ is commutative, R is a *commutative ring*. If R has a multiplicative identity 1, it is a *ring with unity*.

Definition 2.3 (Field)**.** A *field* $(F, +, \cdot)$ is a commutative ring with unity where every non-zero element has a multiplicative inverse.

Example 2.1 (Finite Fields)**.** For a prime p, the integers modulo p, denoted $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z} = \{0, 1, \ldots, p-1\}$, form a field with addition and multiplication performed modulo p. In PQC, we typically use:

- $\mathbb{F}_q$ where $q = 3329$ (Kyber/ML-KEM)
- $\mathbb{F}_q$ where $q = 8380417$ (Dilithium/ML-DSA)

2.1.2 Polynomial Rings

Lattice-based cryptography extensively uses polynomial rings for efficiency.

Definition 2.4 (Polynomial Ring)**.** Let R be a ring. The *polynomial ring* $R[x]$ consists of all polynomials in the variable x with coefficients in R:

$$R[x] = \left\{ \sum_{i=0}^{n} a_i x^i : a_i \in R, n \in \mathbb{N} \right\}$$

Definition 2.5 (Quotient Ring)**.** For a polynomial $f(x) \in R[x]$, the *quotient ring* $R[x]/(f(x))$ consists of equivalence classes of polynomials modulo $f(x)$. Elements can be represented as polynomials of degree less than $\deg(f)$.

Definition 2.6 (Cyclotomic Ring)**.** The ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$, where n is a power of 2 and q is prime with $q \equiv 1 \pmod{2n}$, is called a *cyclotomic ring*. This is the algebraic structure underlying Kyber, Dilithium, and many other lattice-based schemes.

Example 2.2. In ML-KEM (Kyber), we use $R_q = \mathbb{Z}_{3329}[x]/(x^{256} + 1)$. An element $a \in R_q$ is a polynomial:

$$a(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{255} x^{255}$$

where each coefficient $a_i \in \{0, 1, \ldots, 3328\}$. Multiplication is performed modulo both $q = 3329$ (for coefficients) and $x^{256} + 1$ (for the polynomial degree).

> **Note:** The choice of $x^n + 1$ as the modulus polynomial is not arbitrary. For n a power of 2, this polynomial is irreducible over $\mathbb{Q}$ and allows efficient multiplication via the Number Theoretic Transform.

2.1.3 The Number Theoretic Transform (NTT)

The Number Theoretic Transform is a crucial algorithmic tool that enables efficient polynomial multiplication in R_q.

Definition 2.7 (Primitive Root of Unity)**.** An element $\omega \in \mathbb{Z}_q$ is a *primitive n-th root of unity* if $\omega^n = 1$ and $\omega^k \neq 1$ for all $0 < k < n$.

Definition 2.8 (Number Theoretic Transform)**.** For a polynomial $a(x) = \sum_{i=0}^{n-1} a_i x^i \in R_q$ and a primitive n-th root of unity ω, the NTT is defined as:

$$\hat{a}_j = \mathrm{NTT}(a)_j = \sum_{i=0}^{n-1} a_i \omega^{ij} \mod q, \quad j = 0, 1, \ldots, n-1$$

The inverse NTT is:

$$a_i = \mathrm{NTT}^{-1}(\hat{a})_i = n^{-1} \sum_{j=0}^{n-1} \hat{a}_j \omega^{-ij} \mod q$$

The key property is that polynomial multiplication becomes pointwise multiplication in the NTT domain:

$$\mathrm{NTT}(a \cdot b) = \mathrm{NTT}(a) \circ \mathrm{NTT}(b)$$

where $\circ$ denotes componentwise multiplication.

Table 2.1: Complexity comparison of polynomial multiplication methods

Method	**Multiplications**	**Additions**
Schoolbook	$O(n^2)$	$O(n^2)$
Karatsuba	$O(n^{1.585})$	$O(n^{1.585})$
NTT-based	$O(n \log n)$	$O(n \log n)$

Algorithm 1 Cooley-Tukey NTT (In-place, Iterative)

Require: Polynomial coefficients $a = (a_0, \ldots, a_{n-1})$, primitive root ω, modulus q
Ensure: NTT coefficients $\hat{a} = (\hat{a}_0, \ldots, \hat{a}_{n-1})$

```
Bit-reverse the array a
for s = 1 to log_2 n do
    m ← 2^s
    ω_m ← ω^{n/m} mod q
    for k = 0 to n − 1 by m do
        w ← 1
        for j = 0 to m/2 − 1 do
            t ← w · a[k + j + m/2] mod q
            u ← a[k + j]
            a[k + j] ← (u + t) mod q
            a[k + j + m/2] ← (u − t) mod q
            w ← w · ω_m mod q
        end for
    end for
end for
return a
```

2.2 Lattice Theory

2.2.1 Basic Definitions

Definition 2.9 (Lattice)**.** A *lattice* $\mathcal{L}$ is a discrete additive subgroup of $\mathbb{R}^n$. Equivalently, given linearly independent vectors $\mathbf{b}_1, \ldots, \mathbf{b}_k \in \mathbb{R}^n$ (where $k \leq n$), the lattice generated by these vectors is:

$$\mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_k) = \left\{ \sum_{i=1}^{k} z_i \mathbf{b}_i : z_i \in \mathbb{Z} \right\}$$

The vectors $\mathbf{b}_1, \ldots, \mathbf{b}_k$ form a *basis* of the lattice. If $k = n$, the lattice is *full-rank*.

[Placeholder: 3D visualization of a lattice showing basis vectors, lattice points, and the fundamental domain]

Definition 2.10 (Fundamental Domain)**.** The *fundamental domain* (or fundamental

parallelepiped) of a lattice $\mathcal{L}$ with basis $\mathbf{B} = [\mathbf{b}_1 | \cdots | \mathbf{b}_n]$ is:

$$\mathcal{P}(\mathbf{B}) = \left\{ \sum_{i=1}^{n} x_i \mathbf{b}_i : x_i \in [0, 1) \right\}$$

Definition 2.11 (Determinant of a Lattice)**.** The *determinant* (or volume) of an n-dimensional lattice $\mathcal{L}$ with basis matrix $\mathbf{B}$ is:

$$\det(\mathcal{L}) = |\det(\mathbf{B})| = \mathrm{vol}(\mathcal{P}(\mathbf{B}))$$

This value is independent of the choice of basis.

Definition 2.12 (Dual Lattice)**.** The *dual lattice* of $\mathcal{L}$ is:

$$\mathcal{L}^* = \{\mathbf{y} \in \mathbb{R}^n : \langle \mathbf{y}, \mathbf{x} \rangle \in \mathbb{Z} \text{ for all } \mathbf{x} \in \mathcal{L}\}$$

If $\mathbf{B}$ is a basis for $\mathcal{L}$, then $(\mathbf{B}^{-1})^T$ is a basis for $\mathcal{L}^*$.

2.2.2 Successive Minima and the Gaussian Heuristic

Definition 2.13 (Successive Minima)**.** For a lattice $\mathcal{L} \subset \mathbb{R}^n$, the i-th successive minimum $\lambda_i(\mathcal{L})$ is the smallest radius r such that $\mathcal{L}$ contains i linearly independent vectors of length at most r:

$$\lambda_i(\mathcal{L}) = \min\{r : \dim(\mathrm{span}(\mathcal{L} \cap \bar{B}(0, r))) \geq i\}$$

where $\bar{B}(0, r)$ is the closed ball of radius r centered at the origin.

Theorem 2.1 (Minkowski's First Theorem)**.** *For any n-dimensional lattice $\mathcal{L}$:*

$$\lambda_1(\mathcal{L}) \leq \sqrt{n} \cdot \det(\mathcal{L})^{1/n}$$

Definition 2.14 (Gaussian Heuristic)**.** For a "random" n-dimensional lattice $\mathcal{L}$, the Gaussian heuristic predicts:

$$\lambda_1(\mathcal{L}) \approx \sqrt{\frac{n}{2\pi e}} \cdot \det(\mathcal{L})^{1/n}$$

This heuristic is empirically accurate for most lattices encountered in cryptography.

2.2.3 Important Lattice Constructions

q-ary Lattices

Cryptographic lattice problems are typically defined using q-ary lattices, which contain $q\mathbb{Z}^n$ as a sublattice.

Definition 2.15 (q-ary Lattices)**.** Given a matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ and modulus q, define:

$$\Lambda_q(\mathbf{A}) = \{\mathbf{y} \in \mathbb{Z}^m : \mathbf{y} = \mathbf{A}^T\mathbf{s} \mod q \text{ for some } \mathbf{s} \in \mathbb{Z}_q^n\} \tag{2.1}$$

$$\Lambda_q^\perp(\mathbf{A}) = \{\mathbf{y} \in \mathbb{Z}^m : \mathbf{A}\mathbf{y} = \mathbf{0} \mod q\} \tag{2.2}$$

Ideal Lattices

Definition 2.16 (Ideal Lattice)**.** Let $R = \mathbb{Z}[x]/(f(x))$ be a polynomial ring. An *ideal lattice* is a lattice corresponding to an ideal $I \subseteq R$ under the coefficient embedding:

$$\sigma : R \to \mathbb{Z}^n, \quad a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} \mapsto (a_0, a_1, \ldots, a_{n-1})$$

Module Lattices

Definition 2.17 (Module Lattice)**.** A *module lattice* is a lattice corresponding to a module $M \subseteq R^k$ where $R = \mathbb{Z}[x]/(f(x))$. Module lattices generalize both unstructured lattices (k large, $n = 1$) and ideal lattices ($k = 1$).

Note: The NIST standards use module lattices with small rank $k \in \{2, 3, 4\}$ over rings R with $n = 256$. This provides a balance between the efficiency of ideal lattices and the security confidence of less-structured lattices.

2.3 Computational Lattice Problems

2.3.1 The Shortest Vector Problem (SVP)

Definition 2.18 (Shortest Vector Problem (SVP))**.** Given a basis $\mathbf{B}$ of a lattice $\mathcal{L}$, find a non-zero vector $\mathbf{v} \in \mathcal{L}$ such that $\|\mathbf{v}\| = \lambda_1(\mathcal{L})$.

Definition 2.19 (Approximate SVP (γ-SVP))**.** Given a basis $\mathbf{B}$ of a lattice $\mathcal{L}$ and an approximation factor $\gamma \geq 1$, find a non-zero vector $\mathbf{v} \in \mathcal{L}$ such that $\|\mathbf{v}\| \leq \gamma \cdot \lambda_1(\mathcal{L})$.

The exact SVP is known to be NP-hard under randomized reductions. For $\gamma = \text{poly}(n)$, no polynomial-time quantum algorithms are known.

2.3.2 The Closest Vector Problem (CVP)

Definition 2.20 (Closest Vector Problem (CVP))**.** Given a basis $\mathbf{B}$ of a lattice $\mathcal{L}$ and a target vector $\mathbf{t} \in \mathbb{R}^n$, find a lattice vector $\mathbf{v} \in \mathcal{L}$ that minimizes $\|\mathbf{t} - \mathbf{v}\|$.

Definition 2.21 (Bounded Distance Decoding (BDD))**.** Given a basis $\mathbf{B}$ of a lattice $\mathcal{L}$, a target $\mathbf{t}$, and a bound $d < \lambda_1(\mathcal{L})/2$, find $\mathbf{v} \in \mathcal{L}$ with $\|\mathbf{t} - \mathbf{v}\| \leq d$.

CVP is at least as hard as SVP, and the BDD variant is the decryption problem in many lattice-based encryption schemes.

2.3.3 Learning With Errors (LWE)

The LWE problem, introduced by Regev in 2005, is the foundation of modern lattice-based cryptography.

Definition 2.22 (LWE Distribution)**.** For security parameter n, modulus q, and error distribution χ over $\mathbb{Z}$, the LWE distribution $A_{s,\chi}$ for secret $\mathbf{s} \in \mathbb{Z}_q^n$ produces samples $(\mathbf{a}, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ where:

- $\mathbf{a} \xleftarrow{\$} \mathbb{Z}_q^n$ is uniformly random
- $e \leftarrow \chi$
- $b = \langle \mathbf{a}, \mathbf{s} \rangle + e \mod q$

Definition 2.23 (Search-LWE)**.** Given m independent samples from $A_{s,\chi}$, find $\mathbf{s}$.

Definition 2.24 (Decision-LWE)**.** Distinguish between m samples from $A_{s,\chi}$ and m samples from the uniform distribution over $\mathbb{Z}_q^n \times \mathbb{Z}_q$.

Theorem 2.2 (Regev 2005)**.** *For appropriate parameter choices, solving (Decision-)LWE is at least as hard as solving γ-SVP on arbitrary n-dimensional lattices in the worst case, for $\gamma = \tilde{O}(n \cdot q/\alpha)$ where α parameterizes the error distribution.*

Error Distributions

Definition 2.25 (Discrete Gaussian Distribution)**.** The discrete Gaussian distribution $D_{\mathbb{Z},\sigma}$ over the integers with parameter $\sigma > 0$ assigns probability proportional to $\exp(-x^2/(2\sigma^2))$ to each integer x.

Definition 2.26 (Centered Binomial Distribution)**.** The centered binomial distribution β_η samples $a_1, \ldots, a_\eta, b_1, \ldots, b_\eta \xleftarrow{\$} \{0,1\}$ and outputs $\sum_{i=1}^{\eta}(a_i - b_i)$. The result is in $\{-\eta, \ldots, \eta\}$ with variance $\eta/2$.

Note: Kyber/ML-KEM uses the centered binomial distribution β_η with $\eta \in \{2, 3\}$ because it is easy to sample (just XOR random bits) and has good security properties, unlike Gaussian sampling which is complex to implement securely.

Table 2.2: LWE parameters in NIST standards

Scheme	n	k	q	**Error Dist.**	η	**Security**
ML-KEM-512	256	2	3329	β_3	3	128-bit
ML-KEM-768	256	3	3329	β_2	2	192-bit
ML-KEM-1024	256	4	3329	β_2	2	256-bit
ML-DSA-44	256	4×4	8380417	Uniform	—	128-bit
ML-DSA-65	256	6×5	8380417	Uniform	—	192-bit
ML-DSA-87	256	8×7	8380417	Uniform	—	256-bit

2.3.4 Ring-LWE and Module-LWE

Definition 2.27 (Ring-LWE)**.** Let $R = \mathbb{Z}[x]/(x^n + 1)$ and $R_q = R/qR$. The Ring-LWE problem is the LWE problem where $\mathbf{a}, \mathbf{s}, e$ are elements of R_q (polynomials) rather than vectors.

Definition 2.28 (Module-LWE)**.** Let $R_q = \mathbb{Z}_q[x]/(x^n + 1)$. The Module-LWE problem uses:

- Public matrix $\mathbf{A} \in R_q^{k \times k}$ (matrix of polynomials)
- Secret vector $\mathbf{s} \in R_q^k$
- Error vector $\mathbf{e} \in R_q^k$
- Public value $\mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e} \in R_q^k$

The module structure provides a security/efficiency tradeoff:

- **LWE** (k large, $n = 1$): Highest confidence in hardness, least efficient
- **Ring-LWE** ($k = 1$, n large): Most efficient, concerns about ring structure
- **Module-LWE** (k and n both moderate): Balance of efficiency and security confidence

2.3.5 Short Integer Solution (SIS)

Definition 2.29 (SIS Problem)**.** Given a uniformly random matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ and a bound β, find a non-zero vector $\mathbf{z} \in \mathbb{Z}^m$ such that:

$$\mathbf{A}\mathbf{z} = \mathbf{0} \mod q \quad \text{and} \quad \|\mathbf{z}\| \leq \beta$$

Definition 2.30 (Inhomogeneous SIS (ISIS))**.** Given $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$, a target $\mathbf{u} \in \mathbb{Z}_q^n$, and bound β, find $\mathbf{z} \in \mathbb{Z}^m$ with $\mathbf{A}\mathbf{z} = \mathbf{u} \mod q$ and $\|\mathbf{z}\| \leq \beta$.

SIS is dual to LWE and forms the security basis for lattice-based signatures like Dilithium/ML-DSA.

Theorem 2.3 (Ajtai 1996)**.** *For appropriate parameters, solving SIS is at least as hard as solving γ-SVP on worst-case lattices.*

2.4 Algorithms for Lattice Problems

2.4.1 Lattice Reduction

Lattice reduction algorithms find short vectors by transforming a basis into one with shorter, more orthogonal vectors.

Definition 2.31 (LLL Reduction)**.** A basis $\mathbf{B} = [\mathbf{b}_1 | \cdots | \mathbf{b}_n]$ is *LLL-reduced* with parameter $\delta \in (1/4, 1)$ if:

1. Size-reduced: $|\mu_{i,j}| \leq 1/2$ for all $i > j$, where $\mu_{i,j} = \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\|\mathbf{b}_j^*\|^2}$
2. Lovász condition: $\delta \|\mathbf{b}_i^*\|^2 \leq \|\mathbf{b}_{i+1}^*\|^2 + \mu_{i+1,i}^2 \|\mathbf{b}_i^*\|^2$ for all i

where $\mathbf{b}_i^*$ are the Gram-Schmidt orthogonalized vectors.

Theorem 2.4 (LLL Guarantee)**.** *For an LLL-reduced basis with $\delta = 3/4$, the shortest basis vector satisfies:*

$$\|\mathbf{b}_1\| \leq 2^{(n-1)/2} \cdot \lambda_1(\mathcal{L})$$

The LLL algorithm runs in polynomial time $O(n^5 d \log^3 B)$ where B bounds the entries of the input basis.

Definition 2.32 (BKZ Reduction)**.** Block Korkine-Zolotarev (BKZ) with block size β generalizes LLL by solving SVP in blocks of dimension β. Larger block sizes yield shorter vectors but increase runtime exponentially in β.

Table 2.3: Comparison of lattice reduction algorithms

Algorithm	Approximation Factor	Time Complexity	Space
LLL	$2^{O(n)}$	$\text{poly}(n)$	$\text{poly}(n)$
BKZ-β	$2^{O(n/\beta)}$	$2^{O(\beta)} \cdot \text{poly}(n)$	$2^{O(\beta)}$
BKZ 2.0	$2^{O(n/\beta)}$	$2^{O(\beta)} \cdot \text{poly}(n)$	$2^{O(\beta)}$

2.4.2 Sieving Algorithms

Definition 2.33 (Lattice Sieving)**.** Sieving algorithms maintain a list of lattice vectors and repeatedly combine pairs to create shorter vectors. The asymptotically fastest sieving algorithms achieve time $2^{0.292n+o(n)}$ for solving SVP.

2.4.3 Security Estimation

The security of lattice-based schemes is estimated using the *Core-SVP hardness model*:

Definition 2.34 (Core-SVP Hardness)**.** For parameters (n, q, χ), the Core-SVP hardness is the cost of running BKZ with sufficient block size β to solve the underlying lattice problem. The required β is estimated using the Gaussian heuristic and the success condition of BKZ.

Table 2.4: Estimated security levels for NIST PQC standards

Scheme	Classical (bits)	Quantum (bits)	NIST Level
ML-KEM-512	118	107	1 (AES-128)
ML-KEM-768	183	166	3 (AES-192)
ML-KEM-1024	256	232	5 (AES-256)
ML-DSA-44	128	115	2
ML-DSA-65	192	175	3
ML-DSA-87	256	234	5

2.5 Hash Functions in PQC

2.5.1 SHA-3 and SHAKE

The NIST PQC standards rely heavily on the SHA-3 family of hash functions, particularly the extendable-output functions (XOFs).

Definition 2.35 (Extendable-Output Function (XOF))**.** An XOF is a function $H : \{0,1\}^* \to \{0,1\}^\infty$ that can produce an arbitrarily long output. SHAKE128 and SHAKE256 are XOFs based on the Keccak permutation.

Table 2.5: Hash functions used in NIST PQC standards

Function	Security Level	Use in PQC
SHA3-256	128-bit collision	Key derivation, hashing
SHA3-512	256-bit collision	High-security applications
SHAKE128	128-bit security	Matrix expansion, sampling
SHAKE256	256-bit security	Signatures, high-security sampling

In ML-KEM and ML-DSA, SHAKE is used to:

- Expand a seed into the public matrix $\mathbf{A}$
- Sample secret and error polynomials from seeds
- Derive shared secrets from encapsulation outputs
- Generate deterministic randomness for signing

2.5.2 Hashing in Embedded Systems

SHA-3/SHAKE operations often dominate the runtime of PQC implementations on constrained devices. Optimizations include:

- Hardware acceleration (available on some Cortex-M series)
- Assembly-optimized Keccak permutation
- Lazy evaluation (compute only needed output bytes)
- Using AES-based variants where AES acceleration is available

Table 2.6: Keccak/SHA-3 performance on ARM Cortex-M4 (cycles per byte)

Implementation	Absorb (cpb)	Squeeze (cpb)
Reference C	∼45	∼45
Optimized C	∼25	∼25
Assembly (ARMv7-M)	∼12	∼12

2.6 Chapter Summary

This chapter established the mathematical foundations for understanding post-quantum cryptography:

- **Algebraic structures**: Polynomial rings $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ enable efficient operations

- **Number Theoretic Transform**: Reduces polynomial multiplication from $O(n^2)$ to $O(n \log n)$
- **Lattices**: Discrete additive subgroups of $\mathbb{R}^n$ with geometric properties
- **LWE problem**: Finding a secret given noisy linear equations; foundation of Kyber/ML-KEM
- **SIS problem**: Finding short vectors in the kernel of a matrix; foundation of signatures
- **Module structure**: Provides balance between efficiency and security confidence
- **Lattice reduction**: BKZ algorithm determines concrete security levels
- **Hash functions**: SHA-3/SHAKE are critical components, often performance bottlenecks

Note: The mathematical foundations presented here are essential for understanding security proofs and making informed parameter choices. However, using standardized implementations with vetted parameters is strongly recommended for practical deployments.

Chapter 3

Key Encapsulation Mechanisms: ML-KEM (Kyber)

Key Encapsulation Mechanisms (KEMs) are fundamental cryptographic primitives that enable secure key exchange between parties. This chapter provides comprehensive coverage of ML-KEM, the NIST-standardized lattice-based KEM derived from the Kyber submission.

3.1 Introduction to Key Encapsulation

3.1.1 From Key Exchange to Key Encapsulation

Traditional key exchange protocols like Diffie-Hellman allow two parties to establish a shared secret over an insecure channel. However, these protocols are vulnerable to quantum attacks via Shor's algorithm. KEMs provide an alternative paradigm better suited to the post-quantum setting.

Definition 3.1 (Key Encapsulation Mechanism)**.** A Key Encapsulation Mechanism consists of three algorithms:

- $\mathsf{KeyGen}() \to (pk, sk)$: Generate a public/private key pair
- $\mathsf{Encaps}(pk) \to (ct, K)$: Using the public key, generate a ciphertext ct and shared secret K
- $\mathsf{Decaps}(sk, ct) \to K$ or $\perp$: Using the private key, recover K from ct, or output failure

Definition 3.2 (IND-CCA2 Security)**.** A KEM is *IND-CCA2 secure* (indistinguishable under adaptive chosen-ciphertext attack) if no efficient adversary with access to a decapsulation oracle can distinguish the encapsulated key from a random key, except with negligible advantage.

[Placeholder: Diagram showing KEM usage in establishing a secure channel: KeyGen, Encaps, Decaps flow between Alice and Bob]

3.1.2 KEMs vs. Public-Key Encryption

While PKE directly encrypts messages, KEMs generate random shared secrets that are then used with symmetric encryption. This separation has several advantages:

- **Simplicity**: KEM security definitions are cleaner than PKE
- **Efficiency**: Symmetric encryption handles bulk data more efficiently
- **Flexibility**: The shared secret can derive multiple keys (encryption, MAC, etc.)
- **Composability**: KEMs compose well in hybrid constructions

3.2 ML-KEM Overview

ML-KEM (Module-Lattice-Based Key-Encapsulation Mechanism) is specified in FIPS 203, published August 2024. It is derived from CRYSTALS-Kyber, one of the most extensively analyzed lattice-based schemes.

3.2.1 Design Philosophy

ML-KEM follows a modular design:

1. **K-PKE**: A CPA-secure public-key encryption scheme based on Module-LWE
2. **KEM Transform**: The Fujisaki-Okamoto transform converts K-PKE to CCA-secure KEM

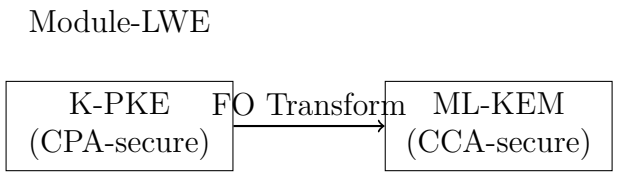


Figure 3.1: ML-KEM construction from K-PKE via Fujisaki-Okamoto transform

3.2.2 Parameter Sets

ML-KEM defines three parameter sets targeting different security levels:

Table 3.1: ML-KEM parameter sets (FIPS 203)

Parameter	ML-KEM-512	ML-KEM-768	ML-KEM-1024
NIST Security Level	1	3	5
Equivalent Classical	AES-128	AES-192	AES-256
Module rank k	2	3	4
Polynomial degree n	256	256	256
Modulus q	3329	3329	3329
Error distribution	β_3	β_2	β_2
(d_u, d_v) compression	(10, 4)	(10, 4)	(11, 5)
Public key size	800 B	1,184 B	1,568 B
Secret key size	1,632 B	2,400 B	3,168 B
Ciphertext size	768 B	1,088 B	1,568 B
Shared secret size	32 B	32 B	32 B

Note: The modulus $q = 3329$ is chosen because: (1) it is prime, (2) $q \equiv 1 \pmod{256}$, enabling NTT with $n = 256$, and (3) it fits in 12 bits, allowing efficient representation.

3.3 ML-KEM Algorithms in Detail

3.3.1 Key Generation

Algorithm 2 ML-KEM.KeyGen()

Require: Security parameter (implicit in parameter set)
Ensure: Public key pk, Secret key sk

$d \xleftarrow{\$} \{0,1\}^{256}$ ▷ Random seed
$(\rho, \sigma) \leftarrow G(d)$ ▷ G: SHA3-512
$\hat{\mathbf{A}} \leftarrow \text{Sam}(\rho)$ ▷ Generate matrix in NTT domain
for $i = 0$ to $k - 1$ **do**
    $\mathbf{s}[i] \leftarrow \text{CBD}_\eta(\text{PRF}(\sigma, i))$ ▷ Sample secret
    $\mathbf{e}[i] \leftarrow \text{CBD}_\eta(\text{PRF}(\sigma, k + i))$ ▷ Sample error
end for
$\hat{\mathbf{s}} \leftarrow \text{NTT}(\mathbf{s})$
$\hat{\mathbf{e}} \leftarrow \text{NTT}(\mathbf{e})$
$\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$ ▷ $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$
$pk \leftarrow (\text{Encode}(\hat{\mathbf{t}}), \rho)$
$sk \leftarrow (\text{Encode}(\hat{\mathbf{s}}), pk, H(pk), z)$ ▷ $z \xleftarrow{\$} \{0,1\}^{256}$
return (pk, sk)

The key generation algorithm:

1. Generates the public matrix $\mathbf{A}$ from seed ρ (saving bandwidth)

2. Samples secret $\mathbf{s}$ and error $\mathbf{e}$ from centered binomial distribution

3. Computes public key as $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$

4. Stores additional data in secret key for CCA security

3.3.2 Encapsulation

Algorithm 3 ML-KEM.Encaps(pk)

Require: Public key $pk = (\hat{\mathbf{t}}, \rho)$
Ensure: Ciphertext ct, Shared secret K

$m \xleftarrow{\$} \{0,1\}^{256}$ ▷ Random message
$(K', r) \leftarrow G(m \| H(pk))$ ▷ Derive key and randomness
$\hat{\mathbf{A}} \leftarrow \text{Sam}(\rho)$ ▷ Regenerate matrix
for $i = 0$ to $k - 1$ **do**
    $\mathbf{r}[i] \leftarrow \text{CBD}_{\eta}(\text{PRF}(r, i))$
    $\mathbf{e}_1[i] \leftarrow \text{CBD}_{\eta_2}(\text{PRF}(r, k + i))$
end for
$e_2 \leftarrow \text{CBD}_{\eta_2}(\text{PRF}(r, 2k))$
$\hat{\mathbf{r}} \leftarrow \text{NTT}(\mathbf{r})$
$\mathbf{u} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$
$v \leftarrow \text{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \text{Decompress}_q(m, 1)$
$c_1 \leftarrow \text{Compress}_q(\mathbf{u}, d_u)$
$c_2 \leftarrow \text{Compress}_q(v, d_v)$
$ct \leftarrow (c_1, c_2)$
$K \leftarrow \text{KDF}(K' \| H(ct))$
return (ct, K)

3.3.3 Decapsulation

Algorithm 4 ML-KEM.Decaps(sk, ct)

Require: Secret key $sk = (\hat{\mathbf{s}}, pk, h, z)$, Ciphertext $ct = (c_1, c_2)$
Ensure: Shared secret K

$\mathbf{u} \leftarrow \text{Decompress}_q(c_1, d_u)$
$v \leftarrow \text{Decompress}_q(c_2, d_v)$
$w \leftarrow v - \text{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \text{NTT}(\mathbf{u}))$
$m' \leftarrow \text{Compress}_q(w, 1)$ ▷ Recover message
$(K', r') \leftarrow G(m' \| h)$
$ct' \leftarrow \text{ML-KEM.Encaps\_inner}(pk, m', r')$ ▷ Re-encrypt
if $ct = ct'$ **then** ▷ Implicit rejection
    $K \leftarrow \text{KDF}(K' \| H(ct))$
else
    $K \leftarrow \text{KDF}(z \| H(ct))$ ▷ Return pseudorandom value
end if
return K

Warning: The implicit rejection mechanism is crucial for CCA security. If ciphertext validation fails, the algorithm returns a pseudorandom value derived from secret z rather than an error, preventing chosen-ciphertext attacks.

3.3.4 Core Operations

Compression and Decompression

To reduce ciphertext size, ML-KEM compresses polynomial coefficients:

Definition 3.3 (Compression)**.** For $x \in \mathbb{Z}_q$ and $d < \lceil \log_2 q \rceil$:

$$\text{Compress}_q(x, d) = \left\lfloor \frac{2^d}{q} \cdot x \right\rceil \mod 2^d$$

$$\text{Decompress}_q(x, d) = \left\lfloor \frac{q}{2^d} \cdot x \right\rceil$$

Compression introduces a small additional error bounded by $\lfloor q/2^{d+1} \rfloor$.

Centered Binomial Distribution Sampling

Algorithm 5 $\text{CBD}_\eta(\text{bytes})$

Require: 64η bytes of randomness
Ensure: Polynomial with coefficients in $\{-\eta, \ldots, \eta\}$

1: Convert bytes to bits $b_0, b_1, \ldots, b_{512\eta-1}$
2: **for** $i = 0$ to 255 **do**
3: $\quad a \leftarrow \sum_{j=0}^{\eta-1} b_{2i\eta+j}$
4: $\quad b \leftarrow \sum_{j=0}^{\eta-1} b_{2i\eta+\eta+j}$
5: $\quad f[i] \leftarrow a - b$
6: **end for**
7: **return** f

Matrix Generation from Seed

The public matrix $\mathbf{A}$ is generated deterministically from seed ρ:

```
void sample_matrix(poly A[K][K], const uint8_t rho[32]) {
    for (int i = 0; i < K; i++) {
        for (int j = 0; j < K; j++) {
            uint8_t seed[34];
            memcpy(seed, rho, 32);
            seed[32] = j;  // Column index
            seed[33] = i;  // Row index
            shake128_absorb_once(seed, 34);
            sample_ntt_poly(&A[i][j]);  // Rejection sampling
        }
    }
}
```

Listing 3.1: Matrix sampling (conceptual)

3.4 Security Analysis

3.4.1 Hardness Assumptions

ML-KEM security relies on two computational assumptions:

Theorem 3.1 (ML-KEM Security)**.** *If the Module-LWE problem is hard, then ML-KEM is IND-CCA2 secure in the random oracle model.*

The security reduction proceeds as:

$$\text{IND-CCA2 (ML-KEM)} \Leftarrow \text{IND-CPA (K-PKE)} \Leftarrow \text{MLWE}_{n,k,q,\eta}$$

3.4.2 Concrete Security Estimates

Security is estimated against the best-known attacks:

Table 3.2: ML-KEM security estimates (Core-SVP model)

Parameter Set	Classical	Quantum	Target
ML-KEM-512	118 bits	107 bits	128 bits
ML-KEM-768	183 bits	166 bits	192 bits
ML-KEM-1024	256 bits	232 bits	256 bits

3.4.3 Known Attack Vectors

Lattice Attacks

The primary attack uses BKZ lattice reduction. The primal attack embeds the LWE instance into a unique-SVP problem; the dual attack targets the SIS structure.

Side-Channel Attacks

ML-KEM implementations must protect against:

- **Timing attacks**: All operations must be constant-time
- **Power analysis**: Particularly during NTT and sampling
- **Fault attacks**: The FO transform provides some protection

Multi-Target Attacks

With many public keys, attackers may gain advantages. ML-KEM's parameters account for multi-target scenarios.

3.5 Implementation Considerations

3.5.1 Memory Layout

Efficient memory management is crucial for constrained devices:

Table 3.3: ML-KEM-768 memory requirements

Data Structure	Size	Notes
Single polynomial (coefficients)	512 B	256×16-bit
Single polynomial (NTT)	512 B	256×16-bit
Matrix $\mathbf{A}$ (full)	4,608 B	$3 \times 3 \times 512$ B
Matrix $\mathbf{A}$ (on-the-fly)	512 B	Generate per-column
Public key	1,184 B	Compressed
Secret key	2,400 B	Includes pk
Ciphertext	1,088 B	Compressed
KeyGen stack (optimized)	∼2,400 B	
Encaps stack (optimized)	∼2,400 B	
Decaps stack (optimized)	∼2,600 B	

3.5.2 Optimization Strategies

NTT Optimization

The NTT is performance-critical. Key optimizations include:

- **Montgomery multiplication**: Avoid expensive modular reduction
- **Barrett reduction**: For final coefficient reduction
- **Precomputed twiddle factors**: Store ω^i values
- **In-place computation**: Minimize memory usage
- **Loop unrolling**: Reduce branch overhead

```
// Montgomery reduction: compute a * R^{-1} mod q
// where R = 2^16 and q = 3329
int16_t montgomery_reduce(int32_t a) {
    int16_t t;
    t = (int16_t)a * QINV;  // QINV = -3327 mod 2^16
    t = (a - (int32_t)t * Q) >> 16;
    return t;
}
```

Listing 3.2: Montgomery reduction for $q = 3329$

Memory-Time Tradeoffs

Table 3.4: Implementation variants and their tradeoffs

Variant	Speed	Stack	Code Size
Reference (clean)	Baseline	High	Small
Speed-optimized (m4fspeed)	∼2× faster	High	Large
Stack-optimized (m4fstack)	∼1.5× faster	Low	Medium

3.5.3 Constant-Time Implementation

All operations must execute in constant time to prevent timing side-channels:

```
// Constant-time conditional move: r = (b ? a : r)
void cmov(uint8_t *r, const uint8_t *a, size_t len, uint8_t
    b) {
     b = -b;  // 0x00 or 0xFF
     for (size_t i = 0; i < len; i++) {
         r[i] ^= b & (r[i] ^ a[i]);
     }
}

// Constant-time comparison: returns 0 if equal
int verify(const uint8_t *a, const uint8_t *b, size_t len) {
     uint8_t r = 0;
     for (size_t i = 0; i < len; i++) {
         r |= a[i] ^ b[i];
     }
     return (-(int)r) >> 31;  // 0 or -1
}
```

Listing 3.3: Constant-time conditional move

3.6 Performance Benchmarks

3.6.1 ARM Cortex-M4 Benchmarks

The following benchmarks are from the pqm4 project on the NUCLEO-L4R5ZI board (STM32L4R5ZI, Cortex-M4 @ 80 MHz):

Table 3.5: ML-KEM performance on ARM Cortex-M4 (cycles)

Scheme	Variant	KeyGen	Encaps	Decaps
ML-KEM-512	clean	519,853	666,050	715,491
	m4fspeed	392,295	391,355	428,037
	m4fstack	415,631	532,478	575,227
ML-KEM-768	clean	867,222	1,085,400	1,159,285
	m4fspeed	642,163	659,069	708,044
	m4fstack	676,896	854,339	912,308
ML-KEM-1024	clean	1,302,069	1,578,656	1,686,574
	m4fspeed	1,019,877	1,031,009	1,094,181
	m4fstack	1,076,065	1,299,949	1,380,086

Table 3.6: ML-KEM stack usage on ARM Cortex-M4 (bytes)

Scheme	Variant	KeyGen	Encaps	Decaps
ML-KEM-512	clean	2,896	3,936	4,288
	m4fspeed	2,632	2,720	3,080
	m4fstack	2,300	2,348	2,332
ML-KEM-768	clean	3,920	4,448	4,800
	m4fspeed	3,144	3,232	3,592
	m4fstack	2,556	2,604	2,588
ML-KEM-1024	clean	4,944	5,472	5,824
	m4fspeed	3,656	3,744	4,104
	m4fstack	2,812	2,860	2,844

3.6.2 Comparison with Classical Algorithms

Table 3.7: ML-KEM vs. classical algorithms on Cortex-M4

Algorithm	Security	KeyGen	Operation	Sizes (pk+ct)
ECDH P-256	128-bit	1.4M	1.4M	64 + 64 B
X25519	128-bit	0.8M	0.8M	32 + 32 B
RSA-2048	112-bit	300M+	0.3M/30M	256 + 256 B
ML-KEM-512	128-bit (PQ)	392K	391K/428K	800 + 768 B
ML-KEM-768	192-bit (PQ)	642K	659K/708K	1,184 + 1,088 B

Note: ML-KEM is **faster** than ECDH on Cortex-M4 while providing post-quantum security. The main cost is larger key and ciphertext sizes, which impacts bandwidth-constrained applications.

3.6.3 Desktop/Server Benchmarks

For comparison, here are benchmarks on an Intel Core i7-6600U (Skylake) with AVX2:

Table 3.8: ML-KEM performance on x86-64 with AVX2 (cycles)

Scheme	KeyGen	Encaps	Decaps
ML-KEM-512	29,000	37,000	32,000
ML-KEM-768	48,000	58,000	52,000
ML-KEM-1024	69,000	84,000	76,000

3.6.4 Hashing Overhead

A significant portion of ML-KEM runtime is spent in hash functions:

Table 3.9: Percentage of cycles spent in SHA-3/SHAKE (Cortex-M4)

Scheme	KeyGen	Encaps	Decaps
ML-KEM-512	42%	54%	51%
ML-KEM-768	48%	57%	55%
ML-KEM-1024	52%	60%	57%

3.7 Code Examples

3.7.1 Using liboqs (C)

```c
#include <oqs/oqs.h>
#include <stdio.h>
#include <string.h>

int main() {
    OQS_KEM *kem = OQS_KEM_new(OQS_KEM_alg_ml_kem_768);
    if (kem == NULL) {
        printf("ML-KEM-768 not available\n");
        return 1;
    }

    // Allocate memory
    uint8_t *public_key = malloc(kem->length_public_key);
    uint8_t *secret_key = malloc(kem->length_secret_key);
    uint8_t *ciphertext = malloc(kem->length_ciphertext);
    uint8_t *shared_secret_e =
       malloc(kem->length_shared_secret);
```

```
17      uint8_t *shared_secret_d =
            malloc(kem->length_shared_secret);
18
19      // Key generation (performed by receiver)
20      OQS_KEM_keypair(kem, public_key, secret_key);
21      printf("Generated keypair: pk=%zu bytes, sk=%zu bytes\n",
22             kem->length_public_key, kem->length_secret_key);
23
24      // Encapsulation (performed by sender)
25      OQS_KEM_encaps(kem, ciphertext, shared_secret_e,
            public_key);
26      printf("Encapsulated: ct=%zu bytes, K=%zu bytes\n",
27             kem->length_ciphertext,
                kem->length_shared_secret);
28
29      // Decapsulation (performed by receiver)
30      OQS_KEM_decaps(kem, shared_secret_d, ciphertext,
            secret_key);
31
32      // Verify shared secrets match
33      if (memcmp(shared_secret_e, shared_secret_d,
34                 kem->length_shared_secret) == 0) {
35          printf("Key exchange successful!\n");
36      }
37
38      // Cleanup
39      OQS_MEM_secure_free(secret_key, kem->length_secret_key);
40      OQS_MEM_secure_free(shared_secret_e,
            kem->length_shared_secret);
41      OQS_MEM_secure_free(shared_secret_d,
            kem->length_shared_secret);
42      free(public_key);
43      free(ciphertext);
44      OQS_KEM_free(kem);
45
46      return 0;
47  }
```

Listing 3.4: ML-KEM key exchange using liboqs

3.7.2 Using pqcrypto (Rust)

```rust
use pqcrypto_kyber::kyber768;
use pqcrypto_traits::kem::{PublicKey, SecretKey, Ciphertext,
    SharedSecret};

fn main() {
    // Key generation
    let (pk, sk) = kyber768::keypair();
    println!("Public key: {} bytes", pk.as_bytes().len());
    println!("Secret key: {} bytes", sk.as_bytes().len());

    // Encapsulation
    let (ss_sender, ct) = kyber768::encapsulate(&pk);
    println!("Ciphertext: {} bytes", ct.as_bytes().len());

    // Decapsulation
    let ss_receiver = kyber768::decapsulate(&ct, &sk);

    // Verify
    assert_eq!(ss_sender.as_bytes(), ss_receiver.as_bytes());
    println!("Shared secret established: {} bytes",
             ss_sender.as_bytes().len());
}
```

Listing 3.5: ML-KEM key exchange in Rust

3.7.3 Using liboqs-python

```python
import oqs

def ml_kem_key_exchange():
    """Demonstrate ML-KEM key exchange."""

    # Create KEM instance
    kem = oqs.KeyEncapsulation("ML-KEM-768")

    # Receiver generates keypair
    public_key = kem.generate_keypair()
    print(f"Public key: {len(public_key)} bytes")
    print(f"Secret key: {len(kem.export_secret_key())}
        bytes")

```

```
14      # Sender encapsulates
15      ciphertext, shared_secret_sender =
            kem.encap_secret(public_key)
16      print(f"Ciphertext: {len(ciphertext)} bytes")
17      print(f"Shared secret: {len(shared_secret_sender)}
            bytes")
18
19      # Receiver decapsulates
20      shared_secret_receiver = kem.decap_secret(ciphertext)
21
22      # Verify
23      assert shared_secret_sender == shared_secret_receiver
24      print("Key exchange successful!")
25
26      return shared_secret_sender
27
28  if __name__ == "__main__":
29      shared_key = ml_kem_key_exchange()
30      print(f"Established key: {shared_key.hex()[:32]}...")
```

Listing 3.6: ML-KEM key exchange in Python

3.8 Hybrid Key Exchange

During the transition period, hybrid schemes combine classical and post-quantum algorithms to hedge against potential weaknesses in either.

3.8.1 X25519-ML-KEM Hybrid

The recommended hybrid approach combines X25519 and ML-KEM:

Algorithm 6 Hybrid X25519-ML-KEM Key Exchange

// Key Generation
$(pk_{x25519}, sk_{x25519}) \leftarrow$ X25519.KeyGen()
$(pk_{mlkem}, sk_{mlkem}) \leftarrow$ ML-KEM.KeyGen()
$pk \leftarrow pk_{x25519} \| pk_{mlkem}$
$sk \leftarrow sk_{x25519} \| sk_{mlkem}$

// Encapsulation
$(ss_{x25519}, ct_{x25519}) \leftarrow$ X25519.DH(pk_{x25519})
$(ss_{mlkem}, ct_{mlkem}) \leftarrow$ ML-KEM.Encaps(pk_{mlkem})
$ct \leftarrow ct_{x25519} \| ct_{mlkem}$
$K \leftarrow \text{KDF}(ss_{x25519} \| ss_{mlkem})$

// Decapsulation
$ss_{x25519} \leftarrow$ X25519.DH(sk_{x25519}, ct_{x25519})
$ss_{mlkem} \leftarrow$ ML-KEM.Decaps(sk_{mlkem}, ct_{mlkem})
$K \leftarrow \text{KDF}(ss_{x25519} \| ss_{mlkem})$

Table 3.10: Hybrid scheme sizes

Scheme	Public Key	Ciphertext	Security
X25519 only	32 B	32 B	Classical 128-bit
ML-KEM-768 only	1,184 B	1,088 B	PQ 192-bit
X25519 + ML-KEM-768	1,216 B	1,120 B	Both

3.9 IoT Deployment Considerations

3.9.1 Device Class Recommendations

Table 3.11: ML-KEM suitability by IoT device class

Device Class	RAM	ML-KEM-512	ML-KEM-768	ML-KEM-1024
Class 0 (<10 KB)	<10 KB	No	No	No
Class 1 (~10 KB)	~10 KB	Limited	No	No
Class 2 (~50 KB)	~50 KB	Yes	Yes	Limited
Class 3 (>256 KB)	>256 KB	Yes	Yes	Yes

3.9.2 Bandwidth Impact

For bandwidth-constrained networks (LoRaWAN, NB-IoT), ML-KEM's larger sizes are significant:

Table 3.12: Transmission time comparison (LoRaWAN SF7, 125 kHz)

Operation	ECDH P-256	ML-KEM-512	ML-KEM-768
Public key TX	12 ms	154 ms	228 ms
Response TX	12 ms	148 ms	210 ms
Total exchange	24 ms	302 ms	438 ms

3.9.3 Energy Consumption

Table 3.13: Estimated energy consumption on Cortex-M4 @ 3.3V, 80 MHz

Operation	ML-KEM-512	ML-KEM-768	ML-KEM-1024
KeyGen	48 µJ	79 µJ	126 µJ
Encaps	48 µJ	81 µJ	127 µJ
Decaps	53 µJ	87 µJ	135 µJ
Full exchange	149 µJ	247 µJ	388 µJ

3.10 Chapter Summary

This chapter provided comprehensive coverage of ML-KEM, the NIST-standardized post-quantum key encapsulation mechanism:

- **Design**: ML-KEM uses Module-LWE with Fujisaki-Okamoto transform for CCA security
- **Parameters**: Three variants (512/768/1024) for security levels 1/3/5
- **Algorithms**: Detailed KeyGen, Encaps, Decaps with compression and NTT
- **Security**: Based on Module-LWE hardness; implicit rejection prevents CCA attacks
- **Performance**: Faster than ECDH on Cortex-M4; 2–3 KB stack usage
- **Sizes**: Public keys 800–1,568 B; ciphertexts 768–1,568 B
- **IoT**: Suitable for Class 2+ devices; bandwidth is main constraint
- **Hybrid**: Combine with X25519 during transition period

Note: For most IoT applications requiring post-quantum key exchange, **ML-KEM-768** offers the best balance of security (192-bit post-quantum) and efficiency. Use ML-KEM-512 for extremely constrained devices where 128-bit security suffices.

Chapter 4

Digital Signature Schemes

Digital signatures provide authentication, integrity, and non-repudiation—essential security properties for IoT systems. This chapter covers the three post-quantum signature schemes selected by NIST: ML-DSA (Dilithium), SLH-DSA (SPHINCS+), and Falcon.

4.1 Introduction to Digital Signatures

Definition 4.1 (Digital Signature Scheme)**.** A digital signature scheme consists of three algorithms:

- $\mathsf{KeyGen}() \to (pk, sk)$: Generate a public/private key pair
- $\mathsf{Sign}(sk, m) \to \sigma$: Create a signature σ on message m
- $\mathsf{Verify}(pk, m, \sigma) \to \{0, 1\}$: Verify signature validity

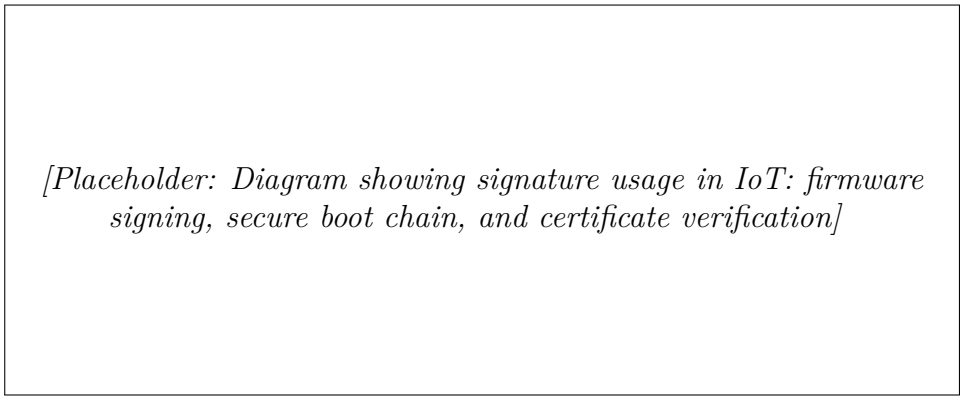
Definition 4.2 (EUF-CMA Security)**.** A signature scheme is *existentially unforgeable under chosen message attack* (EUF-CMA) if no efficient adversary with access to a signing oracle can produce a valid signature on a new message.

4.1.1 Signatures in IoT

Digital signatures serve critical functions in IoT:

- **Firmware updates**: Authenticate software from legitimate vendors
- **Device attestation**: Prove device identity and configuration
- **Data integrity**: Sign sensor readings for audit trails
- **Certificate authentication**: TLS/DTLS handshakes

- **Secure boot**: Verify bootloader and kernel integrity



4.1.2 Comparison Overview

NIST selected three signature schemes with complementary characteristics:

Table 4.1: NIST post-quantum signature schemes comparison

Scheme	Basis	pk Size	sig Size	Sign	Verify
ML-DSA-65	Lattice	1,952 B	3,293 B	Fast	Fast
Falcon-512	Lattice	897 B	666 B	Medium	Very Fast
SLH-DSA-128s	Hash	32 B	7,856 B	Slow	Medium

4.2 ML-DSA (Dilithium)

ML-DSA (Module-Lattice-Based Digital Signature Algorithm), specified in FIPS 204, is based on the CRYSTALS-Dilithium submission. It offers balanced performance and is the recommended general-purpose post-quantum signature scheme.

4.2.1 Design Overview

ML-DSA is a Fiat-Shamir signature scheme based on the "Fiat-Shamir with Aborts" paradigm:

1. Generate a commitment from random masking vector
2. Hash message and commitment to produce challenge
3. Compute response using secret key
4. **Reject and restart** if response would leak secret information

The rejection sampling ensures signatures don't reveal the secret key, but introduces variable signing time.

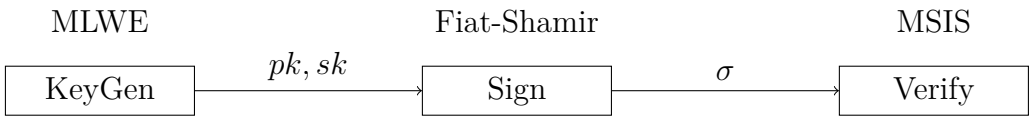


Figure 4.1: ML-DSA construction: security from MLWE and MSIS

4.2.2 Parameters

Table 4.2: ML-DSA parameter sets (FIPS 204)

Parameter	**ML-DSA-44**	**ML-DSA-65**	**ML-DSA-87**
NIST Security Level	2	3	5
Module dimensions (k, ℓ)	$(4, 4)$	$(6, 5)$	$(8, 7)$
Polynomial degree n	256	256	256
Modulus q	8,380,417	8,380,417	8,380,417
Challenge weight τ	39	49	60
Coefficient bound γ_1	2^{17}	2^{19}	2^{19}
Coefficient bound γ_2	95,232	261,888	261,888
Secret key range η	2	4	2
Hint bits ω	80	55	75
Public key size	1,312 B	1,952 B	2,592 B
Secret key size	2,560 B	4,032 B	4,896 B
Signature size	2,420 B	3,293 B	4,595 B

Note: The modulus $q = 8380417 = 2^{23} - 2^{13} + 1$ is chosen for efficient NTT: it is prime, $q \equiv 1 \pmod{512}$, and allows fast reduction using the special form.

4.2.3 Algorithms

Key Generation

Algorithm 7 ML-DSA.KeyGen()

Require: Security parameter (implicit)
Ensure: Public key pk, Secret key sk

$\zeta \xleftarrow{\$} \{0,1\}^{256}$
$(\rho, \rho', K) \leftarrow H(\zeta)$ ▷ Expand seed
$\hat{\mathbf{A}} \leftarrow \text{ExpandA}(\rho)$ ▷ Public matrix in NTT domain
$(\mathbf{s}_1, \mathbf{s}_2) \leftarrow \text{ExpandS}(\rho')$ ▷ Secret vectors, coeffs in $[-\eta, \eta]$
$\hat{\mathbf{s}}_1 \leftarrow \text{NTT}(\mathbf{s}_1)$
$\mathbf{t} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \hat{\mathbf{s}}_1) + \mathbf{s}_2$ ▷ $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
$(\mathbf{t}_1, \mathbf{t}_0) \leftarrow \text{Power2Round}(\mathbf{t})$ ▷ Split for compression
$pk \leftarrow (\rho, \mathbf{t}_1)$
$tr \leftarrow H(pk)$
$sk \leftarrow (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$
return (pk, sk)

Signing

Algorithm 8 ML-DSA.Sign(sk, M)

Require: Secret key $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$, Message M
Ensure: Signature σ

$\hat{\mathbf{A}} \leftarrow \text{ExpandA}(\rho)$
$\mu \leftarrow H(tr \| M)$ ▷ Message representative
$\kappa \leftarrow 0$; $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
$\rho' \leftarrow H(K \| \mu)$ ▷ Deterministic or random
while $(\mathbf{z}, \mathbf{h}) = \perp$ **do** ▷ Rejection sampling loop
    $\mathbf{y} \leftarrow \text{ExpandMask}(\rho', \kappa)$ ▷ Masking vector
    $\hat{\mathbf{y}} \leftarrow \text{NTT}(\mathbf{y})$
    $\mathbf{w} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \hat{\mathbf{y}})$
    $\mathbf{w}_1 \leftarrow \text{HighBits}(\mathbf{w})$
    $\tilde{c} \leftarrow H(\mu \| \mathbf{w}_1)$ ▷ Challenge hash
    $c \leftarrow \text{SampleInBall}(\tilde{c})$ ▷ Challenge polynomial
    $\hat{c} \leftarrow \text{NTT}(c)$
    $\mathbf{z} \leftarrow \mathbf{y} + \text{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s}}_1)$
    $\mathbf{r}_0 \leftarrow \text{LowBits}(\mathbf{w} - \text{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s}}_2))$
    if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ **or** $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ **then**
        $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$ ▷ Reject: would leak secret
    else
        $\mathbf{h} \leftarrow \text{MakeHint}(\mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2)$
        if $\|\mathbf{h}\|_1 > \omega$ **then**
            $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
        end if
    end if
    $\kappa \leftarrow \kappa + 1$
end while
return $\sigma \leftarrow (\tilde{c}, \mathbf{z}, \mathbf{h})$

Warning: The rejection sampling loop typically iterates 4–7 times on average. This makes signing time variable, which may be a concern for real-time systems. Implementations should account for worst-case timing.

Verification

Algorithm 9 ML-DSA.Verify(pk, M, σ)

Require: Public key $pk = (\rho, \mathbf{t}_1)$, Message M, Signature $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$
Ensure: Accept (1) or Reject (0)
1: **if** $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ **then**
2: **return** 0
3: **end if**
4: $\hat{\mathbf{A}} \leftarrow \text{ExpandA}(\rho)$
5: $\mu \leftarrow H(H(pk)\|M)$
6: $c \leftarrow \text{SampleInBall}(\tilde{c})$
7: $\hat{c} \leftarrow \text{NTT}(c)$; $\hat{\mathbf{z}} \leftarrow \text{NTT}(\mathbf{z})$
8: $\mathbf{w}' \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \hat{\mathbf{z}} - \hat{c} \circ \text{NTT}(\mathbf{t}_1 \cdot 2^d))$
9: $\mathbf{w}'_1 \leftarrow \text{UseHint}(\mathbf{h}, \mathbf{w}')$
10: $\tilde{c}' \leftarrow H(\mu\|\mathbf{w}'_1)$
11: **return** $[\tilde{c} = \tilde{c}']$ **and** $[\|\mathbf{h}\|_1 \leq \omega]$

4.2.4 Performance

Table 4.3: ML-DSA performance on ARM Cortex-M4 (cycles)

Scheme	Variant	KeyGen	Sign (avg)	Verify
ML-DSA-44	clean	1,202,964	4,428,842	1,208,160
	m4f	800,974	2,292,188	786,486
ML-DSA-65	clean	2,009,972	6,659,854	1,973,700
	m4f	1,406,298	3,948,962	1,380,582
ML-DSA-87	clean	3,043,684	8,509,260	3,019,802
	m4f	2,222,174	5,383,044	2,148,702

Table 4.4: ML-DSA stack usage on ARM Cortex-M4 (bytes)

Scheme	Variant	KeyGen	Sign	Verify
ML-DSA-44	m4f	10,328	24,512	10,680
ML-DSA-65	m4f	14,472	34,752	15,336
ML-DSA-87	m4f	18,104	43,968	19,992

Note: ML-DSA requires significantly more stack space than ML-KEM due to the larger matrices and the need to store intermediate values during rejection sampling. Ensure adequate stack allocation in embedded systems.

4.3 SLH-DSA (SPHINCS+)

SLH-DSA (Stateless Hash-Based Digital Signature Algorithm), specified in FIPS 205, provides an alternative to lattice-based signatures with security based solely on hash function properties.

4.3.1 Design Philosophy

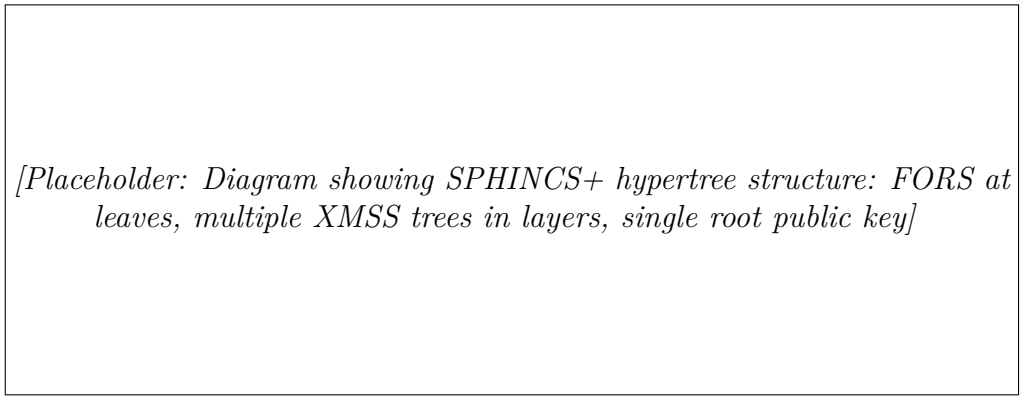
SLH-DSA is conservative by design:

- **Minimal assumptions**: Security relies only on hash function properties (not lattice problems)
- **Well-understood**: Hash-based signatures have decades of cryptanalysis
- **Stateless**: No state management required (unlike XMSS/LMS)
- **Tradeoff flexibility**: Multiple parameter sets for size/speed balance

Hypertree Structure

SLH-DSA uses a hierarchical structure:

1. **FORS**: Few-time signature scheme for signing message digest
2. **WOTS+**: One-time signatures to authenticate FORS keys
3. **Hypertree**: Multiple layers of Merkle trees to reduce public key size



4.3.2 Parameters

SLH-DSA offers many parameter sets with different tradeoffs:

Table 4.5: Selected SLH-DSA parameter sets (FIPS 205)

Parameter Set	Security	pk	sk	sig	Sign	Verify
Small signatures ("s" suffix)						
SLH-DSA-SHA2-128s	Level 1	32 B	64 B	7,856 B	Slow	Medium
SLH-DSA-SHA2-192s	Level 3	48 B	96 B	16,224 B	Slow	Medium
SLH-DSA-SHA2-256s	Level 5	64 B	128 B	29,792 B	Slow	Medium
Fast signing ("f" suffix)						
SLH-DSA-SHA2-128f	Level 1	32 B	64 B	17,088 B	Fast	Fast
SLH-DSA-SHA2-192f	Level 3	48 B	96 B	35,664 B	Fast	Fast
SLH-DSA-SHA2-256f	Level 5	64 B	128 B	49,856 B	Fast	Fast
SHAKE-based variants						
SLH-DSA-SHAKE-128s	Level 1	32 B	64 B	7,856 B	Slow	Medium
SLH-DSA-SHAKE-128f	Level 1	32 B	64 B	17,088 B	Fast	Fast

Warning: SLH-DSA signatures are significantly larger than ML-DSA or Falcon signatures. The "f" (fast) variants are even larger but sign/verify faster. Choose based on your bandwidth vs. computation constraints.

4.3.3 Algorithm Sketch

Key Generation

1. Generate random seeds: SK.seed, SK.prf, PK.seed
2. Compute hypertree root as public key
3. Store seeds as secret key

Signing

1. Compute randomized message digest
2. Select FORS keypair based on digest
3. Sign digest with FORS
4. Authenticate FORS public key through hypertree
5. Signature = FORS signature + authentication path

Verification

1. Recompute message digest
2. Verify FORS signature

3. Verify authentication path up to root
4. Compare computed root with public key

4.3.4 Performance

Table 4.6: SLH-DSA performance on ARM Cortex-M4 (cycles, millions)

Parameter Set	KeyGen	Sign	Verify
SLH-DSA-SHAKE-128s	36.8 M	978 M	37.8 M
SLH-DSA-SHAKE-128f	2.9 M	51.7 M	3.5 M
SLH-DSA-SHAKE-192s	61.9 M	1,271 M	42.4 M
SLH-DSA-SHAKE-192f	4.2 M	94.9 M	5.9 M
SLH-DSA-SHAKE-256s	225 M	3,817 M	103 M
SLH-DSA-SHAKE-256f	5.9 M	151 M	9.9 M

Table 4.7: SLH-DSA vs. ML-DSA: Size and speed tradeoffs (Level 1)

Scheme	pk + sig	Sign (M4)	Verify (M4)	Confidence
ML-DSA-44	3,732 B	2.3 M	0.8 M	High
SLH-DSA-SHAKE-128s	7,888 B	978 M	37.8 M	Very High
SLH-DSA-SHAKE-128f	17,120 B	51.7 M	3.5 M	Very High

4.3.5 Use Cases

SLH-DSA is best suited for:

- **Root certificates**: Long-lived trust anchors where signing is rare
- **Firmware signing**: Server-side signing, device-side verification
- **Conservative deployments**: When lattice assumptions are considered risky
- **Compliance**: When hash-based signatures are mandated

Note: For IoT devices that primarily **verify** signatures (e.g., firmware updates), the "f" variants offer reasonable verification times. Signing is typically done on more powerful servers.

4.4 Falcon

Falcon is an NTRU-lattice-based signature scheme offering the smallest combined public key and signature sizes among NIST selections. It is selected for standardization but requires additional specification work.

4.4.1 Design Overview

Falcon uses the GPV framework (Gentry-Peikert-Vaikuntanathan) with:

- **NTRU lattices**: Structured lattices with compact representation
- **Fast Fourier Sampling**: Efficient trapdoor sampling using FFT
- **Gaussian sampling**: True discrete Gaussian (unlike Dilithium's rejection)

Definition 4.3 (NTRU Problem)**.** Given $h = g \cdot f^{-1} \mod q$ in $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ where f, g have small coefficients, find f and g.

4.4.2 Parameters

Table 4.8: Falcon parameter sets

Parameter	Falcon-512	Falcon-1024
NIST Security Level	1	5
Ring degree n	512	1024
Modulus q	12,289	12,289
Gaussian std. dev. σ	165.74	168.39
Public key size	897 B	1,793 B
Secret key size	1,281 B	2,305 B
Signature size (avg)	666 B	1,280 B
Signature size (max)	809 B	1,577 B

Note: Falcon signatures have variable size due to compression. The table shows average and maximum sizes. Implementations must handle the maximum to avoid buffer overflows.

4.4.3 Algorithm Highlights

Key Generation

1. Generate NTRU key pair: small polynomials f, g, F, G with $fG - gF = q$
2. Public key: $h = g \cdot f^{-1} \mod q$
3. Secret key: Precomputed FFT tree for fast sampling

Signing (Simplified)

1. Hash message to point c in lattice
2. Use trapdoor to sample short vector (s_1, s_2) with $s_1 + s_2 \cdot h = c$
3. Signature: compressed s_2 (or (s_1, s_2))

Verification

1. Decompress signature to get s_2
2. Compute $s_1 = c - s_2 \cdot h$
3. Accept if (s_1, s_2) is sufficiently short

4.4.4 Performance

Table 4.9: Falcon performance on ARM Cortex-M4 (cycles)

Scheme	Variant	KeyGen	Sign	Verify
Falcon-512	clean	90.1 M	14.9 M	0.41 M
	opt	53.3 M	7.0 M	0.34 M
Falcon-1024	clean	265 M	32.0 M	0.84 M
	opt	134 M	14.9 M	0.69 M

Table 4.10: Falcon RAM requirements

Operation	Falcon-512	Falcon-1024
Key generation	14 KB	28 KB
Signing	39 KB	79 KB
Verification	2 KB	3 KB

Warning: Falcon's Gaussian sampling is notoriously difficult to implement securely. Side-channel resistant implementations require careful attention to constant-time floating-point operations or integer-only approximations.

4.4.5 Falcon vs. ML-DSA

Table 4.11: Falcon vs. ML-DSA comparison

Aspect	Falcon-512	ML-DSA-44
Public key	897 B	1,312 B
Signature	666 B	2,420 B
Combined (pk + sig)	1,563 B	3,732 B
Sign cycles (M4)	7.0 M	2.3 M
Verify cycles (M4)	0.34 M	0.79 M
Implementation complexity	High	Medium
Side-channel hardening	Difficult	Easier

Recommendation:

- Use **Falcon** when bandwidth is critical and implementation resources are available
- Use **ML-DSA** for general purposes, especially when simpler implementation is preferred

4.5 Comparative Analysis

4.5.1 Size Comparison

Table 4.12: Complete size comparison of PQ signature schemes

Scheme	pk	sk	sig	pk + sig
Security Level 1 (128-bit)				
ECDSA P-256 (classical)	64 B	32 B	64 B	128 B
ML-DSA-44	1,312 B	2,560 B	2,420 B	3,732 B
Falcon-512	897 B	1,281 B	666 B	1,563 B
SLH-DSA-SHA2-128s	32 B	64 B	7,856 B	7,888 B
SLH-DSA-SHA2-128f	32 B	64 B	17,088 B	17,120 B
Security Level 3 (192-bit)				
ECDSA P-384 (classical)	96 B	48 B	96 B	192 B
ML-DSA-65	1,952 B	4,032 B	3,293 B	5,245 B
SLH-DSA-SHA2-192s	48 B	96 B	16,224 B	16,272 B
Security Level 5 (256-bit)				
ML-DSA-87	2,592 B	4,896 B	4,595 B	7,187 B
Falcon-1024	1,793 B	2,305 B	1,280 B	3,073 B
SLH-DSA-SHA2-256s	64 B	128 B	29,792 B	29,856 B

[Placeholder: Bar chart comparing pk+sig sizes for ECDSA, ML-DSA, Falcon, and SLH-DSA at each security level]

4.5.2 Performance Comparison

Table 4.13: Signature scheme performance summary (Cortex-M4, Level 1)

Scheme	KeyGen	Sign	Verify	Notes
ECDSA P-256	1.4 M	1.8 M	2.1 M	Classical only
ML-DSA-44	0.8 M	2.3 M	0.8 M	Best general choice
Falcon-512	53.3 M	7.0 M	0.34 M	Smallest sizes
SLH-DSA-128f	2.9 M	51.7 M	3.5 M	Conservative
SLH-DSA-128s	36.8 M	978 M	37.8 M	Smallest sig

4.5.3 Selection Guidelines

Table 4.14: Signature scheme selection by use case

Use Case	ML-DSA	Falcon	SLH-DSA-f	SLH-DSA-s
TLS certificates	✓	✓		
Firmware signing	✓	✓	✓	✓
Real-time signing	✓			
Bandwidth critical		✓		
Code signing	✓	✓	✓	
Root CA certificates				✓
General purpose	✓			
Conservative choice			✓	✓

4.6 Code Examples

4.6.1 ML-DSA with liboqs

```c
#include <oqs/oqs.h>
#include <string.h>

int sign_verify_example(void) {
    OQS_SIG *sig = OQS_SIG_new(OQS_SIG_alg_ml_dsa_65);
    if (sig == NULL) return -1;

    uint8_t *public_key = malloc(sig->length_public_key);
    uint8_t *secret_key = malloc(sig->length_secret_key);
    uint8_t *signature = malloc(sig->length_signature);
    size_t signature_len;

    // Key generation
```

```
    OQS_SIG_keypair(sig, public_key, secret_key);

    // Sign message
    const uint8_t message[] = "Hello, Post-Quantum World!";
    OQS_SIG_sign(sig, signature, &signature_len,
                 message, sizeof(message), secret_key);

    printf("Signature: %zu bytes\n", signature_len);

    // Verify signature
    OQS_STATUS result = OQS_SIG_verify(sig, message,
       sizeof(message),
                                       signature,
                                          signature_len,
                                          public_key);

    if (result == OQS_SUCCESS) {
        printf("Signature valid!\n");
    }

    // Cleanup
    OQS_MEM_secure_free(secret_key, sig->length_secret_key);
    free(public_key);
    free(signature);
    OQS_SIG_free(sig);

    return (result == OQS_SUCCESS) ? 0 : -1;
}
```

Listing 4.1: ML-DSA signing and verification

4.6.2 SPHINCS+ with PQClean

```
#include "api.h"  // PQClean SPHINCS+ API
#include <string.h>

int sphincs_example(void) {
    uint8_t pk[CRYPTO_PUBLICKEYBYTES];
    uint8_t sk[CRYPTO_SECRETKEYBYTES];
    uint8_t sig[CRYPTO_BYTES];
    size_t siglen;

```

```
    const uint8_t msg[] = "Firmware v2.0 - SHA256:
        abc123...";

    // Generate keypair
    crypto_sign_keypair(pk, sk);

    // Sign (detached signature)
    crypto_sign_signature(sig, &siglen, msg, sizeof(msg),
        sk);

    printf("SPHINCS+ signature: %zu bytes\n", siglen);

    // Verify
    int valid = crypto_sign_verify(sig, siglen, msg,
        sizeof(msg), pk);

    return valid;  // 0 = valid, -1 = invalid
}
```

Listing 4.2: SLH-DSA using PQClean

4.6.3 Python Example

```
import oqs

def signature_demo():
    """Demonstrate ML-DSA digital signatures."""

    # Create signer
    signer = oqs.Signature("ML-DSA-65")

    # Generate keys
    public_key = signer.generate_keypair()
    print(f"Public key: {len(public_key)} bytes")

    # Sign a message
    message = b"IoT device attestation data: device_id=12345"
    signature = signer.sign(message)
    print(f"Signature: {len(signature)} bytes")

    # Verify (typically on another device)
    verifier = oqs.Signature("ML-DSA-65")
```

```
    is_valid = verifier.verify(message, signature,
        public_key)
    print(f"Signature valid: {is_valid}")

    # Demonstrate invalid signature detection
    tampered_message = b"IoT device attestation data:
        device_id=99999"
    is_valid = verifier.verify(tampered_message, signature,
        public_key)
    print(f"Tampered message valid: {is_valid}")  # False

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

Listing 4.3: Digital signatures in Python

4.7 Implementation Considerations

4.7.1 Deterministic vs. Randomized Signing

Table 4.15: Deterministic vs. randomized signing

Aspect	Deterministic	Randomized
RNG dependency	None	Required
Reproducibility	Same sig for same msg	Different each time
Fault attack resistance	Lower	Higher
Side-channel leakage	Higher risk	Lower risk

NIST recommends **randomized signing** when good randomness is available, but allows deterministic mode for environments with unreliable RNGs.

4.7.2 Batch Verification

For applications verifying many signatures (e.g., blockchain), batch verification can improve throughput:

- ML-DSA: Limited batch optimization possible through NTT sharing
- Falcon: Good batch verification due to shared FFT operations
- SLH-DSA: No significant batch optimization

4.7.3 Side-Channel Considerations

Table 4.16: Side-channel vulnerability by scheme

Attack Type	ML-DSA	Falcon	SLH-DSA
Timing attacks	Medium	High	Low
Power analysis	Medium	High	Low
EM emanation	Medium	High	Low
Fault attacks	Medium	Medium	Low

4.8 Chapter Summary

This chapter covered the three NIST post-quantum signature schemes:

ML-DSA (Dilithium):

- Lattice-based, Fiat-Shamir with Aborts
- Best balance of size and speed
- Recommended for general-purpose use
- pk: 1.3–2.6 KB, sig: 2.4–4.6 KB

Falcon:

- NTRU-lattice with Gaussian sampling
- Smallest signatures (666 B at Level 1)
- Complex implementation, side-channel concerns
- Best for bandwidth-critical applications

SLH-DSA (SPHINCS+):

- Hash-based, conservative security assumptions
- Large signatures (8–50 KB)
- Slow signing, moderate verification
- Best for high-assurance, infrequent signing

Note: For most IoT applications, **ML-DSA-65** is the recommended choice. Use **Falcon** when bandwidth is critical and you have implementation expertise. Use **SLH-DSA** for root certificates and when conservative security is paramount.

Chapter 5

Alternative and Emerging Schemes

While lattice-based cryptography dominates the NIST selections, alternative approaches provide important diversity in the post-quantum ecosystem. This chapter surveys code-based, isogeny-based, and multivariate schemes, with emphasis on those relevant to IoT applications.

5.1 Code-Based Cryptography

Code-based cryptography derives security from the difficulty of decoding random linear codes. This family has the longest history in post-quantum cryptography, dating to McEliece's 1978 proposal.

5.1.1 Theoretical Foundation

Definition 5.1 (Linear Code)**.** A *linear code* $\mathcal{C}$ over a finite field $\mathbb{F}_q$ is a k-dimensional subspace of $\mathbb{F}_q^n$. It can be specified by:

- **Generator matrix** $\mathbf{G} \in \mathbb{F}_q^{k \times n}$: codewords are $\mathbf{c} = \mathbf{mG}$
- **Parity-check matrix** $\mathbf{H} \in \mathbb{F}_q^{(n-k) \times n}$: valid codewords satisfy $\mathbf{Hc}^T = \mathbf{0}$

The code has parameters $[n, k, d]$ where d is the minimum distance.

Definition 5.2 (Syndrome Decoding Problem)**.** Given a parity-check matrix $\mathbf{H} \in \mathbb{F}_2^{r \times n}$ and syndrome $\mathbf{s} \in \mathbb{F}_2^r$, find an error vector $\mathbf{e} \in \mathbb{F}_2^n$ of weight $\leq t$ such that $\mathbf{He}^T = \mathbf{s}$.

The syndrome decoding problem for random codes is NP-complete and believed to be hard for quantum computers. The best quantum algorithms provide only a modest speedup (Grover-like $\sqrt{n}$).

5.1.2 HQC (Hamming Quasi-Cyclic)

HQC is a code-based KEM selected by NIST for standardization in Round 4, providing an alternative to lattice-based ML-KEM.

Design

HQC uses:

- **Quasi-cyclic codes**: Structured codes with efficient representation
- **Tensor product construction**: Combines a quasi-cyclic code with a BCH code
- **QCSD problem**: Quasi-Cyclic Syndrome Decoding as security basis

Definition 5.3 (Quasi-Cyclic Code)**.** A code is quasi-cyclic with index ℓ if every cyclic shift of a codeword by ℓ positions is also a codeword. QC codes can be represented compactly using circulant matrices.

Parameters

Table 5.1: HQC parameter sets

Instance	Security	n	pk	ct	ss
HQC-128	Level 1	17,669	2,249 B	4,433 B	64 B
HQC-192	Level 3	35,851	4,522 B	8,978 B	64 B
HQC-256	Level 5	57,637	7,245 B	14,421 B	64 B

Performance

Table 5.2: HQC performance on Intel Core i7-11850H (AVX2)

Instance	KeyGen (kcycles)	Encaps (kcycles)	Decaps (kcycles)
HQC-128	76	150	353
HQC-192	181	355	732
HQC-256	363	720	1,435

HQC vs. ML-KEM

Table 5.3: HQC vs. ML-KEM comparison (Level 1)

Aspect	HQC-128	ML-KEM-512
Public key	2,249 B	800 B
Ciphertext	4,433 B	768 B
Combined (pk + ct)	6,682 B	1,568 B
Security basis	QCSD	MLWE
Decryption failures	$\sim 2^{-64}$	0
Cryptanalytic maturity	High	High

Warning: HQC has significantly larger public keys and ciphertexts than ML-KEM. It is primarily valuable as cryptographic diversity—a backup in case unexpected weaknesses are found in lattice-based schemes.

5.1.3 BIKE (Bit Flipping Key Encapsulation)

BIKE is another code-based KEM based on QC-MDPC (Quasi-Cyclic Moderate Density Parity-Check) codes.

Design

BIKE features:

- **QC-MDPC codes**: Efficient decoding via bit-flipping algorithms
- **Smaller keys than HQC**: More compact representation
- **IND-CPA construction**: Uses implicit rejection for CCA security

Parameters

Table 5.4: BIKE parameter sets

Instance	Security	r	w	pk	ct	ss
BIKE-L1	Level 1	12,323	142	1,541 B	1,573 B	32 B
BIKE-L3	Level 3	24,659	206	3,083 B	3,115 B	32 B
BIKE-L5	Level 5	40,973	274	5,122 B	5,154 B	32 B

Decoding Failure Rate

A critical consideration for BIKE is the non-zero decoding failure rate (DFR):

Table 5.5: BIKE decoding failure rates

Instance	DFR
BIKE-L1	$< 2^{-128}$
BIKE-L3	$< 2^{-192}$
BIKE-L5	$< 2^{-256}$

Note: While BIKE's DFR is extremely low in theory, implementation bugs or hardware faults could increase it. The decoding algorithm must be carefully implemented to achieve the theoretical DFR.

5.1.4 Classic McEliece

Classic McEliece is the most conservative code-based scheme, using binary Goppa codes with nearly 50 years of cryptanalysis.

Design Philosophy

- **Minimal assumptions**: Uses well-studied Goppa codes
- **Conservative security**: Large safety margins
- **Very large keys**: Main drawback for practical deployment
- **Fast operations**: Once keys are loaded

Parameters

Table 5.6: Classic McEliece parameter sets

Instance	Security	pk	sk	ct	ss
mceliece348864	Level 1	261,120 B	6,492 B	128 B	32 B
mceliece460896	Level 3	524,160 B	13,608 B	188 B	32 B
mceliece6688128	Level 5	1,044,992 B	13,932 B	240 B	32 B
mceliece6960119	Level 5	1,047,319 B	13,948 B	226 B	32 B
mceliece8192128	Level 5	1,357,824 B	14,120 B	240 B	32 B

Warning: Classic McEliece public keys are 256 KB to 1.3 MB—far too large for most IoT devices. However, ciphertexts are very small (128–240 B), making it suitable for scenarios where the public key can be pre-installed.

IoT Applicability

Classic McEliece may be suitable for specific IoT scenarios:

- **Gateway devices**: With sufficient storage for public keys

- **Pre-provisioned keys**: Factory-installed public keys
- **Asymmetric bandwidth**: Small uplink, large storage
- **Long-term security**: Where conservative choices are mandated

5.1.5 Code-Based Schemes Summary

Table 5.7: Code-based schemes comparison (Level 1)

Scheme	pk	ct	Security Basis	DFR	IoT
HQC-128	2,249 B	4,433 B	QCSD	2^{-64}	Limited
BIKE-L1	1,541 B	1,573 B	QC-MDPC	2^{-128}	Limited
McEliece348864	261 KB	128 B	Goppa	0	No*
ML-KEM-512	800 B	768 B	MLWE	0	Yes

*Except for pre-provisioned key scenarios

5.2 Isogeny-Based Cryptography

Isogeny-based cryptography uses the mathematical structure of elliptic curve isogenies. While promising for compact keys, the field has experienced significant setbacks.

5.2.1 Mathematical Background

Definition 5.4 (Isogeny)**.** An *isogeny* $\phi : E_1 \to E_2$ between elliptic curves is a non-constant morphism that preserves the group structure: $\phi(P + Q) = \phi(P) + \phi(Q)$.

Definition 5.5 (Supersingular Isogeny Problem)**.** Given two supersingular elliptic curves E_1 and E_2 over $\mathbb{F}_{p^2}$, find an isogeny $\phi : E_1 \to E_2$.

[Placeholder: Illustration of isogeny graph: nodes are elliptic curves, edges are isogenies of fixed degree]

5.2.2 SIDH/SIKE: A Cautionary Tale

SIKE (Supersingular Isogeny Key Encapsulation) was a NIST Round 4 candidate with remarkably compact parameters:

Table 5.8: SIKE parameters (before break)

Instance	pk	ct	ss
SIKEp434	330 B	346 B	16 B
SIKEp503	378 B	402 B	24 B
SIKEp751	564 B	596 B	32 B

Warning: SIKE was completely broken in 2022. Castryck and Decru demonstrated a polynomial-time attack exploiting the auxiliary torsion point information provided in SIDH. The attack recovers the secret key in minutes on a laptop.

Lessons Learned

The SIKE break provides important lessons:

1. **Cryptanalysis continues**: Even schemes with years of analysis can fall
2. **Diversity matters**: Multiple algorithm families provide insurance
3. **Conservative choices**: Hash-based and code-based schemes have longer track records
4. **Auxiliary data**: Extra information for efficiency can enable attacks

5.2.3 Post-SIKE Isogeny Research

Research continues on isogeny variants not affected by the SIDH attack:

- **CSIDH**: Commutative SIDH using supersingular curves over $\mathbb{F}_p$
- **SQIsign**: Signature scheme based on quaternion isogenies
- **OSIDH**: Oriented SIDH with different auxiliary information

However, these schemes are currently slower and less mature than lattice alternatives.

5.3 Multivariate Cryptography

Multivariate cryptography bases security on the difficulty of solving systems of multivariate polynomial equations.

5.3.1 Theoretical Foundation

Definition 5.6 (MQ Problem)**.** Given a system of m quadratic polynomials $p_1, \ldots, p_m$ in n variables over $\mathbb{F}_q$:

$$p_i(x_1, \ldots, x_n) = \sum_{j \leq k} a_{i,j,k} x_j x_k + \sum_j b_{i,j} x_j + c_i$$

find $(x_1, \ldots, x_n) \in \mathbb{F}_q^n$ such that $p_i(x_1, \ldots, x_n) = 0$ for all i.

The MQ problem is NP-hard, and no efficient quantum algorithm is known.

5.3.2 Signature Schemes

Multivariate schemes excel at signatures (small signatures) but struggle with encryption (large public keys).

Rainbow (Broken)

Rainbow was a NIST Round 3 finalist with very small signatures:

Table 5.9: Rainbow parameters (before break)

Instance	Security	pk	sk	sig
Rainbow-I	Level 1	161 KB	103 KB	66 B
Rainbow-III	Level 3	882 KB	626 KB	164 B
Rainbow-V	Level 5	1.9 MB	1.4 MB	212 B

Warning: Rainbow was broken in 2022. Ward Beullens demonstrated an attack recovering the secret key in about a weekend of computation. This eliminated Rainbow from NIST consideration.

Surviving Multivariate Schemes

Some multivariate schemes remain unbroken:

- **GeMSS**: Based on HFEv- with large public keys
- **MAYO**: Recent design with smaller parameters
- **UOV**: Unbalanced Oil and Vinegar (Rainbow's predecessor)

However, public key sizes remain prohibitive for IoT (hundreds of KB to MB).

5.3.3 Multivariate for IoT

Table 5.10: Multivariate schemes IoT assessment

Aspect	Rating	Notes
Public key size	Poor	100s KB to MBs
Signature size	Excellent	10s to 100s bytes
Signing speed	Good	Fast operations
Verification speed	Good	Fast operations
Security confidence	Medium	Recent breaks

Conclusion: Multivariate schemes are not recommended for general IoT use due to large public keys and recent cryptanalytic advances.

5.4 Stateful Hash-Based Signatures

Unlike SLH-DSA (stateless), stateful hash-based signatures like XMSS and LMS offer smaller signatures but require careful state management.

5.4.1 LMS (Leighton-Micali Signatures)

LMS is standardized in RFC 8554 and NIST SP 800-208.

Design

- **Merkle tree**: Binary tree of one-time signature (OTS) public keys
- **Winternitz OTS**: Efficient one-time signatures at leaves
- **State management**: Must track which OTS keys are used

Parameters

Table 5.11: LMS parameter examples

Tree Height	Signatures	pk	sig	Security
$h = 5$	$2^5 = 32$	56 B	1,616 B	128-bit
$h = 10$	$2^{10} = 1,024$	56 B	1,776 B	128-bit
$h = 15$	$2^{15} = 32,768$	56 B	1,936 B	128-bit
$h = 20$	$2^{20} \approx 1$M	56 B	2,096 B	128-bit
$h = 25$	$2^{25} \approx 33$M	56 B	2,256 B	128-bit

HSS (Hierarchical Signature System)

HSS extends LMS with multiple tree levels:

- Each tree authenticates the next level's public key
- Total signatures: $2^{h_1+h_2+\cdots+h_L}$
- Larger signatures but more signing capacity

5.4.2 XMSS (eXtended Merkle Signature Scheme)

XMSS is standardized in RFC 8391 and offers similar functionality to LMS with different design choices.

Table 5.12: XMSS vs. LMS comparison

Aspect	XMSS	LMS
Standardization	RFC 8391, SP 800-208	RFC 8554, SP 800-208
OTS scheme	WOTS+	LM-OTS
Signature size	Slightly larger	Slightly smaller
Flexibility	More options	Simpler
Adoption	Research focus	Industry focus

5.4.3 State Management Challenges

Warning: Critical: Reusing a one-time signature key in stateful HBS completely breaks security. State must be reliably persisted after every signature operation.

State management requirements:

1. **Atomic updates**: State must be updated before signature release
2. **Persistent storage**: Non-volatile memory required
3. **Crash recovery**: Handle interrupted operations
4. **No rollback**: State must never decrease
5. **Backup coordination**: State replication is dangerous

IoT State Management Approaches

Table 5.13: State management strategies for IoT

Strategy	Reliability	Overhead
EEPROM/Flash counter	Medium	Low
Wear-leveled NVM	High	Medium
Monotonic hardware counter	High	Low
Remote state server	High	High latency
Reserved key ranges	Medium	Wastes keys

5.4.4 Stateful vs. Stateless Comparison

Table 5.14: Stateful (LMS) vs. Stateless (SLH-DSA) signatures

Aspect	LMS/XMSS	SLH-DSA
Signature size	1.6–2.3 KB	7.8–50 KB
Public key	56–64 B	32–64 B
Signing speed	Fast	Slow
State required	Yes (critical)	No
Key lifetime	Limited signatures	Unlimited
Implementation risk	State corruption	None
IoT suitability	Limited	Better

Note: For IoT devices, **SLH-DSA is generally preferred** over stateful schemes despite larger signatures. The risk of state corruption leading to complete security failure is often unacceptable in deployed IoT systems.

5.5 Hybrid Schemes

Hybrid cryptography combines classical and post-quantum algorithms to provide security against both current and future threats.

5.5.1 Motivation

Reasons for hybrid deployment:

1. **Hedging**: Protect against weaknesses in either algorithm
2. **Compliance**: Meet current standards while preparing for PQ
3. **Gradual migration**: Maintain compatibility during transition
4. **Defense in depth**: Multiple security layers

5.5.2 Hybrid KEM Construction

Algorithm 10 Generic Hybrid KEM

// Key Generation
$(pk_1, sk_1) \leftarrow$ Classical.KeyGen() ▷ e.g., X25519
$(pk_2, sk_2) \leftarrow$ PQ.KeyGen() ▷ e.g., ML-KEM
$pk \leftarrow pk_1 \| pk_2$; $sk \leftarrow sk_1 \| sk_2$

// Encapsulation
$(ss_1, ct_1) \leftarrow$ Classical.Encaps(pk_1)
$(ss_2, ct_2) \leftarrow$ PQ.Encaps(pk_2)
$ct \leftarrow ct_1 \| ct_2$
$K \leftarrow \text{KDF}(ss_1 \| ss_2 \| ct)$ ▷ Bind to ciphertext

// Decapsulation
$ss_1 \leftarrow$ Classical.Decaps(sk_1, ct_1)
$ss_2 \leftarrow$ PQ.Decaps(sk_2, ct_2)
$K \leftarrow \text{KDF}(ss_1 \| ss_2 \| ct)$

5.5.3 Deployed Hybrid Schemes

Table 5.15: Hybrid schemes in deployment

Name	Components	pk + ct	User
X25519Kyber768	X25519 + ML-KEM-768	1,248 B	Cloudflare, Google
P256Kyber768	ECDH P-256 + ML-KEM-768	1,280 B	Various
X25519Kyber512	X25519 + ML-KEM-512	896 B	Signal (PQXDH)

5.5.4 Hybrid Signatures

Hybrid signatures are more complex due to binding requirements:

```
// Hybrid signature: both must verify for acceptance
typedef struct {
    uint8_t classical_sig[ECDSA_SIG_SIZE];
    uint8_t pq_sig[MLDSA_SIG_SIZE];
} hybrid_signature_t;

int hybrid_sign(hybrid_signature_t *sig,
                const uint8_t *msg, size_t msg_len,
                const classical_sk_t *classical_sk,
                const pq_sk_t *pq_sk) {
    // Sign with both algorithms
```

```
12      ecdsa_sign(sig->classical_sig, msg, msg_len,
            classical_sk);
13      ml_dsa_sign(sig->pq_sig, msg, msg_len, pq_sk);
14      return 0;
15  }
16
17  int hybrid_verify(const hybrid_signature_t *sig,
18                    const uint8_t *msg, size_t msg_len,
19                    const classical_pk_t *classical_pk,
20                    const pq_pk_t *pq_pk) {
21      // Both must verify
22      int r1 = ecdsa_verify(sig->classical_sig, msg, msg_len,
            classical_pk);
23      int r2 = ml_dsa_verify(sig->pq_sig, msg, msg_len, pq_pk);
24      return (r1 == 0 && r2 == 0) ? 0 : -1;
25  }
```

Listing 5.1: Hybrid signature construction

5.6 Comparative Analysis

5.6.1 Algorithm Family Summary

Table 5.16: PQC algorithm families comparison

Family	Key Size	Speed	Maturity	IoT	Status
Lattice	Medium	Fast	High	Good	NIST Std
Hash-based	Small/Large	Slow/Med	Very High	Limited	NIST Std
Code-based	Large	Medium	High	Limited	Round 4
Isogeny	Small	Slow	Low	No	Broken*
Multivariate	Very Large	Fast	Medium	No	Broken*

*Major schemes broken; research continues on variants

5.6.2 IoT Suitability Matrix

Table 5.17: Scheme suitability by IoT device class

Scheme	**Class 1** (<10KB)	**Class 2** (∼50KB)	**Class 3** (>256KB)	**Gateway** (Linux)
ML-KEM-512				
ML-KEM-768				
ML-DSA-44				
Falcon-512				
SLH-DSA-128f				
HQC-128				
BIKE-L1				
McEliece				

Suitable Limited/Possible Not recommended

5.7 Chapter Summary

This chapter surveyed alternative and emerging post-quantum schemes:

Code-Based (HQC, BIKE, McEliece):

- Long cryptanalytic history, conservative security
- Larger keys/ciphertexts than lattice schemes
- HQC selected for NIST Round 4 standardization
- Limited IoT applicability due to sizes

Isogeny-Based (SIDH/SIKE):

- Compact parameters were attractive
- **Completely broken in 2022**
- Research continues on variants
- Not recommended for deployment

Multivariate (Rainbow, etc.):

- Small signatures but huge public keys
- Rainbow broken in 2022
- Not suitable for IoT

Stateful Hash-Based (LMS, XMSS):

- Smaller signatures than SLH-DSA
- Critical state management requirements
- Risk of catastrophic failure from state corruption
- Generally not recommended for IoT

Hybrid Schemes:

- Combine classical and PQ algorithms
- Recommended during transition period
- Already deployed by major providers

Note: For IoT applications, **lattice-based schemes (ML-KEM, ML-DSA)** remain the primary recommendation. Code-based schemes provide valuable diversity but are secondary choices due to larger sizes. Avoid isogeny and multivariate schemes.

Chapter 6

Implementation for IoT Devices

This chapter provides practical guidance for implementing post-quantum cryptography on resource-constrained IoT devices. We cover hardware platforms, optimization techniques, memory management, and real-world deployment considerations.

6.1 IoT Hardware Landscape

6.1.1 Processor Architectures

IoT devices span a wide range of computational capabilities:

Table 6.1: Common IoT processor families

Family	Bits	Speed	RAM	Use Case
ATmega (AVR)	8	8–20 MHz	2–8 KB	Simple sensors
MSP430	16	8–25 MHz	2–16 KB	Low-power sensors
ARM Cortex-M0/M0+	32	24–48 MHz	8–32 KB	Wearables
ARM Cortex-M3	32	48–120 MHz	32–128 KB	Industrial IoT
ARM Cortex-M4/M4F	32	80–180 MHz	128–512 KB	Advanced IoT
ARM Cortex-M7	32	200–600 MHz	512 KB–1 MB	IoT gateways
ARM Cortex-M33	32	100–200 MHz	256–512 KB	Secure IoT
RISC-V (various)	32	50–400 MHz	64 KB–1 MB	Emerging IoT
ESP32 (Xtensa)	32	160–240 MHz	520 KB	WiFi/BLE IoT

6.1.2 ARM Cortex-M Series Deep Dive

The ARM Cortex-M series dominates IoT applications. Understanding their features is essential for PQC optimization.

Table 6.2: ARM Cortex-M feature comparison for cryptography

Feature	M0+	M3	M4	M7	M33
Pipeline stages	2	3	3	6	3
Hardware multiply	1 or 32 cyc	1 cyc	1 cyc	1 cyc	1 cyc
Hardware divide	No	Yes	Yes	Yes	Yes
DSP instructions	No	No	Yes	Yes	Yes
FPU	No	No	Optional	Yes	Optional
SIMD	No	No	Yes	Yes	Yes
Cache	No	No	No	Yes	Optional
TrustZone	No	No	No	No	Yes
Crypto accelerator	Rare	Rare	Common	Common	Common

DSP Instructions for PQC

The Cortex-M4/M7 DSP extensions provide significant speedups for PQC:

```
// SMUAD: Dual 16-bit multiply with 32-bit accumulate
// Useful for NTT butterfly operations
static inline int32_t dual_multiply_add(int32_t a, int32_t
    b) {
    int32_t result;
    __asm__ volatile (
        "smuad %0, %1, %2"
        : "=r" (result)
        : "r" (a), "r" (b)
    );
    return result;
}

// SMLADX: Dual multiply-accumulate with exchange
// a[15:0]*b[31:16] + a[31:16]*b[15:0] + acc
static inline int32_t butterfly_mac(int32_t a, int32_t b,
    int32_t acc) {
    int32_t result;
    __asm__ volatile (
        "smladx %0, %1, %2, %3"
        : "=r" (result)
        : "r" (a), "r" (b), "r" (acc)
    );
    return result;
}
```

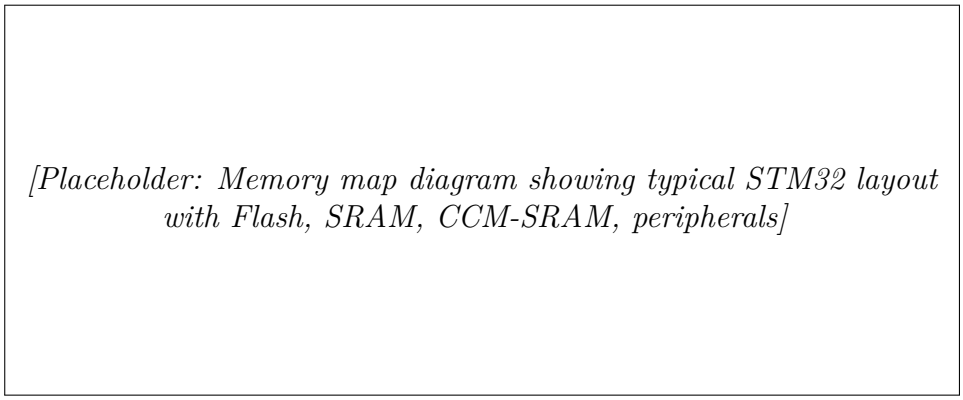
Listing 6.1: Using DSP instructions for coefficient operations

6.1.3 Memory Hierarchy

IoT devices have constrained memory organized in distinct regions:

Table 6.3: Typical memory organization in IoT MCUs

Memory Type	Size Range	Speed	Purpose
SRAM (main)	32–512 KB	Fast	Stack, heap, variables
SRAM (TCM)	0–64 KB	Fastest	Critical code/data
Flash (internal)	256 KB–2 MB	Medium	Program code
Flash (external)	1–16 MB	Slow	Large data, OTA
EEPROM	0–64 KB	Slow	Persistent config



6.2 Memory Optimization

6.2.1 Stack Usage Analysis

PQC algorithms require careful stack management. Here is the detailed stack analysis for the pqm4 implementations:

Table 6.4: Detailed stack usage on ARM Cortex-M4 (bytes)

Algorithm	Variant	KeyGen	Enc/Sign	Dec/Verify
Key Encapsulation				
ML-KEM-512	m4fstack	2,300	2,348	2,332
ML-KEM-768	m4fstack	2,556	2,604	2,588
ML-KEM-1024	m4fstack	2,812	2,860	2,844
ML-KEM-512	m4fspeed	2,632	2,720	3,080
ML-KEM-768	m4fspeed	3,144	3,232	3,592
ML-KEM-1024	m4fspeed	3,656	3,744	4,104
Digital Signatures				
ML-DSA-44	m4f	10,328	24,512	10,680
ML-DSA-65	m4f	14,472	34,752	15,336
ML-DSA-87	m4f	18,104	43,968	19,992
Falcon-512	opt	2,096	39,000	2,024
Falcon-1024	opt	2,472	79,000	3,048

Warning: ML-DSA signing requires up to 44 KB of stack space. Ensure your IoT device has adequate SRAM and that the stack is sized appropriately. Stack overflow causes hard-to-diagnose failures.

6.2.2 Stack Reduction Techniques

On-the-Fly Matrix Generation

Instead of storing the full public matrix $\mathbf{A}$, regenerate columns as needed:

```
// Memory-efficient: generate matrix column by column
void matrix_vector_mul_otf(poly *result, const uint8_t
    seed[32],
                           const polyvec *vec, int k) {
    poly a_col;

    for (int i = 0; i < k; i++) {
        result[i] = 0;
        for (int j = 0; j < k; j++) {
            // Generate A[i][j] on-the-fly
            gen_matrix_entry(&a_col, seed, i, j);
            // Multiply and accumulate
            poly_basemul_acc(&result[i], &a_col,
                &vec->vec[j]);
        }
    }
}
```

```
16
17 // Contrast: Full matrix storage (not recommended for
       constrained devices)
18 // poly A[K][K];  // Would require K*K*512 bytes = 18KB for
       K=3
```

Listing 6.2: On-the-fly matrix generation

Streaming Hash Computation

Process data incrementally instead of buffering:

```
1  // Memory-efficient message hashing
2  void hash_message_streaming(uint8_t *output, size_t outlen,
3                              const uint8_t *prefix, size_t
                                   prefix_len,
4                              const uint8_t *msg, size_t
                                   msg_len) {
5      shake256_ctx ctx;
6      shake256_init(&ctx);
7
8      // Process prefix (e.g., domain separator)
9      shake256_absorb(&ctx, prefix, prefix_len);
10
11     // Process message in chunks (for very large messages)
12     const size_t CHUNK_SIZE = 1024;
13     while (msg_len > CHUNK_SIZE) {
14         shake256_absorb(&ctx, msg, CHUNK_SIZE);
15         msg += CHUNK_SIZE;
16         msg_len -= CHUNK_SIZE;
17     }
18     shake256_absorb(&ctx, msg, msg_len);
19
20     // Finalize and squeeze output
21     shake256_finalize(&ctx);
22     shake256_squeeze(&ctx, output, outlen);
23 }
```

Listing 6.3: Streaming SHAKE for large inputs

6.2.3 Code Size Optimization

Table 6.5: Code size (.text section) on ARM Cortex-M4 (bytes)

Algorithm	Variant	Code Size
ML-KEM-512	clean	6,036
ML-KEM-512	m4fspeed	12,044
ML-KEM-512	m4fstack	10,644
ML-KEM-768	m4fspeed	12,044
ML-KEM-1024	m4fspeed	12,044
ML-DSA-44	clean	14,824
ML-DSA-44	m4f	21,316
ML-DSA-65	m4f	21,892
ML-DSA-87	m4f	22,372
Falcon-512	clean	82,340
Falcon-512	opt	86,180

Code Size Reduction Strategies

1. **Algorithm selection**: Choose reference implementations for size over speed
2. **Compiler flags**: Use `-Os` (optimize for size) instead of `-O3`
3. **Link-time optimization**: `-flto` eliminates unused code
4. **Shared functions**: Common operations (NTT, hashing) shared across algorithms
5. **Parameter specialization**: Compile only for needed parameter sets

```
# Makefile flags for minimal code size
CFLAGS_SIZE = -Os \
              -ffunction-sections \
              -fdata-sections \
              -fno-exceptions \
              -fno-rtti \
              -fno-unwind-tables

LDFLAGS_SIZE = -Wl,--gc-sections \
               -flto \
               -specs=nano.specs
```

Listing 6.4: Compiler flags for size optimization

6.3 Performance Optimization

6.3.1 NTT Optimization

The Number Theoretic Transform is the performance-critical operation in lattice-based PQC.

Montgomery Arithmetic

Montgomery reduction avoids expensive division:

```
// Constants for q = 3329
#define KYBER_Q 3329
#define MONT_R 2285      // R mod q where R = 2^16
#define QINV (-3327)     // q^{-1} mod 2^16

// Montgomery reduction: compute a * R^{-1} mod q
// Input: a in [-q*2^15, q*2^15]
// Output: result in [-q, q]
int16_t montgomery_reduce(int32_t a) {
    int16_t t;
    t = (int16_t)a * QINV;
    t = (a - (int32_t)t * KYBER_Q) >> 16;
    return t;
}

// Barrett reduction for final reduction to [0, q)
int16_t barrett_reduce(int16_t a) {
    int16_t t;
    const int16_t v = ((1 << 26) + KYBER_Q / 2) / KYBER_Q;
    t = ((int32_t)v * a + (1 << 25)) >> 26;
    t *= KYBER_Q;
    return a - t;
}
```

Listing 6.5: Montgomery multiplication for ML-KEM

Optimized NTT Butterfly

```
// NTT butterfly with Montgomery multiplication
// Computes: a = a + w*b, b = a - w*b (mod q)
static inline void butterfly(int16_t *a, int16_t *b, int16_t
    w) {
```

```
    int16_t t = montgomery_reduce((int32_t)w * *b);
    *b = *a - t;
    *a = *a + t;
}

// Full NTT using Cooley-Tukey, in-place
void ntt(int16_t r[256]) {
    unsigned int len, start, j, k;
    int16_t t, zeta;

    k = 1;
    for (len = 128; len >= 2; len >>= 1) {
        for (start = 0; start < 256; start = j + len) {
            zeta = zetas[k++];
            for (j = start; j < start + len; j++) {
                t = montgomery_reduce((int32_t)zeta * r[j +
                    len]);
                r[j + len] = r[j] - t;
                r[j] = r[j] + t;
            }
        }
    }
}
```

Listing 6.6: Optimized NTT butterfly for Cortex-M4

Assembly Optimization

For maximum performance, critical loops are written in assembly:

```
// ARM Cortex-M4 assembly for butterfly operation
// Uses SMULBB, SMLABB for efficient 16x16->32 multiplication
.syntax unified
.thumb

.global butterfly_asm
.type butterfly_asm, %function
butterfly_asm:
    // r0 = pointer to a, r1 = pointer to b, r2 = zeta
    ldrsh r3, [r0]        // Load a
    ldrsh r4, [r1]        // Load b

    // t = zeta * b (Montgomery domain)
```

```
    smulbb r5, r2, r4      // r5 = zeta * b

    // Montgomery reduction
    mul r6, r5, r7         // r6 = t * QINV (low 16 bits)
    sxth r6, r6
    mls r5, r6, r8, r5     // r5 = t - r6 * Q
    asr r5, r5, #16        // r5 = result >> 16

    // Butterfly
    add r6, r3, r5         // a' = a + t
    sub r7, r3, r5         // b' = a - t

    strh r6, [r0]          // Store a'
    strh r7, [r1]          // Store b'
    bx lr
```

Listing 6.7: ARM assembly NTT butterfly (simplified)

6.3.2 Hash Function Optimization

SHA-3/SHAKE often dominates runtime in PQC implementations.

Table 6.6: SHA-3 Keccak-f[1600] performance on various platforms

Platform	Cycles/permutation	Implementation
Cortex-M0+	∼50,000	Reference C
Cortex-M3	∼12,000	Optimized C
Cortex-M4	∼6,000	Optimized C
Cortex-M4	∼4,500	Assembly
Cortex-M7	∼3,000	Assembly
x86-64 (AVX2)	∼500	AVX2 intrinsics

Platform-Specific Keccak

```
// Compile-time selection of Keccak implementation
#if defined(__ARM_ARCH_7EM__)  // Cortex-M4/M7
    #include "keccakf1600_armv7m.h"
    #define KECCAK_IMPL "ARMv7-M assembly"
#elif defined(__ARM_ARCH_6M__)  // Cortex-M0/M0+
    #include "keccakf1600_armv6m.h"
    #define KECCAK_IMPL "ARMv6-M optimized"
#elif defined(__AVX2__)
    #include "keccakf1600_avx2.h"
```

```
    #define KECCAK_IMPL "AVX2"
#else
    #include "keccakf1600_ref.h"
    #define KECCAK_IMPL "Reference C"
#endif
```

Listing 6.8: Selecting optimal Keccak implementation

6.3.3 Parallelism and Pipelining

Instruction-Level Parallelism

Modern MCUs can execute multiple operations per cycle with proper scheduling:

```
// Unrolled polynomial addition for better ILP
void poly_add_unrolled(int16_t r[256], const int16_t a[256],
                       const int16_t b[256]) {
    for (int i = 0; i < 256; i += 8) {
        r[i+0] = a[i+0] + b[i+0];
        r[i+1] = a[i+1] + b[i+1];
        r[i+2] = a[i+2] + b[i+2];
        r[i+3] = a[i+3] + b[i+3];
        r[i+4] = a[i+4] + b[i+4];
        r[i+5] = a[i+5] + b[i+5];
        r[i+6] = a[i+6] + b[i+6];
        r[i+7] = a[i+7] + b[i+7];
    }
}
```

Listing 6.9: Loop unrolling for ILP

SIMD on Cortex-M4

The Cortex-M4 supports limited SIMD via packed operations:

```
#include <arm_acle.h>

// Process two 16-bit coefficients at once
void poly_add_simd(int16_t r[256], const int16_t a[256],
                   const int16_t b[256]) {
    uint32_t *r32 = (uint32_t *)r;
    const uint32_t *a32 = (const uint32_t *)a;
    const uint32_t *b32 = (const uint32_t *)b;

```

```
    for (int i = 0; i < 128; i++) {
        // QADD16: Parallel saturating add of two 16-bit values
        r32[i] = __qadd16(a32[i], b32[i]);
    }
}
```

Listing 6.10: SIMD polynomial operations

6.4 Side-Channel Countermeasures

6.4.1 Timing Attacks

All cryptographic operations must execute in constant time regardless of secret values.

Constant-Time Principles

1. **No secret-dependent branches**: Avoid `if (secret) ...`
2. **No secret-dependent memory access**: Array indices must not depend on secrets
3. **No variable-time instructions**: Avoid division, floating-point on some platforms
4. **No early termination**: Complete all iterations regardless of values

```
// Constant-time conditional select: return a if sel, else b
static inline uint32_t ct_select(uint32_t a, uint32_t b, uint32_t sel) {
    // sel must be 0 or 1
    uint32_t mask = -(uint32_t)sel;  // 0x00000000 or 0xFFFFFFFF
    return (a & mask) | (b & ~mask);
}

// Constant-time comparison: returns 0 if equal, non-zero otherwise
static inline uint32_t ct_compare(const uint8_t *a, const uint8_t *b,
                                  size_t len) {
    uint32_t diff = 0;
    for (size_t i = 0; i < len; i++) {
        diff |= a[i] ^ b[i];
```

```
    }
    return diff;
}

// Constant-time conditional copy
static inline void ct_cmov(uint8_t *dst, const uint8_t *src,
                           size_t len, uint8_t condition) {
    uint8_t mask = -(uint8_t)(condition != 0);
    for (size_t i = 0; i < len; i++) {
        dst[i] ^= mask & (dst[i] ^ src[i]);
    }
}
```

Listing 6.11: Constant-time primitives

Verifying Constant-Time Execution

```
# Compile with debug symbols
gcc -g -O2 -o pqc_test pqc_test.c -lpqc

# Run under ctgrind (Valgrind plugin)
valgrind --tool=ctgrind ./pqc_test

# Alternative: Use timecop
gcc -fsanitize=memory -o pqc_test pqc_test.c -lpqc
./pqc_test
```

Listing 6.12: Using ctgrind for verification

6.4.2 Power Analysis

Power consumption can reveal secret information through:

- **Simple Power Analysis (SPA)**: Direct observation of power traces
- **Differential Power Analysis (DPA)**: Statistical analysis across many traces
- **Correlation Power Analysis (CPA)**: Correlation with hypothetical power models

Countermeasures

Table 6.7: Power analysis countermeasures

Countermeasure	Overhead	Effectiveness
Masking (Boolean)	2–3×	High
Masking (Arithmetic)	3–5×	High
Shuffling	1.5–2×	Medium
Hiding (random delays)	1.2–1.5×	Low
Dual-rail logic	Hardware	Very High

```
// First-order Boolean masking for sensitive data
typedef struct {
    uint8_t share0[32];  // x0
    uint8_t share1[32];  // x1, where x = x0 XOR x1
} masked_key_t;

void mask_key(masked_key_t *masked, const uint8_t key[32]) {
    // Generate random mask
    randombytes(masked->share1, 32);
    // Compute first share: x0 = x XOR x1
    for (int i = 0; i < 32; i++) {
        masked->share0[i] = key[i] ^ masked->share1[i];
    }
}

void unmask_key(uint8_t key[32], const masked_key_t *masked)
   {
    for (int i = 0; i < 32; i++) {
        key[i] = masked->share0[i] ^ masked->share1[i];
    }
}
```

Listing 6.13: First-order Boolean masking

6.4.3 Fault Attacks

Fault attacks induce errors (via voltage glitches, clock manipulation, lasers) to extract secrets.

Countermeasures

1. **Redundant computation**: Compute twice and compare

2. **Integrity checks**: Verify intermediate values

3. **Infection**: Randomize output if fault detected

4. **Hardware countermeasures**: Voltage/clock monitors

```c
// Double computation with comparison
int secure_verify(const uint8_t *pk, const uint8_t *msg,
    size_t msg_len,
                  const uint8_t *sig) {
    int result1, result2;

    // First verification
    result1 = crypto_sign_verify(sig, msg, msg_len, pk);

    // Second verification (ideally with different
       implementation)
    result2 = crypto_sign_verify(sig, msg, msg_len, pk);

    // Both must agree
    if (result1 != result2) {
        // Fault detected! Take appropriate action
        fault_detected_handler();
        return -1;  // Reject
    }

    return result1;
}
```

Listing 6.14: Redundant computation for fault resistance

6.5 Hardware Acceleration

6.5.1 Cryptographic Accelerators

Many modern MCUs include hardware acceleration for common operations:

Table 6.8: Hardware crypto support in popular IoT MCUs

MCU Family	AES	SHA-2	SHA-3	RNG	PKA
STM32L4	Yes	Yes	No	Yes	No
STM32U5	Yes	Yes	Yes	Yes	No
STM32H7	Yes	Yes	No	Yes	Yes
nRF52840	Yes	No	No	Yes	No
ESP32-S3	Yes	Yes	No	Yes	Yes
LPC55S69	Yes	Yes	No	Yes	Yes
MAX32666	Yes	Yes	No	Yes	Yes

Using Hardware SHA-3 (STM32U5)

```
#include "stm32u5xx_hal.h"

HASH_HandleTypeDef hhash;

void sha3_256_hw(uint8_t *output, const uint8_t *input,
    size_t len) {
    // Configure for SHA3-256
    hhash.Init.DataType = HASH_DATATYPE_8B;
    hhash.Init.Algorithm = HASH_ALGOSELECTION_SHA3_256;
    HAL_HASH_Init(&hhash);

    // Process data
    HAL_HASHEx_SHA3_256_Start(&hhash, (uint8_t *)input, len,
                              output, HAL_MAX_DELAY);
}

// SHAKE256 XOF using hardware
void shake256_hw(uint8_t *output, size_t outlen,
                 const uint8_t *input, size_t inlen) {
    hhash.Init.Algorithm = HASH_ALGOSELECTION_SHAKE256;
    HAL_HASH_Init(&hhash);

    // Absorb
    HAL_HASHEx_SHAKE256_Start(&hhash, (uint8_t *)input,
        inlen,
                              output, HAL_MAX_DELAY);

    // Squeeze additional output if needed
    while (outlen > 32) {
```

```
28          HAL_HASHEx_SHAKE256_Squeeze(&hhash, output, 32,
                HAL_MAX_DELAY);
29          output += 32;
30          outlen -= 32;
31      }
32      if (outlen > 0) {
33          HAL_HASHEx_SHAKE256_Squeeze(&hhash, output, outlen,
                HAL_MAX_DELAY);
34      }
35  }
```

Listing 6.15: Hardware SHA-3 on STM32U5

6.5.2 FPGA and Custom Hardware

For high-throughput IoT applications (gateways, base stations), FPGAs provide significant acceleration:

Table 6.9: PQC on FPGA: Performance examples

Algorithm	FPGA	Freq	KeyGen	Ops/sec
ML-KEM-768	Artix-7	200 MHz	3.2 µs	312,500
ML-KEM-768	Zynq-7020	150 MHz	4.5 µs	222,222
ML-DSA-65	Artix-7	200 MHz	15 µs	66,666
Falcon-512	Virtex-7	250 MHz	850 µs	1,176

6.6 Random Number Generation

6.6.1 Importance in PQC

High-quality randomness is essential for:

- Key generation (private keys)
- Nonce/randomness in signing (ML-DSA, Falcon)
- Masking values for side-channel protection
- Session keys and initialization vectors

Warning: Poor randomness is one of the most common causes of cryptographic failures. A predictable RNG can completely undermine PQC security, even with correct algorithm implementation.

6.6.2 Hardware RNG

```c
#include "stm32l4xx_hal.h"

RNG_HandleTypeDef hrng;

int randombytes(uint8_t *buf, size_t len) {
    uint32_t random_word;
    size_t i = 0;

    while (i < len) {
        if (HAL_RNG_GenerateRandomNumber(&hrng,
           &random_word) != HAL_OK) {
            return -1;  // RNG error
        }

        // Copy bytes from random word
        size_t bytes_to_copy = (len - i < 4) ? (len - i) : 4;
        memcpy(buf + i, &random_word, bytes_to_copy);
        i += bytes_to_copy;
    }

    return 0;
}

// Initialize RNG with health checks
int rng_init(void) {
    __HAL_RCC_RNG_CLK_ENABLE();

    hrng.Instance = RNG;
    if (HAL_RNG_Init(&hrng) != HAL_OK) {
        return -1;
    }

    // Health test: verify output is not stuck
    uint32_t test1, test2;
    HAL_RNG_GenerateRandomNumber(&hrng, &test1);
    HAL_RNG_GenerateRandomNumber(&hrng, &test2);

    if (test1 == test2) {
        return -1;  // RNG may be faulty
    }
```

```
40
41      return 0;
42  }
```

Listing 6.16: Hardware RNG on STM32

6.6.3 DRBG Seeding

For devices without hardware RNG, use a DRBG seeded from multiple entropy sources:

```
1  // Collect entropy from multiple sources
2  void collect_entropy(uint8_t *seed, size_t seed_len) {
3      sha3_256_ctx ctx;
4      sha3_256_init(&ctx);
5
6      // Source 1: Hardware RNG (if available)
7      #ifdef HAS_HWRNG
8      uint8_t hw_random[32];
9      hw_rng_get(hw_random, 32);
10     sha3_256_update(&ctx, hw_random, 32);
11     #endif
12
13     // Source 2: ADC noise
14     uint16_t adc_samples[64];
15     for (int i = 0; i < 64; i++) {
16         adc_samples[i] = read_adc_noise_source();
17     }
18     sha3_256_update(&ctx, (uint8_t *)adc_samples,
          sizeof(adc_samples));
19
20     // Source 3: Timer jitter
21     uint32_t timer_values[32];
22     for (int i = 0; i < 32; i++) {
23         timer_values[i] = get_high_res_timer();
24         busy_wait_random();
25     }
26     sha3_256_update(&ctx, (uint8_t *)timer_values,
          sizeof(timer_values));
27
28     // Source 4: Unique device ID
29     uint8_t device_id[12];
30     get_device_unique_id(device_id);
31     sha3_256_update(&ctx, device_id, 12);
```

```
32
33      // Finalize
34      sha3_256_final(&ctx, seed);
35  }
```

Listing 6.17: Multi-source entropy collection

6.7 Testing and Validation

6.7.1 Known Answer Tests (KAT)

Validate implementations against official test vectors:

```
1  // Structure for KAT test case
2  typedef struct {
3      uint8_t seed[48];
4      uint8_t pk[CRYPTO_PUBLICKEYBYTES];
5      uint8_t sk[CRYPTO_SECRETKEYBYTES];
6      uint8_t ct[CRYPTO_CIPHERTEXTBYTES];
7      uint8_t ss[CRYPTO_BYTES];
8  } kem_kat_t;
9
10 int run_kat_test(const kem_kat_t *kat) {
11     uint8_t pk[CRYPTO_PUBLICKEYBYTES];
12     uint8_t sk[CRYPTO_SECRETKEYBYTES];
13     uint8_t ct[CRYPTO_CIPHERTEXTBYTES];
14     uint8_t ss1[CRYPTO_BYTES], ss2[CRYPTO_BYTES];
15
16     // Seed the DRBG with known value
17     drbg_seed(kat->seed, 48);
18
19     // Generate keypair
20     crypto_kem_keypair(pk, sk);
21
22     // Verify keys match expected
23     if (memcmp(pk, kat->pk, CRYPTO_PUBLICKEYBYTES) != 0) {
24         printf("KAT FAIL: public key mismatch\n");
25         return -1;
26     }
27
28     // Test encapsulation
29     crypto_kem_enc(ct, ss1, pk);
```

```
    if (memcmp(ct, kat->ct, CRYPTO_CIPHERTEXTBYTES) != 0) {
        printf("KAT FAIL: ciphertext mismatch\n");
        return -1;
    }

    // Test decapsulation
    crypto_kem_dec(ss2, ct, sk);
    if (memcmp(ss1, ss2, CRYPTO_BYTES) != 0) {
        printf("KAT FAIL: shared secret mismatch\n");
        return -1;
    }

    printf("KAT PASS\n");
    return 0;
}
```

Listing 6.18: KAT verification framework

6.7.2 Benchmark Framework

```
#include "cycle_count.h"

typedef struct {
    uint32_t min;
    uint32_t max;
    uint64_t total;
    uint32_t count;
} benchmark_t;

void benchmark_init(benchmark_t *b) {
    b->min = UINT32_MAX;
    b->max = 0;
    b->total = 0;
    b->count = 0;
}

void benchmark_record(benchmark_t *b, uint32_t cycles) {
    if (cycles < b->min) b->min = cycles;
    if (cycles > b->max) b->max = cycles;
    b->total += cycles;
    b->count++;
}
```

```
23
24 void benchmark_report(const char *name, benchmark_t *b) {
25     printf("%s: min=%lu, max=%lu, avg=%lu (n=%lu)\n",
26            name, b->min, b->max,
27            (uint32_t)(b->total / b->count), b->count);
28 }
29
30 // Example usage
31 void benchmark_mlkem(void) {
32     benchmark_t keygen_bench, encaps_bench, decaps_bench;
33     uint32_t start, end;
34
35     benchmark_init(&keygen_bench);
36     benchmark_init(&encaps_bench);
37     benchmark_init(&decaps_bench);
38
39     for (int i = 0; i < 100; i++) {
40         uint8_t pk[MLKEM_PUBLICKEYBYTES];
41         uint8_t sk[MLKEM_SECRETKEYBYTES];
42         uint8_t ct[MLKEM_CIPHERTEXTBYTES];
43         uint8_t ss[32];
44
45         start = get_cycle_count();
46         crypto_kem_keypair(pk, sk);
47         end = get_cycle_count();
48         benchmark_record(&keygen_bench, end - start);
49
50         start = get_cycle_count();
51         crypto_kem_enc(ct, ss, pk);
52         end = get_cycle_count();
53         benchmark_record(&encaps_bench, end - start);
54
55         start = get_cycle_count();
56         crypto_kem_dec(ss, ct, sk);
57         end = get_cycle_count();
58         benchmark_record(&decaps_bench, end - start);
59     }
60
61     benchmark_report("KeyGen", &keygen_bench);
62     benchmark_report("Encaps", &encaps_bench);
63     benchmark_report("Decaps", &decaps_bench);
```

```
64 }
```

Listing 6.19: Benchmarking framework for IoT

6.8 Chapter Summary

This chapter provided comprehensive implementation guidance for PQC on IoT devices:

Hardware Considerations:

- ARM Cortex-M4 is the sweet spot for PQC (DSP, reasonable RAM)
- DSP instructions provide 2× speedup for NTT operations
- Stack usage ranges from 2–3 KB (ML-KEM) to 44 KB (ML-DSA signing)

Optimization Techniques:

- On-the-fly matrix generation saves memory
- Montgomery/Barrett reduction for efficient modular arithmetic
- Assembly NTT provides significant speedups
- Loop unrolling and SIMD improve ILP

Security Considerations:

- All operations must be constant-time
- Masking protects against power analysis
- Redundant computation detects faults
- Hardware RNG is essential; DRBG as backup

Testing:

- KAT tests verify correctness against standards
- Benchmarking framework for performance validation
- Constant-time verification tools (ctgrind)

Note: Start with reference implementations and well-tested libraries (pqm4, liboqs). Optimize only after profiling shows specific bottlenecks. Security must never be sacrificed for performance.

Chapter 7

Libraries and Tools

This chapter provides a comprehensive survey of software libraries, frameworks, and tools available for implementing post-quantum cryptography. We focus on libraries suitable for IoT development, including both general-purpose and embedded-specific options.

7.1 Overview of PQC Libraries

Table 7.1: Major PQC libraries comparison

Library	Language	IoT	Algorithms	License	Mainta
liboqs	C	Limited	All NIST	MIT	Activ
PQClean	C	Yes	All NIST	Public Domain	Activ
pqm4	C/ASM	Yes	Most NIST	Public Domain	Activ
PQCRYPTO-LWEKE	C	Yes	Kyber	MIT	Activ
wolfSSL/wolfCrypt	C	Yes	ML-KEM, ML-DSA	GPLv2/Commercial	Activ
Botan	C++	Limited	ML-KEM, ML-DSA	BSD-2	Activ
libsodium-pq	C	Yes	Kyber (fork)	ISC	Limit
pqcrypto (Rust)	Rust	Limited	Most NIST	Various	Activ
liboqs-go	Go	No	All NIST	MIT	Activ
liboqs-python	Python	No	All NIST	MIT	Activ

7.2 liboqs (Open Quantum Safe)

liboqs is the reference library from the Open Quantum Safe project, providing a unified API for all NIST PQC algorithms.

7.2.1 Features

- **Comprehensive**: Implements all NIST finalists and alternates

- **Unified API**: Consistent interface across all algorithms
- **Regularly updated**: Tracks NIST specification changes
- **Language bindings**: C, Python, Go, Java, .NET, Rust
- **Integration**: Works with OpenSSL, BoringSSL, OpenSSH

7.2.2 Installation

```
# Clone the repository
git clone https://github.com/open-quantum-safe/liboqs.git
cd liboqs

# Build with CMake
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX=/usr/local \
      -DBUILD_SHARED_LIBS=ON \
      -DOQS_BUILD_ONLY_LIB=ON \
      ..
make -j$(nproc)
sudo make install

# For embedded (minimal build)
cmake -DOQS_MINIMAL_BUILD="KEM_kyber_768;SIG_dilithium3" \
      -DOQS_USE_OPENSSL=OFF \
      -DOQS_USE_AES_OPENSSL=OFF \
      -DBUILD_SHARED_LIBS=OFF \
      ..
```

Listing 7.1: Building liboqs

7.2.3 API Reference

```
#include <oqs/oqs.h>

// List available algorithms
for (size_t i = 0; i < OQS_KEM_alg_count(); i++) {
    const char *alg = OQS_KEM_alg_identifier(i);
    if (OQS_KEM_alg_is_enabled(alg)) {
        printf("Available: %s\n", alg);
    }
```

```
}

// Algorithm names
// "ML-KEM-512", "ML-KEM-768", "ML-KEM-1024"
// "Kyber512", "Kyber768", "Kyber1024" (aliases)

// Create KEM instance
OQS_KEM *kem = OQS_KEM_new(OQS_KEM_alg_ml_kem_768);

// Access properties
printf("Public key length: %zu\n", kem->length_public_key);
printf("Secret key length: %zu\n", kem->length_secret_key);
printf("Ciphertext length: %zu\n", kem->length_ciphertext);
printf("Shared secret length: %zu\n",
   kem->length_shared_secret);

// Key generation
uint8_t *pk = malloc(kem->length_public_key);
uint8_t *sk = malloc(kem->length_secret_key);
OQS_KEM_keypair(kem, pk, sk);

// Encapsulation
uint8_t *ct = malloc(kem->length_ciphertext);
uint8_t *ss_enc = malloc(kem->length_shared_secret);
OQS_KEM_encaps(kem, ct, ss_enc, pk);

// Decapsulation
uint8_t *ss_dec = malloc(kem->length_shared_secret);
OQS_KEM_decaps(kem, ss_dec, ct, sk);

// Cleanup (secure)
OQS_MEM_secure_free(sk, kem->length_secret_key);
OQS_MEM_secure_free(ss_enc, kem->length_shared_secret);
OQS_MEM_secure_free(ss_dec, kem->length_shared_secret);
free(pk);
free(ct);
OQS_KEM_free(kem);
```

Listing 7.2: liboqs KEM API

```
#include <oqs/oqs.h>

```

```c
// Available signature algorithms
// "ML-DSA-44", "ML-DSA-65", "ML-DSA-87"
// "Falcon-512", "Falcon-1024"
// "SPHINCS+-SHA2-128f-simple", "SPHINCS+-SHA2-128s-simple", ...

// Create signature instance
OQS_SIG *sig = OQS_SIG_new(OQS_SIG_alg_ml_dsa_65);

// Key generation
uint8_t *pk = malloc(sig->length_public_key);
uint8_t *sk = malloc(sig->length_secret_key);
OQS_SIG_keypair(sig, pk, sk);

// Signing
const uint8_t msg[] = "Message to sign";
size_t msg_len = sizeof(msg) - 1;
uint8_t *signature = malloc(sig->length_signature);
size_t sig_len;
OQS_SIG_sign(sig, signature, &sig_len, msg, msg_len, sk);

// Verification
OQS_STATUS result = OQS_SIG_verify(sig, msg, msg_len,
                                   signature, sig_len, pk);
if (result == OQS_SUCCESS) {
    printf("Signature valid\n");
} else {
    printf("Signature invalid\n");
}

// Cleanup
OQS_MEM_secure_free(sk, sig->length_secret_key);
free(pk);
free(signature);
OQS_SIG_free(sig);
```

Listing 7.3: liboqs Signature API

7.2.4 Python Bindings

```python
import oqs

```

```
# List available KEMs
print("Available KEMs:", oqs.get_enabled_kem_mechanisms())

# Key exchange example
with oqs.KeyEncapsulation("ML-KEM-768") as client:
    # Server generates keypair
    with oqs.KeyEncapsulation("ML-KEM-768") as server:
        public_key = server.generate_keypair()

        # Client encapsulates
        ciphertext, shared_secret_client =
            client.encap_secret(public_key)

        # Server decapsulates
        shared_secret_server =
            server.decap_secret(ciphertext)

        assert shared_secret_client == shared_secret_server
        print(f"Shared secret:
            {shared_secret_client.hex()[:32]}...")

# Signature example
with oqs.Signature("ML-DSA-65") as signer:
    public_key = signer.generate_keypair()

    message = b"Important document"
    signature = signer.sign(message)

    # Verify (could be different instance)
    with oqs.Signature("ML-DSA-65") as verifier:
        is_valid = verifier.verify(message, signature,
            public_key)
        print(f"Signature valid: {is_valid}")
```

Listing 7.4: liboqs-python usage

7.3 PQClean

PQClean provides clean, portable, and auditable reference implementations.

7.3.1 Design Philosophy

- **Clean code**: Readable, well-documented implementations
- **No dependencies**: Self-contained, no external libraries
- **Consistent API**: Uniform interface across all schemes
- **Multiple variants**: Clean, AVX2, aarch64 optimized versions
- **Public domain**: CC0 license for maximum flexibility

7.3.2 Structure

```
PQClean/
          crypto_kem/
                kyber512/
                      clean/           # Reference
    implementation
                            api.h        # Public API
                            kem.c        # KEM functions
                            indcpa.c     # CPA-secure
    encryption
                            poly.c       # Polynomial
    operations
                            ntt.c        # Number theoretic
    transform
                            ...
                      avx2/            # AVX2 optimized
                      aarch64/         # ARM64 optimized
                kyber768/
                kyber1024/
          crypto_sign/
                dilithium2/
                dilithium3/
                falcon-512/
                sphincs-sha2-128f-simple/
          common/
                fips202.c            # SHA-3/SHAKE
                sha2.c               # SHA-2
                aes.c                # AES
          test/
```

Listing 7.5: PQClean directory structure

7.3.3 API

```c
// api.h defines these constants for each scheme:
#define CRYPTO_SECRETKEYBYTES  ...
#define CRYPTO_PUBLICKEYBYTES  ...
#define CRYPTO_CIPHERTEXTBYTES ...  // KEM only
#define CRYPTO_BYTES           ...  // Shared secret / signature max

#define CRYPTO_ALGNAME "..."

// KEM API
int crypto_kem_keypair(uint8_t *pk, uint8_t *sk);
int crypto_kem_enc(uint8_t *ct, uint8_t *ss, const uint8_t *pk);
int crypto_kem_dec(uint8_t *ss, const uint8_t *ct, const uint8_t *sk);

// Signature API
int crypto_sign_keypair(uint8_t *pk, uint8_t *sk);

// Signed message (signature prepended to message)
int crypto_sign(uint8_t *sm, size_t *smlen,
                const uint8_t *m, size_t mlen,
                const uint8_t *sk);
int crypto_sign_open(uint8_t *m, size_t *mlen,
                     const uint8_t *sm, size_t smlen,
                     const uint8_t *pk);

// Detached signature
int crypto_sign_signature(uint8_t *sig, size_t *siglen,
                          const uint8_t *m, size_t mlen,
                          const uint8_t *sk);
int crypto_sign_verify(const uint8_t *sig, size_t siglen,
                       const uint8_t *m, size_t mlen,
                       const uint8_t *pk);
```

Listing 7.6: PQClean unified API

7.3.4 Integration Example

```c
// Include the specific scheme
```

```c
#include "crypto_kem/kyber768/clean/api.h"

// For multiple schemes, use namespaced versions
#define PQCLEAN_NAMESPACE(name) PQCLEAN_KYBER768_CLEAN_##name

int main() {
    uint8_t pk[CRYPTO_PUBLICKEYBYTES];
    uint8_t sk[CRYPTO_SECRETKEYBYTES];
    uint8_t ct[CRYPTO_CIPHERTEXTBYTES];
    uint8_t ss1[CRYPTO_BYTES], ss2[CRYPTO_BYTES];

    // Generate keypair
    PQCLEAN_NAMESPACE(crypto_kem_keypair)(pk, sk);

    // Encapsulate
    PQCLEAN_NAMESPACE(crypto_kem_enc)(ct, ss1, pk);

    // Decapsulate
    PQCLEAN_NAMESPACE(crypto_kem_dec)(ss2, ct, sk);

    // Verify
    if (memcmp(ss1, ss2, CRYPTO_BYTES) == 0) {
        printf("Key exchange successful!\n");
    }

    return 0;
}
```

Listing 7.7: Integrating PQClean into a project

7.4 pqm4: PQC for ARM Cortex-M4

pqm4 is the premier library for PQC on ARM Cortex-M4 microcontrollers, featuring highly optimized implementations with assembly kernels.

7.4.1 Supported Platforms

Table 7.2: pqm4 supported development boards

Board	MCU	RAM	Flash
NUCLEO-L4R5ZI	STM32L4R5ZI	640 KB	2 MB
NUCLEO-F446RE	STM32F446RE	128 KB	512 KB
STM32F4-Discovery	STM32F407VG	192 KB	1 MB
NUCLEO-L476RG	STM32L476RG	128 KB	1 MB
EFM32GG-STK3701A	EFM32GG11	512 KB	2 MB

7.4.2 Building and Running

```
# Clone repository
git clone https://github.com/mupq/pqm4.git
cd pqm4

# Install dependencies (Ubuntu/Debian)
sudo apt install gcc-arm-none-eabi openocd python3-serial

# Build all implementations for default board
make -j$(nproc)

# Build specific scheme
make IMPLEMENTATION_PATH=crypto_kem/kyber768/m4fspeed

# Flash to board
make flash IMPLEMENTATION_PATH=crypto_kem/kyber768/m4fspeed

# Run benchmarks
python3 benchmarks.py
```

Listing 7.8: Building pqm4

7.4.3 Implementation Variants

Table 7.3: pqm4 implementation variants

Variant	Description
`clean`	Reference C implementation, portable
`m4`	Basic M4 optimization (C with some ASM)
`m4f`	M4 with FPU utilization where applicable
`m4fspeed`	Maximum speed, higher memory usage
`m4fstack`	Optimized for minimal stack usage
`opt`	General optimized version

7.4.4 Benchmark Results

Table 7.4: Complete pqm4 benchmarks (STM32L4R5ZI @ 80 MHz)

Scheme	Variant	KeyGen	Enc/Sign	Dec/Ver	Stack
Key Encapsulation Mechanisms					
ML-KEM-512	m4fspeed	392,295	391,355	428,037	3,080
ML-KEM-768	m4fspeed	642,163	659,069	708,044	3,592
ML-KEM-1024	m4fspeed	1,019,877	1,031,009	1,094,181	4,104
ML-KEM-768	m4fstack	676,896	854,339	912,308	2,588
Digital Signatures					
ML-DSA-44	m4f	800,974	2,292,188	786,486	24,512
ML-DSA-65	m4f	1,406,298	3,948,962	1,380,582	34,752
ML-DSA-87	m4f	2,222,174	5,383,044	2,148,702	43,968
Falcon-512	opt	53.3M	7.0M	340,000	39,000
SLH-SHA2-128f	clean	2.9M	51.7M	3.5M	2,500

Cycles measured at 80 MHz. Stack in bytes. M = millions.

7.4.5 Using pqm4 in Your Project

```
// Project structure
// your_project/
//           src/
//                 main.c
//           pqm4/
//                 crypto_kem/kyber768/m4fspeed/
//           Makefile

// main.c
#include "api.h"
```

```c
#include "randombytes.h"

int main(void) {
    // Initialize hardware
    SystemClock_Config();
    UART_Init();
    RNG_Init();

    // PQC operations
    uint8_t pk[CRYPTO_PUBLICKEYBYTES];
    uint8_t sk[CRYPTO_SECRETKEYBYTES];
    uint8_t ct[CRYPTO_CIPHERTEXTBYTES];
    uint8_t ss[CRYPTO_BYTES];

    // Time measurement
    uint32_t start = DWT->CYCCNT;
    crypto_kem_keypair(pk, sk);
    uint32_t keygen_cycles = DWT->CYCCNT - start;

    printf("KeyGen: %lu cycles\n", keygen_cycles);

    return 0;
}
```

Listing 7.9: Integrating pqm4 implementations

7.5 wolfSSL / wolfCrypt

wolfSSL is a commercial-grade TLS library with comprehensive PQC support, ideal for production IoT deployments.

7.5.1 Features

- **FIPS 140-3 validated**: Certificate #4718
- **Production ready**: Used in billions of devices
- **Full TLS 1.3**: With PQ hybrid key exchange
- **DTLS 1.3**: For IoT protocols
- **Small footprint**: 20–100 KB code size
- **Dual license**: GPLv2 and commercial

7.5.2 PQC Algorithms

Table 7.5: wolfSSL PQC support

Algorithm	Status	FIPS
ML-KEM-512/768/1024	Supported	Hybrid*
ML-DSA-44/65/87	Supported	Planned
Falcon-512/1024	Supported	No
SPHINCS+	Supported	No
LMS/HSS	Supported	Yes
XMSS/XMSS^{MT}	Supported	Yes

*FIPS compliance via hybrid with ECDH

7.5.3 Building with PQC

```
# Clone wolfSSL
git clone https://github.com/wolfSSL/wolfssl.git
cd wolfssl

# Configure with PQC support
./autogen.sh
./configure --enable-kyber \
            --enable-dilithium \
            --enable-falcon \
            --enable-sphincs \
            --enable-tls13 \
            --enable-dtls13

# Build
make -j$(nproc)
sudo make install

# For embedded (ARM Cortex-M)
./configure --host=arm-none-eabi \
            --enable-kyber \
            --enable-singlethreaded \
            --disable-filesystem \
            CFLAGS="-mcpu=cortex-m4 -mthumb -Os"
```

Listing 7.10: Building wolfSSL with PQC

7.5.4 API Usage

```c
#include <wolfssl/wolfcrypt/kyber.h>
#include <wolfssl/wolfcrypt/dilithium.h>

// ML-KEM (Kyber) example
void kyber_example(void) {
    KyberKey key;
    byte pk[KYBER768_PUBLIC_KEY_SIZE];
    byte sk[KYBER768_PRIVATE_KEY_SIZE];
    byte ct[KYBER768_CIPHER_TEXT_SIZE];
    byte ss1[KYBER_SS_SZ], ss2[KYBER_SS_SZ];
    WC_RNG rng;

    wc_InitRng(&rng);
    wc_KyberKey_Init(KYBER768, &key, NULL, INVALID_DEVID);

    // Generate keypair
    wc_KyberKey_MakeKey(&key, &rng);
    wc_KyberKey_ExportPubKey(&key, pk, sizeof(pk));
    wc_KyberKey_ExportPrivKey(&key, sk, sizeof(sk));

    // Encapsulate
    wc_KyberKey_Encapsulate(&key, ct, ss1, &rng);

    // Decapsulate
    wc_KyberKey_Decapsulate(&key, ss2, ct, sizeof(ct));

    // Cleanup
    wc_KyberKey_Free(&key);
    wc_FreeRng(&rng);
}

// ML-DSA (Dilithium) example
void dilithium_example(void) {
    dilithium_key key;
    byte pk[DILITHIUM_LEVEL3_PUB_KEY_SIZE];
    byte sk[DILITHIUM_LEVEL3_PRV_KEY_SIZE];
    byte sig[DILITHIUM_LEVEL3_SIG_SIZE];
    word32 sigLen = sizeof(sig);
    byte msg[] = "Test message";
    WC_RNG rng;

```

```
42      wc_InitRng(&rng);
43      wc_dilithium_init(&key);
44      wc_dilithium_set_level(&key, 3);  // ML-DSA-65
45
46      // Generate keypair
47      wc_dilithium_make_key(&key, &rng);
48
49      // Sign
50      wc_dilithium_sign_msg(msg, sizeof(msg), sig, &sigLen,
            &key, &rng);
51
52      // Verify
53      int ret = wc_dilithium_verify_msg(sig, sigLen, msg,
            sizeof(msg),
54                                        &verified, &key);
55
56      wc_dilithium_free(&key);
57  }
```

Listing 7.11: wolfCrypt PQC API

7.5.5 TLS 1.3 with Hybrid PQC

```
1   #include <wolfssl/ssl.h>
2
3   // Server setup with PQ hybrid key exchange
4   WOLFSSL_CTX* create_pq_server_ctx(void) {
5       WOLFSSL_CTX* ctx =
            wolfSSL_CTX_new(wolfTLSv1_3_server_method());
6
7       // Enable hybrid key exchange
8       wolfSSL_CTX_set_groups(ctx,
9           (int[]){WOLFSSL_ECC_SECP256R1,
10                  WOLFSSL_KYBER_LEVEL3,       // Pure PQ
11                  WOLFSSL_P256_KYBER_LEVEL1}, // Hybrid
12          3);
13
14      // Load certificates
15      wolfSSL_CTX_use_certificate_file(ctx, "server.pem",
16                                       SSL_FILETYPE_PEM);
17      wolfSSL_CTX_use_PrivateKey_file(ctx, "server-key.pem",
18                                      SSL_FILETYPE_PEM);
```

```
19
20      return ctx;
21  }
22
23  // Client connection with PQ
24  int pq_client_connect(const char *host, int port) {
25      WOLFSSL_CTX* ctx =
            wolfSSL_CTX_new(wolfTLSv1_3_client_method());
26
27      // Prefer hybrid, fallback to pure PQ or classical
28      wolfSSL_CTX_set_groups(ctx,
29          (int[]){WOLFSSL_P256_KYBER_LEVEL1,  // Prefer hybrid
30                  WOLFSSL_KYBER_LEVEL3,        // Pure PQ
31                  WOLFSSL_ECC_SECP256R1},      // Classical
                        fallback
32          3);
33
34      // Connect
35      int sockfd = tcp_connect(host, port);
36      WOLFSSL* ssl = wolfSSL_new(ctx);
37      wolfSSL_set_fd(ssl, sockfd);
38
39      if (wolfSSL_connect(ssl) == SSL_SUCCESS) {
40          printf("Connected with %s\n",
                wolfSSL_get_cipher(ssl));
41      }
42
43      return 0;
44  }
```

Listing 7.12: wolfSSL TLS 1.3 with PQ hybrid

7.6 Rust Libraries

7.6.1 pqcrypto Crate

The `pqcrypto` crate provides safe Rust bindings to PQClean implementations.

```
1  // Cargo.toml
2  // [dependencies]
3  // pqcrypto-kyber = "0.8"
4  // pqcrypto-dilithium = "0.5"
```

```
// pqcrypto-traits = "0.3"

use pqcrypto_kyber::kyber768;
use pqcrypto_dilithium::dilithium3;
use pqcrypto_traits::kem::{PublicKey, SecretKey,
    SharedSecret, Ciphertext};
use pqcrypto_traits::sign::{SignedMessage,
    DetachedSignature};

fn kem_example() {
    // Key generation
    let (pk, sk) = kyber768::keypair();

    // Encapsulation
    let (ss_sender, ct) = kyber768::encapsulate(&pk);

    // Decapsulation
    let ss_receiver = kyber768::decapsulate(&ct, &sk);

    assert_eq!(ss_sender.as_bytes(), ss_receiver.as_bytes());
    println!("KEM successful!");
}

fn sign_example() {
    // Key generation
    let (pk, sk) = dilithium3::keypair();

    // Sign message
    let message = b"Important data to sign";
    let signed_msg = dilithium3::sign(message, &sk);

    // Verify and extract message
    match dilithium3::open(&signed_msg, &pk) {
        Ok(verified_msg) => {
            assert_eq!(&verified_msg[..], message);
            println!("Signature valid!");
        }
        Err(_) => println!("Signature invalid!"),
    }

    // Detached signature
```

```
    let sig = dilithium3::detached_sign(message, &sk);
    let valid = dilithium3::verify_detached_signature(&sig,
        message, &pk);
    assert!(valid.is_ok());
}

fn main() {
    kem_example();
    sign_example();
}
```

Listing 7.13: pqcrypto crate usage

7.6.2 RustCrypto PQC

The RustCrypto project is developing pure-Rust implementations:

```
// Cargo.toml
// [dependencies]
// ml-kem = "0.1"

use ml_kem::{MlKem768, KemCore, Encoded};
use rand_core::OsRng;

fn pure_rust_kem() {
    // Generate keypair
    let (dk, ek) = MlKem768::generate(&mut OsRng);

    // Encapsulate
    let (ct, ss_sender) = ek.encapsulate(&mut
        OsRng).unwrap();

    // Decapsulate
    let ss_receiver = dk.decapsulate(&ct).unwrap();

    assert_eq!(ss_sender.as_bytes(), ss_receiver.as_bytes());
}
```

Listing 7.14: ml-kem crate (pure Rust)

7.7 Development Tools

7.7.1 Testing Tools

Table 7.6: PQC testing and analysis tools

Tool	Purpose	URL
NIST KAT	Known Answer Tests	NIST PQC website
pqcgenkat	Generate KAT files	Part of PQClean
SUPERCOP	Benchmarking suite	bench.cr.yp.to
ctgrind	Constant-time check	github.com/agl/ctgrind
valgrind	Memory analysis	valgrind.org
AFL/libFuzzer	Fuzzing	Various

7.7.2 Side-Channel Analysis

Table 7.7: Side-channel analysis tools

Tool	Purpose	Open Source
ChipWhisperer	Power analysis platform	Yes
Lascar	Side-channel analysis library	Yes
Scared	Side-channel evaluation	Yes
REASSURE	Leakage assessment	Yes
Inspector	Commercial SCA suite	No

7.7.3 Formal Verification

Table 7.8: Formal verification for cryptography

Tool	Purpose	Language
EasyCrypt	Cryptographic proofs	EasyCrypt
CryptoVerif	Protocol verification	CryptoVerif
Jasmin	Verified assembly	Jasmin
Fiat-Crypto	Verified field arithmetic	Coq
HACL*	Verified crypto library	F*

7.8 Library Selection Guide

7.8.1 Decision Matrix

Table 7.9: Library selection by use case

Use Case	liboqs	PQClean	pqm4	wolfSSL	pqcrypto
Research/Prototyping	✓	✓			✓
Embedded (Cortex-M)		✓	✓	✓	
Production TLS				✓	
FIPS compliance				✓	
Algorithm comparison	✓	✓			
Minimal dependencies		✓	✓		
Rust projects					✓
Python scripts	✓				

7.8.2 Recommendations by Platform

Table 7.10: Recommended libraries by platform

Platform	Recommended Library
Linux/Desktop	liboqs or wolfSSL
ARM Cortex-M4/M7	pqm4 or wolfSSL
ARM Cortex-M0/M3	PQClean (clean variants)
ESP32	wolfSSL or adapted PQClean
RISC-V	PQClean or liboqs
Web/WASM	liboqs (WASM build)
iOS/Android	liboqs or wolfSSL
Rust anywhere	pqcrypto or ml-kem
Python scripts	liboqs-python

7.9 Chapter Summary

This chapter surveyed the major PQC libraries and tools:

General-Purpose Libraries:

- **liboqs**: Most comprehensive, excellent for research and prototyping
- **PQClean**: Clean reference implementations, no dependencies
- **Botan**: C++ library with PQC support

Embedded/IoT Libraries:

- **pqm4**: Optimized for ARM Cortex-M4, best performance

- **wolfSSL**: Production-ready, FIPS validated, TLS support
- **PQClean clean variants**: Portable to any platform

Language-Specific:

- **Rust**: pqcrypto, ml-kem (pure Rust)
- **Python**: liboqs-python
- **Go**: liboqs-go

Key Recommendations:

- For Cortex-M4 IoT: Use **pqm4** for best performance
- For production TLS: Use **wolfSSL** with hybrid mode
- For research: Use **liboqs** for comprehensive algorithm access
- For minimal footprint: Use **PQClean** clean implementations

Note: Always verify library versions support the final NIST standards (FIPS 203, 204, 205). Earlier versions may implement draft specifications with incompatible parameters or APIs.

Chapter 8

Protocol Integration

This chapter covers the integration of post-quantum cryptography into IoT communication protocols. We examine TLS/DTLS, MQTT, CoAP, Matter, and other protocols essential for IoT security, providing practical guidance for PQC migration.

8.1 TLS 1.3 with Post-Quantum Cryptography

Transport Layer Security (TLS) 1.3 is the foundation for secure communication in IoT. Integrating PQC into TLS requires careful consideration of handshake overhead and backward compatibility.

8.1.1 TLS 1.3 Handshake Overview

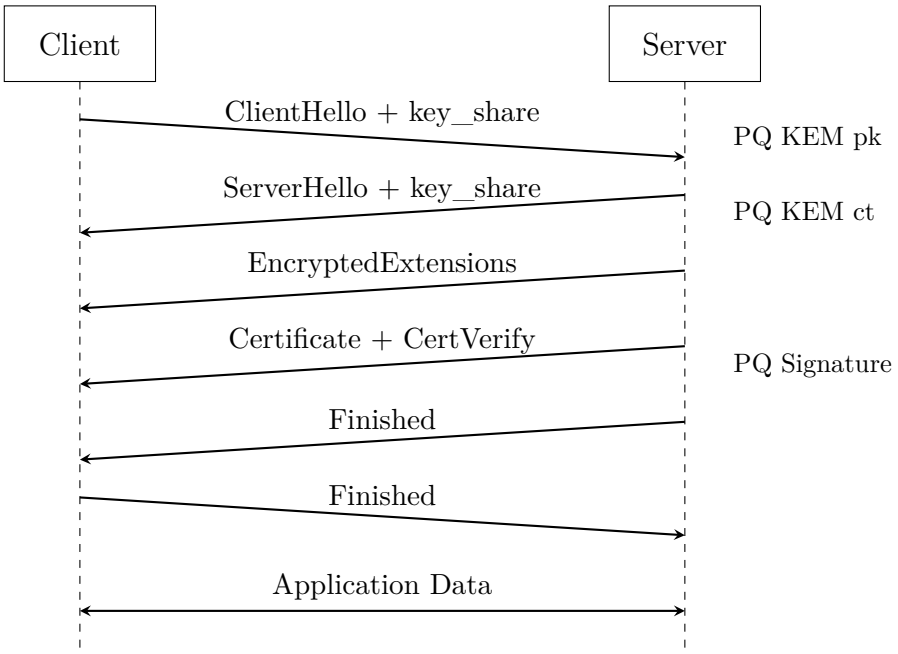


Figure 8.1: TLS 1.3 handshake with PQC integration points

8.1.2 Key Exchange Integration

TLS 1.3 key exchange uses ephemeral Diffie-Hellman, which can be replaced or augmented with PQC KEMs.

Hybrid Key Exchange

The recommended approach combines classical and post-quantum algorithms:

```
// TLS 1.3 key_share extension with hybrid
struct {
    NamedGroup group;  // e.g., x25519_kyber768
    opaque key_exchange<1..2^16-1>;
} KeyShareEntry;

// Hybrid key_exchange format
struct {
    opaque classical_key<1..255>;     // X25519: 32 bytes
    opaque pq_key<1..2^16-1>;         // ML-KEM-768: 1184
       bytes
} HybridKeyShare;

// Total ClientHello key_share: 32 + 1184 = 1216 bytes
// Total ServerHello key_share: 32 + 1088 = 1120 bytes
   (ciphertext)
```

Listing 8.1: Hybrid key share structure

IANA Registered Groups

Table 8.1: TLS named groups for PQC (draft registrations)

Name	Components	Code Point	Status
	Hybrid Groups		
x25519_kyber768	X25519 + ML-KEM-768	0x6399	Draft
secp256r1_kyber768	P-256 + ML-KEM-768	0x639a	Draft
x25519_kyber512	X25519 + ML-KEM-512	0x6398	Draft
secp384r1_kyber1024	P-384 + ML-KEM-1024	0x639b	Draft
	Pure PQ Groups (future)		
mlkem512	ML-KEM-512	TBD	Planned
mlkem768	ML-KEM-768	TBD	Planned
mlkem1024	ML-KEM-1024	TBD	Planned

8.1.3 Authentication Integration

Certificate-based authentication requires PQ signatures in both certificates and handshake messages.

Certificate Chain Impact

Table 8.2: Certificate chain size comparison

Algorithm	pk Size	sig Size	Leaf Cert	Chain (3)
RSA-2048	256 B	256 B	∼1.5 KB	∼4.5 KB
ECDSA P-256	64 B	64 B	∼0.8 KB	∼2.4 KB
ML-DSA-65	1,952 B	3,293 B	∼6 KB	∼18 KB
Falcon-512	897 B	666 B	∼2.5 KB	∼7.5 KB
SLH-DSA-128f	32 B	17,088 B	∼18 KB	∼54 KB

Warning: PQ certificates significantly increase handshake size. A typical ML-DSA certificate chain is 7× larger than ECDSA. Consider certificate compression (RFC 8879) and caching strategies.

Hybrid Certificates

During transition, certificates may contain both classical and PQ keys:

```
# Using OQS-OpenSSL provider
export OPENSSL_MODULES=/usr/local/lib/ossl-modules

# Generate hybrid CA key (ECDSA + ML-DSA)
openssl genpkey -algorithm ec -pkeyopt
    ec_paramgen_curve:P-256 \
    -out ca-classical.key
openssl genpkey -provider oqsprovider -algorithm mldsa65 \
    -out ca-pq.key

# Create composite key (implementation specific)
# Some implementations use X.509 alternative signatures
    extension

# Generate leaf certificate with PQ signature
openssl req -new -x509 \
    -provider oqsprovider \
    -key server.key \
    -sigopt algorithm:mldsa65 \
    -out server-pq.crt \
```

```
    -days 365 \
    -subj "/CN=iot-device.local"
```

Listing 8.2: Creating hybrid certificate with OpenSSL + OQS

8.1.4 Handshake Size Analysis

Table 8.3: TLS 1.3 handshake size comparison

Configuration	ClientHello	ServerHello+	Total	RTT
ECDHE + ECDSA	200 B	3.5 KB	4.2 KB	1-RTT
X25519 + Ed25519	180 B	2.8 KB	3.3 KB	1-RTT
Hybrid (X25519+Kyber768)	1.4 KB	5.5 KB	7.4 KB	1-RTT
+ ML-DSA-65 certs	1.4 KB	22 KB	24 KB	1-RTT
Pure PQ (Kyber768)	1.3 KB	5.2 KB	7 KB	1-RTT
+ ML-DSA-65 certs	1.3 KB	21 KB	23 KB	1-RTT
+ Falcon-512 certs	1.3 KB	11 KB	13 KB	1-RTT

8.1.5 Performance Impact

Table 8.4: TLS 1.3 handshake time on ARM Cortex-M4 @ 80 MHz

Configuration	Client (ms)	Server (ms)	Total (ms)
ECDHE P-256 + ECDSA	45	55	100
X25519 + Ed25519	25	30	55
Hybrid + ECDSA	35	45	80
Hybrid + ML-DSA-65	85	110	195
Hybrid + Falcon-512	95	50	145
Pure PQ + ML-DSA-65	75	100	175
Pure PQ + Falcon-512	85	40	125

8.1.6 Implementation with wolfSSL

```
#include <wolfssl/ssl.h>
#include <wolfssl/wolfcrypt/settings.h>

int pq_tls_client(const char *host, int port) {
    WOLFSSL_CTX *ctx;
    WOLFSSL *ssl;
    int sockfd, ret;

```

```
    // Initialize wolfSSL
    wolfSSL_Init();

    // Create TLS 1.3 context
    ctx = wolfSSL_CTX_new(wolfTLSv1_3_client_method());
    if (!ctx) {
        printf("Failed to create context\n");
        return -1;
    }

    // Configure hybrid key exchange
    // Priority: hybrid > pure PQ > classical
    ret = wolfSSL_CTX_set_groups(ctx, (int[]){
        WOLFSSL_P256_KYBER_LEVEL1,   // P-256 + ML-KEM-512
        WOLFSSL_KYBER_LEVEL3,        // ML-KEM-768
        WOLFSSL_ECC_SECP256R1        // Fallback to
            classical
    }, 3);

    // Load CA certificates (may include PQ certs)
    wolfSSL_CTX_load_verify_locations(ctx, "ca-bundle.pem",
        NULL);

    // Create socket and connect
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr = {
        .sin_family = AF_INET,
        .sin_port = htons(port)
    };
    inet_pton(AF_INET, host, &addr.sin_addr);
    connect(sockfd, (struct sockaddr *)&addr, sizeof(addr));

    // Create SSL object
    ssl = wolfSSL_new(ctx);
    wolfSSL_set_fd(ssl, sockfd);

    // Perform handshake
    ret = wolfSSL_connect(ssl);
    if (ret != SSL_SUCCESS) {
        int err = wolfSSL_get_error(ssl, ret);
        char errBuf[256];
```

```
48          wolfSSL_ERR_error_string(err, errBuf);
49          printf("TLS handshake failed: %s\n", errBuf);
50          return -1;
51      }
52
53      // Print connection info
54      printf("Connected with %s\n", wolfSSL_get_cipher(ssl));
55      printf("Key exchange: %s\n",
            wolfSSL_get_curve_name(ssl));
56
57      // Send/receive data
58      const char *msg = "GET / HTTP/1.1\r\nHost:
            example.com\r\n\r\n";
59      wolfSSL_write(ssl, msg, strlen(msg));
60
61      char buffer[4096];
62      int bytes = wolfSSL_read(ssl, buffer, sizeof(buffer) -
            1);
63      buffer[bytes] = '\0';
64      printf("Response:\n%s\n", buffer);
65
66      // Cleanup
67      wolfSSL_shutdown(ssl);
68      wolfSSL_free(ssl);
69      close(sockfd);
70      wolfSSL_CTX_free(ctx);
71      wolfSSL_Cleanup();
72
73      return 0;
74  }
```

Listing 8.3: Complete TLS 1.3 PQ client

8.2 DTLS 1.3 for Constrained IoT

Datagram TLS (DTLS) provides TLS security over UDP, essential for IoT protocols that cannot use TCP.

8.2.1 DTLS vs TLS for IoT

Table 8.5: DTLS vs TLS comparison for IoT

Aspect	TLS 1.3	DTLS 1.3
Transport	TCP	UDP
Connection state	Required	Optional
Packet ordering	Guaranteed	Not guaranteed
Retransmission	TCP handles	DTLS handles
Fragmentation	TCP handles	DTLS handles
NAT traversal	Easier	Harder
Multicast	Not supported	Supported
IoT protocols	MQTT, HTTP	CoAP, LwM2M

8.2.2 DTLS Fragmentation with PQC

Large PQ key shares and certificates require DTLS fragmentation, which adds complexity and latency.

Table 8.6: DTLS fragmentation impact

MTU	Classical	Hybrid KE	+ ML-DSA cert
1500 B (Ethernet)	1 frag	1 frag	15 frags
1280 B (IPv6 min)	1 frag	2 frags	18 frags
576 B (IPv4 min)	2 frags	4 frags	42 frags
127 B (LoRaWAN)	15 frags	55 frags	180+ frags

Warning: For constrained networks like LoRaWAN (127 B MTU), PQ DTLS may be impractical. Consider connection ID (RFC 9146) to reduce repeated handshakes, or use pre-shared keys with PQ KEM for initial provisioning.

8.2.3 Connection ID for IoT

```
// Enable Connection ID to avoid re-handshakes
// Useful when NAT rebinding or IP changes occur

// wolfSSL configuration
WOLFSSL_CTX *ctx =
    wolfSSL_CTX_new(wolfDTLSv1_3_client_method());

// Enable Connection ID
wolfSSL_CTX_set_options(ctx, WOLFSSL_OP_CONNECTION_ID);

// Set CID length (typically 4-8 bytes)
```

```
wolfSSL_CTX_set_cid_len(ctx, 4);

// After handshake, connection survives IP changes
// Re-handshake only needed for key refresh
```

Listing 8.4: DTLS with Connection ID

8.3 MQTT with PQC

MQTT (Message Queuing Telemetry Transport) is the dominant IoT messaging protocol. PQC integration occurs at the TLS layer.

8.3.1 MQTT Architecture

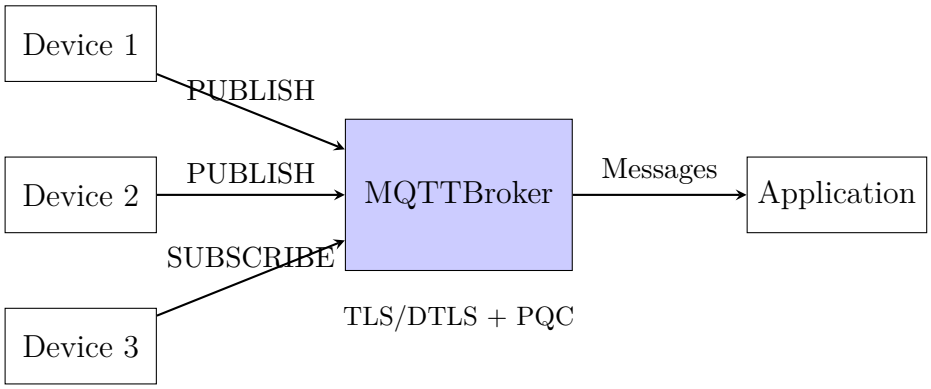


Figure 8.2: MQTT with PQC-secured TLS connections

8.3.2 MQTT over TLS with PQC

```
#include <MQTTClient.h>
#include <wolfssl/ssl.h>

#define BROKER_URI "ssl://iot-broker.example.com:8883"
#define CLIENT_ID "pq-iot-device-001"

// Custom SSL context creation with PQC
SSL_CTX* create_pq_ssl_ctx(void) {
    SSL_CTX *ctx =
       wolfSSL_CTX_new(wolfTLSv1_3_client_method());

    // Configure PQ hybrid key exchange
    wolfSSL_CTX_set_groups(ctx, (int[]){
        WOLFSSL_P256_KYBER_LEVEL1,
```

```
14          WOLFSSL_ECC_SECP256R1
15      }, 2);
16
17      // Load device certificate (could be PQ-signed)
18      wolfSSL_CTX_use_certificate_file(ctx, "device.crt",
           SSL_FILETYPE_PEM);
19      wolfSSL_CTX_use_PrivateKey_file(ctx, "device.key",
           SSL_FILETYPE_PEM);
20
21      // Load CA bundle
22      wolfSSL_CTX_load_verify_locations(ctx, "ca-bundle.pem",
           NULL);
23
24      return ctx;
25  }
26
27  int mqtt_pq_connect(void) {
28      MQTTClient client;
29      MQTTClient_connectOptions conn_opts =
           MQTTClient_connectOptions_initializer;
30      MQTTClient_SSLOptions ssl_opts =
           MQTTClient_SSLOptions_initializer;
31
32      MQTTClient_create(&client, BROKER_URI, CLIENT_ID,
33                        MQTTCLIENT_PERSISTENCE_NONE, NULL);
34
35      // Configure SSL with PQ
36      ssl_opts.ssl_ctx = create_pq_ssl_ctx();
37      ssl_opts.verify = 1;
38      conn_opts.ssl = &ssl_opts;
39
40      // Connect
41      conn_opts.keepAliveInterval = 60;
42      conn_opts.cleansession = 1;
43
44      int rc = MQTTClient_connect(client, &conn_opts);
45      if (rc != MQTTCLIENT_SUCCESS) {
46          printf("MQTT connection failed: %d\n", rc);
47          return -1;
48      }
49
```

```
50      printf("Connected to MQTT broker with PQ TLS\n");
51
52      // Publish sensor data
53      const char *topic = "sensors/temperature";
54      const char *payload = "{\"temp\": 23.5, \"unit\":
           \"C\"}";
55
56      MQTTClient_publish(client, topic, strlen(payload),
           payload, 1, 0, NULL);
57
58      return 0;
59  }
```

Listing 8.5: MQTT client with PQ TLS (Paho + wolfSSL)

8.3.3 MQTT-SN for Constrained Devices

MQTT-SN (Sensor Networks) is designed for constrained devices and can use DTLS:

Table 8.7: MQTT vs MQTT-SN for PQC

Aspect	MQTT	MQTT-SN
Transport	TCP	UDP, ZigBee, etc.
Security layer	TLS	DTLS
Message size	Variable	Optimized (small)
Topic handling	String names	Numeric IDs
Gateway	Not needed	Required
PQ suitability	Good	Limited*

*Limited by DTLS fragmentation overhead

8.4 CoAP with PQC

Constrained Application Protocol (CoAP) is a lightweight RESTful protocol for IoT, secured with DTLS or OSCORE.

8.4.1 CoAP Security Options

Table 8.8: CoAP security modes

Mode	Security	End-to-End	PQ Support
NoSec	None	N/A	N/A
DTLS (PSK)	Transport	No (hop-by-hop)	Limited
DTLS (Cert)	Transport	No (hop-by-hop)	Yes (hybrid)
OSCORE	Object	Yes	Planned

8.4.2 OSCORE: Object Security for CoAP

OSCORE (RFC 8613) provides end-to-end security for CoAP, independent of transport security.

```c
// OSCORE security context derived from PQ KEM
typedef struct {
    uint8_t master_secret[32];   // From ML-KEM shared secret
    uint8_t master_salt[8];      // Optional
    uint8_t sender_id[8];
    size_t sender_id_len;
    uint8_t recipient_id[8];
    size_t recipient_id_len;
    uint8_t common_iv[13];       // Derived
    uint8_t sender_key[16];      // Derived (AES-128)
    uint8_t recipient_key[16];   // Derived
} oscore_ctx_t;

// Initialize OSCORE from ML-KEM exchange
int oscore_init_from_kem(oscore_ctx_t *ctx,
                         const uint8_t kem_shared_secret[32],
                         const uint8_t *sender_id, size_t
                            sender_id_len,
                         const uint8_t *recipient_id, size_t
                            recipient_id_len) {
    // Copy shared secret as master secret
    memcpy(ctx->master_secret, kem_shared_secret, 32);

    // Copy IDs
    memcpy(ctx->sender_id, sender_id, sender_id_len);
    ctx->sender_id_len = sender_id_len;
    memcpy(ctx->recipient_id, recipient_id,
       recipient_id_len);
```

```
26      ctx->recipient_id_len = recipient_id_len;
27
28      // Derive keys using HKDF
29      // Common IV = HKDF(master_secret, info="IV")
30      hkdf_sha256(ctx->common_iv, 13,
31                  ctx->master_secret, 32,
32                  NULL, 0,  // salt
33                  "OSCORE IV", 9);
34
35      // Sender key = HKDF(master_secret, info="Key" ||
            sender_id)
36      uint8_t info[64];
37      size_t info_len = 0;
38      memcpy(info, "Key", 3); info_len += 3;
39      memcpy(info + info_len, sender_id, sender_id_len);
40      info_len += sender_id_len;
41
42      hkdf_sha256(ctx->sender_key, 16,
43                  ctx->master_secret, 32,
44                  NULL, 0, info, info_len);
45
46      // Similarly derive recipient_key...
47
48      return 0;
49  }
```

Listing 8.6: OSCORE context with PQ key establishment

8.4.3 EDHOC: Lightweight Key Exchange

EDHOC (Ephemeral Diffie-Hellman Over COSE) is a compact key exchange protocol designed for constrained IoT.

Table 8.9: EDHOC vs DTLS handshake comparison

Aspect	EDHOC	DTLS 1.3
Messages	3	6+
Total bytes (ECDH)	∼200 B	∼500 B
Total bytes (PQ hybrid)	∼2.5 KB	∼4 KB
Round trips	1.5	2+
Result	OSCORE context	TLS record layer
PQ extension	Draft	Supported

```
// EDHOC message 1 with hybrid key share
typedef struct {
    uint8_t method;              // Key exchange method
    uint8_t suites[4];           // Cipher suites
    uint8_t g_x_classical[32];   // X25519 ephemeral public
    uint8_t g_x_pq[1184];        // ML-KEM-768 encapsulation
       key
    uint8_t c_i;                 // Connection identifier
       initiator
} edhoc_msg1_pq_t;

// EDHOC message 2 with hybrid response
typedef struct {
    uint8_t c_i;                 // Echo connection ID
    uint8_t g_y_classical[32];   // X25519 ephemeral public
    uint8_t g_y_pq[1088];        // ML-KEM-768 ciphertext
    uint8_t ciphertext[];        // Encrypted ID_CRED_R,
       Signature
} edhoc_msg2_pq_t;

// Shared secret derivation
// ss = KDF(X25519_ss || ML-KEM_ss, context)
```

Listing 8.7: EDHOC with PQ (conceptual)

8.5 Matter Protocol

Matter is the new smart home standard supported by Apple, Google, Amazon, and Samsung. PQC migration is being planned for future versions.

8.5.1 Matter Security Architecture

Table 8.10: Matter security layers

Layer	Current	PQ Migration Path
Device attestation	ECDSA P-256	ML-DSA or Falcon
Commissioning	SPAKE2+	SPAKE2+ (quantum-safe?)
Session security	CASE (ECDH)	Hybrid ECDH + ML-KEM
Message encryption	AES-CCM-128	AES-CCM-128 (unchanged)

8.5.2 Matter Commissioning with PQC

```
// Certificate Authenticated Session Establishment (CASE)
// Modified for hybrid PQ key exchange

typedef struct {
    // Session identifiers
    uint16_t session_id;
    uint64_t node_id;

    // Key exchange (hybrid)
    struct {
        uint8_t ecdh_public[65];      // P-256 uncompressed
        uint8_t mlkem_encap[1184];    // ML-KEM-768
           encapsulation key
    } ephemeral_keys;

    // Device certificate (currently ECDSA, future: ML-DSA)
    uint8_t certificate[MAX_CERT_SIZE];

    // Signature over transcript
    uint8_t signature[MAX_SIG_SIZE];

} matter_case_sigma1_pq_t;

// Hybrid shared secret computation
int matter_compute_shared_secret(
    uint8_t shared_secret[32],
    const uint8_t *ecdh_private,
    const uint8_t *ecdh_peer_public,
    const uint8_t *mlkem_private,
    const uint8_t *mlkem_ciphertext
) {
    uint8_t ecdh_ss[32];
    uint8_t mlkem_ss[32];

    // ECDH key agreement
    ecdh_compute_shared(ecdh_ss, ecdh_private,
       ecdh_peer_public);

    // ML-KEM decapsulation
```

```
38      mlkem768_decaps(mlkem_ss, mlkem_ciphertext,
            mlkem_private);
39
40      // Combine: SS = KDF(ECDH_SS || MLKEM_SS)
41      uint8_t combined[64];
42      memcpy(combined, ecdh_ss, 32);
43      memcpy(combined + 32, mlkem_ss, 32);
44
45      hkdf_sha256(shared_secret, 32, combined, 64,
46                  "Matter CASE Hybrid", 18);
47
48      // Securely clear intermediates
49      secure_zero(ecdh_ss, 32);
50      secure_zero(mlkem_ss, 32);
51      secure_zero(combined, 64);
52
53      return 0;
54  }
```

Listing 8.8: Matter CASE with hybrid key exchange

8.6 LoRaWAN Security

LoRaWAN presents unique challenges for PQC due to extreme bandwidth constraints.

8.6.1 LoRaWAN Constraints

Table 8.11: LoRaWAN data rate limitations

Spreading Factor	Data Rate	Max Payload	ToA (51B)
SF7 (DR5)	5.5 kbps	242 B	72 ms
SF8 (DR4)	3.1 kbps	242 B	144 ms
SF9 (DR3)	1.8 kbps	123 B	267 ms
SF10 (DR2)	980 bps	59 B	493 ms
SF11 (DR1)	440 bps	59 B	987 ms
SF12 (DR0)	250 bps	59 B	1,810 ms

Warning: ML-KEM-768 public key (1,184 B) would require 5+ LoRaWAN frames at SF7 or 20+ frames at SF12. Standard PQC key exchange is impractical for LoRaWAN.

8.6.2 PQC Strategies for LoRaWAN

Option 1: Pre-Provisioned Keys

```
// At manufacturing time:
// 1. Generate ML-KEM keypair for network server
// 2. Encapsulate shared secret with device's public key
// 3. Store symmetric key in device secure element

typedef struct {
    uint8_t dev_eui[8];           // Device identifier
    uint8_t app_key[16];          // Derived from ML-KEM SS
       (AES-128)
    uint8_t nwk_key[16];          // Derived from ML-KEM SS
    uint8_t kem_key_id[4];        // Reference to PQ key used
} lorawan_pq_credentials_t;

// Join procedure uses pre-shared keys
// No PQ exchange needed over-the-air
```

Listing 8.9: PQ key pre-provisioning for LoRaWAN

Option 2: Out-of-Band Key Exchange

```
// PQ key exchange via different channel (WiFi, BLE, USB)
// before LoRaWAN commissioning

int lorawan_pq_oob_setup(lorawan_device_t *device) {
    // Step 1: Establish BLE connection for key exchange
    ble_connect(device->ble_address);

    // Step 2: Perform ML-KEM key exchange over BLE
    uint8_t pk[MLKEM768_PK_SIZE];
    uint8_t ct[MLKEM768_CT_SIZE];
    uint8_t ss[32];

    mlkem768_keypair(pk, device->sk);
    ble_send(pk, sizeof(pk));
    ble_receive(ct, sizeof(ct));
    mlkem768_decaps(ss, ct, device->sk);

    // Step 3: Derive LoRaWAN keys from shared secret
    derive_lorawan_keys(&device->credentials, ss);
```

```
20
21      // Step 4: Proceed with standard LoRaWAN join
22      // (now using PQ-derived keys)
23      ble_disconnect();
24      lorawan_join(device);
25
26      return 0;
27  }
```

Listing 8.10: Out-of-band PQ key exchange

Option 3: Lightweight PQC Research

Research is ongoing for PQC schemes optimized for constrained environments:

- **FrodoKEM**: More conservative, but larger keys
- **NTRU LPRime**: Potentially smaller keys
- **Symmetric ratcheting**: Extend initial PQ key

8.7 Zigbee and Thread

8.7.1 Zigbee Security

Zigbee uses AES-128 for link and network security. PQC integration would affect:

- **Trust Center**: Key distribution mechanism
- **Install codes**: Device provisioning
- **Link keys**: Pairwise device keys

8.7.2 Thread Security

Thread uses DTLS for commissioning and MLE (Mesh Link Establishment) for network security.

Table 8.12: Thread security with PQC considerations

Phase	Current Security	PQ Upgrade Path
Commissioning	DTLS 1.2 + J-PAKE	DTLS 1.3 + hybrid
MLE	AES-CCM	AES-CCM (unchanged)
Border Router	TLS 1.2	TLS 1.3 + hybrid
Matter integration	CASE	Hybrid CASE

8.8 Protocol Migration Strategy

8.8.1 Phased Approach

Table 8.13: Recommended PQC protocol migration phases

Phase	Timeframe	Action	Impact
1	2024–2025	Inventory cryptographic assets	None
2	2025–2026	Test hybrid TLS/DTLS	Low
3	2026–2027	Deploy hybrid in production	Medium
4	2027–2030	Transition to pure PQ	High
5	2030+	Deprecate classical-only	Full

8.8.2 Backward Compatibility

```
// Server-side: support both classical and hybrid
int configure_flexible_server(WOLFSSL_CTX *ctx) {
    // Offer multiple options in preference order
    int groups[] = {
        // Hybrid (preferred)
        WOLFSSL_P256_KYBER_LEVEL1,
        WOLFSSL_X25519_KYBER_LEVEL1,
        // Pure PQ (if client supports)
        WOLFSSL_KYBER_LEVEL3,
        // Classical fallback
        WOLFSSL_ECC_SECP256R1,
        WOLFSSL_ECC_X25519
    };

    wolfSSL_CTX_set_groups(ctx, groups,
                           sizeof(groups)/sizeof(groups[0]));

    // Log negotiated algorithm for monitoring
    wolfSSL_CTX_set_info_callback(ctx, connection_info_cb);

    return 0;
}

void connection_info_cb(const WOLFSSL *ssl, int type, int
    val) {
    if (type == SSL_CB_HANDSHAKE_DONE) {
        const char *kex = wolfSSL_get_curve_name(ssl);
```

```
27          const char *cipher = wolfSSL_get_cipher(ssl);
28
29          if (strstr(kex, "KYBER") || strstr(kex, "kyber")) {
30              log_info("PQ-secured connection: %s, %s", kex,
                    cipher);
31          } else {
32              log_warn("Classical-only connection: %s, %s",
                    kex, cipher);
33          }
34      }
35  }
```

Listing 8.11: Negotiating PQ support

8.9 Chapter Summary

This chapter covered PQC integration into IoT protocols:

TLS 1.3:

- Hybrid key exchange (X25519 + ML-KEM) is recommended
- Certificate chains grow 5–7× with ML-DSA
- Handshake overhead is manageable for most IoT

DTLS 1.3:

- Fragmentation adds significant overhead
- Connection ID reduces re-handshake frequency
- May be impractical for very constrained networks

MQTT:

- PQC via underlying TLS layer
- MQTT-SN faces DTLS fragmentation challenges

CoAP:

- DTLS provides transport security
- OSCORE offers end-to-end security
- EDHOC provides lightweight key exchange

LoRaWAN:

- Standard PQC key exchange is impractical
- Use pre-provisioned keys or out-of-band exchange

Matter:

- PQC migration planned for future versions
- Hybrid CASE key exchange

Note: For most IoT deployments, **hybrid TLS 1.3** with X25519 + ML-KEM-768 provides the best balance of security and compatibility. Deploy hybrid now to protect long-lived data against future quantum attacks.

Chapter 9

Security Analysis and Best Practices

The deployment of post-quantum cryptography in IoT environments introduces unique security challenges that extend beyond algorithm selection. This chapter provides a comprehensive security analysis framework, covering threat modeling specific to PQC-enabled IoT systems, side-channel vulnerability assessment, and security parameter selection guidelines. We present practical methodologies for evaluating and mitigating risks in constrained deployments.

9.1 Threat Modeling for PQC-IoT Systems

9.1.1 The STRIDE-PQ Framework

Traditional threat modeling frameworks require adaptation for post-quantum scenarios. We extend the STRIDE model to address quantum-specific threats:

Table 9.1: STRIDE-PQ: Extended Threat Model for Post-Quantum IoT

Category	Classical Threat	Quantum Extension
Spoofing	Identity theft	Forged PQ signatures, compromised device attestation
Tampering	Data modification	Algorithm substitution attacks, parameter manipulation
Repudiation	Denial of actions	Signature scheme migration disputes
Information Disclosure	Data leakage	Harvest-now-decrypt-later (HNDL), side-channel extraction
Denial of Service	Resource exhaustion	PQ computation DoS, memory exhaustion attacks
Elevation of Privilege	Unauthorized access	Quantum-assisted key recovery, hybrid downgrade

9.1.2 Harvest-Now-Decrypt-Later (HNDL) Risk Assessment

The HNDL threat represents the most pressing concern for IoT deployments. Adversaries intercept and store encrypted communications today, planning to decrypt them once quantum computers become available.

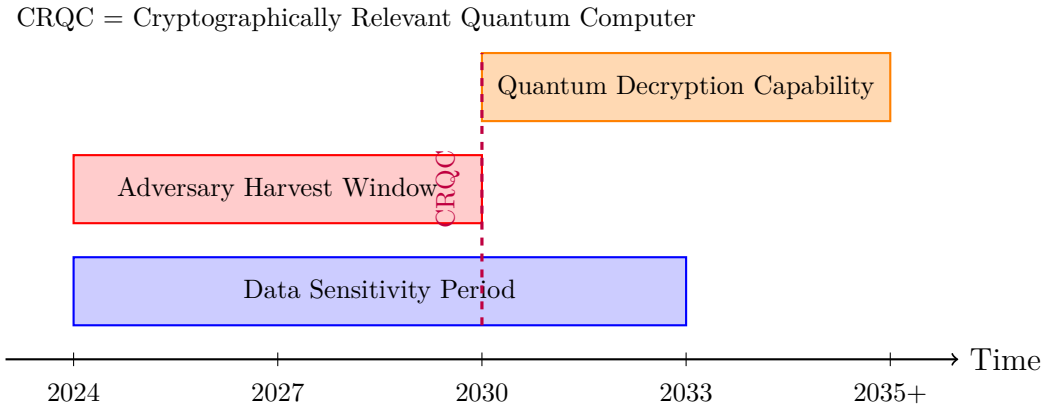


Figure 9.1: HNDL Timeline Risk Visualization

Warning: If your IoT data must remain confidential for X years, and you estimate CRQC arrival in Y years, you must deploy PQC within $(Y - X)$ years. For medical devices with 25-year data retention and 2030 CRQC estimates, PQC deployment should have started in 2020.

HNDL Risk Calculation Formula:

$$\text{HNDL\_Risk} = P(\text{CRQC}_t) \times V(\text{data}) \times \left(1 - e^{-\lambda \cdot T_{\text{sensitive}}}\right) \tag{9.1}$$

Where:

- $P(\text{CRQC}_t)$ = Probability of CRQC availability by time t
- $V(\text{data})$ = Value/sensitivity of the data
- λ = Data capture rate by adversaries
- $T_{\text{sensitive}}$ = Duration data remains sensitive

9.1.3 IoT-Specific Attack Surfaces

IoT devices present unique attack surfaces that differ significantly from traditional computing environments:

```
1 /* IoT PQC Attack Surface Analysis */
2
3 typedef enum {
```

```
4       /* Physical Layer Attacks */
5       ATTACK_PHYSICAL_PROBING,        // Direct hardware access
6       ATTACK_FAULT_INJECTION,         // Voltage/clock glitching
7       ATTACK_COLD_BOOT,               // Memory extraction
8       ATTACK_JTAG_DEBUG,              // Debug interface
           exploitation
9
10      /* Side-Channel Attacks */
11      ATTACK_TIMING,                  // Execution time analysis
12      ATTACK_POWER_ANALYSIS,          // SPA/DPA/CPA
13      ATTACK_EM_EMANATION,            // Electromagnetic analysis
14      ATTACK_ACOUSTIC,                // Sound-based leakage
15      ATTACK_CACHE_TIMING,            // Cache access patterns
16
17      /* Cryptographic Attacks */
18      ATTACK_IMPLEMENTATION,          // Incorrect implementation
19      ATTACK_PARAMETER_MISUSE,        // Wrong security level
20      ATTACK_RNG_WEAKNESS,            // Poor randomness
21      ATTACK_KEY_REUSE,               // Nonce/key reuse
22      ATTACK_HYBRID_DOWNGRADE,        // Force classical-only
23
24      /* Protocol Attacks */
25      ATTACK_REPLAY,                  // Message replay
26      ATTACK_MITM,                    // Man-in-the-middle
27      ATTACK_FRAGMENTATION,           // Reassembly attacks
28      ATTACK_CERTIFICATE_MISUSE,      // Cert validation bypass
29
30      /* Supply Chain Attacks */
31      ATTACK_FIRMWARE_TAMPERING,      // Modified firmware
32      ATTACK_BACKDOOR,                // Malicious code insertion
33      ATTACK_COMPONENT_SUBSTITUTION // Counterfeit components
34  } iot_attack_vector_t;
35
36  /* Risk scoring structure */
37  typedef struct {
38      iot_attack_vector_t vector;
39      uint8_t likelihood;       // 1-10 scale
40      uint8_t impact;           // 1-10 scale
41      uint8_t detectability;    // 1-10 (10 = hard to detect)
42      char mitigation[256];
43  } attack_risk_t;
```

```
44
45 /* Example risk assessment */
46 static const attack_risk_t pqc_risks[] = {
47     {ATTACK_POWER_ANALYSIS, 8, 9, 7,
48      "Constant-time implementation, masking"},
49     {ATTACK_TIMING, 9, 8, 5,
50      "Constant-time code, no secret-dependent branches"},
51     {ATTACK_RNG_WEAKNESS, 6, 10, 8,
52      "Hardware TRNG, NIST SP 800-90A DRBG"},
53     {ATTACK_HYBRID_DOWNGRADE, 5, 9, 4,
54      "Strict hybrid enforcement, no fallback"},
55     {ATTACK_IMPLEMENTATION, 7, 10, 6,
56      "Use certified libraries, formal verification"}
57 };
```

Listing 9.1: Attack Surface Enumeration for PQC-IoT Device

9.1.4 Threat Actor Profiling

Understanding adversary capabilities is essential for appropriate countermeasure selection:

Table 9.2: Threat Actor Capabilities for PQC-IoT Attacks

Actor Type	Quantum Access	Side-Channel	Resources	Persistence
Script Kiddie	None	Basic timing	Low	Low
Cybercriminal	None (future)	Commercial tools	Medium	Medium
Competitor	Cloud (future)	Lab equipment	Medium-High	High
Nation State	Own CRQC	Advanced lab	Unlimited	Very High
Insider	Indirect	Physical access	Variable	Variable

9.2 Side-Channel Vulnerability Analysis

9.2.1 Algorithm-Specific Vulnerabilities

Each PQC algorithm family presents distinct side-channel characteristics:

ML-KEM Side-Channel Profile

```
1 /* CRITICAL: Decapsulation comparison must be constant-time
     */
2
```

Table 9.3: ML-KEM Side-Channel Vulnerability Assessment

Operation	Vulnerability	Severity	Mitigation
NTT butterfly	Timing	Medium	Constant-time multiplication
Modular reduction	Power (DPA)	High	Masking, shuffling
Polynomial sampling	Timing	High	Constant-time rejection sampling
Comparison (decaps)	Timing	Critical	Constant-time comparison
Secret key access	Cache	High	Memory access patterns masking
SHAKE operations	Power	Medium	Masked Keccak implementation

```
#include <stdint.h>
#include <stddef.h>

/*
 * Constant-time byte array comparison
 * Returns 0 if equal, non-zero otherwise
 * Execution time independent of data values
 */
static int ct_memcmp(const uint8_t *a, const uint8_t *b,
   size_t len) {
    volatile uint8_t result = 0;

    for (size_t i = 0; i < len; i++) {
        result |= a[i] ^ b[i];
    }

    /* Convert to 0 or 1 in constant time */
    result = (result - 1) >> 8;  /* 0xFF if zero, 0x00
       otherwise */
    return (int)(result & 1) ^ 1;
}

/*
 * Constant-time conditional select
 * Returns a if selector == 0, b if selector == 1
 */
static void ct_select(uint8_t *out,
                      const uint8_t *a,
                      const uint8_t *b,
```

```
30                          size_t len,
31                          uint8_t selector) {
32      /* Expand selector to full byte mask */
33      uint8_t mask = (uint8_t)(-(int8_t)selector);  /* 0xFF or
           0x00 */
34
35      for (size_t i = 0; i < len; i++) {
36          out[i] = a[i] ^ (mask & (a[i] ^ b[i]));
37      }
38  }
39
40  /*
41   * ML-KEM decapsulation with side-channel protection
42   * Implements implicit rejection per FIPS 203
43   */
44  int mlkem_decaps_protected(uint8_t *shared_secret,
45                             const uint8_t *ciphertext,
46                             const uint8_t *secret_key) {
47      uint8_t expected_ct[MLKEM_CIPHERTEXT_BYTES];
48      uint8_t ss_real[MLKEM_SHARED_SECRET_BYTES];
49      uint8_t ss_fake[MLKEM_SHARED_SECRET_BYTES];
50      uint8_t decrypted_msg[MLKEM_MSG_BYTES];
51
52      /* Step 1: Decrypt ciphertext to recover message */
53      mlkem_decrypt_internal(decrypted_msg, ciphertext,
           secret_key);
54
55      /* Step 2: Re-encrypt to get expected ciphertext */
56      mlkem_encrypt_internal(expected_ct, decrypted_msg,
57                             secret_key +
                                  MLKEM_SECRET_KEY_OFFSET);
58
59      /* Step 3: Constant-time comparison */
60      int cmp_result = ct_memcmp(ciphertext, expected_ct,
61                                 MLKEM_CIPHERTEXT_BYTES);
62
63      /* Step 4: Compute both possible shared secrets */
64      /* Real shared secret from decrypted message */
65      shake256(ss_real, MLKEM_SHARED_SECRET_BYTES,
66               decrypted_msg, MLKEM_MSG_BYTES);
67
```

```
68      /* Fake shared secret from implicit rejection value */
69      shake256(ss_fake, MLKEM_SHARED_SECRET_BYTES,
70               secret_key + MLKEM_Z_OFFSET, MLKEM_Z_BYTES);
71
72      /* Step 5: Constant-time select based on comparison */
73      ct_select(shared_secret, ss_real, ss_fake,
74                MLKEM_SHARED_SECRET_BYTES,
                     (uint8_t)cmp_result);
75
76      /* Clear sensitive data */
77      secure_zero(decrypted_msg, sizeof(decrypted_msg));
78      secure_zero(ss_real, sizeof(ss_real));
79      secure_zero(ss_fake, sizeof(ss_fake));
80      secure_zero(expected_ct, sizeof(expected_ct));
81
82      return 0;  /* Always return success (implicit rejection)
           */
83  }
```

Listing 9.2: Constant-Time Comparison for ML-KEM Decapsulation

ML-DSA Side-Channel Profile

ML-DSA's rejection sampling during signature generation creates significant timing vulnerabilities:

```
1  /*
2   * ML-DSA VULNERABLE rejection sampling (DO NOT USE)
3   * Variable number of iterations leaks information about
       secret key
4   */
5  int mldsa_sign_VULNERABLE(uint8_t *sig, const uint8_t *msg,
6                            size_t msg_len, const uint8_t *sk)
                                {
7      poly_vec y, w, z;
8      uint16_t nonce = 0;
9
10     while (1) {
11         /* Sample y from uniform distribution */
12         expand_mask(&y, sk, nonce++);
13
14         /* Compute w = A*y */
15         matrix_vector_mul(&w, &A, &y);
```

```
16
17          /* Compute challenge c = H(w, msg) */
18          challenge_hash(&c, &w, msg, msg_len);
19
20          /* Compute z = y + c*s1 */
21          poly_vec_mul_add(&z, &c, &s1, &y);
22
23          /* TIMING LEAK: Different messages require different
24           * numbers of iterations, leaking info about s1 */
25          if (check_norm(&z) && check_hints(&h)) {
26              break;  /* Variable timing! */
27          }
28      }
29
30      pack_signature(sig, &z, &h, &c);
31      return 0;
32  }
33
34  /*
35   * Side-channel resistant approach: Deterministic timing
36   * Always perform fixed number of iterations
37   */
38  int mldsa_sign_protected(uint8_t *sig, const uint8_t *msg,
39                           size_t msg_len, const uint8_t *sk) {
40      poly_vec y, w, z;
41      poly_vec z_candidates[MAX_SIGN_ITERATIONS];
42      uint8_t valid[MAX_SIGN_ITERATIONS];
43      uint16_t nonce = 0;
44
45      /* Always perform MAX_SIGN_ITERATIONS attempts */
46      for (int i = 0; i < MAX_SIGN_ITERATIONS; i++) {
47          expand_mask(&y, sk, nonce++);
48          matrix_vector_mul(&w, &A, &y);
49          challenge_hash(&c, &w, msg, msg_len);
50          poly_vec_mul_add(&z_candidates[i], &c, &s1, &y);
51
52          /* Check validity but don't branch */
53          valid[i] = (check_norm(&z_candidates[i]) &&
54                      check_hints(&h)) ? 1 : 0;
55      }
56
```

```
57      /* Constant-time selection of first valid signature */
58      int selected = ct_select_first_valid(valid,
           MAX_SIGN_ITERATIONS);
59
60      if (selected < 0) {
61          return -1;  /* Extremely rare: no valid signature
               found */
62      }
63
64      /* Copy selected signature in constant time */
65      ct_copy_signature(sig, z_candidates, selected,
           MAX_SIGN_ITERATIONS);
66
67      /* Clear all candidates */
68      secure_zero(z_candidates, sizeof(z_candidates));
69
70      return 0;
71  }
```

Listing 9.3: ML-DSA Timing Attack and Mitigation

Falcon Side-Channel Profile

Falcon's use of floating-point arithmetic in its sampler creates unique challenges:

Table 9.4: Falcon Floating-Point Side-Channel Risks

Component	Risk Level	Details
Fast Fourier Sampling	Critical	FPU operations have data-dependent timing on most platforms
Gaussian sampler	High	Rejection sampling with variable iterations
Tree traversal	Medium	Access patterns may leak secret structure
Integer conversion	Medium	Float-to-int conversion timing varies

```
1  /*
2   * Falcon requires constant-time Gaussian sampling
3   * This is a simplified illustration of the approach
4   */
5
```

```
#include <stdint.h>
#include <math.h>

/* Precomputed cumulative distribution table */
static const uint64_t gaussian_cdf[256] = {
    /* Values for sigma = 1.17*sqrt(q/2n) */
    0x0000000000000000ULL, 0x00000000027A3B39ULL,
    /* ... (full table omitted for brevity) */
};

/*
 * Constant-time Gaussian sampler using CDT
 * Avoids rejection sampling timing leaks
 */
static int32_t sample_gaussian_ct(uint64_t random_bits) {
    int32_t result = 0;
    uint64_t mask;

    /* Compare against all table entries in constant time */
    for (int i = 0; i < 256; i++) {
        /* mask = 0xFFFFFFFF if random_bits >=
           gaussian_cdf[i] */
        mask = (uint64_t)(-(int64_t)(random_bits >=
           gaussian_cdf[i]));

        /* Increment result if condition met */
        result += (int32_t)(mask & 1);
    }

    /* Apply sign bit from additional random bit */
    int32_t sign = (int32_t)((random_bits >> 63) & 1);
    sign = 1 - 2 * sign;  /* Convert 0/1 to +1/-1 */

    return result * sign;
}

/*
 * Fixed-point arithmetic alternative for constrained devices
 * Avoids floating-point timing variations entirely
 */
typedef int64_t fxp_t;  /* Q32.32 fixed point */
```

```
45
46 #define FXP_ONE ((fxp_t)1 << 32)
47 #define FXP_HALF ((fxp_t)1 << 31)
48
49 static fxp_t fxp_mul(fxp_t a, fxp_t b) {
50     /* Constant-time 64-bit multiplication */
51     __int128 temp = (__int128)a * b;
52     return (fxp_t)(temp >> 32);
53 }
54
55 static fxp_t fxp_exp_neg(fxp_t x) {
56     /* Polynomial approximation of exp(-x) for x >= 0 */
57     /* Coefficients scaled for Q32.32 format */
58     static const fxp_t coeffs[] = {
59         FXP_ONE,                    /* 1 */
60         -FXP_ONE,                   /* -x */
61         FXP_HALF,                   /* x^2/2 */
62         -0x2AAAAAABLL,              /* -x^3/6 */
63         0x0AAAAAABLL,               /* x^4/24 */
64     };
65
66     fxp_t result = coeffs[4];
67     for (int i = 3; i >= 0; i--) {
68         result = fxp_mul(result, x) + coeffs[i];
69     }
70
71     return result;
72 }
```

Listing 9.4: Falcon Constant-Time Gaussian Sampling (Simplified)

9.2.2 Power Analysis Countermeasures

Masking Techniques

Boolean and arithmetic masking protect against differential power analysis:

```
1 /*
2  * First-order Boolean masking for polynomial coefficients
3  * Each coefficient x is split into shares: x = x1 XOR x2
4  */
5
6 #include <stdint.h>
```

```
7
8  typedef struct {
9      int16_t share1[256];
10     int16_t share2[256];
11 } masked_poly_t;
12
13 /* Generate random mask */
14 static void generate_mask(int16_t *mask, size_t len) {
15     /* Use hardware RNG or DRBG */
16     for (size_t i = 0; i < len; i++) {
17         mask[i] = (int16_t)(get_random_uint32() & 0xFFFF);
18     }
19 }
20
21 /* Boolean-to-arithmetic mask conversion */
22 static int16_t b2a_convert(int16_t x_bool, int16_t r_bool,
23                            int16_t r_arith, int16_t q) {
24     /* Convert (x XOR r_bool) to ((x - r_arith) mod q) */
25     /* This is a simplified version; full implementation is
           complex */
26     int16_t gamma = get_random_int16() % q;
27     int16_t t = (x_bool ^ r_bool) - r_arith + gamma;
28     return ((t % q) + q) % q;
29 }
30
31 /* Masked NTT butterfly operation */
32 static void masked_butterfly(masked_poly_t *p, int i, int j,
33                              int16_t zeta, int16_t q) {
34     /* Process both shares independently */
35     /* Share 1 */
36     int16_t t1 = montgomery_mul(zeta, p->share1[j], q);
37     p->share1[j] = barrett_reduce(p->share1[i] - t1, q);
38     p->share1[i] = barrett_reduce(p->share1[i] + t1, q);
39
40     /* Share 2 - with independent randomization */
41     int16_t t2 = montgomery_mul(zeta, p->share2[j], q);
42     p->share2[j] = barrett_reduce(p->share2[i] - t2, q);
43     p->share2[i] = barrett_reduce(p->share2[i] + t2, q);
44
45     /* Refresh masks periodically to prevent higher-order
           attacks */
```

```
    if ((i + j) % 16 == 0) {
        refresh_masks(p, i, j, q);
    }
}

/* Mask refreshing to prevent accumulation attacks */
static void refresh_masks(masked_poly_t *p, int i, int j,
   int16_t q) {
    int16_t r1 = get_random_int16() % q;
    int16_t r2 = get_random_int16() % q;

    p->share1[i] = barrett_reduce(p->share1[i] + r1, q);
    p->share2[i] = barrett_reduce(p->share2[i] - r1, q);

    p->share1[j] = barrett_reduce(p->share1[j] + r2, q);
    p->share2[j] = barrett_reduce(p->share2[j] - r2, q);
}

/* Unmask at the end of computation */
static void unmask_poly(int16_t *out, const masked_poly_t *p,
                        size_t len, int16_t q) {
    for (size_t i = 0; i < len; i++) {
        out[i] = barrett_reduce(p->share1[i] + p->share2[i],
           q);
    }
}
```

Listing 9.5: First-Order Masking for NTT Operations

Shuffling Countermeasures

Randomizing operation order prevents localized power analysis:

```
/*
 * Shuffled NTT: Randomize butterfly order within each level
 * Prevents adversary from locating specific coefficient
    operations
 */

#include <stdint.h>
#include <stdlib.h>

/* Fisher-Yates shuffle for index permutation */
```

```
static void shuffle_indices(uint16_t *indices, size_t n) {
    for (size_t i = n - 1; i > 0; i--) {
        size_t j = get_random_uint32() % (i + 1);
        /* Swap */
        uint16_t temp = indices[i];
        indices[i] = indices[j];
        indices[j] = temp;
    }
}

/* Shuffled NTT implementation */
void ntt_shuffled(int16_t *poly, const int16_t *zetas,
   int16_t q) {
    uint16_t indices[128];  /* For n=256, we have 128
       butterflies per level */

    int k = 1;
    for (int len = 128; len >= 2; len >>= 1) {
        int num_butterflies = 256 / (2 * len);

        /* Generate shuffled indices for this level */
        for (int i = 0; i < num_butterflies * len; i++) {
            indices[i] = i;
        }
        shuffle_indices(indices, num_butterflies * len);

        /* Perform butterflies in shuffled order */
        for (int idx = 0; idx < num_butterflies * len;
           idx++) {
            int shuffled_idx = indices[idx];
            int start = (shuffled_idx / len) * 2 * len;
            int offset = shuffled_idx % len;

            int i = start + offset;
            int j = i + len;
            int zeta_idx = k + (shuffled_idx / len);

            int16_t t = montgomery_mul(zetas[zeta_idx],
               poly[j], q);
            poly[j] = barrett_reduce(poly[i] - t, q);
            poly[i] = barrett_reduce(poly[i] + t, q);
```

```
47          }
48
49          k += num_butterflies;
50      }
51
52      /* Clear indices */
53      secure_zero(indices, sizeof(indices));
54  }
```

Listing 9.6: Shuffled NTT Implementation

9.2.3 Fault Injection Countermeasures

Fault attacks can bypass cryptographic protections by inducing computational errors:

```
1  /*
2   * Double computation with comparison for fault detection
3   * Critical operations are performed twice and compared
4   */
5
6  #include <stdint.h>
7  #include <stdbool.h>
8
9  typedef struct {
10     bool fault_detected;
11     uint32_t fault_count;
12     void (*fault_handler)(void);
13 } fault_monitor_t;
14
15 static fault_monitor_t g_fault_monitor = {
16     .fault_detected = false,
17     .fault_count = 0,
18     .fault_handler = NULL
19 };
20
21 /* Register fault handler */
22 void register_fault_handler(void (*handler)(void)) {
23     g_fault_monitor.fault_handler = handler;
24 }
25
26 /* Check computation result with redundancy */
27 static bool verify_computation(const uint8_t *result1,
28                                const uint8_t *result2,
```

```
                                    size_t len) {
    volatile uint8_t diff = 0;

    for (size_t i = 0; i < len; i++) {
        diff |= result1[i] ^ result2[i];
    }

    if (diff != 0) {
        g_fault_monitor.fault_detected = true;
        g_fault_monitor.fault_count++;

        if (g_fault_monitor.fault_handler) {
            g_fault_monitor.fault_handler();
        }

        return false;
    }

    return true;
}

/*
 * Fault-protected ML-KEM decapsulation
 */
int mlkem_decaps_fault_protected(uint8_t *ss,
                                 const uint8_t *ct,
                                 const uint8_t *sk) {
    uint8_t ss1[32], ss2[32];

    /* First computation */
    mlkem_decaps_internal(ss1, ct, sk);

    /* Introduce random delay to desynchronize fault timing
       */
    random_delay(100, 1000);  /* 100-1000 cycles */

    /* Second computation with different register allocation
       */
    mlkem_decaps_internal_variant(ss2, ct, sk);

    /* Compare results */
```

```
68      if (!verify_computation(ss1, ss2, 32)) {
69          /* Fault detected - zeroize and return error */
70          secure_zero(ss, 32);
71          secure_zero(ss1, sizeof(ss1));
72          secure_zero(ss2, sizeof(ss2));
73          return -1;
74      }
75
76      /* Copy result and clean up */
77      memcpy(ss, ss1, 32);
78      secure_zero(ss1, sizeof(ss1));
79      secure_zero(ss2, sizeof(ss2));
80
81      return 0;
82  }
83
84  /*
85   * Algorithm-level integrity check using mathematical
        properties
86   * For NTT: verify NTT(INTT(x)) == x
87   */
88  bool verify_ntt_integrity(const int16_t *original,
89                            const int16_t *transformed,
90                            size_t n, int16_t q) {
91      int16_t verify[256];
92
93      /* Apply inverse transform to transformed data */
94      memcpy(verify, transformed, n * sizeof(int16_t));
95      intt(verify, q);
96
97      /* Compare with original */
98      volatile int16_t diff = 0;
99      for (size_t i = 0; i < n; i++) {
100         diff |= original[i] ^ verify[i];
101     }
102
103     secure_zero(verify, sizeof(verify));
104     return (diff == 0);
105 }
106
107 /*
```

```
108  * Signature verification with fault protection
109  * Verify signature twice with different code paths
110  */
111 int verify_signature_fault_protected(const uint8_t *msg,
       size_t msg_len,
112                                      const uint8_t *sig,
                                            size_t sig_len,
113                                      const uint8_t *pk) {
114     volatile int result1, result2;
115
116     /* First verification */
117     result1 = mldsa_verify(msg, msg_len, sig, sig_len, pk);
118
119     /* Random delay */
120     random_delay(50, 500);
121
122     /* Second verification with reordered operations */
123     result2 = mldsa_verify_variant(msg, msg_len, sig,
           sig_len, pk);
124
125     /* Both must agree */
126     if (result1 != result2) {
127         g_fault_monitor.fault_detected = true;
128         g_fault_monitor.fault_count++;
129         return -1;  /* Fault detected */
130     }
131
132     return result1;
133 }
```

Listing 9.7: Fault Detection for ML-KEM Operations

9.3 Security Parameter Selection

9.3.1 NIST Security Levels Explained

NIST defines five security levels based on the computational effort required to break the scheme:

9.3.2 Parameter Selection by Application Domain

Table 9.5: NIST Security Levels and Equivalent Strength

Level	Classical Equivalent	Quantum Equivalent	Reference
1	AES-128 key search	Grover on AES-128	2^{128} classical
2	SHA-256 collision	Grover on SHA-256	2^{128} quantum
3	AES-192 key search	Grover on AES-192	2^{192} classical
4	SHA-384 collision	Grover on SHA-384	2^{192} quantum
5	AES-256 key search	Grover on AES-256	2^{256} classical

Table 9.6: Recommended Security Levels by IoT Application

Application	Data Lifetime	Min. Level	Recommended
Consumer sensors	1-3 years	1	ML-KEM-512
Smart home	5-10 years	1-3	ML-KEM-768
Industrial IoT	10-20 years	3	ML-KEM-768/1024
Medical devices	25+ years	3-5	ML-KEM-1024
Critical infrastructure	30+ years	5	ML-KEM-1024 + backup
Government/Defense	50+ years	5	ML-KEM-1024, ML-DSA-87

```
#!/usr/bin/env python3
"""
PQC Security Level Selection Tool
Helps select appropriate parameters based on requirements
"""

from dataclasses import dataclass
from enum import Enum
from typing import Optional, List

class SecurityLevel(Enum):
    LEVEL_1 = 1  # ~AES-128
    LEVEL_2 = 2  # ~SHA-256 collision
    LEVEL_3 = 3  # ~AES-192
    LEVEL_4 = 4  # ~SHA-384 collision
    LEVEL_5 = 5  # ~AES-256

@dataclass
class DeviceConstraints:
    ram_kb: int
    flash_kb: int
    cpu_mhz: int
    has_hw_aes: bool = False
```

```
    has_hw_sha: bool = False
    battery_powered: bool = False

@dataclass
class SecurityRequirements:
    data_sensitivity: str  # 'low', 'medium', 'high',
       'critical'
    data_lifetime_years: int
    threat_actors: List[str]  # 'script_kiddie', 'criminal',
       'nation_state'
    compliance: List[str]  # 'none', 'hipaa', 'pci', 'cnsa',
       'fips'

# Algorithm parameters database
MLKEM_PARAMS = {
    SecurityLevel.LEVEL_1: {
        'name': 'ML-KEM-512',
        'pk_bytes': 800,
        'sk_bytes': 1632,
        'ct_bytes': 768,
        'ram_required_kb': 8,
        'flash_required_kb': 12,
        'keygen_cycles_m4': 434_000,
        'encaps_cycles_m4': 486_000,
        'decaps_cycles_m4': 524_000,
    },
    SecurityLevel.LEVEL_3: {
        'name': 'ML-KEM-768',
        'pk_bytes': 1184,
        'sk_bytes': 2400,
        'ct_bytes': 1088,
        'ram_required_kb': 12,
        'flash_required_kb': 16,
        'keygen_cycles_m4': 720_000,
        'encaps_cycles_m4': 817_000,
        'decaps_cycles_m4': 879_000,
    },
    SecurityLevel.LEVEL_5: {
        'name': 'ML-KEM-1024',
        'pk_bytes': 1568,
        'sk_bytes': 3168,
```

```
        'ct_bytes': 1568,
        'ram_required_kb': 16,
        'flash_required_kb': 20,
        'keygen_cycles_m4': 1_105_000,
        'encaps_cycles_m4': 1_241_000,
        'decaps_cycles_m4': 1_337_000,
    },
}

MLDSA_PARAMS = {
    SecurityLevel.LEVEL_2: {
        'name': 'ML-DSA-44',
        'pk_bytes': 1312,
        'sk_bytes': 2560,
        'sig_bytes': 2420,
        'ram_required_kb': 32,
        'flash_required_kb': 24,
    },
    SecurityLevel.LEVEL_3: {
        'name': 'ML-DSA-65',
        'pk_bytes': 1952,
        'sk_bytes': 4032,
        'sig_bytes': 3293,
        'ram_required_kb': 48,
        'flash_required_kb': 32,
    },
    SecurityLevel.LEVEL_5: {
        'name': 'ML-DSA-87',
        'pk_bytes': 2592,
        'sk_bytes': 4896,
        'sig_bytes': 4595,
        'ram_required_kb': 64,
        'flash_required_kb': 40,
    },
}

def determine_minimum_level(requirements:
    SecurityRequirements) -> SecurityLevel:
    """Determine minimum required security level from
       requirements."""
    level = SecurityLevel.LEVEL_1
```

```
101
102     # Data sensitivity
103     sensitivity_map = {
104         'low': SecurityLevel.LEVEL_1,
105         'medium': SecurityLevel.LEVEL_3,
106         'high': SecurityLevel.LEVEL_3,
107         'critical': SecurityLevel.LEVEL_5,
108     }
109     level = max(level, sensitivity_map.get(
110         requirements.data_sensitivity,
                SecurityLevel.LEVEL_1),
111         key=lambda x: x.value)
112
113     # Data lifetime (account for HNDL threat)
114     if requirements.data_lifetime_years > 25:
115         level = max(level, SecurityLevel.LEVEL_5, key=lambda
                x: x.value)
116     elif requirements.data_lifetime_years > 15:
117         level = max(level, SecurityLevel.LEVEL_3, key=lambda
                x: x.value)
118     elif requirements.data_lifetime_years > 5:
119         level = max(level, SecurityLevel.LEVEL_3, key=lambda
                x: x.value)
120
121     # Threat actors
122     if 'nation_state' in requirements.threat_actors:
123         level = max(level, SecurityLevel.LEVEL_5, key=lambda
                x: x.value)
124     elif 'criminal' in requirements.threat_actors:
125         level = max(level, SecurityLevel.LEVEL_3, key=lambda
                x: x.value)
126
127     # Compliance requirements
128     if 'cnsa' in requirements.compliance:
129         level = SecurityLevel.LEVEL_5  # CNSA 2.0 requires
                Level 5
130     elif 'fips' in requirements.compliance:
131         level = max(level, SecurityLevel.LEVEL_3, key=lambda
                x: x.value)
132
133     return level
```

```
134
135 def check_device_compatibility(level: SecurityLevel,
136                                 constraints:
                                       DeviceConstraints,
137                                 algorithm: str = 'mlkem') ->
                                       dict:
138     """Check if device can support the required security
           level."""
139
140     params = MLKEM_PARAMS if algorithm == 'mlkem' else
           MLDSA_PARAMS
141
142     if level not in params:
143         # Find closest available level
144         available = sorted(params.keys(), key=lambda x:
               x.value)
145         level = min(available, key=lambda x: abs(x.value -
               level.value))
146
147     algo_params = params[level]
148
149     result = {
150         'compatible': True,
151         'algorithm': algo_params['name'],
152         'issues': [],
153         'recommendations': [],
154     }
155
156     # Check RAM
157     if constraints.ram_kb < algo_params['ram_required_kb']:
158         result['compatible'] = False
159         result['issues'].append(
160             f"Insufficient RAM: {constraints.ram_kb}KB < "
161             f"{algo_params['ram_required_kb']}KB required"
162         )
163         result['recommendations'].append(
164             "Consider stack optimization or lower security
                   level"
165         )
166
167     # Check Flash
```

```
168     if constraints.flash_kb <
            algo_params['flash_required_kb']:
169         result['compatible'] = False
170         result['issues'].append(
171             f"Insufficient Flash: {constraints.flash_kb}KB <
                    "
172             f"{algo_params['flash_required_kb']}KB required"
173         )
174
175     # Performance estimate
176     if algorithm == 'mlkem' and 'keygen_cycles_m4' in
            algo_params:
177         cycles = algo_params['decaps_cycles_m4']
178         time_ms = (cycles / (constraints.cpu_mhz * 1000))
179
180         if time_ms > 1000:  # More than 1 second
181             result['issues'].append(
182                 f"Slow performance: ~{time_ms:.0f}ms per
                        operation"
183             )
184             if constraints.battery_powered:
185                 result['recommendations'].append(
186                     "High energy consumption - consider
                            caching keys"
187                 )
188
189     return result
190
191 def select_parameters(requirements: SecurityRequirements,
192                       constraints: DeviceConstraints) ->
                              dict:
193     """Main parameter selection function."""
194
195     min_level = determine_minimum_level(requirements)
196
197     # Try to meet minimum level
198     kem_result = check_device_compatibility(min_level,
            constraints, 'mlkem')
199     dsa_result = check_device_compatibility(min_level,
            constraints, 'mldsa')
200
```

```
    # If not compatible, find maximum achievable level
    if not kem_result['compatible'] or not
       dsa_result['compatible']:
        for level in sorted(SecurityLevel, key=lambda x:
           x.value, reverse=True):
            kem_check = check_device_compatibility(level,
               constraints, 'mlkem')
            dsa_check = check_device_compatibility(level,
               constraints, 'mldsa')

            if kem_check['compatible'] and
               dsa_check['compatible']:
                return {
                    'status': 'degraded',
                    'requested_level': min_level.value,
                    'achievable_level': level.value,
                    'kem': kem_check,
                    'dsa': dsa_check,
                    'warning': (
                        f"Device constraints limit security
                           to Level {level.value}. "
                        f"Requested Level {min_level.value}
                           not achievable."
                    ),
                }

        return {
            'status': 'incompatible',
            'error': 'No PQC configuration fits device
               constraints',
            'kem_issues': kem_result['issues'],
            'dsa_issues': dsa_result['issues'],
        }

    return {
        'status': 'ok',
        'security_level': min_level.value,
        'kem': kem_result,
        'dsa': dsa_result,
    }

```

```
# Example usage
if __name__ == '__main__':
    # Medical device example
    requirements = SecurityRequirements(
        data_sensitivity='critical',
        data_lifetime_years=25,
        threat_actors=['criminal', 'nation_state'],
        compliance=['hipaa', 'fips'],
    )

    constraints = DeviceConstraints(
        ram_kb=64,
        flash_kb=256,
        cpu_mhz=120,
        has_hw_aes=True,
        has_hw_sha=True,
        battery_powered=True,
    )

    result = select_parameters(requirements, constraints)

    print("=== PQC Parameter Selection Result ===")
    print(f"Status: {result['status']}")
    if result['status'] == 'ok':
        print(f"Security Level: {result['security_level']}")
        print(f"KEM: {result['kem']['algorithm']}")
        print(f"DSA: {result['dsa']['algorithm']}")
    else:
        print(f"Warning: {result.get('warning',
           result.get('error'))}")
```

Listing 9.8: Security Level Selection Tool

9.3.3 Conservative vs. Aggressive Parameter Choices

The cryptographic community debates appropriate security margins:

Warning: Recent lattice cryptanalysis improvements (2023-2024) have eroded some security margins. While NIST parameters remain secure, the trend suggests conservative choices are prudent for long-lived deployments.

Table 9.7: Parameter Selection Philosophy Comparison

Aspect	Conservative	Aggressive
Security margin	+20-30% above estimates	Minimal margin
Level choice	Always Level 5	Minimum required
Rationale	Unknown quantum advances, cryptanalysis improvements	Resource efficiency, measured risk
Best for	Government, critical infrastructure, long-term secrets	Consumer IoT, short-lived data
Risk	Over-provisioning, resource waste	Insufficient security if estimates wrong

9.4 Key Management Best Practices

Effective key management is critical for PQC deployments, particularly given the larger key sizes and new operational considerations introduced by post-quantum algorithms.

9.4.1 Key Lifecycle Management

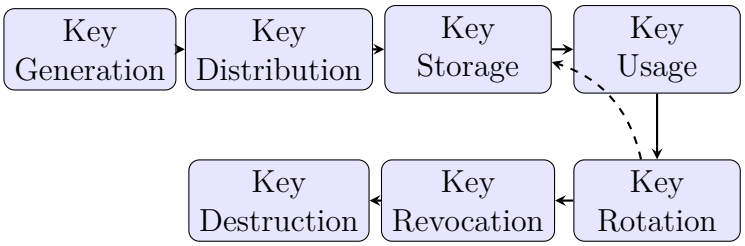


Figure 9.2: PQC Key Lifecycle Stages

9.4.2 Key Generation Requirements

PQC key generation demands high-quality randomness and secure execution environments:

```
/*
 * Secure Key Generation Framework for IoT PQC
 * Implements defense-in-depth for key generation
 */

#include <stdint.h>
#include <stdbool.h>
#include <string.h>

/* Key generation context */
```

```
typedef struct {
    uint8_t entropy_pool[64];
    uint32_t entropy_bits;
    bool hw_rng_available;
    bool health_check_passed;
    uint32_t generation_counter;
} keygen_context_t;

/* Entropy source types */
typedef enum {
    ENTROPY_HW_TRNG,      /* Hardware TRNG */
    ENTROPY_ADC_NOISE,    /* ADC noise sampling */
    ENTROPY_TIMER_JITTER, /* Timer jitter */
    ENTROPY_RADIO_RSSI,   /* Radio signal noise */
    ENTROPY_EXTERNAL      /* External entropy injection */
} entropy_source_t;

/* Health check for entropy source */
static bool entropy_health_check(const uint8_t *samples,
   size_t len) {
    /* Repetition count test (NIST SP 800-90B) */
    uint8_t last = samples[0];
    int repeat_count = 1;
    int max_repeat = 1;

    for (size_t i = 1; i < len; i++) {
        if (samples[i] == last) {
            repeat_count++;
            if (repeat_count > max_repeat) {
                max_repeat = repeat_count;
            }
        } else {
            last = samples[i];
            repeat_count = 1;
        }
    }

    /* Fail if too many repetitions (threshold depends on
       source) */
    if (max_repeat > 6) {
        return false;
```

```
    }

    /* Adaptive proportion test */
    int counts[256] = {0};
    for (size_t i = 0; i < len; i++) {
        counts[samples[i]]++;
    }

    int max_count = 0;
    for (int i = 0; i < 256; i++) {
        if (counts[i] > max_count) {
            max_count = counts[i];
        }
    }

    /* Fail if any value appears too frequently */
    if (max_count > (int)(len * 0.15)) {  /* >15% is
       suspicious */
        return false;
    }

    return true;
}

/* Gather entropy from multiple sources */
static int gather_entropy(keygen_context_t *ctx) {
    uint8_t raw_entropy[256];
    size_t offset = 0;

    /* Source 1: Hardware TRNG if available */
    if (ctx->hw_rng_available) {
        hw_trng_read(raw_entropy + offset, 64);
        offset += 64;
    }

    /* Source 2: ADC noise */
    for (int i = 0; i < 32; i++) {
        raw_entropy[offset++] = adc_read_noise() & 0xFF;
    }

    /* Source 3: Timer jitter */
```

```
90      for (int i = 0; i < 32; i++) {
91          uint32_t t1 = timer_get_cycles();
92          volatile int dummy = 0;
93          for (int j = 0; j < 100; j++) dummy++;
94          uint32_t t2 = timer_get_cycles();
95          raw_entropy[offset++] = (t2 - t1) & 0xFF;
96      }
97
98      /* Health check on raw entropy */
99      if (!entropy_health_check(raw_entropy, offset)) {
100         return -1;  /* Entropy source failure */
101     }
102
103     /* Condition entropy using SHAKE256 */
104     shake256(ctx->entropy_pool, sizeof(ctx->entropy_pool),
105              raw_entropy, offset);
106
107     ctx->entropy_bits = 256;  /* Conservative estimate */
108     ctx->health_check_passed = true;
109
110     secure_zero(raw_entropy, sizeof(raw_entropy));
111     return 0;
112 }
113
114 /* Generate ML-KEM keypair with full security measures */
115 int secure_mlkem_keygen(uint8_t *pk, uint8_t *sk,
116                         keygen_context_t *ctx,
117                         int security_level) {
118     int ret = -1;
119
120     /* Step 1: Verify entropy health */
121     if (!ctx->health_check_passed) {
122         if (gather_entropy(ctx) != 0) {
123             return -1;  /* Cannot generate keys without good
                               entropy */
124         }
125     }
126
127     /* Step 2: Derive seed from entropy pool */
128     uint8_t seed[64];
129     uint8_t counter_bytes[4];
```

```
130
131     counter_bytes[0] = (ctx->generation_counter >> 24) &
           0xFF;
132     counter_bytes[1] = (ctx->generation_counter >> 16) &
           0xFF;
133     counter_bytes[2] = (ctx->generation_counter >> 8) & 0xFF;
134     counter_bytes[3] = ctx->generation_counter & 0xFF;
135
136     /* seed = SHAKE256(entropy_pool || counter) */
137     uint8_t hash_input[68];
138     memcpy(hash_input, ctx->entropy_pool, 64);
139     memcpy(hash_input + 64, counter_bytes, 4);
140     shake256(seed, sizeof(seed), hash_input,
           sizeof(hash_input));
141
142     /* Step 3: Generate keypair */
143     switch (security_level) {
144         case 1:
145             ret = mlkem512_keypair_seed(pk, sk, seed);
146             break;
147         case 3:
148             ret = mlkem768_keypair_seed(pk, sk, seed);
149             break;
150         case 5:
151             ret = mlkem1024_keypair_seed(pk, sk, seed);
152             break;
153         default:
154             ret = -1;
155     }
156
157     /* Step 4: Verify keypair (encaps/decaps test) */
158     if (ret == 0) {
159         uint8_t ct_test[MLKEM_MAX_CT_BYTES];
160         uint8_t ss1[32], ss2[32];
161
162         mlkem_encaps(ct_test, ss1, pk, security_level);
163         mlkem_decaps(ss2, ct_test, sk, security_level);
164
165         if (memcmp(ss1, ss2, 32) != 0) {
166             ret = -1;  /* Keypair verification failed */
167         }
```

```
168
169         secure_zero(ct_test, sizeof(ct_test));
170         secure_zero(ss1, sizeof(ss1));
171         secure_zero(ss2, sizeof(ss2));
172     }
173
174     /* Step 5: Update counter and refresh entropy */
175     ctx->generation_counter++;
176     if (ctx->generation_counter % 100 == 0) {
177         gather_entropy(ctx);  /* Periodic entropy refresh */
178     }
179
180     /* Cleanup */
181     secure_zero(seed, sizeof(seed));
182     secure_zero(hash_input, sizeof(hash_input));
183
184     return ret;
185 }
```

Listing 9.9: Secure PQC Key Generation Framework

9.4.3 Secure Key Storage

IoT devices require careful consideration for key storage given physical access threats:

Table 9.8: Key Storage Options for IoT Devices

Storage Type	Security	Cost	PQC Suitability
Plain Flash	Low	$	Not recommended
Encrypted Flash	Medium	$	Acceptable for public keys
Secure Element (SE)	High	$$	Limited by SE memory
Hardware Security Module	Very High	$$$	Best for gateways
ARM TrustZone	Medium-High	$	Good for Cortex-M33+
PUF-derived keys	High	$$	Excellent for IoT

```
1 /*
2  * Physically Unclonable Function (PUF) based key wrapping
3  * Protects PQC secret keys using device-unique secrets
4  */
5
6 #include <stdint.h>
7 #include <string.h>
8
```

```
#define PUF_RESPONSE_SIZE 32
#define KEY_WRAP_OVERHEAD 16  /* AES-GCM tag */

/* PUF interface (hardware-specific) */
extern int puf_generate_response(uint8_t *response,
                                 const uint8_t *challenge,
                                 size_t challenge_len);
extern int puf_get_helper_data(uint8_t *helper_data,
                               size_t *helper_len);
extern int puf_reconstruct(uint8_t *response,
                           const uint8_t *helper_data,
                           size_t helper_len);

/* Key wrapping structure */
typedef struct {
    uint8_t wrapped_key[MLKEM_MAX_SK_BYTES +
       KEY_WRAP_OVERHEAD];
    size_t wrapped_len;
    uint8_t challenge[32];
    uint8_t helper_data[256];
    size_t helper_len;
    uint8_t nonce[12];
} puf_wrapped_key_t;

/* Wrap a PQC secret key using PUF */
int puf_wrap_key(puf_wrapped_key_t *wrapped,
                 const uint8_t *sk, size_t sk_len) {
    uint8_t puf_response[PUF_RESPONSE_SIZE];
    uint8_t kek[32];  /* Key Encryption Key */
    int ret = -1;

    /* Generate random challenge */
    if (get_random_bytes(wrapped->challenge, 32) != 0) {
        return -1;
    }

    /* Get PUF response */
    if (puf_generate_response(puf_response,
       wrapped->challenge, 32) != 0) {
        goto cleanup;
    }
```

```
48
49      /* Get helper data for reconstruction */
50      if (puf_get_helper_data(wrapped->helper_data,
           &wrapped->helper_len) != 0) {
51          goto cleanup;
52      }
53
54      /* Derive KEK from PUF response */
55      shake256(kek, sizeof(kek), puf_response,
           PUF_RESPONSE_SIZE);
56
57      /* Generate nonce for AES-GCM */
58      if (get_random_bytes(wrapped->nonce, 12) != 0) {
59          goto cleanup;
60      }
61
62      /* Wrap key using AES-256-GCM */
63      wrapped->wrapped_len = sk_len + KEY_WRAP_OVERHEAD;
64      if (aes_gcm_encrypt(wrapped->wrapped_key,
65                          sk, sk_len,
66                          NULL, 0,  /* No AAD */
67                          kek,
68                          wrapped->nonce) != 0) {
69          goto cleanup;
70      }
71
72      ret = 0;
73
74  cleanup:
75      secure_zero(puf_response, sizeof(puf_response));
76      secure_zero(kek, sizeof(kek));
77      return ret;
78  }
79
80  /* Unwrap a PQC secret key using PUF */
81  int puf_unwrap_key(uint8_t *sk, size_t *sk_len,
82                     const puf_wrapped_key_t *wrapped) {
83      uint8_t puf_response[PUF_RESPONSE_SIZE];
84      uint8_t kek[32];
85      int ret = -1;
86
```

```
87      /* Reconstruct PUF response using helper data */
88      if (puf_reconstruct(puf_response,
89                          wrapped->helper_data,
90                          wrapped->helper_len) != 0) {
91          return -1;
92      }
93
94      /* Re-derive KEK */
95      shake256(kek, sizeof(kek), puf_response,
            PUF_RESPONSE_SIZE);
96
97      /* Unwrap key */
98      *sk_len = wrapped->wrapped_len - KEY_WRAP_OVERHEAD;
99      if (aes_gcm_decrypt(sk,
100                         wrapped->wrapped_key,
                               wrapped->wrapped_len,
101                         NULL, 0,
102                         kek,
103                         wrapped->nonce) != 0) {
104         /* Authentication failed - key may be tampered */
105         secure_zero(sk, *sk_len);
106         *sk_len = 0;
107         goto cleanup;
108     }
109
110     ret = 0;
111
112 cleanup:
113     secure_zero(puf_response, sizeof(puf_response));
114     secure_zero(kek, sizeof(kek));
115     return ret;
116 }
```

Listing 9.10: PUF-Based Key Protection for PQC

9.4.4 Key Rotation Strategies

PQC key rotation must balance security with the overhead of larger keys:

```
1 /*
2  * Key Rotation Manager for PQC IoT Devices
3  * Handles automated key lifecycle management
4  */
```

```
5
6  #include <stdint.h>
7  #include <stdbool.h>
8  #include <time.h>
9
10 /* Key metadata */
11 typedef struct {
12     uint32_t key_id;
13     uint32_t creation_time;
14     uint32_t expiration_time;
15     uint32_t usage_count;
16     uint32_t max_usage;
17     uint8_t algorithm;       /* MLKEM512=1, MLKEM768=2, etc.
           */
18     uint8_t status;          /* ACTIVE, DEPRECATED, REVOKED
           */
19     uint8_t pk_hash[32];     /* SHA-256 of public key */
20 } key_metadata_t;
21
22 typedef enum {
23     KEY_STATUS_ACTIVE = 0,
24     KEY_STATUS_DEPRECATED = 1,  /* Still valid but being
           phased out */
25     KEY_STATUS_REVOKED = 2,
26     KEY_STATUS_EXPIRED = 3
27 } key_status_t;
28
29 /* Rotation policy */
30 typedef struct {
31     uint32_t max_age_seconds;      /* Maximum key lifetime */
32     uint32_t max_operations;       /* Maximum encaps/decaps
           operations */
33     uint32_t rotation_overlap;     /* Overlap period for
           smooth transition */
34     bool rotate_on_compromise;     /* Immediate rotation if
           compromise detected */
35 } rotation_policy_t;
36
37 /* Default rotation policies by application */
38 static const rotation_policy_t POLICY_CONSUMER = {
39     .max_age_seconds = 365 * 24 * 3600,  /* 1 year */
```

```
40      .max_operations = 1000000,
41      .rotation_overlap = 7 * 24 * 3600,   /* 1 week */
42      .rotate_on_compromise = true
43  };
44
45  static const rotation_policy_t POLICY_INDUSTRIAL = {
46      .max_age_seconds = 90 * 24 * 3600,   /* 90 days */
47      .max_operations = 100000,
48      .rotation_overlap = 24 * 3600,       /* 1 day */
49      .rotate_on_compromise = true
50  };
51
52  static const rotation_policy_t POLICY_CRITICAL = {
53      .max_age_seconds = 30 * 24 * 3600,   /* 30 days */
54      .max_operations = 10000,
55      .rotation_overlap = 4 * 3600,        /* 4 hours */
56      .rotate_on_compromise = true
57  };
58
59  /* Key rotation manager */
60  typedef struct {
61      key_metadata_t current_key;
62      key_metadata_t next_key;
63      rotation_policy_t policy;
64      bool rotation_pending;
65      uint32_t last_check_time;
66  } key_rotation_manager_t;
67
68  /* Check if rotation is needed */
69  bool rotation_needed(key_rotation_manager_t *mgr, uint32_t
        current_time) {
70      key_metadata_t *key = &mgr->current_key;
71
72      /* Check expiration */
73      if (current_time >= key->expiration_time) {
74          return true;
75      }
76
77      /* Check usage count */
78      if (key->usage_count >= mgr->policy.max_operations) {
79          return true;
```

```
80      }
81
82      /* Check age */
83      uint32_t age = current_time - key->creation_time;
84      if (age >= mgr->policy.max_age_seconds) {
85          return true;
86      }
87
88      /* Check if approaching expiration (start overlap
           period) */
89      uint32_t time_remaining = key->expiration_time -
           current_time;
90      if (time_remaining <= mgr->policy.rotation_overlap &&
91          !mgr->rotation_pending) {
92          return true;  /* Start generating next key */
93      }
94
95      return false;
96  }
97
98  /* Initiate key rotation */
99  int initiate_rotation(key_rotation_manager_t *mgr,
100                        keygen_context_t *keygen_ctx,
101                        uint32_t current_time) {
102     if (mgr->rotation_pending) {
103         return 0;  /* Already in progress */
104     }
105
106     /* Generate new keypair */
107     uint8_t new_pk[MLKEM_MAX_PK_BYTES];
108     uint8_t new_sk[MLKEM_MAX_SK_BYTES];
109
110     int level = mgr->current_key.algorithm;
111     if (secure_mlkem_keygen(new_pk, new_sk, keygen_ctx,
           level) != 0) {
112         return -1;
113     }
114
115     /* Store new key (implementation-specific) */
116     uint32_t new_key_id = mgr->current_key.key_id + 1;
117     if (store_keypair(new_key_id, new_pk, new_sk, level) !=
```

```
        0) {
118         secure_zero(new_sk, sizeof(new_sk));
119         return -1;
120     }
121
122     /* Update next key metadata */
123     mgr->next_key.key_id = new_key_id;
124     mgr->next_key.creation_time = current_time;
125     mgr->next_key.expiration_time = current_time +
           mgr->policy.max_age_seconds;
126     mgr->next_key.usage_count = 0;
127     mgr->next_key.max_usage = mgr->policy.max_operations;
128     mgr->next_key.algorithm = level;
129     mgr->next_key.status = KEY_STATUS_ACTIVE;
130     sha256(mgr->next_key.pk_hash, new_pk,
           mlkem_pk_size(level));
131
132     mgr->rotation_pending = true;
133
134     /* Secure cleanup */
135     secure_zero(new_sk, sizeof(new_sk));
136
137     return 0;
138 }
139
140 /* Complete key rotation */
141 int complete_rotation(key_rotation_manager_t *mgr) {
142     if (!mgr->rotation_pending) {
143         return -1;
144     }
145
146     /* Deprecate current key */
147     mgr->current_key.status = KEY_STATUS_DEPRECATED;
148     update_key_status(mgr->current_key.key_id,
           KEY_STATUS_DEPRECATED);
149
150     /* Promote next key to current */
151     memcpy(&mgr->current_key, &mgr->next_key,
           sizeof(key_metadata_t));
152     memset(&mgr->next_key, 0, sizeof(key_metadata_t));
153
```

```
154     mgr->rotation_pending = false;
155
156     /* Schedule deletion of old key after grace period */
157     schedule_key_deletion(mgr->current_key.key_id - 1,
158                           mgr->policy.rotation_overlap);
159
160     return 0;
161 }
162
163 /* Emergency revocation */
164 int emergency_revoke(key_rotation_manager_t *mgr,
165                      keygen_context_t *keygen_ctx,
166                      uint32_t current_time) {
167     /* Immediately revoke current key */
168     mgr->current_key.status = KEY_STATUS_REVOKED;
169     update_key_status(mgr->current_key.key_id,
        KEY_STATUS_REVOKED);
170
171     /* Generate and activate new key immediately */
172     if (initiate_rotation(mgr, keygen_ctx, current_time) !=
        0) {
173         return -1;
174     }
175
176     /* Skip overlap - immediate promotion */
177     return complete_rotation(mgr);
178 }
```

Listing 9.11: Automated Key Rotation Manager

9.4.5 Key Distribution for IoT Networks

Distributing PQC keys across IoT networks presents unique challenges:

9.5 Secure Boot with Post-Quantum Signatures

Secure boot ensures that only authenticated firmware executes on IoT devices. Transitioning to post-quantum signatures introduces new challenges related to signature sizes, verification times, and boot latency requirements.

Table 9.9: Key Distribution Methods for PQC IoT

Method	Advantages	Disadvantages	Use Case
Pre-provisioning	Simple, no runtime overhead	Inflexible, no rotation	Disposable sensors
Online KEM	Dynamic, supports rotation	Requires connectivity	Smart home
Hierarchical	Scalable	Complex PKI	Industrial IoT
Group keys	Efficient multicast	Compromise affects group	Sensor networks
Hybrid (PSK+KEM)	Quantum-safe bootstrap	Key management burden	Critical systems

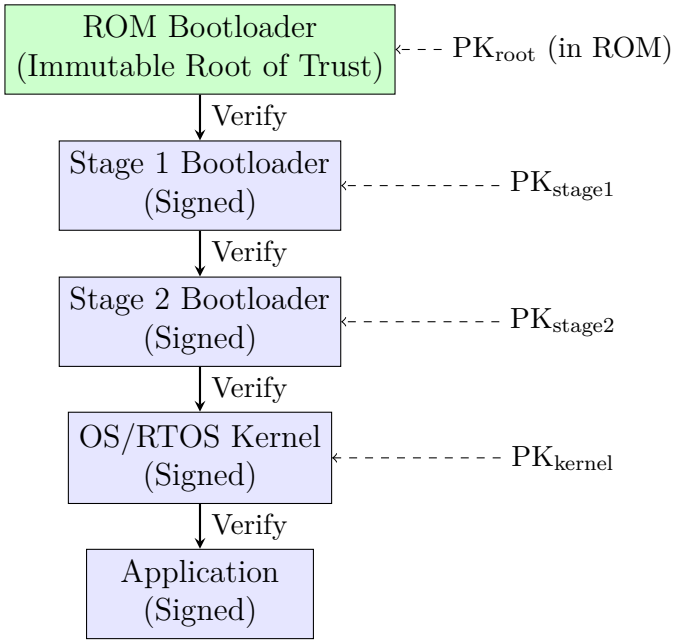


Figure 9.3: Multi-Stage Secure Boot Chain with PQC Signatures

9.5.1 Secure Boot Architecture

9.5.2 Algorithm Selection for Secure Boot

Secure boot has specific requirements that influence PQC algorithm selection:

Table 9.10: PQC Signature Algorithms for Secure Boot

Algorithm	PK Size	Sig Size	Verify (M4)	ROM Impact	Rating
ML-DSA-44	1,312 B	2,420 B	0.8 ms	Medium	Good
ML-DSA-65	1,952 B	3,293 B	1.2 ms	Medium	Good
Falcon-512	897 B	666 B	0.3 ms	High*	Best
Falcon-1024	1,793 B	1,280 B	0.6 ms	High*	Good
SLH-DSA-128s	32 B	7,856 B	2.1 ms	Low	Limited
SLH-DSA-128f	32 B	17,088 B	0.4 ms	Low	Good

* Falcon requires floating-point or complex fixed-point code in ROM

Note: For secure boot, **Falcon-512** offers the best combination of small signatures and fast verification. However, its implementation complexity may favor **ML-DSA-44** for simpler deployments. **SLH-DSA-128f** is ideal when ROM space is extremely limited (only 32-byte public key in ROM).

9.5.3 PQC Secure Boot Implementation

```
/*
 * Post-Quantum Secure Boot for IoT Devices
 * Supports ML-DSA, Falcon, and SLH-DSA
 */

#include <stdint.h>
#include <stdbool.h>
#include <string.h>

/* Boot stage definitions */
typedef enum {
    BOOT_STAGE_ROM = 0,
    BOOT_STAGE_BL1 = 1,
    BOOT_STAGE_BL2 = 2,
    BOOT_STAGE_KERNEL = 3,
    BOOT_STAGE_APP = 4
} boot_stage_t;

/* Signature algorithm identifiers */
```

```
typedef enum {
    SIG_ALG_MLDSA44 = 1,
    SIG_ALG_MLDSA65 = 2,
    SIG_ALG_MLDSA87 = 3,
    SIG_ALG_FALCON512 = 4,
    SIG_ALG_FALCON1024 = 5,
    SIG_ALG_SLHDSA_SHA2_128F = 6,
    SIG_ALG_SLHDSA_SHA2_128S = 7,
    /* Hybrid options */
    SIG_ALG_HYBRID_ECDSA_MLDSA = 0x80,
    SIG_ALG_HYBRID_ED25519_FALCON = 0x81
} sig_algorithm_t;

/* Image header structure */
typedef struct __attribute__((packed)) {
    uint32_t magic;              /* 0x50514654 = "PQFT" */
    uint32_t header_version;     /* Header format version */
    uint32_t image_size;         /* Size of image (excluding
       header) */
    uint32_t load_address;       /* Where to load the image
       */
    uint32_t entry_point;        /* Execution entry point */
    uint8_t  algorithm;          /* Signature algorithm ID */
    uint8_t  security_level;     /* NIST security level
       (1-5) */
    uint16_t pk_size;            /* Public key size in bytes
       */
    uint16_t sig_size;           /* Signature size in bytes
       */
    uint16_t reserved;
    uint8_t  image_hash[64];     /* SHA-512 hash of image */
    uint8_t  pk_hash[32];        /* SHA-256 hash of next
       stage's PK */
    uint32_t version;            /* Firmware version
       (anti-rollback) */
    uint32_t build_timestamp;    /* Build time for audit */
    /* Followed by: signature, then public key for next
       stage */
} pqc_image_header_t;

#define PQC_IMAGE_MAGIC 0x50514654
```

```
53
54 /* Root public key stored in ROM (example: ML-DSA-44) */
55 extern const uint8_t ROM_ROOT_PUBLIC_KEY[];
56 extern const uint16_t ROM_ROOT_PUBLIC_KEY_SIZE;
57 extern const uint8_t ROM_ROOT_ALGORITHM;
58
59 /* Anti-rollback counter in secure storage */
60 extern uint32_t
      secure_storage_read_rollback_counter(boot_stage_t stage);
61 extern int
      secure_storage_update_rollback_counter(boot_stage_t stage,
62                                                 uint32_t
                                                      version);
63
64 /* Signature verification dispatcher */
65 static int verify_signature(const uint8_t *message, size_t
      msg_len,
66                             const uint8_t *signature, size_t
                                  sig_len,
67                             const uint8_t *public_key, size_t
                                  pk_len,
68                             sig_algorithm_t algorithm) {
69     switch (algorithm) {
70         case SIG_ALG_MLDSA44:
71             return mldsa44_verify(signature, sig_len,
72                                   message, msg_len,
                                      public_key);
73
74         case SIG_ALG_MLDSA65:
75             return mldsa65_verify(signature, sig_len,
76                                   message, msg_len,
                                      public_key);
77
78         case SIG_ALG_MLDSA87:
79             return mldsa87_verify(signature, sig_len,
80                                   message, msg_len,
                                      public_key);
81
82         case SIG_ALG_FALCON512:
83             return falcon512_verify(signature, sig_len,
```

```
84                                      message, msg_len,
                                           public_key);
85
86          case SIG_ALG_FALCON1024:
87              return falcon1024_verify(signature, sig_len,
88                                       message, msg_len,
                                            public_key);
89
90          case SIG_ALG_SLHDSA_SHA2_128F:
91              return slhdsa_sha2_128f_verify(signature,
                    sig_len,
92                                             message, msg_len,
                                                  public_key);
93
94          case SIG_ALG_HYBRID_ECDSA_MLDSA:
95              return verify_hybrid_ecdsa_mldsa(message,
                    msg_len,
96                                               signature,
                                                    sig_len,
97                                               public_key,
                                                    pk_len);
98
99          default:
100             return -1;  /* Unknown algorithm */
101     }
102 }
103
104 /* Verify hybrid signature (both must pass) */
105 static int verify_hybrid_ecdsa_mldsa(const uint8_t *msg,
       size_t msg_len,
106                                      const uint8_t *sig,
                                            size_t sig_len,
107                                      const uint8_t *pk,
                                            size_t pk_len) {
108     /* Hybrid signature format:
109      * [ECDSA sig (64 bytes)] [ML-DSA sig (2420 bytes)]
110      * Hybrid public key format:
111      * [ECDSA pk (64 bytes)] [ML-DSA pk (1312 bytes)]
112      */
113
114     const uint8_t *ecdsa_sig = sig;
```

```
115     const uint8_t *mldsa_sig = sig + 64;
116     const uint8_t *ecdsa_pk = pk;
117     const uint8_t *mldsa_pk = pk + 64;
118 
119     /* Both signatures must verify */
120     int ecdsa_result = ecdsa_p256_verify(ecdsa_sig, 64, msg,
           msg_len, ecdsa_pk);
121     int mldsa_result = mldsa44_verify(mldsa_sig, 2420, msg,
           msg_len, mldsa_pk);
122 
123     /* Constant-time AND of results */
124     return (ecdsa_result | mldsa_result);  /* 0 only if both
           are 0 */
125 }
126 
127 /* Main secure boot verification function */
128 typedef enum {
129     BOOT_OK = 0,
130     BOOT_ERR_MAGIC = -1,
131     BOOT_ERR_HASH = -2,
132     BOOT_ERR_SIGNATURE = -3,
133     BOOT_ERR_ROLLBACK = -4,
134     BOOT_ERR_ALGORITHM = -5,
135     BOOT_ERR_SIZE = -6
136 } boot_result_t;
137 
138 boot_result_t verify_boot_image(const uint8_t *image_base,
139                                 const uint8_t *trusted_pk,
140                                 size_t trusted_pk_size,
141                                 sig_algorithm_t trusted_alg,
142                                 boot_stage_t stage) {
143     const pqc_image_header_t *header = (const
           pqc_image_header_t *)image_base;
144 
145     /* Step 1: Validate magic number */
146     if (header->magic != PQC_IMAGE_MAGIC) {
147         return BOOT_ERR_MAGIC;
148     }
149 
150     /* Step 2: Validate algorithm matches expected */
151     if (header->algorithm != trusted_alg) {
```

```
152         return BOOT_ERR_ALGORITHM;
153     }
154 
155     /* Step 3: Check anti-rollback */
156     uint32_t stored_version =
           secure_storage_read_rollback_counter(stage);
157     if (header->version < stored_version) {
158         return BOOT_ERR_ROLLBACK;
159     }
160 
161     /* Step 4: Compute hash of image */
162     const uint8_t *image_data = image_base +
           sizeof(pqc_image_header_t)
163                                 + header->sig_size +
                                       header->pk_size;
164     uint8_t computed_hash[64];
165     sha512(computed_hash, image_data, header->image_size);
166 
167     /* Step 5: Verify hash matches header */
168     if (secure_memcmp(computed_hash, header->image_hash, 64)
           != 0) {
169         return BOOT_ERR_HASH;
170     }
171 
172     /* Step 6: Verify signature over header (which includes
           hash) */
173     const uint8_t *signature = image_base +
           sizeof(pqc_image_header_t);
174 
175     /* Sign header up to (but not including) signature */
176     int sig_result = verify_signature(
177         (const uint8_t *)header,
178         offsetof(pqc_image_header_t, image_hash) + 64 + 32,
               /* Through pk_hash */
179         signature, header->sig_size,
180         trusted_pk, trusted_pk_size,
181         (sig_algorithm_t)header->algorithm
182     );
183 
184     if (sig_result != 0) {
185         return BOOT_ERR_SIGNATURE;
```

```
186     }
187
188     /* Step 7: Update rollback counter if version increased
           */
189     if (header->version > stored_version) {
190         secure_storage_update_rollback_counter(stage,
               header->version);
191     }
192
193     return BOOT_OK;
194 }
195
196 /* Complete boot chain verification */
197 boot_result_t secure_boot_chain(void) {
198     boot_result_t result;
199
200     /* Stage 1: ROM verifies BL1 using root key */
201     const uint8_t *bl1_base = (const uint8_t
           *)BL1_LOAD_ADDRESS;
202     result = verify_boot_image(bl1_base,
203                                ROM_ROOT_PUBLIC_KEY,
204                                ROM_ROOT_PUBLIC_KEY_SIZE,
205                                (sig_algorithm_t)ROM_ROOT_ALGORITHM,
206                                BOOT_STAGE_BL1);
207     if (result != BOOT_OK) {
208         boot_failure_handler(BOOT_STAGE_BL1, result);
209         return result;
210     }
211
212     /* Extract BL1's public key for next stage */
213     const pqc_image_header_t *bl1_hdr = (const
           pqc_image_header_t *)bl1_base;
214     const uint8_t *bl1_pk = bl1_base +
           sizeof(pqc_image_header_t)
215                             + bl1_hdr->sig_size;
216
217     /* Stage 2: BL1 verifies BL2 */
218     const uint8_t *bl2_base = (const uint8_t
           *)BL2_LOAD_ADDRESS;
219     result = verify_boot_image(bl2_base,
220                                bl1_pk, bl1_hdr->pk_size,
```

```
                                    (sig_algorithm_t)bl1_hdr->algorithm,
                                    BOOT_STAGE_BL2);
    if (result != BOOT_OK) {
        boot_failure_handler(BOOT_STAGE_BL2, result);
        return result;
    }

    /* Continue chain for kernel and application... */
    /* (Similar pattern) */

    return BOOT_OK;
}

/* Boot failure handler */
void boot_failure_handler(boot_stage_t stage, boot_result_t
    error) {
    /* Log failure securely */
    secure_log_boot_failure(stage, error);

    /* Options based on policy:
     * 1. Halt system (most secure)
     * 2. Boot recovery image
     * 3. Revert to known-good image (A/B scheme)
     */

    #if defined(BOOT_POLICY_HALT)
        while (1) {
            /* Halt - device bricked until recovery */
            __WFI();
        }
    #elif defined(BOOT_POLICY_RECOVERY)
        boot_recovery_image();
    #elif defined(BOOT_POLICY_AB_FALLBACK)
        boot_alternate_slot();
    #endif
}
```

Listing 9.12: Post-Quantum Secure Boot Implementation

9.5.4 Boot Time Analysis

Boot time is critical for IoT devices. The following analysis shows PQC impact:

Table 9.11: Secure Boot Time Breakdown (Cortex-M4 @ 168 MHz)

Operation	Classical	ML-DSA-44	Falcon-512	SLH-DSA-128f	Hybrid
Hash 64KB image	12 ms	12 ms	12 ms	12 ms	12 ms
Load signature	0.1 ms	0.4 ms	0.1 ms	2.8 ms	0.5 ms
Load public key	0.1 ms	0.2 ms	0.1 ms	0.01 ms	0.3 ms
Verify signature	0.3 ms	0.8 ms	0.3 ms	0.4 ms	1.1 ms
Total per stage	**12.5 ms**	**13.4 ms**	**12.5 ms**	**15.2 ms**	**13.9 ms**
4-stage chain	**50 ms**	**54 ms**	**50 ms**	**61 ms**	**56 ms**

Note: PQC secure boot adds approximately **4-11 ms** per boot stage compared to classical ECDSA. For a 4-stage boot chain, total overhead is **16-44 ms**—acceptable for most IoT applications where boot time requirements are typically <1 second.

9.5.5 Hardware Root of Trust Considerations

```
/*
 * Hardware Root of Trust (HRoT) Integration for PQC
 * Example for ARM TrustZone-M based devices
 */

#include <stdint.h>
#include <arm_cmse.h>  /* ARM CMSE for TrustZone */

/* Secure region definitions */
#define SECURE_ROM_BASE     0x10000000
#define SECURE_ROM_SIZE     0x00010000  /* 64 KB secure ROM
   */
#define SECURE_RAM_BASE     0x30000000
#define SECURE_RAM_SIZE     0x00008000  /* 32 KB secure RAM
   */

/* Root public key in secure ROM (One-Time Programmable or
   ROM) */
__attribute__((section(".secure_rom.root_pk")))
const uint8_t root_public_key_mldsa44[1312] = {
    /* ML-DSA-44 public key programmed at manufacturing */
    /* This cannot be modified after device provisioning */
};

/* Secure boot state - only accessible from secure world */
```

```
typedef struct {
    bool boot_verified;
    boot_stage_t current_stage;
    uint8_t chain_hash[32];     /* Rolling hash of boot
       chain */
    uint32_t boot_flags;
} secure_boot_state_t;

__attribute__((section(".secure_ram")))
static secure_boot_state_t g_boot_state;

/* Secure function - callable from non-secure world */
__attribute__((cmse_nonsecure_entry))
bool secure_get_boot_status(void) {
    return g_boot_state.boot_verified;
}

/* Secure function - get attestation token */
__attribute__((cmse_nonsecure_entry))
int secure_get_attestation(uint8_t *token, size_t
   *token_len) {
    if (!g_boot_state.boot_verified) {
        return -1;  /* Cannot attest unverified boot */
    }

    /* Generate attestation token including:
     * - Boot chain hash
     * - Device ID
     * - Firmware versions
     * - Security configuration
     */
    return generate_attestation_token(token, token_len,
       &g_boot_state);
}

/* Secure boot verification - runs in secure world only */
__attribute__((section(".secure_text")))
boot_result_t secure_verify_image(const uint8_t *image,
   size_t image_size) {
    /* Ensure we're running in secure state */
    if (!__get_CONTROL() & 0x1) {
```

```
60          /* Not in secure state - should not happen */
61          return BOOT_ERR_SECURITY;
62      }
63
64      /* Perform verification using secure ROM key */
65      return verify_boot_image(image,
66                               root_public_key_mldsa44,
67                               sizeof(root_public_key_mldsa44),
68                               SIG_ALG_MLDSA44,
69                               g_boot_state.current_stage);
70  }
71
72  /* Memory Protection Unit configuration for boot stages */
73  void configure_mpu_for_boot_stage(boot_stage_t stage) {
74      /* Disable MPU during reconfiguration */
75      MPU->CTRL = 0;
76
77      /* Region 0: Secure ROM - RO, executable */
78      MPU->RNR = 0;
79      MPU->RBAR = SECURE_ROM_BASE | MPU_RBAR_VALID_Msk | 0;
80      MPU->RASR = MPU_RASR_ENABLE_Msk |
81                  (0x0F << MPU_RASR_SIZE_Pos) |  /* 64 KB */
82                  (0x06 << MPU_RASR_AP_Pos) |    /* RO */
83                  MPU_RASR_C_Msk;
84
85      /* Region 1: Secure RAM - RW, no execute */
86      MPU->RNR = 1;
87      MPU->RBAR = SECURE_RAM_BASE | MPU_RBAR_VALID_Msk | 1;
88      MPU->RASR = MPU_RASR_ENABLE_Msk |
89                  (0x0E << MPU_RASR_SIZE_Pos) |  /* 32 KB */
90                  (0x03 << MPU_RASR_AP_Pos) |    /* RW */
91                  MPU_RASR_XN_Msk |              /* No execute
                       */
92                  MPU_RASR_C_Msk;
93
94      /* Additional regions based on boot stage */
95      switch (stage) {
96          case BOOT_STAGE_BL1:
97              /* BL1 has access to more peripherals */
98              configure_bl1_mpu_regions();
99              break;
```

```
100         case BOOT_STAGE_KERNEL:
101             /* Lock down further before kernel */
102             configure_kernel_mpu_regions();
103             break;
104         default:
105             break;
106     }
107
108     /* Enable MPU with default memory map for privileged */
109     MPU->CTRL = MPU_CTRL_ENABLE_Msk |
            MPU_CTRL_PRIVDEFENA_Msk;
110     __DSB();
111     __ISB();
112 }
```

Listing 9.13: Hardware Root of Trust Integration

9.6 Secure Firmware Updates with PQC

Over-the-air (OTA) firmware updates are essential for maintaining IoT device security throughout their lifecycle. Post-quantum signatures ensure that firmware authenticity remains protected against future quantum attacks, while the larger signature sizes introduce new challenges for bandwidth-constrained networks.

9.6.1 OTA Update Architecture

9.6.2 Update Package Format

```
1  /*
2   * Post-Quantum Secured Firmware Update Package
3   * Supports delta updates and compression
4   */
5
6  #include <stdint.h>
7
8  /* Update package magic numbers */
9  #define UPDATE_MAGIC_FULL   0x50514F54  /* "PQOT" - Full
       image */
10 #define UPDATE_MAGIC_DELTA  0x50514454  /* "PQDT" - Delta
       update */
11
```

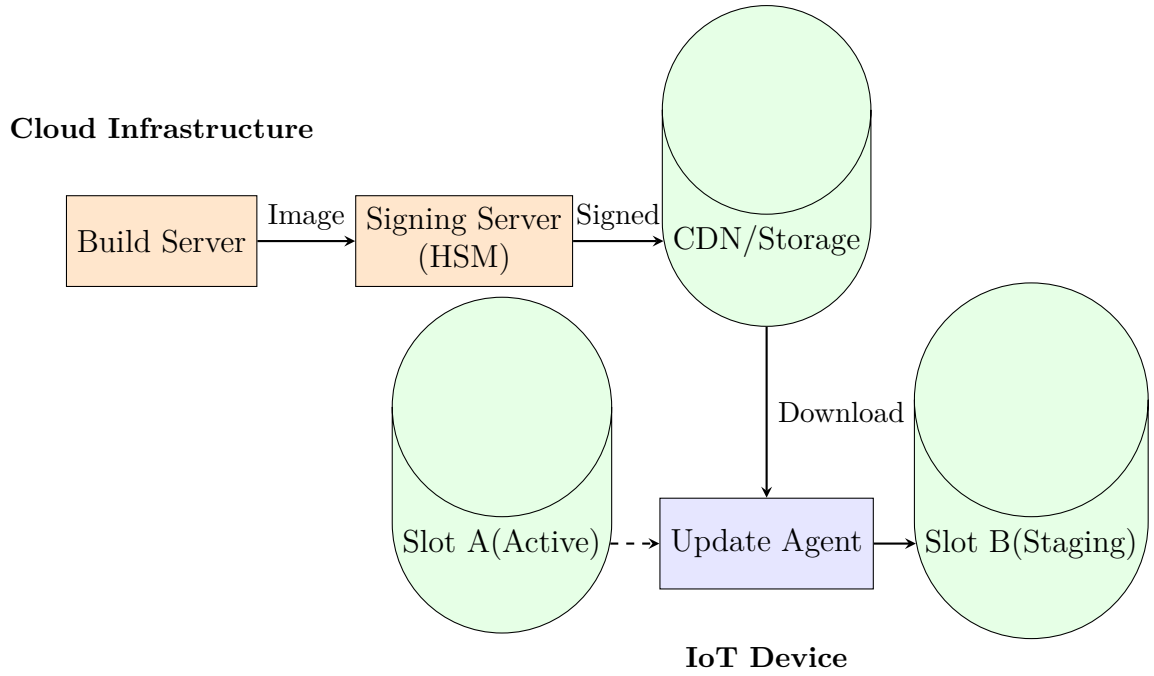


Figure 9.4: PQC-Secured OTA Update Architecture

```
/* Compression algorithms */
typedef enum {
    COMPRESS_NONE = 0,
    COMPRESS_LZ4 = 1,
    COMPRESS_ZSTD = 2,
    COMPRESS_LZMA = 3
} compression_algo_t;

/* Update package header */
typedef struct __attribute__((packed)) {
    uint32_t magic;                  /* Package type
       identifier */
    uint32_t header_version;         /* Header format
       version */
    uint32_t package_size;           /* Total package size */
    uint32_t payload_size;           /* Compressed payload
       size */
    uint32_t uncompressed_size;      /* Original image size
       */

    /* Target information */
    uint32_t target_hw_id;           /* Hardware ID this
       update is for */
```

```
30      uint32_t min_version;            /* Minimum current
           version required */
31      uint32_t new_version;            /* Version after update
           */
32
33      /* For delta updates */
34      uint32_t base_version;           /* Base version for
           delta */
35      uint8_t  base_hash[32];          /* SHA-256 of base
           image */
36
37      /* Cryptographic parameters */
38      uint8_t  sig_algorithm;          /* Signature algorithm
           ID */
39      uint8_t  compression;            /* Compression
           algorithm */
40      uint16_t sig_size;               /* Signature size in
           bytes */
41      uint16_t pk_size;                /* Public key size (if
           included) */
42      uint16_t manifest_size;          /* Manifest size */
43
44      /* Hashes */
45      uint8_t  payload_hash[32];       /* SHA-256 of
           compressed payload */
46      uint8_t  image_hash[32];         /* SHA-256 of final
           uncompressed image */
47
48      /* Timestamps */
49      uint64_t created_timestamp;      /* Package creation
           time */
50      uint64_t expiry_timestamp;       /* Package expiry (0 =
           no expiry) */
51
52      /* Followed by:
53       * 1. Manifest (JSON/CBOR with detailed metadata)
54       * 2. Signature over (header + manifest)
55       * 3. Optional: Public key for next update
56       * 4. Compressed/delta payload
57       */
58  } pqc_update_header_t;
```

```
59
60 /* Manifest structure (CBOR encoded) */
61 typedef struct {
62     char device_class[64];           /* Device class
       identifier */
63     char vendor_id[32];              /* Vendor identifier */
64     uint32_t component_id;           /* Component being
       updated */
65
66     /* Dependencies */
67     uint32_t required_bootloader;    /* Minimum bootloader
       version */
68     uint32_t required_secure_boot;   /* Required secure boot
       version */
69
70     /* Installation instructions */
71     uint8_t  requires_reboot;        /* Reboot required
       after install */
72     uint8_t  allow_downgrade;        /* Allow version
       downgrade */
73     uint8_t  critical_update;        /* Security-critical
       update */
74     uint8_t  install_priority;       /* Installation
       priority (0-255) */
75
76     /* Recovery information */
77     uint32_t rollback_version;       /* Version to rollback
       to on failure */
78     char     release_notes_url[256]; /* URL for release
       notes */
79 } update_manifest_t;
```

Listing 9.14: PQC Firmware Update Package Structure

9.6.3 Bandwidth-Efficient Update Strategies

PQC signatures are larger than classical signatures, impacting bandwidth-constrained IoT networks:

```
1 /*
2  * Delta Update System for Bandwidth-Constrained IoT
3  * Uses bsdiff-style binary differencing
```

Table 9.12: Update Overhead Comparison

Component	ECDSA	ML-DSA-44	Falcon-512	SLH-DSA-128f
Signature	64 B	2,420 B	666 B	17,088 B
Public Key (optional)	64 B	1,312 B	897 B	32 B
Header overhead	256 B	256 B	256 B	256 B
Total overhead	**384 B**	**3,988 B**	**1,819 B**	**17,376 B**
For 100 KB update	0.4%	3.9%	1.8%	17.4%

```
 */

#include <stdint.h>
#include <stdbool.h>

/* Delta patch format */
typedef struct __attribute__((packed)) {
    uint32_t magic;          /* 0x44454C54 "DELT" */
    uint32_t ctrl_len;       /* Control block length */
    uint32_t diff_len;       /* Diff block length */
    uint32_t extra_len;      /* Extra block length */
    uint32_t new_size;       /* Size of new file */
} delta_header_t;

/* Control tuple for delta application */
typedef struct __attribute__((packed)) {
    int32_t diff_len;        /* Bytes to copy from diff
       block */
    int32_t extra_len;       /* Bytes to copy from extra
       block */
    int32_t seek_offset;     /* Seek in old file (can be
       negative) */
} delta_ctrl_t;

/* Stream interface for memory-constrained devices */
typedef struct {
    int (*read)(void *ctx, uint8_t *buf, size_t len);
    int (*write)(void *ctx, const uint8_t *buf, size_t len);
    int (*seek)(void *ctx, int32_t offset, int whence);
    void *context;
} stream_t;

/* Apply delta patch with minimal RAM usage */
```

```
int apply_delta_patch_streaming(stream_t *old_image,
                                stream_t *patch,
                                stream_t *new_image,
                                uint8_t *work_buffer,
                                size_t work_size) {
    delta_header_t header;
    delta_ctrl_t ctrl;
    int32_t old_pos = 0;
    int32_t new_pos = 0;

    /* Read and validate header */
    if (patch->read(patch->context, (uint8_t *)&header,
                    sizeof(header)) != sizeof(header)) {
        return -1;
    }

    if (header.magic != 0x44454C54) {
        return -2;  /* Invalid magic */
    }

    /* Process control tuples */
    size_t ctrl_pos = 0;
    while (ctrl_pos < header.ctrl_len) {
        /* Read control tuple */
        if (patch->read(patch->context, (uint8_t *)&ctrl,
                        sizeof(ctrl)) != sizeof(ctrl)) {
            return -3;
        }
        ctrl_pos += sizeof(ctrl);

        /* Apply diff block: new[i] = old[i] + diff[i] */
        size_t remaining = ctrl.diff_len;
        while (remaining > 0) {
            size_t chunk = (remaining < work_size / 2) ?
                           remaining : work_size / 2;

            /* Read from old image */
            uint8_t *old_buf = work_buffer;
            old_image->read(old_image->context, old_buf,
               chunk);

```

```
74              /* Read diff block */
75              uint8_t *diff_buf = work_buffer + work_size / 2;
76              patch->read(patch->context, diff_buf, chunk);
77
78              /* Apply diff: new = old + diff */
79              for (size_t i = 0; i < chunk; i++) {
80                  old_buf[i] += diff_buf[i];
81              }
82
83              /* Write to new image */
84              new_image->write(new_image->context, old_buf,
                    chunk);
85
86              old_pos += chunk;
87              new_pos += chunk;
88              remaining -= chunk;
89          }
90
91          /* Copy extra block directly to new image */
92          remaining = ctrl.extra_len;
93          while (remaining > 0) {
94              size_t chunk = (remaining < work_size) ?
                    remaining : work_size;
95              patch->read(patch->context, work_buffer, chunk);
96              new_image->write(new_image->context,
                    work_buffer, chunk);
97              new_pos += chunk;
98              remaining -= chunk;
99          }
100
101         /* Seek in old file */
102         old_pos += ctrl.seek_offset;
103         old_image->seek(old_image->context, old_pos,
                SEEK_SET);
104     }
105
106     return (new_pos == (int32_t)header.new_size) ? 0 : -4;
107 }
108
109 /* Calculate delta patch size savings */
110 typedef struct {
```

```
111     size_t full_size;
112     size_t delta_size;
113     float savings_percent;
114     bool delta_viable;      /* True if delta is worth using
           */
115 } delta_analysis_t;
116
117 delta_analysis_t analyze_delta_savings(size_t
       full_image_size,
118                                        size_t delta_size,
119                                        size_t pqc_overhead)
                                              {
120     delta_analysis_t result;
121
122     result.full_size = full_image_size + pqc_overhead;
123     result.delta_size = delta_size + pqc_overhead;
124
125     if (result.delta_size < result.full_size) {
126         result.savings_percent = 100.0f *
127             (1.0f - (float)result.delta_size /
                   result.full_size);
128         /* Delta viable if saves at least 20% */
129         result.delta_viable = (result.savings_percent >
               20.0f);
130     } else {
131         result.savings_percent = 0;
132         result.delta_viable = false;
133     }
134
135     return result;
136 }
```

Listing 9.15: Delta Update Generator and Applier

9.6.4 A/B Update Scheme with Rollback

```
1 /*
2  * A/B Partition Update System
3  * Ensures atomic updates with automatic rollback
4  */
5
6 #include <stdint.h>
```

```
#include <stdbool.h>

/* Partition slot identifiers */
typedef enum {
    SLOT_A = 0,
    SLOT_B = 1
} partition_slot_t;

/* Slot metadata stored in protected flash region */
typedef struct __attribute__((packed)) {
    uint32_t magic;              /* 0x534C4F54 "SLOT" */
    uint32_t version;            /* Firmware version in slot
       */
    uint32_t crc32;              /* CRC32 of slot contents */
    uint8_t  image_hash[32];     /* SHA-256 of image */
    uint8_t  status;             /* Slot status */
    uint8_t  boot_attempts;      /* Failed boot attempts */
    uint8_t  max_boot_attempts;  /* Max attempts before
       rollback */
    uint8_t  reserved;
    uint32_t install_time;       /* When this slot was
       written */
} slot_metadata_t;

typedef enum {
    SLOT_STATUS_EMPTY = 0,       /* No valid image */
    SLOT_STATUS_VALID = 1,       /* Valid, not yet tested */
    SLOT_STATUS_TESTING = 2,     /* Booting for testing */
    SLOT_STATUS_CONFIRMED = 3,   /* Confirmed working */
    SLOT_STATUS_FAILED = 4       /* Failed validation */
} slot_status_t;

/* Boot control block */
typedef struct __attribute__((packed)) {
    uint32_t magic;              /* 0x424F4F54 "BOOT" */
    uint8_t  active_slot;        /* Currently active slot */
    uint8_t  pending_slot;       /* Slot to try on next boot
       */
    uint8_t  rollback_slot;      /* Slot to rollback to */
    uint8_t  flags;
```

```
43      slot_metadata_t slots[2];    /* Metadata for both slots
            */
44  } boot_control_t;
45
46  /* Global boot control (in protected storage) */
47  extern boot_control_t *g_boot_ctrl;
48
49  /* Get the inactive slot for staging updates */
50  partition_slot_t get_update_slot(void) {
51      return (g_boot_ctrl->active_slot == SLOT_A) ? SLOT_B :
            SLOT_A;
52  }
53
54  /* Write update to inactive slot */
55  int stage_update(const pqc_update_header_t *header,
56                   stream_t *payload_stream,
57                   const uint8_t *signature,
58                   const uint8_t *signing_key) {
59      partition_slot_t target = get_update_slot();
60      slot_metadata_t *slot_meta = &g_boot_ctrl->slots[target];
61
62      /* Step 1: Verify signature before writing anything */
63      /* (Signature covers header which contains image hash) */
64      if (verify_update_signature(header, signature,
65                                  header->sig_size,
                                       signing_key) != 0) {
66          return -1;  /* Signature verification failed */
67      }
68
69      /* Step 2: Erase target partition */
70      if (flash_erase_partition(target) != 0) {
71          return -2;
72      }
73
74      /* Step 3: Stream and decompress payload to flash */
75      uint8_t work_buffer[4096];
76      sha256_ctx_t hash_ctx;
77      sha256_init(&hash_ctx);
78
79      size_t written = 0;
80      size_t remaining = header->payload_size;
```

```
81
82      /* Initialize decompressor */
83      decompress_ctx_t decomp;
84      decompress_init(&decomp, header->compression);
85
86      while (remaining > 0) {
87          size_t chunk = (remaining < sizeof(work_buffer)) ?
88                         remaining : sizeof(work_buffer);
89
90          /* Read compressed chunk */
91          if (payload_stream->read(payload_stream->context,
92                                   work_buffer, chunk) !=
                                        (int)chunk) {
93              return -3;
94          }
95
96          /* Decompress and write */
97          uint8_t decompressed[8192];
98          size_t decomp_len;
99
100         if (decompress_chunk(&decomp, work_buffer, chunk,
101                              decompressed, &decomp_len) !=
                                    0) {
102             return -4;
103         }
104
105         /* Update hash */
106         sha256_update(&hash_ctx, decompressed, decomp_len);
107
108         /* Write to flash */
109         if (flash_write_partition(target, written,
110                                   decompressed, decomp_len)
                                        != 0) {
111             return -5;
112         }
113
114         written += decomp_len;
115         remaining -= chunk;
116     }
117
118     decompress_finish(&decomp);
```

```
119
120     /* Step 4: Verify hash of written image */
121     uint8_t computed_hash[32];
122     sha256_final(&hash_ctx, computed_hash);
123
124     if (memcmp(computed_hash, header->image_hash, 32) != 0) {
125         /* Hash mismatch - mark slot as failed */
126         slot_meta->status = SLOT_STATUS_FAILED;
127         return -6;
128     }
129
130     /* Step 5: Update slot metadata */
131     slot_meta->magic = 0x534C4F54;
132     slot_meta->version = header->new_version;
133     slot_meta->crc32 = crc32_partition(target);
134     memcpy(slot_meta->image_hash, computed_hash, 32);
135     slot_meta->status = SLOT_STATUS_VALID;
136     slot_meta->boot_attempts = 0;
137     slot_meta->max_boot_attempts = 3;
138     slot_meta->install_time = get_current_time();
139
140     /* Step 6: Mark slot as pending for next boot */
141     g_boot_ctrl->pending_slot = target;
142
143     /* Persist boot control */
144     save_boot_control(g_boot_ctrl);
145
146     return 0;
147 }
148
149 /* Called early in boot to manage A/B state */
150 partition_slot_t select_boot_slot(void) {
151     slot_metadata_t *pending =
        &g_boot_ctrl->slots[g_boot_ctrl->pending_slot];
152     slot_metadata_t *active =
        &g_boot_ctrl->slots[g_boot_ctrl->active_slot];
153
154     /* Check if we have a pending update to test */
155     if (pending->status == SLOT_STATUS_VALID) {
156         pending->status = SLOT_STATUS_TESTING;
157         pending->boot_attempts++;
```

```
158         save_boot_control(g_boot_ctrl);
159         return g_boot_ctrl->pending_slot;
160     }
161
162     /* Check if we're testing an update */
163     if (pending->status == SLOT_STATUS_TESTING) {
164         if (pending->boot_attempts >=
               pending->max_boot_attempts) {
165             /* Too many failed attempts - rollback */
166             pending->status = SLOT_STATUS_FAILED;
167             g_boot_ctrl->pending_slot =
                   g_boot_ctrl->active_slot;
168             save_boot_control(g_boot_ctrl);
169             return g_boot_ctrl->active_slot;
170         }
171         pending->boot_attempts++;
172         save_boot_control(g_boot_ctrl);
173         return g_boot_ctrl->pending_slot;
174     }
175
176     /* Boot confirmed active slot */
177     return g_boot_ctrl->active_slot;
178 }
179
180 /* Called by application after successful boot to confirm
       update */
181 int confirm_update(void) {
182     slot_metadata_t *current =
           &g_boot_ctrl->slots[g_boot_ctrl->pending_slot];
183
184     if (current->status != SLOT_STATUS_TESTING) {
185         return -1;  /* Not in testing state */
186     }
187
188     /* Confirm the update */
189     current->status = SLOT_STATUS_CONFIRMED;
190     g_boot_ctrl->active_slot = g_boot_ctrl->pending_slot;
191     g_boot_ctrl->rollback_slot = (g_boot_ctrl->active_slot
           == SLOT_A) ?
192                                  SLOT_B : SLOT_A;
193
```

```
194     save_boot_control(g_boot_ctrl);
195     return 0;
196 }
197
198 /* Manual rollback to previous version */
199 int rollback_update(void) {
200     partition_slot_t rollback = g_boot_ctrl->rollback_slot;
201     slot_metadata_t *slot = &g_boot_ctrl->slots[rollback];
202
203     if (slot->status != SLOT_STATUS_CONFIRMED) {
204         return -1;  /* No valid rollback slot */
205     }
206
207     /* Set rollback slot as pending */
208     g_boot_ctrl->pending_slot = rollback;
209     g_boot_ctrl->slots[g_boot_ctrl->active_slot].status =
        SLOT_STATUS_VALID;
210
211     save_boot_control(g_boot_ctrl);
212
213     /* Trigger reboot */
214     system_reboot();
215
216     return 0;  /* Won't reach here */
217 }
```

Listing 9.16: A/B Partition Update Manager

9.6.5 Update Verification Pipeline

```
1  /*
2   * End-to-End Update Verification
3   * Multiple layers of validation
4   */
5
6  typedef enum {
7      UPDATE_OK = 0,
8      UPDATE_ERR_NETWORK = -1,
9      UPDATE_ERR_SIGNATURE = -2,
10     UPDATE_ERR_VERSION = -3,
11     UPDATE_ERR_HARDWARE = -4,
12     UPDATE_ERR_HASH = -5,
```

```
13      UPDATE_ERR_STORAGE = -6,
14      UPDATE_ERR_EXPIRED = -7,
15      UPDATE_ERR_DEPENDENCY = -8
16  } update_result_t;
17
18  /* Complete update verification */
19  update_result_t verify_update_package(const uint8_t *package,
20                                        size_t package_len,
21                                        const uint8_t
                                             *trusted_key) {
22      const pqc_update_header_t *header = (const
            pqc_update_header_t *)package;
23
24      /* 1. Basic header validation */
25      if (header->magic != UPDATE_MAGIC_FULL &&
26          header->magic != UPDATE_MAGIC_DELTA) {
27          return UPDATE_ERR_SIGNATURE;
28      }
29
30      /* 2. Check expiry */
31      if (header->expiry_timestamp != 0 &&
32          get_current_time() > header->expiry_timestamp) {
33          return UPDATE_ERR_EXPIRED;
34      }
35
36      /* 3. Hardware compatibility */
37      if (header->target_hw_id != get_hardware_id()) {
38          return UPDATE_ERR_HARDWARE;
39      }
40
41      /* 4. Version check (anti-rollback) */
42      uint32_t current_version =
            get_current_firmware_version();
43      if (header->new_version <= current_version &&
44          !allow_downgrade_policy()) {
45          return UPDATE_ERR_VERSION;
46      }
47
48      if (header->min_version > current_version) {
49          return UPDATE_ERR_VERSION;  /* Current version too
                old */
```

```
50      }
51
52      /* 5. Parse and validate manifest */
53      const uint8_t *manifest = package +
           sizeof(pqc_update_header_t);
54      update_manifest_t parsed_manifest;
55      if (parse_manifest(manifest, header->manifest_size,
56                         &parsed_manifest) != 0) {
57          return UPDATE_ERR_SIGNATURE;
58      }
59
60      /* 6. Check dependencies */
61      if (parsed_manifest.required_bootloader >
           get_bootloader_version()) {
62          return UPDATE_ERR_DEPENDENCY;
63      }
64
65      /* 7. Verify signature */
66      const uint8_t *signature = manifest +
           header->manifest_size;
67      size_t signed_len = sizeof(pqc_update_header_t) +
           header->manifest_size;
68
69      int sig_result = verify_signature(
70          package, signed_len,
71          signature, header->sig_size,
72          trusted_key, get_key_size(header->sig_algorithm),
73          (sig_algorithm_t)header->sig_algorithm
74      );
75
76      if (sig_result != 0) {
77          return UPDATE_ERR_SIGNATURE;
78      }
79
80      /* 8. Verify payload hash */
81      const uint8_t *payload = signature + header->sig_size;
82      if (header->pk_size > 0) {
83          payload += header->pk_size;
84      }
85
86      uint8_t computed_hash[32];
```

```
87      sha256(computed_hash, payload, header->payload_size);
88
89      if (memcmp(computed_hash, header->payload_hash, 32) !=
           0) {
90          return UPDATE_ERR_HASH;
91      }
92
93      return UPDATE_OK;
94  }
```

Listing 9.17: Complete Update Verification Pipeline

Table 9.13: Update Security Checklist

Check	Purpose	Failure Action
Header magic	Valid package format	Reject immediately
Expiry timestamp	Prevent replay of old updates	Reject
Hardware ID	Prevent cross-device attacks	Reject
Version check	Anti-rollback protection	Reject
Manifest parse	Validate metadata	Reject
Dependency check	Ensure compatibility	Reject
PQC signature	Authenticity verification	Reject
Payload hash	Integrity verification	Reject
Post-install hash	Verify correct installation	Rollback
Boot test	Verify functionality	Auto-rollback

9.7 Certificate Lifecycle Management

X.509 certificates form the foundation of IoT device identity and authentication. The transition to post-quantum cryptography requires careful planning for certificate issuance, renewal, and revocation, particularly given the larger certificate sizes and the need for hybrid approaches during the migration period.

9.7.1 PQC Certificate Formats

Warning: A 3-certificate chain using ML-DSA-65 requires **18 KB** of storage and transmission—nearly 8x larger than ECDSA. For constrained IoT devices, consider Falcon-512 certificates or certificate compression techniques.

9.7.2 Certificate Provisioning Strategies

Table 9.14: X.509 Certificate Size Comparison

Algorithm	Public Key	Signature	Typical Cert	Chain (3 certs)
RSA-2048	256 B	256 B	1.2 KB	3.6 KB
ECDSA P-256	64 B	64 B	0.8 KB	2.4 KB
ML-DSA-44	1,312 B	2,420 B	4.5 KB	13.5 KB
ML-DSA-65	1,952 B	3,293 B	6.0 KB	18.0 KB
Falcon-512	897 B	666 B	2.3 KB	6.9 KB
SLH-DSA-128f	32 B	17,088 B	17.8 KB	53.4 KB
Hybrid (ECDSA+ML-DSA-44)	1,376 B	2,484 B	5.3 KB	15.9 KB

```c
/*
 * Certificate Provisioning for PQC-enabled IoT Devices
 * Supports factory provisioning and runtime enrollment
 */

#include <stdint.h>
#include <stdbool.h>
#include <string.h>

/* Certificate types */
typedef enum {
    CERT_TYPE_DEVICE_IDENTITY,    /* Long-term device identity */
    CERT_TYPE_OPERATIONAL,        /* Short-term operational cert */
    CERT_TYPE_ATTESTATION,        /* Platform attestation */
    CERT_TYPE_MANUFACTURER        /* Manufacturer CA cert */
} cert_type_t;

/* Certificate storage structure */
typedef struct {
    cert_type_t type;
    uint8_t algorithm;            /* Signature algorithm */
    uint32_t not_before;          /* Validity start (Unix timestamp) */
    uint32_t not_after;           /* Validity end */
    uint16_t cert_len;
    uint16_t key_len;             /* Associated private key length */
    uint8_t subject_hash[32];     /* SHA-256 of subject DN */
```

```
27      uint8_t issuer_hash[32];      /* SHA-256 of issuer DN */
28      /* Followed by: DER-encoded certificate, then encrypted
           private key */
29  } cert_entry_t;
30
31  /* Certificate store (in secure flash) */
32  #define MAX_CERTIFICATES 8
33  typedef struct {
34      uint32_t magic;               /* 0x43455254 "CERT" */
35      uint32_t version;
36      uint16_t num_certs;
37      uint16_t reserved;
38      cert_entry_t entries[MAX_CERTIFICATES];
39      /* Certificate data follows */
40  } cert_store_t;
41
42  /* Factory provisioning: Install manufacturer certificate
       chain */
43  int provision_manufacturer_chain(const uint8_t *root_cert,
       size_t root_len,
44                                   const uint8_t
                                        *intermediate_cert,
                                        size_t int_len,
45                                   const uint8_t
                                        *device_cert, size_t
                                        dev_len,
46                                   const uint8_t
                                        *device_key_encrypted,
47                                   size_t key_len) {
48      cert_store_t *store = get_cert_store();
49
50      /* Verify chain before storing */
51      if (verify_cert_chain(root_cert, root_len,
52                            intermediate_cert, int_len,
53                            device_cert, dev_len) != 0) {
54          return -1;
55      }
56
57      /* Store root CA (self-signed) */
58      if (store_certificate(store, CERT_TYPE_MANUFACTURER,
```

```
59                              root_cert, root_len, NULL, 0) !=
                                   0) {
60          return -2;
61      }
62
63      /* Store device certificate with encrypted private key */
64      if (store_certificate(store, CERT_TYPE_DEVICE_IDENTITY,
65                              device_cert, dev_len,
66                              device_key_encrypted, key_len) !=
                                   0) {
67          return -3;
68      }
69
70      /* Mark provisioning complete */
71      store->magic = 0x43455254;
72      persist_cert_store(store);
73
74      return 0;
75  }
76
77  /* Runtime enrollment using EST (Enrollment over Secure
       Transport) */
78  typedef struct {
79      char est_server[256];
80      uint16_t est_port;
81      uint8_t ca_cert_hash[32];     /* Expected CA cert hash
           for pinning */
82      uint8_t auth_cert[4096];      /* Bootstrap/manufacturer
           cert for auth */
83      size_t auth_cert_len;
84  } est_config_t;
85
86  int enroll_certificate_est(const est_config_t *config,
87                              uint8_t algorithm,
88                              uint8_t *new_cert, size_t
                                   *cert_len,
89                              uint8_t *new_key, size_t
                                   *key_len) {
90      int ret = -1;
91
92      /* Step 1: Generate new PQC keypair */
```

```
93      uint8_t pk[MLKEM_MAX_PK_BYTES];
94      uint8_t sk[MLKEM_MAX_SK_BYTES];
95
96      switch (algorithm) {
97          case SIG_ALG_MLDSA44:
98              mldsa44_keypair(pk, sk);
99              *key_len = MLDSA44_SK_BYTES;
100             break;
101         case SIG_ALG_MLDSA65:
102             mldsa65_keypair(pk, sk);
103             *key_len = MLDSA65_SK_BYTES;
104             break;
105         case SIG_ALG_FALCON512:
106             falcon512_keypair(pk, sk);
107             *key_len = FALCON512_SK_BYTES;
108             break;
109         default:
110             return -1;
111     }
112
113     /* Step 2: Create CSR (Certificate Signing Request) */
114     uint8_t csr[8192];
115     size_t csr_len;
116
117     if (create_pqc_csr(csr, &csr_len, pk, sk, algorithm,
118                        get_device_id(), get_device_serial())
                               != 0) {
119         goto cleanup;
120     }
121
122     /* Step 3: Connect to EST server with mutual TLS */
123     tls_context_t *tls = tls_connect_mutual(
124         config->est_server, config->est_port,
125         config->auth_cert, config->auth_cert_len,
126         config->ca_cert_hash
127     );
128
129     if (!tls) {
130         goto cleanup;
131     }
132
```

```
133     /* Step 4: Submit CSR via EST /simpleenroll */
134     if (est_simple_enroll(tls, csr, csr_len, new_cert,
            cert_len) != 0) {
135         tls_close(tls);
136         goto cleanup;
137     }
138 
139     tls_close(tls);
140 
141     /* Step 5: Verify received certificate */
142     if (verify_cert_matches_key(new_cert, *cert_len, pk,
            algorithm) != 0) {
143         goto cleanup;
144     }
145 
146     /* Step 6: Encrypt and output private key */
147     if (encrypt_private_key(new_key, sk, *key_len) != 0) {
148         goto cleanup;
149     }
150 
151     ret = 0;
152 
153 cleanup:
154     secure_zero(sk, sizeof(sk));
155     return ret;
156 }
```

Listing 9.18: IoT Device Certificate Provisioning

9.7.3 Certificate Renewal and Rotation

```
1  /*
2   * Certificate Renewal Manager
3   * Handles proactive renewal before expiration
4   */
5 
6  #include <stdint.h>
7  #include <stdbool.h>
8 
9  /* Renewal policy configuration */
10 typedef struct {
```

```
11      uint32_t renewal_threshold_days;  /* Days before expiry
            to renew */
12      uint32_t retry_interval_hours;    /* Retry interval on
            failure */
13      uint8_t max_retries;              /* Maximum renewal
            attempts */
14      bool allow_expired_renewal;       /* Allow renewal after
            expiry */
15      bool require_hybrid;              /* Require hybrid
            certificates */
16  } renewal_policy_t;
17
18  /* Default policy */
19  static const renewal_policy_t DEFAULT_RENEWAL_POLICY = {
20      .renewal_threshold_days = 30,
21      .retry_interval_hours = 24,
22      .max_retries = 5,
23      .allow_expired_renewal = true,
24      .require_hybrid = false
25  };
26
27  /* Certificate status */
28  typedef enum {
29      CERT_STATUS_VALID,
30      CERT_STATUS_EXPIRING_SOON,
31      CERT_STATUS_EXPIRED,
32      CERT_STATUS_REVOKED,
33      CERT_STATUS_NOT_YET_VALID
34  } cert_status_t;
35
36  /* Check certificate validity */
37  cert_status_t check_cert_status(const cert_entry_t *cert,
38                                  const renewal_policy_t
                                        *policy) {
39      uint32_t now = get_current_time();
40
41      if (now < cert->not_before) {
42          return CERT_STATUS_NOT_YET_VALID;
43      }
44
45      if (now > cert->not_after) {
```

```
46          return CERT_STATUS_EXPIRED;
47      }
48
49      /* Check if within renewal window */
50      uint32_t threshold = policy->renewal_threshold_days * 24
           * 3600;
51      if ((cert->not_after - now) < threshold) {
52          return CERT_STATUS_EXPIRING_SOON;
53      }
54
55      return CERT_STATUS_VALID;
56  }
57
58  /* Renewal state machine */
59  typedef struct {
60      cert_entry_t *target_cert;
61      renewal_policy_t policy;
62      uint8_t attempts;
63      uint32_t next_retry_time;
64      uint8_t new_cert[8192];
65      size_t new_cert_len;
66      uint8_t new_key[8192];
67      size_t new_key_len;
68      bool renewal_pending;
69  } renewal_state_t;
70
71  /* Periodic renewal check - call from main loop or timer */
72  int renewal_tick(renewal_state_t *state, const est_config_t
       *est_config) {
73      uint32_t now = get_current_time();
74
75      /* Check if renewal is needed */
76      cert_status_t status =
           check_cert_status(state->target_cert,
77                                                  &state->policy);
78
79      if (status == CERT_STATUS_VALID &&
           !state->renewal_pending) {
80          return 0;  /* No action needed */
81      }
82
```

```
83      if (status == CERT_STATUS_EXPIRED &&
           !state->policy.allow_expired_renewal) {
84          return -1;  /* Cannot renew expired certificate */
85      }
86
87      /* Check retry timing */
88      if (state->renewal_pending && now <
           state->next_retry_time) {
89          return 0;  /* Waiting for retry interval */
90      }
91
92      /* Attempt renewal */
93      state->attempts++;
94
95      if (state->attempts > state->policy.max_retries) {
96          /* Exceeded retry limit */
97          log_error("Certificate renewal failed after %d
               attempts",
98                    state->attempts);
99          return -2;
100     }
101
102     /* Perform enrollment */
103     int result = enroll_certificate_est(
104         est_config,
105         state->target_cert->algorithm,
106         state->new_cert, &state->new_cert_len,
107         state->new_key, &state->new_key_len
108     );
109
110     if (result != 0) {
111         /* Schedule retry */
112         state->next_retry_time = now +
113             (state->policy.retry_interval_hours * 3600);
114         state->renewal_pending = true;
115         log_warning("Certificate renewal failed, retry
               scheduled");
116         return 0;
117     }
118
119     /* Success - install new certificate */
```

```
    result = install_renewed_certificate(
        state->target_cert,
        state->new_cert, state->new_cert_len,
        state->new_key, state->new_key_len
    );

    if (result == 0) {
        state->renewal_pending = false;
        state->attempts = 0;
        log_info("Certificate renewed successfully");
    }

    /* Clear sensitive data */
    secure_zero(state->new_key, sizeof(state->new_key));

    return result;
}

/* Hybrid certificate renewal - get both classical and PQC certs */
int renew_hybrid_certificates(const est_config_t *est_config) {
    uint8_t ecdsa_cert[2048], mldsa_cert[8192];
    uint8_t ecdsa_key[256], mldsa_key[4096];
    size_t ecdsa_cert_len, mldsa_cert_len;
    size_t ecdsa_key_len, mldsa_key_len;

    /* Enroll ECDSA certificate */
    int ret = enroll_certificate_est(est_config, SIG_ALG_ECDSA_P256,
                                     ecdsa_cert, &ecdsa_cert_len,
                                     ecdsa_key, &ecdsa_key_len);
    if (ret != 0) {
        return -1;
    }

    /* Enroll ML-DSA certificate */
    ret = enroll_certificate_est(est_config, SIG_ALG_MLDSA44,
```

```
                                        mldsa_cert,
                                           &mldsa_cert_len,
                                        mldsa_key, &mldsa_key_len);
    if (ret != 0) {
        secure_zero(ecdsa_key, sizeof(ecdsa_key));
        return -2;
    }

    /* Store both certificates */
    store_certificate(get_cert_store(),
       CERT_TYPE_OPERATIONAL,
                      ecdsa_cert, ecdsa_cert_len,
                      ecdsa_key, ecdsa_key_len);

    store_certificate(get_cert_store(),
       CERT_TYPE_DEVICE_IDENTITY,
                      mldsa_cert, mldsa_cert_len,
                      mldsa_key, mldsa_key_len);

    secure_zero(ecdsa_key, sizeof(ecdsa_key));
    secure_zero(mldsa_key, sizeof(mldsa_key));

    return 0;
}
```

Listing 9.19: Automated Certificate Renewal System

9.7.4 Certificate Revocation for IoT

Traditional CRL (Certificate Revocation List) and OCSP (Online Certificate Status Protocol) approaches are often impractical for constrained IoT devices. Alternative approaches are needed:

```
/*
 * Lightweight Certificate Revocation using Bloom Filters
 * Suitable for memory-constrained IoT devices
 */

#include <stdint.h>
#include <stdbool.h>

/* Bloom filter configuration */
```

Table 9.15: Revocation Methods for IoT Devices

Method	Advantages	Disadvantages	IoT Suitability
Full CRL	Complete, standard	Large downloads, frequent updates	Poor
Delta CRL	Smaller updates	Still requires periodic checks	Limited
OCSP	Real-time status	Requires connectivity per-check	Limited
OCSP Stapling	Server provides status	Not applicable for device certs	N/A
Short-lived certs	No revocation needed	Frequent renewal overhead	Good
CRL lite/bloom filter	Compact representation	False positives possible	Good
Push revocation	Immediate notification	Requires push infrastructure	Excellent

```
#define BLOOM_SIZE_BITS (8 * 1024)  /* 8 Kbit = 1 KB */
#define BLOOM_SIZE_BYTES (BLOOM_SIZE_BITS / 8)
#define BLOOM_NUM_HASHES 7

typedef struct {
    uint8_t bits[BLOOM_SIZE_BYTES];
    uint32_t version;              /* CRL version */
    uint32_t timestamp;            /* Last update time */
    uint32_t num_revoked;          /* Approximate count */
} bloom_crl_t;

/* Hash functions for bloom filter */
static uint32_t bloom_hash(const uint8_t *data, size_t len,
   uint32_t seed) {
    /* MurmurHash3 32-bit */
    uint32_t h = seed;
    const uint32_t c1 = 0xcc9e2d51;
    const uint32_t c2 = 0x1b873593;

    const uint8_t *tail = data + (len / 4) * 4;

    for (size_t i = 0; i < len / 4; i++) {
        uint32_t k = ((uint32_t *)data)[i];
        k *= c1;
        k = (k << 15) | (k >> 17);
```

```
34          k *= c2;
35          h ^= k;
36          h = (h << 13) | (h >> 19);
37          h = h * 5 + 0xe6546b64;
38      }
39
40      uint32_t k = 0;
41      switch (len & 3) {
42          case 3: k ^= tail[2] << 16;
43          case 2: k ^= tail[1] << 8;
44          case 1: k ^= tail[0];
45                  k *= c1;
46                  k = (k << 15) | (k >> 17);
47                  k *= c2;
48                  h ^= k;
49      }
50
51      h ^= len;
52      h ^= h >> 16;
53      h *= 0x85ebca6b;
54      h ^= h >> 13;
55      h *= 0xc2b2ae35;
56      h ^= h >> 16;
57
58      return h;
59  }
60
61  /* Check if certificate might be revoked */
62  bool bloom_check_revoked(const bloom_crl_t *bloom,
63                           const uint8_t *cert_serial,
64                           size_t serial_len) {
65      for (int i = 0; i < BLOOM_NUM_HASHES; i++) {
66          uint32_t hash = bloom_hash(cert_serial, serial_len,
               i * 0x9E3779B9);
67          uint32_t bit_index = hash % BLOOM_SIZE_BITS;
68          uint32_t byte_index = bit_index / 8;
69          uint8_t bit_mask = 1 << (bit_index % 8);
70
71          if (!(bloom->bits[byte_index] & bit_mask)) {
72              return false;  /* Definitely not revoked */
73          }
```

```
74      }
75
76      return true;  /* Possibly revoked (may be false
           positive) */
77  }
78
79  /* Update bloom filter from server */
80  int update_bloom_crl(bloom_crl_t *bloom, const char
       *server_url) {
81      uint8_t response[2048];
82      size_t response_len;
83
84      /* Download compressed bloom filter */
85      if (https_get(server_url, response, &response_len) != 0)
           {
86          return -1;
87      }
88
89      /* Verify signature on bloom filter update */
90      bloom_crl_update_t *update = (bloom_crl_update_t
           *)response;
91
92      if (verify_signature(update->data, update->data_len,
93                           update->signature, update->sig_len,
94                           get_crl_signing_key(), CRL_KEY_LEN,
95                           SIG_ALG_MLDSA44) != 0) {
96          return -2;
97      }
98
99      /* Check version is newer */
100     if (update->version <= bloom->version) {
101         return 0;  /* Already up to date */
102     }
103
104     /* Decompress and apply update */
105     if (decompress_bloom(bloom->bits,
           update->compressed_bits,
106                          update->compressed_len) != 0) {
107         return -3;
108     }
109
```

```
    bloom->version = update->version;
    bloom->timestamp = update->timestamp;
    bloom->num_revoked = update->num_revoked;

    return 0;
}

/* Short-lived certificate approach - no revocation needed */
typedef struct {
    uint32_t cert_lifetime_hours;  /* Typically 24-72 hours */
    uint32_t renewal_threshold;    /* Hours before expiry to renew */
} short_lived_policy_t;

static const short_lived_policy_t SHORT_LIVED_DEFAULT = {
    .cert_lifetime_hours = 24,
    .renewal_threshold = 6
};

/* Short-lived certificates eliminate revocation overhead */
bool should_use_short_lived_certs(const device_profile_t *profile) {
    /* Recommended when:
     * - Device has reliable connectivity
     * - Enrollment server is available
     * - Security requirements are high
     * - Certificate storage is limited
     */
    return (profile->connectivity == CONN_ALWAYS_ON) &&
           (profile->security_level >= 3) &&
           (profile->cert_storage_kb < 16);
}
```

Listing 9.20: Bloom Filter Based Revocation Checking

9.7.5 Certificate Chain Optimization

```
/*
 * Certificate Chain Optimization for Bandwidth-Constrained IoT
 */
```

```
4
5  #include <stdint.h>
6  #include <stdbool.h>
7
8  /* Certificate reference (for caching) */
9  typedef struct {
10     uint8_t issuer_key_hash[32];   /* SHA-256 of issuer's
          public key */
11     uint8_t serial_number[20];      /* Certificate serial
          number */
12 } cert_ref_t;
13
14 /* Compressed certificate chain format */
15 typedef struct __attribute__((packed)) {
16     uint8_t flags;                 /* Compression flags */
17     uint8_t num_certs;             /* Number of certificates
          */
18     uint16_t total_len;            /* Total compressed
          length */
19     /* Followed by: compressed certificate data */
20 } compressed_chain_t;
21
22 #define CHAIN_FLAG_ZLIB       0x01  /* ZLIB compression */
23 #define CHAIN_FLAG_CACHED     0x02  /* Some certs referenced
      by hash */
24 #define CHAIN_FLAG_TRUNCATED  0x04  /* Root CA omitted
      (well-known) */
25
26 /* Certificate cache for frequently seen issuers */
27 #define CERT_CACHE_SIZE 4
28
29 typedef struct {
30     cert_ref_t ref;
31     uint8_t cert_data[4096];
32     size_t cert_len;
33     uint32_t last_used;
34     bool valid;
35 } cached_cert_t;
36
37 static cached_cert_t g_cert_cache[CERT_CACHE_SIZE];
38
```

```
/* Check if certificate is in cache */
const uint8_t *cache_lookup(const cert_ref_t *ref, size_t
    *len) {
    for (int i = 0; i < CERT_CACHE_SIZE; i++) {
        if (g_cert_cache[i].valid &&
            memcmp(&g_cert_cache[i].ref, ref,
                sizeof(cert_ref_t)) == 0) {
            g_cert_cache[i].last_used = get_current_time();
            *len = g_cert_cache[i].cert_len;
            return g_cert_cache[i].cert_data;
        }
    }
    return NULL;
}

/* Add certificate to cache (LRU eviction) */
void cache_add(const cert_ref_t *ref, const uint8_t *cert,
    size_t len) {
    /* Find oldest entry */
    int oldest_idx = 0;
    uint32_t oldest_time = UINT32_MAX;

    for (int i = 0; i < CERT_CACHE_SIZE; i++) {
        if (!g_cert_cache[i].valid) {
            oldest_idx = i;
            break;
        }
        if (g_cert_cache[i].last_used < oldest_time) {
            oldest_time = g_cert_cache[i].last_used;
            oldest_idx = i;
        }
    }

    /* Store in cache */
    memcpy(&g_cert_cache[oldest_idx].ref, ref,
        sizeof(cert_ref_t));
    memcpy(g_cert_cache[oldest_idx].cert_data, cert, len);
    g_cert_cache[oldest_idx].cert_len = len;
    g_cert_cache[oldest_idx].last_used = get_current_time();
    g_cert_cache[oldest_idx].valid = true;
}
```

```
76
77 /* Decompress and reconstruct certificate chain */
78 int decompress_cert_chain(const compressed_chain_t
      *compressed,
79                           uint8_t *chain_out, size_t
                                *chain_len,
80                           size_t max_len) {
81     const uint8_t *src = (const uint8_t *)(compressed + 1);
82     uint8_t *dst = chain_out;
83     size_t dst_offset = 0;
84
85     for (int i = 0; i < compressed->num_certs; i++) {
86         if (compressed->flags & CHAIN_FLAG_CACHED) {
87             /* Check if this entry is a cache reference */
88             uint8_t entry_type = *src++;
89
90             if (entry_type == 0xFF) {
91                 /* Cache reference */
92                 cert_ref_t ref;
93                 memcpy(&ref, src, sizeof(cert_ref_t));
94                 src += sizeof(cert_ref_t);
95
96                 size_t cached_len;
97                 const uint8_t *cached = cache_lookup(&ref,
                       &cached_len);
98
99                 if (!cached) {
100                     return -1;  /* Cache miss - cannot
                           decompress */
101                 }
102
103                 if (dst_offset + cached_len > max_len) {
104                     return -2;  /* Buffer overflow */
105                 }
106
107                 memcpy(dst + dst_offset, cached, cached_len);
108                 dst_offset += cached_len;
109                 continue;
110             }
111
112             src--;  /* Wasn't a cache marker, rewind */
```

```
113         }
114
115         /* Read certificate length */
116         uint16_t cert_len = *(uint16_t *)src;
117         src += 2;
118
119         /* Decompress if needed */
120         if (compressed->flags & CHAIN_FLAG_ZLIB) {
121             size_t decomp_len = max_len - dst_offset;
122             if (zlib_decompress(dst + dst_offset,
                   &decomp_len,
123                                 src, cert_len) != 0) {
124                 return -3;
125             }
126             dst_offset += decomp_len;
127             src += cert_len;
128         } else {
129             if (dst_offset + cert_len > max_len) {
130                 return -2;
131             }
132             memcpy(dst + dst_offset, src, cert_len);
133             dst_offset += cert_len;
134             src += cert_len;
135         }
136     }
137
138     *chain_len = dst_offset;
139     return 0;
140 }
141
142 /* TLS certificate compression extension (RFC 8879) */
143 typedef enum {
144     CERT_COMPRESS_ZLIB = 1,
145     CERT_COMPRESS_BROTLI = 2,
146     CERT_COMPRESS_ZSTD = 3
147 } tls_cert_compression_t;
148
149 /* Advertise compression support in ClientHello */
150 int tls_add_cert_compression_ext(uint8_t *ext_out, size_t
       *ext_len) {
151     /* Extension format:
```

```
152         * uint16 algorithms<2..2^8-2>
153         */
154        ext_out[0] = 0x00;  /* Extension type high byte */
155        ext_out[1] = 0x1B;  /* compress_certificate extension
               (27) */
156        ext_out[2] = 0x00;  /* Length high byte */
157        ext_out[3] = 0x03;  /* Length: 3 bytes */
158        ext_out[4] = 0x02;  /* Algorithm list length */
159        ext_out[5] = 0x00;  /* ZLIB high byte */
160        ext_out[6] = 0x01;  /* ZLIB = 1 */
161
162        *ext_len = 7;
163        return 0;
164    }
```

Listing 9.21: Certificate Chain Compression and Caching

Table 9.16: Certificate Lifecycle Best Practices Summary

Practice	Recommendation
Algorithm choice	Falcon-512 for size-constrained devices; ML-DSA-44 for general use
Validity period	1-3 years for device identity; 24-72 hours for short-lived operational
Renewal timing	Begin renewal at 10-20% remaining lifetime
Chain length	Minimize to 2-3 certificates; use cached intermediates
Revocation	Bloom filter CRL or short-lived certificates for IoT
Compression	Enable TLS certificate compression (RFC 8879)
Hybrid approach	Maintain both classical and PQC certificates during transition
Key protection	Store private keys in secure element or PUF-wrapped

9.8 Compliance Frameworks and Standards

Organizations deploying PQC in IoT systems must navigate an evolving landscape of compliance requirements, government mandates, and industry standards. This section provides guidance on meeting regulatory requirements while implementing post-quantum cryptography.

9.8.1 CNSA 2.0 (Commercial National Security Algorithm Suite)

The U.S. National Security Agency (NSA) published CNSA 2.0 in September 2022, mandating the transition to quantum-resistant algorithms for National Security Systems (NSS).

Table 9.17: CNSA 2.0 Algorithm Requirements

Function	CNSA 2.0 Algorithm	Parameters	Deadline
Software/Firmware Signing	ML-DSA	ML-DSA-87	2025
Key Establishment	ML-KEM	ML-KEM-1024	2025
Web/Cloud Authentication	ML-DSA	ML-DSA-87	2025
Traditional Networking	ML-KEM + ML-DSA	Level 5	2026
Operating Systems	ML-DSA	ML-DSA-87	2027
Custom Applications	ML-KEM + ML-DSA	Level 5	2027
Legacy Equipment	Hybrid or exclusion	Case-by-case	2033

Warning: CNSA 2.0 requires **NIST Security Level 5** (ML-KEM-1024, ML-DSA-87) for all National Security Systems. This is more stringent than typical commercial requirements and significantly impacts resource-constrained IoT devices.

```
/*
 * CNSA 2.0 Compliance Verification Module
 * Validates cryptographic configurations against NSA
    requirements
 */

#include <stdint.h>
#include <stdbool.h>
#include <time.h>

/* CNSA 2.0 compliance levels */
typedef enum {
    CNSA_NOT_COMPLIANT = 0,
    CNSA_TRANSITIONAL = 1,      /* Hybrid mode acceptable */
    CNSA_COMPLIANT = 2,         /* Full CNSA 2.0 compliance */
    CNSA_PREFERRED = 3          /* Exceeds requirements */
} cnsa_compliance_t;

/* Algorithm compliance status */
typedef struct {
```

```
20      bool kem_compliant;
21      bool dsa_compliant;
22      bool hash_compliant;
23      bool symmetric_compliant;
24      const char *kem_algorithm;
25      const char *dsa_algorithm;
26      cnsa_compliance_t overall;
27      char deficiencies[512];
28  } compliance_report_t;
29
30  /* CNSA 2.0 required parameters */
31  static const struct {
32      const char *name;
33      int min_security_level;
34      bool quantum_resistant;
35  } cnsa2_requirements[] = {
36      {"ML-KEM-1024", 5, true},
37      {"ML-DSA-87", 5, true},
38      {"SHA-384", 3, false},      /* Hash minimum */
39      {"SHA-512", 5, false},      /* Hash preferred */
40      {"AES-256", 5, false},      /* Symmetric required */
41  };
42
43  /* Check KEM compliance */
44  static bool check_kem_cnsa2(const char *algorithm, int
        security_level) {
45      if (strcmp(algorithm, "ML-KEM-1024") == 0 &&
            security_level >= 5) {
46          return true;
47      }
48      /* Hybrid acceptable during transition (until 2033) */
49      if (strstr(algorithm, "ML-KEM-1024") != NULL &&
50          get_current_year() < 2033) {
51          return true;  /* Hybrid with ML-KEM-1024 */
52      }
53      return false;
54  }
55
56  /* Check DSA compliance */
57  static bool check_dsa_cnsa2(const char *algorithm, int
        security_level) {
```

```
58      if (strcmp(algorithm, "ML-DSA-87") == 0 &&
           security_level >= 5) {
59          return true;
60      }
61      /* SLH-DSA also acceptable */
62      if (strstr(algorithm, "SLH-DSA") != NULL &&
           security_level >= 5) {
63          return true;
64      }
65      return false;
66  }
67
68  /* Generate compliance report */
69  compliance_report_t check_cnsa2_compliance(
70      const char *kem_alg, int kem_level,
71      const char *dsa_alg, int dsa_level,
72      const char *hash_alg,
73      const char *symmetric_alg) {
74
75      compliance_report_t report = {0};
76      report.kem_algorithm = kem_alg;
77      report.dsa_algorithm = dsa_alg;
78      report.deficiencies[0] = '\0';
79
80      /* Check KEM */
81      report.kem_compliant = check_kem_cnsa2(kem_alg,
           kem_level);
82      if (!report.kem_compliant) {
83          strcat(report.deficiencies, "KEM: Requires
               ML-KEM-1024. ");
84      }
85
86      /* Check DSA */
87      report.dsa_compliant = check_dsa_cnsa2(dsa_alg,
           dsa_level);
88      if (!report.dsa_compliant) {
89          strcat(report.deficiencies, "DSA: Requires ML-DSA-87
               or SLH-DSA. ");
90      }
91
92      /* Check hash */
```

```
93      report.hash_compliant = (strcmp(hash_alg, "SHA-384") ==
            0 ||
94                               strcmp(hash_alg, "SHA-512") ==
                                     0 ||
95                               strcmp(hash_alg, "SHA3-384") ==
                                     0 ||
96                               strcmp(hash_alg, "SHA3-512") ==
                                     0);
97      if (!report.hash_compliant) {
98          strcat(report.deficiencies, "Hash: Requires SHA-384
                or higher. ");
99      }
100
101     /* Check symmetric */
102     report.symmetric_compliant = (strcmp(symmetric_alg,
            "AES-256") == 0);
103     if (!report.symmetric_compliant) {
104         strcat(report.deficiencies, "Symmetric: Requires
                AES-256. ");
105     }
106
107     /* Determine overall compliance */
108     if (report.kem_compliant && report.dsa_compliant &&
109         report.hash_compliant && report.symmetric_compliant)
                {
110         report.overall = CNSA_COMPLIANT;
111     } else if (strstr(kem_alg, "ML-KEM") || strstr(dsa_alg,
            "ML-DSA")) {
112         report.overall = CNSA_TRANSITIONAL;
113     } else {
114         report.overall = CNSA_NOT_COMPLIANT;
115     }
116
117     return report;
118 }
119
120 /* Print compliance report */
121 void print_compliance_report(const compliance_report_t
        *report) {
122     printf("=== CNSA 2.0 Compliance Report ===\n");
123     printf("KEM Algorithm: %s [%s]\n",
```

```
124             report->kem_algorithm,
125             report->kem_compliant ? "COMPLIANT" :
                   "NON-COMPLIANT");
126     printf("DSA Algorithm: %s [%s]\n",
127             report->dsa_algorithm,
128             report->dsa_compliant ? "COMPLIANT" :
                   "NON-COMPLIANT");
129     printf("Hash: [%s]\n",
130             report->hash_compliant ? "COMPLIANT" :
                   "NON-COMPLIANT");
131     printf("Symmetric: [%s]\n",
132             report->symmetric_compliant ? "COMPLIANT" :
                   "NON-COMPLIANT");
133     printf("\nOverall Status: ");
134     switch (report->overall) {
135         case CNSA_COMPLIANT:
136             printf("FULLY COMPLIANT\n");
137             break;
138         case CNSA_TRANSITIONAL:
139             printf("TRANSITIONAL (Hybrid acceptable until
                       2033)\n");
140             break;
141         default:
142             printf("NOT COMPLIANT\n");
143             printf("Deficiencies: %s\n",
                       report->deficiencies);
144     }
145 }
```

Listing 9.22: CNSA 2.0 Compliance Checker

9.8.2 NIST Post-Quantum Cryptography Standards

NIST finalized the first set of PQC standards in August 2024:

9.8.3 Industry-Specific Compliance

Healthcare (HIPAA/HITECH)

Financial Services (PCI DSS)

```
1 #!/usr/bin/env python3
2 """
```

Table 9.18: NIST PQC Standards (August 2024)

Standard	Algorithm	Type	Status
FIPS 203	ML-KEM (Kyber)	KEM	Final Standard
FIPS 204	ML-DSA (Dilithium)	Signature	Final Standard
FIPS 205	SLH-DSA (SPHINCS+)	Signature	Final Standard
–	FN-DSA (Falcon)	Signature	Draft (expected 2025)
–	HQC	KEM	Round 4 selection
–	BIKE	KEM	Round 4 selection

Table 9.19: PQC Considerations for Healthcare IoT

Requirement	HIPAA Mandate	PQC Implementation
Data at rest	Encryption required	AES-256 with PQC key wrap
Data in transit	Encryption required	TLS 1.3 with ML-KEM-768
Access control	Unique user ID	PQC-signed certificates
Audit controls	Activity logging	Signed audit logs (ML-DSA)
Integrity	Data integrity verification	HMAC-SHA-256 or PQC signatures
PHI retention	6+ years minimum	HNDL requires PQC now

```
"""
PCI DSS Post-Quantum Compliance Assessment Tool
Evaluates payment card industry compliance for PQC migration
"""

from dataclasses import dataclass
from typing import List, Dict
from enum import Enum

class PCIRequirement(Enum):
    REQ_3 = "Protect stored cardholder data"
    REQ_4 = "Encrypt transmission of cardholder data"
    REQ_6 = "Develop and maintain secure systems"
    REQ_8 = "Identify and authenticate access"
    REQ_10 = "Track and monitor access"
    REQ_12 = "Maintain security policy"

@dataclass
class CryptoAsset:
    name: str
    asset_type: str  # 'key', 'certificate', 'connection'
    algorithm: str
```

```
    key_size: int
    quantum_safe: bool
    data_sensitivity: str  # 'CHD', 'SAD', 'other'
    retention_years: int

@dataclass
class ComplianceGap:
    requirement: PCIRequirement
    description: str
    severity: str  # 'critical', 'high', 'medium', 'low'
    remediation: str
    deadline: str

def assess_pci_pqc_compliance(assets: List[CryptoAsset]) ->
   List[ComplianceGap]:
    """Assess PCI DSS compliance gaps for PQC migration."""
    gaps = []

    for asset in assets:
        # Check cardholder data protection (Req 3)
        if asset.data_sensitivity in ('CHD', 'SAD'):
            if not asset.quantum_safe and
               asset.retention_years > 5:
                gaps.append(ComplianceGap(
                    requirement=PCIRequirement.REQ_3,
                    description=f"{asset.name}: CHD
                       encrypted with {asset.algorithm} "
                            f"retained
                               {asset.retention_years}
                               years",
                    severity='high',
                    remediation="Migrate to ML-KEM-768 or
                       higher for key encryption",
                    deadline="2026"
                ))

        # Check transmission encryption (Req 4)
        if asset.asset_type == 'connection':
            if asset.algorithm in ('RSA', 'ECDHE') and not
               asset.quantum_safe:
                gaps.append(ComplianceGap(
```

```
58                      requirement=PCIRequirement.REQ_4,
59                      description=f"{asset.name}: Using
                            {asset.algorithm} for "
60                                  "key exchange",
61                      severity='medium',
62                      remediation="Deploy hybrid TLS with
                            X25519+ML-KEM-768",
63                      deadline="2027"
64                  ))
65
66          # Check authentication (Req 8)
67          if asset.asset_type == 'certificate':
68              if asset.algorithm in ('RSA', 'ECDSA') and not
                    asset.quantum_safe:
69                  gaps.append(ComplianceGap(
70                      requirement=PCIRequirement.REQ_8,
71                      description=f"{asset.name}: Certificate
                            using {asset.algorithm}",
72                      severity='medium',
73                      remediation="Issue hybrid or pure PQC
                            certificates (ML-DSA)",
74                      deadline="2028"
75                  ))
76
77      return gaps
78
79  def generate_pci_pqc_roadmap(gaps: List[ComplianceGap]) ->
        Dict:
80      """Generate migration roadmap from compliance gaps."""
81      roadmap = {
82          'immediate': [],  # < 1 year
83          'short_term': [], # 1-2 years
84          'medium_term': [], # 2-3 years
85          'long_term': []   # 3+ years
86      }
87
88      for gap in gaps:
89          deadline_year = int(gap.deadline)
90          current_year = 2025
91          years_remaining = deadline_year - current_year
92
```

```
93          if years_remaining <= 1:
94              roadmap['immediate'].append(gap)
95          elif years_remaining <= 2:
96              roadmap['short_term'].append(gap)
97          elif years_remaining <= 3:
98              roadmap['medium_term'].append(gap)
99          else:
100             roadmap['long_term'].append(gap)
101
102     return roadmap
103
104 # Example usage
105 if __name__ == '__main__':
106     # Define crypto assets in payment system
107     assets = [
108         CryptoAsset(
109             name="Card vault encryption key",
110             asset_type="key",
111             algorithm="AES-256",
112             key_size=256,
113             quantum_safe=False,  # Key wrapped with RSA
114             data_sensitivity="CHD",
115             retention_years=7
116         ),
117         CryptoAsset(
118             name="Payment gateway TLS",
119             asset_type="connection",
120             algorithm="ECDHE",
121             key_size=256,
122             quantum_safe=False,
123             data_sensitivity="CHD",
124             retention_years=0
125         ),
126         CryptoAsset(
127             name="Merchant certificate",
128             asset_type="certificate",
129             algorithm="RSA",
130             key_size=2048,
131             quantum_safe=False,
132             data_sensitivity="other",
133             retention_years=2
```

```
        ),
    ]

    gaps = assess_pci_pqc_compliance(assets)
    roadmap = generate_pci_pqc_roadmap(gaps)

    print("=== PCI DSS PQC Compliance Assessment ===\n")
    for gap in gaps:
        print(f"[{gap.severity.upper()}]
           {gap.requirement.value}")
        print(f"  Issue: {gap.description}")
        print(f"  Action: {gap.remediation}")
        print(f"  Deadline: {gap.deadline}\n")
```

Listing 9.23: PCI DSS PQC Compliance Assessment

Automotive (ISO/SAE 21434)

Table 9.20: Automotive Cybersecurity PQC Requirements

UN R155 Requirement	PQC Relevance	Recommendation
Secure boot	Firmware authentication	ML-DSA-65 or Falcon-512
Secure OTA	Update integrity	Hybrid signatures
V2X communication	Message authentication	SLH-DSA-128f (stateless)
Key management	Long-term key protection	ML-KEM-768/1024
ECU authentication	Inter-ECU trust	Lightweight PQC (ML-KEM-512)

9.8.4 FIPS 140-3 Validation

For IoT devices requiring FIPS 140-3 validation with PQC:

Table 9.21: FIPS 140-3 PQC Validation Status (December 2024)

Algorithm	CAVP Testing	CMVP Modules	Availability
ML-KEM	Available	In progress	Q1 2025
ML-DSA	Available	In progress	Q1 2025
SLH-DSA	Available	In progress	Q2 2025
Falcon	Pending	Not yet	2025-2026

```
/*
 * FIPS 140-3 Mode Configuration for PQC Operations
 * Ensures compliance with validated module requirements
```

```
 */

#include <stdint.h>
#include <stdbool.h>

/* FIPS mode status */
typedef struct {
    bool fips_mode_enabled;
    bool self_test_passed;
    bool integrity_verified;
    uint32_t module_version;
    char validation_cert[32];  /* CMVP certificate number */
} fips_status_t;

static fips_status_t g_fips_status = {0};

/* FIPS-approved algorithms in PQC context */
typedef enum {
    FIPS_ALG_MLKEM_512 = 1,
    FIPS_ALG_MLKEM_768 = 2,
    FIPS_ALG_MLKEM_1024 = 3,
    FIPS_ALG_MLDSA_44 = 4,
    FIPS_ALG_MLDSA_65 = 5,
    FIPS_ALG_MLDSA_87 = 6,
    FIPS_ALG_SLHDSA_SHA2_128F = 7,
    FIPS_ALG_SLHDSA_SHA2_128S = 8,
    FIPS_ALG_SLHDSA_SHA2_192F = 9,
    FIPS_ALG_SLHDSA_SHA2_192S = 10,
    FIPS_ALG_SLHDSA_SHA2_256F = 11,
    FIPS_ALG_SLHDSA_SHA2_256S = 12,
    /* Non-FIPS approved (for reference) */
    NON_FIPS_FALCON_512 = 0x80,
    NON_FIPS_FALCON_1024 = 0x81,
} fips_algorithm_t;

/* Check if algorithm is FIPS-approved */
bool is_fips_approved(fips_algorithm_t alg) {
    return (alg >= FIPS_ALG_MLKEM_512 && alg <=
       FIPS_ALG_SLHDSA_SHA2_256S);
}

```

```
/* Power-on self-test for PQC algorithms */
int fips_pqc_self_test(void) {
    int result = 0;

    /* ML-KEM Known Answer Test (KAT) */
    result |= mlkem_kat_test();

    /* ML-DSA Known Answer Test */
    result |= mldsa_kat_test();

    /* SLH-DSA Known Answer Test */
    result |= slhdsa_kat_test();

    /* Pairwise consistency test for key generation */
    result |= pqc_pairwise_consistency_test();

    /* DRBG health test */
    result |= drbg_health_test();

    g_fips_status.self_test_passed = (result == 0);

    return result;
}

/* Module integrity check */
int fips_integrity_check(void) {
    /* Compute HMAC-SHA-256 over module code */
    uint8_t computed_mac[32];
    extern const uint8_t __module_start[];
    extern const uint8_t __module_end[];
    extern const uint8_t __expected_mac[32];

    size_t module_size = __module_end - __module_start;
    hmac_sha256(computed_mac, FIPS_INTEGRITY_KEY, 32,
                __module_start, module_size);

    g_fips_status.integrity_verified =
        (secure_memcmp(computed_mac, __expected_mac, 32) ==
           0);

    return g_fips_status.integrity_verified ? 0 : -1;
```

```
}

/* Initialize FIPS mode */
int fips_init(void) {
    /* Step 1: Integrity check */
    if (fips_integrity_check() != 0) {
        return -1;
    }

    /* Step 2: Power-on self-tests */
    if (fips_pqc_self_test() != 0) {
        return -2;
    }

    /* Step 3: Enable FIPS mode */
    g_fips_status.fips_mode_enabled = true;
    g_fips_status.module_version = FIPS_MODULE_VERSION;
    strcpy(g_fips_status.validation_cert, "Pending-2025");

    return 0;
}

/* FIPS-compliant key generation wrapper */
int fips_keygen(fips_algorithm_t alg, uint8_t *pk, uint8_t
   *sk) {
    /* Verify FIPS mode is active */
    if (!g_fips_status.fips_mode_enabled) {
        return -1;
    }

    /* Verify algorithm is approved */
    if (!is_fips_approved(alg)) {
        return -2;
    }

    /* Continuous RNG test */
    if (drbg_continuous_test() != 0) {
        return -3;
    }

    /* Generate keys based on algorithm */
```

```
124     switch (alg) {
125         case FIPS_ALG_MLKEM_768:
126             return mlkem768_keypair(pk, sk);
127         case FIPS_ALG_MLDSA_65:
128             return mldsa65_keypair(pk, sk);
129         /* Add other algorithms */
130         default:
131             return -4;
132     }
133 }
```

Listing 9.24: FIPS Mode Configuration for PQC

9.8.5 European Regulations (eIDAS 2.0, Cyber Resilience Act)

Table 9.22: European PQC Regulatory Landscape

Regulation	Scope	PQC Requirement	Timeline
eIDAS 2.0	Digital identity, signatures	PQC for long-term signatures	2027+
Cyber Resilience Act	IoT product security	Security updates for lifecycle	2027
NIS2 Directive	Critical infrastructure	Risk-based crypto selection	2024
GDPR	Personal data protection	Encryption best practices	Ongoing

9.8.6 Compliance Implementation Checklist

Note: Compliance is not just about using the right algorithms—it requires documented processes, audit trails, and evidence of due diligence. Start your compliance journey with a comprehensive cryptographic inventory and risk assessment.

9.9 Chapter Summary

This chapter has provided a comprehensive examination of security analysis methodologies and best practices for post-quantum cryptography deployments in IoT environments. The key topics covered include:

Threat Modeling and Risk Assessment:

- The STRIDE framework adapted for PQC threat analysis
- HNDL (Harvest Now, Decrypt Later) attack scenarios and their implications

Table 9.23: PQC Compliance Implementation Checklist

#	Category	Action Item
1	Inventory	Catalog all cryptographic assets and their quantum vulnerability
2	Risk Assessment	Evaluate HNDL risk based on data sensitivity and retention
3	Algorithm Selection	Choose NIST-approved algorithms matching security requirements
4	Compliance Mapping	Map regulatory requirements to PQC implementations
5	Hybrid Strategy	Plan hybrid classical+PQC during transition period
6	Testing	Implement Known Answer Tests for FIPS compliance
7	Key Management	Update key lifecycle procedures for larger PQC keys
8	Certificate Planning	Plan PKI migration timeline and certificate validity
9	Documentation	Maintain cryptographic module documentation
10	Audit Trail	Log all cryptographic operations for compliance audits
11	Training	Train development and operations teams on PQC
12	Vendor Assessment	Evaluate third-party PQC library compliance claims

- Risk quantification methodologies based on data sensitivity and retention periods
- Threat modeling specific to IoT deployment scenarios

Side-Channel Analysis and Countermeasures:

- Timing attacks on lattice-based algorithms and constant-time implementations
- Power analysis (SPA/DPA) vulnerabilities in polynomial arithmetic
- Electromagnetic emanation analysis and shielding techniques
- Implementation-level protections including masking and shuffling

Security Parameter Selection:

- NIST security levels (I–V) and their classical/quantum equivalencies
- Algorithm selection criteria based on security requirements
- Parameter choices for different IoT use cases
- Long-term security considerations for 15–20 year device lifecycles

Key Management Best Practices:

- Complete key lifecycle management from generation to destruction
- Secure key storage strategies for constrained devices
- Key rotation and rekeying considerations for larger PQC keys
- Hardware security integration with TEEs and secure elements

Secure System Design:

- Secure boot chain implementation with PQC signature verification
- TrustZone-M integration for cryptographic isolation
- OTA firmware update architectures with A/B scheme support
- Certificate lifecycle management and PKI considerations

Compliance and Standards:

- CNSA 2.0 suite requirements and migration timeline
- NIST post-quantum cryptography standards (FIPS 203, 204, 205)
- FIPS 140-3 validation requirements for cryptographic modules

- Industry-specific compliance (automotive, medical, industrial)
- European regulatory landscape (eIDAS 2.0, Cyber Resilience Act)

The overarching message of this chapter is that security in PQC deployments extends far beyond algorithm selection. A holistic approach encompassing implementation security, key management, system architecture, and regulatory compliance is essential for robust quantum-resistant IoT systems.

Chapter 10

Migration Roadmaps and Cost Analysis

The transition from classical to post-quantum cryptography represents one of the most significant cryptographic migrations in history. This chapter provides practical roadmaps for IoT deployments, comprehensive cost analysis frameworks, and risk assessment methodologies to guide organizations through this transition.

10.1 Migration Timeline Overview

10.1.1 Industry Consensus Timeline

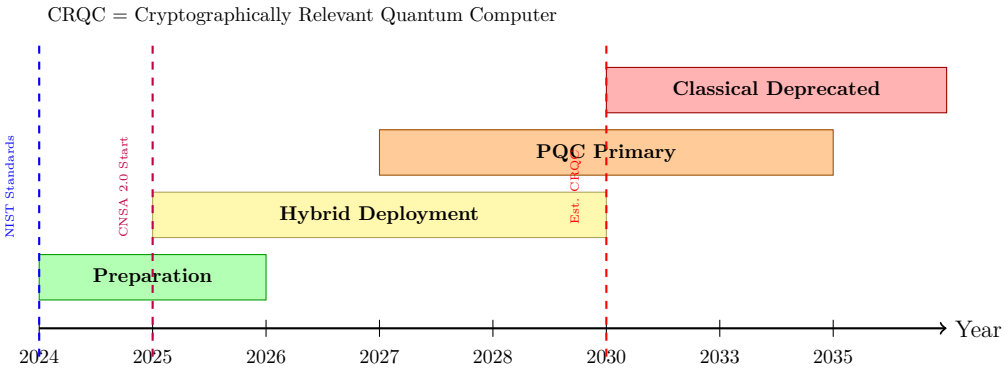


Figure 10.1: Post-Quantum Cryptography Migration Timeline

Table 10.1: Migration Phase Definitions

Phase	Period	Objectives
Preparation	2024–2026	Inventory, risk assessment, pilot projects, training
Hybrid Deployment	2025–2030	Deploy hybrid classical+PQC, maintain interoperability
PQC Primary	2027–2035	PQC as default, classical for legacy only
Classical Deprecation	2030–2035+	Remove classical crypto, full PQC operation

Table 10.2: IoT Migration Challenges vs. Enterprise IT

Challenge	Enterprise IT	IoT Devices
Update capability	Regular patching assumed	Many devices unpatchable
Resource constraints	Abundant	Severely limited
Deployment scale	Thousands	Millions to billions
Physical access	Data centers, offices	Field deployed, remote
Lifecycle	3–5 years	10–25 years
Interoperability	Standard protocols	Proprietary protocols common
Cost per unit	$1000s	$1–$100

10.1.2 IoT-Specific Migration Challenges

10.2 Migration Strategies

10.2.1 Strategy 1: Crypto-Agility First

Implement cryptographic agility before migrating to PQC:

```
/*
 * Crypto-Agile Framework for IoT Devices
 * Enables runtime algorithm selection and future migration
 */

#include <stdint.h>
#include <stdbool.h>
#include <string.h>

/* Algorithm registry */
typedef enum {
    /* Key Exchange Mechanisms */
    ALG_ECDH_P256 = 0x0101,
    ALG_X25519 = 0x0102,
```

```
    ALG_MLKEM_512 = 0x0201,
    ALG_MLKEM_768 = 0x0202,
    ALG_MLKEM_1024 = 0x0203,
    ALG_HYBRID_X25519_MLKEM768 = 0x0301,

    /* Signature Algorithms */
    ALG_ECDSA_P256 = 0x1101,
    ALG_ED25519 = 0x1102,
    ALG_MLDSA_44 = 0x1201,
    ALG_MLDSA_65 = 0x1202,
    ALG_MLDSA_87 = 0x1203,
    ALG_FALCON_512 = 0x1301,
    ALG_HYBRID_ECDSA_MLDSA = 0x1401,
} crypto_algorithm_t;

/* Algorithm capabilities */
typedef struct {
    crypto_algorithm_t id;
    const char *name;
    uint16_t pk_size;
    uint16_t sk_size;
    uint16_t ct_or_sig_size;
    uint16_t security_level;
    bool quantum_safe;
    bool fips_approved;

    /* Function pointers for operations */
    int (*keygen)(uint8_t *pk, uint8_t *sk);
    int (*encaps)(uint8_t *ct, uint8_t *ss, const uint8_t
       *pk);
    int (*decaps)(uint8_t *ss, const uint8_t *ct, const
       uint8_t *sk);
    int (*sign)(uint8_t *sig, size_t *sig_len,
                const uint8_t *msg, size_t msg_len, const
                   uint8_t *sk);
    int (*verify)(const uint8_t *sig, size_t sig_len,
                  const uint8_t *msg, size_t msg_len, const
                     uint8_t *pk);
} algorithm_info_t;

/* Algorithm registry - populated at compile time or runtime
```

```
     */
#define MAX_ALGORITHMS 32
static algorithm_info_t g_algorithm_registry[MAX_ALGORITHMS];
static int g_num_algorithms = 0;

/* Register an algorithm */
int crypto_register_algorithm(const algorithm_info_t *alg) {
    if (g_num_algorithms >= MAX_ALGORITHMS) {
        return -1;
    }
    memcpy(&g_algorithm_registry[g_num_algorithms], alg,
           sizeof(algorithm_info_t));
    g_num_algorithms++;
    return 0;
}

/* Find algorithm by ID */
const algorithm_info_t
   *crypto_get_algorithm(crypto_algorithm_t id) {
    for (int i = 0; i < g_num_algorithms; i++) {
        if (g_algorithm_registry[i].id == id) {
            return &g_algorithm_registry[i];
        }
    }
    return NULL;
}

/* Policy-based algorithm selection */
typedef struct {
    bool require_quantum_safe;
    bool require_fips;
    uint16_t min_security_level;
    uint16_t max_pk_size;
    uint16_t max_ct_size;
    crypto_algorithm_t preferred_kem;
    crypto_algorithm_t preferred_sig;
} crypto_policy_t;

/* Select best algorithm matching policy */
crypto_algorithm_t select_kem_algorithm(const
    crypto_policy_t *policy) {
```

```
90      /* Try preferred first */
91      const algorithm_info_t *pref =
           crypto_get_algorithm(policy->preferred_kem);
92      if (pref && algorithm_matches_policy(pref, policy)) {
93          return policy->preferred_kem;
94      }
95
96      /* Fall back to best available */
97      for (int i = 0; i < g_num_algorithms; i++) {
98          const algorithm_info_t *alg =
               &g_algorithm_registry[i];
99
100         /* Skip signature algorithms */
101         if ((alg->id & 0x1000) != 0) continue;
102
103         if (algorithm_matches_policy(alg, policy)) {
104             return alg->id;
105         }
106     }
107
108     return 0;  /* No suitable algorithm */
109 }
110
111 /* Crypto-agile key exchange */
112 typedef struct {
113     crypto_algorithm_t algorithm;
114     uint8_t public_key[4096];
115     uint16_t pk_len;
116     uint8_t secret_key[8192];
117     uint16_t sk_len;
118 } agile_keypair_t;
119
120 int agile_keygen(agile_keypair_t *kp, crypto_algorithm_t
       alg) {
121     const algorithm_info_t *info = crypto_get_algorithm(alg);
122     if (!info || !info->keygen) {
123         return -1;
124     }
125
126     kp->algorithm = alg;
127     kp->pk_len = info->pk_size;
```

```
128     kp->sk_len = info->sk_size;
129
130     return info->keygen(kp->public_key, kp->secret_key);
131 }
132
133 int agile_encaps(uint8_t *ct, uint16_t *ct_len,
134                  uint8_t *shared_secret,
135                  const agile_keypair_t *recipient_pk) {
136     const algorithm_info_t *info =
           crypto_get_algorithm(recipient_pk->algorithm);
137     if (!info || !info->encaps) {
138         return -1;
139     }
140
141     *ct_len = info->ct_or_sig_size;
142     return info->encaps(ct, shared_secret,
           recipient_pk->public_key);
143 }
144
145 int agile_decaps(uint8_t *shared_secret,
146                  const uint8_t *ct, uint16_t ct_len,
147                  const agile_keypair_t *kp) {
148     const algorithm_info_t *info =
           crypto_get_algorithm(kp->algorithm);
149     if (!info || !info->decaps) {
150         return -1;
151     }
152
153     return info->decaps(shared_secret, ct, kp->secret_key);
154 }
155
156 /* Algorithm negotiation for protocols */
157 typedef struct {
158     crypto_algorithm_t supported_kems[8];
159     uint8_t num_kems;
160     crypto_algorithm_t supported_sigs[8];
161     uint8_t num_sigs;
162 } crypto_capabilities_t;
163
164 crypto_algorithm_t negotiate_kem(const crypto_capabilities_t
       *local,
```

```
165                                     const
                                           crypto_capabilities_t
                                           *remote,
166                                     const crypto_policy_t
                                           *policy) {
167     /* Find strongest mutually supported algorithm */
168     for (int i = 0; i < local->num_kems; i++) {
169         crypto_algorithm_t local_alg =
               local->supported_kems[i];
170
171         for (int j = 0; j < remote->num_kems; j++) {
172             if (local_alg == remote->supported_kems[j]) {
173                 const algorithm_info_t *info =
                       crypto_get_algorithm(local_alg);
174                 if (info && algorithm_matches_policy(info,
                       policy)) {
175                     return local_alg;
176                 }
177             }
178         }
179     }
180
181     return 0;  /* No common algorithm */
182 }
```

Listing 10.1: Crypto-Agile Architecture for IoT

10.2.2 Strategy 2: Phased Hybrid Deployment

Table 10.3: Phased Hybrid Deployment Plan

Phase	Timeline	Configuration	Action Items
1	2024–2025	Classical only	Inventory, plan, pilot hybrid
2	2025–2026	Hybrid optional	Deploy hybrid to new devices
3	2026–2028	Hybrid default	Upgrade existing devices to hybrid
4	2028–2030	PQC preferred	Classical as fallback only
5	2030+	PQC only	Remove classical algorithms

10.2.3 Strategy 3: Risk-Based Prioritization

```
#!/usr/bin/env python3
"""
Risk-Based PQC Migration Prioritization
Calculates migration priority based on multiple risk factors
"""

from dataclasses import dataclass
from typing import List
from enum import Enum
import math

class DataSensitivity(Enum):
    PUBLIC = 1
    INTERNAL = 2
    CONFIDENTIAL = 3
    SECRET = 4
    TOP_SECRET = 5

class ThreatExposure(Enum):
    INTERNAL_ONLY = 1
    LIMITED_EXTERNAL = 2
    INTERNET_FACING = 3
    CRITICAL_INFRASTRUCTURE = 4

@dataclass
class CryptoSystem:
    name: str
    system_type: str  # 'kem', 'signature', 'both'
    current_algorithm: str
    data_sensitivity: DataSensitivity
    data_retention_years: int
    threat_exposure: ThreatExposure
    device_count: int
    upgrade_difficulty: int  # 1-10
    business_criticality: int  # 1-10
    compliance_requirements: List[str]

@dataclass
class MigrationPriority:
    system: CryptoSystem
    hndl_risk_score: float
```

```
    compliance_urgency: float
    operational_risk: float
    overall_priority: float
    recommended_deadline: int  # Year

def calculate_hndl_risk(system: CryptoSystem, crqc_year: int
    = 2030) -> float:
    """
    Calculate Harvest-Now-Decrypt-Later risk.
    Higher score = higher risk = migrate sooner
    """
    current_year = 2025
    years_until_crqc = crqc_year - current_year

    # Data will be decryptable when: retention ends OR CRQC
       arrives
    years_data_vulnerable = max(0,
        system.data_retention_years - years_until_crqc)

    # Base risk from data sensitivity
    sensitivity_weight = system.data_sensitivity.value / 5.0

    # Exposure multiplier
    exposure_multiplier = system.threat_exposure.value / 2.0

    # HNDL risk score (0-100)
    risk = (years_data_vulnerable * 10 * sensitivity_weight *
            exposure_multiplier)

    return min(100, risk)

def calculate_compliance_urgency(system: CryptoSystem) ->
    float:
    """Calculate urgency based on compliance deadlines."""
    urgency = 0
    current_year = 2025

    compliance_deadlines = {
        'CNSA_2.0': 2025,
        'CNSA_2.0_FULL': 2033,
        'PCI_DSS_4.0': 2025,
```

```
        'HIPAA': 2027,
        'FIPS_140_3': 2026,
        'eIDAS_2.0': 2027,
    }

    for req in system.compliance_requirements:
        if req in compliance_deadlines:
            years_until = compliance_deadlines[req] -
               current_year
            if years_until <= 0:
                urgency += 100  # Already past deadline
            elif years_until <= 1:
                urgency += 80
            elif years_until <= 2:
                urgency += 50
            else:
                urgency += 20

    return min(100, urgency)

def calculate_operational_risk(system: CryptoSystem) ->
   float:
    """Calculate risk of migration disruption."""
    # Higher difficulty and criticality = higher operational
       risk
    risk = (system.upgrade_difficulty * 5 +
            system.business_criticality * 5)

    # Scale by device count (logarithmic)
    scale_factor = math.log10(max(1, system.device_count)) /
       3
    risk *= (1 + scale_factor * 0.5)

    return min(100, risk)

def prioritize_migration(systems: List[CryptoSystem]) ->
   List[MigrationPriority]:
    """
    Calculate migration priority for all systems.
    Returns sorted list, highest priority first.
    """
```

```
    priorities = []

    for system in systems:
        hndl = calculate_hndl_risk(system)
        compliance = calculate_compliance_urgency(system)
        operational = calculate_operational_risk(system)

        # Overall priority: weighted combination
        # HNDL and compliance drive urgency, operational risk is a penalty
        overall = (hndl * 0.4 + compliance * 0.4 - operational * 0.2)
        overall = max(0, min(100, overall))

        # Recommended deadline based on priority
        if overall >= 80:
            deadline = 2025
        elif overall >= 60:
            deadline = 2026
        elif overall >= 40:
            deadline = 2027
        elif overall >= 20:
            deadline = 2028
        else:
            deadline = 2030

        priorities.append(MigrationPriority(
            system=system,
            hndl_risk_score=hndl,
            compliance_urgency=compliance,
            operational_risk=operational,
            overall_priority=overall,
            recommended_deadline=deadline
        ))

    # Sort by priority (highest first)
    priorities.sort(key=lambda x: x.overall_priority, reverse=True)

    return priorities

```

```
def generate_migration_roadmap(priorities: List[MigrationPriority]) -> dict:
    """Generate year-by-year migration roadmap."""
    roadmap = {}

    for p in priorities:
        year = p.recommended_deadline
        if year not in roadmap:
            roadmap[year] = []

        roadmap[year].append({
            'system': p.system.name,
            'priority_score': p.overall_priority,
            'hndl_risk': p.hndl_risk_score,
            'device_count': p.system.device_count,
            'current_algo': p.system.current_algorithm
        })

    return roadmap

# Example usage
if __name__ == '__main__':
    systems = [
        CryptoSystem(
            name="Medical Device Fleet",
            system_type="both",
            current_algorithm="ECDSA + ECDH",
            data_sensitivity=DataSensitivity.SECRET,
            data_retention_years=25,
            threat_exposure=ThreatExposure.LIMITED_EXTERNAL,
            device_count=50000,
            upgrade_difficulty=8,
            business_criticality=9,
            compliance_requirements=['HIPAA', 'FIPS_140_3']
        ),
        CryptoSystem(
            name="Smart Meter Network",
            system_type="kem",
            current_algorithm="ECDH P-256",
            data_sensitivity=DataSensitivity.CONFIDENTIAL,
            data_retention_years=15,
```

```
            threat_exposure=ThreatExposure.CRITICAL_INFRASTRUCTURE,
            device_count=2000000,
            upgrade_difficulty=6,
            business_criticality=8,
            compliance_requirements=['CNSA_2.0']
        ),
        CryptoSystem(
            name="Consumer IoT Hub",
            system_type="both",
            current_algorithm="RSA-2048",
            data_sensitivity=DataSensitivity.INTERNAL,
            data_retention_years=5,
            threat_exposure=ThreatExposure.INTERNET_FACING,
            device_count=500000,
            upgrade_difficulty=4,
            business_criticality=5,
            compliance_requirements=[]
        ),
    ]

    priorities = prioritize_migration(systems)

    print("=== PQC Migration Priority Analysis ===\n")
    for i, p in enumerate(priorities, 1):
        print(f"{i}. {p.system.name}")
        print(f"   Priority Score:
           {p.overall_priority:.1f}/100")
        print(f"   HNDL Risk: {p.hndl_risk_score:.1f}")
        print(f"   Compliance Urgency:
           {p.compliance_urgency:.1f}")
        print(f"   Operational Risk:
           {p.operational_risk:.1f}")
        print(f"   Recommended Deadline:
           {p.recommended_deadline}")
        print()

    roadmap = generate_migration_roadmap(priorities)
    print("=== Migration Roadmap ===")
    for year in sorted(roadmap.keys()):
        print(f"\n{year}:")
        for item in roadmap[year]:
```

```
231                 print(f"  - {item['system']}
                        ({item['device_count']:,} devices)")
```

Listing 10.2: Risk-Based Migration Prioritization Tool

10.3 Cost Analysis Framework

10.3.1 Total Cost of Migration

Table 10.4: PQC Migration Cost Categories

Category	Components	Typical Range
Assessment	Crypto inventory, risk assessment, gap analysis	$50K–$500K
Development	Library integration, testing, code changes	$200K–$2M
Infrastructure	Key management, PKI, HSMs	$100K–$1M
Deployment	OTA updates, field upgrades, roll-out	$100K–$5M
Certification	FIPS 140-3, CC, industry certs	$200K–$1M
Training	Developer, operations, security teams	$50K–$200K
Ongoing	Monitoring, incident response, updates	$100K–$500K/yr

```
1  #!/usr/bin/env python3
2  """
3  PQC Migration Cost Estimation Tool
4  Provides detailed cost breakdown for IoT PQC migration
       projects
5  """
6
7  from dataclasses import dataclass, field
8  from typing import List, Optional
9  from enum import Enum
10
11 class DeviceCategory(Enum):
12     ULTRA_CONSTRAINED = "8-bit MCU, <32KB RAM"
13     CONSTRAINED = "Cortex-M0/M3, 32-128KB RAM"
14     MODERATE = "Cortex-M4/M7, 128KB-1MB RAM"
15     CAPABLE = "Cortex-A class, >1MB RAM"
16     GATEWAY = "Linux-based gateway"
```

```
17
18 class MigrationComplexity(Enum):
19     LOW = 1       # Drop-in library replacement
20     MEDIUM = 2    # Some protocol changes
21     HIGH = 3      # Significant redesign
22     VERY_HIGH = 4  # Complete cryptographic overhaul
23
24 @dataclass
25 class DeviceFleet:
26     name: str
27     category: DeviceCategory
28     count: int
29     current_crypto: str
30     ota_capable: bool
31     average_age_years: float
32     expected_lifetime_years: float
33
34 @dataclass
35 class CostFactors:
36     # Labor rates (USD/hour)
37     developer_rate: float = 150
38     security_engineer_rate: float = 200
39     qa_rate: float = 100
40     project_manager_rate: float = 175
41
42     # External costs
43     fips_validation_cost: float = 350000
44     cc_evaluation_cost: float = 250000
45     hsm_cost_per_unit: float = 15000
46     pki_infrastructure: float = 100000
47
48     # Per-device costs
49     ota_cost_per_device: float = 0.10
50     field_upgrade_cost: float = 150
51     device_replacement_cost_factor: float = 1.5
52
53 @dataclass
54 class CostEstimate:
55     category: str
56     description: str
57     low_estimate: float
```

```
    high_estimate: float
    confidence: str  # 'high', 'medium', 'low'

@dataclass
class MigrationCostReport:
    project_name: str
    total_low: float = 0
    total_high: float = 0
    cost_breakdown: List[CostEstimate] =
       field(default_factory=list)
    timeline_months: int = 0
    risk_factors: List[str] = field(default_factory=list)

def estimate_development_cost(
    fleet: DeviceFleet,
    complexity: MigrationComplexity,
    factors: CostFactors
) -> CostEstimate:
    """Estimate software development costs."""

    # Base hours by complexity
    base_hours = {
        MigrationComplexity.LOW: 400,
        MigrationComplexity.MEDIUM: 1200,
        MigrationComplexity.HIGH: 3000,
        MigrationComplexity.VERY_HIGH: 6000
    }

    # Category multiplier
    category_mult = {
        DeviceCategory.ULTRA_CONSTRAINED: 2.5,  # Very
           difficult
        DeviceCategory.CONSTRAINED: 1.8,
        DeviceCategory.MODERATE: 1.2,
        DeviceCategory.CAPABLE: 1.0,
        DeviceCategory.GATEWAY: 0.8
    }

    hours = base_hours[complexity] *
       category_mult[fleet.category]

```

```
    # Calculate costs
    dev_hours = hours * 0.6
    sec_hours = hours * 0.25
    qa_hours = hours * 0.15

    cost = (dev_hours * factors.developer_rate +
            sec_hours * factors.security_engineer_rate +
            qa_hours * factors.qa_rate)

    return CostEstimate(
        category="Development",
        description=f"Code development for {fleet.name}",
        low_estimate=cost * 0.8,
        high_estimate=cost * 1.4,
        confidence="medium"
    )

def estimate_deployment_cost(
    fleet: DeviceFleet,
    factors: CostFactors
) -> CostEstimate:
    """Estimate deployment and rollout costs."""

    if fleet.ota_capable:
        # OTA update costs
        cost_per_device = factors.ota_cost_per_device
        # Account for multiple update attempts, bandwidth
        total = fleet.count * cost_per_device * 1.5

        return CostEstimate(
            category="Deployment",
            description=f"OTA deployment to {fleet.count:,}
               devices",
            low_estimate=total * 0.8,
            high_estimate=total * 1.5,
            confidence="high"
        )
    else:
        # Estimate percentage needing field upgrade vs
           replacement
        old_devices = fleet.count * min(1.0,
```

```
            fleet.average_age_years /
135                                         fleet.expected_lifetime_years)
136         upgradeable = fleet.count - old_devices
137
138         upgrade_cost = upgradeable *
               factors.field_upgrade_cost
139         # Old devices may need replacement
140         replacement_cost = old_devices *
               factors.field_upgrade_cost * \
141                           factors.device_replacement_cost_factor
142
143         total = upgrade_cost + replacement_cost
144
145         return CostEstimate(
146             category="Deployment",
147             description=f"Field deployment ({fleet.count:,}
                   devices, "
148                         f"{int(old_devices):,} may need
                            replacement)",
149             low_estimate=total * 0.7,
150             high_estimate=total * 2.0,
151             confidence="low"
152         )
153
154 def estimate_certification_cost(
155     require_fips: bool,
156     require_cc: bool,
157     num_platforms: int,
158     factors: CostFactors
159 ) -> CostEstimate:
160     """Estimate certification costs."""
161
162     total = 0
163     desc_parts = []
164
165     if require_fips:
166         # FIPS 140-3 validation per platform
167         total += factors.fips_validation_cost * num_platforms
168         desc_parts.append(f"FIPS 140-3 ({num_platforms}
               platforms)")
169
```

```
    if require_cc:
        total += factors.cc_evaluation_cost * num_platforms
        desc_parts.append(f"Common Criteria ({num_platforms}
           platforms)")

    if not desc_parts:
        return CostEstimate(
            category="Certification",
            description="No certification required",
            low_estimate=0,
            high_estimate=0,
            confidence="high"
        )

    return CostEstimate(
        category="Certification",
        description=", ".join(desc_parts),
        low_estimate=total * 0.9,
        high_estimate=total * 1.3,
        confidence="medium"
    )

def estimate_infrastructure_cost(
    num_hsms: int,
    new_pki_required: bool,
    factors: CostFactors
) -> CostEstimate:
    """Estimate infrastructure costs."""

    total = 0

    if num_hsms > 0:
        total += num_hsms * factors.hsm_cost_per_unit

    if new_pki_required:
        total += factors.pki_infrastructure

    return CostEstimate(
        category="Infrastructure",
        description=f"{num_hsms} HSMs, {'new' if
           new_pki_required else 'existing'} PKI",
```

```
209         low_estimate=total * 0.85,
210         high_estimate=total * 1.2,
211         confidence="high"
212     )
213
214 def generate_cost_report(
215     project_name: str,
216     fleets: List[DeviceFleet],
217     complexity: MigrationComplexity,
218     require_fips: bool = False,
219     require_cc: bool = False,
220     num_hsms: int = 0,
221     new_pki: bool = False,
222     factors: Optional[CostFactors] = None
223 ) -> MigrationCostReport:
224     """Generate comprehensive cost report."""
225
226     if factors is None:
227         factors = CostFactors()
228
229     report = MigrationCostReport(project_name=project_name)
230
231     # Assessment costs (fixed estimate)
232     report.cost_breakdown.append(CostEstimate(
233         category="Assessment",
234         description="Cryptographic inventory and gap
               analysis",
235         low_estimate=50000,
236         high_estimate=150000,
237         confidence="medium"
238     ))
239
240     # Development costs per fleet
241     for fleet in fleets:
242         dev_cost = estimate_development_cost(fleet,
               complexity, factors)
243         report.cost_breakdown.append(dev_cost)
244
245         deploy_cost = estimate_deployment_cost(fleet,
               factors)
246         report.cost_breakdown.append(deploy_cost)
```

```
247
248     # Certification
249     num_platforms = len(set(f.category for f in fleets))
250     cert_cost = estimate_certification_cost(
251         require_fips, require_cc, num_platforms, factors)
252     report.cost_breakdown.append(cert_cost)
253
254     # Infrastructure
255     infra_cost = estimate_infrastructure_cost(num_hsms,
            new_pki, factors)
256     report.cost_breakdown.append(infra_cost)
257
258     # Training
259     report.cost_breakdown.append(CostEstimate(
260         category="Training",
261         description="Developer and operations training",
262         low_estimate=50000,
263         high_estimate=150000,
264         confidence="medium"
265     ))
266
267     # Calculate totals
268     report.total_low = sum(c.low_estimate for c in
            report.cost_breakdown)
269     report.total_high = sum(c.high_estimate for c in
            report.cost_breakdown)
270
271     # Estimate timeline
272     total_devices = sum(f.count for f in fleets)
273     base_months = 12 + (complexity.value * 6)
274     device_scale = min(12, total_devices / 100000)
275     report.timeline_months = int(base_months + device_scale)
276
277     # Risk factors
278     for fleet in fleets:
279         if not fleet.ota_capable:
280             report.risk_factors.append(
281                 f"{fleet.name}: No OTA capability increases
                        deployment risk")
282         if fleet.category ==
                DeviceCategory.ULTRA_CONSTRAINED:
```

```
            report.risk_factors.append(
                f"{fleet.name}: Ultra-constrained devices may not support PQC")

    return report

def print_cost_report(report: MigrationCostReport):
    """Print formatted cost report."""
    print(f"{'='*60}")
    print(f"PQC Migration Cost Report: {report.project_name}")
    print(f"{'='*60}\n")

    print("Cost Breakdown:")
    print("-" * 60)
    for cost in report.cost_breakdown:
        print(f"\n{cost.category}: {cost.description}")
        print(f"  Estimate: ${cost.low_estimate:,.0f} - ${cost.high_estimate:,.0f}")
        print(f"  Confidence: {cost.confidence}")

    print("\n" + "=" * 60)
    print(f"TOTAL ESTIMATE: ${report.total_low:,.0f} - ${report.total_high:,.0f}")
    print(f"Timeline: {report.timeline_months} months")
    print("=" * 60)

    if report.risk_factors:
        print("\nRisk Factors:")
        for risk in report.risk_factors:
            print(f"  ! {risk}")

# Example usage
if __name__ == '__main__':
    fleets = [
        DeviceFleet(
            name="Smart Sensors",
            category=DeviceCategory.CONSTRAINED,
            count=100000,
            current_crypto="ECDH + ECDSA",
            ota_capable=True,
```

```
            average_age_years=2,
            expected_lifetime_years=10
        ),
        DeviceFleet(
            name="Industrial Controllers",
            category=DeviceCategory.MODERATE,
            count=5000,
            current_crypto="RSA-2048",
            ota_capable=False,
            average_age_years=5,
            expected_lifetime_years=15
        ),
    ]

    report = generate_cost_report(
        project_name="Industrial IoT PQC Migration",
        fleets=fleets,
        complexity=MigrationComplexity.MEDIUM,
        require_fips=True,
        num_hsms=2,
        new_pki=True
    )

    print_cost_report(report)
```

Listing 10.3: PQC Migration Cost Estimator

10.3.2 Cost-Benefit Analysis

Table 10.5: PQC Migration Cost vs. Risk of Inaction

Scenario	Migration Cost	Risk of Inaction
Small IoT deployment (1K devices)	\$100K–\$500K	Data breach: \$1M+
Medium fleet (100K devices)	\$500K–\$2M	Regulatory fines: \$5M+
Large deployment (1M+ devices)	\$2M–\$10M	Mass compromise: \$50M+
Critical infrastructure	\$5M–\$20M	National security risk

Warning: The cost of a post-quantum attack on unprepared systems will far exceed migration costs. Organizations storing data with long sensitivity periods (healthcare, finance, government) face the highest risk-adjusted costs from inaction.

10.4 Implementation Checklist

Table 10.6: PQC Migration Implementation Checklist

Phase	#	Action Item
Prepare	1	Complete cryptographic asset inventory
	2	Assess quantum risk for each asset (HNDL analysis)
	3	Identify compliance requirements and deadlines
	4	Evaluate device upgrade capabilities (OTA, memory, CPU)
	5	Select target PQC algorithms per use case
Plan	6	Develop crypto-agility architecture
	7	Create phased migration roadmap
	8	Estimate costs and secure budget
	9	Identify pilot projects for early deployment
	10	Plan PKI and key management updates
Execute	11	Integrate PQC libraries (liboqs, wolfSSL, etc.)
	12	Implement hybrid mode (classical + PQC)
	13	Update protocols (TLS, DTLS, custom)
	14	Deploy to pilot devices and validate
	15	Execute phased rollout to production
Verify	16	Perform security testing and penetration testing
	17	Validate interoperability with partners
	18	Complete required certifications (FIPS, etc.)
	19	Document all changes for audit
	20	Establish monitoring and incident response

10.5 Lessons Learned and Recommendations

10.5.1 Key Success Factors

Based on early PQC migration projects, the following factors contribute to success:

1. **Start Early**: Begin assessment and planning now, even if full deployment is years away

2. **Prioritize Crypto-Agility**: Design systems to switch algorithms without major redesign

3. **Adopt Hybrid Approach**: Deploy hybrid classical+PQC to maintain security during transition
4. **Focus on High-Risk Data**: Prioritize systems handling long-lived sensitive data
5. **Test Thoroughly**: PQC algorithms are less battle-tested than classical crypto
6. **Plan for Larger Sizes**: Account for increased key, signature, and ciphertext sizes
7. **Monitor Standards**: NIST standards are final, but implementation guidance evolves
8. **Engage Supply Chain**: Ensure vendors and partners are also preparing for PQC

10.5.2 Common Pitfalls to Avoid

Table 10.7: Common PQC Migration Pitfalls

Pitfall	Mitigation
Waiting for "perfect" solution	Deploy hybrid now; pure PQC can come later
Underestimating size impact	Test with actual PQC sizes early in design
Ignoring legacy devices	Plan for device replacement where upgrade impossible
Single algorithm selection	Maintain agility for algorithm updates
Skipping interoperability testing	Test with multiple implementations and partners
Neglecting key management	Update KMS, HSMs, and PKI for PQC
Insufficient testing	PQC implementations may have subtle bugs
Forgetting about performance	Benchmark on actual target hardware

10.5.3 Final Recommendations

Note:

Summary Recommendations for IoT PQC Migration:

1. **2024–2025**: Complete inventory, risk assessment, and pilot hybrid deployments
2. **2025–2027**: Deploy hybrid mode to all new devices; begin upgrading existing fleet
3. **2027–2030**: Transition to PQC-primary with classical fallback

4. **2030+**: Complete transition to pure PQC for all new communications

For IoT devices specifically:

- Use **ML-KEM-768** for key exchange (balance of security and size)
- Use **ML-DSA-65** or **Falcon-512** for signatures (based on size constraints)
- Implement **hybrid mode** to protect against both classical and quantum attacks
- Design for **crypto-agility** to accommodate future algorithm updates
- Plan for devices that **cannot be upgraded** (replacement or isolation)

10.6 Chapter Summary

This chapter has provided comprehensive guidance for planning and executing the migration from classical to post-quantum cryptography in IoT deployments. The key areas covered include:

Migration Timeline and Planning:

- Industry consensus timeline from 2024 through 2035+
- Phased migration approach: preparation, pilot, production, transition, completion
- Critical milestones aligned with regulatory deadlines (CNSA 2.0, FIPS requirements)
- Risk-based prioritization for different asset categories

Migration Strategies:

- Greenfield vs. brownfield deployment considerations
- Hybrid cryptography as the recommended transitional approach
- Device fleet segmentation and upgrade path determination
- Legacy device handling: upgrade, replace, isolate, or accept risk

Cost Analysis Framework:

- Comprehensive cost categories: development, hardware, infrastructure, operations
- TCO (Total Cost of Ownership) modeling for 5–10 year periods
- ROI calculation methodologies and risk-adjusted analysis

- Hidden costs: bandwidth increases, battery impact, interoperability testing

Resource Requirements:

- Hardware upgrades: memory expansion, crypto accelerators, secure elements
- Network infrastructure: bandwidth provisioning for larger PQC payloads
- Personnel: training, hiring specialized cryptographic expertise
- Testing and validation: interoperability, performance, security audits

Risk Assessment:

- Quantum risk timeline estimation (5–15 year window)
- HNDL threat assessment based on data sensitivity and retention
- Migration risk categories: technical, operational, compliance, business
- Risk mitigation strategies and contingency planning

Implementation Guidance:

- Detailed implementation checklists for each migration phase
- Lessons learned from early adopters and pilot deployments
- Common pitfalls and their mitigations
- Success metrics and progress tracking methodologies

The central conclusion of this chapter is that successful PQC migration requires careful planning, adequate resource allocation, and a pragmatic phased approach. Organizations should begin their migration journey immediately, starting with inventory and risk assessment, while using hybrid cryptography to protect against both current and future threats during the transition period.

Conclusion

The transition to post-quantum cryptography represents a fundamental shift in how we secure digital communications. For the Internet of Things, this transition presents unique challenges due to resource constraints, long device lifecycles, and the scale of deployments. However, as this document has demonstrated, practical solutions exist for implementing quantum-resistant security even on highly constrained devices.

Key Takeaways:

1. **The quantum threat is real and approaching**. While cryptographically relevant quantum computers may be 5–15 years away, the harvest-now-decrypt-later threat means that sensitive data transmitted today using classical cryptography may be compromised in the future.

2. **NIST standards are ready**. With FIPS 203 (ML-KEM), FIPS 204 (ML-DSA), and FIPS 205 (SLH-DSA) finalized in August 2024, organizations can begin deploying standardized post-quantum algorithms immediately.

3. **IoT implementation is feasible**. Benchmarks from pqm4 and real-world deployments demonstrate that ML-KEM and ML-DSA can run on Cortex-M4 class devices with acceptable performance. Even more constrained devices can leverage optimized implementations or offload cryptographic operations.

4. **Hybrid deployment is the recommended approach**. Combining classical and post-quantum algorithms provides defense against both current and future threats while maintaining interoperability during the transition period.

5. **Migration planning should begin now**. Organizations should inventory cryptographic assets, assess quantum risk, and develop migration roadmaps. Early planning reduces costs and risks.

Future Outlook:

The post-quantum cryptography landscape continues to evolve. Additional signature schemes (Falcon) and code-based KEMs (HQC, BIKE) are progressing through standardization. Hardware acceleration for lattice operations is emerging in new microcontrollers. Protocol integration (TLS 1.3, DTLS, MQTT, CoAP) is maturing rapidly.

For IoT practitioners, the message is clear: the time to prepare for the post-quantum era is now. By following the guidance in this document—selecting appropriate algorithms, implementing crypto-agile architectures, and executing phased migration plans—organizations can protect their IoT deployments against both current and future cryptographic threats.

The future of secure IoT is post-quantum.

References

Bibliography

[1] National Institute of Standards and Technology, "FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard," August 2024.

[2] National Institute of Standards and Technology, "FIPS 204: Module-Lattice-Based Digital Signature Standard," August 2024.

[3] National Institute of Standards and Technology, "FIPS 205: Stateless Hash-Based Digital Signature Standard," August 2024.

[4] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, "pqm4: Post-quantum crypto library for the ARM Cortex-M4," https://github.com/mupq/pqm4, 2024.

[5] Open Quantum Safe Project, "liboqs: C library for quantum-safe cryptographic algorithms," https://github.com/open-quantum-safe/liboqs, 2024.

[6] R. Avanzi et al., "CRYSTALS-Kyber Algorithm Specifications and Supporting Documentation," NIST PQC Standardization, 2023.

[7] L. Ducas et al., "CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation," NIST PQC Standardization, 2023.

[8] T. Prest et al., "Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU," NIST PQC Standardization, 2023.

[9] J.-P. Aumasson et al., "SPHINCS+ Specification," NIST PQC Standardization, 2023.

[10] National Security Agency, "Commercial National Security Algorithm Suite 2.0," September 2022.

[11] wolfSSL Inc., "wolfSSL Embedded SSL/TLS Library," https://www.wolfssl.com/, 2024.

[12] Cloudflare, "Post-Quantum Cryptography," https://blog.cloudflare.com/post-quantum-cryptography/, 2024.

[13] C. Aguilar Melchor et al., "HQC: Hamming Quasi-Cyclic," NIST PQC Round 4, 2023.

[14] N. Aragon et al., "BIKE: Bit Flipping Key Encapsulation," NIST PQC Round 4, 2023.

[15] D. J. Bernstein et al., "Classic McEliece," NIST PQC Round 4, 2023.

[16] D. Stebila and S. Fluhrer, "Hybrid Key Exchange in TLS 1.3," Internet-Draft, IETF, 2024.

[17] National Institute of Standards and Technology, "SP 800-208: Recommendation for Stateful Hash-Based Signature Schemes," October 2020.

[18] ISO/IEC, "ISO/IEC 18033-2: Encryption algorithms — Part 2: Asymmetric ciphers," 2024.

[19] ETSI, "Quantum-Safe Cryptography (QSC) Technical Committee," https://www.etsi.org/committee/qsc, 2024.

[20] ARM Ltd., "TrustZone Technology for ARMv8-M Architecture," 2024.

Appendix A

Algorithm Parameter Reference

A.1 ML-KEM Parameters

Table A.1: Complete ML-KEM Parameter Sets (FIPS 203)

Parameter	**ML-KEM-512**	**ML-KEM-768**	**ML-KEM-1024**
Security Level	1	3	5
n	256	256	256
k	2	3	4
q	3329	3329	3329
η_1	3	2	2
η_2	2	2	2
d_u	10	10	11
d_v	4	4	5
Public Key (bytes)	800	1,184	1,568
Secret Key (bytes)	1,632	2,400	3,168
Ciphertext (bytes)	768	1,088	1,568
Shared Secret (bytes)	32	32	32

A.2 ML-DSA Parameters

A.3 SLH-DSA Parameters

Table A.2: Complete ML-DSA Parameter Sets (FIPS 204)

Parameter	**ML-DSA-44**	**ML-DSA-65**	**ML-DSA-87**
Security Level	2	3	5
n	256	256	256
q	8,380,417	8,380,417	8,380,417
k	4	6	8
ℓ	4	5	7
η	2	4	2
τ	39	49	60
β	78	196	120
γ_1	2^{17}	2^{19}	2^{19}
γ_2	95,232	261,888	261,888
ω	80	55	75
Public Key (bytes)	1,312	1,952	2,592
Secret Key (bytes)	2,560	4,032	4,896
Signature (bytes)	2,420	3,293	4,595

Table A.3: SLH-DSA Parameter Sets (FIPS 205) — SHA-2 Variants

Parameter Set	**128s**	**128f**	**192s**	**192f**	**256s**	**256f**
Security Level	1	1	3	3	5	5
Public Key (B)	32	32	48	48	64	64
Secret Key (B)	64	64	96	96	128	128
Signature (B)	7,856	17,088	16,224	35,664	29,792	49,856

Appendix B

Benchmark Data

B.1 pqm4 Benchmarks (STM32F4 @ 168 MHz)

Table B.1: Complete pqm4 Benchmark Results (CPU Cycles)

Algorithm	KeyGen	Encaps/Sign	Decaps/Verify	Stack (B)
Key Encapsulation Mechanisms				
ML-KEM-512	434,000	486,000	524,000	2,280
ML-KEM-768	720,000	817,000	879,000	2,904
ML-KEM-1024	1,105,000	1,241,000	1,337,000	3,528
Digital Signatures				
ML-DSA-44	1,428,000	4,824,000	1,478,000	35,000
ML-DSA-65	2,355,000	6,324,000	2,456,000	51,000
ML-DSA-87	3,412,000	8,946,000	3,587,000	67,000
Falcon-512	65,000,000	42,000,000	430,000	39,000
SLH-DSA-128f	4,200,000	91,000,000	5,100,000	2,500

Appendix C

Code Examples Index

This appendix provides a quick reference to all code examples in this document:

Table C.1: Code Examples by Chapter

Chapter	Language	Description
3	C	ML-KEM encapsulation/decapsulation
3	Python	ML-KEM with liboqs bindings
4	C	ML-DSA signing and verification
4	C	Falcon fixed-point sampler
5	C	HQC decoding
6	C	Memory-optimized NTT
6	C	Side-channel resistant implementation
7	C	liboqs integration
7	Rust	pqcrypto usage
8	C	TLS 1.3 hybrid key exchange
8	C	MQTT with PQC
9	C	Constant-time comparison
9	C	Masking countermeasures
9	Python	Security level selection tool
9	C	FIPS mode configuration
10	C	Crypto-agile framework
10	Python	Migration cost estimator

Appendix D

Glossary of Terms

CRQC Cryptographically Relevant Quantum Computer — A quantum computer capable of breaking current public-key cryptography

HNDL Harvest Now, Decrypt Later — Attack strategy of storing encrypted data for future quantum decryption

KEM Key Encapsulation Mechanism — Asymmetric primitive for establishing shared secrets

LWE Learning With Errors — Mathematical problem underlying lattice-based cryptography

ML-KEM Module-Lattice Key Encapsulation Mechanism (formerly Kyber)

ML-DSA Module-Lattice Digital Signature Algorithm (formerly Dilithium)

NTT Number Theoretic Transform — Fast polynomial multiplication technique

OTA Over-The-Air — Remote firmware update capability

PQC Post-Quantum Cryptography — Cryptographic algorithms resistant to quantum attacks

SCA Side-Channel Attack — Attack exploiting physical implementation characteristics

SLH-DSA Stateless Hash-based Digital Signature Algorithm (formerly SPHINCS+)