PQC Algorithm Integration on RISC-V IoT Device

Cyber Security Low Prep — Team ID: 82

I. Problem Statement

The problem statement seeks to develop and show a post-quantum secure DTLS 1.3 communication channel that fully communicates the full mutual authentication between a server and a RISC-V–based IoT device running in a bare-metal (BIOS-level) virtual environment. That is, a post-quantum cryptographic algorithm must be included in the DTLS 1.3 protocol as well as maintaining the handshake protocol in compliance with DTLS 1.3.

The challenges come in adapting PQC mechanisms which are usually larger key sizes, more computational complexity, and more communication overhead to limited embedded hardware constraints. The implementation must thus improve latency, memory footprint, execution time, and overall computational overhead on the RISC-V platform. The final system will be successful in setting up a secure DTLS 1.3 session from PQC but also addresses the limitations of low power IoT devices, and uses efficient, reliable post-quantum communication when conditions are tight.

II. Architecture and Design Approach

Our architecture consists mainly of 5 components, namely, the application layer (main.c) that implements the certificate-based authentication. Then, the WolfSSL DTLS 1.3 Stack that implements the DTLS 1.3 protocol engine and the WolfCrypt Crypto Library for integration of the ML-KEM-512 and Dilithium Level 2. The Hardware Abstraction Layer consists of a custom UART I/O, a timer RNG and the last part is the LiteX RISC-V SoC Hardware Simulation. All of this is connected via UART (Serial) to a UART to UDP Bridge that helps it connect to the Linux DTLS 1.3 Server.

The key design principles considered and implemented are:

- **Modularity:** Clean separation between protocol, crypto, and hardware layers enables independent testing
- **Bare-metal constraints:** Direct hardware access with no OS dependencies.
- **Certificate-based PKI:** Three-tier hierarchy (CA, server, client) with mutual authentication.
- **Dual-mode operation:** Same codebase supports both client and server roles via compile-time flag

A. Additional Tools

The client was sending the keys in chunks that led to invalid handshakes. To correct this, a tool was implemented that would merge the chunks and send them through the bridge to the server correctly to facilitate this handshake.

III. PQC Algorithm Selection

A. ML-KEM-512 (Key Exchange)

Selection Rationale:

- NIST FIPS 203 standardized (2024)
- NIST Security Level 1 ($\approx$128-bit equivalent)
- Smallest footprint among ML-KEM variants
- Fast operations: $\sim$0.5 ms key generation, $\sim$0.7 ms encapsulation/decapsulation

Technical Specifications:

- Public key: 800 bytes
- Ciphertext: 768 bytes
- Shared secret: 32 bytes

B. Dilithium Level 2 (Digital Signatures)

Selection Rationale:

- NIST FIPS 204 standardized (ML-DSA)
- NIST Security Level 2 ($\approx$192-bit equivalent)
- Balanced performance for authentication use cases
- Memory-efficient with small-memory optimizations

Technical Specifications:

- Public key: 1,312 bytes
- Signature: $\sim$2,420 bytes
- Suitable for long-term certificate authentication

C. Symmetric Cryptography

- **AEAD Ciphers:** AES-128-GCM (primary), ChaCha20-Poly1305 (fallback)
- **Hash Functions:** SHA3-256, SHAKE256 (required by PQC algorithms)
- All algorithms are TLS 1.3 mandatory or recommended

TABLE I
COMPARISON OF CANDIDATE PQC ALGORITHMS

Algorithm	Type	Security Level	Key Size	Observation
Kyber-768	KEM	Level 3	Medium	Fast, NIST standardized. Moderate Memory Usage
Dilithium-2	Signature	Level 2	Large pub-key	Strong security, widely supported. Large Signature Size
Falcon-512	Signature	Level 1	Small signa-ture	Very efficient. Very complex integration

IV. FIRMWARE DESIGN OF IOT DEVICE

A. Memory Organisation

Component	Size	Percent
WolfSSL Core + DTLS 1.3	∼150 KB	29%
WolfCrypt PQC (ML-KEM + ML-DSA)	∼260 KB	50%
Embedded Certificates (DER)	∼12 KB	2%
Application Logic (DTLS Server + I/O)	∼30 KB	6%
LiteX BIOS + Runtime Libraries	∼70 KB	13%
Total ROM / Flash	**∼520 KB**	**100%**

RAM Usage Scenario	Size
Boot Stage	2–3 KB
DTLS + PQC Handshake Peak	14–18 KB
Steady-State Operation	4–6 KB

B. Hardware Abstraction Layer

1) Custom I/O Implementation:

- Defined **WOLFSSL_USER_IO** and **WOLFSSL_NO_SOCK** to bypass BSD sockets.
- Implemented **my_IORecv()** and **my_IOSend()** using direct UART register access.
- Non-blocking reads implemented using **WOLFSSL_CBIO_ERR_WANT_READ** return codes.

2) Time and Entropy Sources:

- **LowResTimer()**: Second-resolution monotonic timer derived from hardware counter.
- **CustomRngGenerateBlock()**: Timer-seeded linear congruential generator (LCG).

3) Certificate Embedding:

- X.509 certificates embedded as static byte arrays in **pqc_certs.h**.
- Automated toolchain generates Dilithium-signed certificates and converts DER to C arrays.
- WolfSSL's **wolfSSL_CTX_use_certificate_buffer()** loads certificates without filesystem support.

V. WOLFSSL/WOLFCRYPT INTEGRATION

A. Configuration Strategy

Created a custom **user_settings.h** with the following properties:

- **Protocol:** DTLS 1.3 only (all legacy TLS versions disabled)
- **PQC Algorithms:** ML-KEM, Dilithium, SHA3/SHAKE
- **Disabled:** RSA, DH, DSA, ECC-DHE (PQC-only operation enforced)
- **Platform Adaptations:** No filesystem, single-threaded, custom I/O callbacks

B. Build System Integration

- Selective compilation of WolfSSL sources into the LiteX firmware build.
- Excluded incompatible components (e.g., **conf.c** for config file parsing).
- Enabled **WOLFSSL_USER_SETTINGS** for bare-metal configuration.
- Cross-compiled using the **riscv64-unknown-elf-gcc** toolchain.

C. DTLS 1.3 Handshake Flow

The DTLS 1.3 handshake performs full mutual authentication with post-quantum primitives:

1) Client sends **ClientHello** containing its ML-KEM-512 public key and certificate.
2) Server replies with **ServerHello** containing the ML-KEM ciphertext and its certificate.
3) Both sides exchange **CertificateVerify** messages signed using Dilithium.
4) **Finished** messages confirm handshake completion.
5) **Application Data** is protected using AES-128-GCM.

VI. CHALLENGES

- **Integration Complexity:** Combining wolfSSL, PQC (ML-KEM/ML-DSA), and LiteX required extensive modifications; upstream configurations were not directly compatible.
- **Memory Constraints:** Large PQC key sizes required custom linker scripts and careful stack/heap optimization.
- **UART Transport Limits:** PQC handshake messages exceeded UART payload capacity, necessitating fragmentation and timeout tuning.

- **Custom I/O Backend:** Lack of BSD sockets required manual implementation of I/O callbacks such as `my_IOSend` and `my_IORecv`.
- **Entropy Source Limitations:** With no hardware TRNG available, timer-based pseudo-randomness had to be used temporarily.
- **Certificate Embedding:** Large DER certificates (6–10 KB) increased Flash usage and required compression and static-array embedding.
- **Build System Patching:** Required selective compilation, manual removal of unsupported files (e.g., `conf.c`), and patching Makefiles.
- **DTLS Fragmentation & Replay Handling:** Implementing anti-replay and record buffering within very small RAM constraints was challenging.

VII. Security Considerations

A. Security Strengths

- **Post-quantum key exchange:** ML-KEM-512 resists Shor's algorithm.
- **Post-quantum authentication:** Dilithium prevents quantum signature forgery.
- **Mutual authentication:** Both endpoints validate certificates.
- **Perfect forward secrecy:** Ephemeral ML-KEM keys are generated for every session.
- **AEAD encryption:** AES-GCM ensures confidentiality and integrity.

B. Applied Security Best Practices

- Disabled legacy protocols (TLS $\leq$ 1.2, RSA, DH).
- Using Strong cipher suites: AES-128-GCM, ChaCha20-Poly1305.
- Setting up with timing resistance enabled (**TFM_TIMING_RESISTANT**).
- No sensitive information leaked through error messages.

VIII. Performance Metrics

A. Latency Measurements

Operation	Time (ms)
ML-KEM-512 Key Generation	$\sim$1.5
ML-KEM-512 Encapsulation	$\sim$2.0
ML-KEM-512 Decapsulation	$\sim$2.5
Dilithium Sign	$\sim$8–12
Dilithium Verify	$\sim$15–25
AES-128-GCM Encrypt (1 KB)	$\sim$0.5
Complete Handshake	**$\sim$80–150**

Test Environment: 10 MHz RISC-V core, GCC 11.4.0 with `-O2`, no hardware accelerators.

B. Memory Usage Summary

Metric	Value
Total Firmware (ROM)	393 KB
Peak RAM (Handshake)	13–16 KB
Steady-state RAM	5 KB
Embedded Certificates	9 KB

Bottleneck Analysis: Throughput limited by UART speed (115,200 bps $\approx$ 10 KB/s), not cryptographic operations.

IX. Session Resumption

Not yet implemented due to bare-metal constraints:

- Requires PSK storage (no persistent storage available).
- Needs a ticket system with expiration tracking (no real-time clock).
- Increases RAM footprint for maintaining a session cache.

X. Low-Power Optimizations

A. Code Size Reduction

- `WOLFSSL_DILITHIUM_NO_LARGE_CODE`: Enables smaller Dilithium code paths.
- Removed legacy protocol support (TLS $\leq$1.2), reducing firmware size by $\sim$30%.

B. Memory Efficiency

- `WOLFSSL_SMALL_STACK`: Reduces stack frame sizes.
- PQC small-memory modes: Use iterative algorithms instead of lookup tables.
- Single-threaded design eliminates synchronization overhead.

C. Power-Specific Techniques

- **Dynamic clock gating:** Disable unused peripherals when idle.
- **Sleep modes:** Use WFI (Wait-For-Interrupt) during UART stalls.

Annexure: References

Standards

- RFC 9147 — *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3*
- FIPS 203 — *Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM)*
- FIPS 204 — *Module-Lattice-Based Digital Signature Standard (ML-DSA / Dilithium)*

Libraries

- WolfSSL: Embedded TLS/DTLS library
- LiteX: FPGA SoC builder framework