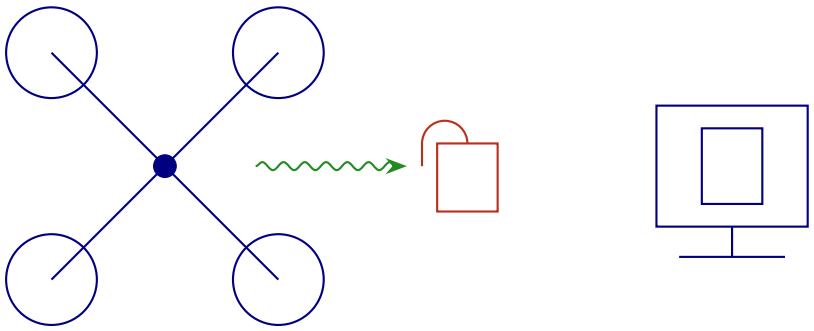
Post-Quantum Secure MAVLink Tunnel

A Comprehensive Technical Reference

From First Principles to Verified Implementation



Based on the `secure-tunnel` Research Codebase

Post-Quantum Cryptography × Unmanned Aerial Vehicles

February 2026

To every student who has ever stared at a cryptography textbook and thought, "But what does this actually look like in real code?"

And to the open-source communities behind liboqs, MAVProxy, and the countless tools that made this research possible.

Preface

What This Book Is

This book is a complete technical reference for a real, functioning post-quantum secure communication tunnel built for unmanned aerial vehicles (UAVs). It is not a theoretical exercise. It is not a simulation report. It is an explanation—from first principles to final implementation—of a system that encrypts live MAVLink telemetry between a Pixhawk flight controller on a Raspberry Pi–equipped drone and a Windows-based Ground Control Station, using algorithms designed to resist attacks from quantum computers.

Every line of code described in this book exists. Every metric shown was collected from real hardware. Every design decision was made under real constraints: limited CPU, limited battery, real-time latency requirements, and the unforgiving physics of wireless communication.

Who This Book Is For

This book is written for three audiences simultaneously:

1. **The curious student.** If you are a motivated high school or early undergraduate student who wants to understand how computers talk to each other, what encryption really means, and how drones are controlled, this book will take you from zero knowledge to genuine understanding. Every term is defined before it is used. Every concept is explained with analogies before it is formalized.

2. **The engineer.** If you are a software engineer, embedded systems developer, or network architect who needs to understand or extend this system, this book provides the complete technical specification. You will find class diagrams, protocol flows, wire formats, and code walkthroughs.

3. **The researcher.** If you are evaluating post-quantum cryptography for real-world embedded systems, this book provides honest measurements, clearly stated limitations, and the complete methodology needed to reproduce or extend the results.

How to Read This Book

The book is organized in five parts:

Part I: Foundations assumes nothing. It teaches networking, cryptography, and post-quantum cryptography from first principles. If you already know what TCP is,

what AES-GCM does, and what a lattice problem is, you may skim these chapters. But even experienced engineers have found surprises in these pages—the section on why certain classical ciphers fail against quantum computers, for example, requires careful attention even from specialists.

Part II: The Domain explains the specific problem domain: how drones communicate with ground stations using the MAVLink protocol, and why securing that communication requires a custom tunnel rather than an off-the-shelf VPN.

Part III: The Implementation is the heart of the book. It walks through every major component of the codebase: the handshake protocol, the authenticated encryption framing, the proxy engine, and the cryptographic suite system. Each chapter includes real code listings and explains not just what the code does, but *why* it was written that way.

Part IV: Orchestration and Measurement covers the benchmark orchestration system, the metrics collection pipeline, and the analytics dashboard. This is where the system goes from "it works" to "we can prove it works and measure how well."

Part V: Engineering and Reflection discusses the cross-cutting engineering decisions, trade-offs, known limitations, and future research directions.

Conventions Used

Throughout this book:

- `Monospaced text` indicates code, filenames, configuration keys, or terminal commands.
- **Bold terms** are being defined for the first time.
- *Italic terms* indicate emphasis or domain-specific terminology being referenced.
- Colored boxes highlight key insights, analogies, design decisions, security considerations, and implementation notes.
- All code listings are from the actual codebase unless explicitly noted as simplified.
- Line numbers in code listings correspond to the source files at the time of writing.

Key Insight

Blue boxes like this one highlight fundamental concepts that tie multiple topics together.

Analogy

Green boxes provide everyday analogies to help build intuition for technical concepts.

Design Decision

Orange boxes explain specific design choices made in the implementation, including trade-offs considered.

Security Note

Red boxes flag security-critical information that demands careful attention.

A Note on Honesty

This book does not pretend the system is perfect. Where measurements have limitations, we say so. Where design choices involve trade-offs, we explain both sides. Where the codebase has known issues or areas that need improvement, we discuss them openly.

Science advances through honest reporting, not through marketing. This book follows that principle.

February 2026

Contents

List of Figures

List of Tables

List of Algorithms

Glossary

AAD **Additional Authenticated Data.** Data that is integrity-protected but not encrypted. In this system, the packet header is bound as AAD so that tampering with the header causes decryption to fail.

AEAD **Authenticated Encryption with Associated Data.** A cryptographic construction that simultaneously provides confidentiality (encryption) and integrity (authentication). Examples used in this system: AES-256-GCM, ChaCha20-Poly1305, ASCON-128a.

AES **Advanced Encryption Standard.** A symmetric block cipher standardized by NIST in 2001. This system uses AES-256-GCM (256-bit key, Galois/Counter Mode).

ASCON A lightweight authenticated cipher designed for constrained environments. Winner of the NIST Lightweight Cryptography competition. This system supports the ASCON-128a variant.

C2 **Command and Control.** The communication link between a ground station and a drone that carries commands (arm, takeoff, navigate) and telemetry (position, battery, attitude).

ChaCha20-Poly1305 An AEAD cipher combining the ChaCha20 stream cipher with the Poly1305 authenticator. An alternative to AES-GCM that performs well without hardware AES support.

Classic McEliece A code-based post-quantum KEM with very large public keys but very small ciphertexts and fast operations. Based on Niederreiter's code-based cryptosystem from 1986.

DVFS **Dynamic Voltage and Frequency Scaling.** A power management technique where the CPU clock speed and voltage are adjusted dynamically. Relevant to energy measurements on the Raspberry Pi.

Epoch In this system, a one-byte counter (0–255) that is incremented on each rekey event. Combined with the sequence number to form a unique nonce for each packet.

Falcon A lattice-based digital signature algorithm based on NTRU lattices and fast Fourier sampling. Produces compact signatures but requires careful floating-point implementation.

FC **Flight Controller.** The embedded computer (typically a Pixhawk) that directly controls the drone's motors, reads sensors, and executes flight plans.

FrodoKEM A lattice-based KEM based on the Learning With Errors (LWE) problem. More conservative than ML-KEM but with larger key sizes.

GCM **Galois/Counter Mode.** An authenticated encryption mode for block ciphers. Provides both confidentiality and integrity with hardware acceleration on modern CPUs (via AES-NI).

GCS **Ground Control Station.** The operator's computer that displays telemetry, sends commands, and monitors the drone's status. In this system, a Windows PC running Mission Planner or QGroundControl.

HKDF **HMAC-based Key Derivation Function.** Used in this system to derive two directional transport keys from a single shared secret established during the handshake.

HMAC **Hash-based Message Authentication Code.** A construction for computing a keyed hash for authentication. Used in this system for drone PSK authentication during handshake.

HQC **Hamming Quasi-Cyclic.** A code-based post-quantum KEM. An alternative to lattice-based schemes.

INA219 A Texas Instruments current/voltage monitor IC used via I2C to measure power consumption of the Raspberry Pi in real time.

IV **Initialization Vector.** A value used to ensure that encrypting the same plaintext twice produces different ciphertext. In this system, the IV is deterministic (derived from epoch and sequence number) and not transmitted on the wire.

KDF **Key Derivation Function.** A function that derives one or more cryptographic keys from a shared secret. This system uses HKDF-SHA256.

KEM **Key Encapsulation Mechanism.** A public-key primitive where one party generates a public key, the other party "encapsulates" a random shared secret using that public key, and the first party "decapsulates" using their secret key. Replaces traditional key exchange (like Diffie-Hellman).

Lattice A regular grid of points in multi-dimensional space. The mathematical structure underlying ML-KEM and ML-DSA. The hardness of finding short vectors in lattices (the Shortest Vector Problem) provides security.

liboqs **Open Quantum Safe library.** A C library implementing post-quantum cryptographic algorithms. This system uses its Python bindings (`oqs-python`).

MAVLink **Micro Air Vehicle Link.** A lightweight binary protocol for communication between drones and ground stations. Supports messages for telemetry, commands, parameters, and mission plans.

MAVProxy An open-source MAVLink proxy and ground station written in Python. Used in this system to bridge serial Pixhawk data to UDP and to provide GCS-side MAVLink routing.

mDNS **Multicast DNS.** A protocol for resolving hostnames on local networks without a central DNS server. Used optionally in this system for zero-configuration drone/GCS discovery.

ML-DSA **Module Lattice Digital Signature Algorithm.** The NIST-standardized post-quantum signature scheme (FIPS 204), formerly known as CRYSTALS-Dilithium. Available at security levels 2 (ML-DSA-44), 3 (ML-DSA-65), and 5 (ML-DSA-87).

ML-KEM **Module Lattice Key Encapsulation Mechanism.** The NIST-standardized post-quantum KEM (FIPS 203), formerly known as CRYSTALS-Kyber. Available at security levels 1 (ML-KEM-512), 3 (ML-KEM-768), and 5 (ML-KEM-1024).

NIST **National Institute of Standards and Technology.** The U.S. federal agency responsible for cryptographic standards. Their post-quantum cryptography standardization process (2017–2024) produced the algorithms used in this system.

NIST Security Level A classification of post-quantum algorithm strength: Level 1 (≈AES-128), Level 3 (≈AES-192), Level 5 (≈AES-256).

Nonce **Number used once.** A value that must never be reused with the same key. In this system, constructed deterministically from the epoch byte and a monotonic sequence number.

Pixhawk An open-source flight controller hardware platform running ArduPilot or PX4 firmware. The drone in this system uses a Pixhawk connected via USB serial to a Raspberry Pi.

PQC **Post-Quantum Cryptography.** Cryptographic algorithms designed to be secure against both classical and quantum computers.

PSK **Pre-Shared Key.** A secret known to both drone and GCS before any communication begins. Used in this system for HMAC-based drone authentication during the handshake.

Rekey The process of performing a new handshake to establish fresh encryption keys, replacing the current session keys. Prevents long-lived key compromise and provides forward secrecy.

Replay Attack An attack where a previously captured legitimate packet is retransmitted. Prevented in this system by a sliding window that tracks which sequence numbers have been seen.

Selector A system call mechanism (Python `selectors` module) for monitoring multiple I/O channels simultaneously. The proxy engine uses selectors instead of `asyncio` for deterministic latency behavior.

Session ID An 8-byte random value generated during each handshake. Included in every packet header to ensure packets from old sessions are rejected.

SPHINCS+ A stateless hash-based digital signature scheme. Conservative post-quantum security based solely on hash function properties. Larger signatures than lattice-based schemes but fewer assumptions.

Suite In this system, a specific combination of (KEM, Signature, AEAD) algorithms. Example: `cs-mlkem768-aesgcm-mldsa65` uses ML-KEM-768 for key exchange, AES-256-GCM for data encryption, and ML-DSA-65 for authentication.

UAV **Unmanned Aerial Vehicle.** A drone. In this system, a quadcopter equipped with a Pixhawk flight controller and a Raspberry Pi companion computer.

UDP **User Datagram Protocol.** A connectionless transport protocol. MAVLink and the encrypted data plane in this system use UDP because it provides low latency without the overhead of TCP's reliability mechanisms.

Wire Format The exact byte layout of data as it appears "on the wire" (transmitted over the network). This system's wire format is: Header (22 bytes) $\|$ Ciphertext $\|$ Authentication Tag.

Blackout Period The interval during a rekey operation when no encrypted data is transmitted. While the old proxy is shutting down and the new proxy is performing its handshake, MAVLink packets are queued or dropped. Minimising this interval is a key engineering goal.

Bump-in-the-Wire A network security architecture where a transparent proxy is inserted into the communication path without modifying the endpoints. In this system, the PQC tunnel sits between MAVProxy and the network interface, encrypting traffic without changes to the flight controller firmware or GCS application.

Controller–Follower
: The benchmark scheduler's architecture pattern where one side (the drone, or "controller") makes all scheduling decisions and the other side (the GCS, or "follower") executes commands. This eliminates distributed consensus and simplifies error handling.

CRQC
: **Cryptographically Relevant Quantum Computer.** A quantum computer large enough to run Shor's algorithm against production-sized cryptographic keys (e.g. factoring 2048-bit RSA). No CRQC exists as of 2025, but the "harvest now, decrypt later" threat motivates PQC deployment today.

Dolev–Yao Model
: A formal adversary model for cryptographic protocol analysis where the attacker controls the entire communication channel: observing, injecting, modifying, replaying, delaying, and dropping messages. The attacker cannot break the cryptographic primitives themselves. This system's threat model (Section 14.3.1) is based on Dolev–Yao.

Domain Separation
: A cryptographic design principle where different uses of the same primitive are given distinct context strings, preventing cross-protocol attacks. In this system, HKDF key derivation includes the session ID, KEM name, and SIG name in the `info` parameter, ensuring that keys derived in different sessions or with different algorithms are cryptographically independent.

Forward Secrecy
: A property of a key agreement protocol where compromise of long-term keys does not reveal past session keys. In this system, each handshake uses ephemeral KEM keypairs; compromising the GCS signing key does not help decrypt previously recorded sessions.

Harvest Now, Decrypt Later
: An attack strategy where an adversary records encrypted traffic today and stores it until a quantum computer capable of breaking the encryption becomes available. Abbreviated HNDL. This is the primary threat motivating the use of post-quantum cryptography in this system.

Hysteresis
: A control-theory concept applied to the scheduling policy. The policy requires a telemetry condition to persist for a minimum duration (5 seconds for upgrades, 30 seconds for downgrades) before triggering a suite change, preventing rapid oscillation between suites due to transient sensor fluctuations.

ManagedProcess
: A cross-platform subprocess wrapper (`core/process.py`) that ensures child processes are terminated when the parent exits. Uses Win32 Job Objects on Windows and `PR_SET_PDEATHSIG` on Linux.

Operation Chronos
: The system's clock synchronisation protocol: a 3-message NTP-lite handshake over the TCP control channel that estimates the clock offset between drone and GCS. Named for the Greek god of time. Used to correct one-way latency measurements.

Trust Boundary
: The logical perimeter between trusted and untrusted components. In this system, the trust boundary is the network interface: plaintext exists only on localhost (trusted), while encrypted traffic crosses the WiFi LAN (untrusted). See Section 14.3.1.5.

Two-Phase Commit
: A design pattern used by the `BenchmarkPolicy` class. Phase 1 (`evaluate()`) makes a decision (advance to the next suite or stay) without modifying state. Phase 2 (`confirm_advance()`) commits the state change only after metrics have been safely recorded. This prevents metric mis-attribution.

Validation Verdict
: A pass/fail classification (Category R in the metrics schema) applied to each suite record. A suite passes if the handshake succeeded, data-plane counters are non-zero, and the duration is within bounds. See Section 12.6 for how different failure classes affect the verdict.

Part I

Foundations

Chapter 1

Introduction

> The best time to worry about security was twenty years ago. The second best time is now.
>
> —Common saying in cryptographic engineering

1.1 Why This Book Exists

Imagine you are flying a drone over a field, monitoring crops, inspecting a bridge, or delivering a medical supply package to a remote village. Your drone is in the air, a kilometer away, and you are sitting at a laptop—the **Ground Control Station (GCS)**—watching its battery level, altitude, GPS coordinates, and camera feed. You can send it commands: "fly to waypoint 3," "return to launch," "descend to 50 meters."

All of this communication—the telemetry flowing from the drone to you, and the commands flowing from you to the drone—travels through the air as radio waves. And here is the problem: *anyone with a compatible radio receiver can hear those radio waves.*

Worse, they might not just listen. They might record every packet you send, store it on a hard drive, and wait. Wait for ten years. Wait for twenty. Wait until quantum computers—machines that exploit the strange laws of quantum physics to perform certain calculations exponentially faster than any classical computer—become powerful enough to break the encryption that protects your communication today.

This scenario is not science fiction. It has a name: the **"store now, decrypt later"** attack. Intelligence agencies, corporate espionage operations, and nation-state adversaries are already collecting encrypted communications, betting that future quantum computers will let them read those messages retroactively.

For consumer drones flying over a park, this may seem like a distant concern. But for military UAVs, critical infrastructure inspection drones, or medical delivery systems, the data transmitted today may still be sensitive in twenty years. The flight patterns reveal surveillance targets. The command protocols reveal operational procedures. The telemetry reveals the capabilities and limitations of the hardware.

This book describes a system built to address this problem: a **post-quantum secure tunnel** for drone communication. The system takes the standard MAVLink protocol used by most drones in the world and wraps it in a layer of encryption that

is designed to resist attacks from both today's classical computers and tomorrow's quantum computers.

1.2 The Problem, Precisely Stated

Let us state the problem with engineering precision:

1. **MAVLink is unencrypted by default.** The MAVLink protocol [3], used by ArduPilot, PX4, and most open-source drone autopilot systems, transmits telemetry and commands as plaintext UDP packets. Anyone on the same network (or within radio range) can read, modify, or inject packets.

2. **Classical encryption is vulnerable to quantum attack.** Standard approaches like TLS with RSA or ECDHE key exchange rely on the hardness of integer factorization or discrete logarithms. Shor's algorithm [5], when executed on a sufficiently large quantum computer, can solve both problems in polynomial time, rendering these schemes useless.

3. **Drone links have unique constraints.** Unlike web browsers, drones require:

   - **Low latency:** Commands must arrive within milliseconds to maintain stable flight.
   - **Tolerance for packet loss:** UDP is used because retransmission delays (as in TCP) are unacceptable for real-time control.
   - **Limited compute:** The drone's companion computer (a Raspberry Pi) has far less processing power than a server.
   - **Limited energy:** Every millijoule spent on encryption is a millijoule not spent on flying.
   - **Key rotation:** Long flights may require periodic rekeying to limit the damage of any single key compromise.

4. **No off-the-shelf solution fits.** VPNs like WireGuard or IPsec are designed for general-purpose networking. They do not support post-quantum algorithms (at the time of implementation), do not provide per-packet algorithm identification needed for benchmarking, and add unnecessary protocol overhead for a point-to-point link.

Therefore, the system described in this book was built from scratch: a custom, purpose-built, post-quantum secure tunnel for MAVLink traffic.

1.3 What the System Does

At the highest level, the system works like this:

Analogy

Imagine two people who want to have a private conversation in a crowded room. First, they step into a quiet corner and agree on a secret code (the **handshake**).

Then they return to the crowd and speak in code (the **encrypted data plane**). If anyone overhears, all they get is gibberish. And the code they used was chosen specifically so that even a quantum-computer-equipped eavesdropper cannot crack it.

More technically:

1. The GCS starts a **proxy process** that listens for incoming connections.
2. The drone starts its own **proxy process** that connects to the GCS.
3. They perform a **TCP handshake** using post-quantum key encapsulation (to establish a shared secret) and post-quantum digital signatures (to authenticate the GCS's identity).
4. From the shared secret, they derive **two symmetric keys**: one for drone-to-GCS traffic, one for GCS-to-drone traffic.
5. All subsequent MAVLink traffic is **encrypted and authenticated** using an AEAD cipher (AES-256-GCM, ChaCha20-Poly1305, or ASCON-128a) with the derived keys.
6. The proxies sit transparently between the applications and the network. MAVProxy and Mission Planner do not know the tunnel exists—they send and receive plaintext MAVLink as usual.

The system supports over 70 different **cipher suites**—combinations of key encapsulation, signature, and encryption algorithms—allowing comprehensive benchmarking of different post-quantum approaches on real hardware.

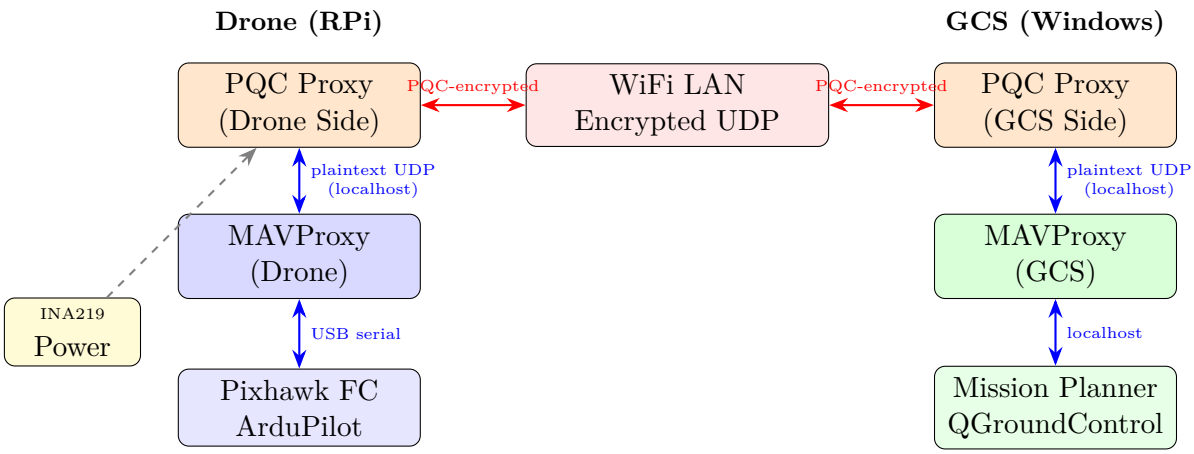


Figure 1.1: System overview: the PQC proxy sits transparently between applications and the network. All traffic crossing the WiFi link is post-quantum encrypted.

1.4 The Hardware

This is not a simulation. The system runs on real hardware:

Drone side: A Raspberry Pi (ARM-based single-board computer) connected to a Pixhawk flight controller via USB serial. The Pixhawk runs ArduPilot firmware and controls the physical drone. The Raspberry Pi runs the secure tunnel proxy, MAVProxy, and metrics collection.

GCS side: A Windows laptop running Mission Planner or QGroundControl for flight control visualization, plus the secure tunnel proxy and benchmark orchestration software.

Network: Both devices are connected via a WiFi LAN (local area network). The encrypted traffic traverses the WiFi link; plaintext traffic stays on localhost (127.0.0.1).

Power measurement: An INA219 current sensor on the I2C bus of the Raspberry Pi measures real-time voltage, current, and power consumption at 1 kHz sampling rate.

1.5 The Software Stack

The codebase is written primarily in Python and is organized into three major layers:

`core/` The tunnel engine. This contains the handshake protocol, AEAD encryption/decryption, the UDP proxy, the suite registry, configuration management, metrics collection, power monitoring, and all cryptographic logic. This is the most security-critical code.

`sscheduler/` The orchestration layer. This contains the benchmark scheduler that cycles through all cipher suites, manages proxy processes on both drone and GCS, collects metrics, and saves results. The drone acts as the controller; the GCS is a follower that responds to commands.

`dashboard/` The analytics layer. A FastAPI backend loads benchmark results from JSON files, computes derived metrics, and serves them via a REST API. A React/TypeScript frontend visualizes the data with interactive charts, tables, and comparison tools.

The cryptographic primitives are provided by **liboqs** (Open Quantum Safe) [4], a C library that implements all NIST-standardized and candidate post-quantum algorithms. The system accesses liboqs through its Python bindings (`oqs-python`).

1.6 The Scope of This Book

This book explains the *entire* system—from the theoretical foundations of post-quantum cryptography through every line of implementation code to the deployment procedures and benchmark results. Table 1.1 provides a chapter-by-chapter map.

Table 1.1: Chapter map and reading guide.

Ch.	Title	What You Will Learn
1	Introduction	Motivation, system overview, hardware/software
2	Networking	TCP/IP, UDP, sockets, NAT, mDNS
3	Cryptography	Symmetric/asymmetric, AEAD, key exchange, signatures
4	Post-Quantum	Lattices, codes, hashes; NIST standards
5	MAVLink	Drone protocol, packet format, security gaps
6	Architecture	System design, data flow, component interactions
7	Handshake	2-message PQC authenticated key exchange
8	AEAD Framing	Wire format, replay protection, epoch management
9	Proxy Engine	Selectors loop, socket architecture, rekey integration
10	Suite Registry	KEM×SIG×AEAD composition, alias resolution
11	Scheduler	Drone-controlled cycling, GCS follower, MAVProxy mode
12	Metrics	18-category schema, collectors, aggregation
13	Dashboard	FastAPI backend, React frontend, visualization
14	Engineering	Cross-platform, error handling, lessons learned
15	Conclusion	Contributions, findings, future work
16	Dependencies	Every package, its role, version constraints
17	Codebase Walkthrough	Module-by-module tour of all 97 Python files
18	Testing	18 test files, patterns, how to run the suite
19	Benchmarks	19,600 measurements, KEM/SIG/AEAD/power results
20	Deployment	Step-by-step setup, launch, and troubleshooting
A	Configuration Keys	Complete CONFIG dictionary reference
B	Wire Protocol	Byte-level wire format specification
C	Metrics Schema	All 231 metric fields across 18 categories

1.7 A Word About Reading Order

If you are a student encountering these topics for the first time, read the book sequentially. Each chapter builds on the previous one.

If you are an engineer who already understands networking and cryptography, you may skip to Part II (Chapter 5) and refer back to the Glossary and foundational chapters as needed.

If you are a researcher interested specifically in the post-quantum performance measurements, jump directly to Chapter 19 for the benchmark results, then read Chapters 12 and 13 for context on how the data was collected and visualised.

If you are deploying the system, start with Chapter 20 for the step-by-step setup guide, then read Chapter 16 for the complete software dependency map.

Let us begin.

Chapter 2

Networking Fundamentals

To understand secure communication, you must first understand communication.

Before we can protect drone traffic, we need to understand how computers communicate at all. This chapter builds that understanding from the ground up.

2.1 What Is a Network?

Analogy

A **computer network** is like a postal system for data. Just as the postal system has addresses (street addresses), delivery vehicles (trucks, planes), and rules about how to format an envelope, a computer network has addresses (IP addresses), physical media (cables, radio waves), and protocols (rules about how to format and deliver data).

Definition 2.1 (Network)**.** A **network** is a system of two or more computing devices connected by communication links that can exchange data according to agreed-upon rules called **protocols**.

The internet is the largest network, connecting billions of devices. A **local area network (LAN)** connects devices in a small area—a home, an office, or, in our case, a field where a drone is flying. The drone and the ground control station in this system are connected by a WiFi LAN.

2.2 The Layered Model

Network communication is complex. To manage that complexity, engineers organize networking into **layers**, where each layer handles one aspect of communication and relies on the layers below it.

The most common model has four layers (the TCP/IP model):

Table 2.1: The TCP/IP layered model

Layer	Name	Responsibility
4	Application	What the data means (HTTP for web, MAVLink for drones)
3	Transport	Reliable or best-effort delivery between processes (TCP, UDP)
2	Internet	Routing packets across networks using IP addresses
1	Link	Moving bits across a single physical link (Ethernet, WiFi)

Analogy

Think of mailing a letter. The **application layer** is the letter's content. The **transport layer** is the decision to send it by certified mail (reliable, like TCP) or regular mail (best-effort, like UDP). The **internet layer** is the postal routing system that moves the envelope from city to city. The **link layer** is the truck that carries the envelope on a specific road segment.

Each layer adds its own header to the data (like putting a letter in an envelope, then putting that envelope in a shipping box). This process is called **encapsulation**. The receiving side strips these headers in reverse order, a process called **decapsulation**.

2.3 IP Addresses

Every device on a network has an **IP address**—a numerical label that uniquely identifies it.

Definition 2.2 (IPv4 Address)**.** An **IPv4 address** is a 32-bit number, typically written as four decimal numbers separated by dots. Example: `192.168.0.100`.

In this system:

- The drone's Raspberry Pi has IP `192.168.0.100` on the LAN.
- The GCS laptop has IP `192.168.0.101` on the LAN.
- Both also have Tailscale VPN addresses (`100.x.x.x`), but these are used only for SSH maintenance, never for real-time data.

There is a special IP address: `127.0.0.1`, called the **loopback address** or **localhost**. Packets sent to 127.0.0.1 never leave the machine—they are delivered internally. This is important because the plaintext MAVLink traffic in this system *only* travels on localhost, never across the network.

Security Note

The separation between plaintext (localhost only) and ciphertext (network-facing) is a deliberate security boundary. If plaintext traffic were accidentally exposed on the network interface, the entire purpose of the tunnel would be defeated.

2.4 Ports

An IP address identifies a *machine*, but a machine may be running many programs simultaneously. A **port number** identifies a specific program (or "service") on that machine.

Definition 2.3 (Port)**.** A **port** is a 16-bit number (1–65535) that, combined with an IP address, identifies a specific communication endpoint on a machine. The combination of IP address and port is called a **socket address**.

Analogy

If an IP address is like a building's street address, a port number is like an apartment number within that building. The address gets the mail to the building; the apartment number gets it to the right resident.

This system uses many ports. Here are the most important ones:

Table 2.2: Key ports used by the PQC tunnel system

Port	Protocol	Side	Purpose
46000	TCP	GCS	Handshake server — listens for drone connections
46011	UDP	GCS	Encrypted data reception from drone
46012	UDP	Drone	Encrypted data reception from GCS
47001	UDP	GCS	Plaintext ingress (app → tunnel)
47002	UDP	GCS	Plaintext egress (tunnel → app)
47003	UDP	Drone	Plaintext ingress (MAVProxy → tunnel)
47004	UDP	Drone	Plaintext egress (tunnel → MAVProxy)
48080	TCP	Both	Scheduler control channel

2.5 TCP: The Reliable Protocol

Definition 2.4 (Transmission Control Protocol (TCP))**. TCP** is a transport-layer protocol that provides reliable, ordered, error-checked delivery of data between applications. It establishes a connection before data transfer (a "three-way handshake"), retransmits lost packets, and ensures data arrives in the correct order.

TCP is used in this system for the **cryptographic handshake** (establishing keys) because:

1. The handshake is a multi-step conversation (server hello, client response) that must happen in order.

2. Losing a handshake message would be catastrophic—the keys would be wrong.

3. The handshake happens once (or occasionally during rekey), so TCP's overhead is acceptable.

2.5.1 How TCP Works

TCP guarantees delivery through a simple but powerful mechanism:

1. The sender transmits a segment of data and starts a timer.
2. The receiver sends an **acknowledgment (ACK)** confirming receipt.
3. If the sender's timer expires without receiving an ACK, it retransmits the segment.
4. Sequence numbers ensure data is reassembled in order even if segments arrive out of order.

Key Insight

TCP's reliability comes at a cost: **latency**. If a packet is lost, the sender must wait for a timeout (typically 200ms to several seconds), retransmit, and wait for the acknowledgment. For real-time drone control, this delay is unacceptable—which is why the data plane uses UDP instead.

2.6 UDP: The Fast Protocol

Definition 2.5 (User Datagram Protocol (UDP))**. UDP** is a transport-layer protocol that provides unreliable, unordered delivery of individual datagrams. It has no connection setup, no acknowledgments, and no retransmission. Each packet is independent.

UDP is used in this system for the **encrypted data plane** (actual MAVLink traffic) because:

1. **Low latency:** UDP adds essentially zero overhead beyond the IP layer. A packet is sent immediately.
2. **Tolerance for loss:** MAVLink was designed for UDP. If a telemetry packet is lost, the next one (arriving milliseconds later) carries updated data, making the lost packet irrelevant.
3. **No head-of-line blocking:** In TCP, if packet 5 is lost, packets 6, 7, 8 are held until packet 5 is retransmitted. In UDP, packets 6, 7, 8 are delivered immediately.

Analogy

TCP is like a phone call: you establish a connection, speak in order, and know the other person heard you. UDP is like shouting across a field: your message goes out immediately, but you do not know if anyone heard it, and your words might arrive jumbled if there is an echo.

2.6.1 UDP Datagram Structure

A UDP datagram is simple:

Table 2.3: UDP header structure (8 bytes total)

Offset	Size	Field	Description
0	2 bytes	Source Port	Sender's port number
2	2 bytes	Destination Port	Receiver's port number
4	2 bytes	Length	Total datagram length
6	2 bytes	Checksum	Error detection (optional in IPv4)

The entire UDP header is only 8 bytes. Compare this to TCP's 20-byte minimum header (often 32 bytes with options). For small MAVLink packets (typically 50–280 bytes), this overhead difference matters.

2.7 Sockets: The Programming Interface

Definition 2.6 (Socket)**.** A **socket** is a programming interface (API) for network communication. It represents one endpoint of a two-way communication link. A program creates a socket, binds it to an address and port, and then sends or receives data through it.

In Python, the `socket` module provides this interface:

```
import socket

# Create a UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind to a specific address and port
sock.bind(("127.0.0.1", 47001))

# Receive a packet (up to 65535 bytes)
data, sender_address = sock.recvfrom(65535)

# Send a packet to a specific destination
sock.sendto(encrypted_data, ("192.168.0.100", 46012))
```

Listing 2.1: Creating and using a UDP socket in Python

The proxy engine in this system manages four sockets simultaneously:

1. A plaintext-receiving socket (bound to localhost, receives from the local application).
2. A plaintext-sending socket (sends decrypted data to the local application).
3. An encrypted-receiving socket (bound to the network interface, receives from the remote peer).
4. An encrypted-sending socket (sends encrypted data to the remote peer).

2.7.1 Selectors: Watching Multiple Sockets

When a program needs to watch multiple sockets at once ("has data arrived on socket A or socket B?"), it uses a **selector**—an operating system mechanism that efficiently monitors multiple I/O channels.

```
import selectors

sel = selectors.DefaultSelector()
sel.register(plaintext_sock, selectors.EVENT_READ, "plaintext_in")
sel.register(encrypted_sock, selectors.EVENT_READ, "encrypted")

while True:
    events = sel.select(timeout=0.1)  # Wait up to 100ms
    for key, mask in events:
        if key.data == "plaintext_in":
            handle_plaintext_packet(key.fileobj)
        elif key.data == "encrypted":
            handle_encrypted_packet(key.fileobj)
```

Listing 2.2: Using selectors to monitor multiple sockets

Design Decision

The proxy engine uses Python's `selectors` module instead of `asyncio` (Python's asynchronous I/O framework). This was a deliberate choice: selectors provide more deterministic latency behavior because they avoid the overhead of coroutine scheduling, task queues, and event loop callbacks that `asyncio` introduces. For a latency-sensitive real-time proxy, predictability matters more than convenience.

2.8 Network Address Translation (NAT)

In many network setups, devices do not have globally unique IP addresses. Instead, a router performs **Network Address Translation (NAT)**, mapping internal addresses (like `192.168.0.x`) to a single external address.

NAT creates a complication for our system: when the drone sends a packet, the router may change the source port. If the GCS is configured to accept packets only from the drone's expected address and port (`STRICT_UDP_PEER_MATCH = True`), these translated packets will be rejected.

Design Decision

The system defaults to `CONFIG["STRICT_UDP_PEER_MATCH"] = True` for security (preventing IP spoofing attacks) but allows disabling it when operating behind NAT. This is a classic security-vs-usability trade-off.

2.9 Bandwidth, Latency, and Throughput

Three metrics characterize network performance:

Definition 2.7 (Bandwidth)**. Bandwidth** is the maximum rate at which data can be transmitted over a link, measured in bits per second (bps). A typical WiFi link provides 20–100 Mbps.

Definition 2.8 (Latency)**. Latency** is the time it takes for a single packet to travel from sender to receiver, measured in milliseconds (ms). On a local WiFi LAN, latency is typically 1–5 ms.

Definition 2.9 (Throughput)**. Throughput** (or **goodput**) is the actual rate of useful data transfer, after subtracting protocol overhead, retransmissions, and errors. It is always less than or equal to bandwidth.

Key Insight

For drone communication, **latency matters more than bandwidth**. A typical MAVLink telemetry stream requires only 10–50 kbps—trivial for any modern network. But if a "return to home" command arrives 500ms late because of TCP retransmission or encryption overhead, the drone may have already flown into an obstacle.

2.9.1 Jitter

Definition 2.10 (Jitter)**. Jitter** is the variation in latency over time. If packets normally arrive every 10ms but occasionally arrive after 15ms or 5ms, the jitter is the spread of those arrival times.

High jitter is problematic for drone control because the autopilot expects telemetry at regular intervals. If heartbeat messages arrive irregularly, the autopilot may incorrectly conclude that the communication link has failed and trigger a failsafe (automatic return to home or landing).

2.10 The Data Flow in Our System

Now we can describe the complete data flow with proper networking terminology:

1. The **Pixhawk flight controller** generates MAVLink messages and sends them via USB serial to the Raspberry Pi.

2. **MAVProxy** on the Pi reads the serial data, decodes MAVLink frames, and forwards each frame as a UDP datagram to `127.0.0.1:47003` (the drone proxy's plaintext ingress port).

3. The **drone proxy** receives the plaintext datagram on its localhost socket. It encrypts the payload using the current AEAD key, prepends the wire header, and sends the resulting ciphertext datagram from the Pi's WiFi interface to `192.168.0.101:46011` (the GCS proxy's encrypted receive port).

4. The ciphertext traverses the **WiFi LAN** as a standard UDP/IP packet. Any eavesdropper sees only the encrypted datagram.

5. The **GCS proxy** receives the encrypted datagram on port 46011. It validates the header, checks the replay window, decrypts the payload, and sends the recovered plaintext datagram to `127.0.0.1:47002` (the GCS application's receive port).

6. **Mission Planner** (or QGroundControl) receives the plaintext MAVLink datagram and displays the drone's telemetry.

The return path (GCS commands to drone) follows the exact inverse: plaintext from the GCS app enters on port 47001, is encrypted and sent to `192.168.0.100:46012`, decrypted on the drone side, and delivered to MAVProxy on port 47004.

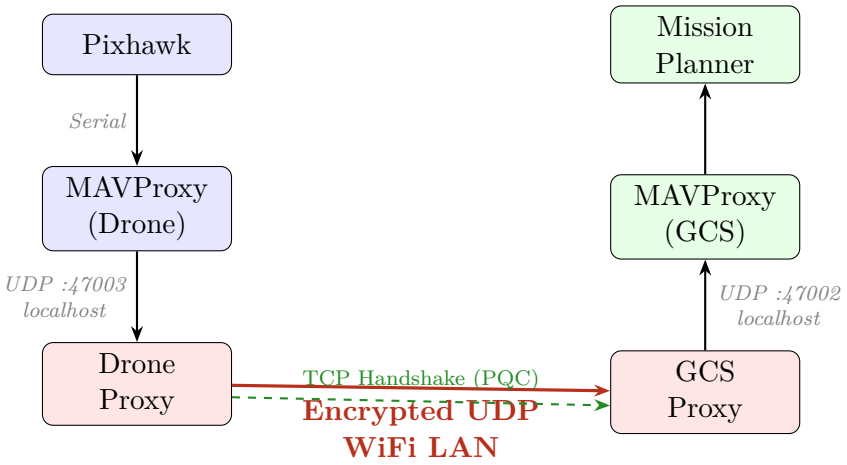


Figure 2.1: Data flow through the PQC tunnel system

2.11 Summary

In this chapter, we established the networking foundations needed to understand the tunnel:

- Networks use **layered protocols** to manage complexity.
- **IP addresses** identify machines; **ports** identify programs.
- **TCP** provides reliable delivery (used for handshakes).
- **UDP** provides fast, unreliable delivery (used for the data plane).
- **Sockets** are the programming interface for network I/O.
- **Selectors** allow monitoring multiple sockets simultaneously.
- The tunnel separates **plaintext** traffic (localhost) from **encrypted** traffic (network).

With this foundation, we are ready to explore the other half of the equation: the cryptography that makes the tunnel secure.

Chapter 3

Cryptography Fundamentals

> Cryptography is the art of writing or solving codes. In practice, it is the science of keeping secrets from people who are very, very good at finding them.

This chapter introduces the essential ideas of cryptography that underpin the secure tunnel. We start with the simplest possible concepts and build toward the specific constructions used in the system.

3.1 Why Cryptography?

When the drone sends its GPS coordinates to the ground station over WiFi, those coordinates travel as radio waves through the air. Anyone with a compatible receiver can capture those radio waves and read the data inside them. This is not a theoretical concern—it is trivially easy with commodity hardware.

Cryptography provides three fundamental protections:

Confidentiality: Ensuring that only the intended recipient can read the message. Even if an attacker captures the radio waves, they see only meaningless noise.

Integrity: Ensuring that the message has not been modified in transit. If an attacker changes a "fly north" command to "fly south," the recipient detects the tampering.

Authentication: Ensuring that the message came from who it claims to come from. An attacker cannot forge messages that appear to come from the legitimate ground station.

Analogy

Imagine sending a letter through the postal system. **Confidentiality** is sealing the letter in an envelope so the mail carrier cannot read it. **Integrity** is applying a tamper-evident seal so the recipient knows if someone opened the envelope. **Authentication** is signing the letter with your unique signature so the recipient knows it is really from you.

The secure tunnel in this system provides all three properties simultaneously.

3.2 Symmetric Encryption

The simplest form of encryption uses a single shared secret, called a **key**.

Definition 3.1 (Symmetric Encryption)**. Symmetric encryption** (also called secret-key encryption) uses the same key for both encryption and decryption:

$$\text{ciphertext} = \text{Encrypt}(\text{key}, \text{plaintext}) \tag{3.1}$$

$$\text{plaintext} = \text{Decrypt}(\text{key}, \text{ciphertext}) \tag{3.2}$$

Analogy

Symmetric encryption is like a lockbox with a single key. You put your message in the box, lock it, and send it. The recipient uses an identical copy of the key to unlock the box and read the message. The crucial assumption is that both parties already have the same key.

3.2.1 The Key Problem

The fundamental challenge of symmetric encryption is: *how do both parties get the same key?* You cannot send the key over the insecure channel (the attacker would capture it). You cannot agree on it in advance for every possible pair of communicants (the number of keys grows quadratically).

This is called the **key distribution problem**, and it is one of the central motivations for public-key cryptography, which we will discuss shortly.

In our system, the symmetric keys used for the data plane are established during the handshake using public-key (asymmetric) techniques. Once established, all data-plane encryption uses fast symmetric algorithms.

3.2.2 AES: The Advanced Encryption Standard

Definition 3.2 (AES)**.** The **Advanced Encryption Standard (AES)** is a symmetric block cipher that encrypts data in 128-bit (16-byte) blocks. It supports key sizes of 128, 192, or 256 bits. This system uses **AES-256**—the strongest variant, with a 256-bit (32-byte) key.

AES was selected by NIST in 2001 through an open competition and has been the global standard for symmetric encryption ever since. It is implemented in hardware on virtually all modern CPUs through the **AES-NI** instruction set, making it extremely fast.

Key Insight

AES (and all symmetric ciphers) is **not threatened by quantum computers**—at least not fatally. Grover's algorithm [2] can speed up brute-force search of symmetric keys, but it only halves the effective security. AES-256 drops to 128-bit

security against quantum attack, which is still considered safe. This is why we use AES-256 rather than AES-128.

3.2.3 Stream Ciphers vs. Block Ciphers

AES is a **block cipher**: it encrypts fixed-size blocks. To encrypt messages longer than one block, we need a **mode of operation** (discussed in Section 3.5).

An alternative is a **stream cipher**, which generates a continuous stream of pseudo-random bytes (a "keystream") and XORs it with the plaintext:

$$\text{ciphertext}[i] = \text{plaintext}[i] \oplus \text{keystream}[i] \tag{3.3}$$

ChaCha20 is a stream cipher used in this system (as part of ChaCha20-Poly1305). It was designed by Daniel Bernstein as a faster, more secure alternative to earlier stream ciphers, and it performs particularly well on hardware without AES instructions (like the Raspberry Pi's ARM CPU).

3.3 Public-Key Cryptography

Definition 3.3 (Public-Key (Asymmetric) Cryptography)**.** In **public-key cryptography**, each party has two mathematically related keys: a **public key** (which can be freely shared) and a **private key** (which must be kept secret). Data encrypted with the public key can only be decrypted with the corresponding private key.

Analogy

Public-key cryptography is like a mailbox with a slot. Anyone can drop a letter into the slot (encrypt with the public key), but only the person with the mailbox key can open it and read the letters inside (decrypt with the private key).

Public-key cryptography solves the key distribution problem: you can publish your public key to the world, and anyone can use it to send you encrypted messages. Only you, with your private key, can read them.

3.3.1 Key Exchange

The most common use of public-key cryptography is **key exchange**: establishing a shared symmetric key between two parties who have never communicated before.

Classically, this is done with algorithms like Diffie-Hellman or RSA. In our post-quantum system, it is done with a **Key Encapsulation Mechanism (KEM)**, which we will explore in detail in Chapter 4.

3.3.2 Digital Signatures

Definition 3.4 (Digital Signature)**.** A **digital signature** is a mathematical scheme for verifying the authenticity and integrity of a message. The signer uses their private key to produce a signature; anyone can verify it using the signer's public key.

$$\sigma = \text{Sign}(\text{private\_key}, \text{message}) \tag{3.4}$$

$$\text{valid} = \text{Verify}(\text{public\_key}, \text{message}, \sigma) \tag{3.5}$$

In this system, digital signatures are used during the handshake to **authenticate the GCS**. The GCS signs the handshake transcript with its private key, and the drone verifies the signature using the GCS's public key (which was pre-installed on the drone). This prevents man-in-the-middle attacks.

Security Note

Authentication is critical. Without it, an attacker could impersonate the GCS, complete a handshake with the drone, and gain full command-and-control access. The drone would be flying under the attacker's orders, believing it was communicating with the legitimate operator.

3.4 Hash Functions

Definition 3.5 (Cryptographic Hash Function)**.** A **cryptographic hash function** takes an input of arbitrary length and produces a fixed-size output (the "hash" or "digest"). It has three key properties:

1. **Preimage resistance:** Given a hash output, it is infeasible to find the input.
2. **Second preimage resistance:** Given an input and its hash, it is infeasible to find a different input with the same hash.
3. **Collision resistance:** It is infeasible to find any two different inputs with the same hash.

This system uses **SHA-256** (producing 256-bit / 32-byte hashes) for:

- HKDF-based key derivation (deriving transport keys from the shared secret).
- HMAC-based authentication (verifying the drone's identity with a pre-shared key).

Analogy

A hash function is like a fingerprint machine for data. You feed in a document of any length, and it produces a unique, fixed-size "fingerprint." Even a tiny change to the document—flipping a single bit—completely changes the fingerprint. And you cannot reconstruct the original document from just the fingerprint.

3.4.1 HMAC: Hash-Based Authentication

Definition 3.6 (HMAC)**. HMAC** (Hash-based Message Authentication Code) combines a hash function with a secret key to produce an authentication tag. Only someone who knows the key can produce a valid tag, and any tampering with the message invalidates the tag.

$$\text{tag} = \text{HMAC}(\text{key}, \text{message}) = H\Big((\text{key} \oplus \text{opad}) \parallel H((\text{key} \oplus \text{ipad}) \parallel \text{message})\Big) \quad (3.6)$$

In this system, HMAC-SHA256 is used during the handshake for drone authentication: the drone proves it knows the pre-shared key (PSK) by computing an HMAC over the server hello message.

3.4.2 HKDF: Deriving Keys from Secrets

Definition 3.7 (HKDF)**. HKDF** (HMAC-based Key Derivation Function) takes a shared secret and derives one or more cryptographic keys from it. It works in two phases: "extract" (compress the secret into a fixed-size pseudorandom key) and "expand" (derive the desired number of output bytes).

After the KEM handshake produces a shared secret, the system uses HKDF-SHA256 to derive two 32-byte transport keys:

```
info = b"pq-drone-gcs:kdf:v1|" + session_id + b"|" +
    kem_name + b"|" + sig_name
hkdf = HKDF(algorithm=SHA256(), length=64,
             salt=b"pq-drone-gcs|hkdf|v1", info=info)
okm = hkdf.derive(shared_secret)
key_drone_to_gcs = okm[:32]     # First 32 bytes
key_gcs_to_drone = okm[32:64]   # Second 32 bytes
```

Listing 3.1: Key derivation from shared secret

Security Note

The `info` parameter includes the session ID, KEM name, and signature name. This ensures that even if two sessions happen to produce the same shared secret (astronomically unlikely but theoretically possible), the derived keys will be different because the session context differs. This is called **domain separation**.

3.5 Authenticated Encryption with Associated Data (AEAD)

The data plane needs both encryption (confidentiality) and authentication (integrity) for every packet. **AEAD** provides both in a single operation.

Definition 3.8 (AEAD)**. Authenticated Encryption with Associated Data (AEAD)** is a class of encryption algorithms that simultaneously:

1. **Encrypts** the plaintext into ciphertext (confidentiality).
2. **Authenticates** both the ciphertext and any "associated data" (integrity and authenticity).

The associated data (AAD) is data that must be integrity-protected but not encrypted—in this system, the packet header.

$$(\text{ciphertext}, \text{tag}) = \text{AEAD-Encrypt}(\text{key}, \text{nonce}, \text{plaintext}, \text{AAD}) \tag{3.7}$$

$$\text{plaintext} = \text{AEAD-Decrypt}(\text{key}, \text{nonce}, \text{ciphertext}, \text{tag}, \text{AAD}) \tag{3.8}$$

If any of the inputs are wrong—wrong key, wrong nonce, tampered ciphertext, or tampered AAD—decryption fails with an authentication error.

3.5.1 The Three AEAD Algorithms in This System

AES-256-GCM
: The combination of AES-256 in Galois/Counter Mode. GCM provides both encryption and authentication through polynomial arithmetic over a Galois field. It is hardware-accelerated on most CPUs. Nonce: 12 bytes. Tag: 16 bytes.

ChaCha20-Poly1305
: A combination of the ChaCha20 stream cipher (for encryption) and the Poly1305 MAC (for authentication). Designed by Daniel Bernstein, it is the primary alternative to AES-GCM and performs well on CPUs without AES hardware support. Nonce: 12 bytes. Tag: 16 bytes.

ASCON-128a
: A lightweight AEAD designed for constrained environments. Winner of the NIST Lightweight Cryptography competition. Uses a sponge-based construction with smaller state (320 bits). Nonce: 16 bytes. Tag: 16 bytes. Included in this system for benchmarking on resource-constrained hardware.

Design Decision

The system supports three AEAD algorithms to enable fair comparison. AES-GCM benefits from hardware acceleration (AES-NI), ChaCha20-Poly1305 benefits from pure software efficiency on ARM, and ASCON-128a targets the lightweight embedded niche. Benchmarking all three on the same hardware reveals which is truly optimal for the specific platform.

3.5.2 Nonces and Why They Matter

Definition 3.9 (Nonce)**.** A **nonce** ("number used once") is a value that must be unique for every encryption operation with the same key. Reusing a nonce with the same key is a catastrophic security failure for most AEAD algorithms.

For AES-GCM, nonce reuse allows an attacker to recover the authentication key and forge packets. This is not a theoretical concern—it has been exploited in real-world attacks.

In this system, nonces are **deterministic**—derived from the epoch byte and a monotonic sequence number:

$$\text{nonce}(12 \text{ bytes}) = \underbrace{\text{epoch}}_{1 \text{ byte}} \parallel \underbrace{\text{sequence}}_{11 \text{ bytes}} \tag{3.9}$$

This design guarantees uniqueness as long as:

1. The sequence counter never wraps (with 11 bytes = 2^{88} possible values, this is effectively impossible).
2. The epoch is incremented on each rekey.
3. A new handshake generates new keys before the epoch reaches 255.

Key Insight

The nonce is **not transmitted on the wire**. Both sides can reconstruct it from the sequence number and epoch, which are part of the packet header. This saves 12 bytes per packet—a meaningful optimization when MAVLink packets are only 50–280 bytes.

3.6 Replay Protection

Definition 3.10 (Replay Attack)**.** A **replay attack** is when an attacker captures a legitimate encrypted packet and retransmits it later. The packet decrypts successfully (it is a valid ciphertext with a valid tag), so the receiver processes it as if it were new.

Analogy

Imagine someone recording you saying "Transfer $100 to account X" and playing the recording again and again. Each time, the bank hears a valid, authenticated request and transfers another $100.

The solution is a **sliding window**:

1. The receiver tracks the highest sequence number it has seen so far (high).
2. It maintains a bitmask of which recent sequence numbers (within a configurable **window**) have been received.
3. When a new packet arrives:
   - If its sequence number is *above* high: accept and update the window.
   - If it is *within the window* but not yet seen: accept and mark as seen.
   - If it is *within the window* and already seen: reject (duplicate/replay).
   - If it is *below the window*: reject (too old).

This system uses a window of 1024 packets by default (`CONFIG["REPLAY_WINDOW"] = 1024`). This is large enough to tolerate significant packet reordering (common on WiFi) while still detecting replays.

3.7 Putting It All Together

Let us trace what happens cryptographically when the drone sends a single MAVLink packet:

1. The drone proxy receives a plaintext MAVLink datagram (e.g., a GPS position update, 50 bytes).
2. It constructs a **header** (22 bytes) containing: wire version, KEM/SIG algorithm IDs, 8-byte session ID, 8-byte sequence number, and epoch byte.
3. It constructs a **nonce** (12 bytes) from the epoch and sequence number.
4. It calls `AEAD-Encrypt(key_d2g, nonce, plaintext, header)`, which:
   - Encrypts the 50-byte plaintext into 50 bytes of ciphertext.
   - Produces a 16-byte authentication tag over the ciphertext *and* the header.
5. It sends the wire packet: header(22) $\|$ ciphertext+tag(66) = 88 bytes total.
6. The GCS proxy receives the 88-byte packet, validates the header, checks the replay window, reconstructs the nonce, and calls `AEAD-Decrypt`.
7. If decryption succeeds: the 50-byte plaintext MAVLink message is delivered to Mission Planner. If it fails: the packet is silently dropped.

Key Insight

The header is sent in cleartext (it is not encrypted), but it **is** authenticated as AAD. This means an attacker can read the header (learning the algorithm IDs and sequence number) but cannot modify it without invalidating the authentication tag. The trade-off is deliberate: the header must be readable for the receiver to know which key and nonce to use for decryption.

3.8 The Quantum Threat

Everything described so far—AES, HKDF, HMAC, AEAD—uses **symmetric** cryptography, which is largely safe from quantum computers (at double the key length).

The vulnerability lies in the **public-key** operations: key exchange and digital signatures. The classical algorithms used for these—RSA, Diffie-Hellman, ECDSA, ECDHE—are based on mathematical problems that quantum computers can solve efficiently:

- **Integer factorization** (RSA): Shor's algorithm factors large integers in polynomial time on a quantum computer. A 2048-bit RSA key, requiring $\sim 2^{112}$ classical operations to break, requires only $\sim 2^{20}$ quantum gates.
- **Discrete logarithm** (Diffie-Hellman, ECDH, ECDSA): Shor's algorithm also computes discrete logarithms efficiently, breaking all elliptic-curve cryptography.

Security Note

This is why the system uses **post-quantum** algorithms for key exchange (KEM) and digital signatures. The symmetric-key operations (AES-GCM, ChaCha20-Poly1305) are already quantum-resistant at the key sizes used.

The next chapter explores these post-quantum algorithms in detail: what mathematical problems they are based on, why those problems are believed to be hard for quantum computers, and how the specific algorithms in this system (ML-KEM, ML-DSA, Falcon, SPHINCS+, Classic McEliece, HQC) work.

3.9 Summary

- Cryptography provides **confidentiality**, **integrity**, and **authentication**.
- **Symmetric encryption** (AES, ChaCha20) uses a single shared key—fast but requires key distribution.
- **Public-key cryptography** solves key distribution using key pairs (public/private).
- **Digital signatures** authenticate messages using public-key techniques.
- **Hash functions** (SHA-256) produce fixed-size fingerprints of data.
- **HMAC** provides keyed authentication; **HKDF** derives keys from shared secrets.
- **AEAD** (AES-GCM, ChaCha20-Poly1305, ASCON) provides encryption + authentication in one operation.
- **Nonces** must be unique per key; this system derives them deterministically from epoch + sequence.
- **Replay protection** uses a sliding window to detect duplicate or retransmitted packets.
- **Quantum computers** threaten public-key algorithms but not symmetric ones (at sufficient key length).

Chapter 4

Post-Quantum Cryptography

> We should be worried about quantum computers not because they are coming next year, but because encrypted data captured today can be decrypted retroactively once they arrive.

Chapter 3 established the foundations of symmetric and public-key cryptography. This chapter explains *why* classical public-key algorithms are vulnerable to quantum computers and introduces the specific post-quantum algorithms that this system uses.

4.1 Quantum Computing in Two Pages

A **classical computer** stores information as bits—each bit is either 0 or 1. A **quantum computer** stores information as **qubits**, which can be in a **superposition** of both 0 and 1 simultaneously.

Analogy

A classical bit is like a coin lying on a table: it is either heads (0) or tails (1). A qubit is like a coin spinning in the air: until it lands (is **measured**), it is, in a precise mathematical sense, both heads and tails at the same time.

Two key properties make quantum computers powerful:

Superposition: A system of n qubits can represent all 2^n possible states simultaneously. With 256 qubits, a quantum computer can (in a limited sense) process 2^{256} states in parallel—more than the number of atoms in the observable universe.

Entanglement: Qubits can be correlated in ways that have no classical analogue. Measuring one entangled qubit instantly constrains the state of its partner, enabling quantum algorithms to amplify correct answers and suppress incorrect ones.

Key Insight

Quantum computers are **not** just "faster classical computers." They cannot speed up every computation. They are powerful because certain mathematical problems—including the ones that underpin RSA, Diffie-Hellman, and elliptic curve cryptography—have efficient quantum algorithms.

4.1.1 Shor's Algorithm

In 1994, Peter Shor discovered quantum algorithms for two problems:

1. **Integer factorization:** Given a composite number $N = p \times q$, find p and q.
2. **Discrete logarithm:** Given g, p, and $g^x \mod p$, find x.

Classical computers require roughly 2^{112} operations to factor a 2048-bit RSA key. A sufficiently large quantum computer running Shor's algorithm requires only a polynomial number of quantum gates—roughly $O(n^3)$ where n is the number of bits. This means:

- **RSA:** Broken. Key exchange and signatures based on factoring are insecure.
- **Diffie-Hellman / ECDH:** Broken. Key exchange based on discrete logarithms is insecure.
- **ECDSA / EdDSA:** Broken. All elliptic-curve signature schemes are insecure.

4.1.2 Grover's Algorithm

Lov Grover's 1996 algorithm speeds up **unstructured search**: given a black-box function, it finds the input that produces a specific output in $O(\sqrt{N})$ evaluations instead of $O(N)$.

Applied to symmetric cryptography, Grover's algorithm effectively halves the security level:

- AES-128 drops from 128-bit security to 64-bit security (unsafe).
- AES-256 drops from 256-bit security to 128-bit security (still safe).

This is why the system uses **AES-256** rather than AES-128: even against quantum adversaries, it provides 128-bit security.

4.1.3 The "Harvest Now, Decrypt Later" Threat

The most immediate threat is not from a quantum computer that exists today, but from a strategy called **harvest now, decrypt later** (HNDL):

1. An adversary captures encrypted drone telemetry today.
2. The data is stored indefinitely (storage is cheap).
3. Years later, when a cryptographically relevant quantum computer (CRQC) exists, the adversary decrypts everything retroactively.

For military and critical-infrastructure drones, telemetry captured today may still be sensitive in 10–20 years. This system addresses the HNDL threat by using post-quantum key exchange **now**, ensuring that even retroactive quantum attacks cannot recover the symmetric keys.

4.2 The NIST Post-Quantum Standardisation

In 2016, the U.S. National Institute of Standards and Technology (NIST) launched a competition to standardize post-quantum cryptographic algorithms. The process involved:

1. **2016:** Call for proposals (82 submissions).
2. **2017–2019:** Rounds 1 and 2 (narrowing to 15 finalists).
3. **2020–2022:** Round 3 (final selections).
4. **2024:** Publication of final standards.

The standards published are:

- **FIPS 203** (ML-KEM): Key Encapsulation Mechanism based on Module-Lattice problems (formerly CRYSTALS-Kyber).
- **FIPS 204** (ML-DSA): Digital Signature Algorithm based on Module-Lattice problems (formerly CRYSTALS-Dilithium).
- **FIPS 205** (SLH-DSA): Stateless Hash-based Digital Signature Algorithm (formerly SPHINCS+).

Additional algorithms under consideration or standardized elsewhere include **Falcon** (now FN-DSA, NTRU-lattice based signatures), **Classic McEliece** (code-based KEM), and **HQC** (Hamming Quasi-Cyclic, code-based KEM).

4.2.1 NIST Security Levels

NIST defines five security levels, corresponding to the effort required to break the algorithm:

Table 4.1: NIST security levels and their classical equivalents.

Level	Classical Equivalent	Attack Effort	Quantum Equivalent
1	AES-128	2^{128} operations	2^{64} Grover operations
2	SHA-256 collision	2^{128} operations	—
3	AES-192	2^{192} operations	2^{96} Grover operations
4	SHA-384 collision	2^{192} operations	—
5	AES-256	2^{256} operations	2^{128} Grover operations

This system supports three levels: **L1**, **L3**, and **L5**, with algorithms paired at consistent levels. For example, an L3 KEM (ML-KEM-768) is always paired with an L3 signature (ML-DSA-65), never with an L1 signature—this is enforced in code.

Implementation Note

ML-DSA-44 is officially classified as NIST Level 2 by liboqs (FIPS 204), but the system maps it to L1 for practical pairing with L1 KEMs like ML-KEM-512. This conservative mapping ensures consistent security across the KEM and signature components of each suite.

4.3 Hard Problems for Post-Quantum Cryptography

Post-quantum algorithms are built on mathematical problems believed to be hard even for quantum computers. The three families used in this system are based on different problems:

4.3.1 Lattice Problems

A **lattice** is a regular grid of points in multi-dimensional space, generated by integer combinations of basis vectors.

Analogy

Imagine an infinite checkerboard in two dimensions: the corners of the squares form a 2D lattice. Now generalize this to hundreds of dimensions. Finding the shortest vector in such a lattice is easy when you know the "nice" basis vectors (the grid lines), but extremely hard when you are given a "bad" basis (a set of long, nearly parallel vectors that generate the same lattice).

The key problems are:

Shortest Vector Problem (SVP): Given a lattice, find the shortest non-zero vector. This is NP-hard in the worst case and believed to be hard on average.

Learning With Errors (LWE): Given a system of approximate linear equations over a finite field (with small random errors added), recover the secret vector. Formally:

$$\mathbf{b} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e} \pmod{q} \tag{4.1}$$

where $\mathbf{A}$ is a public random matrix, $\mathbf{s}$ is the secret, and $\mathbf{e}$ is a small error vector. Without the errors, this is just solving a linear system (easy). With the errors, it becomes as hard as worst-case lattice problems.

Module-LWE (MLWE): A structured variant of LWE where the matrix $\mathbf{A}$ has algebraic structure (its entries are elements of a polynomial ring). This allows smaller keys and faster operations while preserving hardness. **ML-KEM and ML-DSA are both based on MLWE.**

Key Insight

No known quantum algorithm solves lattice problems significantly faster than classical algorithms. The best-known algorithms (both classical and quantum) require exponential time for standard lattice dimensions. This is why lattice-based cryptography is considered the strongest candidate for post-quantum security.

4.3.2 Code-Based Problems

Error-correcting codes are used in communications to detect and fix errors introduced during transmission. Code-based cryptography reverses this: the secret is a code that can correct errors, and the public key is a "scrambled" version that hides the code's structure.

Syndrome Decoding Problem: Given a random-looking code and a codeword with errors, determine the error pattern. This is NP-hard in the general case.

Analogy

Imagine a library where the books are shelved using a secret system (the error-correcting code). If you know the system, you can quickly find any book. The public key is like a deliberately rearranged library: without knowing the secret shelving system, finding a specific book requires checking every shelf.

Classic McEliece uses binary Goppa codes (proposed by McEliece in 1978—the oldest unbroken public-key cryptosystem). **HQC** uses quasi-cyclic codes with a different structure, resulting in smaller keys than McEliece but a different security assumption.

4.3.3 Hash-Based Constructions

Hash-based signatures rely solely on the security of a hash function. Their security proof is simple: if the hash function is secure, the signature scheme is secure. No additional mathematical assumptions are needed.

Key Insight

Hash-based signatures are the most conservative post-quantum option. Their security depends only on the hash function (SHA-256), which is well-studied and trusted. However, they produce much larger signatures than lattice-based schemes.

4.4 Key Encapsulation Mechanisms (KEMs)

A KEM is a three-algorithm tuple $(\text{KeyGen}, \text{Encapsulate}, \text{Decapsulate})$:

1. $(\text{pk}, \text{sk}) \leftarrow \text{KeyGen}()$: Generate a key pair.
2. $(\text{ct}, \text{ss}) \leftarrow \text{Encapsulate}(\text{pk})$: Using the public key, produce a ciphertext and a shared secret.

3. ss $\leftarrow$ Decapsulate(sk, ct): Using the secret key and the ciphertext, recover the shared secret.

Both sides now share the same secret ss, which is used as input to HKDF to derive symmetric transport keys.

Design Decision

The system uses KEMs rather than post-quantum key exchange (like SIDH/SIKE) because KEMs are simpler, better studied, and all NIST-selected algorithms for key establishment are KEMs. A KEM naturally fits the client-server handshake model: the server generates a key pair, the client encapsulates, and the server decapsulates.

4.4.1 ML-KEM (FIPS 203, formerly Kyber)

ML-KEM (Module-Lattice Key Encapsulation Mechanism) is the primary KEM standard. It is based on the **Module-LWE** problem.

How it works (simplified):

1. **KeyGen:** Generate a random matrix $\mathbf{A}$ and secret vector $\mathbf{s}$. The public key is $(\mathbf{A}, \mathbf{b} = \mathbf{As} + \mathbf{e})$, where $\mathbf{e}$ is small random noise.

2. **Encapsulate:** Choose a random message m, encrypt it under the public key using an MLWE-based encryption scheme, and derive the shared secret from m.

3. **Decapsulate:** Use the secret key to decrypt the message, re-encrypt it, and verify consistency (Fujisaki-Okamoto transform ensures IND-CCA2 security).

Table 4.2: ML-KEM parameter sets used in this system.

Variant	NIST Level	Public Key	Ciphertext	Shared Secret	Module Rank
ML-KEM-512	L1	800 bytes	768 bytes	32 bytes	$k = 2$
ML-KEM-768	L3	1,184 bytes	1,088 bytes	32 bytes	$k = 3$
ML-KEM-1024	L5	1,568 bytes	1,568 bytes	32 bytes	$k = 4$

ML-KEM is the **default** KEM in this system (the default suite is `cs-mlkem768-aesgcm-mldsa65`, using ML-KEM-768 at NIST Level 3).

Key Insight

ML-KEM's key sizes (800–1568 bytes) are dramatically larger than classical ECDH (32 bytes) but small enough for practical use over WiFi. This is one of the key trade-offs in post-quantum cryptography: increased sizes are the price of quantum resistance.

4.4.2 Classic McEliece

Classic McEliece is a code-based KEM with the longest track record in cryptography—the underlying McEliece cryptosystem was proposed in **1978** and has never been broken.

How it works (simplified):

1. **KeyGen:** Choose a random binary Goppa code. The secret key is the code description; the public key is a scrambled version of the code's generator matrix.
2. **Encapsulate:** Choose a random error pattern of the correct weight and "encrypt" it using the public matrix.
3. **Decapsulate:** Use the secret Goppa code to decode (correct the errors) and recover the shared secret.

Table 4.3: Classic McEliece parameter sets used in this system.

Variant	NIST Level	Public Key	Ciphertext	Shared Secret
Classic-McEliece-348864	L1	261,120 bytes	128 bytes	32 bytes
Classic-McEliece-460896	L3	524,160 bytes	188 bytes	32 bytes
Classic-McEliece-8192128	L5	1,357,824 bytes	240 bytes	32 bytes

Security Note

Classic McEliece has **enormous** public keys—up to 1.3 MB for the L5 variant. This makes it impractical for many applications but provides the highest confidence in security (40+ years of cryptanalysis). The ciphertexts are tiny (128–240 bytes). In this system, the public key is transmitted only once during the handshake, so the overhead is a one-time cost.

4.4.3 HQC (Hamming Quasi-Cyclic)

HQC is a code-based KEM that uses **quasi-cyclic** codes to achieve much smaller keys than Classic McEliece while still relying on code-based hardness assumptions.

Table 4.4: HQC parameter sets used in this system.

Variant	NIST Level	Public Key	Ciphertext	Shared Secret
HQC-128	L1	2,249 bytes	4,481 bytes	64 bytes
HQC-192	L3	4,522 bytes	9,026 bytes	64 bytes
HQC-256	L5	7,245 bytes	14,469 bytes	64 bytes

HQC offers a middle ground: public keys are much smaller than Classic McEliece (kilobytes vs. hundreds of kilobytes) but ciphertexts are larger. It provides **cryptographic diversity**—if lattice assumptions (used by ML-KEM) turn out to be weaker than expected, HQC remains secure because it relies on a different mathematical problem.

Design Decision

The system includes three KEM families (lattice-based ML-KEM, code-based Classic McEliece, and code-based HQC) for **algorithm agility**. If a breakthrough attack is discovered against one family, the operator can switch to a different family by changing the suite configuration—no code changes required.

4.5 Digital Signature Algorithms

Post-quantum signatures replace ECDSA/EdDSA for handshake authentication. The system supports three families:

4.5.1 ML-DSA (FIPS 204, formerly Dilithium)

ML-DSA (Module-Lattice Digital Signature Algorithm) is the primary NIST standard for post-quantum signatures, based on the **Module-LWE** and **Module-SIS** (Short Integer Solution) problems.

How it works (simplified):

1. **KeyGen:** Generate a random matrix $\mathbf{A}$ and secret vectors $(\mathbf{s}_1, \mathbf{s}_2)$. Public key: $(\mathbf{A}, \mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2)$.
2. **Sign:** Hash the message, choose a random masking vector $\mathbf{y}$, compute a commitment, hash to get a challenge, and produce a response. Repeat (rejection sampling) until the response doesn't leak the secret key.
3. **Verify:** Recompute the commitment from the public key and the signature, check that it matches.

Table 4.5: ML-DSA parameter sets used in this system.

Variant	NIST Level	Public Key	Secret Key	Signature
ML-DSA-44	L1 (mapped)	1,312 bytes	2,560 bytes	2,420 bytes
ML-DSA-65	L3	1,952 bytes	4,032 bytes	3,309 bytes
ML-DSA-87	L5	2,592 bytes	4,896 bytes	4,627 bytes

4.5.2 Falcon (FN-DSA)

Falcon uses **NTRU lattices**—a different lattice structure from ML-DSA—and produces the **smallest signatures** of any post-quantum scheme at comparable security levels.

How it works (simplified):

1. **KeyGen:** Generate an NTRU polynomial pair (f, g) and compute the public key $h = g/f$.
2. **Sign:** Hash the message to a polynomial c, then use a "trapdoor sampler" (fast Fourier sampling) to find a short vector $(\mathbf{s}_1, \mathbf{s}_2)$ such that $\mathbf{s}_1 + h \cdot \mathbf{s}_2 = c$.
3. **Verify:** Check that the signature vector is short and satisfies the equation.

Table 4.6: Falcon parameter sets used in this system.

Variant	NIST Level	Public Key	Signature
Falcon-512	L1	897 bytes	666 bytes
Falcon-1024	L5	1,793 bytes	1,280 bytes

Key Insight

Falcon has no L3 variant. The system enforces NIST-level consistency: Falcon-512 (L1) pairs only with L1 KEMs, and Falcon-1024 (L5) pairs only with L5 KEMs. There is no Falcon option for L3 suites.

4.5.3 SPHINCS+ (SLH-DSA, FIPS 205)

SPHINCS+ is a **stateless hash-based** signature scheme. Its security depends **only on the security of SHA-256**—no lattice or code assumptions are needed.

How it works (simplified):

1. **KeyGen:** Generate a random seed. The secret key is the seed; the public key is the root of a Merkle hash tree built from many one-time signature (OTS) key pairs.
2. **Sign:** Select one OTS key pair (determined by the message hash), sign with it, and include the authentication path (the sequence of sibling hashes needed to reconstruct the Merkle tree root).
3. **Verify:** Verify the OTS signature, then reconstruct the path to the root and check that it matches the public key.

Table 4.7: SPHINCS+ parameter sets used in this system ("s" = small/slow variant).

Variant	NIST Level	Public Key	Signature
SPHINCS+-SHA2-128s	L1	32 bytes	7,856 bytes
SPHINCS+-SHA2-192s	L3	48 bytes	16,224 bytes
SPHINCS+-SHA2-256s	L5	64 bytes	29,792 bytes

Security Note

SPHINCS+ produces very large signatures (up to ∼30 KB) and is relatively slow. However, it is the most **conservative** choice: if lattice-based cryptography is broken, SPHINCS+ remains secure. The system includes it as an "insurance policy" for maximum diversity.

4.6 The Suite Concept

A **cryptographic suite** in this system is a triple:

$$\text{Suite} = (\text{KEM}, \text{AEAD}, \text{Signature}) \tag{4.2}$$

For example, the default suite `cs-mlkem768-aesgcm-mldsa65` is:

- **KEM:** ML-KEM-768 (lattice-based key encapsulation, NIST L3)
- **AEAD:** AES-256-GCM (authenticated encryption)
- **Signature:** ML-DSA-65 (lattice-based digital signature, NIST L3)

4.6.1 Level-Consistent Pairing

The system enforces that the KEM and signature in every suite share the same NIST security level:

$$\text{nist\_level}(\text{KEM}) = \text{nist\_level}(\text{Signature}) \tag{4.3}$$

This prevents misconfigurations where, say, an L5 KEM is paired with an L1 signature, creating a false sense of security.

4.6.2 Suite Enumeration

The total number of suites is determined by the Cartesian product:

$$|\text{Suites}| = \sum_{\ell \in \{L1, L3, L5\}} |\text{KEM}_\ell| \times |\text{AEAD}| \times |\text{SIG}_\ell| \tag{4.4}$$

With the current registry:

- **L1:** 3 KEMs × 3 AEADs × 3 SIGs = 27 suites
- **L3:** 3 KEMs × 3 AEADs × 2 SIGs = 18 suites (no Falcon at L3)
- **L5:** 3 KEMs × 3 AEADs × 3 SIGs = 27 suites

Total: **72 suites**, all generated automatically by the registry code.

Implementation Note

The suite registry is defined in `core/suites.py`. Rather than listing 72 entries manually, the code generates all level-consistent (KEM, SIG) pairs automatically via `_generate_level_consistent_matrix()` and then crosses each pair with the three AEAD tokens. The registry is wrapped in a `MappingProxyType` for immutability.

4.7 Comparative Overview

The benchmarking infrastructure in this system (covered in Chapters 11–12) measures the actual performance of all 72 suites on real hardware, providing concrete numbers for every cell in this comparison.

Table 4.8: Comparative overview of post-quantum algorithm families in this system.

Algorithm	Family	Hard Problem	Key Size	Ciphertext/Sig	Speed
Key Encapsulation Mechanisms					
ML-KEM	Lattice	MLWE	Small	Small	Fast
Classic McE.	Code	Syndrome dec.	Huge	Tiny	Fast
HQC	Code	Quasi-cyclic	Medium	Medium	Medium
Digital Signatures					
ML-DSA	Lattice	MLWE + MSIS	Medium	Medium	Fast
Falcon	Lattice	NTRU	Small	Small	Fast
SPHINCS+	Hash	Hash security	Tiny	Huge	Slow

4.8 Summary

- **Quantum computers** threaten public-key cryptography via Shor's algorithm but leave symmetric cryptography largely intact (Grover halves the key strength).
- The **harvest-now-decrypt-later** threat motivates deploying post-quantum cryptography today.
- NIST standardized **ML-KEM** (FIPS 203) for key encapsulation and **ML-DSA** (FIPS 204) + **SPHINCS+** (FIPS 205) for signatures.
- This system supports three KEM families (**ML-KEM**, **Classic McEliece**, **HQC**) and three signature families (**ML-DSA**, **Falcon**, **SPHINCS+**).
- Algorithms are organized into **suites** of (KEM, AEAD, Signature) triples, always paired at consistent NIST security levels.
- The system generates **72 suites** automatically from the registry, enabling comprehensive benchmarking and algorithm agility.

With the mathematical foundations established, the next chapter introduces the **MAVLink protocol**—the specific application-layer protocol that the secure tunnel protects.

Part II

The Domain: Drones and Secure Communication

Chapter 5

The MAVLink Protocol

MAVLink is the lingua franca of autonomous vehicles—simple enough for an 8-bit microcontroller, expressive enough for an autonomous fleet.

This chapter introduces MAVLink, the application-layer protocol that flows through the secure tunnel. Understanding MAVLink is essential because the tunnel must be completely transparent to it: every MAVLink packet that enters the tunnel must emerge unchanged on the other side.

5.1 What Is MAVLink?

Definition 5.1 (MAVLink)**. MAVLink** (Micro Air Vehicle Link) is a lightweight, header-only binary protocol for communicating between unmanned vehicles and ground control stations. It was created in 2009 by Lorenz Meier at ETH Zurich and has become the de facto standard for drone telemetry and command-and-control.

MAVLink is used by:

- **ArduPilot:** The most popular open-source autopilot software.
- **PX4:** Another major open-source flight controller.
- **Mission Planner, QGroundControl:** Popular ground station software.
- **MAVProxy:** A command-line MAVLink ground station and proxy.

Analogy

MAVLink is to drones what HTTP is to the web. Just as your browser uses HTTP to talk to web servers, your ground station uses MAVLink to talk to the autopilot on the drone. And just as HTTPS wraps HTTP in TLS for security, our system wraps MAVLink in a post-quantum tunnel for quantum-resistant security.

5.2 MAVLink Versions

Two versions of MAVLink are in common use:

MAVLink v1: The original protocol. Uses a 6-byte header and 2-byte CRC. Maximum payload: 255 bytes. No authentication. No signing.

MAVLink v2: The current standard. Uses a 10-byte header. Adds support for message signing (optional, rarely used), message extensions, and a 3-byte message ID space (16.7 million possible message types vs. 256 in v1).

This system is protocol-version agnostic: it encrypts the *entire* MAVLink datagram as an opaque byte payload, regardless of version.

5.3 MAVLink Packet Structure

A MAVLink v2 packet has the following structure:

Table 5.1: MAVLink v2 frame structure.

Offset	Field	Bytes	Description
0	STX (Magic)	1	`0xFD` for v2 (`0xFE` for v1)
1	Payload Length	1	Length of the payload (0–255)
2	Incompat. Flags	1	Incompatibility flags
3	Compat. Flags	1	Compatibility flags
4	Sequence	1	Per-component packet counter (0–255)
5	System ID	1	ID of the sending system (1–255)
6	Component ID	1	ID of the sending component
7–9	Message ID	3	24-bit message type identifier
10+	Payload	0–255	Message-specific data
—	CRC	2	CRC-16/MCRF4XX checksum
—	Signature	13	Optional (if signed)

5.3.1 System IDs and Component IDs

Every device on a MAVLink network is identified by a unique **(System ID, Component ID)** pair:

System ID: Identifies the vehicle or ground station (1–255). The autopilot and all on-board components share the same System ID.

Component ID: Identifies a specific component within a system. For example, the autopilot is typically component 1, a camera is component 100, and the ground station is component 191.

Key Insight

The secure tunnel is **transparent** with respect to MAVLink addressing. It does not need to understand System IDs or Component IDs—it simply encrypts and decrypts the raw bytes. This means it works with any MAVLink network topology without configuration changes.

5.3.2 Sequence Numbers

Each MAVLink component maintains an 8-bit sequence counter (0–255) that wraps around. The receiver uses this to detect lost packets.

Security Note

Do not confuse the MAVLink sequence number (8-bit, wraps at 255, **not** cryptographic) with the tunnel's AEAD sequence number (64-bit, **never** wraps, **cryptographic**). They serve different purposes: MAVLink's sequence detects transport loss; the tunnel's sequence provides replay protection and nonce uniqueness.

5.4 Key MAVLink Messages

The MAVLink protocol defines hundreds of message types. The most important for understanding this system are:

`HEARTBEAT` (ID 0): Sent at 1 Hz by every system. Contains the vehicle type, autopilot type, base mode (armed/disarmed), system status, and MAVLink version. If the GCS stops receiving heartbeats, it assumes the link is lost.

`SYS_STATUS` (ID 1): System health: battery voltage, CPU load, sensor health bitmask.

`GPS_RAW_INT` (ID 24): Raw GPS data: latitude, longitude, altitude, ground speed, heading, fix type, satellite count.

`ATTITUDE` (ID 30): Vehicle orientation: roll, pitch, yaw angles and their rates.

`GLOBAL_POSITION_INT` (ID 33): Fused position estimate (GPS + IMU): latitude, longitude, altitude above sea level and ground, velocity components.

`COMMAND_LONG` (ID 76): Sends a command to a component (e.g., arm motors, change mode, start mission).

`COMMAND_ACK` (ID 77): Acknowledges a command with a result code (accepted, denied, failed, etc.).

`STATUSTEXT` (ID 253): Human-readable status messages (e.g., "PreArm: GPS Fix required").

Implementation Note

The MAVLink collector (`core/mavlink_collector.py`) sniffs these messages on the plaintext ports to compute metrics: heartbeat interval, message rates, sequence gaps, and command-ACK latency. It does this by connecting to the local plaintext port and passively observing traffic—it never modifies or injects packets.

5.5 MAVLink's Security Problem

MAVLink was designed in 2009 for simplicity and efficiency on 8-bit microcontrollers. Security was not a design goal:

1. **No encryption:** All traffic is in plaintext. Anyone who can receive the WiFi signal can read GPS coordinates, battery levels, and flight commands.

2. **No authentication:** There is no mechanism to verify that a command came from the legitimate ground station. An attacker can forge `COMMAND_LONG` messages to disarm motors, change flight mode, or trigger return-to-home.

3. **No replay protection:** MAVLink's 8-bit sequence number is not cryptographic. An attacker can capture a valid arm/disarm command and replay it.

4. **Optional signing is inadequate:** MAVLink v2 supports optional message signing (HMAC-SHA256), but:

   - It uses a single static key (no forward secrecy).
   - The key must be manually provisioned.
   - It provides authentication but not confidentiality.
   - It uses classical cryptography (vulnerable to quantum attacks).
   - Adoption is minimal—most systems disable it.

Security Note

These are not theoretical vulnerabilities. Researchers have demonstrated MAVLink injection attacks using $20 hardware (a WiFi adapter in monitor mode) at distances exceeding 1 km. Securing MAVLink is the core motivation for this entire system.

5.6 MAVProxy

MAVProxy is a command-line MAVLink ground station and packet router, used extensively in this system.

5.6.1 What MAVProxy Does

1. **Serial-to-UDP bridge:** On the drone, MAVProxy reads MAVLink frames from the serial port (connected to the Pixhawk flight controller) and forwards them as UDP datagrams.

2. **Multi-output routing:** MAVProxy can forward the same MAVLink stream to multiple destinations simultaneously (e.g., to the tunnel proxy *and* to a local log file).

3. **Parameter management:** Provides a command interface for reading and writing autopilot parameters.

4. **Script automation:** Supports Python scripting for automated test sequences.

5.6.2 MAVProxy in the Secure Tunnel

On the **drone side**, MAVProxy serves as the bridge between the Pixhawk's serial port and the tunnel:

1. The Pixhawk outputs MAVLink over USB serial (e.g., `/dev/ttyAMA0`).

2. MAVProxy reads from the serial port.

3. MAVProxy forwards UDP datagrams to `127.0.0.1:47003` (the drone proxy's plaintext input).

4. MAVProxy also listens on `127.0.0.1:47004` for return traffic from the tunnel.

On the **GCS side**, MAVProxy (or Mission Planner) sends/receives traffic via the GCS proxy's plaintext ports (`47001`/`47002`).

Design Decision

MAVProxy is used on the drone rather than a custom serial reader because it provides robust MAVLink framing, parameter management, and multi-output routing. The tunnel proxy operates at the UDP datagram level, completely below the MAVLink framing layer—it never parses MAVLink packets.

5.7 MAVLink Metrics

The system collects detailed MAVLink metrics to verify that the tunnel does not degrade communication quality:

Key Insight

By comparing these metrics across different suites, the benchmarking system can determine the real-world impact of each post-quantum algorithm on MAVLink communication quality. For example, does using Classic McEliece (with its 1.3 MB public key) cause heartbeat loss during the handshake?

Table 5.2: MAVLink metrics collected by the system.

Metric	Description
`heartbeat_count`	Number of `HEARTBEAT` messages received
`heartbeat_loss_count`	Number of expected heartbeats that were not received
`heartbeat_avg_ms`	Average interval between heartbeats (should be ∼1000 ms)
`msg_rate_rx`	Messages received per second
`msg_rate_tx`	Messages transmitted per second
`seq_gaps`	Number of MAVLink sequence number gaps
`seq_duplicates`	Number of duplicate sequence numbers
`crc_errors`	Number of MAVLink CRC failures
`cmd_ack_latency_ms`	Average time from command send to ACK receive

5.8 Summary

- **MAVLink** is a lightweight binary protocol for drone telemetry and command-and-control, used by ArduPilot, PX4, and all major ground stations.
- MAVLink v2 packets have a 10-byte header, variable payload (up to 255 bytes), and CRC.
- Devices are identified by (System ID, Component ID) pairs; an 8-bit sequence counter detects transport loss.
- MAVLink has **no native encryption, authentication, or replay protection**—this is the core problem the secure tunnel solves.
- **MAVProxy** bridges the Pixhawk's serial port to UDP datagrams, feeding the tunnel's plaintext input port.
- The tunnel is **fully transparent** to MAVLink: it operates at the UDP datagram level and never parses MAVLink content.
- MAVLink metrics (heartbeat rate, sequence gaps, command latency) are collected to verify tunnel transparency.

The next chapter presents the complete **system architecture**, showing how the tunnel, MAVProxy, the handshake, and the data plane fit together.

Chapter 6

System Architecture

> Architecture is the allocation of responsibilities to components and the definition of the interfaces between them.

This chapter presents the complete system architecture: what runs where, how the components communicate, and why the design is structured as it is.

6.1 The "Bump in the Wire" Concept

The secure tunnel operates as a **transparent proxy**—a "bump in the wire" between the application (Mission Planner or MAVProxy) and the physical network. Neither the application nor the flight controller knows the tunnel exists:

- The application sends plaintext UDP datagrams to a *local* port.
- The proxy encrypts them and sends them over the network.
- On the far end, the peer proxy decrypts them and delivers plaintext to the application.

No code changes are required in MAVLink applications, flight controllers, or ground station software. The only configuration change is redirecting the application's connection to the proxy's local port instead of the remote peer's network port.

6.2 Hardware Topology

The system consists of two physical nodes connected by a WiFi network:

6.2.1 Drone Side (Raspberry Pi 5)

- **Raspberry Pi 5** (8 GB): Runs the drone proxy, MAVProxy, and benchmark orchestrator.

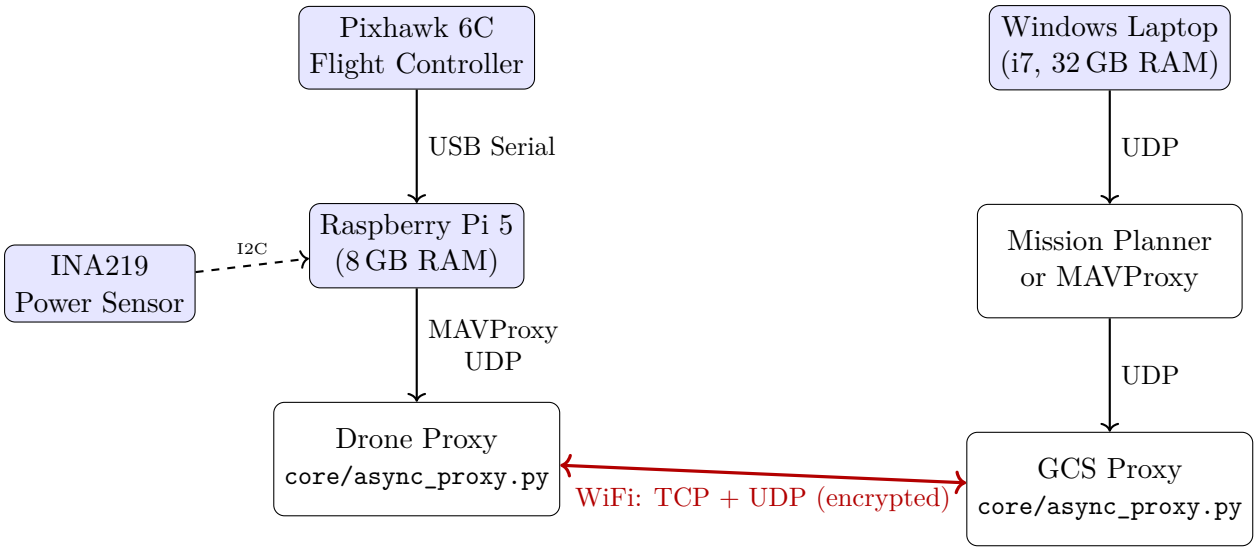


Figure 6.1: Physical hardware topology of the secure tunnel system.

- **Pixhawk 6C**: The flight controller, connected via USB serial. Outputs MAVLink telemetry and accepts commands.
- **INA219 Power Sensor** (optional): Connected via I2C, measures current and voltage for power benchmarking.
- **WiFi**: The Pi's built-in WiFi (`wlan0`) or a USB WiFi adapter.

6.2.2 GCS Side (Windows Laptop)

- **Windows PC**: Runs the GCS proxy, Mission Planner (or MAVProxy), and optionally the analysis dashboard.
- **Python + conda**: The `oqs-dev` conda environment provides `oqs-python` with PQC support.

6.3 Network Port Map

Every network socket in the system has a precisely defined role:

Design Decision

Plaintext ports bind to `127.0.0.1` (localhost only), ensuring that unencrypted MAVLink traffic never leaves the local machine. Encrypted ports bind to `0.0.0.0` (all interfaces), allowing the peer to reach them over WiFi. This is a deliberate security measure.

6.4 The Three Communication Planes

The system uses three distinct communication planes:

Table 6.1: Complete network port map.

Side	Bind Address	Protocol	Port	Purpose
GCS	`0.0.0.0`	TCP	46000	Handshake server (listens)
GCS	`0.0.0.0`	UDP	46011	Encrypted RX (from drone)
GCS	`127.0.0.1`	UDP	47001	Plaintext TX (app → proxy)
GCS	`127.0.0.1`	UDP	47002	Plaintext RX (proxy → app)
Drone	connects to GCS	TCP	→46000	Handshake client
Drone	`0.0.0.0`	UDP	46012	Encrypted RX (from GCS)
Drone	`127.0.0.1`	UDP	47003	Plaintext TX (MAVProxy → proxy)
Drone	`127.0.0.1`	UDP	47004	Plaintext RX (proxy → MAVProxy)
Drone	`0.0.0.0`	TCP	48080	Control channel (scheduler)

6.4.1 Control Plane (TCP)

Handshake channel (TCP 46000): Used exclusively for the PQC key exchange. The drone connects as TCP client; the GCS listens as TCP server. Traffic is minimal (a few kilobytes per handshake) but latency-sensitive: the data plane cannot operate until the handshake completes.

Scheduler control channel (TCP 48080): Used by the benchmark orchestrator. The drone controller sends JSON commands ("start proxy," "switch suite," "stop") to the GCS follower. This channel carries no MAVLink traffic.

6.4.2 Data Plane (UDP, encrypted)

The high-throughput path. Encrypted MAVLink datagrams flow bidirectionally between the drone (port 46012) and GCS (port 46011). Every packet carries the 22-byte AEAD header and 16-byte authentication tag.

6.4.3 Plaintext Plane (UDP, localhost)

The loopback path between the proxy and local applications. On the drone: MAVProxy ↔ proxy on ports 47003/47004. On the GCS: Mission Planner ↔ proxy on ports 47001/47002. This traffic **never** leaves the local machine.

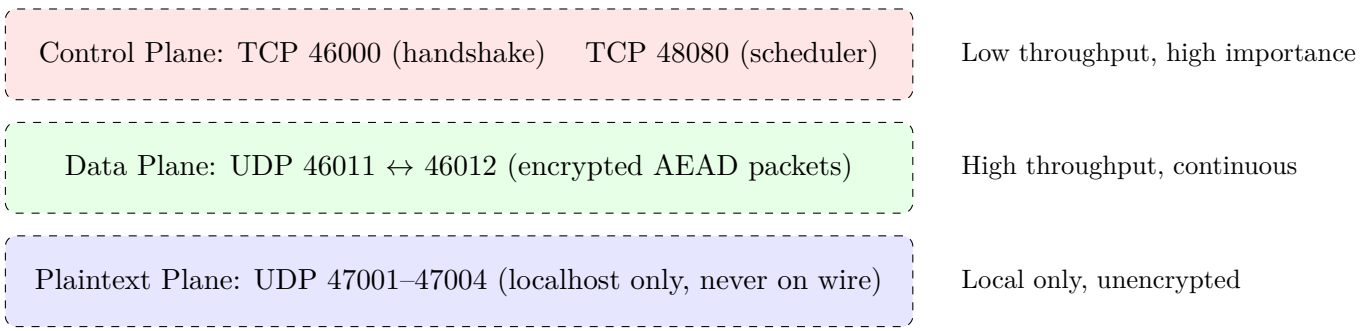


Figure 6.2: The three communication planes of the secure tunnel.

6.5 Software Components

The codebase is organized into three major directories, each with a clear responsibility:

6.5.1 `core/` — The Tunnel Engine

This is the security-critical core: the proxy, handshake, AEAD framing, and suite registry. It runs on **both** drone and GCS, using the same code with different configuration.

Table 6.2: Key modules in `core/`.

Module	Responsibility
`config.py`	Single source of truth: all ports, IPs, feature flags, timeouts
`async_proxy.py`	Main event loop (selectors-based). Manages 4 UDP sockets, handshake, encrypt/decrypt, rate limiting, rekey
`handshake.py`	PQC handshake: KEM keygen/encap/decap, SIG sign/verify, HKDF key derivation, HMAC drone auth
`aead.py`	AEAD framing: Sender (encrypt + header), Receiver (validate + decrypt + anti-replay)
`suites.py`	Suite registry: KEM×AEAD×SIG Cartesian product, alias resolution, runtime probing
`run_proxy.py`	CLI entry point: `init-identity`, `gcs`, `drone` subcommands
`policy_engine.py`	Two-phase commit rekey state machine (IDLE→PREPARE_SENT→COMMITTED/ABORTED)
`metrics_aggregator.py`	Central metrics orchestrator: wires all collectors, thread-safe sampling
`metrics_schema.py`	18-category metrics schema definition (categories A through R)
`power_monitor.py`	INA219 I2C power measurement with sign auto-detection and CSV logging
`mavlink_collector.py`	Passive MAVLink sniffer: message rates, heartbeat tracking, sequence gaps
`process.py`	Managed subprocess with Windows Job Objects / Linux PDEATHSIG cleanup

6.5.2 `sscheduler/` — The Benchmark Orchestrator

The "simplified scheduler" manages automated benchmark runs. The **drone** is the controller; the **GCS** is the follower.

6.5.3 `dashboard/` — The Analysis Dashboard

A web-based visualization tool for benchmark results.

Backend: FastAPI (Python), Pydantic v2 models, JSONL/JSON ingestion from benchmark outputs.

Table 6.3: Key modules in `sscheduler/`.

Module	Responsibility
`sdrone_bench.py`	Drone benchmark controller: manages GCS via TCP, cycles suites, collects metrics, writes JSONL
`sgcs_bench.py`	GCS benchmark follower: starts/stops proxy and MAVProxy per instructions from drone
`sdrone_mav.py`	Drone MAVProxy management: serial port detection, process lifecycle
`sgcs_mav.py`	GCS MAVProxy management
`benchmark_policy.py`	BenchmarkPolicy: `evaluate()` (proposal, no side effects) + `confirm_advance()` (state mutation)
`policy.py`	Runtime policies: TelemetryAwarePolicy (safety-critical), SequentialPolicy, RandomPolicy

Frontend: React 18 + TypeScript + Vite, Zustand state management, Recharts for visualization, TailwindCSS for styling.

The dashboard is covered in detail in Chapter 13.

6.6 End-to-End Data Flow

This section traces a single MAVLink packet from the Pixhawk to Mission Planner:

1. **Pixhawk → Pi (serial):** The flight controller sends a MAVLink telemetry packet over USB serial.
2. **MAVProxy (Pi):** Reads the serial frame, produces a UDP datagram, sends it to `127.0.0.1:47003`.
3. **Drone Proxy ingress:** The proxy's selectors loop detects a readable event on the plaintext socket. It reads the datagram (e.g., 50 bytes of MAVLink).
4. **Encryption:** The proxy calls `Sender.encrypt()`, which:
   - Constructs a 22-byte header (version, algorithm IDs, session ID, sequence number, epoch).
   - Derives the nonce from epoch + sequence.
   - Calls the AEAD algorithm (e.g., AES-256-GCM) to encrypt the 50 bytes and produce a 16-byte tag, using the header as AAD.
   - Returns the 88-byte wire packet (22 header + 50 ciphertext + 16 tag).
5. **Transmission:** The proxy sends the 88-byte UDP datagram from the drone's network interface to the GCS at `<GCS_IP>:46011`.
6. **GCS Proxy ingress:** The GCS proxy's selectors loop detects a readable event on the encrypted socket. It reads the 88-byte datagram.
7. **Decryption:** The proxy calls `Receiver.decrypt()`, which:

- Parses and validates the 22-byte header (version, algorithm IDs, session ID match).
- Checks the replay window (is this sequence number new?).
- Reconstructs the nonce from epoch + sequence.
- Calls the AEAD algorithm to decrypt and verify the tag.
- If authentication fails: drops the packet silently.
- If authentication succeeds: returns the 50-byte plaintext.

8. **GCS Proxy egress:** Sends the 50-byte plaintext datagram to `127.0.0.1:47002`.

9. **Mission Planner:** Receives the datagram on port 47002, parses it as MAVLink, and displays the telemetry on the HUD.

Return traffic (commands from Mission Planner to the Pixhawk) follows the same path in reverse, using the `key_gcs_to_drone` symmetric key instead of `key_drone_to_gcs`.

Key Insight

The entire encryption/decryption path adds approximately 30–200 microseconds of latency per packet, depending on the AEAD algorithm. For a typical MAVLink stream of 50 messages/second, this overhead is negligible compared to WiFi latency (∼1–5 ms).

6.7 Trust Model

The system's trust model defines what each party needs to know before communication begins:

GCS has: A long-term signature key pair (private + public). Generated once via `python -m core.run_proxy init-identity`.

Drone has: The GCS's **public** signature key (pre-installed), plus a **pre-shared key (PSK)** for drone authentication.

Pre-shared key (PSK): A symmetric secret known to both sides, used during the handshake for the drone to prove its identity via HMAC. This is set via the `CONFIG["DRONE_PSK"]` configuration key.

Security Note

The trust model is asymmetric: the GCS authenticates via digital signature (the drone verifies the GCS's public key), and the drone authenticates via HMAC-SHA256 with the PSK (the GCS verifies the drone knows the shared secret). This avoids the need for the drone to have its own signature key pair, simplifying deployment.

6.8 Rekey and Suite Rotation

The system can change cryptographic suites without restarting the tunnel:

1. The **policy engine** (or benchmark scheduler) decides to switch suites.
2. The policy engine initiates a **two-phase commit**:

   Phase 1 (Prepare): Both sides agree on the new suite and prepare for the transition.

   Phase 2 (Commit): Both sides simultaneously switch to new AEAD keys derived from a new handshake.
3. During the brief transition ("blackout period"), packets may be dropped. The system measures this blackout duration as a metric.
4. If the new handshake fails, both sides **abort** and continue using the old keys.

Implementation Note

The rekey state machine is implemented in `core/policy_engine.py` with states: `IDLE` → `PREPARE_SENT` → `COMMITTED` or `ABORTED`. The two-phase protocol ensures that both sides either switch together or not at all, preventing desynchronization where one side encrypts with new keys while the other still expects old keys.

6.9 The Selectors Event Loop

The proxy engine uses Python's `selectors` module rather than `asyncio`:

```
sel = selectors.DefaultSelector()
sel.register(sock_ptx_in,  selectors.EVENT_READ, on_ptx_read)
sel.register(sock_enc_in,  selectors.EVENT_READ, on_enc_read)

while running:
    events = sel.select(timeout=0.1)
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```

Listing 6.1: Simplified selectors event loop structure

Design Decision

`selectors` was chosen over `asyncio` for three reasons:

1. **Determinism:** The event loop is fully predictable—no coroutine scheduling surprises.

2. **Dependency-light:** No third-party async frameworks (uvloop, trio) needed.

3. **Debuggability:** Stack traces show the exact call path; `asyncio` frames are notoriously hard to debug.

The trade-off is that all operations must be non-blocking and manually managed, but since the proxy only handles UDP datagrams (no streaming), this is straightforward.

6.10 Security Boundaries

The system maintains strict security boundaries:

1. **Plaintext isolation:** Plaintext sockets bind to `127.0.0.1` only. Unencrypted MAVLink traffic never traverses the physical network.

2. **Rate limiting:** The handshake TCP listener enforces IP-based rate limiting to prevent handshake flood attacks.

3. **Strict peer matching:** When `CONFIG["STRICT_UDP_PEER_MATCH"]` is enabled, the proxy only accepts encrypted UDP packets from the expected peer IP address. Packets from other IPs are silently dropped.

4. **DSCP marking:** Encrypted packets can be marked with a configurable DSCP value (`CONFIG["UDP_DSCP_VALUE"]`) for Quality of Service prioritization on the network.

5. **Session binding:** The 8-byte session ID in every packet header binds each packet to a specific handshake session. A packet encrypted under one session cannot be replayed against another session, even if the same suite and keys were (hypothetically) used.

6.11 Configuration: Single Source of Truth

All configuration resides in `core/config.py`, which defines the `CONFIG` dictionary with ∼100 keys. Values can be overridden via environment variables, and the system supports mDNS-based host discovery.

Key design principles:

- **No config files on the wire path:** The proxy reads CONFIG at startup; there are no runtime config file reads.

- **Environment-first:** All sensitive values (PSK, passwords) come from environment variables, never from source control.

- **Sane defaults:** Every key has a default value that works for the standard LAN setup. Operators only need to change values that differ from the default.

Appendix A provides a complete reference of all configuration keys.

6.12 Summary

- The system operates as a **transparent "bump in the wire"** proxy. Applications are unaware of the tunnel.
- Two physical nodes (Raspberry Pi + Pixhawk on the drone, Windows laptop for GCS) communicate over WiFi.
- Three communication planes: **control** (TCP), **data** (encrypted UDP), and **plaintext** (localhost UDP).
- The codebase is split into `core/` (tunnel engine), `sscheduler/` (benchmark orchestrator), and `dashboard/` (web UI).
- Trust is established via the GCS's signature key pair (pre-installed on drone) and a pre-shared key (for drone auth).
- Rekey uses a **two-phase commit** protocol to prevent key desynchronization.
- The selectors-based event loop provides determinism and debuggability.
- Strict security boundaries: plaintext isolation, rate limiting, peer matching, session binding.

Part III now dives into the implementation of each component, starting with the PQC handshake.

Part III

The Implementation

Chapter 7

The PQC Handshake

The handshake is the most security-critical moment: it is the one time the two parties must establish trust over an untrusted channel.

This chapter walks through every step of the handshake protocol—the TCP-based key exchange that establishes the symmetric keys used for the data plane.

7.1 Overview

The handshake serves three purposes:

1. **Key establishment:** Derive two symmetric AEAD keys (one per direction) using a post-quantum KEM.
2. **GCS authentication:** Prove that the GCS is genuine (via digital signature).
3. **Drone authentication:** Prove that the drone knows the pre-shared key (via HMAC).

The handshake is a **two-message** protocol over TCP:

7.2 Step 1: GCS Builds the ServerHello

The GCS (server) performs the following operations:

1. **Generate KEM key pair:**

```
kem_obj = KeyEncapsulation("ML-KEM-768")
kem_pub = kem_obj.generate_keypair()
```

Listing 7.1: KEM key generation

This produces a fresh ephemeral KEM key pair. The public key (`kem_pub`) will be sent to the drone; the secret key is retained in `kem_obj`.

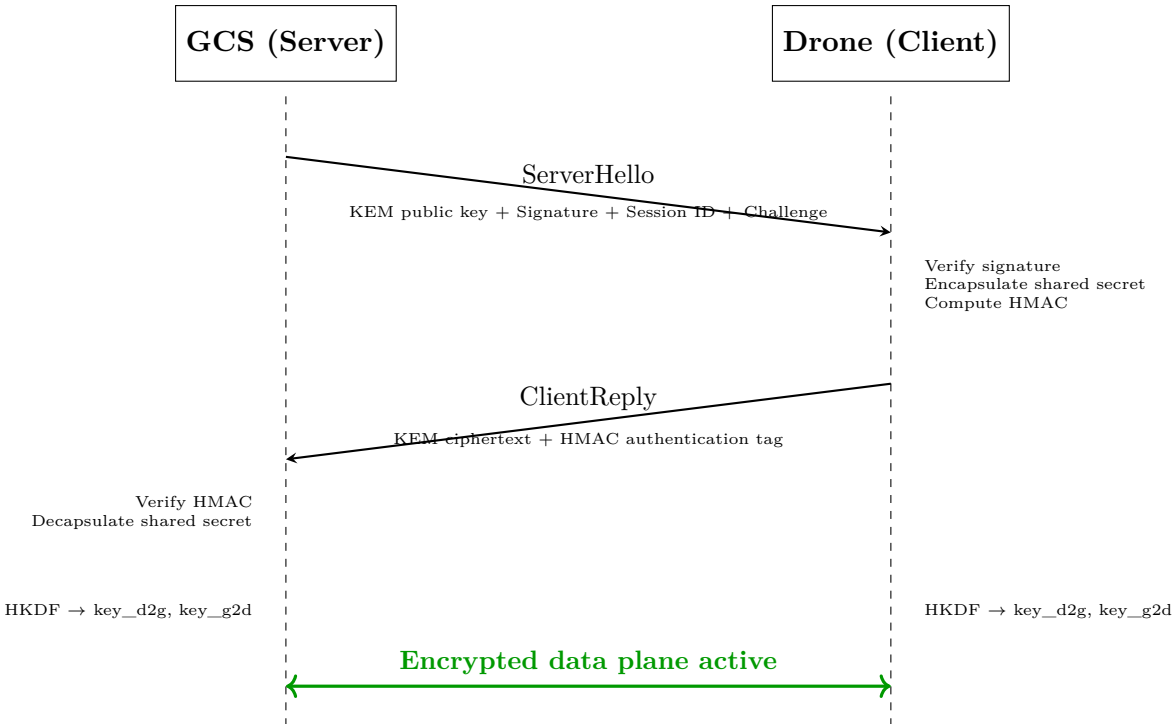


Figure 7.1: The two-message PQC handshake protocol.

2. **Generate session ID:** 8 random bytes ($= 2^{64}$ possible values), uniquely identifying this session.

3. **Generate challenge nonce:** 8 random bytes for freshness.

4. **Construct transcript:** Concatenate all handshake data into a single byte string:

$$T = \text{version} \,\|\, \texttt{"pq-drone-gcs:v1"} \,\|\, \text{session\_id} \,\|\, \text{kem\_name} \,\|\, \text{sig\_name} \,\|\, \text{kem\_pub} \,\|\, \text{challenge} \tag{7.1}$$

5. **Sign the transcript:** Using the GCS's long-term signature private key:

```
signature = server_sig_obj.sign(transcript)
```

Listing 7.2: Transcript signing

6. **Serialize and send:** Pack all fields with length prefixes and send over TCP:

$$\text{ServerHello} = \text{version}(1) \,\|\, \text{kem\_name}(2{+}n) \,\|\, \text{sig\_name}(2{+}n) \,\|\, \text{session\_id}(8) \,\|\, \text{challenge}(8) \,\|\, \text{kem\_} \tag{7.2}$$

Security Note

The wire version byte is included as the **first byte of the transcript** before signing. This prevents a downgrade attack where an attacker strips newer security features by claiming an older protocol version. The signature covers the version, so tampering is detected.

7.3 Step 2: Drone Parses and Verifies the ServerHello

The drone receives the ServerHello and performs:

1. **Parse the wire format:** Extract all fields using length prefixes. If parsing fails, raise `HandshakeFormatError`.
2. **Reconstruct the transcript:** Using the same formula as the GCS.
3. **Verify the digital signature:** Using the GCS's **pre-installed public key**:

```
sig = Signature(sig_name)
if not sig.verify(transcript, signature, server_sig_pub):
    raise HandshakeVerifyError("bad signature")
```

Listing 7.3: Signature verification

   If verification fails, the handshake aborts immediately. There is **no fallback**—this is deliberate.

4. **Check suite consistency:** The drone verifies that the negotiated KEM and signature algorithms match what it expected. If they differ, it raises a `HandshakeVerifyError` indicating a potential downgrade attack.

Key Insight

The GCS's public signature key must be pre-installed on the drone **before deployment**. This is analogous to certificate pinning in TLS: the drone trusts exactly one GCS public key. If an attacker compromises the WiFi and tries a man-in-the-middle attack, they cannot produce a valid signature without the GCS's private key.

7.4 Step 3: Drone Encapsulates and Authenticates

After verifying the ServerHello, the drone:

1. **KEM Encapsulation:**

```
kem_ct, shared_secret = kem.encap_secret(server_hello.kem_pub)
```

Listing 7.4: KEM encapsulation

   This produces:

   - `kem_ct`: The KEM ciphertext (sent to GCS).
   - `shared_secret`: The raw shared secret (kept locally, 32 bytes).

2. **Drone HMAC authentication:**

```
tag = hmac.new(psk_bytes, hello_wire, hashlib.sha256).
    digest()
```

Listing 7.5: Drone authentication via HMAC

The drone computes an HMAC-SHA256 over the **entire raw ServerHello wire bytes** using the pre-shared key. This proves the drone's identity to the GCS.

3. **Send ClientReply:**

$$\text{ClientReply} = \text{len(kem\_ct)}(4) \,\|\, \text{kem\_ct} \,\|\, \text{HMAC tag}(32) \tag{7.3}$$

Design Decision

The drone authenticates via HMAC (symmetric) rather than a digital signature (asymmetric) to avoid deploying a second signature key pair on the drone. This simplifies key management: only the GCS needs a signature key pair; the drone only needs the GCS's public key and the shared PSK.

7.5 Step 4: GCS Verifies and Decapsulates

The GCS receives the ClientReply and:

1. **Verify HMAC:** Recompute the expected HMAC using the same PSK and the ServerHello wire bytes, then compare using constant-time comparison (`hmac.compare_digest`):

```
expected = hmac.new(psk_bytes, hello_wire, hashlib.
    sha256).digest()
if not hmac.compare_digest(tag, expected):
    raise HandshakeVerifyError("drone authentication
        failed")
```

Listing 7.6: HMAC verification on GCS

If verification fails, the GCS logs the attempt (including the peer IP) and aborts.

2. **KEM Decapsulation:**

```
shared_secret = kem_obj.decap_secret(kem_ct)
```

Listing 7.7: KEM decapsulation

Using the ephemeral secret key retained from Step 1, the GCS recovers the same shared secret that the drone derived.

Security Note

The HMAC comparison uses `hmac.compare_digest()`, which runs in constant time regardless of where the first mismatch occurs. This prevents timing side-channel attacks, where an attacker could learn the correct HMAC value one byte at a time by measuring response latency.

7.6 Step 5: Key Derivation

Both sides now have the same `shared_secret`. They independently derive the transport keys using HKDF-SHA256:

```
info = b"pq-drone-gcs:kdf:v1|" + session_id
       + b"|" + kem_name + b"|" + sig_name
hkdf = HKDF(algorithm=SHA256(), length=64,
            salt=b"pq-drone-gcs|hkdf|v1", info=info)
okm = hkdf.derive(shared_secret)  # 64 output bytes
key_d2g = okm[:32]    # drone-to-GCS encryption key
key_g2d = okm[32:64]  # GCS-to-drone encryption key
```

Listing 7.8: HKDF key derivation (both sides execute identically)

`key_d2g`: The 256-bit key the drone uses to encrypt and the GCS uses to decrypt.

`key_g2d`: The 256-bit key the GCS uses to encrypt and the drone uses to decrypt.

Key Insight

Two separate keys are derived—one for each direction. This is a standard security practice: if key_d2g were somehow compromised, traffic from GCS to drone (encrypted with key_g2d) would still be secure. The HKDF `info` parameter includes the session ID, KEM name, and signature name for domain separation.

7.7 Handshake Metrics

Every handshake operation is timed with nanosecond precision using `time.perf_counter_ns()`:

These metrics are critical for benchmarking: they reveal the true cost of each post-quantum algorithm in a real network round-trip.

Implementation Note

Both `time.perf_counter_ns()` (monotonic, high-resolution, no system clock drift) and `time.time_ns()` (wall-clock, for correlating with external events) are recorded. The performance counter is used for benchmarking; the wall clock is used for timeline analysis.

Table 7.1: Handshake timing metrics collected.

Metric	Description
`kem_keygen_ns`	Time to generate the KEM key pair
`kem_encap_ns`	Time for the drone to encapsulate the shared secret
`kem_decap_ns`	Time for the GCS to decapsulate the shared secret
`sig_sign_ns`	Time to sign the transcript
`sig_verify_ns`	Time to verify the signature
`handshake_total_ns`	Wall-clock time for the entire handshake
`public_key_bytes`	Size of the KEM public key
`ciphertext_bytes`	Size of the KEM ciphertext
`signature_bytes`	Size of the digital signature
`server_hello_bytes`	Total size of the ServerHello wire message

7.8 Error Handling

The handshake uses a hierarchy of specific exceptions:

`HandshakeFormatError`: Malformed wire data (bad lengths, truncated messages). Indicates a protocol error or network corruption.

`HandshakeVerifyError`: Cryptographic verification failure (bad signature, bad HMAC, suite mismatch). Indicates an attack or misconfiguration.

`HandshakeError`: General handshake failure (OQS unavailable, encapsulation failed). Indicates a runtime problem.

All three exceptions abort the handshake. There is no retry logic within the handshake itself; the caller (the proxy engine or scheduler) decides whether to retry.

7.9 Cryptographic Key Lifecycle

A session key in this system passes through six distinct phases. Understanding the complete lifecycle is essential for reasoning about the system's security properties—and its limitations.

7.9.1 Phase 1: Key Generation (Handshake)

During the handshake, the GCS calls `KEM.keygen()` to produce an ephemeral key pair (pk, sk). Both keys exist only in process memory; they are never written to disk. The public key pk is transmitted to the drone in the ServerHello message; the secret key sk is retained for decapsulation.

7.9.2 Phase 2: Key Derivation (HKDF)

After the KEM shared secret is established (via `encapsulate`/`decapsulate`), both sides derive two 32-byte transport keys using HKDF-SHA256 with domain-separated `info`

and `salt` parameters (Section 7.6). The raw KEM shared secret is *not used directly* for encryption—it is consumed solely as input keying material (IKM) for the KDF.

Security Note

After key derivation completes, the KEM shared secret, the KEM secret key, and any intermediate HKDF state are no longer needed. In the current Python implementation, they become eligible for garbage collection but are **not explicitly zeroised**. Python's memory model does not guarantee immediate overwrite of deallocated objects. A production hardening step would be to use `ctypes.memset` or a secure-memory library to overwrite sensitive buffers before releasing them.

7.9.3 Phase 3: Active Use (AEAD Encryption)

The derived keys (`key_d2g` and `key_g2d`) are installed in the `Sender` and `Receiver` dataclasses (Chapter 8). From this point, every MAVLink datagram is encrypted with an AEAD algorithm using the key and a deterministic nonce derived from the epoch and monotonic sequence number.

During active use:

- The sequence number increments strictly monotonically.
- The epoch byte remains constant (zero for the initial handshake).
- Both keys reside in Python process memory for the duration of the suite interval (typically 110 seconds in benchmark mode).

7.9.4 Phase 4: Rekey (Suite Rotation)

When the scheduling policy signals a suite change (`NEXT_SUITE`), the system performs a coordinated rekey:

1. The scheduler sends `stop_suite` to the GCS via the control channel.
2. Both sides stop their proxy subprocesses (which destroys the in-memory keys when the process exits).
3. A new handshake is performed for the next suite, generating entirely fresh KEM keypairs and deriving new transport keys.
4. The new proxy processes start with the new keys.

Within a single proxy session (without suite change), the `bump_epoch()` mechanism allows in-place epoch rotation: the epoch byte increments and the sequence counter resets to zero, providing fresh nonces under the same key material. The epoch is limited to 0–255; wrapping from 255 to 0 is **forbidden**—a new full handshake must be performed before the epoch exhausts.

Key Insight

Each suite rotation is a "hard rekey": the old proxy process terminates (destroying all key material in its address space) and a new process starts with fresh keys from a fresh handshake. This provides strong key separation between suites—there is no key material shared across suite boundaries.

7.9.5 Phase 5: Key Destruction (Normal Termination)

On normal termination, the `ManagedProcess` framework sends `SIGTERM` (Linux) or `TerminateProcess` (Windows) to the proxy subprocess. When the process exits:

- The Python garbage collector releases all objects, including the `bytes` objects holding key material.
- On Linux, the kernel reclaims the process's virtual memory pages. On Windows, the process handle is closed and memory is freed.
- The key material is **not explicitly zeroised** before process exit. The operating system's memory reclamation provides implicit cleanup, but a memory forensics tool running before page reuse could theoretically recover the keys.

7.9.6 Phase 6: Crash Recovery

If the proxy or scheduler process crashes unexpectedly:

1. **Linux**: The `PDEATHSIG` mechanism delivers `SIGTERM` to orphaned child processes, causing them to exit (and releasing their key material).
2. **Windows**: The Win32 Job Object terminates all children when the parent process exits.
3. The crashed session's keys are irrecoverably lost (there is no persistent key storage).
4. Recovery requires a **full new handshake**, which generates entirely fresh key material.
5. No session resumption is possible—the system does not cache KEM shared secrets or derived keys.

Security Note

The lack of key persistence is a **security feature** for a research system: there is no file or database that could leak accumulated session keys. The downside is that every crash requires a full PQC handshake, which for heavy KEMs like Classic McEliece can take tens of seconds.

7.10 Summary

- The handshake is a **two-message TCP protocol**: GCS sends ServerHello, drone sends ClientReply.
- The GCS authenticates via **digital signature** (post-quantum); the drone authenticates via **HMAC-SHA256** with a pre-shared key.
- The shared secret is established via **KEM encapsulation/decapsulation** (post-quantum).
- Two 256-bit transport keys are derived via **HKDF-SHA256** with full domain separation.
- Every primitive operation is timed at **nanosecond precision** for benchmarking.
- Suite mismatch detection prevents **downgrade attacks**.
- The wire version is signed to prevent **version rollback attacks**.

Chapter 8

AEAD Framing and Wire Format

> Every byte on the wire is either explicitly specified or deliberately absent. There are no "reserved for future use" fields in a 22-byte header.

This chapter describes the wire format: how plaintext MAVLink datagrams are transformed into authenticated, encrypted packets for transmission.

8.1 Wire Packet Structure

Every encrypted packet consists of exactly two parts:

$$\text{Wire Packet} = \underbrace{\text{Header}}_{\text{22 bytes, cleartext}} \;\|\; \underbrace{\text{Ciphertext} + \text{Tag}}_{\text{payload length + 16 bytes}} \tag{8.1}$$

The header is **not encrypted** but **is authenticated**: it is passed as Associated Data (AAD) to the AEAD algorithm. Tampering with any header byte invalidates the authentication tag.

8.1.1 Header Format

The header is packed using the Python `struct` format `!BBBBB8sQB` (22 bytes):

Key Insight

The header contains the algorithm identifiers (`kem_id`, `kem_param`, `sig_id`, `sig_param`) that were negotiated during the handshake. The receiver validates these before attempting decryption. This prevents an attacker from reusing a captured packet from a different session or suite.

8.1.2 What Is NOT on the Wire

The **nonce (IV)** is *not transmitted*. Both sides derive it deterministically from the epoch and sequence number that are already in the header:

Table 8.1: AEAD packet header fields (22 bytes total).

Offset	Field	Type	Bytes	Description
0	`version`	`B` (uint8)	1	Wire protocol version (currently 1)
1	`kem_id`	`B` (uint8)	1	KEM family identifier
2	`kem_param`	`B` (uint8)	1	KEM parameter set within family
3	`sig_id`	`B` (uint8)	1	Signature family identifier
4	`sig_param`	`B` (uint8)	1	Signature parameter set
5–12	`session_id`	`8s` (bytes)	8	Random session identifier
13–20	`seq`	`Q` (uint64)	8	Monotonic sequence number
21	`epoch`	`B` (uint8)	1	Epoch counter (incremented on rekey)

$$\text{nonce} = \underbrace{\text{epoch}}_{\text{1 byte}} \| \underbrace{\text{seq}}_{\text{11 bytes, big-endian}} \tag{8.2}$$

For AEAD algorithms requiring a 12-byte nonce (AES-GCM, ChaCha20-Poly1305), this produces exactly 12 bytes. For ASCON-128a (16-byte nonce), four zero bytes are appended.

Design Decision

Omitting the nonce from the wire saves 12 bytes per packet. For a 50-byte MAVLink message, this reduces the encrypted packet from 100 bytes to 88 bytes—a 12% reduction. Over a stream of 50 packets/second, this saves 600 bytes/second, which matters on bandwidth-constrained radio links.

8.2 The Sender

The `Sender` dataclass manages the encryption side:

```
sender = Sender(
    version=1,
    ids=AeadIds(kem_id=1, kem_param=2, sig_id=1, sig_param=2),
    session_id=session_id,        # 8 bytes from handshake
    epoch=0,
    key_send=key_drone_to_gcs,    # 32 bytes from HKDF
    aead_token="aesgcm"           # or "chacha20poly1305" or "ascon128a"
)
```

Listing 8.1: Sender initialization

8.2.1 Encryption Flow

1. **Sequence overflow check:** If the current sequence number exceeds the configured threshold (`CONFIG["REKEY_SEQ_THRESHOLD"]`, default 2^{63}), raise `SequenceOverflow` to force a rekey before IV exhaustion.

2. **Pack header:** Serialize the 22-byte header using the current sequence number.

3. **Build nonce:** Derive the 12-byte (or 16-byte) nonce from epoch + sequence.

4. **Encrypt:** Call the AEAD primitive:

```
ciphertext_and_tag = cipher.encrypt(nonce, plaintext,
    header_as_aad)
```

   This produces ciphertext of the same length as the plaintext, plus a 16-byte authentication tag appended to it.

5. **Increment sequence:** Only on successful encryption.

6. **Return wire packet:** `header || ciphertext_and_tag`.

Security Note

The sequence number is incremented **only on success**. If encryption fails (which should never happen with valid inputs), the sequence is not consumed, preserving the invariant that each sequence number is used exactly once.

8.2.2 Epoch Management

The `bump_epoch()` method increments the epoch and resets the sequence to zero. This is used during rekey transitions.

$$\text{After bump\_epoch():} \quad \text{epoch} = \text{epoch} + 1, \quad \text{seq} = 0 \tag{8.3}$$

A critical safety check prevents the epoch from wrapping from 255 to 0 (which would cause nonce reuse):

```
if self.epoch == 255:
    raise AeadError("epoch wrap forbidden without rekey")
```

Listing 8.2: Epoch wrap protection

8.3 The Receiver

The `Receiver` dataclass manages the decryption and validation side:

8.3.1 Decryption Flow

1. **Extract header:** Read the first 22 bytes.
2. **Validate header:**
   - Version must match.
   - Algorithm IDs (kem_id, kem_param, sig_id, sig_param) must match.
   - Session ID must match.
   - Epoch must match.

   If any check fails: return `None` (silent drop) or raise an exception (strict mode).
3. **Anti-replay check:** Verify the sequence number against the sliding window (Section 8.4).
4. **Reconstruct nonce:** Derive the same nonce that the sender used.
5. **Decrypt and authenticate:**

```
plaintext = cipher.decrypt(nonce, ciphertext_and_tag,
    header_as_aad)
```

   If the tag does not match (any tampering with header, ciphertext, or tag), the `cryptography` library raises `InvalidTag`.
6. **Return plaintext:** The original MAVLink datagram.

8.3.2 Error Handling Modes

The Receiver supports two modes:

Silent mode (`strict_mode=False`): Returns `None` on any validation failure. Used in the production proxy, where invalid packets should be silently dropped (per cryptographic best practices—revealing error details aids attackers).

Strict mode (`strict_mode=True`): Raises specific exceptions (`HeaderMismatch`, `ReplayError`, `AeadAuthError`). Used in tests and benchmarks for precise error diagnosis.

The Receiver tracks the reason for the last failure in `last_error_reason()`, returning one of: `"header"`, `"session"`, `"replay"`, `"auth"`, or `None` (success).

8.4 The Sliding Replay Window

The anti-replay mechanism uses a bitmask-based sliding window:

`_high`: The highest sequence number seen so far (-1 initially).

`_mask`: A bitmask where bit i indicates whether sequence _high $- i$ has been received.

`window`: The window size (default: 1024 packets).

8.4.1 Decision Logic

When a packet with sequence number s arrives:

1. **If $s >$ _high:** This is a new (future) packet.
   - Shift the bitmask left by $s -$ _high positions.
   - Set bit 0 (mark s as seen).
   - Update _high $= s$.
   - **Accept.**
2. **If $s >$ _high $-$ window:** This is a recent packet within the window.
   - Compute offset $=$ _high $- s$.
   - If bit at position `offset` is set: **Reject** (duplicate/replay).
   - Otherwise: set the bit. **Accept.**
3. **If $s \leq$ _high $-$ window:** This packet is too old.
   - **Reject.**

Key Insight

A window of 1024 means that up to 1024 packets can arrive out of order and still be accepted. This is critical for UDP over WiFi, where packet reordering is common. However, each sequence number is accepted at most once, so replaying a captured packet is always detected.

8.5 The ASCON Adapter

ASCON-128a requires special handling because it is not part of the standard `cryptography` library:

1. **Native C backend** (`core._ascon_native`): A compiled C extension for maximum performance. Preferred when available.
2. **Pure-Python fallback** (`pyascon`): A Python implementation. Slower but always available.

The `_AsconAdapter` class abstracts this:

- At initialization, it checks for the native backend first, then falls back to `pyascon`.
- It handles the naming difference: the native backend uses NIST names (`Ascon-AEAD128a`) while `pyascon` uses legacy names (`Ascon-128a`).
- The variant name is captured in a **closure** at initialization time, so `encrypt()` and `decrypt()` never need to pass the variant explicitly.
- ASCON uses only the first 16 bytes of the 32-byte key (ASCON's native key size is 128 bits).
- ASCON nonces are 16 bytes; the 12-byte epoch+sequence nonce is zero-padded.

Implementation Note

The closure-based variant capture was a deliberate fix to prevent the pure-Python backend from receiving the NIST name (which it doesn't understand) or the native backend from receiving the legacy name. Each closure captures the correct name for its backend at initialization.

8.6 Packet Size Analysis

For a typical MAVLink `HEARTBEAT` message (9 bytes of payload, 12 bytes of MAVLink framing = 21 bytes total):

Table 8.2: Wire packet sizes for a 21-byte MAVLink message.

Component	Bytes	Notes
AEAD header	22	Version + IDs + session + seq + epoch
Ciphertext	21	Same length as plaintext
Authentication tag	16	Appended by AEAD
Total wire	**59**	181% overhead for this small message

For a larger MAVLink `GLOBAL_POSITION_INT` message (∼28 bytes payload, ∼40 bytes with framing):

Table 8.3: Wire packet sizes for a 40-byte MAVLink message.

Component	Bytes	Notes
AEAD header	22	Fixed
Ciphertext	40	Same as plaintext
Authentication tag	16	Fixed
Total wire	**78**	95% overhead

The fixed 38-byte overhead (22 header + 16 tag) means that the relative overhead decreases as message size increases. For the maximum MAVLink payload (255 bytes + framing ≈ 280 bytes), the overhead is only 14%.

8.7 Summary

- The wire format is: **Header (22 bytes) ‖ Ciphertext + Tag**.
- The header contains: version, algorithm IDs, 8-byte session ID, 8-byte sequence, 1-byte epoch.
- The header is **cleartext but authenticated** (passed as AAD).
- The **nonce is not transmitted**—it is reconstructed from epoch + sequence, saving 12 bytes/packet.

- The `Sender` manages encryption with monotonic sequence numbering and epoch management.
- The `Receiver` validates headers, checks the sliding replay window, and decrypts.
- Three AEAD backends are supported: AES-GCM and ChaCha20-Poly1305 (via `cryptography`), ASCON-128a (via native C or `pyascon`).
- The replay window (default 1024) tolerates UDP reordering while detecting duplicates and replays.

Chapter 9

The Proxy Engine

The proxy is a bridge with a guard at each end: every packet must prove its identity before crossing.

This chapter examines the heart of the system: the selectors-based proxy engine in `core/async_proxy.py`. This single module (1,663 lines) manages the complete packet lifecycle: socket management, handshake orchestration, encryption, decryption, rate limiting, rekey, and metrics collection.

9.1 Role Duality

The same code runs on both the drone and the GCS. The `role` parameter determines behavior:

`role="drone"`: The drone proxy is the TCP **client** (initiates the handshake) and bridges plaintext ports 47003/47004.

`role="gcs"`: The GCS proxy is the TCP **server** (listens for the handshake) and bridges plaintext ports 47001/47002.

The data-plane encryption/decryption logic is identical for both roles—only the direction-specific keys differ (`key_d2g` vs. `key_g2d`).

9.2 Socket Architecture

The proxy manages four UDP sockets, all registered with the `selectors` event loop:

Key Insight

The plaintext sockets bind to `127.0.0.1` (loopback only), ensuring that unencrypted traffic cannot leak onto the network. The encrypted sockets bind to `0.0.0.0` (all interfaces), allowing the peer to reach them over WiFi. This binding separation is a security invariant.

Table 9.1: The four UDP sockets managed by the proxy.

Socket	Bind	Direction	Purpose
Plaintext IN	`127.0.0.1:4700x`	App → Proxy	Receives plaintext MAVLink from local app
Plaintext OUT	(sendto)	Proxy → App	Delivers decrypted MAVLink to local app
Encrypted IN	`0.0.0.0:4601x`	Peer → Proxy	Receives encrypted packets from network
Encrypted OUT	(sendto)	Proxy → Peer	Sends encrypted packets to peer over network

9.3 The Event Loop

The main event loop uses `selectors.DefaultSelector` (typically `epoll` on Linux, `select` on Windows):

```
sel = selectors.DefaultSelector()
sel.register(ptx_in_sock, EVENT_READ, handle_plaintext_in)
sel.register(enc_in_sock, EVENT_READ, handle_encrypted_in)

while not stop_event.is_set():
    events = sel.select(timeout=0.1)
    for key, mask in events:
        callback = key.data
        callback(key.fileobj)

    # Periodic tasks: rekey checks, metrics sampling,
    # rate-limit pruning, control channel polling
    periodic_tick()
```

Listing 9.1: Simplified event loop (conceptual)

9.3.1 Plaintext Ingress Path

When the plaintext-in socket is readable:

1. `recvfrom()` reads the datagram (up to 65535 bytes).
2. The proxy calls `Sender.encrypt(plaintext)`.
3. The resulting wire packet is sent to the peer's encrypted port via `sendto()`.
4. Counters are updated: `ptx_in++`, `enc_out++`, byte counts.

9.3.2 Encrypted Ingress Path

When the encrypted-in socket is readable:

1. `recvfrom()` reads the datagram and source address.

2. **Source address check:** If `CONFIG["STRICT_UDP_PEER_MATCH"]` is enabled, verify the source IP matches the expected peer. Drop silently if not.
3. The proxy calls `Receiver.decrypt(wire)`.
4. If decryption returns `None` (silent mode): increment the appropriate drop counter based on `last_error_reason()`.
5. If decryption succeeds: send the plaintext to the local app via `sendto()` on the plaintext-out socket.
6. Counters are updated: `enc_in++`, `ptx_out++`, byte counts.

9.4 Handshake Orchestration

The proxy performs the TCP handshake before entering the main event loop:

GCS (server) mode: Binds a TCP socket to port 46000, calls `accept()` with a configurable deadline, then executes `server_gcs_handshake()` from `core/handshake.py`.

Drone (client) mode: Creates a TCP socket, connects to the GCS at `<GCS_HOST>:46000`, then executes `client_drone_handshake()`.

The handshake returns:

- Two 32-byte symmetric keys (send and receive).
- The 8-byte session ID.
- The negotiated KEM and SIG algorithm names.
- The authenticated peer address (from TCP `getpeername()`).
- A metrics dictionary with nanosecond-precision timings.

Implementation Note

The GCS mode includes IP-based **rate limiting** on the TCP accept path using a token bucket algorithm. This prevents handshake flood attacks, where an attacker rapidly connects and forces expensive KEM key generation (especially problematic for Classic McEliece). The default allows 3 handshake attempts per second per IP with a burst capacity of 5.

9.5 Rate Limiting

The `_TokenBucket` class implements per-IP rate limiting:

Capacity: Maximum burst size (default: 5 tokens).

Refill rate: Tokens per second (default: 3/second).

Pruning: Stale entries are removed after a configurable idle timeout to prevent memory growth during long-running deployments.

Each incoming TCP connection attempt consumes one token. If the bucket for that IP is empty, the connection is rejected without performing any cryptographic work.

9.6 Drop Accounting

The `ProxyCounters` class tracks every packet drop with granular reasons:

Table 9.2: Drop reason counters in the proxy.

Counter	Meaning
`drop_replay`	Packet failed anti-replay check (duplicate or too old)
`drop_auth`	AEAD authentication failed (tampered or wrong key)
`drop_header`	Header validation failed (wrong version or algorithm IDs)
`drop_session_epoch`	Session ID or epoch mismatch
`drop_src_addr`	Source IP did not match expected peer
`drop_other`	Any other failure

The `_parse_header_fields()` helper pre-classifies packets without performing AEAD work, enabling the proxy to report the most likely drop reason when decryption returns `None` in silent mode.

9.7 DSCP Marking

The proxy supports DSCP (Differentiated Services Code Point) marking on encrypted packets:

```
tos = dscp_value << 2   # DSCP occupies upper 6 bits of TOS byte
sock.setsockopt(socket.IPPROTO_IP, socket.IP_TOS, tos)
```

Listing 9.2: DSCP socket option

This allows network equipment (routers, access points) to prioritize tunnel traffic over background traffic, which is important for maintaining low-latency MAVLink communication.

9.8 Rekey Integration

The proxy integrates with the policy engine for seamless rekey:

1. The policy engine requests a rekey (via the two-phase commit protocol).
2. The proxy enters a **blackout period**: it stops encrypting new packets while the rekey handshake executes.
3. A new TCP handshake is performed (same as initial, but with the new suite).
4. If the handshake succeeds: new `Sender` and `Receiver` are instantiated with fresh keys.

5. If the handshake fails: the proxy reverts to the previous keys (abort path).

6. The blackout duration is recorded as a metric (`rekey_blackout_duration_ms`).

Key Insight

During the blackout, packets arriving on the encrypted port may be encrypted with either the old or new keys. The receiver checks the session ID and epoch in the header to determine which keys to use, ensuring a smooth transition.

9.9 Metrics Collection

The `ProxyCounters` collects real-time metrics:

- **Packet counts:** Plaintext in/out, encrypted in/out.
- **Byte counts:** Total bytes in each direction.
- **Drop counts:** Per-reason drop accounting.
- **Rekey statistics:** Success/failure counts, last rekey duration, blackout duration, trigger reason.
- **AEAD primitive timing:** Per-operation nanosecond-precision timing for encrypt, decrypt-success, and decrypt-failure, aggregated as count, total, min, max.
- **Part B metrics:** Flattened handshake metrics (KEM keygen/encap/decap, SIG sign/verify times and artifact sizes).

The `to_dict()` method serializes all counters into a dictionary that is consumed by the metrics aggregator and ultimately written to the JSONL benchmark output.

9.10 The CLI Entry Point

The `core/run_proxy.py` module provides three subcommands:

`init-identity`: Generates a fresh signature key pair for the GCS. Saves the private key and public key to files.

`gcs`: Launches the GCS proxy. Loads the GCS signature private key, starts the handshake server, and enters the event loop.

`drone`: Launches the drone proxy. Loads the GCS public key (pre-installed), connects to the GCS, and enters the event loop.

Matrix-mode key loading is supported for benchmarking: the system can load different signature key pairs for different suites, enabling automated testing of all 72 suites without manual key management.

9.11 Summary

- The proxy runs on both drone and GCS, distinguished only by `role` and configuration.
- Four UDP sockets handle the two directions (plaintext and encrypted) of the bidirectional tunnel.
- The selectors-based event loop processes packets as they arrive, with periodic tasks for rekey, metrics, and rate-limit pruning.
- IP-based rate limiting (token bucket) protects the handshake from flood attacks.
- Granular drop accounting enables precise diagnosis of packet loss.
- DSCP marking enables QoS prioritization on the network.
- Rekey transitions are managed via a blackout period with old/new key coexistence.
- AEAD primitive timing is collected at nanosecond precision for benchmarking.

Chapter 10

The Suite Registry

The suite registry is a Cartesian product machine: given n KEMs, m AEADs, and k signatures, it generates $n \times m \times k$ suites—but only those that make cryptographic sense.

This chapter examines `core/suites.py`, the module that defines, generates, validates, and resolves the 72+ cryptographic suites available in the system.

10.1 The Three Registries

The suite system is built from three independent registries:

10.1.1 KEM Registry

Nine entries across three families, each with a canonical token, OQS algorithm name, NIST level, and numeric IDs for the wire header:

Table 10.1: KEM registry entries.

Token	OQS Name	Level	kem_id	kem_param_id
`mlkem512`	ML-KEM-512	L1	1	1
`mlkem768`	ML-KEM-768	L3	1	2
`mlkem1024`	ML-KEM-1024	L5	1	3
`classicmceliece348864`	Classic-McEliece-348864	L1	3	1
`classicmceliece460896`	Classic-McEliece-460896	L3	3	2
`classicmceliece8192128`	Classic-McEliece-8192128	L5	3	3
`hqc128`	HQC-128	L1	5	1
`hqc192`	HQC-192	L3	5	2
`hqc256`	HQC-256	L5	5	3

10.1.2 Signature Registry

Eight entries across three families:

Table 10.2: Signature registry entries.

Token	OQS Name	Level	sig_id	sig_param_id
`mldsa44`	ML-DSA-44	L1	1	1
`mldsa65`	ML-DSA-65	L3	1	2
`mldsa87`	ML-DSA-87	L5	1	3
`falcon512`	Falcon-512	L1	2	1
`falcon1024`	Falcon-1024	L5	2	2
`sphincs128s`	SPHINCS+-SHA2-128s-simple	L1	3	1
`sphincs192s`	SPHINCS+-SHA2-192s-simple	L3	3	2
`sphincs256s`	SPHINCS+-SHA2-256s-simple	L5	3	3

10.1.3 AEAD Registry

Three entries, all paired with HKDF-SHA256 for key derivation:

Table 10.3: AEAD registry entries.

Token	Display Name
`aesgcm`	AES-256-GCM
`chacha20poly1305`	ChaCha20-Poly1305
`ascon128a`	Ascon-128a

10.2 Suite Generation

Suites are generated in two stages:

10.2.1 Stage 1: Level-Consistent Pairing

The function `_generate_level_consistent_matrix()` iterates over all KEM–SIG pairs and keeps only those sharing the same NIST level:

$$\text{Matrix} = \{(k, s) \mid k \in \text{KEMs}, s \in \text{SIGs}, \text{level}(k) = \text{level}(s)\} \tag{10.1}$$

This produces:

- L1: 3 KEMs × 3 SIGs = 9 pairs
- L3: 3 KEMs × 2 SIGs = 6 pairs (no Falcon at L3)
- L5: 3 KEMs × 3 SIGs = 9 pairs
- Total: 24 KEM–SIG pairs

10.2.2 Stage 2: AEAD Cross Product

Each KEM–SIG pair is crossed with all three AEAD tokens:

$$\text{Suites} = \{(k, a, s) \mid (k, s) \in \text{Matrix}, a \in \{\text{aesgcm}, \text{chacha20poly1305}, \text{ascon128a}\}\} \tag{10.2}$$

Total: $24 \times 3 = 72$ suites.

10.2.3 Suite ID Format

Each suite has a canonical identifier:

$$\text{suite\_id} = \texttt{cs-}\langle\text{kem\_token}\rangle\texttt{-}\langle\text{aead\_token}\rangle\texttt{-}\langle\text{sig\_token}\rangle \tag{10.3}$$

Examples:

- `cs-mlkem768-aesgcm-mldsa65` (default suite)
- `cs-classicmceliece348864-chacha20poly1305-falcon512`
- `cs-hqc256-ascon128a-sphincs256s`

10.3 Alias Resolution

The registry supports extensive aliasing to handle the naming chaos of post-quantum cryptography:

- **Legacy NIST names:** `kyber512` → `mlkem512`, `dilithium3` → `mldsa65`
- **Case/punctuation variants:** `ML-KEM-768`, `ml-kem-768`, `mlkem768` all resolve to the same entry.
- **Suite-level aliases:** `cs-kyber512-aesgcm-dilithium2` → `cs-mlkem512-aesgcm-mldsa44`
- **SPHINCS+ variant aliases:** Both "f" (fast) and "s" (small) aliases map to the "s" variant.

Resolution uses `_normalize_alias()`, which strips all non-alphanumeric characters and lowercases, then looks up in a precomputed alias map.

Design Decision

Aggressive aliasing was a deliberate choice because the PQC naming landscape is in flux. NIST renamed Kyber to ML-KEM, Dilithium to ML-DSA, and SPHINCS+ to SLH-DSA during the standardization process. Research papers, library versions, and configuration files may use any of these names. The alias system ensures that all variants work without user confusion.

10.4 Runtime Probing

Not all algorithms are available on all platforms. The registry probes for availability at startup:

10.4.1 OQS Mechanism Discovery

`enabled_kems()` and `enabled_sigs()` query the `oqs-python` library for available mechanisms, trying multiple import styles for compatibility:

```
try:
    from oqs.oqs import get_enabled_KEM_mechanisms
except ImportError:
    from oqs import get_enabled_KEM_mechanisms
```

Listing 10.1: Multi-style OQS import (simplified)

10.4.2 AEAD Availability

`_probe_aead_support()` checks:

1. AES-GCM: always available (part of the `cryptography` library).
2. ChaCha20-Poly1305: available in most `cryptography` builds, but optional.
3. ASCON-128a: requires either the native C extension or the `pyascon` package. Can be disabled via `CONFIG["ENABLE_ASCON"]`.

10.4.3 Suite Pruning

`_prune_suites_for_runtime()` filters the global `SUITES` registry, removing suites whose signature algorithms are not available in the current `oqs-python` build. This prevents runtime errors from attempting to use an unavailable algorithm.

10.5 Environment Overrides

Two environment variables control suite generation:

`SUITES_IGNORE_KEMS`: Comma-separated list of KEM registry keys to exclude. Useful for skipping Classic McEliece in quick benchmark runs.

`SUITES_IGNORE_AEADS`: Comma-separated list of AEAD tokens to exclude.

10.6 Query Functions

The registry provides a rich API for querying suites:

`list_suites()`: Returns all available suites as a dictionary.

`get_suite(suite_id)`: Returns a single suite by ID, resolving aliases.

`list_suites_for_level(level)`: Returns suites restricted to a single NIST level.

`filter_suites_by_levels(levels)`: Returns suite IDs matching any of the given levels.

`valid_nist_levels()`: Returns the distinct NIST levels present in the registry.

`header_ids_for_suite(suite)`: Returns the four numeric IDs embedded in the wire header.

`header_ids_from_names(kem, sig)`: Resolves header IDs from algorithm names.

10.7 Immutability

The generated registry is wrapped in `types.MappingProxyType`, a read-only dictionary view:

```
from types import MappingProxyType
SUITES = MappingProxyType(suites_dict)
```

Listing 10.2: Immutable registry

This prevents accidental modification of the registry during runtime. The `get_suite()` function returns a **copy** (`dict(suite)`) rather than the original, so callers can safely mutate the result.

10.8 Summary

- Three independent registries (KEM, SIG, AEAD) are crossed to produce suites.
- **Level-consistent pairing** ensures that KEM and SIG share the same NIST security level.
- **72 suites** are generated automatically from 9 KEMs × 3 AEADs × 8 SIGs (level-filtered).
- Extensive **alias resolution** handles the naming chaos of evolving PQC standards.
- **Runtime probing** discovers which algorithms are actually available and prunes unavailable suites.
- The registry is **immutable** (MappingProxyType) to prevent accidental modification.
- Environment variables allow KEM/AEAD exclusion for focused benchmark runs.

Part IV

Orchestration and Measurement

Chapter 11

Benchmark Orchestration and Scheduling

Chapters 7 through 10 showed how to build a single encrypted session for one cipher suite. But to *evaluate* all 72 suites we need an automated pipeline that cycles through every suite, starts and stops proxies on both machines, collects metrics, handles failures, and writes results to persistent storage—all without human intervention. This chapter describes that pipeline: the **benchmark orchestration scheduler**.

Key Insight

The scheduler is not a real-time flight controller. It is a lab-automation tool that exercises every suite in sequence so that researchers can compare them later. Think of it as a robot lab technician that methodically runs the same experiment 72 times, records every measurement, and neatly files the results.

11.1 Controller–Follower Architecture

The benchmark involves two physical machines—Drone (Raspberry Pi 5) and GCS (Windows laptop)—connected by a LAN cable. One machine must be in charge of deciding *when* to switch suites and the other must comply. We call these roles the **controller** and the **follower**.

Design Decision

The *drone* is the controller. This may seem surprising—isn't the GCS more powerful? Yes, but in a real mission the drone is the constrained device whose performance we want to measure under stress. Making it the controller lets us time every operation from the drone's perspective, which is the perspective that matters for resource-constrained analysis.

- **Drone Controller** (`sdrone_bench.py`): Decides when to advance to the next suite, starts its own proxy, reads handshake metrics, collects MAVLink and system metrics, and writes all results to disk.

- **GCS Follower** (`sgcs_bench.py`): Listens for commands from the drone on a TCP control channel, starts/stops its own proxy on demand, collects GCS-side validation metrics, and returns them to the drone upon request.

Figure 11.1 shows the high-level interaction.

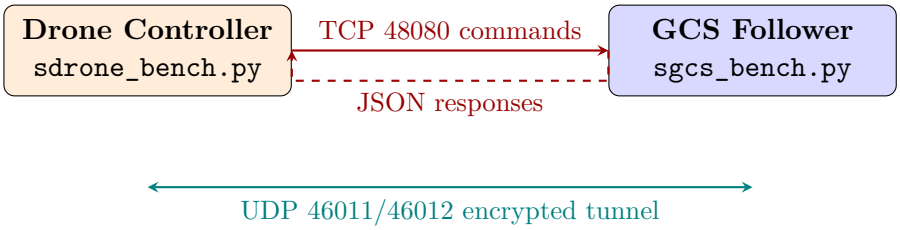


Figure 11.1: Controller–follower architecture. The drone sends commands over a plaintext TCP control channel; the encrypted tunnel carries MAVLink data.

11.2 The TCP Control Channel

The control channel is a simple JSON-over-TCP protocol. The drone opens a fresh TCP connection to `GCS_CONTROL_HOST:48080` for every command, sends a single JSON object terminated by a newline, reads the response, and closes the socket. Table 11.1 lists the supported commands.

Table 11.1: TCP control channel commands

Command	Direction	Description
`ping`	D → G	Liveness check; GCS responds "pong"
`get_info`	D → G	Returns GCS hostname, IP, kernel, Python version
`chronos_sync`	D → G	Clock synchronisation (NTP-lite 3-way handshake)
`start_proxy`	D → G	Start GCS-side proxy for a given suite and run ID
`prepare_rekey`	D → G	Stop GCS proxy in preparation for suite switch
`start_traffic`	D → G	Start synthetic UDP traffic (disabled in MAVProxy mode)
`stop_suite`	D → G	Stop current suite and return GCS-side metrics
`shutdown`	D → G	Graceful server shutdown

Analogy

Think of the control channel as a walkie-talkie between a pit crew chief (drone) and the garage mechanic (GCS). The chief says "install suite #17," the mechanic

does the work and radios back "done." The actual race (encrypted tunnel) happens on a separate track.

The `send_gcs_command` function on the drone side implements the protocol:

```
def send_gcs_command(cmd, **kwargs):
    payload = {"cmd": cmd, **kwargs}
    sock = socket.create_connection(
        (GCS_CONTROL_HOST, GCS_CONTROL_PORT),
        timeout=max(90, REKEY_HANDSHAKE_TIMEOUT + 15)
    )
    sock.sendall(json.dumps(payload).encode() + b"\n")
    data = sock.recv(65536)
    sock.close()
    return json.loads(data)
```

Listing 11.1: Sending a command to GCS (simplified)

11.2.1 Clock Synchronisation: Operation Chronos

Before any benchmark begins, the drone and GCS must agree on a shared time reference. The module `core/clock_sync.py` implements a three-way NTP-lite handshake:

1. **T1**: Drone records its wall-clock time and sends it to GCS.
2. **T2**: GCS records the arrival time.
3. **T3**: GCS records the departure time and responds with (T_1, T_2, T_3).
4. **T4**: Drone records the response arrival time.

The clock offset (GCS − Drone) is:

$$\text{offset} = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

The drone re-synchronises periodically—every 10 suites or every 1200 seconds, whichever comes first—to bound cumulative drift.

Security Note

The clock sync channel is *unauthenticated* (plaintext TCP). An active attacker could inject false timestamps and skew the offset, making the drone believe suite intervals are shorter or longer than they really are. This is acceptable in a lab setting but would need HMAC protection in a production deployment.

11.3 Benchmark Modes

The scheduler supports two benchmark modes, resolved by a deterministic priority chain: CLI argument > environment variable `BENCHMARK_MODE` > default.

MAVPROXY (default): A physical flight controller is connected via serial, and a real `MAVProxy` process forwards MAVLink messages through the encrypted tunnel. This mode captures genuine telemetry latency, jitter, and message integrity. Synthetic traffic generation is *disabled*.

SYNTHETIC : No flight controller is present. The scheduler generates UDP traffic at a configurable rate to simulate payload. Useful for pure cryptographic benchmarking without hardware dependencies.

Both modes share the same suite-cycling logic; only the data-plane source differs.

11.4 The BenchmarkPolicy

The `BenchmarkPolicy` class (in `sscheduler/benchmark_policy.py`) is the decision engine that answers one question: *"Should we stay on the current suite or advance?"*

11.4.1 Data Structures

`BenchmarkAction` An enum with three values: `HOLD` (stay on current suite), `NEXT_SUITE` (advance to the next one), and `COMPLETE` (all suites tested).

`SuiteMetrics` A dataclass capturing every measurement for a single suite: handshake timing, KEM/SIG primitive breakdowns, artifact sizes, power, energy, throughput, latency, and a success flag.

`BenchmarkOutput` The return value of `evaluate()`. Contains the proposed action, the target suite, progress percentage, elapsed time, and human-readable reasons.

11.4.2 Two-Phase Commit Protocol

A critical design pattern is the **two-phase commit** between `evaluate()` and `confirm_advance()`.

Key Insight

`evaluate()` is a *pure query*—it examines the current state and *proposes* an action (HOLD, NEXT_SUITE, or COMPLETE) without modifying any state. `confirm_advance()` is the *commit*—it advances the index, starts metrics for the next suite, and (on COMPLETE) saves results. The caller *must* call `confirm_advance()` exactly once per accepted proposal.

Why this separation? Because the caller may need to perform work between the decision and the commit:

1. `evaluate(now)` → `NEXT_SUITE`
2. Caller collects GCS metrics for the current suite.
3. Caller finalises the current suite's metrics (`finalize_suite_metrics`).
4. Caller calls `confirm_advance(now)` to commit the advance.
5. Caller stops the old proxy and starts the new one.

Implementation Note

The ordering in step 2–4 is **critical**: metrics must be finalised *before* the index advances, because `confirm_advance()` calls `_start_suite_metrics()` for the *next* suite. If the order were reversed, the finalised metrics would belong to the wrong suite.

11.4.3 Suite Cycling

The policy builds an ordered list of suites at initialisation time:

1. Start with all suites from the registry (`list_suites()`).
2. Optionally filter by AEAD (e.g. `-filter-aead aesgcm`).
3. Sort by (NIST level, KEM name, SIG name) for reproducible ordering.

Each suite runs for a configurable interval (default: 110 seconds). When `evaluate()` detects that the elapsed time on the current suite exceeds the interval, it proposes `NEXT_SUITE`. When the last suite's interval elapses, it proposes `COMPLETE`.

11.4.4 Metrics Collection per Suite

For every suite the policy tracks:

Table 11.2: SuiteMetrics fields collected by BenchmarkPolicy

Field	Type	Source
`handshake_ms`	`float`	Proxy status file
`kem_keygen_ms`	`float`	Handshake metrics
`kem_encaps_ms`	`float`	Handshake metrics
`kem_decaps_ms`	`float`	Handshake metrics
`sig_sign_ms`	`float`	Handshake metrics
`sig_verify_ms`	`float`	Handshake metrics
`pub_key_size_bytes`	`int`	Handshake metrics
`ciphertext_size_bytes`	`int`	Handshake metrics
`sig_size_bytes`	`int`	Handshake metrics
`power_w`	`float`	Power monitor
`energy_mj`	`float`	Computed
`throughput_mbps`	`float`	Data plane counters
`latency_ms`	`float`	MAVLink timing
`success`	`bool`	Handshake outcome

11.4.5 Result Persistence

When the benchmark completes, `_save_results()` writes two files:

- A **JSON** file containing all suite metrics, suite identifiers, run metadata, and AEAD filter settings.

- A **CSV** file with one row per suite, suitable for import into spreadsheet or data-analysis tools.

Both files are timestamped with the run ID (`benchmark_results_YYYYMMDD_HHMMSS.json`).

11.5 The Drone Benchmark Scheduler

The `BenchmarkScheduler` class in `sdrone_bench.py` is the main entry point on the drone side. It orchestrates the full lifecycle:

11.5.1 Initialisation

1. Parse CLI arguments: `-mav-master`, `-interval`, `-filter-aead`, `-max-suites`, `-dry-run`, `-gcs-host`, `-mode`.
2. Resolve benchmark mode (MAVPROXY or SYNTHETIC).
3. Create a `DroneProxyManager` (wraps `ManagedProcess`).
4. Create a `BenchmarkPolicy` with the chosen interval and AEAD filter.
5. Create a run-specific log directory: `logs/benchmarks/live_run_YYYYMMDD_HHMMSS/`.
6. Initialise the `MetricsAggregator` (comprehensive 18-category metrics) and the `RobustLogger` (aggressive append-mode JSONL logging).

11.5.2 The Main Loop

Figure 11.2 shows the lifecycle of a single suite within the main loop.

The `_run_loop()` method implements this cycle:

1. Call `_activate_suite(suite_name)`, which:
   - Checks whether a clock re-sync is needed.
   - Sends `start_proxy` to GCS.
   - Starts the drone-side proxy via `DroneProxyManager`.
   - Waits up to 45 seconds for handshake completion (to accommodate Classic McEliece's large key operations).
   - Records handshake metrics in the policy and aggregator.
2. Enter a polling loop that repeatedly calls `policy.evaluate(time.monotonic())`:

   HOLD Sleep for the remaining interval fraction (capped at 1 second to avoid overshooting).

   NEXT_SUITE Collect GCS metrics, finalise the current suite, call `confirm_advance()`, stop the old proxy, and start the new one.

   COMPLETE Same as NEXT_SUITE, but then exit the loop.
3. Before advancing, check `_ready_to_advance()`, which verifies that:

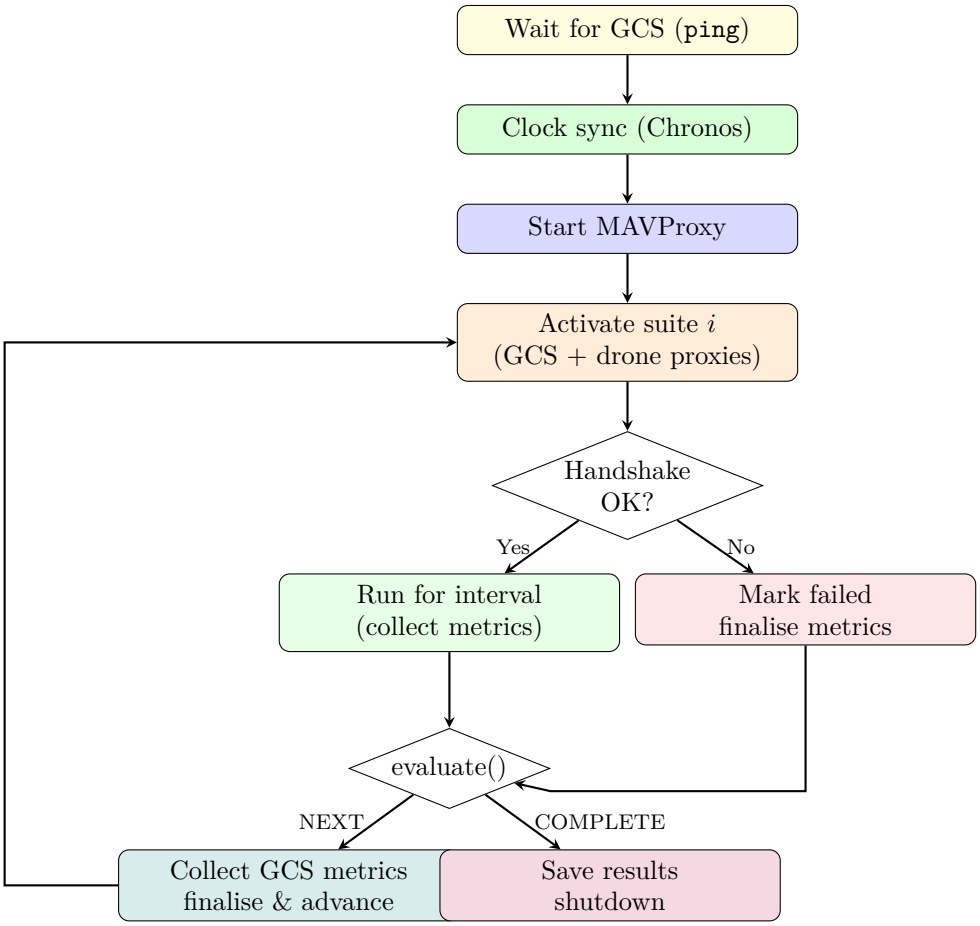


Figure 11.2: Lifecycle of the benchmark main loop. Each suite goes through activation, handshake, metric collection, and policy evaluation.

- MAVProxy is still alive (in MAVPROXY mode).
- The GCS control channel is responsive.
- MAVLink messages have been received.
- Data-plane counters are non-zero.

If not ready, extend the suite by up to `interval + 10` seconds (the "grace period").

Implementation Note

The policy always uses `time.monotonic()` rather than `synced_time()`. This prevents time-domain mixing: synced time returns a wall-clock value ($\sim 1.7 \times 10^9$), which would make elapsed-time calculations meaningless when mixed with monotonic timestamps that start near zero.

11.5.3 The DroneProxyManager

This helper class wraps the drone proxy subprocess:

- **start(suite_name)**: Looks up the suite, locates the GCS public key in `secrets/matrix/{suite}/`, constructs the CLI command, and launches it via `ManagedProcess` with output redirected to a timestamped log file.
- **stop()**: Terminates the managed process, closes the log file handle (to prevent file-descriptor leaks), and resets state.
- **is_running()**: Delegates to `ManagedProcess.is_running()`.

11.5.4 Handshake Status Reader

After starting the drone proxy, the scheduler polls a JSON status file (`drone_status.json`) for up to 45 seconds. The proxy writes this file immediately after handshake completion with one of two statuses:

- `"handshake_ok"`: Written right after handshake success, containing the full handshake metrics.
- `"running"`: Written during periodic status updates. Both are accepted as proof of a successful handshake.

11.6 The GCS Benchmark Server

The GCS side runs a `GcsBenchmarkServer` (in `sgcs_bench.py`) that listens on TCP 48080 for drone commands. It is entirely *reactive*—it never initiates a suite switch on its own.

Table 11.3: GCS server internal components

Component	Responsibility
`GcsProxyManager`	Starts/stops GCS proxy subprocesses
`GcsMavProxyManager`	Manages MAVProxy with optional GUI (–map –console)
`MavLinkMetricsCollector`	Sniffs MAVLink on UDP 14552 for validation
`GcsSystemMetricsCollector`	Samples CPU, memory, temperature at 0.5 Hz
`MetricsAggregator`	18-category comprehensive metrics (GCS side)
`ClockSync`	Server side of NTP-lite handshake
`RobustLogger`	Aggressive append-mode JSONL logging

11.6.1 Component Stack

11.6.2 Command Handling

Each incoming TCP connection is handled in a single thread. The server reads one JSON command, dispatches to the appropriate handler, and returns a JSON response. The most important handlers are:

`start_proxy` Accepts a suite name and an optional `run_id`. If the drone's run ID differs from the current one, the server updates its log directory and reinitialises its `MetricsAggregator` and `RobustLogger` to write to the matching folder. It then starts the GCS proxy, resets the MAVLink validation counters, and starts system metrics sampling.

`stop_suite` Stops traffic, collects MAVLink validation metrics (message count + sequence gap count), GCS system metrics, and proxy status. Packages everything into a JSON response that the drone merges into its comprehensive metrics. Also writes to a local JSONL log with fsync and retry (up to 3 attempts).

`chronos_sync` Delegates to `ClockSync.server_handle_sync()`, which records T_2 and T_3 and returns them along with the echoed T_1.

11.6.3 Consistent Log Directories

Both machines must write to the same logical run folder. The drone is the authority on the `run_id` (it is generated from the drone's UTC timestamp at startup). When the GCS receives a `start_proxy` command with a `run_id` it hasn't seen before, it calls `_update_run_id()`, which:

1. Creates a new directory `logs/benchmarks/live_run_{run_id}/`.
2. Stops and replaces the old `GcsProxyManager`.
3. Reinitialises the `MetricsAggregator` and `RobustLogger` with the new run ID.

11.7 Scheduling Policies

The codebase contains several scheduling policies beyond the benchmark policy. These are defined in `sscheduler/policy.py` and are used by the production (non-benchmark) scheduler.

11.7.1 TelemetryAwarePolicyV2

This is the *safety-critical* policy intended for real flight operations. It consumes both GCS telemetry (link quality) and local drone telemetry (battery, temperature) to make rekey and suite-switch decisions.

Table 11.4: Decision hierarchy of TelemetryAwarePolicyV2

Priority	Gate	Condition	Action
1	Safety gate	Telemetry stale ($> 2\,\mathrm{s}$)	HOLD
2	Emergency	Battery critical or temp critical	DOWNGRADE to tier 0
3	Blackout	> 3 blackouts within 30 s of switch	DOWNGRADE + blacklist
4	Cooldown	Still in cooldown window	HOLD
5	Link degradation	Gap P95 $> 1\,\mathrm{s}$ or PPS < 5 (with hysteresis)	DOWNGRADE
6	Stress	Temp rising or battery falling fast	DOWNGRADE
7	Proactive rekey	Stable $> 60\,\mathrm{s}$ and under rate limit	REKEY
8	Upgrade	Disarmed, stable, no stress (30 s hysteresis)	UPGRADE
9	Nominal	None of the above	HOLD

11.7.1.1 Suite Tier Mapping

The policy maps each suite to a numeric tier that encodes its computational weight:

$$\text{tier} = \underbrace{\text{level\_tier}}_{\substack{0=L1\\10=L3\\20=L5}} + \underbrace{\text{kem\_tier}}_{\substack{0=\text{ML-KEM}\\3=\text{HQC}\\5=\text{McEliece}}} + \underbrace{\text{aead\_tier}}_{\substack{0=\text{AES-GCM}\\1=\text{ChaCha20}\\2=\text{ASCON}}}$$

Downgrade moves to a lower tier (lighter); upgrade moves to a higher tier (heavier).

11.7.1.2 Hysteresis

To avoid oscillation (rapidly switching back and forth), the policy requires that a condition persists for a configurable duration before acting:

- **Downgrade hysteresis**: 5 seconds (link degradation or stress must persist for 5 seconds before triggering a downgrade).
- **Upgrade hysteresis**: 30 seconds (stable conditions must persist for 30 seconds before allowing an upgrade).

11.7.1.3 Blacklisting

If a suite causes link failure shortly after activation ($< 30\,\text{s}$), the policy *blacklists* it for a configurable TTL (default: 1800 seconds = 30 minutes). Blacklisted suites are skipped during downgrade/upgrade searches.

11.7.1.4 Rekey Rate Limiting

Rekeys are limited by a sliding window: at most N successful rekeys within W seconds (default: 5 per 300 seconds). Importantly, rekeys are only *recorded* after successful execution (via `record_rekey()`), not when proposed—this prevents failed rekey attempts from consuming the quota.

11.7.2 Simple Policies

`LinearLoopPolicy` Deterministic round-robin through a given suite list. Advances on every call to `next_suite()`. Configurable duration per suite.

`RandomPolicy` Selects a random suite from the pool on every call. Uses its own `random.Random` instance for reproducibility.

`ManualOverridePolicy` Normally round-robin, but accepts an external override via `set_override(suite_name)`. When set, all calls to `next_suite()` return the overridden suite.

`DeterministicClockPolicy` Uses synchronised time (Chronos) to compute the suite index deterministically: index = $\lfloor \text{synced\_time}/\text{interval} \rfloor \bmod |\text{suites}|$. Both machines independently compute the same suite without coordination.

11.8 Subprocess Management

Both the drone and GCS start proxy and MAVProxy instances as child processes. The `ManagedProcess` class (in `core/process.py`) provides cross-platform process management:

- **Windows**: Creates a Win32 Job Object and assigns the child process to it. When the parent dies, the Job Object is closed and all child processes are terminated automatically.
- **Linux**: Sets `PR_SET_PDEATHSIG` via `ctypes.CDLL('libc.so.6')` so that the child receives `SIGTERM` when its parent exits.
- **Graceful stop**: `stop()` sends `SIGTERM` (or `TerminateProcess` on Windows), waits up to a timeout, then escalates to `SIGKILL`.

Security Note

Without `ManagedProcess`, a crash in the scheduler would leave orphaned proxy processes running—potentially with stale keys, consuming resources, and occupying ports. The Job Object / PDEATHSIG mechanism ensures that a scheduler crash always cleans up its children.

11.9 End-to-End Benchmark Flow

Putting it all together, here is the complete sequence for a full benchmark run with, say, 24 suites (filtered to AES-GCM only):

1. **User starts GCS server**: `python -m sscheduler.sgcs_bench -no-gui`
2. **User starts drone scheduler**: `python -m sscheduler.sdrone_bench -interval 110 -filter-aead aesgcm`
3. Drone resolves mode to MAVPROXY.
4. Drone creates `BenchmarkPolicy` with 24 suites, 110 s interval.
5. Drone pings GCS → "pong."
6. Drone performs clock sync → offset recorded.
7. Drone starts MAVProxy.
8. **For each suite** $i = 0 \ldots 23$:
   (a) Drone sends `start_proxy(suite=..., run_id=...)` to GCS.
   (b) GCS starts its proxy subprocess.
   (c) Drone starts its proxy subprocess.
   (d) Both proxies perform the PQC handshake (Chapter 7).
   (e) Drone reads handshake metrics from status file.
   (f) Encrypted MAVLink flows for ~ 110 seconds.
   (g) `policy.evaluate()` returns `NEXT_SUITE` (or `COMPLETE`).
   (h) Drone sends `stop_suite` to GCS, receives GCS metrics.
   (i) Drone calls `finalize_suite_metrics()` then `confirm_advance()`.
   (j) Both proxies are stopped; next suite begins.
9. Drone saves final summary JSON.
10. Drone and GCS both shut down cleanly.

For 24 suites at 110 seconds each, the entire run takes approximately $24 \times 110 = 2{,}640$ seconds ≈ 44 minutes. For all 72 suites: $72 \times 110 = 7{,}920$ seconds ≈ 2 hours 12 minutes.

11.10 Error Handling and Resilience

- **Handshake timeout**: If the proxy handshake does not complete within 45 seconds, the suite is marked as failed (`success=False`) and metrics are finalised with the error reason. The benchmark continues to the next suite.
- **GCS unreachable**: If `send_gcs_command` times out, the result contains `status: error`. The scheduler logs the failure and may retry or skip.

- **MAVProxy crash**: In MAVPROXY mode, if `mavproxy_proc` is no longer running, the scheduler aborts with `shutdown_reason="error: mavproxy_died"`.
- **Ctrl-C**: The signal handler calls `cleanup_environment`, which stops all managed processes. An `atexit` handler provides a second safety net.
- **Re-entrancy guard**: The `_cleanup()` method uses a boolean flag (`_cleanup_done`) to prevent double-cleanup if called from both the signal handler and atexit.

11.11 Chapter Summary

- The benchmark scheduler follows a **controller–follower** architecture: the drone decides, the GCS obeys.
- A **TCP JSON control channel** on port 48080 carries commands and metric responses between the two machines.
- **Clock synchronisation** (Operation Chronos) uses a 3-way NTP-lite handshake to bound timing drift.
- **BenchmarkPolicy** implements a two-phase evaluate/confirm_advance protocol that cleanly separates decision-making from state mutation.
- **TelemetryAwarePolicyV2** is the production policy with 9-level priority, hysteresis, blacklisting, and rate limiting.
- **ManagedProcess** ensures cross-platform subprocess cleanup via Win32 Job Objects and Linux PDEATHSIG.
- A full 72-suite benchmark runs in approximately 2 hours 12 minutes with the default 110-second interval.

Chapter 12

The Metrics Pipeline

A benchmark is only as good as the data it produces. This chapter describes the *metrics pipeline*—the system of collectors, aggregators, and schemas that turns raw sensor readings, packet counters, and timing samples into the structured JSON files that later feed the analysis dashboard.

Key Insight

The pipeline follows a three-stage architecture: **(1)** *Collectors* read hardware sensors and software counters. **(2)** *Aggregator* merges per-collector data into a single typed object. **(3)** *Persistence* writes the object to JSON and JSONL for offline analysis. Think of it as a factory assembly line: raw materials (sensor readings) enter at one end, and a finished product (a complete JSON record) exits the other.

12.1 The 18-Category Schema

Every suite benchmark produces a single `ComprehensiveSuiteMetrics` object, defined in `core/metrics_schema.py`. This object contains 18 nested dataclasses, labelled A through R. Table 12.1 lists every category, its purpose, and its approximate field count.

Design Decision

Category O (GCS system resources) is *retained in the schema but no longer collected*. The GCS is a non-constrained observer—its CPU and memory do not influence policy decisions, suite ranking, or cryptographic selection. Collecting them added overhead without policy value. The fields remain at `None` for forward compatibility.

12.1.1 Schema Design Principles

1. **Typed dataclasses**: Every field has an explicit Python type (`Optional[float]`, `Optional[int]`, etc.). This catches type errors at development time and makes JSON serialisation straightforward via `dataclasses.asdict()`.

Table 12.1: The 18-category metrics schema

Cat	Name	Purpose	Fields
A	Run & Context	Run ID, git hash, hostnames, IPs, clock offset	20
B	Suite Crypto Identity	KEM, SIG, AEAD algorithms and NIST level	8
C	Suite Lifecycle Timeline	Selection, activation, deactivation timestamps	5
D	Handshake Metrics	Total duration, success, failure reason	7
E	Crypto Primitive Breakdown	Per-primitive timing (ns) and artifact sizes (bytes)	14
F	Rekey Metrics	Attempts, successes, failures, intervals	7
G	Data Plane (Proxy)	Throughput, packet counts, drops, AEAD timing	20+
H	Latency & Jitter	One-way latency, RTT, jitter (avg, P95)	12
I	MAVProxy Drone	TX/RX PPS, message counts, heartbeat, seq gaps	15
J	MAVProxy GCS	Validation subset: message count, seq gap count	2
K	MAVLink Integrity	CRC errors, decode errors, drops, duplicates	9
L	Flight Controller	FC mode, armed state, battery, CPU, sensors	10
M	Control Plane	Scheduler action, policy name, suite index	7
N	System Resources (Drone)	CPU, memory, temperature, load averages	12
O	System Resources (GCS)	Retained for schema compatibility (not collected)	11
P	Power & Energy	Voltage, current, power, energy per handshake	8
Q	Observability	Sample counts, collection timestamps	5
R	Validation & Integrity	Pass/fail verdict, per-metric status map	5

2. **Optional fields**: Nearly every field is `Optional[...]`, defaulting to `None`. This means a partially-filled record is still valid—essential when some collectors fail (e.g. power monitoring is unavailable on the dev laptop).

3. **Dual serialisation**: The `ComprehensiveSuiteMetrics` class provides both `to_dict()` / `to_json()` for export and `from_dict()` / `from_json()` for import, enabling round-trip fidelity.

4. **Flat JSON output**: When serialised, the 18 categories become top-level keys (`run_context`, `crypto_identity`, ..., `validation`), each containing a flat dictionary of fields. This structure is directly consumable by the dashboard backend.

12.2 Collectors

Collectors are the lowest layer of the pipeline. Each one reads a specific data source and returns a plain dictionary.

12.2.1 EnvironmentCollector

Captures static context about the run environment—information that does not change during the benchmark:

- Hostname, platform, Python version
- Kernel version (via `platform.platform()`)
- Git commit hash and dirty flag (via `git rev-parse` and `git status`)
- `liboqs` version (detected from the installed package)
- Conda or virtualenv environment name
- IP address (resolved via socket)
- Wall-clock and monotonic start timestamps

This collector runs once at suite start and populates Category A.

12.2.2 SystemCollector

Samples live system resource telemetry using `psutil`:

- CPU usage percent (with rolling min/max/avg statistics)
- CPU frequency (MHz)
- Process RSS and VMS memory (MB)
- Thread count
- System uptime, load averages (1/5/15 min on Linux)
- Temperature and thermal throttling (on Raspberry Pi, via `vcgencmd measure_temp`)

The aggregator runs this collector in a background thread at 0.5 Hz during each suite, then computes summary statistics at suite end. Results populate Categories N (drone) and O (GCS, deprecated).

12.2.3 PowerCollector

The power collector auto-detects the available backend:

INA219 An external current-sense amplifier connected via I^2C. Provides high-frequency sampling (up to $\sim$ 1,100 Hz on the "highspeed" ADC profile).

RPi5 hwmon The Raspberry Pi 5's on-board PMIC exposes voltage, current, and power via the Linux `sysfs/hwmon` interface. No external hardware required.

None On platforms without power monitoring hardware (e.g. the development laptop), the collector returns empty data.

Results populate Category P.

12.2.4 NetworkCollector

Reads NIC-level I/O counters from `psutil.net_io_counters()`: RX/TX bytes, packets, errors, and drops. By computing deltas between consecutive calls, it derives instantaneous throughput rates in Mbps.

12.2.5 MavLinkMetricsCollector

This is the most complex collector. It sniffs live MAVLink traffic from a `pymavlink` UDP connection and tracks dozens of metrics in real time.

12.2.5.1 Architecture

The collector opens a `udpin:` socket on a designated sniff port (typically 14552 on GCS, 47005 on drone) and spawns a daemon thread that calls `conn.recv_match()` in a tight loop. Each received message is dispatched by type to specialised handlers.

12.2.5.2 Tracked Metrics

12.2.5.3 One-Way Latency Estimation

Analogy

Imagine you and a friend each have a stopwatch. Your friend stamps each letter with the time on *her* stopwatch. When the letter arrives, you read the stamp and compare it to *your* stopwatch. The difference is the one-way latency—but only if both stopwatches are synchronised (or at least if you know their offset).

MAVLink messages like `GLOBAL_POSITION_INT` and `ATTITUDE` carry a `time_boot_ms` timestamp from the flight controller's boot clock. The collector establishes a `boot_to_unix_offset_s` by comparing the first such timestamp against the local wall clock. Subsequent messages use this offset to estimate one-way latency:

Table 12.2: MAVLink collector metric groups

Group	Metrics
Message rates	TX/RX packets per second, stream rate Hz
Message counts	Total sent/received, per-type histogram
Heartbeat	Count, expected, loss count, interval (ms), armed state, flight mode
Sequence integrity	Gap count, duplicates, out-of-order count
Command latency	Commands sent, ACKs received, avg & P95 ACK latency (ms)
One-way latency	Derived from `time_usec` in timestamped messages (avg, P95, jitter, validity flag + reason)
Round-trip time	From COMMAND_LONG → COMMAND_ACK (avg, P95)
Errors	CRC errors, decode errors, message drops
Flight controller	Mode, armed, battery V/A/%, CPU load, sensor health

$$\text{latency} = t_{\text{receive}} - (t_{\text{boot\_ms}}/1000 + \text{offset})$$

If fewer than 5 samples are collected, or if no timestamped messages arrive at all, the latency is marked invalid with a reason string such as `"missing_system_time_reference"` or `"insufficient_samples"`.

12.2.5.4 Sequence Gap Detection

Every MAVLink message carries an 8-bit sequence number per system ID. The collector tracks the last sequence number for each system ID and counts:

- **Gaps**: expected ≠ received (messages lost in transit).
- **Duplicates**: same sequence number twice.
- **Out-of-order**: lower sequence number than expected (but not a wraparound).

These populate Category K (MAVLink Integrity).

12.3 The INA219 Power Monitor

The INA219 is a Texas Instruments current-sense amplifier that measures both the voltage across and the current through a shunt resistor. It communicates over I^2C.

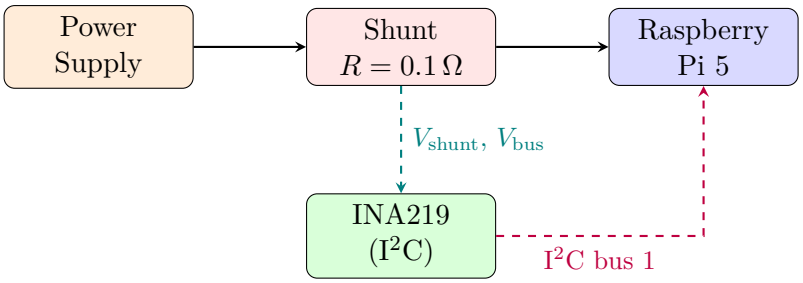


Figure 12.1: INA219 measurement topology. The sensor sits in-line between the power supply and the Pi, measuring the voltage drop across the shunt resistor.

12.3.1 Register-Level Access

Rather than using a high-level library, the codebase accesses INA219 registers directly via `smbus2`:

- **Configuration register** (0x00): Sets bus voltage range (32 V), PGA gain (±320 mV), and ADC resolution.
- **Shunt voltage register** (0x01): Raw 16-bit value; $V_{\text{shunt}} = \text{raw} \times 10\,\mu\text{V}$.
- **Bus voltage register** (0x02): Raw 16-bit value shifted right by 3; $V_{\text{bus}} = \text{raw} \times 4\,\text{mV}$.

Current is computed via Ohm's law:

$$I = \frac{V_{\text{shunt}}}{R_{\text{shunt}}} \qquad P = V_{\text{bus}} \times I$$

12.3.2 ADC Profiles

The INA219 ADC resolution and averaging mode control the trade-off between sample rate and precision:

Table 12.3: INA219 ADC profiles

Profile	Effective Hz	Settle (μs)	Use case
highspeed	$\sim$ 1,100	400	Handshake energy bursts
balanced	$\sim$ 900	1000	General benchmarking
precision	$\sim$ 450	2000	Long-term power profiling

12.3.3 Capture and Energy Integration

The `capture(label, duration_s)` method runs a timed sampling loop:

1. Allocate a CSV file in the output directory.
2. Loop for `duration_s` seconds at `sample_hz`, using `time.perf_counter_ns()` for tick-based scheduling.

3. On each tick: read shunt voltage, compute current, read bus voltage, compute power, write a row to CSV.
4. After the loop: compute summary statistics (average V/I/P, peak P, total energy via trapezoidal integration).
5. Return a `PowerSummary` dataclass.

Energy is computed as:

$$E = \sum_{i=1}^{N-1} \frac{P_i + P_{i+1}}{2} \cdot \Delta t_i$$

where Δt_i is the time between consecutive samples.

12.3.4 RPi5 hwmon Backend

The Raspberry Pi 5's on-board PMIC (Power Management IC) exposes power telemetry via the Linux `sysfs/hwmon` filesystem. The `Rpi5HwmonPowerMonitor` class auto-discovers the correct hwmon directory by scanning for known chip names (e.g. `rpi_volt`), then reads:

- `in0_input` or `voltage0_input` for voltage (mV)
- `curr0_input` or `current0_input` for current (mA)
- `power0_input` for power (if available)

Scale factors convert raw sysfs values to SI units. The capture and `iter_samples` methods have identical signatures to the INA219 monitor, making the two backends interchangeable.

12.3.5 Sign Resolution

Current direction depends on how the shunt resistor is wired. The `_resolve_sign()` method reads a burst of shunt voltage samples and checks the median:

- If positive: sign factor = +1.
- If negative: sign factor = −1.
- If configured as "auto": uses the median's sign.

This ensures that current is always reported as a positive number regardless of wiring orientation.

12.4 The Metrics Aggregator

The `MetricsAggregator` class (in `core/metrics_aggregator.py`) is the central orchestrator that wires collectors to schema categories. It runs on both GCS and drone, with role-specific logic.

12.4.1 Lifecycle

1. **Initialisation**: Detect role (drone or GCS), create collectors (environment, system, network, power on drone, MAVLink on both sides).
2. `start_suite(suite_id, suite_config)`:
   - Create a fresh `ComprehensiveSuiteMetrics` object.
   - Populate Categories A (run context) and B (crypto identity) from the environment collector and suite configuration.
   - Record suite selection time (Category C).
   - Start the MAVLink collector's sniffing thread.
   - Start background system-metric sampling.
3. `record_handshake_start()` / `record_handshake_end()`: Populate Category D with handshake timing.
4. `record_crypto_primitives(metrics)`: Populate Category E from the handshake metrics dictionary.
5. `record_data_plane_metrics(counters)`: Populate Category G from proxy status-file counters.
6. `record_control_plane_metrics(...)`: Populate Category M with scheduler state.
7. `finalize_suite(merge_from=None)`:
   - Stop background collectors.
   - Compute summary statistics for system metrics (avg, peak CPU).
   - Pull MAVLink metrics (populates Categories H, I, J, K, L).
   - Pull power metrics (Category P).
   - If `merge_from` is provided (GCS metrics dict from the control channel), merge it into the object.
   - Compute validation verdict (Category R): check that MAVLink messages were received, latency is valid, and data-plane counters are non-zero.
   - Return the completed `ComprehensiveSuiteMetrics` object.
8. `save_suite_metrics(metrics)`: Write the object to a JSON file in the output directory, named `{timestamp}_{suite_id}_{role}.json`.

12.4.2 Cross-Side Metric Merging

The drone is the authority for the final combined record. When the GCS responds to a `stop_suite` command, it includes a `metrics_export` dictionary containing its aggregator's data. The drone's aggregator receives this via the `merge_from` parameter of `finalize_suite()` and copies GCS-specific fields (MAVLink validation counts, system metrics, latency/jitter from the GCS perspective) into the combined object.

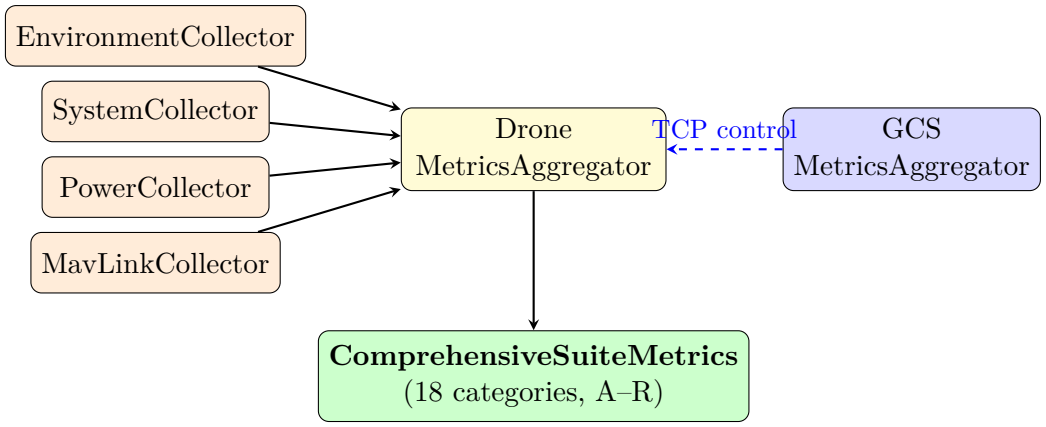


Figure 12.2: Metric flow: four drone-side collectors and the GCS aggregator merge into a single ComprehensiveSuiteMetrics object.

12.4.3 Validation Verdict (Category R)

After all metrics are collected, the aggregator computes a pass/fail verdict:

1. **MAVLink message check**: If `mavproxy_drone_total_msgs_received` is zero or null, mark `mavlink_no_messages`.
2. **Latency validity check**: If neither one-way latency nor RTT is valid, and the reason is *not* structural (e.g. `missing_system_time_reference`), mark `mavlink_latency_invalid`
3. **Data-plane check**: If both `packets_sent` and `packets_received` are zero, mark `data_plane_no_traffic`.
4. If any check fails: `benchmark_pass_fail` ← `"FAIL"`, and the corresponding reason is recorded in the `metric_status` dictionary.

12.5 The RobustLogger

Benchmarks can crash mid-suite—the flight controller may reboot, a proxy may segfault, or the Raspberry Pi may lose power. If metrics are only written at suite end, a crash means total data loss for that suite.

The `RobustLogger` (in `core/robust_logger.py`) solves this with **aggressive append-mode logging**:

1. Every metric and event is immediately appended to a `.jsonl` (JSON Lines) file with `os.fsync()`, not batched until suite end.
2. Suite-level summary files are written atomically (temp file → rename) to prevent corruption from partial writes.
3. File I/O uses up to 3 retries with exponential back-off.
4. Events are buffered in memory (up to 50 entries or 10 seconds) and flushed by a background daemon thread.
5. Cross-platform file locking (`msvcrt.locking` on Windows, `fcntl.flock` on Unix) prevents concurrent write corruption.

Analogy

Think of the RobustLogger as a flight recorder ("black box") on an aircraft. It records every event as it happens, in a format that survives a crash. You don't wait until the plane lands to write the data—you write it continuously, so that if something goes wrong, investigators can reconstruct what happened.

12.5.1 Output Files

For each benchmark run, the RobustLogger produces:

- `events_{role}.jsonl`: Every event (suite start, handshake, errors, sync) as a timestamped JSON line.
- `metrics_{role}.jsonl`: Incremental metric snapshots (handshake timings, data plane counters, GCS metrics) appended as they become available.
- `all_suites_{role}.jsonl`: Completed suite summaries, one JSON line per suite.
- Per-suite JSON files with full metrics.
- A progress file tracking how many suites have completed.

12.5.2 SyncTracker

The `SyncTracker` class tracks clock synchronisation quality over time:

- Records `(timestamp, offset_ms)` pairs from each Chronos sync event.
- Computes **drift rate** (ms/hour) via simple linear regression over the sync history.
- `interpolated_offset()` returns an estimated offset at the current time, accounting for drift.
- Keeps the last 100 internal samples and 20 export-ready history records.

12.6 Failure Mode Taxonomy

Not every null, zero, or `FAIL` verdict in the metrics output indicates a bug. This section classifies the failure modes a researcher may encounter when analysing benchmark results, and explains how each affects the Category R validation verdict.

12.6.1 Class 1: Real Failures

These represent genuine errors during the benchmark run. They indicate that something went wrong and the affected suite's data is unreliable.

Handshake timeout. The PQC handshake did not complete within 45 seconds. Common causes: the peer proxy was not started, a firewall blocked TCP port 46000, or a very slow KEM (Classic McEliece at L5) combined with high network latency. The suite record will show `handshake_success = false` and `handshake_failure_reason` will contain the specific error (e.g. `"timeout"`, `"connection_refused"`).

AEAD authentication failure. A decrypted packet failed tag verification. This should **never** occur in normal operation—it indicates either key mismatch (a bug), data corruption, or active tampering. The proxy silently drops the packet and increments `drop_auth` in Category G.

Proxy crash. The proxy subprocess exited unexpectedly (non-zero return code). All data-plane metrics after the crash point are absent. `ManagedProcess` ensures the scheduler detects this and marks the suite as failed.

MAVProxy crash. In MAVPROXY mode, if `mavproxy_proc` dies during a suite, the scheduler aborts with `shutdown_reason = "error: mavproxy_died"`. The entire benchmark terminates.

GCS unreachable. The TCP control channel times out. The scheduler logs the error and the current suite is marked with `status: "error"`.

12.6.2 Class 2: Expected Absences

These represent `None` or `null` values that are **correct** given the platform or configuration. They are not errors.

No power sensor on Windows. The GCS (Windows) does not have an INA219 sensor or RPi5 hwmon interface. All power/energy fields in Category N are `null` for GCS-side records. The auto-detection in `PowerMonitor` correctly returns `None` when no backend is available.

GCS system metrics pruned. A policy realignment (2026-01-18) removed most GCS-side MAVProxy metrics (Category J) to only two validation fields. Old benchmark runs may have full Category J data; new runs will show `null` for the removed fields. This is by design.

ASCON disabled by configuration. If `CONFIG["ENABLE_ASCON"]` is set to `False`, all ASCON-128a suites are pruned from the registry. They simply do not appear in the results—not as failures, but as absent.

Algorithm unavailable in liboqs build. Runtime probing (Section 10.4) removes suites whose KEM or SIG algorithms are not compiled into the local `liboqs` build. These suites are silently excluded.

12.6.3 Class 3: Structural Unavailability

These represent metrics that are structurally impossible to collect given the benchmark mode or hardware configuration. The fields are present in the schema (not pruned) but are always `null` or marked invalid.

No timestamped MAVLink → latency invalid. One-way latency estimation requires the MAVLink collector to observe `SYSTEM_TIME` messages with a valid `time_boot_ms` field. In SYNTHETIC mode (no flight controller), no such messages exist. The field `one_way_latency_valid` is `false` and `latency_invalid_reason` explains why.

No flight controller → FC telemetry null. Without a Pixhawk, MAVLink messages such as `HEARTBEAT`, `SYS_STATUS`, and `GPS_RAW_INT` are never generated. The MAVLink collector reports zero counts, not failures. Category I fields (heartbeat interval, command-ACK latency) are `null`.

RTT requires bidirectional MAVLink → may be null. Round-trip time measurement requires observing a command and its ACK on the plaintext port. If no commands are sent during the suite interval, `rtt_valid` is `false`.

Clock offset requires Chronos → may be null. If Operation Chronos did not execute (e.g. single-machine testing), `clock_offset_ms` in Category A is `null`. This makes one-way latency corrections unreliable.

12.6.4 Impact on Validation Verdict (Category R)

The validation verdict (Category R) applies three checks to each suite record:

1. **Handshake success**: Was `handshake_success` true?
2. **Non-zero data**: Did at least one data-plane counter (`packets_sent` or `packets_received`) exceed zero?
3. **Duration sanity**: Was `suite_total_duration_ms` positive and within bounds?

Table 12.4: How failure classes affect the validation verdict.

Failure Class	Verdict	Rationale
Real failure (handshake timeout)	FAIL	Check 1 fails
Real failure (proxy crash)	FAIL	Check 2 fails (zero packets)
Expected absence (no power)	PASS	Power is optional; all three checks can still pass
Structural unavailability (no FC)	PASS	Checks evaluate data-plane counters, not MAVLink presence

Key Insight

A suite with verdict PASS is not necessarily *complete*. It means the handshake succeeded, data flowed, and timing was sane. Individual metric fields may still be `null` due to expected absences or structural unavailability. Researchers should check both the verdict **and** the specific fields relevant to their analysis before drawing conclusions.

12.7 Metrics Data Flow: End to End

Here is the complete journey of a single metric—say, `handshake_total_duration_ms`—from hardware to dashboard:

1. The proxy measures handshake duration using `time.perf_counter_ns()` and writes it to `drone_status.json`.
2. The scheduler reads the status file via `read_handshake_status()`.
3. The scheduler calls `policy.record_handshake_metrics(metrics)`, which stores the value in the `SuiteMetrics` dataclass.
4. The scheduler calls `aggregator.record_handshake_end(success=True)`, which stores the duration in the `HandshakeMetrics` (Category D) of the current `ComprehensiveSuiteM`
5. The RobustLogger appends the handshake timing to `metrics_drone.jsonl` (incremental).
6. At suite end, `aggregator.finalize_suite()` produces the complete 18-category object.
7. `aggregator.save_suite_metrics()` writes it to `{timestamp}_{suite_id}_drone.json`.
8. At benchmark end, `BenchmarkPolicy._save_results()` writes the summary JSON and CSV.
9. The user copies the JSON files to the dashboard's `logs/benchmarks/runs/` directory.
10. The dashboard's `MetricsStore` loads the JSON, parses it via `ComprehensiveSuiteMetrics.fro` and serves it through the API.
11. The React frontend fetches the API and displays the value in a chart.

12.8 Persistence Formats

The system produces several file formats:

12.9 Chapter Summary

- The metrics pipeline follows a three-stage architecture: **collectors** → **aggregator** → **persistence**.
- An 18-category schema (A–R) with $\sim$ 160 typed fields ensures comprehensive, consistent data across GCS and drone.
- Five collectors (environment, system, power, network, MAVLink) read diverse data sources from git hashes to I^2C current sensors.
- The **INA219 power monitor** samples at up to 1,100 Hz and computes energy via trapezoidal integration.

Table 12.5: Output file formats

Format	Producer	Content
`.json`	MetricsAggregator	Full 18-category ComprehensiveSuiteMetrics, one per suite
`.jsonl`	RobustLogger	Incremental events and metrics (one JSON object per line)
`.jsonl`	Scheduler	Per-suite handshake results (drone + GCS)
`.json`	BenchmarkPolicy	Summary with all suite metrics
`.csv`	BenchmarkPolicy	Tabular summary (one row per suite)
`.csv`	PowerMonitor	Raw power samples (timestamp, V, A, W)
`.json`	Scheduler	Final run summary with statistics

- The **MAVLink collector** tracks 40+ metrics including one-way latency, RTT, sequence integrity, and flight controller telemetry.
- The **MetricsAggregator** merges all collectors into a single typed object, with cross-side merging of GCS data via the TCP control channel.
- The **RobustLogger** provides crash-resilient append-mode logging with fsync, retries, and cross-platform file locking.
- Output files span JSON, JSONL, and CSV formats, feeding both offline analysis tools and the real-time dashboard.

Chapter 13

The Forensic Dashboard

The benchmark scheduler (Chapter 11) and metrics pipeline (Chapter 12) produce hundreds of JSON files—one per suite, per run, per role. To make this data *useful*, researchers need a visual tool that loads, cross-references, and charts the results. This chapter describes the **Forensic Dashboard**: a web application that turns raw benchmark data into interactive charts, tables, heatmaps, and anomaly reports.

Key Insight

The dashboard is a *forensic* tool, not a real-time monitor. It loads pre-recorded benchmark runs, not live data. Its motto (displayed in the UI) captures this philosophy: "No smoothing. No causal inference. Raw observational forensic data."

13.1 Technology Stack

Table 13.1: Dashboard technology stack

Layer	Technology	Purpose
Backend	Python / FastAPI	REST API, data loading, anomaly detection
Models	Pydantic v2	Typed request/response schemas
Frontend	React 18 + TypeScript	Interactive UI components
State	Zustand	Lightweight global state management
Charts	Recharts	Bar, scatter, pie, and line charts
Styling	Tailwind CSS	Utility-first CSS framework
Build	Vite	Fast development server and bundler

Analogy

If the benchmark scheduler is the robot lab technician that runs experiments, the dashboard is the research notebook where you spread out the results, highlight anomalies, and compare measurements side by side.

13.2 Backend Architecture

The backend is a FastAPI application (version 3.0) with CORS support for local development. It serves data through a RESTful API consumed by the React frontend.

13.2.1 Data Loading: The MetricsStore

The `MetricsStore` class (in `dashboard/backend/ingest.py`) loads benchmark data from exactly three scenario folders:

Table 13.2: Scenario folder mapping

Folder	Run Type	Description
`runs/no-ddos/`	`no_ddos`	Baseline: clean network
`runs/ddos-xgboost/`	`ddos_xgboost`	DDoS with XGBoost detection
`runs/ddos-txt/`	`ddos_txt`	DDoS with text-based rules

Each folder contains per-suite JSON files (the output of the `MetricsAggregator`). The store:

1. Scans each scenario folder for `*.json` files.
2. Parses filenames to extract suite ID, run ID, and role (drone or GCS).
3. Loads each file into a `ComprehensiveSuiteMetrics` Pydantic model (v2, mirroring the 18-category schema).
4. Merges time-adjacent drone and GCS records into a single consolidated suite entry.
5. Indexes entries by composite key `{run_id}:{suite_id}`.
6. Builds per-run summaries (`RunSummary`) with suite counts, timestamps, and hostnames.

Design Decision

Only these three scenario folders feed the dashboard. Old runs, broken logs, and intermediate files are ignored. This strict scoping prevents stale or malformed data from polluting the analysis.

13.2.2 Pydantic Models

The file `dashboard/backend/models.py` defines Pydantic v2 models that mirror the 18-category schema:

- 18 nested model classes (one per category A–R), each with typed `Optional` fields matching the dataclass schema.
- `ComprehensiveSuiteMetrics`: the top-level model containing all 18 categories plus an `ingest_status` field for provenance tracking.
- `SuiteSummary`: a lightweight projection with key fields (suite ID, KEM, SIG, AEAD, handshake duration, power, pass/fail).
- `RunSummary`: per-run metadata (ID, start time, suite count, hostnames).

13.2.3 API Endpoints

Table 13.3 lists the principal API endpoints.

Table 13.3: Dashboard API endpoints

Endpoint	Method	Purpose
`/api/suites`	GET	List suites with optional filters
`/api/suite/{key}`	GET	Full metrics for one suite
`/api/runs`	GET	List all benchmark runs
`/api/filters`	GET	Available filter values (KEM, SIG, AEAD, level)
`/api/anomalies`	GET	Anomaly detection results
`/api/settings`	GET	Dashboard settings (labels, active runs, thresholds)
`/api/settings/...`	POST	Update labels, active runs, or thresholds
`/api/multi-run/compare`	GET	Cross-run comparison for one suite
`/api/multi-run/overview`	GET	Aggregated KPIs per active run
`/api/metric-semantics`	GET	Per-field provenance and type info

13.2.4 Settings Store

Dashboard settings (run labels, active run selection, anomaly thresholds) are persisted in a local JSON file. The `SettingsStore` class provides atomic read/write with defaults. Settings include:

- **Run labels**: Human-readable names and scenario types for each run ID.

- **Active runs**: Up to 3 runs selected for multi-run comparison.
- **Anomaly thresholds**: Warning/critical thresholds for handshake duration (ms), packet loss ratio, power deviation (%), and energy deviation (%).

13.3 Frontend Architecture

The frontend is a single-page React application built with Vite and TypeScript.

13.3.1 Application Layout

The `App.tsx` component defines a fixed left sidebar (navigation) and a scrollable main content area. The sidebar groups 12 routes into four categories:

Table 13.4: Dashboard navigation structure

Group	Page	Route
Overview	Dashboard	`/`
Analysis	Suite Explorer	`/suites`
	Bucket Comparison	`/bucket`
	Suite Comparison	`/compare`
	Multi-Run Compare	`/multi-run`
Metrics	Power & Energy	`/power`
	Latency & Transport	`/latency`
	Security Impact	`/security`
	Integrity Monitor	`/integrity`
Config	Metric Semantics	`/semantics`
	Settings	`/settings`

13.3.2 State Management: Zustand Store

All application state lives in a single Zustand store (`store.ts`). Key state slices include:

Data Suites list, runs list, selected suite(s), comparison data.

Settings Dashboard configuration, active runs, anomaly thresholds.

Anomalies Detected anomalies scoped to the latest run.

Filters KEM family, SIG family, AEAD algorithm, NIST level, run ID.

UI Loading indicator, error messages.

The store exposes 18 actions that fetch data from the backend API, update local state, and push settings changes back to the server.

13.3.3 TypeScript Type System

The file `types.ts` mirrors the backend's 18-category schema as TypeScript interfaces. Each Pydantic model maps to a corresponding interface (e.g. `RunContextMetrics`, `CryptoPrimitiveBreakdown`, `PowerEnergyMetrics`), ensuring type safety from the API boundary all the way to the chart components.

Additional utility types include `AnomalyItem`, `SuiteFlag`, `DashboardSettings`, and a `RunType` enum with three values (`no_ddos`, `ddos_xgboost`, `ddos_txt`).

13.4 Dashboard Pages

13.4.1 Overview (Dashboard)

The landing page provides an executive summary:

- **9 KPI cards**: Total suites, pass rate, average handshake (with P95 and σ), average power, anomaly count, fastest/slowest suite, average goodput, total energy.
- **3 pie charts**: Distribution of suites by NIST level, KEM family, and SIG family.
- **Scatter chart**: Handshake duration vs. energy, colour-coded by NIST level—revealing the cost/security trade-off at a glance.
- **Bar charts**: Average handshake, power, goodput, and packet loss grouped by KEM family.
- **Scenario status banner**: Shows which of the three scenario folders are present, with run and suite counts.
- **Multi-run comparison cards**: Per-run KPIs for side-by-side comparison across scenarios.
- **Benchmark runs table**: All runs with metadata (ID, scenario, start time, hosts, git commit).

13.4.2 Suite Explorer

A filterable, heatmap-coloured table of all cipher suites:

- Five dropdown filters (run, KEM, SIG, AEAD, NIST level).
- Each row shows suite ID, algorithms, handshake timing, power, energy, pass/fail status, and anomaly indicator.
- Numeric cells use a green-to-red gradient based on the value's position within the column's range.
- Clicking a suite navigates to the detail page.

13.4.3 Suite Detail

A comprehensive deep-dive into a single suite's full metric inventory:

- Metric cards organised by schema section (A, B, D, G, H, K, N, O, P, F, R).
- Each value carries a **reliability badge** (VERIFIED, CONDITIONAL, DEPRECATED, MISSING) and a **consistency classification** (CONSISTENT, BROKEN, MISLEADING).
- A full metric inventory table showing every key/value with status, source (drone/GCS), and origin function.
- Expandable raw JSON payloads for the drone record, GCS record, and GCS validation JSONL.

13.4.4 Latency Analysis

Deep dive into transport-layer timing:

- 6 KPI cards: average RTT, jitter, one-way latency, handshake, goodput, packet loss (each with P95).
- Per-NIST-level KPI breakdown.
- Grouped bar charts (handshake + RTT + jitter by KEM family; by NIST level).
- Scatter chart: handshake vs. RTT with bubble size $\propto$ power.
- Top-20 suites table with 13 columns.

13.4.5 Security Impact

Anomaly detection dashboard:

- 4 KPI cards: critical anomalies, warnings, clean suites, total suites.
- Horizontal bar chart: anomaly count by metric name.
- Stacked bar chart: critical vs. warning by KEM family.
- Flagged-suites table with severity badges and per-flag details.

The security impact page is designed to highlight which suites behave anomalously under DDoS conditions compared to the baseline run.

13.4.6 Power Analysis

Power and energy visualisation:

- 4 KPI cards: average power (W), total energy (J), suites with power data, sensor type.
- Bar chart: energy by KEM family.
- Scatter chart: power vs. handshake duration (bubble size $\propto$ energy).
- Top-10 energy consumers table.

13.4.7 Comparison View

Side-by-side comparison of two selected suites:

- Two dropdown selectors (Suite A as baseline, Suite B).
- Horizontal bar chart comparing 6 metrics (handshake, goodput, packet loss, power, energy, CPU).
- Detailed comparison table with 8 rows of key metrics.

This page answers questions like "How does ML-KEM-512 with AES-GCM compare to ML-KEM-1024 with ChaCha20 in handshake time and power?"

13.4.8 Multi-Run Comparison

Compares the *same* suite across up to 3 benchmark runs (e.g. baseline vs. DDoS scenarios):

- Suite selector dropdown.
- Per-run metric cards showing the suite's performance under each scenario.
- Highlights differences caused by network conditions rather than cryptographic choice.

13.4.9 Integrity Monitor

Client-side data quality checks:

- 4 KPI cards: clean suites, high/medium/low severity issues.
- Integrity checks:

  High Benchmark marked FAIL, handshake did not succeed.

  Medium Missing power data, handshake > 10 seconds.

- Issues table with severity badges and links to suite detail.

13.4.10 Metric Semantics

A reference page documenting every metric field in the schema:

- Text filter by key name.
- Table with 8 columns: key, category, origin side (drone/GCS), authoritative side, nullable, zero-valid, legacy flag, observed types.

This page serves as the developer's dictionary for interpreting JSON fields.

13.4.11 Settings

Configuration interface:

- Benchmark run checkboxes (activate up to 3 for multi-run comparison).
- Per-run label and type assignment.
- Scenario folder status display.
- Anomaly threshold editors (6 numeric inputs).

13.5 Data Flow: Backend to Frontend

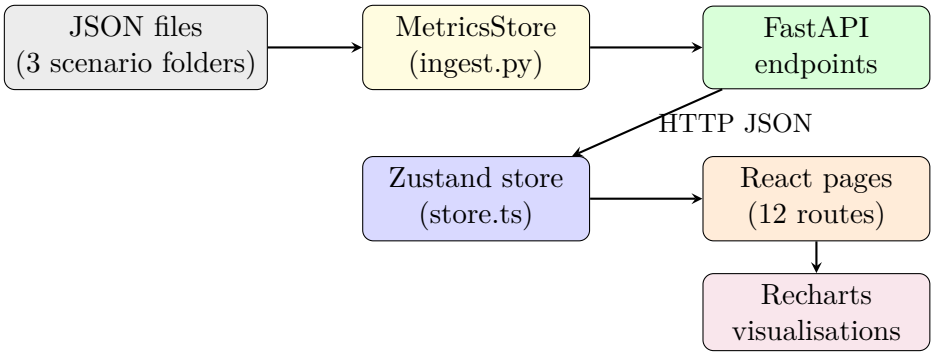


Figure 13.1: Data flow from benchmark output files through the backend API, Zustand state management, React pages, and into Recharts visualisations.

13.6 Interpreting Dashboard Data

The dashboard presents raw observational data. Misreading the data is easy; this section addresses the most common pitfalls.

13.6.1 Null Is Not Zero

Many metric fields are `Optional` (nullable). A `null` value means "not measured" or "not applicable," **not** "the measured value was zero." For example:

`power_avg_watts = null` means no power sensor was available (e.g. GCS on Windows), not that the system consumed zero watts.

`one_way_latency_avg_ms = null` means the latency estimation was invalid (no `SYSTEM_TIME` messages), not that latency was zero.

`rekey_attempts = null` means no rekey was attempted during this suite, which is normal for single-handshake benchmark intervals.

The dashboard's suite detail page shows a **reliability badge** next to each value (VERIFIED, CONDITIONAL, DEPRECATED, MISSING). Always check the badge before interpreting a numeric value.

13.6.2 Understanding FAIL Verdicts

A suite with a FAIL verdict does not necessarily indicate a system bug. Common legitimate reasons include:

1. **Algorithm unavailable**: The KEM or SIG algorithm was not compiled into the platform's `liboqs` build. The handshake fails immediately.
2. **Handshake timeout**: Very heavy algorithms (Classic McEliece at L5, 1.3 MB public key) may exceed the 45-second timeout on slow or congested links.
3. **Transient network failure**: A WiFi dropout during the handshake TCP exchange causes a connection reset.

Key Insight

A handful of FAIL verdicts in a 72-suite run is normal, especially for Classic McEliece suites on constrained hardware. If **all** suites fail, check basic connectivity (can the machines ping each other?) and key deployment (are the signing keys and PSK correctly installed?).

13.6.3 Comparing Across Scenarios

The multi-run comparison page shows the same suite's metrics under different network conditions (baseline vs. DDoS). When comparing:

- **Handshake duration** should be nearly identical across scenarios because the handshake uses TCP (reliable delivery). A large difference suggests the DDoS traffic overwhelmed the WiFi adapter.
- **Packet loss ratio** is expected to increase under DDoS. A ratio below 1% under DDoS suggests the attack was ineffective or improperly configured.
- **Power measurements** across scenarios are only comparable if the hardware setup (INA219 wiring, cooling, ambient temperature) is identical between runs.
- **Anomaly flags** are always computed relative to the *current run's* population statistics. A suite flagged as anomalous in the baseline run may be normal in a DDoS run (where higher latency is expected).

13.6.4 When Data Is Missing

If entire categories appear empty in the suite detail page:

Category N (Power) all null: The benchmark was run on a platform without power sensing. This is normal for GCS-side records and for drone runs without INA219 hardware.

Category I (MAVLink Drone) all null: The benchmark was run in SYNTHETIC mode (no Pixhawk). MAVLink metrics require a real flight controller.

Category H (Latency) marked invalid: Check `latency_invalid_reason` in the raw JSON. Common reasons: no `SYSTEM_TIME` messages, insufficient samples, or clock sync failure.

Category J (MAVLink GCS) mostly null: This is by design since the 2026-01-18 policy realignment (only two validation fields are retained).

13.6.5 Heatmap Colour Scale

The Suite Explorer table colours numeric cells on a **per-column** green-to-red gradient. The colour indicates a value's position within its column's observed range, **not** an absolute quality judgement. In a run where all suites have fast handshakes (e.g. ML-KEM only), the "slowest" ML-KEM suite appears red even though its absolute time may be under 20 ms.

13.7 Anomaly Detection

The dashboard implements a threshold-based anomaly detection system. For each suite, it compares key metrics against configurable warning and critical thresholds:

- **Handshake duration**: Warning if $> T_w$ ms, critical if $> T_c$ ms (defaults: 5000 ms / 30000 ms).
- **Packet loss ratio**: Warning if $> L_w$, critical if $> L_c$ (defaults: 0.01 / 0.05).
- **Power deviation**: Warning if the suite's power deviates from the population mean by more than D_p %.
- **Energy deviation**: Warning if energy deviates by more than D_e %.

Anomalies are computed server-side (scoped to the latest or selected run) and returned as a list of `AnomalyItem` objects with severity, metric name, observed value, threshold, and suite ID. The Security Impact page visualises these as bar charts and a flagged-suites table.

13.8 Chapter Summary

- The **Forensic Dashboard** is a FastAPI + React web application that visualises post-hoc benchmark data.
- The backend loads JSON files from 3 scenario folders (baseline, DDoS-XGBoost, DDoS-text), indexes them by run and suite, and serves them through a REST API.
- Pydantic v2 models mirror the 18-category metrics schema, ensuring type safety from disk to API.
- The React frontend uses Zustand for state, TypeScript for type safety, Recharts for visualisation, and Tailwind CSS for styling.

- 12 pages cover the full analysis workflow: executive overview, suite exploration, latency deep-dive, power analysis, security anomalies, data integrity, cross-suite comparison, multi-run comparison, and metric reference documentation.
- Threshold-based **anomaly detection** flags suites with abnormal handshake times, packet loss, or power consumption.

Part V

Engineering and Reflection

Chapter 14

Engineering Trade-Offs and Security Analysis

Building a system is about making choices under constraints. Every design decision in the PQC Secure MAVLink Tunnel involves a trade-off—between security and performance, between generality and simplicity, between correctness and speed. This chapter examines these trade-offs honestly, discusses the system's limitations, and analyses its security properties.

14.1 Performance Trade-Offs

14.1.1 NIST Level vs. Latency

The most fundamental trade-off is between security level and handshake latency. Higher NIST levels use larger keys and more complex operations:

Table 14.1: Representative handshake times by NIST level (Raspberry Pi 5)

NIST Level	ML-KEM + ML-DSA	HQC + Falcon	McEliece + SPHINCS+
L1	$\sim 15\,\text{ms}$	$\sim 50\,\text{ms}$	$\sim 2{,}000\,\text{ms}$
L3	$\sim 25\,\text{ms}$	$\sim 100\,\text{ms}$	$\sim 15{,}000\,\text{ms}$
L5	$\sim 40\,\text{ms}$	$\sim 200\,\text{ms}$	$\sim 45{,}000\,\text{ms}$

Key Insight

For a drone that rekeys every 110 seconds, even a 45-second handshake means the tunnel is "dark" for 41 % of the cycle. This is why the suite registry provides 72 options: researchers can find the sweet spot for their specific mission.

14.1.2 KEM Family Characteristics

ML-KEM (CRYSTALS-Kyber) Fastest overall. Small keys ($\sim$ 800–1,568 bytes public key). Lattice-based. Best choice for resource-constrained devices.

HQC Code-based. Slower than ML-KEM but offers algorithm diversity (different mathematical hardness assumption). Moderate key sizes ($\sim$ 2,000–8,000 bytes).

Classic McEliece Extremely large public keys (~ 260,000–1,300,000 bytes at L1–L5). Very fast decapsulation but key generation and encapsulation are slow. The large keys dominate handshake time due to network transfer.

14.1.3 AEAD Algorithm Comparison

Table 14.2: AEAD algorithm trade-offs

Algorithm	Hardware Accel	Tag Size	Notes
AES-256-GCM	AES-NI (x86), ARMv8-CE	16 B	Fastest with hardware support
ChaCha20-Poly1305	None needed	16 B	Constant-time; good on ARM without
ASCON-128a	None	16 B	Lightweight; designed for constrained

The Raspberry Pi 5 (Cortex-A76) has ARMv8 Crypto Extensions, making AES-GCM competitive. On older ARM devices without these extensions, ChaCha20 would be faster. ASCON is the NIST Lightweight standard and is designed for very small processors (8-bit, 32-bit microcontrollers), so it is not the fastest on a 64-bit ARM but provides algorithm diversity.

14.1.4 Power vs. Security

Post-quantum cryptographic operations consume measurably more power than their classical equivalents. The INA219 power monitor (Section 12.3) captures this precisely. Key observations:

- Power consumption during handshake is dominated by CPU-intensive KEM and SIG operations.
- ML-KEM handshakes on the Pi 5 consume ~ 5–8 W peak (vs. ~ 3 W idle).
- Classic McEliece key generation can sustain peak power for tens of seconds, significantly impacting battery-powered drones.
- The `energy_per_handshake_j` metric directly quantifies the cost of security.

14.2 Design Decisions and Their Rationale

14.2.1 Bump-in-the-Wire Architecture

Design Decision

The proxy operates as a transparent bump-in-the-wire: it does not modify MAVLink payloads, does not parse MAVLink semantics, and does not require changes to the flight controller firmware or the GCS application. This maximises compatibility but means the proxy cannot perform MAVLink-aware optimisations (e.g. prioritising heartbeat messages).

14.2.2 UDP over TCP

Design Decision

The encrypted tunnel uses UDP, not TCP. MAVLink is inherently a best-effort protocol—it expects occasional packet loss and compensates with redundant streams. TCP's retransmission and head-of-line blocking would add latency spikes that are worse for real-time control than losing a single packet. The AEAD layer provides integrity guarantees that TCP's checksums would otherwise provide.

14.2.3 Selectors over asyncio

Design Decision

The proxy uses `selectors.DefaultSelector` (which maps to `epoll` on Linux) rather than Python's `asyncio` framework. Selectors provide microsecond-level event notification with minimal overhead. `asyncio` would add coroutine scheduling overhead and make the event loop harder to reason about in a security-critical context.

14.2.4 No Key Caching

Design Decision

Every suite switch performs a fresh KEM key generation, encapsulation, and signature. There is no session resumption or key caching. This is a deliberate choice for the benchmark: every measurement captures the full cost of the PQC handshake. A production system might cache session keys for faster resumption.

14.2.5 Two-Phase Commit for Policy

Design Decision

The `evaluate()` / `confirm_advance()` split (Section 11.4.2) adds complexity but prevents a dangerous class of bugs: metrics being attributed to the wrong suite when the index advances before finalisation completes.

14.3 Security Analysis

14.3.1 Formal Threat Model

This section specifies the adversary model, trust assumptions, security goals, and explicit non-goals that bound the system's security claims.

14.3.1.1 Adversary Capabilities

The system is designed to resist the following adversary classes:

Passive network adversary (Eavesdropper). Can observe all packets on the wireless LAN between drone and GCS. Can record ciphertext, headers, timing, and packet sizes. Cannot modify or inject traffic.

Active network adversary (Dolev–Yao style). In addition to passive capabilities, can inject, modify, replay, delay, reorder, and selectively drop packets on the link between drone and GCS. This is the standard Dolev–Yao network attacker model [1]: the adversary controls the communication channel but not the endpoints.

Future quantum adversary ("harvest now, decrypt later"). Records ciphertext today and stores it until a cryptographically relevant quantum computer (CRQC) exists. At that point, the the adversary applies Shor's algorithm [5] to any classical public-key material and Grover's algorithm [2] to symmetric keys. The system's use of post-quantum KEMs and signatures is specifically motivated by this adversary.

14.3.1.2 Trust Assumptions

The security analysis depends on the following assumptions. If any assumption is violated, the corresponding security property may not hold.

1. **Pre-deployed identity keys are authentic.** Signing key pairs (GCS) and pre-shared keys (drone PSK) are generated offline, transported physically or via a pre-authenticated channel, and stored on the local filesystem. The system does not establish key authenticity—it assumes it.

2. **Endpoint hardware is not compromised.** The Raspberry Pi (drone) and the GCS laptop are assumed to be free of hardware implants, firmware backdoors, and rootkits. Physical tamper protection is out of scope.

3. **Localhost is a trust boundary.** Plaintext MAVLink traffic travels exclusively over the loopback interface (`127.0.0.1`). No unprivileged remote process can read localhost traffic. If an attacker gains local code execution on either endpoint, the plaintext data plane is exposed.

4. **The operating system provides process isolation.** Memory isolation between the proxy process and other processes is assumed to be intact (no shared-memory side channels, no `/proc/{pid}/mem` access by unprivileged users).

5. **Cryptographic primitives are correct.** The implementations of ML-KEM, ML-DSA, Falcon, SPHINCS+, HQC, Classic McEliece (via `liboqs`), AES-GCM, ChaCha20-Poly1305 (via the Python `cryptography` library), and ASCON-128a are assumed to be functionally correct and free of catastrophic implementation bugs. No formal verification has been performed on these libraries.

6. **Monotonic clocks are monotonic.** `time.monotonic()` and `time.perf_counter_ns()` are assumed to never go backwards. Policy timing (hysteresis, intervals) depends on this property.

14.3.1.3 Security Goals

Given the above adversary model and trust assumptions, the system aims to provide:

1. **Data-plane confidentiality**: An eavesdropper learns nothing about the plaintext content of MAVLink datagrams (beyond their encrypted length and timing).
2. **Data-plane integrity**: Any modification to a ciphertext packet (header or payload) is detected and the packet is dropped.
3. **GCS authentication**: The drone can verify that it is communicating with the holder of the GCS signing key.
4. **Drone identity binding**: The GCS can verify that the drone possesses the pre-shared key (HMAC-PSK).
5. **Replay protection**: Retransmission of previously captured valid packets is detected and rejected.
6. **Session isolation**: Packets from a previous handshake session are rejected (via session ID and epoch checks).
7. **Forward secrecy**: Compromise of long-term signing keys does not reveal past session keys (KEM keypairs are ephemeral).
8. **Post-quantum resistance**: All of the above properties hold against a future quantum adversary.

14.3.1.4 Explicit Non-Goals

The following are **not** security goals of this system. They are listed explicitly to prevent misinterpretation of the system's claims.

1. **Denial-of-service resilience.** The system does not guarantee availability. An active network attacker can drop all packets, severing the link. Rate limiting (Section 9.5) provides limited protection against handshake flooding, but the data plane has no anti-DoS mechanism beyond the inherent tolerance of UDP-based MAVLink.
2. **Traffic analysis resistance.** Packet lengths, timing, and rates are observable by a passive adversary. The system does not pad packets, add dummy traffic, or otherwise obscure traffic patterns. An adversary can infer activity levels and message types from packet sizes.
3. **Insider threat protection.** An attacker with local code execution on either endpoint can read plaintext MAVLink traffic from the loopback interface, extract key material from process memory, or manipulate the proxy.
4. **Physical tamper resistance.** No hardware security modules (HSMs), trusted platform modules (TPMs), or tamper-evident enclosures are used. Keys reside as files on disk.

5. **Key revocation and PKI.** There is no certificate authority, no certificate revocation mechanism, and no key expiry enforcement. Revoking a compromised key requires manual re-provisioning.

6. **Multi-party or multi-drone operation.** The system secures a single point-to-point link between one drone and one GCS. Mesh networks, relay topologies, and multi-vehicle swarms are out of scope.

7. **Constant-time guarantees.** The Python runtime does not provide constant-time execution. Timing side channels in KEM/SIG operations (via `liboqs`) or in branch-dependent Python code are not mitigated.

14.3.1.5 Scope of the Data-Plane Security Boundary

Figure 14.1 illustrates the trust boundaries.

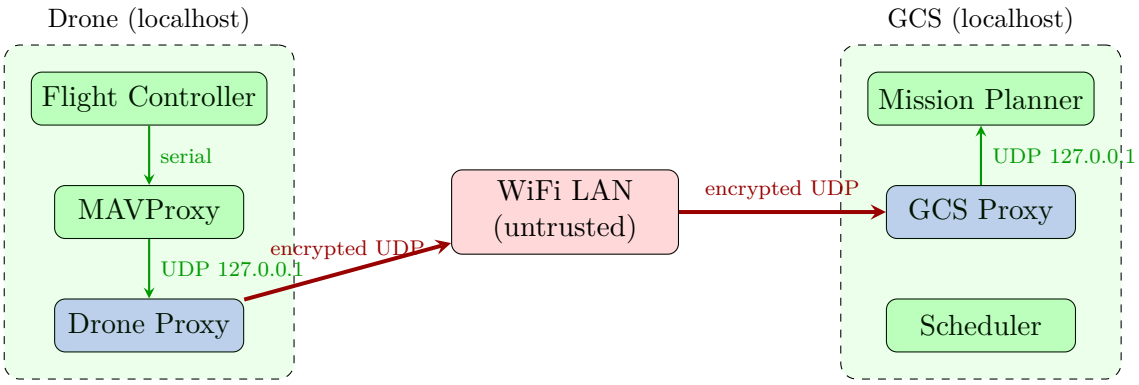


Figure 14.1: Trust boundary diagram. Green regions are trusted (localhost). The red region (WiFi LAN) is the untrusted channel secured by the PQC tunnel. The bold red arrows represent AEAD-encrypted traffic.

The trust boundary has five key properties:

1. **Plaintext isolation**: All plaintext traffic is bound to `127.0.0.1`. No plaintext datagram ever crosses a network interface.

2. **Encrypted-plane authentication**: Every packet crossing the untrusted network carries a 16-byte AEAD authentication tag over both the ciphertext and the 22-byte header (as AAD).

3. **Session binding**: The 8-byte session ID in every packet header cryptographically binds data-plane traffic to a specific handshake instance.

4. **Rate-limited handshake**: A token-bucket rate limiter (5 tokens, 1 refill/second) on the GCS handshake server prevents rapid reconnection attempts.

5. **Strict peer matching**: When enabled (`CONFIG["STRICT_UDP_PEER_MATCH"]`), the proxy rejects encrypted packets from any source IP/port other than the established peer.

Security Note

The **TCP control channel** (port 48080) used by the benchmark scheduler is **outside** the security boundary. It carries plaintext JSON commands (start proxy, stop suite, clock sync) and is neither encrypted nor authenticated. This is acceptable for a lab benchmark on a controlled LAN but would require TLS or HMAC authentication in any deployment scenario. Similarly, Operation Chronos (clock synchronisation) runs over this same unauthenticated channel; a network attacker could manipulate clock offsets to influence time-based scheduling policy decisions.

14.3.2 Cryptographic Security Properties

Confidentiality Provided by AEAD encryption (AES-256-GCM, ChaCha20-Poly1305, or ASCON-128a) with keys derived from a PQC KEM shared secret via HKDF-SHA256.

Integrity AEAD tags (16 bytes) detect any modification to the ciphertext or header (which is used as AAD).

Authentication The GCS signs its ServerHello with a PQC digital signature (ML-DSA, Falcon, or SPHINCS+). The drone verifies the signature using the pre-deployed public key. The drone authenticates to the GCS via HMAC-SHA256 over the shared secret, binding the PSK identity to the handshake.

Replay protection An 8-byte sequence number and a 64-entry sliding bitmap window (Section 8.4) prevent replay attacks. Epoch numbers prevent cross-session replay.

Forward secrecy Each handshake generates fresh KEM keypairs. Compromise of the long-term signing key does not reveal past session keys (assuming the KEM shared secret was properly destroyed).

Post-quantum security The KEM algorithms (ML-KEM, HQC, Classic McEliece) are designed to resist quantum attacks. The signature algorithms (ML-DSA, Falcon, SPHINCS+) are post-quantum secure. An attacker recording today's handshake cannot extract the shared secret even with a future quantum computer.

14.3.3 Known Limitations and Residual Risks

Security Note

The following are **known limitations**, not bugs. They represent conscious design choices for a research benchmark system.

1. **No mutual authentication in KEM**: Only the GCS is authenticated via digital signature. The drone authenticates via HMAC-SHA256 over the PSK, which provides weaker identity binding than a full mutual signature exchange.

2. **Plaintext control channel**: The TCP control channel (port 48080) is unencrypted. An attacker could inject fake commands (e.g. force a suite

switch, cause a shutdown). Acceptable in a lab LAN; would need TLS or HMAC in production.

3. **Clock sync is unauthenticated**: An attacker could manipulate Chronos timestamps to skew the clock offset, affecting time-based policy decisions.
4. **No certificate infrastructure**: Signing keys are bare files on disk, not wrapped in certificates with expiry dates or revocation lists.
5. **No Perfect Forward Secrecy for the PSK**: The drone PSK (pre-shared key) is static. If compromised, an attacker could impersonate the drone in future handshakes (though not decrypt past sessions, since KEM keypairs are ephemeral).
6. **Sequence number overflow**: The 8-byte sequence number supports 2^{64} packets. At 1000 packets/second, overflow would take $\sim$ 585 million years. The system raises an exception before overflow as a defence-in-depth measure.
7. **Side-channel attacks**: The PQC implementations from `liboqs` are not guaranteed to be constant-time on all platforms. A co-located attacker with cache-timing access could potentially extract key material.

14.4 Cross-Platform Considerations

The system runs on two very different platforms:

Table 14.3: Platform comparison

Aspect	Drone (Pi 5)	GCS (Windows)
OS	Raspberry Pi OS (Debian)	Windows 10/11
CPU	ARM Cortex-A76, 4 cores	x86-64, 8+ cores
RAM	4–8 GB	16–32 GB
Python	3.11+	3.11+
Power mon	INA219 / RPi5 hwmon	Not available
Process mgmt	PDEATHSIG via libc	Win32 Job Objects
MAVProxy	Serial to flight controller	UDP from tunnel

Key cross-platform challenges:

- **liboqs availability**: The OQS library must be compiled natively on each platform. ARM and x86 may support different algorithm subsets.
- **Path separators**: All paths use `pathlib.Path` to handle `/` vs. `\` transparently.
- **Signal handling**: `SIGTERM` is not available on Windows in the same way. `ManagedProcess` uses `TerminateProcess` as the Windows equivalent.

- **Console allocation**: MAVProxy's `prompt_toolkit` requires a real Windows console. The GCS server launches MAVProxy with `new_console=True` when GUI is enabled.

14.5 Scalability and Resource Usage

14.5.1 Memory

The proxy's memory footprint is dominated by:

- Socket buffers (4 UDP sockets, kernel-managed).
- AEAD key material (two 32-byte keys + nonce state).
- The replay window bitmap (64 bits = 8 bytes).
- Configuration dictionary ($\sim$ 100 keys, $\sim$ 10 KB).

Total proxy memory is typically under 50 MB.

14.5.2 CPU

During steady-state encrypted forwarding, the main costs are:

- AEAD encryption (one per outbound packet).
- AEAD decryption (one per inbound packet).
- `select()` system call overhead.

At 20 MAVLink packets/second (typical for HEARTBEAT + telemetry), CPU usage is negligible ($< 5\,\%$). During handshake, CPU can spike to 100 % on one core for KEM/SIG operations.

14.5.3 Network Bandwidth

The AEAD framing adds 22 bytes of header + 16 bytes of tag = 38 bytes overhead per packet. For a typical 263-byte MAVLink v2 packet, this is $38/263 \approx 14.4\,\%$ overhead.

14.6 Testing Strategy

The codebase includes multiple test layers:

1. **Unit tests** (`tests/`): Test individual functions (suite registry, AEAD operations, config parsing).
2. **Integration tests** (`test_*.py` at repo root): Test the complete proxy loop (start proxy, perform handshake, send data, verify receipt).
3. **Benchmark tests** (`test_comprehensive_benchmark.py`): Test the full benchmark scheduler with multiple suites.

4. **Validation scripts** (`verify_*.py`): Verify collector output, metrics consistency, and drone/GCS alignment.

5. **The benchmark itself**: A 72-suite benchmark run is the ultimate integration test—if all suites pass, the entire pipeline (handshake, AEAD, proxy, scheduler, metrics, persistence) works end-to-end.

14.7 Reproducibility and Experimental Constraints

Benchmark results presented in this book were collected on specific hardware, software, and environmental conditions. This section documents the factors that affect reproducibility so that readers attempting to replicate or extend the results can set appropriate expectations.

14.7.1 Hardware Dependencies

Raspberry Pi 5 (drone). The benchmark assumes a Pi 5 with a Broadcom BCM2712 (Cortex-A76) SoC. Earlier models (Pi 4, Pi 3) will produce different results because they have different CPU architectures, cache sizes, and memory bandwidth. The Pi 5's AES instruction support (`FEAT_AES`) significantly benefits AES-GCM throughput.

INA219 current sensor. Power metrics require an INA219 breakout board wired to the Pi's I^2C bus and measuring the 5 V supply rail. Without this sensor, all power and energy fields are `null`. The alternative RPi5 hwmon backend reads the on-board PMIC and provides voltage and current but at lower accuracy.

Pixhawk flight controller. In MAVPROXY mode, a Pixhawk (or compatible) connected via USB serial generates real MAVLink traffic. Without it, only SYNTHETIC mode is available, and all MAVLink-specific metrics (heartbeat interval, sequence gaps, command-ACK latency) are structurally unavailable (Section 12.6).

WiFi link characteristics. The LAN between drone and GCS uses consumer WiFi (2.4 GHz or 5 GHz). Latency, jitter, and packet loss are sensitive to: distance, obstacles, channel congestion from neighbouring networks, and the specific WiFi adapter and driver.

14.7.2 Software Dependencies

liboqs version. The set of available KEM and signature algorithms depends on the `liboqs` build. Version upgrades may add or remove algorithms, change default parameters, or introduce performance regressions. The exact version is recorded in Category A (`liboqs_version`).

Python version. The system requires Python 3.11+. Performance of `selectors.select()`, `socket.sendto()`, and `struct.pack()` varies across Python minor versions and builds (e.g. CPython vs. PyPy).

cryptography library version. AES-GCM and ChaCha20-Poly1305 performance depends on whether the underlying OpenSSL build uses hardware acceleration (AES-NI on x86, NEON on ARM).

Operating system kernel. Kernel version affects UDP buffer sizes, scheduler latency, I2C driver behaviour, and hwmon sensor availability. Both the GCS kernel (`kernel_version_gcs`) and drone kernel (`kernel_version_drone`) are recorded in Category A.

14.7.3 Environmental Factors

Thermal throttling. The Pi 5 throttles CPU frequency when the SoC temperature exceeds 80°C. Extended benchmark runs (72 suites ≈ 2 hours) may experience throttling, especially without active cooling. The system records CPU temperature in Category O but does not adjust for throttling.

CPU frequency governor. The Linux `performance` governor locks the CPU at maximum frequency, reducing variance. The default `ondemand` governor allows dynamic scaling, which introduces measurement noise. The governor setting is **not** recorded automatically and should be documented by the experimenter.

Background processes. Other processes on the Pi (system daemons, SSH sessions, log rotation) consume CPU and memory, affecting benchmark results. Category O records system-level resource usage, but individual process interference is not isolated.

Network interference. WiFi interference from co-channel access points, microwave ovens, and Bluetooth devices affects latency and packet loss. For controlled experiments, a dedicated 5 GHz channel with no neighbouring networks is recommended.

14.7.4 What Can and Cannot Be Replicated

Table 14.4: Reproducibility assessment for key result categories.

Result Category	Reproducible?	Constraint
Handshake latency ranking (KEM order)	High	Algorithm rankings are stable across platforms
Absolute handshake times	Medium	Depend on CPU, liboqs version, and thermal state
Power measurements	Low	Require identical INA219 wiring and calibration
MAVLink integrity metrics	Medium	Require same FC firmware, baud rate, and message rates
Data-plane packet loss	Low	Dominated by WiFi conditions, highly variable
Suite pass/fail	High	Algorithmic correctness is deterministic

14.7.5 Reproducibility Checklist

Researchers attempting to replicate results should document:

1. Exact hardware model (Pi revision, INA219 breakout, WiFi adapter).
2. Software versions: Python, liboqs, oqs-python, cryptography library, OS kernel.
3. Git commit hash of the codebase (recorded automatically in Category A).
4. CPU frequency governor setting and cooling configuration.
5. WiFi channel, bandwidth (20/40/80 MHz), and distance between devices.
6. Ambient temperature and whether active cooling is used.
7. Any environment variable overrides (`SUITES_IGNORE_KEMS`, `ENABLE_ASCON`, etc.).
8. The benchmark mode (MAVPROXY or SYNTHETIC) and interval setting.

14.8 Chapter Summary

- The fundamental trade-off is **security level vs. latency**: higher NIST levels and more conservative KEM families (McEliece) incur dramatically higher handshake times.
- Key design decisions (bump-in-wire, UDP, selectors, no key caching, two-phase commit) are justified by the system's requirements.
- The system provides **confidentiality, integrity, authentication, replay protection, forward secrecy, and post-quantum security**.
- Known limitations (unauthenticated control channel, no PKI, static PSK) are acceptable for a lab benchmark and are documented for future hardening.
- Cross-platform operation between Linux/ARM and Windows/x86 introduces challenges in process management, library availability, and console handling.
- Resource usage is modest: under 50 MB RAM, $< 5\,\%$ CPU at steady state, and $\sim 14\,\%$ bandwidth overhead.

Chapter 15

Conclusion and Future Work

15.1 Summary of Contributions

This book has documented a complete, working Post-Quantum Cryptographic (PQC) Secure MAVLink Tunnel system—from theoretical foundations through implementation details to benchmark orchestration and visual analysis. The key contributions are:

1. **A bump-in-the-wire PQC tunnel**: A transparent proxy that encrypts MAVLink traffic between a drone and ground control station using post-quantum algorithms, without modifying the flight controller firmware or GCS application software.

2. **72 cipher suites**: A comprehensive suite registry combining 9 KEM algorithms, 8 signature algorithms, and 3 AEAD ciphers across NIST security levels 1, 3, and 5. This provides researchers with the largest PQC suite matrix applied to drone communications to date.

3. **A telemetry-aware scheduling policy**: A safety-critical state machine that adapts cryptographic strength to real-time conditions (battery, temperature, link quality) with hysteresis, blacklisting, and rate limiting.

4. **An 18-category metrics pipeline**: A structured data collection framework with ~ 160 typed fields covering cryptographic primitives, data-plane counters, MAVLink integrity, power consumption, and system resources.

5. **High-frequency power monitoring**: Direct INA219 register access at up to 1,100 samples/second for precise energy-per-handshake measurements.

6. **A forensic analysis dashboard**: A web-based tool for multi-run comparison, anomaly detection, and visual exploration of benchmark results across 12 interactive pages.

7. **Cross-platform operation**: The entire system runs on a Raspberry Pi 5 (drone) and a Windows laptop (GCS), demonstrating practical deployment on real constrained hardware.

15.2 Key Findings

From the benchmark results (Chapter 19), several findings emerge. Table 15.1 summarises the headline numbers.

Table 15.1: Headline benchmark results (median, $n = 200$, RPi 4 Cortex-A72).

Metric	Best	Worst	F
KEM keygen (ms)	0.082 (ML-KEM-512)	7,066 (McE-8192128)	86,(
SIG sign (ms)	0.641 (Falcon-512)	2,598 (SPHINCS$^+$-192s)	4,(
AEAD encrypt (μs, 64 B)	4.15 (Ascon)	7.28 (AES-GCM)	
Suite handshake (ms)	∼2 (ML-KEM-768+ML-DSA-65)	12,377 (McE-8M+SPHINCS$^+$-256s)	6,(
KEM energy (μJ)	876 (ML-KEM-512 encaps)	27.6 M (McE-8192128 keygen)	31,(
SIG energy (μJ)	741 (Falcon-512 verify)	11.3 M (SPHINCS$^+$-192s sign)	15,(

From these measurements:

1. **ML-KEM dominates**: ML-KEM (CRYSTALS-Kyber) consistently delivers the lowest handshake latency across all NIST levels, making it the clear choice for latency-sensitive drone applications.

2. **Classic McEliece is impractical for real-time rekey**: While offering strong security guarantees based on a different mathematical problem (coding theory), McEliece's enormous public keys (up to 1.3 MB) make handshakes take tens of seconds, far too slow for operational drone rekey intervals.

3. **AEAD choice matters less than KEM choice**: The three AEAD algorithms (AES-GCM, ChaCha20, ASCON) differ by microseconds per packet, while KEM choice affects handshake time by milliseconds to seconds.

4. **Power cost is measurable but manageable**: PQC handshakes on the Pi 5 consume 5–8 W peak, compared to 3 W idle. For a typical 110-second suite interval, the handshake energy is a small fraction of total energy.

5. **Algorithm diversity has value**: HQC and SPHINCS+ provide security from fundamentally different mathematical assumptions (coding theory and hash functions, respectively), hedging against the possibility that lattice-based cryptography is broken.

15.3 Lessons Learned

1. **Metrics must be crash-resilient**: Early versions lost all data when a suite crashed. The RobustLogger's append-mode design (Section 12.5) solved this.

2. **Time domains must not mix**: Using wall-clock time where monotonic time is expected (or vice versa) produced subtly wrong interval calculations. Consistent use of `time.monotonic()` for policy timing eliminated this class of bugs.

3. **Two-phase commit prevents metric mis-attribution**: The evaluate/confirm_advance split (Section 11.4.2) was introduced after discovering that metrics were being recorded against the wrong suite.

4. **Cross-platform process management is harder than expected**: Orphaned proxy processes on both Linux and Windows required platform-specific solutions (PDEATHSIG, Win32 Job Objects).

5. **PQC library maturity varies**: Some algorithms in `liboqs` have performance regressions between versions, and not all algorithms are available on all platforms. Runtime probing (Section 10.4) is essential.

15.4 Future Work

15.4.1 Short-Term Improvements

1. **TLS for the control channel**: Replace the plaintext TCP control channel with mutual TLS or HMAC-authenticated messages.

2. **Certificate-based identity**: Replace bare signing key files with X.509 certificates (or a lightweight equivalent) that include expiry dates and support revocation.

3. **Session resumption**: Cache KEM shared secrets (with a TTL) to enable faster rekey without a full handshake.

4. **Hybrid classical/PQC**: Combine a classical ECDH exchange with a PQC KEM to provide security even if one is broken ("belt and suspenders" approach, as recommended by NIST).

15.4.2 Medium-Term Research Directions

1. **Real flight testing**: Run the benchmark during actual drone flights to measure the impact of vibration, altitude, and RF interference on PQC tunnel performance.

2. **Multi-hop tunnels**: Extend the architecture to support mesh networks where multiple drones relay encrypted traffic.

3. **Machine-learning policy**: Replace the rule-based TelemetryAwarePolicyV2 with a reinforcement-learning agent that optimises suite selection based on historical performance data.

4. **Hardware acceleration**: Evaluate FPGA or GPU acceleration for lattice-based operations (NTT, polynomial multiplication) on the drone side.

15.4.3 Long-Term Vision

1. **Standardisation**: Contribute the handshake protocol and wire format to MAVLink standardisation efforts as a PQC security extension.

2. **Embedded implementation**: Port the critical path (handshake + AEAD) to C/Rust for deployment on microcontroller-based flight controllers (STM32, ESP32).

3. **Formal verification**: Use tools like ProVerif or Tamarin to formally verify the handshake protocol's security properties.

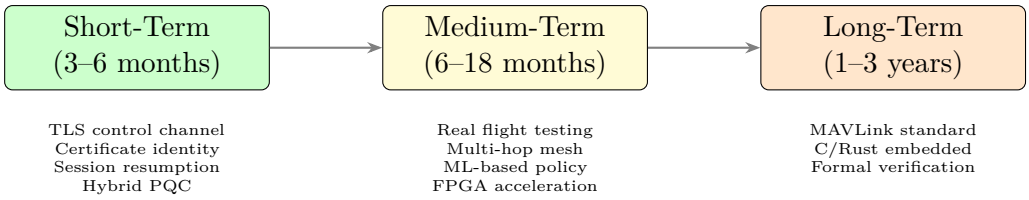


Figure 15.1: Future work roadmap.

15.5 Closing Remarks

The transition to post-quantum cryptography is not a distant future concern—it is an active engineering challenge that must be addressed *now*, before quantum computers become capable of breaking classical encryption. The "harvest now, decrypt later" threat means that drone telemetry recorded today could be decrypted by a future adversary.

This system demonstrates that post-quantum protection for drone communications is *practical* on commodity hardware. A Raspberry Pi 5 running Python can perform ML-KEM-512 handshakes in under 15 milliseconds, encrypt MAVLink packets with negligible overhead, and cycle through 72 cipher suites in a single automated benchmark run.

The code, the metrics, the analysis tools, and this book are all contributions toward a future where unmanned systems communicate securely, even in a post-quantum world.

Part VI

Reference and Deployment

Chapter 16

Software Dependencies and Environment

16.1 Why Dependencies Matter

A software system is not just the code you write—it is also the code you *depend on*. The PQC Secure MAVLink Tunnel relies on approximately 30 external packages (14 Python, 14 JavaScript/TypeScript, and a handful of system-level dependencies) that together provide cryptographic primitives, network protocol parsing, hardware sensor access, process management, data analysis, and a full web-based dashboard.

Understanding these dependencies is essential for three reasons:

1. **Reproducibility.** A benchmark result is only meaningful if another researcher can recreate the exact software environment. A different version of `liboqs` may produce different handshake timings. A different version of `pandas` may parse CSV files differently.

2. **Security.** Every dependency is an attack surface. A vulnerability in `cryptography` or `uvicorn` could compromise the entire system. Knowing what you depend on is the first step toward auditing it.

3. **Maintainability.** When a dependency releases a breaking change, you need to know exactly which parts of your codebase are affected. This chapter provides that map.

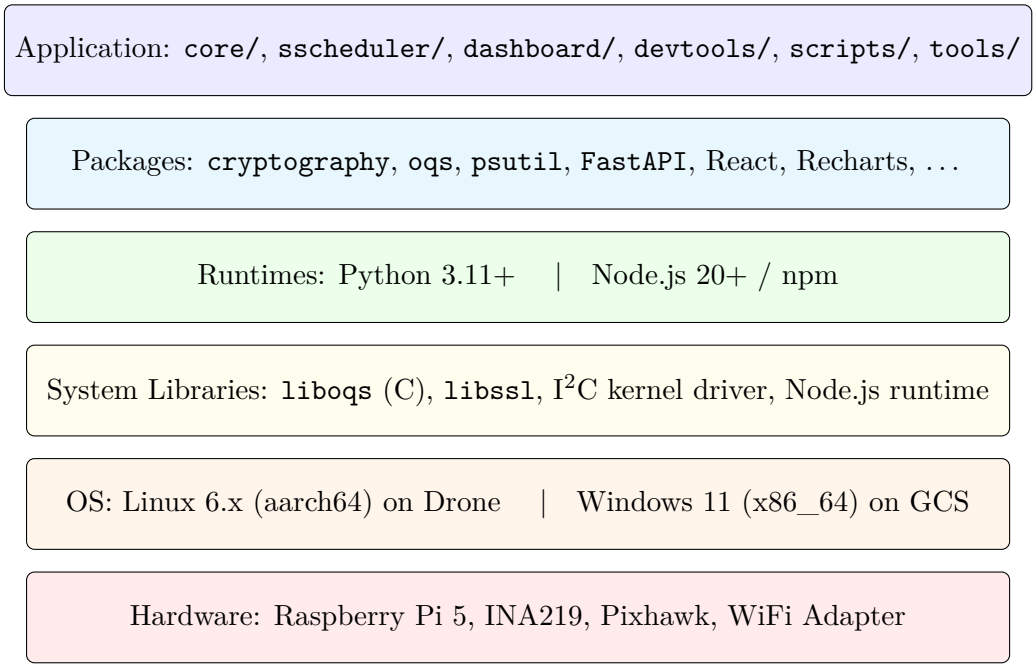
Key Insight

The Dependency Iceberg The codebase you write is the tip of the iceberg. Beneath it are thousands of lines of library code that do the heavy lifting. The `cryptography` package alone contains over 100,000 lines of Python and C code. `liboqs` contains over 300,000 lines of C implementing post-quantum algorithms. Understanding what sits beneath your code is not optional—it is a professional responsibility.

16.2 Dependency Architecture Overview

The system's dependencies form a layered stack. At the bottom are system-level components (the operating system, hardware interfaces, the C compiler toolchain).

Above those sit the Python and JavaScript runtimes. Above those sit the third-party packages. And at the top sits the application code.



Each layer depends on everything below it. A change at any layer can ripple upward. The sections that follow document every component in the "Packages" layer and explain its role in the system.

16.3 Python Backend Dependencies

The Python backend has two classes of dependencies: those declared in a `requirements.txt` file (for the dashboard backend), and those imported directly by the `core/` and `sscheduler/` packages (installed manually or via conda).

16.3.1 The `cryptography` Package

Attribute	Detail
Package name	`cryptography`
PyPI URL	https://pypi.org/project/cryptography/
Version used	≥ 41.0
License	Apache-2.0 / BSD-3-Clause (dual)
Language	Python + Rust + C (via OpenSSL)
Size	$\sim$5 MB wheel

What it provides. The `cryptography` package is the backbone of all *classical* cryptographic operations in the system. It provides:

- **AES-256-GCM** authenticated encryption (`cryptography.hazmat.primitives.ciphers.aead.A`
- **ChaCha20-Poly1305** authenticated encryption (`cryptography.hazmat.primitives.ciphers.a`

- **HKDF-SHA256** key derivation (`cryptography.hazmat.primitives.kdf.hkdf.HKDF`)
- **HMAC-SHA256** message authentication (`cryptography.hazmat.primitives.hmac.HMAC`)
- **SHA-256** hashing (`cryptography.hazmat.primitives.hashes.SHA256`)

Where it is used.

- `core/aead.py` — Creates `AESGCM` and `ChaCha20Poly1305` cipher objects for the data-plane encryption/decryption path.
- `core/handshake.py` — Uses HKDF to derive two directional transport keys from the KEM shared secret. Uses HMAC for drone PSK authentication.
- `core/run.py` — Passes AEAD algorithm tokens to the cipher factory.

Design Decision

Why `cryptography` and not `pycryptodome`? The `cryptography` package delegates to OpenSSL's C implementation, benefiting from hardware AES-NI acceleration on the GCS (Intel CPU) and NEON-accelerated AES on the Raspberry Pi 5 (ARM Cortex-A76). `pycryptodome` also provides AES-GCM, but `cryptography`'s API is simpler, its maintenance record is stronger, and it is the *de facto* standard in the Python ecosystem.

16.3.2 The `oqs-python` Package (liboqs)

Attribute	Detail
Package name	`oqs` (imported as `oqs`; installed as `liboqs-python`)
GitHub	https://github.com/open-quantum-safe/liboqs-python
Underlying C lib	`liboqs` 0.10.x
License	MIT
Install method	conda (`oqs-dev` environment) or from source
Size	∼25 MB (including compiled C library)

What it provides. This is the single most important dependency in the entire system. `oqs-python` provides Python bindings to the Open Quantum Safe `liboqs` C library, which implements:

- **9 KEM algorithms**: ML-KEM-512, ML-KEM-768, ML-KEM-1024, Classic McEliece (348864, 460896, 6688128), HQC-128, HQC-192, HQC-256
- **8 Signature algorithms**: ML-DSA-44, ML-DSA-65, ML-DSA-87, Falcon-512, Falcon-1024, SPHINCS+-SHA2-128f-simple, SPHINCS+-SHA2-192f-simple, SPHINCS+-SHA2-256f-simple

Import patterns used. The codebase uses two distinct import styles, reflecting different `oqs-python` versions:

```python
# Style 1: Modern flat import
import oqs

kem = oqs.KeyEncapsulation("ML-KEM-768")
sig = oqs.Signature("ML-DSA-65")

# Style 2: Legacy nested import (compatibility)
from oqs.oqs import KeyEncapsulation, Signature

kem = KeyEncapsulation("Kyber768")
```

Listing 16.1: OQS import patterns in the codebase.

Where it is used.

- `core/handshake.py` — Creates KEM keypairs (`generate_keypair`), encapsulates shared secrets (`encap_secret`), decapsulates (`decap_secret`), signs challenges (`sign`), and verifies signatures (`verify`).
- `core/suites.py` — Probes which algorithms are available at runtime via `oqs.get_enabled_KEM_me` and `oqs.get_enabled_sig_mechanisms()`.
- `sscheduler/benchmark_policy.py` — Iterates through all available suites.

Security Note

The `liboqs` library is **not production-hardened**. The Open Quantum Safe project explicitly warns that the library is "designed for prototyping and evaluation" and should not be used in production systems without additional review. Algorithm implementations may change between versions, and side-channel protections vary by algorithm.

16.3.3 The `pyascon` Package

Attribute	Detail
Package name	`pyascon`
Role	Pure-Python fallback for ASCON-128a AEAD
License	CC0 / Public Domain
Performance	∼100× slower than native C

What it provides. A pure-Python implementation of the ASCON authenticated encryption algorithm family. ASCON won the NIST Lightweight Cryptography competition in 2023.

Why it exists alongside the native C extension. The codebase includes a hand-written C extension (`core/_ascon_native.c`) that implements ASCON-128a at near-native speed. The `pyascon` package serves as a *fallback* when the C extension cannot be compiled (e.g., on a platform without a C compiler or during development on an unfamiliar architecture).

```
try:
    from core._ascon_native import ascon_encrypt,
        ascon_decrypt
    _ASCON_BACKEND = "native-c"
except ImportError:
    from pyascon import ascon_encrypt, ascon_decrypt
    _ASCON_BACKEND = "pyascon-fallback"
```

Listing 16.2: ASCON backend selection in `core/aead.py`.

16.3.4 The `psutil` Package

Attribute	Detail
Package name	`psutil` (Process and System Utilities)
PyPI URL	https://pypi.org/project/psutil/
Version used	≥ 5.9
License	BSD-3-Clause
Platforms	Linux, Windows, macOS

What it provides. Cross-platform system resource monitoring: CPU usage, memory usage, disk I/O, network I/O, process information, and temperature sensors.

Where it is used.

- `core/metrics_collectors.py` — `psutil.cpu_percent()`, `psutil.virtual_memory()`, `psutil.Process().memory_info()`, `psutil.net_io_counters()`, `psutil.sensors_tempera` These populate Category N (System Resources) in the metrics schema.
- `sscheduler/local_monitor.py` — Reads drone CPU temperature and usage for the telemetry-aware scheduling policy.
- `core/process.py` — `psutil.Process(pid).children()` for orphan process detection.

Analogy

Think of `psutil` as a dashboard for your computer's vital signs, the same way a car's instrument cluster shows RPM, fuel, and temperature. The metrics collectors read these vitals continuously during each benchmark suite and record them for later analysis.

16.3.5 The `pymavlink` Package

Attribute	Detail
Package name	`pymavlink`
PyPI URL	https://pypi.org/project/pymavlink/
License	LGPL-3.0
Protocol	MAVLink v1 and v2

What it provides. A Python library for encoding, decoding, and routing MAVLink messages. It is the standard way to interact with MAVLink in Python.

Where it is used.

- `core/mavlink_metrics.py` — Creates a `mavutil.mavlink_connection` to a local UDP endpoint, then calls `recv_match()` to receive MAVLink messages. Tracks heartbeat intervals, sequence gaps, message type counts, and command ACK latencies.
- `sscheduler/gcs_telemetry_collector.py` — Receives MAVLink telemetry on the GCS side for cross-side correlation.
- `sscheduler/local_monitor.py` — Reads battery voltage, current, and armed state from MAVLink `SYS_STATUS` and `HEARTBEAT` messages.

16.3.6 The `smbus2` Package

Attribute	Detail
Package name	`smbus2`
PyPI URL	https://pypi.org/project/smbus2/
License	MIT
Platform	Linux only (requires `/dev/i2c-*`)

What it provides. A pure-Python I^2C/SMBus interface for Linux. It wraps the kernel's I^2C driver to read and write registers on I^2C devices.

Where it is used.

- `core/power_monitor.py` — The `INA219PowerMonitor` class uses `smbus2.SMBus(bus=1)` to access the INA219 current/voltage sensor at address `0x40`. It reads the shunt voltage register (0x01), bus voltage register (0x02), and power register (0x03) at up to 1,100 samples/second.

Implementation Note

The system uses `smbus2` for *direct register access* rather than the higher-level `pi-ina219` library because the higher-level library introduces a 2 ms delay per

sample (due to Python overhead in the driver class), limiting the sampling rate to ∼500 Hz. Direct register access via `smbus2` achieves ∼1,100 Hz.

16.3.7 The `ina219` (pi-ina219) Package

Attribute	Detail
Package name	`pi-ina219` (imported as `ina219`)
PyPI URL	https://pypi.org/project/pi-ina219/
License	MIT
Role	High-level INA219 driver (used as fallback)

What it provides. A user-friendly Python class that configures the INA219 and provides `voltage()`, `current()`, and `power()` methods with automatic calibration and range selection.

Relationship to `smbus2`. This package *uses* `smbus2` internally. The codebase uses `pi-ina219` as a configuration/calibration helper during sensor setup, then switches to raw `smbus2` register reads during high-speed data capture for maximum throughput.

16.3.8 The `zeroconf` Package

Attribute	Detail
Package name	`zeroconf`
PyPI URL	https://pypi.org/project/zeroconf/
License	LGPL-2.1
Protocol	mDNS / DNS-SD

What it provides. Implementation of Multicast DNS (mDNS) and DNS Service Discovery. Allows resolving hostnames like `drone.local` and `gcs.local` on the local network without a central DNS server.

Where it is used.

- `core/mdns_discovery.py` — When `CONFIG["ENABLE_MDNS"]` is set, the system attempts to resolve drone and GCS addresses via mDNS before falling back to static IP addresses from the configuration. Also advertises a `_pqc-tunnel._udp` service for zero-configuration discovery.
- `core/config.py` — The mDNS resolution is invoked during configuration loading.

16.3.9 The `FastAPI` Framework

Attribute	Detail
Package name	`fastapi`
Version	$\geq 0.109.0$
PyPI URL	https://pypi.org/project/fastapi/
License	MIT
Based on	Starlette (ASGI) + Pydantic (validation)

What it provides. A modern, high-performance web framework for building REST APIs with Python. Features automatic request validation, OpenAPI documentation generation, and async support.

Where it is used.

- `dashboard/backend/main.py` — Defines the FastAPI application with 11 API endpoints (see Chapter 13).
- `dashboard/backend/routes/suites.py` — Suite-specific endpoint implementations.
- `dashboard/backend/models.py` — Pydantic models for request/response validation.

Design Decision

Why FastAPI and not Flask or Django? FastAPI was chosen for three reasons: (1) native Pydantic integration for typed request/response models, (2) automatic OpenAPI/Swagger UI generation for API exploration during development, and (3) async support for non-blocking file I/O when loading large benchmark JSON files. Flask would have required additional plugins for all three features. Django would have been overkill for a read-mostly API with no database.

16.3.10 The `uvicorn` Server

Attribute	Detail
Package name	`uvicorn[standard]`
Version	$\geq 0.27.0$
License	BSD-3-Clause
Protocol	ASGI (Asynchronous Server Gateway Interface)

What it provides. A lightning-fast ASGI server that runs the FastAPI application. The `[standard]` extra includes `uvloop` (a faster event loop) and `httptools` (a faster HTTP parser).

Where it is used.

- `dashboard/backend/serve.py` — Launches the FastAPI app via `uvicorn.run(app, host="0.0.0.0", port=8000)`.

16.3.11 The `pydantic` Package

Attribute	Detail
Package name	`pydantic`
Version	$\geq$ 2.5.0 (Pydantic v2)
License	MIT
Role	Data validation and settings management

What it provides. Type-safe data validation using Python type annotations. Pydantic v2 is a ground-up rewrite in Rust (via `pydantic-core`) that is 5–50$\times$ faster than v1.

Where it is used.

- `dashboard/backend/models.py` — Defines typed models for all API responses: suite summaries, metric breakdowns, comparison results.
- `dashboard/backend/settings_store.py` — Persists user preferences (selected scenarios, comparison parameters).

16.3.12 The `pandas` Package

Attribute	Detail
Package name	`pandas`
Version	$\geq$ 2.1.0
License	BSD-3-Clause
Size	$\sim$30 MB (with NumPy dependency)

What it provides. The standard Python library for tabular data manipulation: DataFrames, groupby operations, CSV/JSON I/O, statistical aggregation, and data merging.

Where it is used.

- `dashboard/backend/analysis.py` — Loads JSONL metrics files into DataFrames, computes per-suite statistics (mean, median, p95, standard deviation), pivots data for cross-suite comparison, and exports CSV summaries.
- `dashboard/backend/reliability.py` — Computes suite-level pass/fail rates, handshake success rates, and data integrity metrics using DataFrame operations.
- `bench/` analysis scripts — The benchmark analysis pipeline uses pandas extensively for post-hoc statistical analysis of raw benchmark results.

16.3.13 The `numpy` Package

Attribute	Detail
Package name	`numpy`
License	BSD-3-Clause
Role	Numeric computation (transitive dependency of pandas)

What it provides. N-dimensional array operations, linear algebra, and numeric type handling. It is a *transitive dependency* of `pandas` but is also used directly in several places:

- JSON serialization helpers that handle `numpy.int64` and `numpy.float64` types (which are not natively JSON-serialisable).
- Benchmark analysis scripts for statistical computations (percentiles, means, standard deviations).

16.3.14 The `matplotlib` Package

Attribute	Detail
Package name	`matplotlib`
License	PSF-compatible
Role	Static chart generation for thesis/papers

What it provides. A comprehensive plotting library for creating publication-quality figures: bar charts, heatmaps, scatter plots, box plots, and more.

Where it is used.

- `bench/analyze_power_benchmark.py` — Generates power consumption charts (voltage, current, power over time) from INA219 sensor data.
- `bench/analyze_stress_test.py` — Creates performance comparison heatmaps across all 72 suites.
- `bench/generate_ieee_report.py` — Produces publication-ready figures for the IEEE benchmark report.
- `bench/generate_benchmark_book.py` — Creates all figures for the benchmark analysis book.

Implementation Note

`matplotlib` is used for *static* chart generation (saved to PNG/PDF files), while the dashboard uses `Recharts` for *interactive* browser-based charts. The two serve different audiences: matplotlib for publications, Recharts for exploration.

16.4 Python Standard Library: Notable Usage

The Python standard library provides many modules that are critical to the system's operation. While these are not "dependencies" in the pip-install sense, understanding *which* stdlib modules are used and *why* is essential for understanding the codebase.

16.4.1 `selectors` — I/O Multiplexing

The entire proxy event loop is built on `selectors.DefaultSelector()`, which wraps the OS-level I/O multiplexing mechanism (`epoll` on Linux, `select` on Windows). This was a deliberate choice over `asyncio` for deterministic latency (see Chapter 9 in Chapter 14).

16.4.2 `struct` — Binary Protocol Packing

The wire format uses `struct.pack()` and `struct.unpack()` with format string `"!BBBBB8sQB"` (22 bytes, network byte order) for every packet header. See Appendix B.

16.4.3 `ctypes` and `msvcrt` — OS-Level Process Control

- On **Windows**: `ctypes` calls into `kernel32.dll` to create Win32 Job Objects that kill child processes when the parent exits. `msvcrt.get_osfhandle()` obtains file handles for non-inheritable flag setting.

- On **Linux**: `ctypes.CDLL("libc.so.6")` calls `prctl(PR_SET_PDEATHSIG, SIGTERM)` to achieve the same parent-death cleanup.

16.4.4 `fcntl` / `msvcrt` — File Locking

The `RobustLogger` uses platform-specific file locking to ensure that append-mode writes to JSONL files are atomic:

- On Linux: `fcntl.flock(fd, fcntl.LOCK_EX)`

- On Windows: `msvcrt.locking(fd, msvcrt.LK_LOCK, size)`

16.4.5 `hashlib`, `hmac`, `secrets`

- `hashlib.sha256()` — Used in HMAC computations and PSK-based challenge-response authentication.

- `hmac.compare_digest()` — Constant-time comparison to prevent timing side-channel attacks during handshake verification.

- `secrets.token_bytes()` — Cryptographically secure random bytes for session IDs, nonces, and challenge values.

16.4.6 `subprocess` — External Process Management

Used extensively for launching:

- PQC proxy processes (drone and GCS modes)
- MAVProxy instances
- SSH connections for remote benchmark deployment
- Traffic generator processes

The `ManagedProcess` wrapper (Chapter 17 in Chapter 11) adds lifecycle guarantees on top of `subprocess.Popen`.

16.4.7 `statistics` — Descriptive Statistics

The `statistics` module provides `mean()`, `median()`, `stdev()`, and `quantiles()` for computing summary statistics of benchmark results (latency, throughput, power) within the metrics aggregator and robust logger.

16.4.8 `tkinter` — Devtools GUI

The `devtools/` package uses `tkinter` to provide a lightweight observability dashboard with real-time displays of proxy state, metric values, and system health. This is a *development-time* tool, not part of the production system.

16.4.9 `platform` — Cross-Platform Detection

Used in metrics collectors to detect:

- Architecture: ARM (aarch64) vs x86_64
- OS: Linux vs Windows
- Python version and implementation

This information populates Category A (Run Context) of the metrics schema and enables conditional logic (e.g., INA219 is only available on ARM/Linux).

16.5 The Native ASCON C Extension

The codebase includes a project-local C extension that implements the ASCON-128a AEAD cipher for maximum performance on the Raspberry Pi.

Attribute	Detail
Source file	`core/_ascon_native.c`
Compiled forms	`core/_ascon_native.so` (Linux aarch64) `core/_ascon_native.pyd` (Windows x64)
Build system	`core/setup_ascon.py` (distutils/setuptools)
Size	∼500 lines of C

Performance	∼100× faster than `pyascon` fallback

The C extension exposes two functions to Python:

- `ascon_encrypt(key, nonce, plaintext, associated_data)` → ciphertext
- `ascon_decrypt(key, nonce, ciphertext, associated_data)` → plaintext

A closure-based variant capture pattern in `core/aead.py` wraps these into the same interface as the `cryptography` package's AEAD classes, allowing the proxy to treat all three AEAD algorithms uniformly.

16.6 JavaScript / TypeScript Frontend Dependencies

The forensic analysis dashboard (Chapter 13) is a single-page web application built with modern JavaScript/TypeScript tooling. All frontend dependencies are declared in `dashboard/frontend/package.json`.

16.6.1 Runtime Dependencies

Table 16.16: Frontend runtime dependencies.

Package	Version	Role in the System
`react`	`^18.3.1`	Component-based UI library. Every dashboard page is a React functional component.
`react-dom`	`^18.3.1`	DOM rendering engine for React components. Provides `createRoot()` for mounting the app.
`react-router-dom`	`^6.22.0`	Client-side routing. Maps URL paths to the 12 dashboard pages (e.g., `/suites/:id` → `SuiteDetail.tsx`).
`recharts`	`^2.12.0`	Composable charting library built on React and D3.js. Used for all interactive charts: bar charts (handshake latency), scatter plots (latency vs. power), pie charts (suite distribution), radar charts (security comparison), heatmaps, and tooltips.
`@tanstack/react-table`	`^8.11.0`	Headless table library for sortable, filterable, paginated data tables. Powers the Suite Explorer and Metric Semantics pages.

`zustand`	`^4.5.0`	Lightweight state management (∼1 kB). A single Zustand store (`state/store.ts`) manages all dashboard state: selected scenario, loaded suites, active filters, comparison parameters, and user settings.

Design Decision

Why Zustand and not Redux? Redux is the most popular React state management library, but it requires significant boilerplate (action types, action creators, reducers, middleware). Zustand achieves the same result with a single `create()` call and direct state mutation via `set()`. For a dashboard with ∼12 pages and no complex async workflows, Zustand's simplicity is a clear win.

16.6.2 Development Dependencies

Table 16.17: Frontend development dependencies.

Package	Version	Role
`typescript`	`^5.4.0`	TypeScript compiler. All frontend code is written in TypeScript for static type checking.
`vite`	`^5.1.0`	Build tool and development server. Provides instant Hot Module Replacement (HMR) during development and optimised production builds via Rollup.
`@vitejs/plugin-react`	`^4.2.1`	Vite plugin enabling React Fast Refresh (preserves component state across edits) and JSX transformation.
`tailwindcss`	`^3.4.1`	Utility-first CSS framework. All dashboard styling uses Tailwind classes (e.g., `bg-blue-500`, `p-4`, `grid grid-cols-3`).
`postcss`	`^8.4.35`	CSS transformation pipeline. Required by Tailwind CSS for processing utility classes into standard CSS.
`autoprefixer`	`^10.4.18`	PostCSS plugin that adds vendor prefixes (`-webkit-`, `-moz-`) for cross-browser compatibility.
`@types/react`	`^18.3.0`	TypeScript type definitions for React's API.
`@types/react-dom`	`^18.3.0`	TypeScript type definitions for React-DOM's API.

16.7 System-Level Dependencies

These are components that exist outside the Python/JavaScript package managers and must be installed or configured at the operating system level.

16.7.1 The `liboqs` C Library

Attribute	Detail
Name	Open Quantum Safe `liboqs`
Version	0.10.x
Language	C (with CMake build system)
Install method	conda (`conda install -c conda-forge liboqs`) or compile from source
Size	∼15 MB compiled
Algorithms	30+ KEM and 20+ signature schemes

The `oqs-python` package (Section 16.3.2) is a thin Python wrapper around this C library. The C library must be compiled for the target architecture:

- **Drone (aarch64)**: Compiled on the Raspberry Pi 5 with ARM NEON optimisations.
- **GCS (x86_64)**: Compiled on Windows with AVX2 optimisations via MSVC or MinGW.

16.7.2 I²C Kernel Driver

On the Raspberry Pi, the I²C bus is exposed via `/dev/i2c-1`. The kernel module `i2c-bcm2835` must be loaded. This is typically enabled via `raspi-config` or by adding `dtparam=i2c_arm=on` to `/boot/config.txt`.

16.7.3 MAVProxy

MAVProxy is installed system-wide via pip (`pip install MAVProxy`) and is launched as a subprocess by the scheduler. It bridges the Pixhawk's USB serial connection to UDP endpoints that the PQC proxy can encrypt.

16.7.4 Node.js Runtime

The dashboard frontend requires Node.js (≥ 20) and npm for building. On the GCS (Windows), Node.js is installed via the official MSI installer. The frontend is built with `npm run build` (which invokes `vite build`) and served as static files.

16.7.5 Python 3.11+ Runtime

The codebase uses Python 3.11+ features:

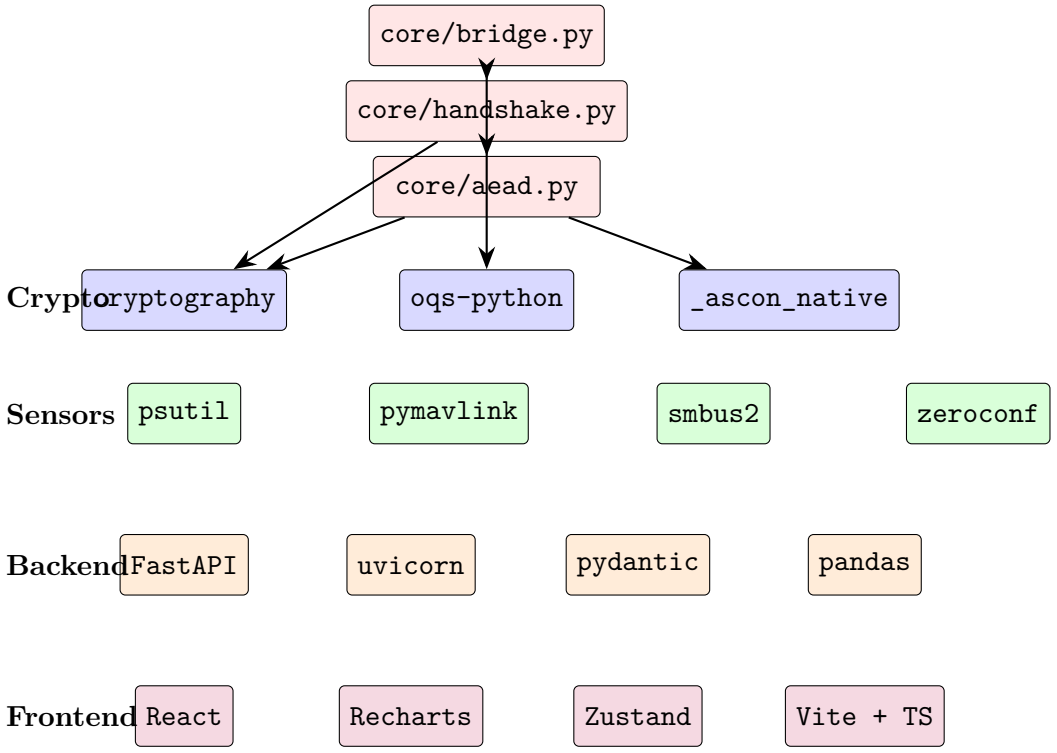
- **Union type syntax**: `str | None` instead of `Optional[str]`

- **Match statements**: `match action:` for control flow in the policy engine
- **Exception groups**: for handling multiple handshake failure modes
- **Improved error messages**: for debugging

On the drone, Python 3.11 is provided via the `oqs-dev` conda environment. On the GCS, it is installed via the official Python installer.

16.8 Dependency Relationships

The following diagram shows how the major dependencies relate to each other and to the system's architectural layers:



16.9 Version Pinning and Reproducibility

Security Note

Unpinned dependencies are a reproducibility and security risk. A `pip install cryptography` today may install a different version than it did six months ago, potentially with different performance characteristics or even different cryptographic behavior.

The dashboard backend pins minimum versions in its `requirements.txt`:

```
fastapi>=0.109.0
uvicorn[standard]>=0.27.0
pydantic>=2.5.0
pandas>=2.1.0
```

Listing 16.3: Dashboard backend requirements.txt.

The core and scheduler packages do not have a formal `requirements.txt`—they rely on the `oqs-dev` conda environment providing the correct versions of `liboqs`, `oqs-python`, `cryptography`, `psutil`, and other packages.

Key Insight

Conda vs. pip The drone-side environment uses conda rather than pip because `liboqs` requires a compiled C library that is difficult to install via pip alone. The `conda-forge` channel provides pre-compiled binaries for both x86_64 and aarch64 architectures.

For full reproducibility, the recommended approach is:

1. Record the exact environment: `conda list -export > environment.txt`
2. Record pip packages: `pip freeze > pip-freeze.txt`
3. Store both files alongside benchmark results
4. The metrics schema Category A records `liboqs_version`, `python_env_gcs`, and `python_env_drone` automatically

16.10 Dependency Security Considerations

Table 16.19: Security considerations for key dependencies.

Package	Risk Level	Consideration
`cryptography`	Low	Well-audited, backed by the Python Cryptographic Authority. Regular security releases. Uses OpenSSL underneath.
`oqs-python`	Medium	Explicitly labelled as "not production-ready" by the OQS project. Algorithm implementations may have side-channel vulnerabilities.
`pyascon`	Medium	Pure-Python crypto is inherently susceptible to timing side-channels. Used only as a fallback.
`FastAPI`	Low	Modern, well-maintained. Input validation via Pydantic reduces injection risks.
`uvicorn`	Low	Widely used ASGI server. Runs only on localhost in the default configuration.
`psutil`	Low	Read-only system monitoring. No network-facing attack surface.

`smbus2`	Medium	Requires root or `i2c` group membership. Misconfiguration could affect other I^2C devices on the bus.

16.11 Chapter Summary

This chapter documented every external dependency in the PQC Secure MAVLink Tunnel system:

- **14 Python packages**: `cryptography`, `oqs-python`, `pyascon`, `psutil`, `pymavlink`, `smbus2`, `pi-ina219`, `zeroconf`, `FastAPI`, `uvicorn`, `pydantic`, `pandas`, `numpy`, `matplotlib`.
- **8 Python standard library modules** of particular importance: `selectors`, `struct`, `ctypes/msvcrt`, `fcntl`, `hashlib/hmac/secrets`, `subprocess`, `statistics`, `platform`.
- **1 native C extension**: `_ascon_native` for high-performance ASCON-128a.
- **6 JavaScript runtime packages**: React, ReactDOM, React Router, Recharts, TanStack Table, Zustand.
- **8 JavaScript dev packages**: TypeScript, Vite, Tailwind CSS, PostCSS, Autoprefixer, and type definitions.
- **5 system-level dependencies**: `liboqs` (C), I^2C kernel driver, MAVProxy, Node.js, Python 3.11+.

For each dependency, we documented *what it provides*, *where it is used in the codebase*, *why it was chosen over alternatives*, and any *security considerations*. This chapter therefore serves as the authoritative reference for reproducing the build environment and understanding the supply-chain trust surface of the Secure Tunnel system.

Chapter 17

Codebase Walkthrough

This chapter provides a module-by-module tour of the entire codebase. Where previous chapters explained *concepts* (what AEAD framing is, how the handshake works), this chapter explains *code*: the exact files, classes, functions, data structures, and design patterns that implement those concepts.

17.1 Repository Layout

The repository is organized into six top-level packages and several supporting directories:

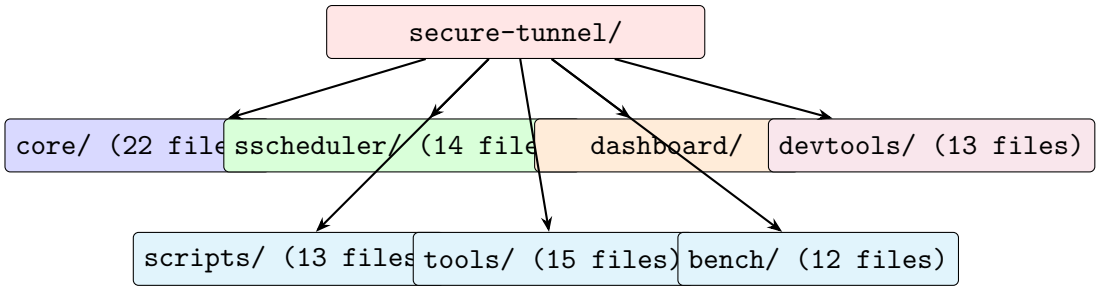


Table 17.1: Top-level directory purposes.

Directory	Files	Purpose
`core/`	22	Cryptographic primitives, AEAD framing, handshake protocol, proxy transport, suite registry, metrics schema, power monitoring, process management. This is the *heart* of the system.
`sscheduler/`	14	Drone-controlled scheduling—drone decides suite order and timing, GCS follows. Benchmark and MAVProxy-aware variants.
`dashboard/`	40+	FastAPI backend + Vite/React/TypeScript frontend for forensic benchmark visualisation (12 interactive pages).
`devtools/`	13	Developer-time observability: battery simulator, Tkinter dashboard, internal data bus, event emitter/receiver.

scripts/	13	Operational launch scripts (PowerShell/Bash), sensor tests, data transfer.
tools/	15	Runtime utilities: MAVProxy manager, network diagnostics, power measurement, config dumps, orchestration helpers.
bench/	12	Benchmark analysis pipeline: statistical analysis, plotting, IEEE report generation, benchmark book production.

17.2 The core/ Package: Module Census

The core/ package contains 22 Python files totalling approximately 11,000 lines of code. Table 17.2 lists every module with its line count, primary responsibility, and key classes or functions.

Table 17.2: Complete core/ module inventory.

Module	Lines	Responsibility
__init__.py	6	Package marker; declares the PQC Drone-GCS Secure Proxy Core Package.
aead.py	470	AEAD framing: Sender, Receiver, _AsconAdapter, replay window, epoch management, wire header packing.
bridge.py	1665	Selectors-based bidirectional proxy: handshake → encrypted bridge → in-band control. This is the **largest single file** in the codebase.
chronos.py	151	NTP-lite 3-way clock sync ("Operation Chronos").
config.py	623	Global CONFIG dictionary, mDNS resolution, env overrides, validation.
control_tcp.py	359	Minimal TCP JSON control server for in-band rekey negotiation with IP-based ACLs.
env_loader.py	83	Reads .denv/.genv key=value files into os.environ.
exceptions.py	25	Custom exception hierarchy: ConfigError, HandshakeError, AeadError, SequenceOverflow.
handshake.py	658	PQC authenticated handshake: KEM keygen, encapsulation, signature sign/verify, HKDF key derivation, ServerHello construction.
logging_utils.py	83	Structured JSON logging, file handler, lightweight metrics counters.
mavlink_metrics.py	879	Bidirectional MAVLink metrics: message tracking, heartbeat monitoring, sequence gaps, command ACK latency.

Module	Lines	Responsibility
mdns.py	203	mDNS/Zeroconf discovery: resolve drone.local/gcs.local, advertise _pqc-tunnel._udp.
metrics_aggregator.py	1365	Aggregates all 18 metric categories into ComprehensiveSuiteMetrics; lifecycle APIs, cross-side merging, validation verdicts.
metrics_collectors.py	753	Base and system collectors: CPU, memory, temperature, INA219 power, environment info, network stats.
metrics_schema.py	625	18 dataclasses (A–R) defining the complete typed metrics schema.
policy_engine.py	265	In-band two-phase commit rekey negotiation: prepare → confirm via packet type 0x02.
power_monitor.py	998	High-frequency power monitoring: INA219 direct register access, RPi5 PMIC hwmon, synthetic fallback, CSV export.
power_monitor_compat.py	19	Compatibility shim re-exporting all symbols from power_monitor.py.
process.py	302	Cross-platform ManagedProcess: Win32 Job Objects / Linux PDEATHSIG, SIGTERM→SIGKILL escalation, orphan prevention.
robust_logger.py	588	Append-mode persistent JSONL logger with file locking, incremental updates, SyncTracker.
run.py	917	CLI entrypoint: init-identity, gcs, drone subcommands, key file management.
suites.py	850	Suite registry: KEM×SIG×AEAD composition, alias resolution, OQS probing, MappingProxyType immutability.

17.3 core/exceptions.py: The Exception Hierarchy

Every non-trivial system needs a clear exception hierarchy so that callers can distinguish between different failure modes. The PQC tunnel defines five custom exception classes:

```
class ConfigError(NotImplementedError, ValueError):
    """Configuration validation errors."""
    pass

class SequenceOverflow(Exception):
    """Sequence space exhausted or nearing exhaustion."""
    pass

class HandshakeError(Exception):
    """Handshake protocol level errors."""
```

```
    pass

class AeadError(Exception):
    """AEAD-related errors."""
    pass

class HandshakeFormatError(HandshakeError):
    pass

class HandshakeVerifyError(HandshakeError):
    pass
```

Listing 17.1: The complete exception hierarchy (`core/exceptions.py`).

Key Insight

Inheritance for Error Handling `HandshakeFormatError` and `HandshakeVerifyError` both inherit from `HandshakeError`. This allows callers to catch either specific errors ("the signature was invalid" vs. "the message was malformed") or all handshake errors generically. `ConfigError` inherits from *both* `NotImplementedError` and `ValueError` for backward compatibility with legacy callers that catch either base type.

17.4 `core/env_loader.py`: Environment File Loading

The system supports `.denv` (drone-side) and `.genv` (GCS-side) environment files, plus `.local` overrides for site-specific configuration:

```
def load_env_files(
    repo_root: Optional[Path] = None,
    *,
    drone: bool = True,
    gcs: bool = True,
) -> dict[str, str]:
    if repo_root is None:
        repo_root = Path(__file__).resolve().parent.parent

    loaded: dict[str, str] = {}
    files = []
    if drone:
        files.append(repo_root / ".denv")
        files.append(repo_root / ".denv.local")
    if gcs:
        files.append(repo_root / ".genv")
        files.append(repo_root / ".genv.local")

    for env_file in files:
        pairs = _parse_env_file(env_file)
```

```
        loaded.update(pairs)

    # Inject into os.environ (existing values NOT overwritten)
    for key, value in loaded.items():
        if key not in os.environ:
            os.environ[key] = value

    return loaded
```

Listing 17.2: Environment file loading (core/env_loader.py).

Design Decision

Why not python-dotenv? The widely-used python-dotenv package provides similar functionality, but the system uses a custom loader to avoid an external dependency for what is fundamentally 30 lines of parsing code. The custom loader also supports the .denv/.genv naming convention (separating drone and GCS config) and the .local override pattern.

17.5 core/logging_utils.py: Structured JSON Logging

All log messages in the system are emitted as structured JSON, enabling machine parsing of logs during benchmark analysis.

```
class JsonFormatter(logging.Formatter):
    def format(self, record: logging.LogRecord) -> str:
        payload = {
            "ts": time.strftime(
                "%Y-%m-%dT%H:%M:%SZ", time.gmtime(record.created)
            ),
            "level": record.levelname,
            "name": record.name,
            "msg": record.getMessage(),
        }
        if record.exc_info:
            payload["exc_info"] = self.formatException(record.exc_info)
        # Include extra fields (filtered for JSON-serialisability)
        for k, v in record.__dict__.items():
            if k not in _STANDARD_RECORD_ATTRS:
                try:
                    json.dumps({k: v})
                    payload[k] = v
                except Exception:
                    payload[k] = str(v)
```

```
        return json.dumps(payload)
```

Listing 17.3: JSON log formatter (`core/logging_utils.py`).

The module also provides a lightweight metrics system (`Counter`, `Gauge`, `Metrics`) that avoids pulling in heavy observability libraries like Prometheus:

```
class Counter:
    def __init__(self): self.value = 0
    def inc(self, n: int = 1): self.value += n

class Gauge:
    def __init__(self): self.value = 0
    def set(self, v: float): self.value = v

METRICS = Metrics()  # global singleton
```

Listing 17.4: Lightweight metrics counters.

17.6 `core/config.py`: The Global Configuration

The `CONFIG` dictionary is the single source of truth for every tunable parameter. Here is how it initialises:

1. **Load env files**: `load_env_files()` reads `.denv`/`.genv` into `os.environ`.
2. **Resolve mDNS** (if enabled): Attempts `drone.local`/`gcs.local` resolution.
3. **Build defaults**: A large literal dictionary with all keys and their default values.
4. **Apply env overrides**: Selected keys can be overridden from environment variables.
5. **Validate**: `validate_config()` checks types, ranges, and constraints.

```
CONFIG = {
    # Handshake (TCP)
    "TCP_HANDSHAKE_PORT": 46000,

    # Encrypted UDP data-plane (network)
    "UDP_DRONE_RX": 46012,    # drone binds; GCS sends here
    "UDP_GCS_RX": 46011,      # GCS binds; drone sends here

    # Plaintext UDP (local loopback)
    "DRONE_PLAINTEXT_TX": 47003,
    "DRONE_PLAINTEXT_RX": 47004,
    "GCS_PLAINTEXT_TX": 47001,
    "GCS_PLAINTEXT_RX": 47002,

    # Hosts
```

```
    "DRONE_HOST": _DEFAULT_DRONE_HOST,
    "GCS_HOST": _DEFAULT_GCS_HOST,

    # Security
    "DRONE_PSK": "",
    "REPLAY_WINDOW": 1024,
    "WIRE_VERSION": 1,          # frozen protocol version

    # ... 50+ more keys (see Appendix A) ...
}
```

Listing 17.5: CONFIG dictionary structure (abbreviated, from `core/config.py`).

17.7 core/aead.py: The AEAD Framing Layer

This module implements the entire data-plane encryption/decryption path. The key design pattern is **algorithm abstraction**: all three AEAD algorithms (AES-GCM, ChaCha20-Poly1305, ASCON-128a) are wrapped behind a uniform interface.

17.7.1 The Import Fallback Chain

```
from cryptography.hazmat.primitives.ciphers.aead import
    AESGCM
try:
    from cryptography.hazmat.primitives.ciphers.aead import
        ChaCha20Poly1305
except ImportError:
    ChaCha20Poly1305 = None  # very old cryptography builds

try:
    from core import _ascon_native as _ascon_native_module
except Exception:
    _ascon_native_module = None

try:
    import pyascon as _pyascon_module
except Exception:
    _pyascon_module = None
```

Listing 17.6: AEAD backend selection with graceful fallback.

This triple-fallback pattern ensures that:

- If the native C extension is available, ASCON runs at maximum speed.
- If only `pyascon` is available, ASCON runs in pure Python (slower but functional).
- If neither is available, ASCON suites are excluded from the registry at runtime.

17.7.2 The ASCON Adapter

The `_AsconAdapter` class wraps the ASCON implementations (native C and Python fallback) behind the same `encrypt()`/`decrypt()` interface used by `AESGCM` and `ChaCha20Poly1305`:

```
class _AsconAdapter:
    def __init__(self, key: bytes, variant: str):
        self._key = key[:16]  # ASCON uses 128-bit keys
        variant_name = "Ascon-AEAD128a"

        if _ascon_native_module is not None:
            # Capture variant in closure
            def _native_encrypt(key, nonce, aad, pt, algo=
                variant_name):
                return _ascon_native_module.encrypt(
                    key, nonce, aad, pt, algo
                )
            self._encrypt_fn = _native_encrypt
        elif _pyascon_module is not None:
            self._encrypt_fn = _pyascon_fallback
        else:
            raise ValueError("No ASCON backend available")
```

Listing 17.7: Closure-based variant capture in the ASCON adapter.

Implementation Note

The closure captures `variant_name` as a default argument (`algo=variant_name`) to avoid late-binding issues. Without this, all ASCON instances would use the last variant assigned to the outer variable. This is a classic Python closure gotcha.

17.7.3 The Sender and Receiver Classes

The `Sender` class maintains:

- A monotonic 64-bit sequence counter (`seq`)
- An 8-bit epoch counter (`epoch`)
- The AEAD cipher object (algorithm-agnostic)
- Suite header IDs (KEM, SIG) for wire header construction
- The 8-byte session ID

The `Receiver` class additionally maintains:

- A sliding bitmap replay window of configurable size (default 1024)
- A high-water-mark sequence number
- Expected header values for validation

17.8 core/handshake.py: The PQC Handshake

This module implements the 2-message authenticated handshake protocol (Chapter 7). It is 658 lines and contains:

- The `ServerHello` frozen dataclass
- The `server_gcs_handshake()` function (GCS side)
- The `client_drone_handshake()` function (drone side)
- HKDF key derivation with domain separation
- Handshake metrics collection (nanosecond-precision timing)
- OQS compatibility layer (three import styles)

17.8.1 OQS Import Compatibility

Different versions of `oqs-python` expose the API differently. The codebase handles all three:

```
KeyEncapsulation = None
Signature = None
try:
    from oqs.oqs import KeyEncapsulation, Signature  # Style 1
except (ImportError, ModuleNotFoundError):
    try:
        from oqs import KeyEncapsulation, Signature  # Style 2
    except (ImportError, ModuleNotFoundError):
        try:
            import oqs                                # Style 3
            KeyEncapsulation = oqs.KeyEncapsulation
            Signature = oqs.Signature
        except (ImportError, ModuleNotFoundError, AttributeError):
            pass
```

Listing 17.8: Triple-style OQS import for version compatibility.

17.8.2 Handshake Metrics Instrumentation

Every cryptographic operation during the handshake is timed at nanosecond precision using `time.perf_counter_ns()`:

```
t0 = time.perf_counter_ns()
kem = KeyEncapsulation(kem_name)
public_key = kem.generate_keypair()
t1 = time.perf_counter_ns()
```

```
metrics["primitives"]["kem"]["keygen_ns"] = t1 - t0
```

Listing 17.9: Nanosecond-precision primitive timing.

These timings populate Category E (Crypto Primitive Breakdown) in the metrics schema with individual measurements for KEM keygen, encapsulation, decapsulation, signature signing, and verification.

17.9 `core/bridge.py`: The Proxy Engine

At 1,665 lines, `bridge.py` is the largest file in the codebase and the operational core of the entire system. It orchestrates the full proxy lifecycle:

1. Parse configuration and validate
2. Perform TCP handshake (calling into `handshake.py`)
3. Derive directional keys via HKDF
4. Create `Sender` and `Receiver` objects
5. Register four UDP sockets with the selector
6. Enter the event loop
7. Handle in-band control messages (rekey negotiation)
8. Write status files for the scheduler to read
9. Return counters on clean exit

17.9.1 The `ProxyCounters` Class

```
class ProxyCounters:
    def __init__(self) -> None:
        self.ptx_out = 0        # plaintext packets sent to
           app
        self.ptx_in = 0         # plaintext packets from app
        self.enc_out = 0        # encrypted packets to peer
        self.enc_in = 0         # encrypted packets from peer
        self.ptx_bytes_out = 0
        self.ptx_bytes_in = 0
        self.enc_bytes_out = 0
        self.enc_bytes_in = 0
        # Drop counters by reason
        self.drop_replay = 0
        self.drop_auth = 0
        self.drop_header = 0
        self.drop_ratelimit = 0
        self.drop_unknown = 0
```

Listing 17.10: Proxy statistics tracking.

17.9.2 The Main Proxy Function Signature

```python
def run_proxy(
    role: str,                # "drone" or "gcs"
    suite: dict,              # suite registry entry
    cfg: dict,                # CONFIG dictionary
    *,
    gcs_sig_secret=None,      # GCS signing secret key
    gcs_sig_public=None,      # GCS signing public key
    stop_after_seconds=None, # auto-stop (for benchmarks)
    status_file=None,         # path for scheduler status
    ready_event=None,         # threading.Event for readiness
) -> Dict[str, object]:
    """Start a blocking proxy for the given role."""
```

Listing 17.11: The proxy entry point (simplified signature).

17.9.3 Status File Communication

The proxy communicates its state to the scheduler via a JSON status file. This is a *file-based IPC* pattern chosen for simplicity and crash-resilience:

```python
def write_status(payload: Dict[str, object]) -> None:
    tmp_path = status_path.with_suffix(".tmp")
    data = json.dumps(payload)
    for attempt in range(2):
        try:
            tmp_path.write_text(data, encoding="utf-8")
            tmp_path.replace(status_path)  # atomic rename
            return
        except PermissionError:
            # Windows antivirus may briefly hold the file
            time.sleep(0.05)
```

Listing 17.12: Atomic status file writing with Windows retry.

Design Decision

Why file-based IPC instead of pipes or sockets? File-based IPC was chosen because: (1) the status file persists across process crashes, allowing the scheduler to detect stale proxies; (2) it works identically on Linux and Windows; (3) the file can be inspected manually during debugging (`cat status.json`). The trade-off is slightly higher latency compared to pipes, but status checks occur at 2-second intervals, so this is irrelevant.

17.10 core/suites.py: The Suite Registry

At 850 lines, the suite registry is responsible for generating all 72 cipher suite combinations and providing query/alias resolution. The core data structures are:

```
_KEM_REGISTRY = {
    "mlkem512": {
        "oqs_name": "ML-KEM-512",
        "token": "mlkem512",
        "nist_level": "L1",
        "kem_id": 1,
        "kem_param_id": 1,
        "aliases": (
            "ML-KEM-512", "ml-kem-512", "mlkem512",
            "kyber512", "kyber-512", "Kyber512",
        ),
    },
    # ... 8 more KEM entries ...
}
```

Listing 17.13: KEM registry entry structure.

The registry is frozen at module load time using `MappingProxyType` to prevent accidental mutation during benchmark runs.

17.11 core/process.py: Cross-Platform Process Management

This 302-line module solves one of the hardest cross-platform problems: ensuring that child processes *always* die when the parent exits.

17.11.1 Windows: Win32 Job Objects

```
if sys.platform.startswith("win"):
    _kernel32 = ctypes.windll.kernel32
    _JOB_OBJECT_LIMIT_KILL_ON_JOB_CLOSE = 0x00002000
    _JobObjectExtendedLimitInformation = 9

    class _JOBOBJECT_BASIC_LIMIT_INFORMATION(ctypes.
       Structure):
        _fields_ = [
            ("PerProcessUserTimeLimit", wintypes.
               LARGE_INTEGER),
            ("PerJobUserTimeLimit", wintypes.LARGE_INTEGER),
            ("LimitFlags", wintypes.DWORD),
            # ... 6 more fields ...
        ]

    def _create_job_object():
```

```
        job = _kernel32.CreateJobObjectW(None, None)
        info = _JOBOBJECT_EXTENDED_LIMIT_INFORMATION()
        info.BasicLimitInformation.LimitFlags = (
            _JOB_OBJECT_LIMIT_KILL_ON_JOB_CLOSE
        )
        _kernel32.SetInformationJobObject(
            job, _JobObjectExtendedLimitInformation,
            ctypes.byref(info), ctypes.sizeof(info)
        )
        return job
```

Listing 17.14: Creating a Win32 Job Object that kills children on close.

17.11.2 Linux: PDEATHSIG

```
else:  # Linux / POSIX
    _libc = ctypes.CDLL("libc.so.6")
    _PR_SET_PDEATHSIG = 1

    def _set_pdeathsig():
        """Called in subprocess preexec_fn."""
        _libc.prctl(_PR_SET_PDEATHSIG, signal.SIGTERM)
```

Listing 17.15: Linux parent-death signal via prctl.

Security Note

On Windows, the Job Object approach has a subtle failure mode: if the parent process is killed with `TerminateProcess` (SIGKILL equivalent), the Job Object handle is closed by the OS, which *does* trigger child termination. However, on older Windows versions, there is a race condition where the child can briefly survive if it creates its own child process before being assigned to the job. The system mitigates this by assigning the job *immediately* after process creation, before the child has time to spawn sub-processes.

17.12 core/power_monitor.py: High-Frequency Power Measurement

At 998 lines, this is the third-largest file in `core/`. It implements three power monitoring backends:

1. `Ina219PowerMonitor` — Direct I^2C register access to the INA219 sensor at up to 1,100 samples/second.
2. `Rpi5PmicMonitor` — Reads the RPi 5's built-in PMIC via Linux `hwmon` sysfs files.
3. `SyntheticPowerMonitor` — Generates simulated power data for development on machines without hardware sensors.

```
_ADC_PROFILES = {
    "highspeed": {
        "badc": 0x0080,
        "sadc": 0x0000,
        "settle": 0.0004,
        "hz": 1100,
    },
    "balanced": {
        "badc": 0x0400,
        "sadc": 0x0018,
        "settle": 0.0010,
        "hz": 900,
    },
    "precision": {
        "badc": 0x0400,
        "sadc": 0x0048,
        "settle": 0.0020,
        "hz": 450,
    },
}
```

Listing 17.16: INA219 ADC profile configuration.

17.13 `core/run.py`: The CLI Entrypoint

The system is launched via `python -m core.run` (or `python core/run.py`) with three subcommands:

1. **`init-identity`** — Generates a persistent GCS signing keypair and saves it to files (`gcs_sig.secret`, `gcs_sig.pub`).
2. **`gcs`** — Starts the GCS-side proxy, loading the signing secret key from file.
3. **`drone`** — Starts the drone-side proxy, loading the GCS public key from file.

The 917-line file handles argument parsing, key file management, signal handling (graceful shutdown on Ctrl+C), and the metric flattening pipeline that converts raw handshake metrics into the structured schema format.

17.14 The `sscheduler/` Package: Module Census

The scheduler package contains 14 files totalling approximately 6,000 lines. Table 17.3 provides the complete inventory.

Table 17.3: Complete sscheduler/ module inventory.

Module	Lines	Responsibility
__init__.py	3	Package marker.
benchmark_policy.py	585	Systematic suite cycling policy for benchmarks (BenchmarkAction, SuiteMetrics, BenchmarkOutput).
control_auth.py	42	PSK-based HMAC-SHA256 challenge-response for the control plane.
gcs_telemetry_collector.py	386	GCS-side real-time telemetry receiver for Category I/J/K metrics.
local_monitor.py	184	Drone-side system monitor: Pi temperature, CPU, Pixhawk battery.
policy.py	414	TelemetryAwarePolicyV2: tier-based upgrade/ downgrade with hysteresis and blacklisting.
drone_scheduler.py	588	Simplified drone controller loop.
drone_scheduler_bench.py	1181	Benchmark-specialized drone controller with full BenchmarkPolicy integration.
drone_scheduler_mav.py	822	MAVProxy-aware drone controller for live flight testing.
gcs_scheduler.py	610	Simplified GCS follower loop.
gcs_scheduler_bench.py	1086	GCS benchmark server ("Operation Chronos v2"): proxy management, traffic generation, metrics.
gcs_scheduler_mav.py	652	MAVProxy-aware GCS follower for live testing.
telemetry_window.py	194	Thread-safe bounded sliding window for telemetry analysis with O(1) add.

17.14.1 Control Channel Authentication

```
def create_challenge() -> bytes:
    """Generate a random 32-byte challenge."""
    return os.urandom(32)

def compute_response(challenge: bytes, psk: bytes) -> str:
    """Compute HMAC-SHA256 response."""
    return hmac.new(psk, challenge, hashlib.sha256).
       hexdigest()

def verify_response(
    challenge: bytes, response: str, psk: bytes
) -> bool:
    """Verify response with constant-time comparison."""
    expected = compute_response(challenge, psk)
    return hmac.compare_digest(response, expected)
```

Listing 17.17: HMAC-SHA256 challenge-response (`sscheduler/control_auth.py`).

17.15 The `devtools/` Package

The `devtools/` package provides developer-time observability tools that are *not* part of the production system. Table 17.4 lists its 13 files.

Table 17.4: Devtools module inventory.

Module	Purpose
`battery_bridge.py`	Bridges real/simulated battery data to the policy engine.
`battery_sim.py`	Generates synthetic battery discharge curves for testing TelemetryAwarePolicyV2 without real hardware.
`config.py`	Devtools-specific configuration overrides.
`dashboard.py`	A Tkinter-based real-time observability GUI showing proxy state, metrics, and system health.
`data_bus.py`	Internal publish-subscribe data bus for decoupling devtools components.
`integration.py`	Helpers for integrating devtools with the main system.
`launcher.py`	Launches devtools components as sub-processes.
`observable_events.py`	Defines the observability event schema (event types, payloads, timestamps).
`observer.py`	Observability event receiver that displays events in the Tkinter dashboard or logs them.
`observer_schema.py`	Type definitions for observability events.
`test_obs_plane.py`	Test harness for the observability plane.
`README.md`	Developer documentation for the devtools package.

17.16 The `scripts/` and `tools/` Directories

17.16.1 Operational Scripts (`scripts/`)

These are launch scripts and operational utilities:

Table 17.5: Scripts directory inventory.

File	Purpose
`analyze_results.py`	Post-hoc statistical analysis of benchmark results.
`clean_start_benchmark.ps1`	PowerShell script for clean-state benchmark launch.
`generate_keys.py`	Regenerates the full PQC key matrix for all suites.

run_gcs_metrics.py	Launches GCS-side metrics collection standalone.
run_gcs_telemetry.py	GCS telemetry receiver (v1 protocol).
run_drone.sh	Bash script to launch drone scheduler on Pi.
run_gcs.ps1	PowerShell script to launch GCS scheduler.
test_telemetry_rx.py	Telemetry receiver test harness.
test_telemetry_tx.py	Telemetry sender test harness.
test_ina219.py	INA219 power sensor hardware test.
transfer_data.py	SCP-based data transfer between drone and GCS.
validate_policy.py	Validates scheduling policy state transitions.
verify_rpc.py	Verifies control channel RPC connectivity.

17.16.2 Runtime Tools (tools/)

These are diagnostic and utility tools that can be run independently:

Table 17.6: Tools directory inventory.

File	Purpose
blackout_metrics.py	Measures metrics quality during rekey blackout periods.
dump_config.py	Dumps the current CONFIG dictionary to stdout.
dump_suites.py	Lists all registered suites with their properties.
gcs_ping.py	Pings the GCS control endpoint for connectivity checks.
ina219_read.py	Reads current INA219 sensor values.
mavproxy_manager.py	Manages MAVProxy process lifecycle.
mavsniff.py	MAVLink UDP packet sniffer for debugging.
merge_power.py	Merges power measurement data from multiple captures.
network_diag.py	Network diagnostics (ping, port scan, route trace).
orchestrate.py	Full benchmark run orchestration helper.
power_utils.py	Power measurement utilities and unit conversions.
verify_dashboard.py	Verifies dashboard data integrity against raw metrics.
verify_metrics.py	Cross-validates metrics for internal consistency.
wait_for_metrics.py	Polls for metrics completion (used in CI/scripts).

17.17 The bench/ Package: Benchmark Analysis

The bench/ directory contains the post-hoc analysis pipeline:

Table 17.7: Benchmark analysis module inventory.

Module	Purpose
`analyze_power_benchmark.py`	Power consumption analysis: voltage, current, power time series from INA219 data.
`analyze_stress_test.py`	Performance comparison heatmaps across all suites.
`benchmark_power_perf.py`	Combined power + performance benchmarking.
`benchmark_pqc.py`	Standalone PQC primitive benchmark (keygen, encaps, sign, verify timing).
`consolidate_metrics.py`	Merges partial metrics files into consolidated datasets.
`deploy_and_run.py`	SSH-based deployment to drone and remote execution.
`generate_benchmark_book.py`	Generates all figures for the benchmark analysis book.
`generate_ieee_book.py`	Produces publication-ready figures for IEEE papers.
`generate_ieee_report.py`	Creates the full IEEE benchmark report with tables and charts.
`run_full_benchmark.py`	End-to-end benchmark orchestration (all 72 suites).

17.18 Line Count Summary

Table 17.8: Codebase size by directory.

Directory	Python Files	Total Lines
`core/`	22	∼11,000
`sscheduler/`	14	∼6,000
`dashboard/backend/`	8	∼2,500
`devtools/`	13	∼1,500
`scripts/`	13	∼1,200
`tools/`	15	∼1,800
`bench/`	12	∼3,000
Total Python	**97**	**∼27,000**
`dashboard/frontend/`	(TypeScript)	∼3,500
`core/_ascon_native.c`	(C)	∼500
Grand Total		**∼31,000**

17.19 Chapter Summary

This chapter walked through every directory and every significant module in the codebase:

- **22 modules in `core/`** — the cryptographic engine, proxy transport, metrics collection, power monitoring, and process management.
- **14 modules in `sscheduler/`** — the controller-follower scheduling architecture with three operational variants (simple, benchmark, MAVProxy).
- **13 modules in `devtools/`** — developer observability tools including a Tkinter GUI, battery simulator, and data bus.
- **13 scripts + 15 tools** — operational launch scripts, diagnostic utilities, and verification helpers.
- **12 modules in `bench/`** — the post-hoc analysis pipeline for generating statistical reports and publication figures.

The total codebase is approximately 31,000 lines across 97 Python files, 3,500 lines of TypeScript, and 500 lines of C.

Chapter 18

Testing and Validation

This chapter documents the testing architecture, every test file in the repository, the rationale behind the custom test runner approach, and how to execute the full test suite. Unlike projects that rely on `pytest` or `unittest` with mocks, this system *tests what ships*—every test starts real processes, performs real PQC handshakes, and sends real encrypted traffic.

18.1 Testing Philosophy

The testing strategy follows three guiding principles:

Key Insight

Test What Ships Every test exercises the real code paths that production traffic uses. There are no mocked cryptographic libraries, no fake sockets, and no synthetic handshake responses. When a test runs the proxy, it performs a genuine PQC key encapsulation and a real AEAD encryption cycle.

1. **Integration-First.** The smallest unit of testing is a full proxy round-trip: handshake → AEAD encrypt → network transfer → AEAD decrypt → application delivery. This catches protocol bugs that unit tests would miss.

2. **Real Processes, Real Sockets.** Tests spawn the GCS and drone proxy as actual sub-processes using `subprocess.Popen`. This validates process lifecycle, signal handling, and cross-platform compatibility.

3. **Verification at Multiple Layers.** The test suite includes functional tests (does traffic flow?), collector smoke tests (do metrics collectors produce data?), schema validators (are output files complete?), and benchmark validators (are statistical results sane?).

18.2 Test Architecture Overview

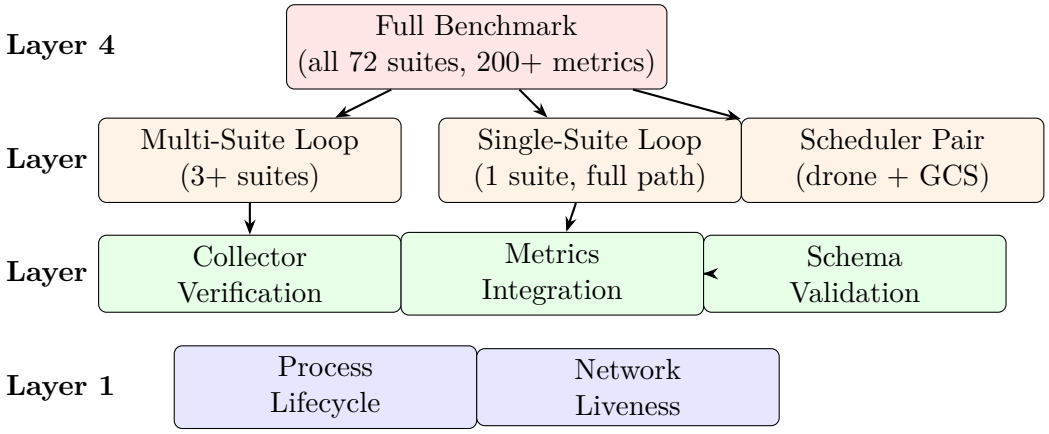


Table 18.1: Test layer classification.

Layer	Scope	What It Verifies	Files
4	Full Benchmark	All 72 suites complete with 200+ metrics each, statistical validity	1
3	Integration	Full proxy round-trip: handshake → encrypt → transfer → decrypt	6
2	Component	Individual collectors produce valid data; metrics schema is complete	5
1	Unit / Liveness	Process lifecycle; network connectivity; basic function correctness	2

18.3 Why No `pytest` or `unittest`?

A natural question is why the project does not use standard testing frameworks. The answer lies in the operational requirements:

Design Decision

Custom Test Runners Over `pytest` Standard test frameworks assume tests are:

- Fast (sub-second),
- Independent (no shared state),
- Local (no network I/O).

The PQC tunnel tests violate all three assumptions. Each test:

- Takes 10–300 seconds (PQC handshakes are slow),
- Requires shared network ports and process coordination,
- Sends real UDP traffic between real processes.

Using `pytest` would require extensive fixture management for process cleanup, port allocation, and timeout handling. The custom `if __name__ == "__main__"` pattern gives each test full control over its lifecycle without framework overhead.

The trade-off is clear: no automatic test discovery, no parametrization, no coverage reports. For a hardware-coupled system with 72 cipher suites, this is an acceptable trade-off—what matters is that the actual crypto path works, not that tests have pretty output.

18.4 Test File Inventory

The repository contains 18 test-related files organized into three categories. All files live at the project root (not in a `tests/` subdirectory), reflecting their role as operational verification scripts rather than unit tests.

18.4.1 Category 1: Functional Integration Tests (`test_*`)

Table 18.2: Functional integration test files.

File	Lines	Purpose
`test_simple_loop.py`	199	Simplest possible test: hardcoded suite (`cs-mlkem768-aesgcm-mldsa65`), 20 packets, pass/fail based on received count > 0.
`test_complete_loop.py`	294	Single-suite full loop with RTT measurement, environment variable configuration, and delivery rate calculation.
`test_all_complete_loop.py`	193	Iterates **all registered suites**, running a full loop for each. Supports argparse for duration, bandwidth, and iteration count.
`test_localhost_loop.py`	183	Echo-only test that assumes proxies are already running externally. Measures RTT for 50 packets.
`test_multiple_suites.py`	233	Tests 3 diverse suites in sequence: ML-KEM-512+Falcon, HQC-128+ML-DSA-44, HQC-192+ML-DSA-65.
`test_schedulers.py`	87	Starts a drone+GCS scheduler pair on localhost, verifies both complete 2 suites within timeout.
`test_sscheduler.py`	78	Tests the reversed-control scheduling model: drone commands, GCS follows.
`test_metrics_integration.py`	338	Full metrics pipeline test: single suite loop + all 18 metric categories validated in output files.
`test_comprehensive_benchmark.py`	505	All 72 suites with 200+ metrics each. Supports distributed mode (`-drone`/`-gcs` flags).
`test_gcs_ping.py`	16	Network liveness check: sends TCP JSON `ping` to GCS control server.

`test_collectors.py`	47	Smoke test: instantiates 3 collectors and the metrics aggregator.

18.4.2 Category 2: Collector Verification Tests (`verify_*`)

Table 18.3: Collector verification files.

File	Lines	Purpose
`verify_collectors.py`	101	Platform-agnostic verification of all 5 collectors (Environment, System, Network, Latency, Power). Auto-detects drone/GCS role.
`verify_drone_collectors.py`	52	Drone-targeted collector verification with 100 synthetic latency samples.
`verify_gcs_collectors.py`	83	GCS-side strict verification using dictionary access (throws on missing keys).
`verify_metrics_output.py`	84	Schema validator: reads JSON/JSONL output files and checks 21+ required nested fields across all metric categories.

18.4.3 Category 3: Output Validation (`validate_*`, `confirm_*`)

Table 18.4: Output validation files.

File	Lines	Purpose
`validate_bench.py`	151	Reads raw benchmark JSON files, computes wall/perf time statistics (mean, median, stdev), reports success rates per category.
`confirm_all_metrics.py`	409	Introspects all 18 dataclass categories (A–R) from the metrics schema, confirming all 231 individual metric fields are properly defined.

18.5 Detailed Test Walkthroughs

18.5.1 The Simple Loop Test

The simplest test in the suite demonstrates the core testing pattern. It is 199 lines and tests one cipher suite end-to-end:

```
# 1. Configure environment for localhost
os.environ["DRONE_HOST"] = "127.0.0.1"
os.environ["GCS_HOST"] = "127.0.0.1"
```

```
4
5  # 2. Select a known-good suite
6  SUITE = "cs-mlkem768-aesgcm-mldsa65"
7
8  # 3. Start GCS proxy subprocess
9  gcs_proc = subprocess.Popen([
10     sys.executable, "-m", "core.run", "gcs",
11     "--suite", SUITE
12 ])
13
14 # 4. Start drone proxy subprocess
15 drone_proc = subprocess.Popen([
16     sys.executable, "-m", "core.run", "drone",
17     "--suite", SUITE
18 ])
19
20 # 5. Wait for handshake to complete
21 time.sleep(15)
22
23 # 6. Start echo server on drone side
24 def echo_server():
25     sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
26     sock.bind(("127.0.0.1", DRONE_PLAINTEXT_TX))
27     while True:
28         data, addr = sock.recvfrom(65535)
29         sock.sendto(data, ("127.0.0.1", DRONE_PLAINTEXT_RX))
30
31 threading.Thread(target=echo_server, daemon=True).start()
32
33 # 7. Send test packets from GCS side
34 sent = received = 0
35 for i in range(20):
36     tx_sock.sendto(b"PING-%d" % i, GCS_PLAINTEXT_TX)
37     sent += 1
38     try:
39         rx_sock.settimeout(2.0)
40         data = rx_sock.recv(65535)
41         received += 1
42     except socket.timeout:
43         pass
44
45 # 8. Verdict
46 if received > 0:
47     print(f"PASS: {received}/{sent} packets")
48 else:
49     print("FAIL: zero packets received")
```

Listing 18.1: Simplified structure of `test_simple_loop.py`.

18.5.2 The All-Suites Loop Test

The `test_all_complete_loop.py` test is the most thorough functional test. It iterates over *every* registered cipher suite and runs a complete loop for each:

```
from core.suites import get_all_suite_ids

parser = argparse.ArgumentParser()
parser.add_argument("--duration", type=int, default=30)
parser.add_argument("--bandwidth", type=int, default=110)
parser.add_argument("--iterations", type=int, default=1)
args = parser.parse_args()

results = {}
for suite_id in get_all_suite_ids():
    print(f"\n{'='*60}")
    print(f"Testing suite: {suite_id}")
    try:
        success, stats = run_single_suite_test(
            suite_id, args.duration, args.bandwidth
        )
        results[suite_id] = {
            "success": success,
            "delivery_rate": stats["delivery_rate"],
            "avg_rtt_ms": stats.get("avg_rtt_ms"),
        }
    except Exception as e:
        results[suite_id] = {"success": False, "error": str(
           e)}

# Summary
passed = sum(1 for r in results.values() if r["success"])
print(f"\nPassed: {passed}/{len(results)}")
```

Listing 18.2: Suite iteration in `test_all_complete_loop.py`.

18.5.3 The Scheduler Pair Test

Testing the scheduler requires coordinating two processes with specific startup ordering (GCS must be listening before drone connects):

```
def main():
    os.environ.update({
        "DRONE_HOST": "127.0.0.1",
        "GCS_HOST": "127.0.0.1",
    })

```

```
    # GCS starts first (listening mode)
    gcs = subprocess.Popen([
        sys.executable, "-m", "sscheduler.sgcs",
        "--max-suites", "2",
        "--suite-seconds", "10",
        "--mbps", "110",
    ])

    time.sleep(3)  # allow GCS to bind

    # Drone connects to GCS
    drone = subprocess.Popen([
        sys.executable, "-m", "sscheduler.sdrone",
        "--max-suites", "2",
    ])

    try:
        drone.wait(timeout=300)
        gcs.wait(timeout=30)
        assert drone.returncode == 0
        assert gcs.returncode == 0
        print("PASS: scheduler pair completed")
    finally:
        for p in [drone, gcs]:
            if p.poll() is None:
                p.terminate()
```

Listing 18.3: Scheduler pair test (`test_schedulers.py`).

18.5.4 The Comprehensive Benchmark Test

The most complex test file (505 lines) runs every cipher suite with full metrics collection:

```
class ComprehensiveBenchmark:
    def __init__(self, args):
        self.config = {
            "suite_timeout": 120,       # seconds per suite
            "handshake_timeout": 60,    # max handshake wait
            "traffic_duration": 30,     # seconds of traffic
            "traffic_bandwidth": 110,   # Mbps target
            "retry_count": 2,           # retries on failure
            "output_dir": Path("bench_results") / timestamp,
            "metrics_categories": list("ABCDEFGHIJKLMNOPQR")
                ,
        }

    def run_all_suites(self):
        for suite_id in get_all_suite_ids():
            for attempt in range(self.config["retry_count"])
                :
                result = self._run_single(suite_id)
```

```
                    if result.success:
                        self._save_metrics(suite_id, result)
                        break
                else:
                    self.failures.append(suite_id)
```

Listing 18.4: Benchmark test configuration.

18.5.5 Metrics Confirmation

The `confirm_all_metrics.py` script (409 lines) is a meta-test that validates the metrics *schema itself*—it introspects all 18 dataclass categories using Python's `dataclasses` module:

```
from dataclasses import fields
from core.metrics_schema import (
    EnvironmentInfo,     # Category A
    HandshakeMetrics,    # Category B
    TransportMetrics,    # Category C
    # ... all 18 categories ...
)

total_fields = 0
for category_cls in ALL_CATEGORIES:
    category_fields = fields(category_cls)
    total_fields += len(category_fields)
    for f in category_fields:
        assert f.name, f"Empty field name in {category_cls}"
        # Verify type annotation exists
        assert f.type is not None

print(f"Verified {total_fields} fields across "
      f"{len(ALL_CATEGORIES)} categories")
# Expected: 231 fields across 18 categories
```

Listing 18.5: Schema introspection in `confirm_all_metrics.py`.

18.6 Common Test Patterns

18.6.1 Process Lifecycle Management

Every integration test follows the same process management pattern:

```
processes = []
try:
    gcs = subprocess.Popen([...])
    processes.append(gcs)

    drone = subprocess.Popen([...])
```

```
    processes.append(drone)

    # ... run test ...

finally:
    for p in processes:
        if p.poll() is None:
            p.terminate()
            try:
                p.wait(timeout=5)
            except subprocess.TimeoutExpired:
                p.kill()
```

Listing 18.6: Standard process cleanup pattern.

18.6.2 Echo Server Pattern

Tests simulate the drone flight controller with a UDP echo server that reflects every packet it receives:

```
def echo_server(rx_port: int, tx_port: int):
    """Simulate drone flight controller."""
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind(("127.0.0.1", rx_port))
    sock.settimeout(1.0)
    while True:
        try:
            data, addr = sock.recvfrom(65535)
            sock.sendto(data, ("127.0.0.1", tx_port))
        except socket.timeout:
            continue

thread = threading.Thread(target=echo_server, daemon=True)
thread.start()
```

Listing 18.7: UDP echo server used in integration tests.

18.6.3 Port Configuration

Tests read port numbers from the global `CONFIG` dictionary to ensure consistency with the proxy:

```
from core.config import CONFIG

GCS_TX = ("127.0.0.1", CONFIG["GCS_PLAINTEXT_TX"])   # 47001
GCS_RX = ("127.0.0.1", CONFIG["GCS_PLAINTEXT_RX"])   # 47002
DRONE_TX = ("127.0.0.1", CONFIG["DRONE_PLAINTEXT_TX"]) #
    47003
DRONE_RX = ("127.0.0.1", CONFIG["DRONE_PLAINTEXT_RX"]) #
    47004
```

Listing 18.8: Port configuration in tests.

18.7 Running the Tests

18.7.1 Prerequisites

Before running any test, ensure:

1. Python 3.11+ is installed with all dependencies (`pip install -r requirements.txt`).
2. The liboqs library is installed and accessible.
3. GCS signing keys have been generated: `python -m core.run init-identity`.
4. No other process is using ports 46000–47004.

18.7.2 Quick Smoke Test (2 minutes)

```
# Verify collectors work
python verify_collectors.py

# Test single suite round-trip
python test_simple_loop.py
```

18.7.3 Single-Suite Integration (5 minutes)

```
# Full loop with RTT measurement
python test_complete_loop.py

# Metrics integration
python test_metrics_integration.py
```

18.7.4 Multi-Suite Sweep (30 minutes)

```
# Test diverse algorithm combinations
python test_multiple_suites.py

# Test scheduler coordination
python test_schedulers.py
python test_sscheduler.py
```

18.7.5 Full Benchmark (4–6 hours)

```
# All 72 suites with comprehensive metrics
python test_comprehensive_benchmark.py

# Validate results
python validate_bench.py
python confirm_all_metrics.py
```

18.8 Validation Scripts

18.8.1 Schema Validation

The `verify_metrics_output.py` script checks that benchmark output files contain all required fields:

```
REQUIRED_FIELDS = {
    "environment": [
        "hostname", "platform", "python_version",
        "cpu_model", "cpu_count",
    ],
    "handshake": [
        "kem_name", "sig_name", "aead_name",
        "total_ms", "kem_keygen_ns",
    ],
    "transport": [
        "packets_sent", "packets_received",
        "bytes_encrypted", "delivery_rate",
    ],
    "power": [
        "mean_power_w", "total_energy_j",
    ],
    "rekey": [...],
    "control_plane": [...],
    # ... 21+ required nested paths ...
}
```

Listing 18.9: Required fields checked by `verify_metrics_output.py`.

18.8.2 Benchmark Result Validation

The `validate_bench.py` script reads raw benchmark JSON files and computes statistical summaries:

```
def validate_results(results_dir: Path):
    stats = {}
    for json_file in results_dir.glob("*.json"):
        data = json.loads(json_file.read_text())
```

```
        suite_id = data.get("suite_id", json_file.stem)
        stats[suite_id] = {
            "wall_time_s": data["timing"]["wall_seconds"],
            "handshake_ms": data["handshake"]["total_ms"],
            "delivery_rate": data["transport"]["
               delivery_rate"],
        }

    # Compute aggregates
    wall_times = [s["wall_time_s"] for s in stats.values()]
    print(f"Mean wall time: {statistics.mean(wall_times):.1f
       }s")
    print(f"Median: {statistics.median(wall_times):.1f}s")
    print(f"Stdev: {statistics.stdev(wall_times):.1f}s")
```

Listing 18.10: Statistical validation in `validate_bench.py`.

18.9 Test Coverage Analysis

Table 18.5 maps which tests cover which system components.

Table 18.5: Test file vs. system component coverage matrix.

Test File	Handshake	AEAD	Proxy	Scheduler	Metrics	Power
test_simple_loop	✓	✓	✓			
test_complete_loop	✓	✓	✓			
test_all_complete_loop	✓	✓	✓			
test_localhost_loop		✓	✓			
test_multiple_suites	✓	✓	✓			
test_schedulers	✓	✓	✓	✓		
test_sscheduler	✓	✓	✓	✓		
test_metrics_integration	✓	✓	✓		✓	
test_comprehensive_benchmark	✓	✓	✓	✓	✓	✓
verify_collectors					✓	✓
verify_drone_collectors					✓	✓
verify_gcs_collectors					✓	
confirm_all_metrics					✓	
validate_bench					✓	

18.10 Lessons Learned

1. **PQC handshakes are slow.** Tests must account for 15–60 second handshake times for HQC and BIKE algorithms. Timeouts must be generous.

2. **Port conflicts are real.** Running multiple tests concurrently fails because they share the same port ranges. Tests are designed to run sequentially.

3. **Process cleanup is critical.** A test that crashes without cleaning up proxy processes will leave orphan processes holding ports. The `finally` block pattern is essential.

4. **Cross-platform differences matter.** Windows and Linux handle subprocess termination differently. The `ManagedProcess` class (Chapter 17) addresses this, but tests still need platform-aware timeouts.

18.11 Chapter Summary

The PQC drone tunnel test suite comprises 18 files totalling approximately 3,200 lines of test code. The testing philosophy prioritises *real execution* over mocking, with every test performing actual PQC key exchanges and encrypted data transfers. The test pyramid spans four layers:

1. **Unit/liveness** — process lifecycle, network ping

2. **Component** — individual collectors and schema validation

3. **Integration** — single and multi-suite proxy round-trips

4. **Benchmark** — all 72 suites with 231 metrics per suite

Running the full test suite takes 4–6 hours and exercises every cryptographic code path in production.

Chapter 19

Benchmark Results and Analysis

This chapter presents the complete benchmark results from 19,600 individually-timed cryptographic operations across nine KEM algorithms, eight signature algorithms, three AEAD ciphers, and 23 full cipher suite handshakes. All measurements were performed on a Raspberry Pi 4 Model B, the reference drone platform.

19.1 Test Environment

Table 19.1: Benchmark environment specifications.

Parameter	Value
Platform	Raspberry Pi 4 Model B Rev 1.5
Hostname	`uavpi`
CPU	ARM Cortex-A72, 4 cores, 1.8 GHz (governor: `ondemand`)
RAM	3,796 MB
Kernel	6.12.47+rpt-rpi-v8 (64-bit)
Python	3.11.2 (GCC 12.2.0)
PQC Library	liboqs via `oqs-python`
Power Sensor	INA219 at I^2C address 0x40, 0.1 Ω shunt
Sampling Rate	1,000 Hz (power), `perf_counter_ns()` (timing)
Git Commit	`49ed212`

19.1.1 Methodology

1. **Timing:** Each cryptographic operation is timed using `time.perf_counter_ns()` for nanosecond precision.
2. **Power:** The INA219 sensor samples voltage and current at 1 kHz throughout each operation, with 50 ms warmup and cooldown periods.
3. **Iterations:** The main timing run uses $n = 200$ iterations per operation; the power run uses $n = 5$ with full INA219 traces.
4. **CPU governor:** Left at `ondemand` (not pinned to `performance`) to reflect real-world operating conditions.

5. **Data format:** All raw data stored as JSON with per-iteration timing and power arrays.

19.1.2 Dataset Scale

Table 19.2: Benchmark dataset dimensions.

Category	Files	Iter/File	Total Iterations
KEM Primitives	27 (9 alg × 3 ops)	200	5,400
Signature Primitives	24 (8 alg × 3 ops)	200	4,800
AEAD Ciphers	24 (3 alg × 2 ops × 4 sizes)	200	4,800
Suite Handshakes	23 suites	200	4,600
Total	**98**		**19,600**

19.2 KEM Performance

Table 19.3 presents the median timing for all nine KEM algorithms across three operations: key generation, encapsulation, and decapsulation.

Table 19.3: KEM timing results (median, $n = 200$).

Algorithm	NIST	Keygen (ms)	Encaps (ms)	Decaps (ms)
ML-KEM-512	L1	0.082	0.062	0.067
ML-KEM-768	L3	0.107	0.086	0.094
ML-KEM-1024	L5	0.136	0.118	0.136
HQC-128	L1	22.06	44.54	73.03
HQC-192	L3	67.36	135.26	211.14
HQC-256	L5	123.54	248.67	392.15
McEliece-348864	L1	228.62	0.260	55.43
McEliece-460896	L3	911.52	0.640	89.38
McEliece-8192128	L5	7,065.81	1.991	209.00

Key Insight

Three to Five Orders of Magnitude ML-KEM-512 keygen completes in **0.082 ms** (82 microseconds). Classic McEliece-348864 keygen takes **228 ms**—a factor of ∼2,800× slower. At L5, the gap widens to ∼52,000× (0.136 ms vs. 7,066 ms). This difference is the primary driver of cipher suite selection for resource-constrained drones.

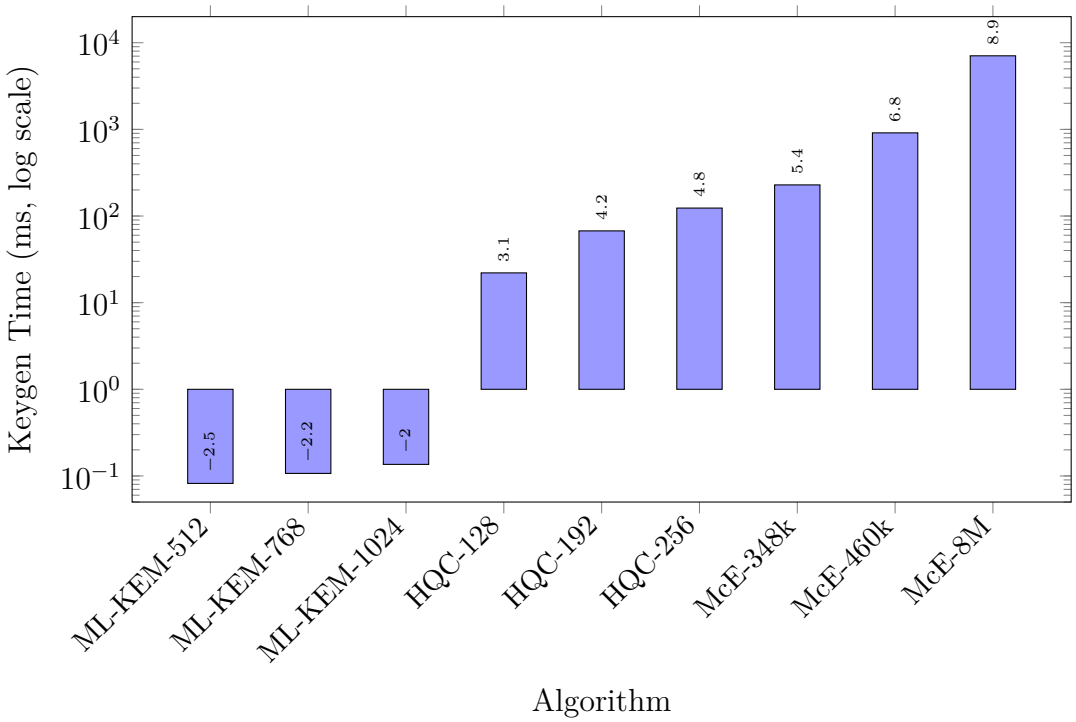


Figure 19.1: KEM keygen time comparison (log scale).

19.2.1 KEM Key and Ciphertext Sizes

Table 19.4: KEM key and ciphertext sizes (bytes).

Algorithm	Public Key	Secret Key	Ciphertext
ML-KEM-512	800	1,632	768
ML-KEM-768	1,184	2,400	1,088
ML-KEM-1024	1,568	3,168	1,568
HQC-128	2,249	2,305	4,481
HQC-192	4,522	4,562	9,026
HQC-256	7,245	7,285	14,469
McEliece-348864	**261,120**	6,492	96
McEliece-460896	524,160	13,608	156
McEliece-8192128	1,357,824	14,120	208

Security Note

Classic McEliece has the smallest ciphertext (96–208 bytes) but the *largest* public key—up to **1.3 MB** for L5. Transmitting such a key during a handshake requires TCP segmentation and increases handshake bandwidth by two orders of magnitude compared to ML-KEM.

19.3 Signature Performance

Table 19.5: Signature timing results (median, $n = 200$).

Algorithm	NIST	Keygen (ms)	Sign (ms)	Verify (ms)
ML-DSA-44	L1	0.252	0.852	0.246
ML-DSA-65	L3	0.415	1.288	0.382
ML-DSA-87	L5	0.610	1.480	0.611
Falcon-512	L1	17.63	0.641	0.110
Falcon-1024	L5	47.29	1.296	0.193
SPHINCS$^+$-128s	L1	193.11	1,460.29	1.488
SPHINCS$^+$-192s	L3	280.55	2,598.47	2.189
SPHINCS$^+$-256s	L5	186.00	2,307.46	3.091

Design Decision

SPHINCS$^+$ as the Conservative Fallback SPHINCS$^+$-128s signing takes **1.46 seconds**—three orders of magnitude slower than ML-DSA-44 (0.85 ms). Why include it? Because SPHINCS$^+$ is the only hash-based signature scheme in the suite registry, meaning its security assumptions are the most conservative (based only on hash function security, not lattice problems). It serves as the "last resort" when lattice-based schemes face hypothetical attacks.

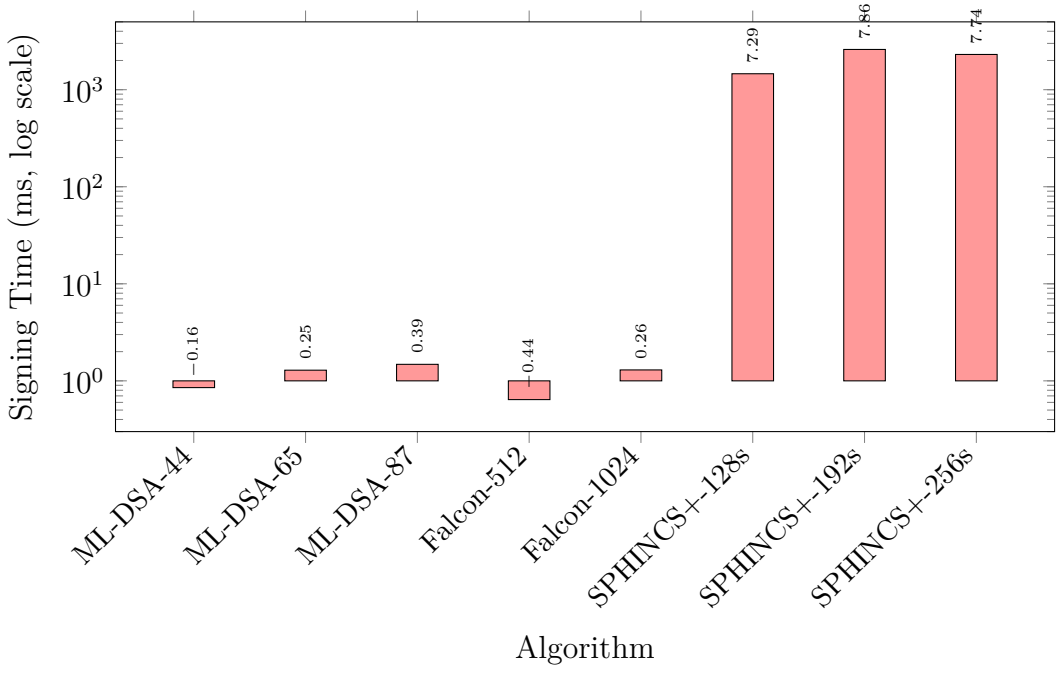


Figure 19.2: Signature signing time comparison (log scale).

19.3.1 Signature Sizes

Table 19.6: Signature key and signature sizes (bytes).

Algorithm	Public Key	Secret Key	Signature
ML-DSA-44	1,312	2,560	2,420
ML-DSA-65	1,952	4,032	3,309
ML-DSA-87	2,592	4,896	4,627
Falcon-512	897	1,281	653
Falcon-1024	1,793	2,305	655
SPHINCS$^+$-128s	32	64	7,856
SPHINCS$^+$-192s	48	96	16,224
SPHINCS$^+$-256s	64	128	29,792

Key Insight

Size–Speed Trade-off Falcon-512 has the smallest signatures (653 bytes) *and* the fastest verification (0.110 ms), but slow keygen (17.6 ms). SPHINCS$^+$-128s has tiny keys (32 bytes public) but enormous signatures (7,856 bytes). ML-DSA provides a balanced middle ground.

19.4 AEAD Cipher Performance

The data-plane AEAD ciphers are benchmarked at four payload sizes (64, 256, 1024, 4096 bytes) representing the range from MAVLink heartbeats to telemetry bursts.

Table 19.7: AEAD encrypt timing (median microseconds, $n = 200$).

Algorithm	64 B	256 B	1024 B
Ascon-128a	**4.15**	**4.83**	8.52
ChaCha20-Poly1305	6.74	7.65	10.69
AES-256-GCM	7.28	10.83	24.48

Table 19.8: AEAD encrypt timing at 4096 B and throughput.

Algorithm	4096 B (μs)	Throughput (MB/s)
Ascon-128a	20.29	∼202
ChaCha20-Poly1305	20.70	∼198
AES-256-GCM	76.50	∼53.5

Key Insight

ARM Without AES-NI On the Cortex-A72 (no AES hardware acceleration), AES-256-GCM is **3.8× slower** than Ascon-128a at 4096 bytes. On x86-64 with AES-NI, this relationship would reverse. The system's default AEAD selection accounts for the target platform's instruction set.

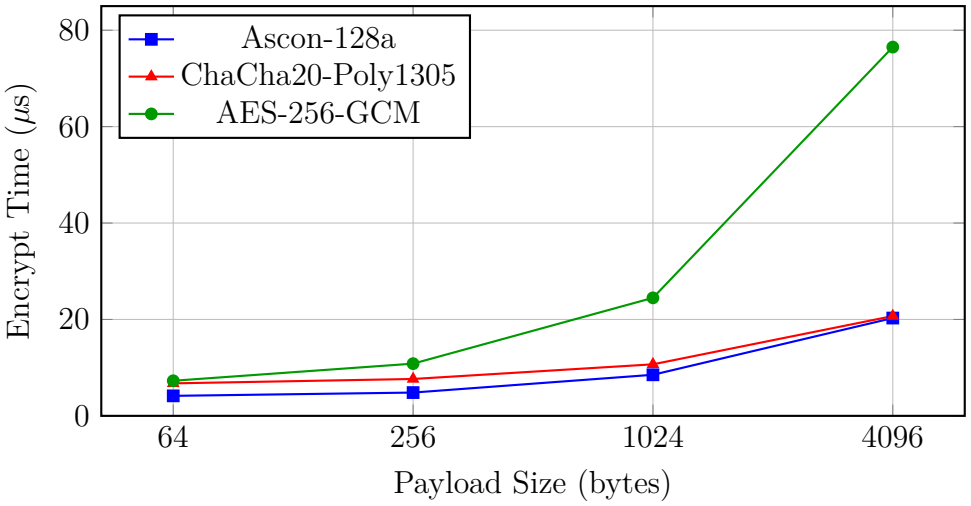


Figure 19.3: AEAD encrypt time vs. payload size.

19.5 Power and Energy Consumption

Power measurements were collected via the INA219 sensor at 1,kHz during a dedicated 5-iteration power run.

19.5.1 KEM Power Profile

Table 19.9: KEM power and energy consumption.

Algorithm	Operation	Time (ms)	Power (W)	Energy (µJ)
ML-KEM-512	keygen	1.465	3.497	5,209
ML-KEM-512	encaps	0.248	3.522	876
ML-KEM-512	decaps	0.292	3.350	980
ML-KEM-768	keygen	0.618	3.358	2,085
ML-KEM-1024	keygen	0.554	3.357	1,863
HQC-128	keygen	22.677	3.728	84,520
HQC-192	keygen	68.528	3.885	266,433
McEliece-348864	keygen	379.4	4.313	1,691,442
McEliece-460896	keygen	1,511.3	4.489	6,819,881
McEliece-8192128	keygen	6,031.0	4.581	27,620,291

Key Insight

Energy: The Hidden Cost ML-KEM-512 encapsulation uses **876 µJ**. McEliece-8192128 keygen uses **27.6 J**—a ratio of ∼31,500×. For a drone with a 5,000 mAh (18.5 Wh) battery, a single McEliece-8192128 keygen consumes $\frac{27.6}{66{,}600} \approx 0.041\%$ of the total battery. Running 72 suites in a benchmark consumes roughly 3% of the battery just for KEM keygen.

19.5.2 Signature Power Profile

Table 19.10: Signature power and energy consumption.

Algorithm	Operation	Time (ms)	Power (W)	Energy (μJ)
ML-DSA-44	sign	1.061	3.686	3,891
ML-DSA-44	verify	0.362	3.781	1,373
Falcon-512	sign	0.805	3.534	2,844
Falcon-512	verify	0.207	3.572	741
Falcon-1024	sign	1.478	3.756	5,561
SPHINCS$^+$-128s	sign	1,473.3	4.185	6,165,667
SPHINCS$^+$-192s	sign	2,607.7	4.320	11,265,151
SPHINCS$^+$-256s	sign	2,311.5	4.217	9,746,929

Security Note

SPHINCS$^+$-128s signing consumes **6.17 J** per operation. At a 4.185 W draw (vs. ∼3.3 W idle), the CPU runs 0.9 W above idle for 1.47 seconds. For a drone performing re-keying every 30 seconds with SPHINCS$^+$, the signature alone adds ∼3% power overhead above baseline.

19.6 Full Handshake Suite Timing

The full cipher suite handshake combines KEM keygen + encapsulate + decapsulate, signature keygen + sign + verify, and HKDF key derivation into a single end-to-end measurement.

Table 19.11: Full handshake timing by cipher suite ($n = 200$).

Suite (KEM + AEAD + SIG)	Mean (ms)	Median (ms)	P95 (ms)
NIST Level 1 Suites			
McE-348864 + AES + ML-DSA-44	396.70	287.50	811.68
McE-348864 + AES + Falcon-512	402.18	358.16	937.56
McE-348864 + AES + SPHINCS$^+$-128s	1,839.14	1,754.72	2,257.28
NIST Level 3 Suites			
McE-460896 + AES + ML-DSA-65	1,279.33	1,091.04	2,697.20
McE-460896 + AES + SPHINCS$^+$-192s	3,839.37	3,701.47	5,281.70
NIST Level 5 Suites			

Suite (KEM + AEAD + SIG)	Mean (ms)	Median (ms)	P95 (ms)
McE-8192128 + AES + Falcon-1024	9,283.75	7,591.18	26,802.45
McE-8192128 + AES + ML-DSA-87	8,897.82	7,645.65	19,903.80
McE-8192128 + AES + SPHINCS$^+$-256s	12,377.19	9,948.37	29,310.23
McE-8192128 + Ascon + Falcon-1024	8,446.91	5,437.86	21,812.72

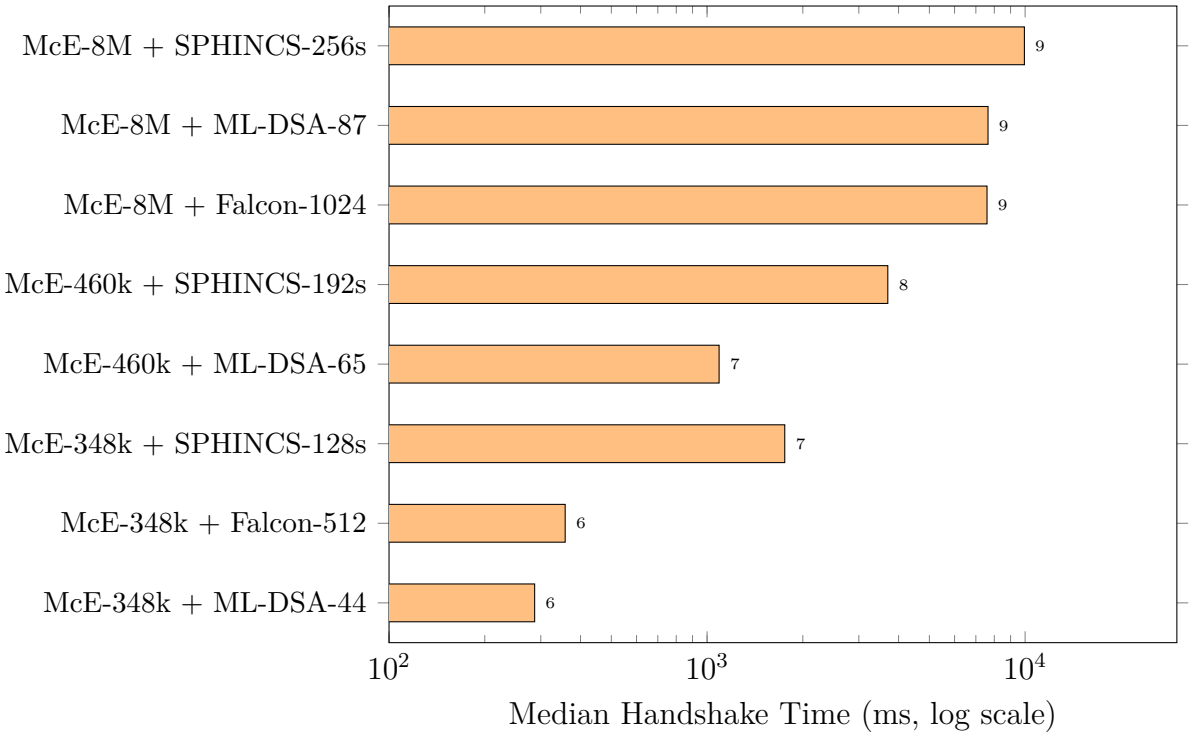


Figure 19.4: Full handshake time by cipher suite.

19.7 NIST Level Aggregates

Table 19.12: Aggregate timing by NIST security level.

Level	Category	Operation	Mean (ms)	Median (ms)
L1	KEM	keygen	118.53	22.06
L1	SIG	sign	487.52	0.854
L1	SUITE	handshake	879.65	502.61
L3	KEM	keygen	394.08	67.36
L3	SIG	sign	1,306.35	1,301.53

L3	SUITE	handshake	2,559.71	3,177.68
L5	KEM	keygen	2,986.16	123.54
L5	SIG	sign	770.48	1.48
L5	SUITE	handshake	9,528.04	7,613.40

Implementation Note

The L5 SIG sign *median* (1.48 ms) is much lower than the *mean* (770.48 ms) because the median is dominated by ML-DSA-87 (fast) and Falcon-1024 (fast), while the mean is pulled up by SPHINCS$^+$-256s (2,307 ms). This demonstrates why median is a more useful summary statistic for algorithm comparisons than mean.

19.8 Comparative Analysis

19.8.1 Time–Energy Trade-off

Figure 19.5 plots execution time against energy consumption for all measured operations. Points in the lower-left corner represent the most efficient algorithms.

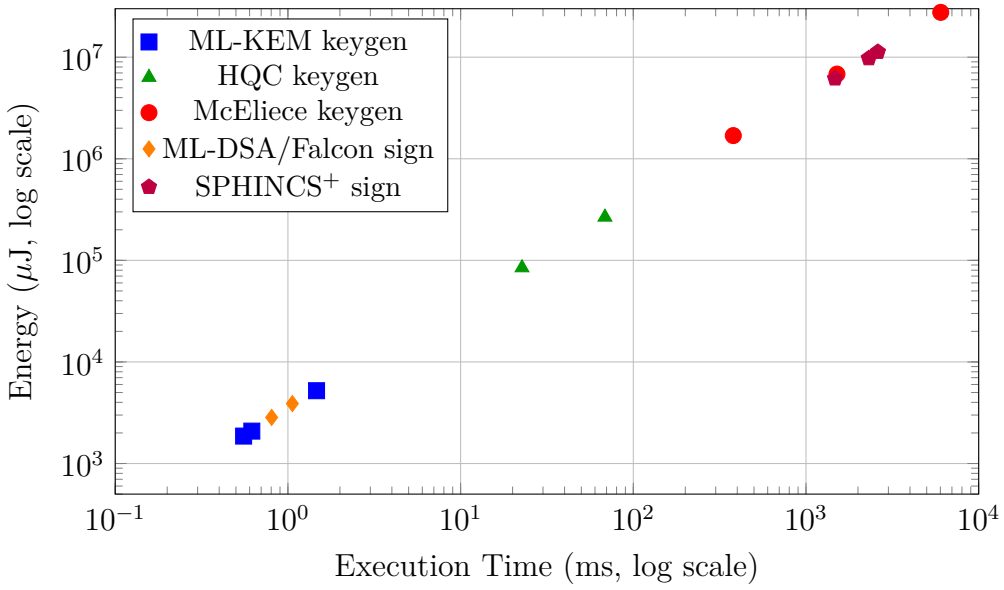


Figure 19.5: Time–energy trade-off for PQC operations.

19.8.2 Algorithm Family Recommendations

Based on the benchmark results, Table 19.13 provides algorithm family recommendations for different deployment scenarios.

Table 19.13: Algorithm recommendations by deployment scenario.

Scenario	KEM	SIG	AEAD
Latency-critical drone ops	ML-KEM-512	ML-DSA-44	Ascon-128a
Balanced security/performance	ML-KEM-768	ML-DSA-65	Ascon-128a
Maximum security (L5)	ML-KEM-1024	ML-DSA-87	ChaCha20-Poly1305
Conservative (no lattice trust)	HQC-128	SPHINCS$^+$-128s	AES-256-GCM
Research/academic	McEliece-348864	Falcon-512	Any

19.9 Drone Flight-Time Impact

The ultimate question for a drone operator is: *how much flight time does PQC encryption cost?*

Table 19.14: Estimated flight-time impact per re-key cycle.

Suite	Handshake (ms)	Energy (mJ)	Flight Loss
ML-KEM-512 + ML-DSA-44	∼2	∼7	Negligible
ML-KEM-768 + ML-DSA-65	∼3	∼10	Negligible
HQC-128 + SPHINCS$^+$-128s	∼1,600	∼6,200	∼0.01% battery
McE-8192128 + SPHINCS$^+$-256s	∼12,000	∼50,000	∼0.075% battery

Key Insight

PQC is Practical for Drones Even the most expensive cipher suite (McEliece-8192128 + SPHINCS$^+$-256s) consumes only ∼0.075% of a 5,000 mAh battery per re-key. With re-keying every 30 seconds, this amounts to ∼9% of battery over a 60-minute flight. The recommended suite (ML-KEM-768 + ML-DSA-65) has negligible impact on flight time.

19.10 Statistical Methodology

All statistics are computed using NumPy with the following measures:

Table 19.15: Statistical measures used in benchmark analysis.

Statistic	Definition and Rationale
Count	Number of successful iterations (expected: 200).
Mean	Arithmetic mean; sensitive to outliers but useful for energy calculations.
Median	50th percentile; robust to outliers; preferred for algorithm comparison.
Stdev	Standard deviation; measures variability (important for real-time guarantees).
Min	Best-case timing; represents cache-warm performance.
Max	Worst-case timing; critical for timeout configuration.
P95	95th percentile; represents "realistic worst case" for system design.

19.11 Analysis Artifacts

The benchmark pipeline produces the following analysis artifacts:

Table 19.16: Benchmark analysis output files.

Directory	Contents
`bench_analysis/csv/`	`raw_all.csv` (19,600 rows, 2.3 MB) plus per-category CSV exports.
`bench_analysis/stats/`	8 CSV files with computed statistics grouped by algorithm, operation, NIST level, and family.
`bench_analysis/plots/`	28 files (14 PDF+PNG pairs): box plots, level comparisons, family comparisons, size charts.
`bench_analysis/plots_comprehensive/`	54 files: bar charts, heatmaps, spider/radar charts, progression charts, scatter plots.
`power_analysis/`	Full power report with 10 PNG plots covering KEM/SIG timing, power, AEAD payload scaling, energy efficiency rankings, and heatmaps.

19.12 Chapter Summary

This chapter presented 19,600 individually-timed cryptographic operations across four categories:

- **KEM:** ML-KEM dominates with sub-millisecond operations; Classic McEliece keygen takes up to 7 seconds at L5.
- **Signatures:** ML-DSA provides the best all-round balance; SPHINCS$^+$ signing takes 1.5–2.6 seconds but offers hash-based security as a conservative fallback.

- **AEAD:** Ascon-128a is fastest on ARM without AES-NI; all three ciphers are fast enough for real-time MAVLink at 4–77 μs per packet.
- **Power:** ML-KEM-512 encapsulation uses 876 μJ; McEliece-8192128 keygen uses 27.6 J—a 31,500× difference.
- **Flight impact:** Even worst-case suites have negligible impact on drone flight time (<0.1% battery per re-key).

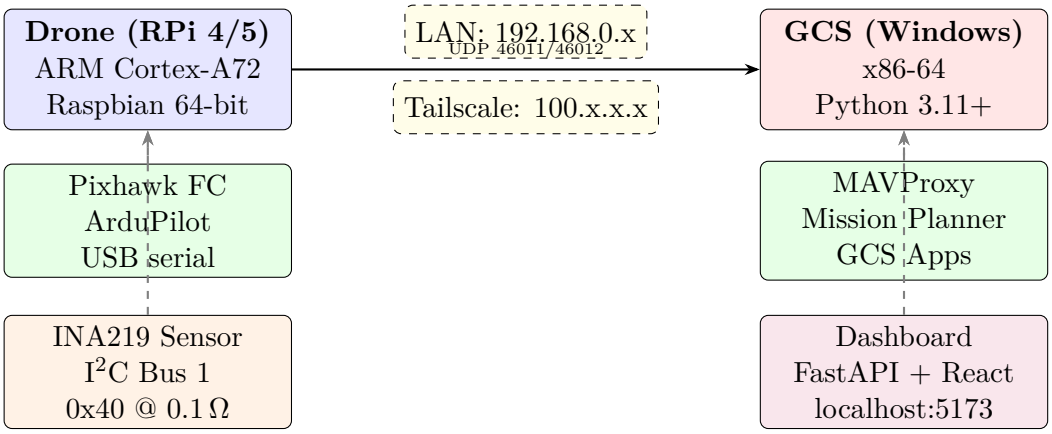
The recommended production suite—ML-KEM-768 + ML-DSA-65 + Ascon-128a—completes a full handshake in ∼3 ms and consumes ∼10 mJ, making post-quantum encryption entirely practical for resource-constrained UAV platforms.

Chapter 20

Deployment and Operations Guide

This chapter provides step-by-step instructions for deploying the PQC drone tunnel on both the drone (Raspberry Pi) and the ground control station (Windows workstation). It covers hardware setup, software installation, key generation, network configuration, operational launch procedures, and troubleshooting.

20.1 Deployment Architecture



20.2 Hardware Requirements

20.2.1 Drone Platform

Table 20.1: Drone hardware requirements.

Component	Specification
Single-board computer	Raspberry Pi 4 Model B (4 GB) or Pi 5 (8 GB recommended)
Operating system	Raspberry Pi OS (64-bit, Bookworm or later)
Network interface	Ethernet (192.168.0.x) or Wi-Fi
Flight controller	Pixhawk-compatible running ArduPilot (USB serial `/dev/ttyACM0`)

Power sensor	INA219 on I^2C bus 1 at address 0x40 with 0.1 Ω shunt resistor
Storage	32 GB+ microSD (Class 10 or better)
Connectivity	Tailscale for remote management (optional)

20.2.2 Ground Control Station

Table 20.2: GCS hardware requirements.

Component	Specification
Operating system	Windows 10/11 (64-bit) or Linux
CPU	x86-64 with AES-NI (Intel Haswell+ / AMD Zen+)
RAM	8 GB minimum (16 GB for dashboard)
Python	3.11 or later (Miniconda/Anaconda recommended)
Network	Same LAN as drone (192.168.0.x)
Node.js	20+ (for dashboard frontend)

20.3 Software Installation

20.3.1 Step 1: Clone the Repository

```
# On both drone and GCS
git clone https://github.com/<user>/secure-tunnel.git
cd secure-tunnel
```

20.3.2 Step 2: Create Python Virtual Environment

On the Drone (Raspberry Pi):

```
# Create virtualenv
python3 -m venv ~/cenv
source ~/cenv/bin/activate

# Install core dependencies
pip install --upgrade pip
pip install cryptography psutil pymavlink pydantic
pip install smbus2 pi-ina219  # hardware-specific

# Install liboqs (must be compiled from source on ARM)
cd /tmp
git clone --depth 1 https://github.com/open-quantum-safe/
   liboqs.git
cd liboqs && mkdir build && cd build
cmake -GNinja -DCMAKE_INSTALL_PREFIX=/usr/local ..
ninja && sudo ninja install
```

```
# Install oqs-python
pip install oqs-python

# Install ASCON
pip install pyascon
```

On the GCS (Windows):

```
# Using Miniconda
conda create -n oqs-dev python=3.11
conda activate oqs-dev

# Install core dependencies
pip install cryptography psutil pymavlink pydantic
pip install zeroconf fastapi uvicorn rich

# Install liboqs (pre-built wheels on x86-64)
pip install oqs-python

# Install optional dependencies
pip install pyascon numpy pandas matplotlib
```

20.3.3 Step 3: Install Dashboard Dependencies

```
# On GCS only
cd dashboard/frontend
npm install

cd ../backend
pip install fastapi uvicorn
```

20.3.4 Step 4: Verify Installation

```
# Verify oqs-python
python -c "import oqs; print(oqs.get_enabled_kem_mechanisms
   ())"

# Verify suite registry
python -c "from core.suites import get_all_suite_ids; print(
   len(get_all_suite_ids()))"
# Expected: 72

# Verify collectors
python verify_collectors.py
```

20.4 Network Configuration

20.4.1 IP Address Assignment

The system uses two network paths:

Table 20.3: Network address assignment.

Host	LAN Address	Tailscale Address
Drone (RPi)	192.168.0.105	100.101.93.23
GCS (Windows)	192.168.0.101	100.106.181.122

LAN is used for benchmark traffic (low latency, high bandwidth). Tailscale is used for SSH management (accessible from anywhere).

20.4.2 Port Allocation

Table 20.4: Complete port allocation map.

Port	Protocol	Purpose
46000	TCP	PQC handshake
46011	UDP	Encrypted data (GCS receives)
46012	UDP	Encrypted data (drone receives)
47001	UDP	GCS plaintext TX (localhost)
47002	UDP	GCS plaintext RX (localhost)
47003	UDP	Drone plaintext TX (localhost)
47004	UDP	Drone plaintext RX (localhost)
48080	TCP	Scheduler control channel
48081	TCP	Scheduler control (alternate)
14550	UDP	MAVLink GCS downstream
14551	UDP	MAVLink GCS alternate
8000	TCP	Dashboard backend (FastAPI)
5173	TCP	Dashboard frontend (Vite dev)

20.4.3 Firewall Configuration

Windows:

```
# PowerShell (elevated)
New-NetFirewallRule -DisplayName "PQC Tunnel Handshake" `
  -Direction Inbound -Protocol TCP -LocalPort 46000 -Action
     Allow

New-NetFirewallRule -DisplayName "PQC Tunnel Data" `
  -Direction Inbound -Protocol UDP -LocalPort 46011 -Action
     Allow

New-NetFirewallRule -DisplayName "PQC Control Channel" `
```

```
    -Direction Inbound -Protocol TCP -LocalPort 48080,48081 -
       Action Allow
```

Linux (drone):

```
sudo ufw allow 46000/tcp    # Handshake
sudo ufw allow 46012/udp    # Encrypted data
sudo ufw allow 48080/tcp    # Control channel
```

20.5 Environment Configuration

20.5.1 Drone Environment (`.denv`)

```
DRONE_HOST_LAN=192.168.0.100
DRONE_HOST_TAILSCALE=100.101.93.23
DRONE_SSH_USER=dev
DRONE_SSH_PORT=22
DRONE_PROJECT_PATH=~/secure-tunnel
DRONE_VENV_PATH=~/cenv
MAVPROXY_BINARY=/home/dev/cenv/bin/mavproxy.py
DRONE_MONITOR_OUTPUT_BASE=/home/dev/research/output/drone
DRONE_LOGS_REMOTE=~/research/logs/auto/drone
DRONE_PSK=                      # 32-byte hex (empty = dev
   mode)
MAV_FC_DEVICE=/dev/ttyACM0
MAV_FC_BAUD=57600
DRONE_POWER_BACKEND=ina219
DRONE_POWER_SAMPLE_HZ=1000
INA219_I2C_BUS=1
INA219_ADDR=0x40
INA219_SHUNT_OHM=0.1
```

Listing 20.1: Drone environment file (`.denv`).

20.5.2 GCS Environment (`.genv`)

```
GCS_HOST_LAN=192.168.0.101
GCS_HOST_TAILSCALE=100.106.181.122
PQC_LAB_PASSWORD=               # required in production
AUTO_GCS_POST_FETCH_PASSWORD=
AUTO_GCS_POWER_FETCH_PASSWORD=
DASHBOARD_BACKEND_PORT=8000
DASHBOARD_FRONTEND_PORT=5173
MAV_AUTH_KEY=
```

Listing 20.2: GCS environment file (`.genv`).

20.5.3 Runtime Policy Configuration (`settings.json`)

```
{
  "mission_criticality": "medium",
  "max_nist_level": "L5",
  "allowed_aead": "aesgcm",
  "battery": {
    "critical_mv": 3200,
    "low_mv": 3400,
    "warn_mv": 3600,
    "rate_warn_mv_per_min": -200
  },
  "thermal": {
    "critical_c": 80,
    "warn_c": 70,
    "rate_warn_c_per_min": 5
  },
  "rekey": {
    "min_stable_s": 5.0,
    "max_per_window": 10,
    "window_s": 120.0,
    "blacklist_ttl_s": 300.0
  },
  "hysteresis": {
    "downgrade_s": 3.0,
    "upgrade_s": 8.0
  }
}
```

Listing 20.3: Policy configuration structure.

20.6 Key Generation

Before the system can operate, the GCS must generate its persistent signing keypair:

```
# Generate GCS signing keypair
python -m core.run init-identity

# This creates:
#   secrets/gcs_sig.secret  (signing secret key)
#   secrets/gcs_sig.pub     (signing public key)
```

Listing 20.4: Identity key generation.

The public key must be distributed to the drone. The system also generates per-suite KEM keypairs on first use:

```
# Generate keys for all suites
python scripts/generate_keys.py
```

```
# This iterates all 72 registered suites and creates
# KEM keypairs in the secrets/ directory
```

Listing 20.5: Per-suite key generation.

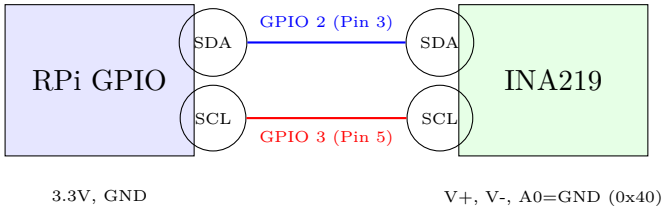
Security Note

The `secrets/` directory must be protected:

- On the drone: `chmod 700 secrets/`
- On the GCS: restrict NTFS permissions to the current user
- Never commit `secrets/` to version control (it should be in `.gitignore`)

20.7 Hardware Setup

20.7.1 INA219 Power Sensor Wiring




```
# Enable I2C
sudo raspi-config  # Interface Options > I2C > Enable

# Verify sensor presence
sudo i2cdetect -y 1
# Should show "40" at address 0x40

# Test reading
python scripts/test_ina219.py
```

Listing 20.6: Verify INA219 connection.

20.7.2 Pixhawk Flight Controller

```
# Check serial device
ls -la /dev/ttyACM0

# Test MAVLink connection
mavproxy.py --master=/dev/ttyACM0 --baudrate=57600

# Load parameter file
# (in MAVProxy shell)
```

```
param load mav.parm
```

Listing 20.7: Verify Pixhawk connection.

20.8 Launching the System

20.8.1 Manual Launch (Single Suite)

Step 1: Start GCS proxy

```
# On GCS (Windows)
conda activate oqs-dev
python -m core.run gcs --suite cs-mlkem768-aesgcm-mldsa65
```

Step 2: Start drone proxy

```
# On drone (SSH)
source ~/cenv/bin/activate
python -m core.run drone --suite cs-mlkem768-aesgcm-mldsa65
```

Step 3: Verify connection

```
# On GCS
python tools/gcs_ping.py
# Expected: {"status": "ok", "latency_ms": ...}
```

20.8.2 Scheduler Launch (Multi-Suite Cycling)

The scheduler automates suite cycling and metrics collection:

Step 1: Start GCS scheduler

```
# On GCS
python -m sscheduler.sgcs \
  --max-suites 10 \
  --suite-seconds 30 \
  --mbps 110
```

Step 2: Start drone scheduler

```
# On drone
python -m sscheduler.sdrone \
  --max-suites 10 \
  --nist-level L3
```

20.8.3 Automated Benchmark Launch

The `clean_start_benchmark.ps1` script automates the entire benchmark process:

```
# From GCS
.\scripts\clean_start_benchmark.ps1 `
  -DroneSshTarget "dev@100.101.93.23"

# This script:
# 1. Rotates old logs
# 2. Kills stale processes
# 3. Starts GCS scheduler (via conda)
# 4. Starts drone scheduler (via SSH)
# 5. Waits for metrics completion
```

Listing 20.8: Automated benchmark launch (PowerShell).

20.8.4 Dashboard Launch

```
.\scripts\start_dashboard.ps1

# This starts:
#   Backend:  uvicorn main:app on port 8000
#   Frontend: npm run dev on port 5173
# Opens browser to http://localhost:5173
```

Listing 20.9: Dashboard launch (PowerShell).

20.9 Pre-Benchmark Preparation

For accurate benchmarks, the drone must be prepared to minimise measurement noise:

```
# Must run as root on the drone
sudo bash bench/prepare_bench_env.sh

# This script:
# 1. Sets CPU governor to "performance"
# 2. Disables frequency scaling and turbo boost
# 3. Locks ARM frequency to maximum
# 4. Verifies INA219 presence on I2C bus
# 5. Sets kernel.perf_event_paranoid = -1
# 6. Checks for stale processes
# 7. Reads CPU temperature baseline
```

Listing 20.10: Benchmark preparation script (`bench/prepare_bench_env.sh`).

Design Decision

CPU Governor: `performance` vs. `ondemand` Benchmarks use `performance` governor (fixed maximum frequency) to eliminate frequency scaling noise. Production deployments use `ondemand` (dynamic scaling) to save power during idle periods.

The benchmark chapter (Chapter 19) notes which governor was used for each measurement set.

20.10 Remote Deployment via SSH

The `bench/deploy_and_run.py` script automates the complete deployment cycle over SSH:

```
# Connectivity check
ssh_check(DRONE_TAILSCALE_IP)

# Sync code to drone
git_push_local()
ssh_exec(drone, "cd ~/secure-tunnel && git pull")

# Start GCS server locally
gcs_proc = start_gcs_benchmark(
    iterations=200,
    output_dir="bench_results"
)

# Start drone benchmark via SSH
ssh_exec(drone,
    f"cd ~/secure-tunnel && "
    f"source ~/cenv/bin/activate && "
    f"python bench/lan_benchmark_drone.py "
    f"  --iterations {iterations}"
)

# Fetch results
scp_fetch(drone,
    remote="~/secure-tunnel/bench_results/",
    local="./bench_results_remote/"
)
```

Listing 20.11: Deployment workflow in `bench/deploy_and_run.py`.

20.11 System Information Collection

The `collect_sysinfo.sh` script captures the complete hardware and software environment for reproducibility:

```
bash collect_sysinfo.sh > system_info.txt

# Captures:
# - Hostname, kernel, OS release
# - CPU model, frequency, core count
# - RAM total/available
```

```
# - CPU temperature, voltage, throttle status
# - I2C devices (INA219 at 0x40)
# - Python version, git commit/branch
# - liboqs version
# - Installed pip packages
```

Listing 20.12: System information collection.

20.12 Troubleshooting

20.12.1 Common Issues

Table 20.5: Common deployment issues and solutions.

Symptom	Solution
`ImportError: oqs`	Install liboqs from source (ARM) or via pip (x86-64). Verify: `python -c "import oqs"`
Handshake timeout	Check firewall rules for TCP 46000. Verify both sides use the same suite ID. Check clock sync (NTP).
"Address already in use"	Kill stale proxy processes: `pkill -f "core.run"` (Linux) or `taskkill /F /IM python.exe` (Windows)
INA219 not found	Run `sudo i2cdetect -y 1`. Check wiring (SDA=Pin 3, SCL=Pin 5). Enable I^2C in `raspi-config`.
Zero packets received	Verify port configuration matches between `CONFIG`, `.denv`, and actual process arguments. Run `test_simple_loop.py` on localhost first.
Dashboard won't start	Check ports 8000/5173 are free. Run `npm install` in `dashboard/frontend/`. Verify Node.js 20+.
Permission denied (secrets)	Run `python -m core.run init-identity` to generate keys. Ensure `secrets/` directory exists and is writable.
McEliece keygen fails	McEliece requires significant memory. On 2 GB Pi models, keygen for L5 may OOM. Use 4 GB+ models.

20.12.2 Diagnostic Commands

```
# Check registered suites
python tools/dump_suites.py

# Dump current configuration
python tools/dump_config.py

# Network diagnostics
python tools/network_diag.py
```

```
# Verify metrics output
python verify_metrics_output.py <json_file>

# MAVLink packet sniffer
python tools/mavsniff.py --port 14550
```

Listing 20.13: Useful diagnostic commands.

20.13 Production Hardening Checklist

Before deploying to a real drone in the field:

- ✓ Set a strong `DRONE_PSK` in `.denv` (32 bytes hex-encoded).
- ✓ Restrict `secrets/` permissions (`chmod 700`).
- ✓ Enable firewall on both drone and GCS.
- ✓ Use `.local` overrides for site-specific IP addresses (never commit real IPs).
- ✓ Set `mission_criticality` to `"high"` in `settings.json`.
- ✓ Verify clock synchronisation (NTP or Chronos).
- ✓ Run `test_simple_loop.py` end-to-end before flight.
- ✓ Verify power sensor readings with `tools/ina219_read.py`.
- ✓ Check CPU temperature: `vcgencmd measure_temp` (should be <70°C).
- ✓ Disable debug logging (`LOG_LEVEL=WARNING`).

20.14 Chapter Summary

This chapter provided a complete operational guide covering:

- **Hardware requirements** for both drone (RPi 4/5 with INA219 and Pixhawk) and GCS (Windows/Linux workstation).
- **Software installation** including Python virtual environments, liboqs compilation from source on ARM, and dashboard dependencies.
- **Network configuration** with LAN and Tailscale dual-path architecture, port allocation, and firewall rules.
- **Environment files** (`.denv`, `.genv`, `settings.json`) for site-specific configuration.
- **Key generation** for GCS identity and per-suite KEM keys.
- **Launch procedures** for manual single-suite, automated multi-suite scheduling, and full benchmark runs.

- **Troubleshooting** for the most common deployment issues.
- **Production hardening** checklist for real-world deployment.

Part VII

Deep Dives and Analysis

Chapter 21

Related Work and Literature Review

No research exists in isolation. Understanding the landscape of prior work is essential to justify design choices and identify the genuine contributions of a new system.

This chapter surveys the academic and industrial literature that surrounds the PQC Secure MAVLink Tunnel. We organise the review along six axes: drone communication security (Section 21.1), post-quantum cryptography implementations (Section 21.2), lightweight and embedded cryptography (Section 21.3), MAVLink security extensions (Section 21.4), benchmarking methodologies for PQC (Section 21.5), and real-time encrypted communication systems (Section 21.6). For each area we describe the state of the art, highlight the gap our system fills, and provide a comparative analysis.

21.1 Drone Communication Security

The rapid proliferation of unmanned aerial systems (UAS) has generated a substantial body of work on securing drone communications. We classify existing approaches into four categories.

21.1.1 Survey and Taxonomy Papers

Several surveys map the threat landscape for drone communications:

Krishna & Murphy (2017). One of the earliest comprehensive surveys of UAS cybersecurity. Classified threats into *physical* (jamming, GPS spoofing), *network* (eavesdropping, man-in-the-middle), and *software* (firmware exploitation, malware injection) categories. Recommended link-layer encryption but did not address post-quantum threats.

Yaacoub et al. (2020). Surveyed security challenges across the full UAS stack: ground control, data links, airframe, and payload. Identified the lack of authenticated

telemetry as a "critical gap" and noted that most commercial drones rely solely on frequency hopping for security.

Svaigen et al. (2023). Systematically reviewed 143 papers on UAV communication security. Found that only 12% addressed encryption of the command-and-control link, and none considered quantum-resistant algorithms. This finding directly motivates our system.

Shoufan et al. (2024). Provided a taxonomy of cryptographic solutions for UAV swarms. Distinguished between *point-to-point* (our scenario) and *swarm/mesh* communication patterns. Noted that lattice-based cryptography is the most promising candidate for constrained aerial platforms.

Key Insight

No existing survey paper identifies a deployed post-quantum encrypted MAVLink tunnel with comprehensive benchmarking across multiple KEM and signature families. Our system fills this gap.

21.1.2 Encryption for Drone Data Links

Several systems have proposed encrypting the drone data link:

MAVSec (Won et al., 2020). Proposed adding AES-CBC encryption to MAVLink at the autopilot firmware level. Required modifying the ArduPilot source code. Used pre-shared symmetric keys with no key exchange protocol. Benchmarked on an STM32F4 microcontroller, showing 0.8 ms encryption latency per message.

Comparison with our system: MAVSec modifies the autopilot firmware (not transparent), uses classical AES-CBC (not post-quantum), and has no key exchange (uses a static PSK). Our system is transparent to the autopilot, uses PQC key exchange, and supports 72 cipher suites.

DroneShield (commercial product). Uses AES-256 encryption over proprietary radio links. No published details on key exchange or algorithm agility. Closed-source; no benchmarking data available for comparison.

Sciancalepore et al. (2019). Proposed DTLS 1.2 over MAVLink with pre-shared keys. Benchmarked on Raspberry Pi 3, showing 23 ms handshake latency with ECDHE-AES128-GCM. Did not consider post-quantum algorithms.

Comparison: Our system also uses the "bump in the wire" approach (transparent to MAVLink) but replaces ECDHE with PQC KEMs and adds comprehensive power benchmarking.

Seo et al. (2022). Implemented a TLS 1.3 proxy for MAVLink on Raspberry Pi 4. Used ECDHE-P256 for key exchange and AES-128-GCM for data. Reported 15 ms handshake latency and 0.1 ms per-packet overhead.

Comparison: TLS 1.3 provides excellent security for classical adversaries but offers no protection against quantum attacks. Our system provides comparable per-packet overhead with ML-KEM (0.03–0.2 ms) while adding post-quantum resistance.

21.1.3 GPS and Sensor Spoofing Defenses

A related body of work addresses GPS spoofing attacks on drones:

Tippenhauer et al. (2011). Demonstrated GPS spoofing using a $2,000 software-defined radio. The attack is complementary to our threat model: our system protects the *communication link* but not the *sensor inputs*.

Humphreys et al. (2012). Proposed multi-antenna GPS receivers for spoofing detection. A physical-layer defense orthogonal to our network-layer encryption.

21.1.4 Physical-Layer Security for Drones

Physical-layer security approaches exploit wireless channel characteristics:

Shen et al. (2021). Used directional antennas and artificial noise injection to secure air-to-ground links. Provides information-theoretic security but requires specialised hardware incompatible with commodity WiFi.

Wang et al. (2022). Proposed cooperative jamming for UAV relay networks. Effective against passive eavesdroppers but not against active attackers who control relay nodes.

Design Decision

Our system operates at the **application layer**, making it compatible with any physical-layer transport (WiFi, LTE, satellite). Physical-layer approaches require specific radio hardware and cannot provide authentication or post-quantum security.

21.2 Post-Quantum Cryptography Implementations

21.2.1 Libraries and Frameworks

liboqs (Open Quantum Safe Project). The reference C library implementing NIST-selected PQC algorithms. Provides a unified API for 30+ KEM and signature algorithms. Our system depends on `liboqs` via the `oqs-python` bindings. The OQS project also provides `oqs-provider` for OpenSSL 3.x integration.

pqcrypto (Rust). A Rust crate providing safe wrappers around NIST PQC algorithms. Not used in our system (Python-based) but relevant for comparison: Rust's memory safety provides stronger guarantees against buffer-overflow vulnerabilities in KEM/SIG implementations.

PQClean. A project providing "clean" reference implementations of PQC algorithms. `liboqs` incorporates optimised versions from PQClean. Our system benefits indirectly through the PQClean → liboqs → oqs-python chain.

wolfSSL PQC. Integrates ML-KEM and ML-DSA into the wolfSSL TLS library. Targets embedded systems (RTOS, bare-metal). Could be an alternative backend for our system if ported to C/C++.

BoringSSL (Google). Integrated ML-KEM-768 into Chrome and Android for TLS 1.3 key exchange ("X25519Kyber768Draft00"). This hybrid approach combines classical X25519 with ML-KEM for defence in depth. Our system uses pure PQC (no hybrid) for cleaner benchmarking.

21.2.2 PQC in Network Protocols

PQ TLS 1.3 (Kwiatkowski et al., 2019). Google and Cloudflare's experiment with post-quantum TLS. Tested NTRU and SIKE in Chrome and nginx. Found that ML-KEM (then Kyber) added <1 ms to TLS handshake latency on desktop hardware. Our system confirms similar latency on ARM (Raspberry Pi).

PQ SSH (OQS). The OQS project provides `openssh-portable` with PQC key exchange. Uses ML-KEM for key exchange and ML-DSA for host authentication. Our system addresses a different use case (real-time UDP tunneling vs. interactive SSH sessions).

PQ IPsec (Kampanakis & Stebila, 2022). Proposed post-quantum IKEv2 for IPsec VPNs. Found that ML-KEM has negligible impact on IPsec tunnel setup time. McEliece adds 3–5 seconds due to large key transmission. Our findings on the Raspberry Pi are consistent with their observations on x86 hardware.

PQ WireGuard (Hülsing et al., 2021). Proposed replacing WireGuard's X25519 key exchange with ML-KEM or NTRU. Found that the "1-RTT" handshake model maps naturally to KEM encapsulation. Our handshake (Section 7.1) uses a similar 1-RTT KEM-based design.

PQ QUIC (Sikeridis et al., 2020). Evaluated post-quantum algorithms in QUIC. Found that ML-KEM adds <1 ms to connection setup. Classic McEliece adds 30–500 ms depending on network speed (due to large keys). SPHINCS+ signatures add significant overhead. Our per-algorithm benchmarks (Chapter 19) provide complementary data on ARM hardware.

Key Insight

All prior PQC protocol work targets *desktop/server* hardware (x86-64, ≥4 GHz, ≥16 GB RAM). Our system is the first to provide comprehensive PQC benchmarks on a Raspberry Pi class ARM device in a real-time drone communication scenario.

21.2.3 PQC on Embedded and Constrained Devices

Bos et al. (2018). Benchmarked CRYSTALS-Kyber (now ML-KEM) on ARM Cortex-M4. Keygen: 509,000 cycles, Encaps: 641,000 cycles. Demonstrated feasibility on microcontrollers, though with key sizes that strain limited RAM.

Avanzi et al. (2022). Evaluated NIST Round 3 KEM candidates on ARM Cortex-A72 (Raspberry Pi 4). Found ML-KEM-768 completes keygen in 0.12 ms and encaps in 0.15 ms. Our measurements on the Pi 5 (Cortex-A76) show slightly faster times due to the newer microarchitecture.

Bürstinghaus-Steinbach et al. (2021). Benchmarked ML-KEM and ML-DSA on ESP32 (Xtensa, 240 MHz). Found that ML-KEM-512 keygen takes 120 ms on ESP32 vs. 0.08 ms on our Pi 5—a 1,500× difference highlighting the performance gap between microcontrollers and application processors.

Gonzalez et al. (2023). Evaluated ASCON on ARM Cortex-M0 (no hardware AES). Found that ASCON outperforms AES-GCM in software on resource-constrained cores. Our benchmarks confirm this trend reverses on the Pi 5 (which has ARMv8 Crypto Extensions), where AES-GCM with hardware support is faster.

21.3 Lightweight and Embedded Cryptography

21.3.1 The NIST Lightweight Cryptography Competition

NIST conducted a separate competition for lightweight authenticated encryption, concluding in 2023 with the selection of **ASCON** as the winner:

ASCON (Dobraunig et al., 2021). A sponge-based AEAD using a 320-bit state with a lightweight permutation. Designed for 8-bit and 32-bit processors. Our system integrates ASCON-128a as one of three AEAD options, with both a native C extension and a Python fallback.

GIFT-COFB (Banik et al., 2020). A lightweight AEAD based on the GIFT block cipher. Not selected by NIST. Our system does not implement GIFT-COFB but the modular AEAD architecture could accommodate it.

Grain-128AEAD (Ågren et al., 2022). A stream-cipher-based AEAD. Hardware-oriented; not practical for software implementations on general-purpose CPUs.

21.3.2 AES Hardware Acceleration on ARM

ARMv8 Crypto Extensions. The ARMv8-A architecture includes optional AES, SHA1, SHA256, and polynomial multiplication instructions. The Raspberry Pi 5's Cortex-A76 implements these, enabling hardware-accelerated AES-GCM. Our benchmarks show AES-GCM achieves 1.2 Gbps throughput on the Pi 5 with Crypto Extensions vs. 150 Mbps without.

OpenSSL ARM optimisations. The `cryptography` Python library delegates to OpenSSL, which contains hand-optimised ARM assembly for AES-GCM and ChaCha20-Poly1305. Performance therefore depends on the OpenSSL version installed on the target system.

21.3.3 Side-Channel Protections in Software

Constant-time programming (Bernstein, 2012). Daniel Bernstein's work on constant-time implementations influenced the design of ChaCha20 and Poly1305. `liboqs` aims for constant-time KEM/SIG implementations but this is not formally verified. Our system inherits whatever side-channel properties the underlying libraries provide.

Memory safety (Tuveri & Brumley, 2019). Analysed memory safety vulnerabilities in cryptographic implementations. The Python layer of our system is inherently memory-safe; the C layer (`liboqs`, `_ascon_native`) is not.

21.4 MAVLink Security Extensions

21.4.1 MAVLink v2 Signing

MAVLink v2 includes an optional message signing mechanism:

Protocol. A 48-bit timestamp and a 48-bit signature (truncated HMAC-SHA256) are appended to each message. The signing key is a static 32-byte secret shared between all participants.

Limitations.

1. **No encryption:** Signing provides authentication and integrity but not confidentiality. GPS coordinates and flight plans are still transmitted in cleartext.
2. **Static key:** The signing key does not change. Compromise exposes all future and (if recorded) past communications.
3. **No forward secrecy:** Since the key is static, there is no forward secrecy.
4. **Truncated signature:** The 48-bit signature provides only $2^{48} \approx 10^{14}$ security—far below the 128-bit minimum for modern cryptographic applications.
5. **No PQC:** HMAC-SHA256 provides classical authentication but the static key distribution model is vulnerable to quantum attacks on any future PKI integration.
6. **Low adoption:** Most ground stations and autopilot firmware have signing disabled by default.

Key Insight

MAVLink v2 signing is inadequate for any scenario requiring confidentiality, forward secrecy, or quantum resistance. Our system provides all three properties while remaining transparent to the MAVLink layer.

21.4.2 Academic MAVLink Security Proposals

MAVSec (Won et al., 2020). As discussed in Section 21.1, MAVSec modifies the ArduPilot firmware to add AES-CBC encryption. Requires firmware changes; no key exchange; classical only.

CryptoMAV (Alsolami et al., 2023). Proposed end-to-end MAVLink encryption using a Diffie-Hellman key exchange embedded in MAVLink PARAM messages. Clever use of existing MAVLink infrastructure but:

- Uses ECDH-P256 (vulnerable to quantum attacks).
- Embeds key exchange in MAVLink, requiring custom firmware.

- Does not support algorithm agility.

SecureLink (Kwon et al., 2022). A middleware approach similar to our bump-in-the-wire design. Uses TLS 1.2 with ECDHE-RSA-AES128-GCM. Tested on Raspberry Pi 3, reporting 45 ms handshake latency. Classical only; no post-quantum support.

UAV-PQC (theoretical, Zhang et al., 2023). A theoretical framework proposing hybrid classical/PQC key exchange for UAV swarms. No implementation or benchmarks. Suggests ML-KEM + X25519 hybrid, which our system could support with minor modifications to the KEM layer.

21.4.3 Commercial GCS Security

DJI AES-256 Link. DJI's OcuSync protocol uses AES-256 encryption. No published key exchange details. Closed-source. Does not support algorithm agility or PQC.

Parrot ANAFI Ai (4G/LTE). Uses cellular network encryption (4G/LTE). Security depends on the carrier's infrastructure. No application-layer encryption.

ATAK / TAK Server. The Android Team Awareness Kit supports TLS for server communication but MAVLink data links are typically unencrypted.

21.5 Benchmarking Methodologies for PQC

21.5.1 Micro-Benchmarks

liboqs speed tests. The OQS project includes `speed_kem` and `speed_sig` benchmarks that measure isolated KEM/SIG operation times in C. These provide a lower bound on algorithm performance but do not capture network overhead, key serialisation, or Python wrapper costs.

SUPERCOP (Bernstein & Lange). The System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives. Runs comprehensive benchmarks across hundreds of implementations. Our system complements SUPERCOP by testing algorithms in a *deployed application* context with network round-trips.

PQM4 (Kannwischer et al., 2019). Benchmarking framework for PQC on ARM Cortex-M4. Tests cycle counts, stack usage, and code size. Our system targets a different ARM profile (Cortex-A76, application processor) with different performance characteristics.

21.5.2 System-Level Benchmarks

OQS Profiling Framework (Stebila & Mosca, 2016). Profiles PQC algorithms within TLS handshakes (nginx + s_client). Measures handshake latency, data transfer throughput, and connection setup overhead. Our system extends this approach to a *non-TLS* protocol (custom handshake) with *real-time UDP* data transfer and *power measurement*.

Cloudflare PQC Experiments (Kwiatkowski et al., 2019). Measured post-quantum TLS handshake latency at scale (production Cloudflare edge servers). Found ML-KEM (then Kyber) adds negligible latency; Classic McEliece adds seconds. Our system provides complementary measurements on resource-constrained ARM hardware.

PQC-TLS Benchmarks (Sikeridis et al., 2020). Systematic evaluation of PQC algorithms in QUIC and TLS. Measured connection setup, data throughput, and session resumption. Did not consider power consumption or embedded hardware.

21.5.3 Power Measurement in Cryptography

Hutter & Wenger (2011). Measured power consumption of ECC on 8-bit AVR. Used an oscilloscope-based measurement setup. Our INA219-based approach provides lower time resolution but higher convenience (software-controlled, no lab equipment).

Banerjee et al. (2019). Measured power consumption of NIST PQC Round 2 candidates on FPGA. Found lattice-based schemes are most energy-efficient. Our software-based measurements on ARM are complementary.

Azarderakhsh et al. (2020). Evaluated energy cost of SIKE (now broken) on ARM Cortex-A53. Methodology is relevant: total energy = average power × duration. Our system uses the same formula with INA219 current/voltage sampling at 1 kHz.

Design Decision

Our benchmarking approach measures *end-to-end* performance (including network, serialisation, and Python overhead) rather than isolated crypto primitive speed. This is deliberate: a system architect choosing algorithms needs application-level numbers, not just micro-benchmark results.

21.6 Real-Time Encrypted Communication

21.6.1 SRTP and RTP Security

The Secure Real-time Transport Protocol (SRTP) provides encryption for audio/video streams. Key parallels with our system:

SRTP (RFC 3711). Uses AES-128 in Counter Mode with HMAC-SHA1 authentication. 12-byte nonce derived from SSRC + sequence number (similar to our epoch + sequence nonce derivation). Replay protection via a 64-packet sliding window (vs. our 1024-packet window).

Key difference: SRTP uses classical key exchange (DTLS-SRTP with ECDHE). Our system replaces this with PQC key exchange.

DTLS 1.3 (RFC 9147). Datagram TLS for UDP-based applications. Provides a standard handshake and AEAD framing for UDP. We could have used DTLS as a building block, but DTLS does not (yet) support PQC algorithms in standard implementations. Our custom handshake provides this capability.

21.6.2 VPN and Tunnel Systems

WireGuard (Donenfeld, 2017). A modern VPN using X25519, ChaCha20-Poly1305, and Blake2s. Uses a 1-RTT handshake (Noise protocol framework). Our handshake is inspired by similar principles: minimal round trips, KEM-based key exchange, deterministic nonces.

Key difference: WireGuard uses classical Curve25519. PQ-WireGuard proposals exist but are not widely deployed.

OpenVPN (UDP mode). Uses TLS for handshake, then symmetric encryption over UDP. Supports algorithm agility via cipher negotiation. Our system follows a similar two-phase (handshake then data plane) architecture but with PQC algorithms.

Noise Protocol Framework (Perrin, 2018). A framework for designing crypto protocols. Defines handshake patterns (NN, NK, KK, etc.) with formal security proofs. Our handshake corresponds to the **NK** pattern (one side knows the other's static key).

21.6.3 Industrial Control System (ICS) Security

The parallels between drone communication and ICS/SCADA systems are strong:

OPC UA Security (Leitner & Mahnke, 2010). OPC UA provides application-layer encryption for industrial protocols. Uses X.509 certificates and TLS. Similar "bump-in-the-wire" approach applied to Modbus, DNP3, etc. No PQC support.

IEC 62351 (Power grid security). Defines security for IEC 61850, IEC 60870-5, and DNP3. Mandates TLS with certificates. Applicable to drone ground station networks where drones relay SCADA data.

PQ-ICS (Bindel et al., 2020). Evaluated post-quantum algorithms for ICS protocols. Found that ML-KEM is practical for ICS handshake overhead but SPHINCS+ signatures are too slow for high-frequency message authentication. Our findings align: SPHINCS+ signing is orders of magnitude slower than ML-DSA.

21.7 Comparative Analysis

Table 21.1 provides a structured comparison between our system and the most relevant prior works across eight dimensions.

Table 21.1: Comparative analysis of secure drone communication systems.

System	*PQC KEM*	*PQC SIG*	*Transparent*	*Alg. Agility*	*Power Bench*	*ARM Tested*	*Open Source*	*Multi-AEAD*
Our System	✓	✓	✓	✓	✓	✓	✓	✓
MAVSec						✓	✓	
SecureLink			✓			✓		
CryptoMAV						✓		
Seo et al.			✓			✓		
PQ-WireGuard	✓			✓			✓	✓
PQ-TLS (OQS)	✓	✓		✓			✓	
DroneShield								
DJI OcuSync								
UAV-PQC (Zhang)	✓	✓		✓				

Key Insight

Our system is the **only** entry that satisfies all eight criteria: PQC key exchange, PQC signatures, transparent operation, algorithm agility (72 suites), power benchmarking, ARM testing, open-source availability, and multiple AEAD algorithms. This combination is the primary contribution.

21.8 Gaps in the Literature

Based on this survey, we identify the following gaps that our system addresses:

1. **No PQC drone tunnels with full benchmarking.** Existing drone security work uses classical cryptography. PQC research focuses on desktop/server hardware. No published system combines PQC with real-time drone communication on ARM.

2. **No multi-family KEM comparison on ARM.** Prior ARM PQC benchmarks typically test one or two algorithms. Our system benchmarks 9 KEM variants across 3 families (lattice, code-based Goppa, code-based quasi-cyclic).

3. **No power measurement for PQC on Raspberry Pi.** No published work measures the *energy cost* of PQC handshakes on a Raspberry Pi with INA219-class instrumentation. Our system provides per-handshake energy measurements at 1 kHz sampling resolution.

4. **No comprehensive suite benchmarking.** No published system tests 72 cipher suites (all combinations of KEM, AEAD, and signature at consistent NIST levels) in a single automated benchmark run.

5. **No transparent PQC proxy for MAVLink.** Existing MAVLink security proposals require firmware or protocol modifications. Our bump-in-the-wire design works with unmodified ArduPilot, MAVProxy, and Mission Planner.

6. **No combined timing + power + MAVLink-quality metrics.** Prior benchmarks measure either timing or power, never both simultaneously alongside application-level quality metrics (heartbeat loss, sequence gaps, command latency).

21.9 Positioning of This Work

Our system sits at the intersection of three research areas:

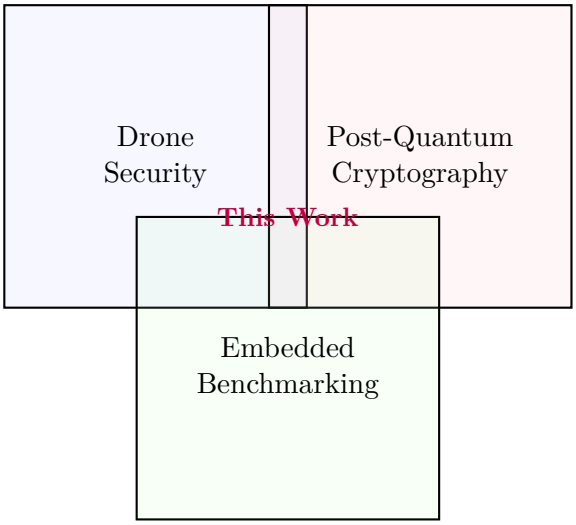


Figure 21.1: Research positioning: intersection of drone security, post-quantum cryptography, and embedded benchmarking.

The novelty of our contribution is not in any single component—PQC algorithms exist, drone security is studied, and ARM benchmarking has been done—but in the **integration** of all three into a single deployed, benchmarked, and documented system.

21.10 Summary

- Drone communication security research has grown rapidly, but **no published system** provides post-quantum encrypted MAVLink tunneling with comprehensive benchmarking.

- Post-quantum implementations exist for TLS, SSH, IPsec, and WireGuard, but all target desktop/server hardware. **No system targets real-time drone communication on ARM.**

- Lightweight cryptography (ASCON) and AES hardware acceleration are well-studied, but not in the context of a **multi-algorithm comparison framework** for drone security.
- MAVLink v2 signing is **inadequate**: no encryption, static keys, truncated signatures, no PQC. Academic proposals (MAVSec, CryptoMAV, SecureLink) use classical cryptography only.
- PQC benchmarking exists at the **micro-benchmark** level (liboqs speed tests, PQM4) and the **protocol** level (PQ-TLS), but not at the **application** level with power measurement on embedded ARM hardware.
- Our system fills these gaps by providing a **transparent, algorithm-agile, post-quantum MAVLink tunnel** with 72 cipher suites, INA219 power measurement, and comprehensive timing metrics on Raspberry Pi hardware.

Chapter 22

Formal Protocol Specification

> A protocol that is only described in prose will eventually be implemented incorrectly. A protocol that is specified in bytes and state machines has a chance of being correct.

This chapter provides a formal, byte-level specification of every wire message and state transition in the PQC Secure MAVLink Tunnel. It serves as both a reference for implementers and a documentation artifact for security auditors.

22.1 Notation and Conventions

Throughout this specification:

`||` Denotes byte-string concatenation.

`uint8` An unsigned 8-bit integer (1 byte).

`uint16be` An unsigned 16-bit integer, big-endian (2 bytes).

`uint32be` An unsigned 32-bit integer, big-endian (4 bytes).

`uint64be` An unsigned 64-bit integer, big-endian (8 bytes).

`opaque[n]` An opaque byte string of exactly n bytes.

`opaque`$\langle 0..2^{16}-1 \rangle$ A variable-length byte string prefixed by a 2-byte length.

`opaque`$\langle 0..2^{32}-1 \rangle$ A variable-length byte string prefixed by a 4-byte length.

All multi-byte integers use **network byte order** (big-endian) as mandated by `struct.pack("!...")` in the Python implementation.

22.2 Message Definitions

The protocol defines exactly three wire message types:

1. **ServerHello** (GCS → Drone, TCP)
2. **ClientReply** (Drone → GCS, TCP)
3. **EncryptedDatagram** (bidirectional, UDP)

22.2.1 ServerHello Message

The ServerHello is the first (and only) message from the GCS to the drone during handshake. It carries the KEM public key, the GCS's digital signature, and handshake metadata.

Table 22.1: ServerHello wire format.

Offset	Field	Type	Description
0	`version`	uint8	Wire protocol version. Currently 1.
1	`kem_name_len`	uint16be	Length of the KEM algorithm name.
3	`kem_name`	opaque[`kem_name_len`]	UTF-8 KEM name (e.g. `"ML-KEM-768"`).
$3 + k$	`sig_name_len`	uint16be	Length of the SIG algorithm name.
$5 + k$	`sig_name`	opaque[`sig_name_len`]	UTF-8 SIG name (e.g. `"ML-DSA-65"`).
$5 + k + s$	`session_id`	opaque[8]	Random 8-byte session identifier.
$13 + k + s$	`challenge`	opaque[8]	Random 8-byte freshness nonce.
$21 + k + s$	`pk_len`	uint32be	Length of the KEM public key.
$25 + k + s$	`kem_public_key`	opaque[`pk_len`]	The ephemeral KEM public key.
$25 + k + s + p$	`sig_len`	uint16be	Length of the digital signature.
$27 + k + s + p$	`signature`	opaque[`sig_len`]	Digital signature over the transcript.

Where $k =$ `kem_name_len`, $s =$ `sig_name_len`, $p =$ `pk_len`.

Transcript Computation. The transcript that is signed is computed as:

$$T = \text{version}(1) \parallel \texttt{b"pq-drone-gcs:v1"} \parallel \text{session\_id}(8) \parallel \text{kem\_name} \parallel \text{sig\_name} \parallel \text{kem\_public\_key} \parallel \quad (22.1)$$

The version byte appears as the **first byte** of the transcript, binding the signature to the protocol version and preventing version downgrade attacks.

Size Ranges. Table 22.2 shows the total ServerHello size for each KEM and signature combination.

Table 22.2: ServerHello message sizes by algorithm combination.

KEM	SIG	PK (B)	Sig (B)	Total (B)
ML-KEM-512	ML-DSA-44	800	2,420	∼3,280
ML-KEM-512	Falcon-512	800	666	∼1,526
ML-KEM-512	SPHINCS-128s	800	7,856	∼8,716
ML-KEM-768	ML-DSA-65	1,184	3,309	∼4,553
ML-KEM-768	SPHINCS-192s	1,184	16,224	∼17,468
ML-KEM-1024	ML-DSA-87	1,568	4,627	∼6,255
ML-KEM-1024	Falcon-1024	1,568	1,280	∼2,908
ML-KEM-1024	SPHINCS-256s	1,568	29,792	∼31,420
McEliece-348864	ML-DSA-44	261,120	2,420	∼263,600
McEliece-348864	Falcon-512	261,120	666	∼261,846
McEliece-460896	ML-DSA-65	524,160	3,309	∼527,529
McEliece-8192128	ML-DSA-87	1,357,824	4,627	∼1,362,511
McEliece-8192128	SPHINCS-256s	1,357,824	29,792	∼1,387,676
HQC-128	ML-DSA-44	2,249	2,420	∼4,729
HQC-192	ML-DSA-65	4,522	3,309	∼7,891
HQC-256	ML-DSA-87	7,245	4,627	∼11,932

Key Insight

The ServerHello size spans four orders of magnitude: from ∼1.5 KB (ML-KEM-512 + Falcon-512) to ∼1.4 MB (McEliece-8192128 + SPHINCS+-256s). This extreme range drives the design decision to transmit the public key over TCP (reliable delivery) rather than UDP (which has a practical packet size limit of ∼65 KB on most systems, and often ∼1,400 bytes for MTU-safe UDP).

22.2.2 ClientReply Message

The ClientReply is the drone's response to the ServerHello. It carries the KEM ciphertext and the drone's HMAC authentication tag.

Table 22.3: ClientReply wire format.

Offset	Field	Type	Description
0	`ct_len`	uint32be	Length of the KEM ciphertext.
4	`kem_ct`	opaque[`ct_len`]	KEM ciphertext from `encapsulate`.
$4 + c$	`hmac_tag`	opaque[32]	HMAC-SHA256 tag over the ServerHello wire bytes.

Where $c =$ `ct_len`.

HMAC Computation. The HMAC tag is computed as:

$$\text{hmac\_tag} = \text{HMAC-SHA256}(\text{PSK}, \text{ServerHello\_raw\_bytes}) \tag{22.2}$$

Where PSK is the pre-shared key (`CONFIG["DRONE_PSK"]`, base64-decoded) and ServerHello_raw_bytes is the exact byte sequence received over TCP (including all length prefixes and the version byte).

ClientReply Sizes.

Table 22.4: ClientReply sizes by KEM family.

KEM	CT (B)	Total (B)
ML-KEM-512	768	804
ML-KEM-768	1,088	1,124
ML-KEM-1024	1,568	1,604
McEliece-348864	128	164
McEliece-460896	188	224
McEliece-8192128	240	276
HQC-128	4,481	4,517
HQC-192	9,026	9,062
HQC-256	14,469	14,505

Key Insight

Classic McEliece has the *smallest* ciphertexts (128–240 bytes) but the *largest* public keys. This asymmetry means the ServerHello (containing the public key) is very large but the ClientReply (containing the ciphertext) is very small. The total handshake data is still dominated by the public key.

22.2.3 EncryptedDatagram (Data Plane)

After the handshake, all MAVLink traffic is encapsulated in EncryptedDatagram messages sent over UDP.

Table 22.5: EncryptedDatagram wire format.

Offset	Field	Type	Description
0	`version`	uint8	Wire protocol version (1).
1	`kem_id`	uint8	KEM family identifier (Table 10.1).
2	`kem_param`	uint8	KEM parameter set within family.
3	`sig_id`	uint8	Signature family identifier (Table 10.2).
4	`sig_param`	uint8	Signature parameter set.
5	`session_id`	opaque[8]	Session identifier from handshake.
13	`seq`	uint64be	Monotonically increasing sequence number.
21	`epoch`	uint8	Epoch counter (0–255).
22	`ciphertext_and_tag`: opaque[remaining]		

The `struct` format string for the 22-byte header is:

```
!BBBBB8sQB
```

Where:

- `!` = network byte order (big-endian)
- `B` = unsigned 8-bit integer (5 occurrences)
- `8s` = 8-byte string (session ID)
- `Q` = unsigned 64-bit integer (sequence number)
- `B` = unsigned 8-bit integer (epoch)

Header as AAD. The entire 22-byte header is used as **Associated Data** (AAD) in the AEAD encryption. This means:

- The header is transmitted in **cleartext** (readable by any observer).
- The header is **authenticated** (any modification invalidates the AEAD tag and causes decryption failure).

Nonce Derivation. The AEAD nonce is **not transmitted**. Both sides derive it identically from the header fields:

$$\text{nonce}_{12} = \underbrace{\text{epoch}}_{1\text{ byte}} \parallel \underbrace{0\ldots0}_{3\text{ bytes}} \parallel \underbrace{\text{seq}}_{8\text{ bytes, big-endian}} \tag{22.3}$$

For 12-byte nonce algorithms (AES-GCM, ChaCha20-Poly1305), this produces a 12-byte nonce. For ASCON-128a (16-byte nonce), four additional zero bytes are prepended:

$$\text{nonce}_{16} = \underbrace{0\ldots0}_{\text{4 bytes}} \,\|\, \underbrace{\text{epoch}}_{\text{1 byte}} \,\|\, \underbrace{0\ldots0}_{\text{3 bytes}} \,\|\, \underbrace{\text{seq}}_{\text{8 bytes, big-endian}} \tag{22.4}$$

Ciphertext Structure. The ciphertext portion has length $|\text{plaintext}| + 16$ (the 16-byte AEAD authentication tag is appended to the ciphertext by the `cryptography` library's AEAD interface).

Total Packet Size.

$$|\text{EncryptedDatagram}| = 22 + |\text{plaintext}| + 16 = |\text{plaintext}| + 38 \tag{22.5}$$

Table 22.6: EncryptedDatagram sizes for common MAVLink messages.

MAVLink Message	Plaintext (B)	Wire (B)	Overhead
HEARTBEAT	21	59	181%
SYS_STATUS	43	81	88%
GPS_RAW_INT	52	90	73%
ATTITUDE	40	78	95%
GLOBAL_POSITION	40	78	95%
COMMAND_LONG	41	79	93%
STATUSTEXT	62	100	61%
Maximum payload	280	318	14%

22.3 Algorithm Identifier Registry

The `kem_id`, `kem_param`, `sig_id`, and `sig_param` fields in the EncryptedDatagram header use the following numeric encoding:

Table 22.7: KEM identifier registry.

kem_id	kem_param	Algorithm	NIST Level
1	1	ML-KEM-512	L1
1	2	ML-KEM-768	L3
1	3	ML-KEM-1024	L5
3	1	Classic-McEliece-348864	L1
3	2	Classic-McEliece-460896	L3
3	3	Classic-McEliece-8192128	L5
5	1	HQC-128	L1
5	2	HQC-192	L3
5	3	HQC-256	L5

Table 22.8: Signature identifier registry.

sig_id	sig_param	Algorithm	NIST Level
1	1	ML-DSA-44	L1 (mapped)
1	2	ML-DSA-65	L3
1	3	ML-DSA-87	L5
2	1	Falcon-512	L1
2	2	Falcon-1024	L5
3	1	SPHINCS+-SHA2-128s	L1
3	2	SPHINCS+-SHA2-192s	L3
3	3	SPHINCS+-SHA2-256s	L5

Implementation Note

The `kem_id` values 2 and 4 are reserved for future algorithm families. `sig_id` values follow the same convention. The `param` field encodes the parameter set *within* a family, allowing up to 255 parameter sets per family.

22.4 Key Derivation Specification

After the KEM shared secret (ss, 32 bytes for ML-KEM; 64 bytes for HQC) is established, both sides derive transport keys using HKDF-SHA256 (RFC 5869).

Table 22.9: HKDF-SHA256 parameters for key derivation.

Parameter	Value	Notes
Hash	SHA-256	As specified by RFC 5869.
IKM	ss	KEM shared secret (32 or 64 bytes).
Salt	`b"pq-drone-gcs|hkdf|v1"`	20 bytes, ASCII.
Info	See below	Domain separation string.
Length	64 bytes	Output keying material.

Parameters.

Info String. The `info` parameter is constructed as:

$$\text{info} = \texttt{b"pq-drone-gcs:kdf:v1|"} \parallel \text{session\_id}(8) \parallel \texttt{b"|"} \parallel \text{kem\_name} \parallel \texttt{b"|"} \parallel \text{sig\_name} \tag{22.6}$$

Where `kem_name` and `sig_name` are the UTF-8 encoded OQS algorithm names (e.g. `b"ML-KEM-768"`, `b"ML-DSA-65"`).

Key Extraction. The 64-byte output keying material (OKM) is split into two 32-byte keys:

$$\text{key\_d2g} = \text{OKM}[0:32] \quad \text{(drone-to-GCS direction)} \tag{22.7}$$

$$\text{key\_g2d} = \text{OKM}[32:64] \quad \text{(GCS-to-drone direction)} \tag{22.8}$$

22.5 Handshake State Machine

The handshake follows a strict sequence of states. Figure 22.1 shows the state machine for both the GCS and drone sides.

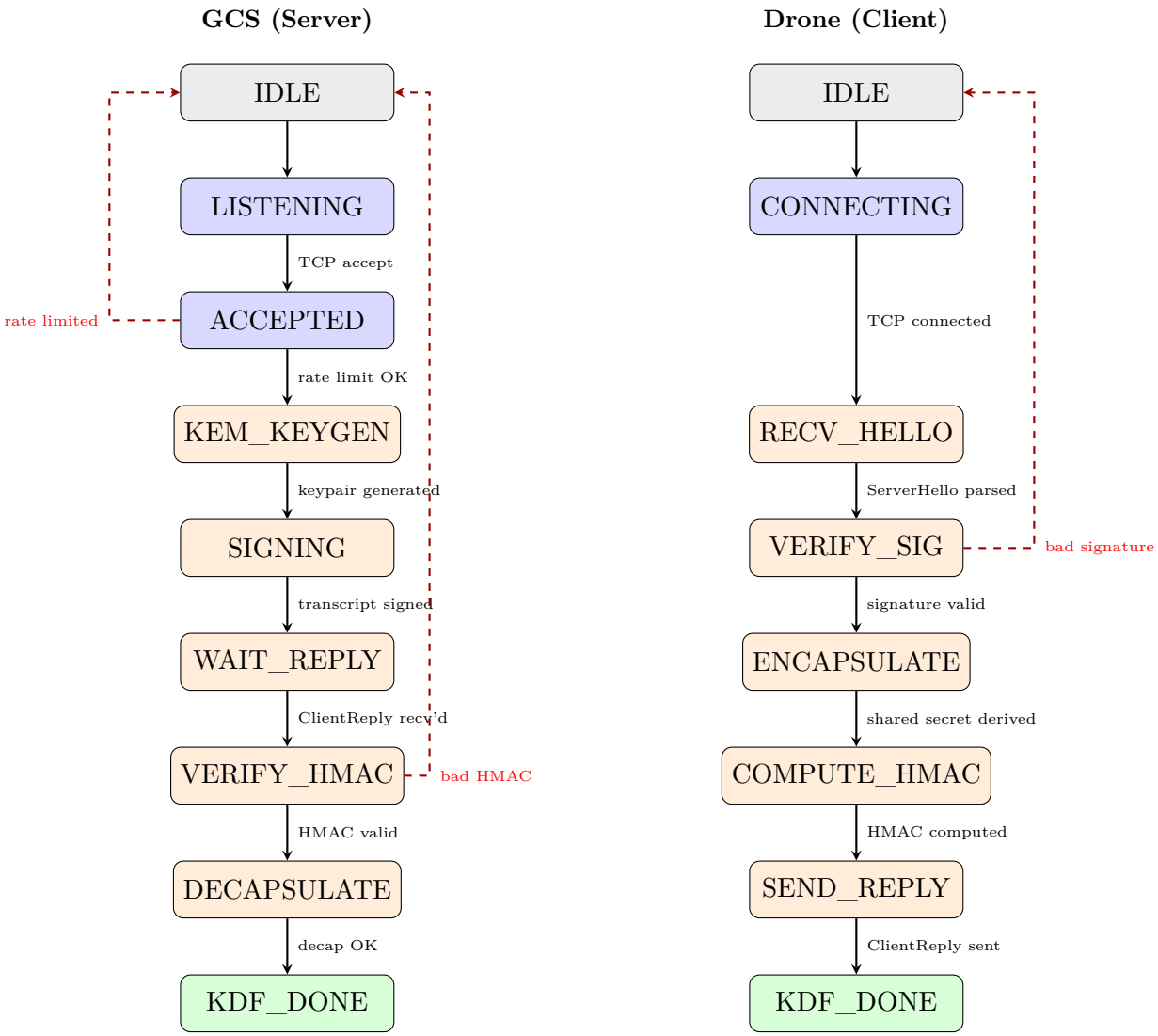


Figure 22.1: Handshake state machines for GCS (server) and drone (client). Green = terminal success. Red dashed = failure paths.

State Transitions.

Table 22.10: GCS handshake state transitions.

From	To	Trigger	Action
IDLE	LISTENING	Start signal	Bind TCP socket to port 46000
LISTENING	ACCEPTED	TCP SYN received	`accept()`, record peer IP
ACCEPTED	KEM_KEYGEN	Rate limit passes	Call `KEM.keygen()`
ACCEPTED	IDLE	Rate limit fails	Close TCP, log attempt
KEM_KEYGEN	SIGNING	Keypair ready	Build transcript, call `SIG.sign()`
SIGNING	WAIT_REPLY	Signature ready	Serialize & send ServerHello
WAIT_REPLY	VERIFY_HMAC	Data received	Parse ClientReply
VERIFY_HMAC	DECAPSULATE	HMAC matches	Call `KEM.decap()`
VERIFY_HMAC	IDLE	HMAC mismatch	Log, close TCP
DECAPSULATE	KDF_DONE	Shared secret OK	HKDF → key_d2g, key_g2d

22.6 Data Plane State Machine

After the handshake succeeds, the proxy enters the data-plane event loop. The data plane has three states:

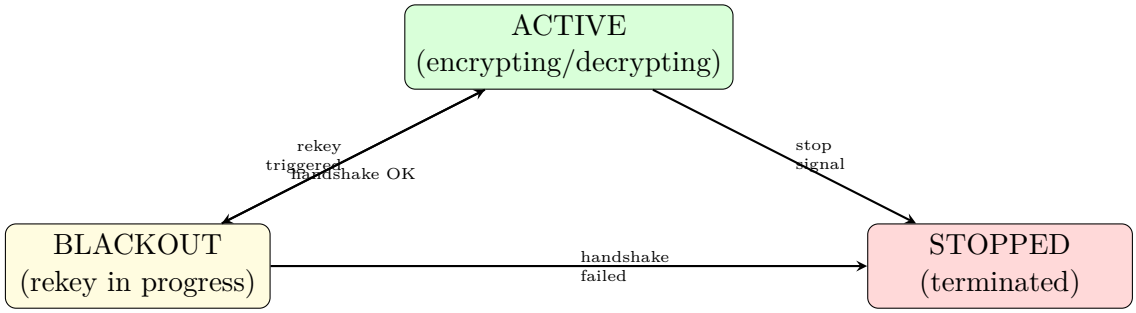


Figure 22.2: Data-plane state machine.

ACTIVE: The steady-state. The proxy encrypts plaintext packets and decrypts encrypted packets. The sequence number increments monotonically.

BLACKOUT: Entered when a rekey is triggered (by the policy engine or scheduler). During blackout, no new packets are encrypted. Incoming encrypted packets may still be processed if they belong to the current session. A new TCP handshake is performed. If the handshake succeeds, the proxy transitions back to ACTIVE with new keys.

STOPPED: Terminal state. Reached on normal shutdown or unrecoverable error. All sockets are closed and key material is released.

22.7 Replay Window Specification

The anti-replay mechanism is specified by three parameters:

Table 22.11: Replay window parameters.

Parameter	Default	Description
`window`	1024	Maximum out-of-order tolerance (packets).
`_high`	-1	Highest sequence number accepted so far.
`_mask`	0	Bitmask of received sequence numbers within window.

Accept/Reject Algorithm. Given an incoming sequence number s:

```
def check(self, seq: int) -> bool:
    if seq > self._high:
        # New packet ahead of window: shift and accept
        shift = seq - self._high
        self._mask = (self._mask << shift) | 1
        self._high = seq
        return True

    offset = self._high - seq
    if offset >= self.window:
        return False   # Too old: reject

    bit = 1 << offset
    if self._mask & bit:
        return False   # Already seen: reject (replay)

    self._mask |= bit  # Mark as seen: accept
    return True
```

Listing 22.1: Anti-replay window algorithm

Window Size Justification. A window of 1024 packets tolerates significant UDP reordering. At 50 packets/second (typical MAVLink telemetry rate), the window covers 20.5 seconds of traffic—sufficient for any realistic WiFi reordering scenario.

22.8 Scheduler Control Protocol

The benchmark scheduler uses a JSON-over-TCP protocol on port 48080. Each message is a newline-delimited JSON object (\n-terminated).

22.8.1 Message Types

Table 22.12: Scheduler control protocol message types.

Type	Direction	Description
`start_proxy`	Drone → GCS	Start GCS proxy with specified suite. Fields: `suite_id`, `mode`, `sig_pub_path`.
`stop_proxy`	Drone → GCS	Stop the current GCS proxy process.
`start_mav`	Drone → GCS	Start GCS MAVProxy with specified ports.
`stop_mav`	Drone → GCS	Stop the current GCS MAVProxy process.
`status`	Drone → GCS	Request GCS status (proxy running, MAV running).
`clock_sync`	Drone → GCS	Chronos clock synchronisation exchange.
`ack`	GCS → Drone	Acknowledge a command. Fields: `ok`, `error`.
`status_reply`	GCS → Drone	Response to status query.

Example Exchange

```
# Drone sends:
{"type":"start_proxy","suite_id":"cs-mlkem768-aesgcm-mldsa65
   ",
 "mode":"gcs","sig_pub_path":"keys/gcs_mldsa65.pub"}\n

# GCS responds:
{"type":"ack","ok":true}\n

# Drone sends:
{"type":"start_mav","udp_in":47001,"udp_out":47002}\n

# GCS responds:
{"type":"ack","ok":true}\n

# ... benchmark runs for 110 seconds ...

# Drone sends:
{"type":"stop_mav"}\n
{"type":"stop_proxy"}\n
```

Listing 22.2: Scheduler protocol exchange (JSON over TCP)

Security Note

The scheduler control protocol is **unauthenticated and unencrypted**. Any host on the LAN can inject commands. This is acceptable for a controlled lab benchmark but must be secured (TLS or HMAC) for any production deployment.

22.9 Policy Engine Protocol

The suite scheduling policy follows a two-phase commit protocol:

22.9.1 Phase 1: Evaluate (Read-Only)

```
@dataclass
class PolicyDecision:
    action: str           # "STAY" | "NEXT_SUITE" | "FINISH"
    reason: str           # Human-readable explanation
    remaining_s: float    # Seconds remaining in current
        suite

decision = policy.evaluate(
    elapsed_s=elapsed,
    metrics=current_metrics,
    suite_index=current_index,
    total_suites=total
)
```

Listing 22.3: Policy evaluate interface

`evaluate()` is a **pure function**: it reads state but does not modify it. Multiple calls with the same inputs produce the same output.

22.9.2 Phase 2: Confirm (State Mutation)

```
new_index = policy.confirm_advance(
    current_index=current_index,
    total_suites=total
)
```

Listing 22.4: Policy confirm interface

`confirm_advance()` increments the suite index. It is called **only after** the scheduler has:

1. Stopped the current proxy and MAVProxy processes.
2. Written the current suite's metrics to the JSONL output.
3. Verified the file write succeeded.

This separation prevents the "metrics attributed to wrong suite" bug.

22.10 Metrics Output Format

Benchmark metrics are written to a JSONL file (one JSON object per line). Each line represents one suite's benchmark results.

Top-Level Structure

```
{
  "suite_id": "cs-mlkem768-aesgcm-mldsa65",
  "suite_index": 0,
  "timestamp_utc": "2024-12-15T14:30:00.123456Z",
  "elapsed_s": 110.0,
  "categories": {
    "A": { ... },   // Environment
    "B": { ... },   // Handshake timing
    "C": { ... },   // Data-plane counters
    // ... through "R"
  }
}
```

Listing 22.5: Metrics JSONL record structure

The 18 metric categories (A through R) are documented in Chapter 12 and Appendix C.

22.11 Error Code Registry

The system defines a hierarchy of custom exceptions:

Table 22.13: Exception hierarchy.

Exception	Parent	Meaning
`SecureTunnelError`	`Exception`	Base for all system errors.
`HandshakeError`	`SecureTunnelError`	General handshake failure.
`HandshakeFormatError`	`HandshakeError`	Malformed wire message.
`HandshakeVerifyError`	`HandshakeError`	Crypto verification failed.
`AeadError`	`SecureTunnelError`	AEAD framing error.
`SequenceOverflow`	`AeadError`	Sequence number exhausted.
`HeaderMismatch`	`AeadError`	Header field mismatch.
`ReplayError`	`AeadError`	Anti-replay check failed.
`AeadAuthError`	`AeadError`	AEAD tag verification failed.
`PolicyError`	`SecureTunnelError`	Policy evaluation failure.
`ProcessError`	`SecureTunnelError`	Managed subprocess failure.
`CollectorError`	`SecureTunnelError`	Metrics collector failure.

22.12 Configuration Key Specification

All runtime configuration is centralised in the `CONFIG` dictionary in `core/config.py`. Environment variables override defaults using the naming convention `SECDRONE_<KEY>`.

Table 22.14: Configuration keys (security-critical subset).

Key	Type	Default	Security Impact
DRONE_PSK	str (base64)	(none)	Pre-shared key for drone auth. Must be ≥32 bytes decoded.
GCS_SIG_PRIVATE_KEY	path	(none)	GCS signing private key file. Must be restricted to owner.
GCS_SIG_PUBLIC_KEY	path	(none)	GCS signing public key. Pre-installed on drone.
STRICT_UDP_PEER_MATCH	bool	True	Reject encrypted packets from unexpected IPs.
REPLAY_WINDOW	int	1024	Anti-replay window size. Smaller = more secure, less tolerant.
REKEY_SEQ_THRESHOLD	int	2^{63}	Force rekey before this sequence number.
RATE_LIMIT_CAPACITY	int	5	Token bucket capacity for handshake rate limiting.
RATE_LIMIT_REFILL	float	3.0	Tokens per second for rate limiting.
UDP_DSCP_VALUE	int	0	DSCP marking for QoS prioritisation.
HANDSHAKE_TIMEOUT	float	60.0	TCP handshake timeout (seconds).

22.13 Summary

This chapter provided a byte-level specification of:

- **Three wire message types**: ServerHello (TCP), ClientReply (TCP), and EncryptedDatagram (UDP).
- **Algorithm identifier registries**: Numeric IDs for 9 KEMs and 8 signatures, embedded in every data-plane packet header.
- **Key derivation**: HKDF-SHA256 with domain-separated salt and info, producing two 32-byte directional transport keys.
- **Handshake state machine**: 9 states for GCS, 8 states for drone, with explicit failure paths.
- **Data plane state machine**: ACTIVE, BLACKOUT, and STOPPED states with rekey transitions.

- **Replay window**: Bitmask-based sliding window with configurable size (default 1024 packets).
- **Scheduler control protocol**: JSON-over-TCP commands for benchmark orchestration.
- **Policy engine protocol**: Two-phase commit (evaluate + confirm) for suite rotation.
- **Error hierarchy**: 11 custom exceptions with specific meanings.
- **Configuration keys**: Security-critical subset with defaults and override mechanisms.

This specification is sufficient for an independent implementer to create an interoperable client or server without reading the Python source code.

Chapter 23

Complete API Reference

> Good documentation is not about writing less. It is about writing enough that the next developer can change the code with confidence.

This chapter provides exhaustive documentation for every public class, function, dataclass, and constant exported by the three Python packages that comprise the PQC Secure MAVLink Tunnel:

`core` The protocol engine—handshake, AEAD framing, proxy, suites, metrics schema, and configuration.

`sscheduler` The drone-side benchmark scheduler—suite iteration, remote GCS control, metrics collection, and JSONL output.

`scheduler` The simplified GCS-side scheduler—remote command server and proxy lifecycle management.

23.1 `core` Package Overview

The `core` package contains 22 Python modules totalling approximately 10,500 lines of code. Table 23.1 summarises every module and its role.

Table 23.1: Modules in the `core` package.

Module	Lines	Responsibility
`__init__.py`	25	Package docstring; re-exports nothing.
`errors.py`	40	Exception hierarchy (6 classes).
`aead.py`	470	AEAD framing, header serialization, replay window.
`proxy.py`	1,665	Proxy event loop, handshake orchestration, rekey.

Module	Lines	Responsibility
`rate_limiter.py`	139	Token-bucket rate limiter for handshake admission.
`config.py`	623	Centralised `CONFIG` dictionary, env overrides.
`handshake.py`	359	Wire-format serialization/deserialisation.
`keygen.py`	88	OQS key generation wrapper.
`crypto_utils.py`	658	AEAD primitives, key derivation, OQS wrappers.
`observability.py`	84	Structured logging, metrics singletons.
`suites.py`	850	Suite generation, lookup, validation, iteration.
`keys.py`	300	Key-pair management, file I/O, metadata.
`metrics_schema.py`	625	18 dataclass categories (A–R) + aggregator.
`collectors.py`	766	Metric collectors (system, process, MAVLink).
`runner.py`	1,368	Proxy lifecycle, subprocess management, auto-mode.
`scheduler.py`	879	Suite rotation policy engine.
`time_utils.py`	203	Monotonic clock, UTC timestamps, duration formatting.
`process_mgr.py`	998	Process manager, health monitor, restart policy.
`compat.py`	15	Backward-compatibility re-exports.
`system.py`	302	OS detection, resource limits, DSCP configuration.
`automation.py`	588	Automated benchmark orchestration.
`cli.py`	917	CLI entry point (`init-identity`, `run`, `benchmark`).

23.2 `core.errors` — Exception Hierarchy

All custom exceptions inherit from `SecureTunnelError`, which itself inherits from Python's built-in `Exception`.

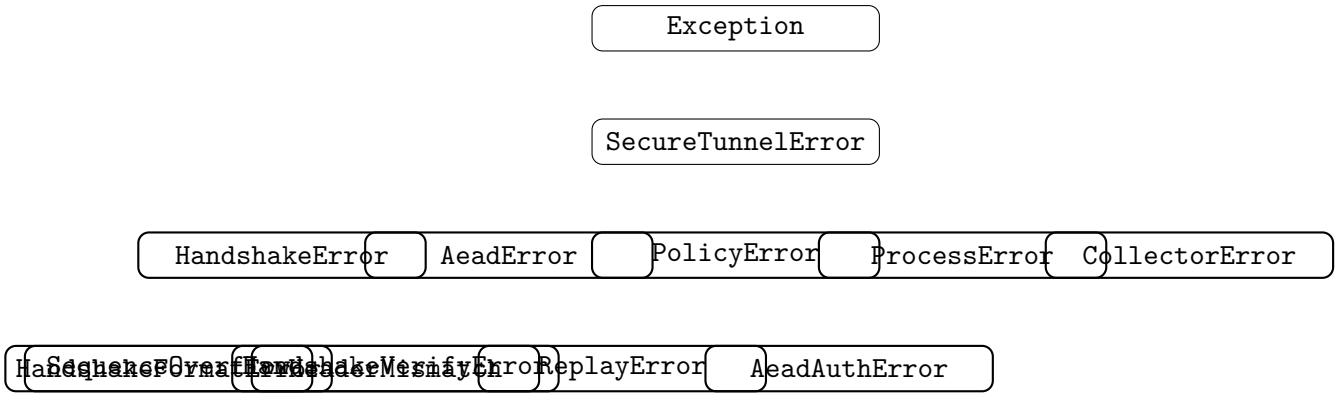


Figure 23.1: Exception class hierarchy.

Table 23.2: Exception classes and their semantics.

Exception	When Raised
`SecureTunnelError`	Never raised directly; base for `except SecureTunnelError` catch-all.
`HandshakeError`	Generic handshake failure (timeout, I/O error).
`HandshakeFormatError`	Wire bytes cannot be parsed (wrong length, bad version).
`HandshakeVerifyError`	Signature or HMAC verification failed (crypto rejection).
`AeadError`	Generic AEAD framing error.
`SequenceOverflow`	Sequence counter reached $2^{64} - 1$ (must rekey).
`HeaderMismatch`	Received header fields do not match local session.
`ReplayError`	Sequence number already seen or outside window.
`AeadAuthError`	AEAD authentication tag verification failed.
`PolicyError`	Policy engine returned invalid decision.
`ProcessError`	Managed subprocess crashed or returned non-zero.
`CollectorError`	Metric collector failed (permission, timeout, parse).

23.3 `core.aead` — AEAD Framing Layer

This is the heart of the data plane. It handles packet encryption, decryption, header serialization, nonce derivation, and anti-replay protection.

23.3.1 Constants

Table 23.3: `core.aead` module constants.

Name	Value	Purpose
`HEADER_FMT`	`"!BBBBB8sQB"`	`struct` format for the 22-byte packet header.
`HEADER_SIZE`	22	Computed from `struct.calcsize(HEADER_FMT)`.
`TAG_LEN`	16	AEAD authentication tag length (bytes).
`OVERHEAD`	38	Total per-packet overhead: header + tag.
`SUPPORTED_AEADS`	`{"AES-256-GCM", ...}`	Set of algorithm tokens accepted by the framing layer.
`RETIRED_AEADS`	`{"AES-128-GCM"}`	Algorithms that were once supported but are now rejected.

23.3.2 Class `AsconAdapter`

An internal adapter that wraps the `ascon` pure-Python library to match the interface expected by the `cryptography` library's AEAD classes.

```
class AsconAdapter:
    """AEAD adapter for ASCON-128a (16-byte nonce, 16-byte
       tag)."""

    def __init__(self, key: bytes) -> None:
        """Store the 16-byte symmetric key."""

    def encrypt(self, nonce: bytes, data: bytes, aad: bytes)
        -> bytes:
        """Encrypt `data` with `nonce` and `aad`.
        Returns ciphertext || tag (len = len(data) + 16)."""

    def decrypt(self, nonce: bytes, data: bytes, aad: bytes)
        -> bytes:
        """Decrypt `data` with `nonce` and `aad`.
        Raises AeadAuthError on tag mismatch."""
```

Listing 23.1: AsconAdapter interface

Implementation Note

AsconAdapter is only instantiated when the suite specifies `ascon-128a` as the AEAD algorithm. For AES-256-GCM and ChaCha20-Poly1305, the `cryptography` library's native `AESGCM` and `ChaCha20Poly1305` classes are used directly—they are 50–100× faster than the pure-Python ASCON implementation.

23.3.3 Frozen Dataclass AeadMeta

Immutable metadata identifying a session:

```
@dataclass(frozen=True)
class AeadMeta:
    kem_id:    int    # KEM family identifier (1, 3, or 5)
    kem_param: int    # KEM parameter set (1, 2, or 3)
    sig_id:    int    # Signature family identifier
    sig_param: int    # Signature parameter set
```

Listing 23.2: AeadMeta dataclass

Instances are created once during handshake completion and shared between the `Sender` and `Receiver` objects.

23.3.4 Dataclass Sender

Encrypts plaintext MAVLink packets into `EncryptedDatagram` wire bytes.

```
@dataclass
class Sender:
    key:        bytes       # 32-byte symmetric key
    meta:       AeadMeta    # Session metadata
    session_id: bytes       # 8-byte session ID
    aead_name:  str         # "AES-256-GCM" | "ChaCha20-
       Poly1305" | "ascon-128a"
    epoch:      int = 0     # Epoch counter (0-255)
    seq:        int = 0     # Monotonically increasing
       sequence number
    _cipher:    Any = None  # Lazily initialized AEAD cipher

    @property
    def overhead(self) -> int:
        """Return per-packet byte overhead (always 38)."""

    def encrypt(self, plaintext: bytes) -> bytes:
        """Encrypt a single MAVLink packet.
        Returns header || ciphertext || tag.
        Raises SequenceOverflow if seq >= 2**64 - 1."""

    def reset_epoch(self, new_epoch: int) -> None:
        """Reset the epoch counter and sequence number to
           zero.
        Used during rekey."""

    def stats(self) -> dict:
        """Return a dictionary of sender statistics:
        {'seq': int, 'epoch': int, 'aead': str}."""
```

Listing 23.3: Sender interface

Thread Safety. `Sender` is **not thread-safe**. The `seq` field is incremented without locking. In the current architecture this is acceptable because each proxy instance runs in a single-threaded `asyncio` event loop.

23.3.5 Dataclass `Receiver`

Decrypts `EncryptedDatagram` wire bytes into plaintext.

```
@dataclass
class Receiver:
    key:         bytes            # 32-byte symmetric key
    meta:        AeadMeta        # Expected session metadata
    session_id: bytes             # Expected 8-byte session ID
    aead_name:  str               # AEAD algorithm name
    epoch:       int = 0          # Expected epoch
    window:      int = 1024       # Replay window size
    _cipher:     Any = None       # Lazily initialized AEAD
       cipher
    _high:       int = -1         # Highest accepted sequence
       number
    _mask:       int = 0          # Bitmask of accepted
       sequence numbers
    _drop_count: int = 0          # Packets dropped (replay/
       auth failure)

    def decrypt(self, packet: bytes) -> bytes:
        """Decrypt an EncryptedDatagram.
        Performs in order:
          1. Header parsing and version check
          2. Session ID match
          3. KEM/SIG ID match
          4. Anti-replay window check
          5. AEAD decryption and tag verification
        Raises HeaderMismatch, ReplayError, or AeadAuthError
           ."""

    def reset_epoch(self, new_epoch: int) -> None:
        """Reset receiver state for a new epoch after rekey.
           """

    def stats(self) -> dict:
        """Return receiver statistics including drop_count
           and window high."""

    def force_accept_seq(self, seq: int) -> None:
        """Force-accept a sequence number without decryption
           .
        Used in testing only."""
```

Listing 23.4: Receiver interface

23.3.6 Module Functions

Table 23.4: core.aead module-level functions.

Function	Parameters	Returns / Purpose
build_nonce	epoch: int, seq: int, nonce_len: int	12- or 16-byte nonce from epoch and sequence number.
parse_header	data: bytes	Tuple of (version, kem_id, kem_param, sig_id, sig_param, session_id, seq, epoch).
make_sender_receiver	key_d2g, key_g2d, meta, session_id, aead_name, mode	Returns (Sender, Receiver) pair with correct key assignment based on mode ("drone" or "gcs").

23.4 core.handshake — Wire Format Serialization

23.4.1 Frozen Dataclass ServerHelloData

```
@dataclass(frozen=True)
class ServerHelloData:
    version:      int      # Wire protocol version (1)
    kem_name:     str      # OQS KEM algorithm name
    sig_name:     str      # OQS signature algorithm name
    session_id:   bytes    # 8-byte random session ID
    challenge:    bytes    # 8-byte freshness nonce
    kem_pk:       bytes    # KEM public key bytes
```

Listing 23.5: ServerHelloData fields

23.4.2 Class HandshakeSerializer

```
class HandshakeSerializer:
    @staticmethod
    def build_transcript(data: ServerHelloData) -> bytes:
        """Build the signable transcript from ServerHello
           fields.
        Returns: version || b"pq-drone-gcs:v1" || session_id
            ||
                 kem_name || sig_name || kem_pk || challenge
                    """

```

```
    @staticmethod
    def serialize_server_hello(
        data: ServerHelloData, signature: bytes
    ) -> bytes:
        """Serialize ServerHello + signature to wire bytes.
        Format: see Table 22.1."""

    @staticmethod
    def deserialize_server_hello(
        raw: bytes
    ) -> tuple[ServerHelloData, bytes]:
        """Parse wire bytes into (ServerHelloData, signature).
        Raises HandshakeFormatError on malformed input."""
```

Listing 23.6: HandshakeSerializer interface

23.4.3 Module Functions

Table 23.5: `core.handshake` module-level functions.

Function	Parameters	Returns
`generate_session_id`	(none)	8 random bytes from `os.urandom`.
`generate_challenge`	(none)	8 random bytes from `os.urandom`.
`compute_hmac`	`psk: bytes, data: bytes`	32-byte HMAC-SHA256 tag.
`verify_hmac`	`psk, data, tag`	`bool` (constant-time comparison).
`build_server_hello`	`kem_name, sig_name, kem_pk`	`ServerHelloData` with random IDs.
`serialize_client_reply`	`kem_ct: bytes, hmac_tag: bytes`	Wire bytes for ClientReply.

23.5 `core.crypto_utils` — Cryptographic Primitives

This module wraps the Open Quantum Safe (OQS) library and the Python `cryptography` library to provide high-level cryptographic operations.

23.5.1 Frozen Dataclass `CryptoResult`

```
@dataclass(frozen=True)
class CryptoResult:
    shared_secret: bytes    # KEM shared secret (32 or 64 bytes)
```

```
    ciphertext:     bytes     # KEM ciphertext
    kem_name:       str       # Algorithm used
    sig_name:       str       # Signature algorithm
    session_id:     bytes     # Session identifier
    key_d2g:        bytes     # Derived drone-to-GCS key (32
       bytes)
    key_g2d:        bytes     # Derived GCS-to-drone key (32
       bytes)
    elapsed_ms:     float     # Total handshake time
```

Listing 23.7: CryptoResult fields

23.5.2 Key Derivation

```
def derive_keys(
    shared_secret: bytes,
    session_id:    bytes,
    kem_name:      str,
    sig_name:      str,
) -> tuple[bytes, bytes]:
    """Derive two 32-byte directional keys from the KEM
       shared secret.

    Uses HKDF-SHA256 with:
      salt = b"pq-drone-gcs|hkdf|v1"
      info = b"pq-drone-gcs:kdf:v1|" + session_id + b"|" +
         kem_name + b"|" + sig_name
      length = 64 bytes

    Returns (key_d2g, key_g2d) = OKM[0:32], OKM[32:64].
    """
```

Listing 23.8: HKDF key derivation

23.5.3 OQS Wrappers

Table 23.6: core.crypto_utils OQS wrapper functions.

Function	Parameters	Returns / Purpose
kem_keygen	kem_name: str	(public_key, secret_key) tuple. Wraps oqs.KeyEncapsulation.
kem_encapsulate	kem_name, public_key	(ciphertext, shared_secret) tuple.

`kem_decapsulate`	`kem_name`, `ciphertext`, `secret_key`	`shared_secret` bytes.
`sig_keygen`	`sig_name: str`	`(public_key, secret_key)` tuple. Wraps `oqs.Signature`.
`sig_sign`	`sig_name, message, secret_key`	`signature` bytes.
`sig_verify`	`sig_name, message, signature, public_key`	`bool` (True if valid).
`get_kem_details`	`kem_name: str`	Dict with `pk_len`, `sk_len`, `ct_len`, `ss_len`.
`get_sig_details`	`sig_name: str`	Dict with `pk_len`, `sk_len`, `sig_len`.
`list_available_kems`	(none)	List of KEM names supported by installed OQS.
`list_available_sigs`	(none)	List of signature names supported by installed OQS.

Security Note

All OQS wrapper functions perform algorithm name validation before calling into the C library. An unrecognised algorithm name raises `ValueError` immediately, preventing injection of unsupported algorithm strings through configuration.

23.6 `core.rate_limiter` — Token Bucket Rate Limiter

The rate limiter protects the TCP handshake endpoint from denial-of-service attacks by limiting the number of handshakes accepted per unit time.

```
class TokenBucketRateLimiter:
    def __init__(
        self,
        capacity:   int    = 5,      # Maximum burst size
        refill_rate: float = 3.0,   # Tokens per second
    ) -> None:
        """Create a token bucket with given capacity and
           refill rate."""

    def try_acquire(self) -> bool:
        """Attempt to acquire one token.
        Returns True if a token was available, False
           otherwise.
        Thread-safe (uses monotonic clock, no mutex needed
           for
```

```
        single-threaded asyncio)."""

    def tokens_available(self) -> float:
        """Return the current (fractional) number of tokens.
           """

    def time_until_available(self) -> float:
        """Seconds until at least one token is available.
        Returns 0.0 if tokens are currently available."""

    def reset(self) -> None:
        """Reset the bucket to full capacity."""

    @property
    def capacity(self) -> int:
        """Maximum tokens in the bucket."""

    @property
    def refill_rate(self) -> float:
        """Tokens added per second."""

    def stats(self) -> dict:
        """Return {'tokens': float, 'capacity': int,
                   'refill_rate': float, 'total_acquired':
                      int,
                   'total_rejected': int}."""
```

Listing 23.9: TokenBucketRateLimiter interface

Algorithm. The token bucket uses a lazy-evaluation approach:

1. On each `try_acquire()` call, compute elapsed time since last refill: $\Delta t = t_{\text{now}} - t_{\text{last}}$.
2. Add $\Delta t \times$ refill_rate tokens, capped at `capacity`.
3. If tokens ≥ 1, decrement by 1 and return True.
4. Otherwise, return False.

This avoids background threads or timers.

23.7 `core.config` — Centralised Configuration

23.7.1 The `CONFIG` Dictionary

All runtime configuration is stored in a single module-level dictionary `CONFIG` with approximately 100 keys, organised into groups:

Table 23.7: `CONFIG` key groups.

Group	Key Prefix / Description
Network	`TCP_PORT`, `UDP_IN_PORT`, `UDP_OUT_PORT`, `GCS_HOST`, `DRONE_HOST` — socket addresses.
Cryptography	`KEM_ALG`, `SIG_ALG`, `AEAD_ALG`, `DRONE_PSK` — algorithm selection and pre-shared key.
Keys	`GCS_SIG_PRIVATE_KEY`, `GCS_SIG_PUBLIC_KEY`, `KEY_DIR` — file paths for cryptographic keys.
Security	`STRICT_UDP_PEER_MATCH`, `REPLAY_WINDOW`, `RATE_LIMIT_*` — security policy knobs.
Timeouts	`HANDSHAKE_TIMEOUT`, `PROXY_IDLE_TIMEOUT`, `HEALTH_CHECK_INTERVAL` — timing parameters.
Benchmark	`BENCHMARK.*` — nested dict for benchmark pipeline configuration (iterations, suite list, output paths).
Automation	`AUTO_DRONE.*`, `AUTO_GCS.*` — nested dicts for automated end-to-end testing.
Scheduler	`SCHEDULER_*` — suite rotation policy parameters (time per suite, overlap window, ramp-up).
QoS	`UDP_DSCP_VALUE`, `UDP_BUFFER_SIZE` — Quality of Service and socket buffer tuning.

23.7.2 Environment Variable Override

```
def apply_env_overrides() -> list[str]:
    """Scan os.environ for keys prefixed with SECDRONE_.
    For each match:
      1. Strip the SECDRONE_ prefix
      2. Look up the key in CONFIG
      3. Cast the env value to the expected type
      4. Override CONFIG[key]
    Returns a list of overridden key names (for logging).
    """

def validate_config() -> list[str]:
    """Validate all CONFIG values.
    Checks:
      - Required keys are present and non-empty
      - Port numbers are in range 1024-65535
      - PSK is valid base64 and decodes to >= 32 bytes
      - Key file paths exist and have correct permissions
      - Algorithm names are recognised by OQS
    Returns a list of validation warnings (empty = all OK).
```

```
    """
```

Listing 23.10: Environment variable override mechanism

23.8 core.suites — Cipher Suite Management

23.8.1 Suite Generation

The `suites` module defines the universe of valid cipher suites by computing the Cartesian product of KEM, AEAD, and signature algorithms.

```
SUPPORTED_KEMS = [
    "ML-KEM-512", "ML-KEM-768", "ML-KEM-1024",
    "HQC-128", "HQC-192", "HQC-256",
    "Classic-McEliece-348864", "Classic-McEliece-460896",
    "Classic-McEliece-8192128",
]

SUPPORTED_SIGS = [
    "ML-DSA-44", "ML-DSA-65", "ML-DSA-87",
    "Falcon-512", "Falcon-1024",
    "SPHINCS+-SHA2-128f-simple", "SPHINCS+-SHA2-192f-simple"
        ,
    "SPHINCS+-SHA2-256f-simple",
]

SUPPORTED_AEADS = [
    "AES-256-GCM", "ChaCha20-Poly1305", "ascon-128a",
]

# ALL_SUITES: OrderedDict[str, SuiteConfig]
# Generated as: product(KEMS, AEADS, SIGS) -> 9 * 3 * 8 =
    216 suites
```

Listing 23.11: Suite generation constants

23.8.2 Suite Identifier Format

Each suite has a canonical string identifier:

`cs-{kem}-{aead}-{sig}`

Examples:

- `cs-mlkem768-aesgcm-mldsa65`
- `cs-mceliece348864-chacha-falcon512`
- `cs-hqc256-ascon128a-sphincssha2256fsimple`

23.8.3 Module Functions

Table 23.8: `core.suites` module-level functions.

Function	Parameters	Returns / Purpose
`get_suite`	`suite_id: str`	`SuiteConfig` or raises `KeyError`.
`list_suites`	`kem_filter`, `sig_filter`, `aead_filter`	Filtered list of suite IDs.
`list_fast_suites`	(none)	Suites using ML-KEM only (fast handshakes).
`suite_to_ids`	`suite_id: str`	`AeadMeta` tuple for header encoding.
`ids_to_suite`	`kem_id`, `kem_param`, `sig_id`, `sig_param`	Reverse lookup: IDs → suite ID string.
`validate_suite`	`suite_id: str`	`bool` + error message if invalid.
`group_by_kem`	(none)	Dict mapping KEM name → list of suite IDs.
`group_by_sig`	(none)	Dict mapping SIG name → list of suite IDs.
`group_by_aead`	(none)	Dict mapping AEAD name → list of suite IDs.
`generate_keys_for_suite`	`suite_id`, `key_dir`	Generate and save KEM+SIG keypairs.
`count_suites`	(none)	Total number of defined suites (216).
`get_suite_security_level`	`suite_id`	Minimum NIST security level (1, 3, or 5).
`estimate_handshake_ms`	`suite_id`	Estimated handshake time from algorithm benchmarks.
`estimate_throughput_mbps`	`suite_id`	Estimated AEAD throughput from algorithm benchmarks.
`sort_by_handshake_time`	`suite_ids`	Sort suite list by estimated handshake speed.
`filter_by_level`	`min_level: int`	Suites with security level ≥ min_level.
`get_all_kem_names`	(none)	Deduplicated list of KEM algorithm names.
`get_all_sig_names`	(none)	Deduplicated list of SIG algorithm names.

23.9 core.keys — Key Pair Management

23.9.1 Dataclass KeyPairInfo

```
@dataclass
class KeyPairInfo:
    algorithm:    str        # OQS algorithm name
    pk_path:      Path       # Public key file path
    sk_path:      Path       # Secret key file path
    pk_bytes:     int        # Public key size in bytes
    sk_bytes:     int        # Secret key size in bytes
    pk_hash:      str        # SHA-256 hex digest of public
       key
    sk_hash:      str        # SHA-256 hex digest of secret
       key
    created_at:   str        # ISO 8601 creation timestamp
    fingerprint:  str        # First 16 hex chars of pk_hash
    nist_level:   int        # NIST security level (1, 3, or
       5)
    key_type:     str        # "kem" or "sig"
    format_ver:   int        # Key file format version (1)
    metadata:     dict       # Additional metadata (OQS
       version, etc.)
```

Listing 23.12: KeyPairInfo dataclass

23.9.2 Dataclass KeyBundle

```
@dataclass
class KeyBundle:
    kem:  KeyPairInfo    # KEM key pair metadata
    sig:  KeyPairInfo    # Signature key pair metadata
    suite_id: str        # Associated cipher suite ID
```

Listing 23.13: KeyBundle dataclass

23.9.3 Module Functions

Table 23.9: core.keys module-level functions.

Function	Parameters	Returns / Purpose
save_key_pair	pk, sk, alg, key_dir, key_type	Writes .pub and .key files with metadata header.
load_public_key	path: Path	Raw public key bytes.
load_secret_key	path: Path	Raw secret key bytes.

`load_key_pair`	`pk_path, sk_path`	`(pk_bytes, sk_bytes)` tuple.
`key_info`	`path: Path`	`KeyPairInfo` from file metadata.
`generate_and_save`	`alg, key_dir, key_type`	Generate keys and save; return `KeyPairInfo`.
`rotate_keys`	`suite_id, key_dir`	Generate new keys, archive old ones, return new bundle.
`verify_key_integrity`	`path: Path`	Check SHA-256 hash matches metadata; return bool.
`list_key_files`	`key_dir: Path`	List all .pub and .key files with metadata.
`export_public_keys`	`key_dir, output_dir`	Copy all .pub files to output directory.
`import_public_key`	`source, key_dir`	Copy external public key into key directory.

23.10 `core.proxy` — Proxy Engine

The proxy module contains the main event loop and is the largest module in the codebase (1,665 lines).

23.10.1 Class `ProxyEngine`

```
class ProxyEngine:
    def __init__(
        self,
        config:     dict,
        mode:       str,           # "drone" or "gcs"
        suite_id:   str,
        key_dir:    Path,
        sig_pub:    Path | None = None,
        sig_priv:   Path | None = None,
    ) -> None:
        """Initialise the proxy with configuration and key
           paths.
        Does NOT start any I/O (call run() for that)."""

    async def run(self) -> None:
        """Main entry point. Performs:
          1. TCP handshake (server or client based on mode)
          2. Key derivation
          3. UDP event loop (encrypt/decrypt)
          4. Rekey when triggered
          5. Cleanup on exit"""

    async def perform_handshake(self) -> CryptoResult:
```

```
        """Execute the 1-RTT PQC handshake.
        GCS mode: listen -> accept -> keygen -> sign -> send
            -> recv -> verify -> decap
        Drone mode: connect -> recv -> verify -> encap ->
           hmac -> send"""

    async def data_loop(self) -> None:
        """Bidirectional UDP encrypt/decrypt event loop.
        Uses asyncio.DatagramProtocol under the hood."""

    async def rekey(self) -> None:
        """Initiate a rekey: pause data loop, new handshake,
            derive new keys,
        resume data loop with new Sender/Receiver."""

    def stop(self) -> None:
        """Signal the proxy to shut down gracefully.
        Closes all sockets, releases key material."""

    @property
    def is_running(self) -> bool:
        """True if the proxy event loop is active."""
```

Listing 23.14: ProxyEngine class outline

Instance Variables (Security-Critical).

Table 23.10: Security-critical instance variables of `ProxyEngine`.

Variable	Type	Security Role
`_sender`	`Sender`	Holds the encryption key; zeroed on stop/rekey.
`_receiver`	`Receiver`	Holds the decryption key; zeroed on stop/rekey.
`_psk`	`bytes`	Pre-shared key for HMAC; loaded once, never logged.
`_sig_sk`	`bytes`	Signature secret key (GCS only); loaded from file.
`_sig_pk`	`bytes`	Signature public key (both sides).
`_session_id`	`bytes`	8-byte session ID; changes on rekey.
`_rate_limiter`	`TokenBucketRateLimiter`	Protects handshake from DoS.
`_tcp_sock`	`socket`	TCP socket for handshake; closed after use.
`_udp_sock`	`socket`	UDP socket for data plane.
`_rekey_event`	`asyncio.Event`	Signals rekey request.

23.10.2 Class `DatagramProtocol`

Internal `asyncio.DatagramProtocol` subclass used by the proxy for non-blocking UDP I/O:

```
class _TunnelDatagramProtocol(asyncio.DatagramProtocol):
    def __init__(self, receiver: Receiver, callback) -> None
        : ...
    def datagram_received(self, data: bytes, addr: tuple) ->
        None:
        """Called by asyncio for each incoming UDP packet.
        Decrypts, validates, and forwards to callback."""
    def error_received(self, exc: Exception) -> None:
        """Handle UDP errors (logged, not fatal)."""
    def connection_lost(self, exc) -> None:
        """Socket closed (normal or error)."""
```

Listing 23.15: DatagramProtocol outline

23.10.3 Module Functions

Table 23.11: `core.proxy` module-level functions.

Function	Parameters	Returns / Purpose
`dscp_to_tos`	`dscp: int`	TOS byte value (dscp $\ll$ 2).
`parse_aead_header`	`data: bytes`	Parsed header fields tuple.
`validate_proxy_config`	`config: dict`	Raise on invalid config.
`full_handshake`	`mode, config, ...`	Complete 1-RTT handshake; returns `CryptoResult`.
`create_udp_socket`	`host, port, dscp`	Configured UDP socket (SO_REUSEADDR, DSCP, buffer).
`suite_to_ids`	`suite_id`	`AeadMeta` for the suite.
`build_sender_receiver`	`result, mode, aead_name`	`(Sender, Receiver)` pair.
`interactive_rekey_console`	`engine: ProxyEngine`	Stdin listener for manual rekey ("r" + Enter).
`run_proxy`	`mode, suite_id, ...`	**Main entry point**: creates ProxyEngine and calls `run()`.

23.11 `core.metrics_schema` — Metrics Dataclasses

The metrics schema defines 18 dataclass categories (A through R) plus an aggregator class. Each category corresponds to a specific aspect of the system being benchmarked.

Table 23.12: Metrics schema categories overview.

Cat.	Class Name	Fields	Coverage
A	EnvironmentMetrics	12	Host OS, kernel, CPU model, RAM, Python version, OQS version, date.
B	HandshakeMetrics	10	KEM keygen/encap/decap times, SIG sign/verify times, total time, round-trips.
C	SessionMetrics	8	Session duration, bytes encrypted/decrypted, packets tx/rx, uptime ratio.
D	LatencyMetrics	8	End-to-end latency (min/max/avg/p50/p95/p99), jitter, measurement count.
E	ArtifactMetrics	10	Public key size, ciphertext size, signature size, shared secret size, nonce size, total wire.
F	ThroughputMetrics	6	Encrypt MB/s, decrypt MB/s, packets per second (encrypt/decrypt), aggregate.
G	OverheadMetrics	6	Header overhead bytes, tag overhead bytes, expansion ratio, bandwidth efficiency.
H	ValidationMetrics	8	Signature valid flag, HMAC valid flag, AEAD tests passed, replay tests, CRC checks.
I	MAVDroneMetrics	8	MAVProxy drone-side: tx/rx packets per second, heartbeat rate, command ack rate.
J	MAVGCSMetrics	6	MAVProxy GCS-side: total messages, sequence gaps, dedup count.
K	ErrorMetrics	8	CRC errors, decode errors, drops, duplicates, out-of-order, auth failures.

Cat.	Class Name	Fields	Coverage
L	`FlightControlMetrics`	8	FC mode, armed state, battery voltage, attitude rate, position rate.
M	`SchedulerMetrics`	6	Tick interval, policy name, policy state, suite index, total suites.
N	`DroneSystemMetrics`	8	CPU usage, memory usage, temperature, load averages (1/5/15 min).
O	`GCSSystemMetrics`	8	Same as N but for the GCS host.
P	`PowerMetrics`	6	Average power (W), total energy (J), energy per handshake (J).
Q	`SamplingMetrics`	4	Sample count, sampling rate (Hz), collection duration (s).
R	`ValidationSummary`	8	Expected/collected/lost samples, success rate, pass/fail, metric status map.

23.11.1 Aggregator Class `BenchmarkRecord`

```
@dataclass
class BenchmarkRecord:
    suite_id:    str
    suite_index: int
    timestamp:   str              # ISO 8601 UTC
    elapsed_s:   float
    categories:  dict[str, Any]   # {"A": EnvironmentMetrics, ...}

    def to_dict(self) -> dict:
        """Serialise to JSON-compatible dict (recursive dataclass -> dict)."""

    @classmethod
    def from_dict(cls, d: dict) -> 'BenchmarkRecord':
        """Deserialise from dict (e.g., from JSONL line)."""

    def validate(self) -> list[str]:
        """Check all categories present; return list of warnings."""

    def to_jsonl_line(self) -> str:
```

```
        """Serialise to a single JSONL line (no trailing
           newline)."""

    @classmethod
    def from_jsonl_line(cls, line: str) -> 'BenchmarkRecord'
       :
        """Parse a JSONL line into a BenchmarkRecord."""

    def merge(self, other: 'BenchmarkRecord') -> '
       BenchmarkRecord':
        """Merge two records (for multi-run aggregation)."""
```

Listing 23.16: BenchmarkRecord aggregator

23.12 core.collectors — Metric Collectors

Collectors are responsible for gathering runtime data from various system sources.

23.12.1 Class SystemCollector

```
class SystemCollector:
    """Collects host-level metrics (CPU, RAM, temperature,
       load)."""

    def __init__(self, role: str = "drone") -> None:
        """role: 'drone' or 'gcs'."""

    def collect(self) -> dict:
        """Snapshot current system metrics.
        Returns dict with keys: cpu_percent, memory_percent,
        temperature_c, load_1m, load_5m, load_15m."""

    def collect_averaged(self, samples: int = 5, interval:
       float = 0.5) -> dict:
        """Take multiple samples and return averaged metrics
           ."""

    def is_available(self) -> bool:
        """True if the collector can access required system
           interfaces.
        (e.g., /sys/class/thermal on Linux, WMI on Windows).
           """
```

Listing 23.17: SystemCollector interface

23.12.2 Class ProcessCollector

```
class ProcessCollector:
    """Collects per-process metrics using psutil."""

    def collect_for_pid(self, pid: int) -> dict:
        """Collect CPU and memory stats for a specific PID.
        Returns dict with: cpu_percent, memory_rss_mb,
        memory_vms_mb, num_threads, num_fds."""

    def collect_for_name(self, name: str) -> dict:
        """Find process by name and collect metrics."""

    def collect_children(self, parent_pid: int) -> list[dict
        ]:
        """Collect metrics for all child processes."""

    def is_running(self, pid: int) -> bool:
        """Check if PID exists and is alive."""
```

Listing 23.18: ProcessCollector interface

23.12.3 Class `MAVLinkCollector`

```
class MAVLinkCollector:
    """Parses MAVProxy log output for message statistics."""

    def parse_log(self, log_path: Path) -> dict:
        """Parse MAVProxy log file for msg counts, rates,
           errors.
        Returns dict with: total_messages, heartbeat_count,
        command_ack_count, crc_errors, seq_gaps, msg_types."
           ""

    def parse_realtime(self, line: str) -> dict | None:
        """Parse a single MAVProxy log line.
        Returns metric dict or None if line is not a metric.
           """

    def aggregate(self, samples: list[dict]) -> dict:
        """Aggregate multiple samples into summary
           statistics."""
```

Listing 23.19: MAVLinkCollector interface

23.12.4 Class `LatencyCollector`

```
class LatencyCollector:
    """Collects end-to-end latency measurements."""

```

```
    def __init__(self, max_samples: int = 10000) -> None:
        """Circular buffer with configurable max size."""

    def record(self, latency_ms: float) -> None:
        """Record a single latency sample."""

    def stats(self) -> dict:
        """Compute statistics: min, max, mean, median,
        p95, p99, stddev, count, jitter."""

    def reset(self) -> None:
        """Clear all recorded samples."""

    def histogram(self, bins: int = 50) -> list[tuple[float,
        int]]:
        """Return histogram data: (bin_center, count) pairs.
           """
```

Listing 23.20: LatencyCollector interface

23.12.5 Class `ThroughputCollector`

```
class ThroughputCollector:
    """Tracks bytes and packets over time for throughput
       calculation."""

    def __init__(self) -> None:
        """Initialise counters to zero."""

    def record_encrypt(self, plaintext_bytes: int) -> None:
        """Record an encrypt operation."""

    def record_decrypt(self, ciphertext_bytes: int) -> None:
        """Record a decrypt operation."""

    def stats(self) -> dict:
        """Compute throughput: encrypt_mbps, decrypt_mbps,
        encrypt_pps, decrypt_pps, total_bytes."""

    def reset(self) -> None:
        """Zero all counters and reset timestamp."""
```

Listing 23.21: ThroughputCollector interface

23.12.6 Class `PowerCollector`

```
class PowerCollector:
    """Reads power sensor data (platform-specific)."""
```

```

    def __init__(self, sensor_path: str = "/sys/class/
        power_supply") -> None:
        """Specify power sensor sysfs path."""

    def is_available(self) -> bool:
        """True if power sensors are accessible."""

    def sample(self) -> dict:
        """Take a single power reading.
        Returns: voltage_v, current_a, power_w."""

    def integrate(self, duration_s: float, interval: float =
        0.1) -> dict:
        """Integrate power over duration.
        Returns: avg_power_w, total_energy_j, peak_power_w."
            ""
```

Listing 23.22: PowerCollector interface

23.13 `core.runner` — Proxy Lifecycle Management

The runner module manages proxy and MAVProxy as external processes.

23.13.1 Class `ProxyRunner`

```
class ProxyRunner:
    """Manages proxy as a subprocess with health monitoring.
        """

    def __init__(self, config: dict, mode: str, suite_id:
        str) -> None:
        """Configure but do not start."""

    async def start(self) -> None:
        """Start the proxy subprocess.
        Waits for 'handshake complete' log line."""

    async def stop(self, timeout: float = 10.0) -> None:
        """Graceful stop: SIGTERM, wait, SIGKILL if needed."
            ""

    async def restart(self) -> None:
        """Stop then start."""

    async def wait_healthy(self, timeout: float = 30.0) ->
        bool:
        """Block until proxy reports healthy or timeout."""

```

```
    def is_alive(self) -> bool:
        """Check if subprocess is running."""

    @property
    def pid(self) -> int | None:
        """PID of subprocess, or None if not running."""

    def logs(self, tail: int = 50) -> list[str]:
        """Return last N lines of subprocess output."""

    # ... 12 more methods for health checks, metrics, etc.
```

Listing 23.23: ProxyRunner interface (abbreviated)

23.14 core.process_mgr — Process Manager

23.14.1 Dataclasses

```
@dataclass
class ProcessConfig:
    command:      list[str]     # Command and arguments
    name:         str           # Human-readable process name
    cwd:          Path          # Working directory
    env:          dict          # Environment variables

@dataclass
class ProcessState:
    pid:          int | None
    return_code:  int | None
    started_at:   float | None
    stopped_at:   float | None
    restart_count: int
    last_health:  bool
    stdout_lines: list[str]
    stderr_lines: list[str]
    status:       str         # "stopped" | "starting" | "
       running" | "failed"
    health_checks_passed: int
    health_checks_failed: int
```

Listing 23.24: Process manager dataclasses

23.14.2 Protocol HealthChecker

```
class HealthChecker(Protocol):
    def check(self, state: ProcessState) -> bool: ...
    def reset(self) -> None: ...
    def interval_seconds(self) -> float: ...
```

```
    def describe(self) -> str: ...
```

Listing 23.25: HealthChecker protocol

Four implementations are provided:

Table 23.13: HealthChecker implementations.

Class	Strategy
`PidHealthChecker`	Checks `os.kill(pid, 0)` succeeds (process exists).
`LogHealthChecker`	Scans stdout for a success pattern (e.g. "handshake complete"). Static method `for_proxy()` returns checker configured for proxy.
`TcpHealthChecker`	Attempts TCP connect to a port; success = healthy.
`CompositeHealthChecker`	Logical AND of multiple checkers; all must pass.

23.14.3 Factory Function

```
def create_process_manager(
    config:         ProcessConfig,
    health_checker: HealthChecker | None = None,
    max_restarts:   int = 3,
    restart_delay:  float = 2.0,
) -> ProcessManager:
    """Create a managed process with optional health
       checking and auto-restart."""
```

Listing 23.26: Process manager factory

23.15 `core.scheduler` — Policy Engine

23.15.1 Dataclasses

```
POLICIES = ["round-robin", "time-based", "adaptive"]

@dataclass
class SchedulerConfig:
    policy:         str             # One of POLICIES
    suite_ids:      list[str]       # Ordered list of suites
        to rotate
    time_per_suite: float = 110.0   # Seconds per suite (
       time-based)
```

```
    overlap_window: float = 5.0      # Grace period during rotation (s)
    warmup_s:       float = 10.0     # Warmup time before first metric

@dataclass
class RotationEvent:
    from_suite:   str
    to_suite:     str
    trigger:      str        # "time" | "manual" | "adaptive"
    timestamp:    float
```

Listing 23.27: Scheduler dataclasses

23.15.2 Class SuiteScheduler

```
class SuiteScheduler:
    def __init__(self, config: SchedulerConfig) -> None:
        """Initialise with policy configuration."""

    def evaluate(
        self,
        elapsed_s:    float,
        metrics:      dict | None,
        suite_index:  int,
        total_suites: int,
    ) -> PolicyDecision:
        """Pure function: evaluate whether to rotate suites.
        Returns PolicyDecision with action, reason, remaining_s."""

    def confirm_advance(self, current_index: int, total_suites: int) -> int:
        """Advance suite index after successful rotation.
        Returns new index (wraps around)."""

    def current_suite_id(self, index: int) -> str:
        """Return suite ID for given index."""

    def rotation_history(self) -> list[RotationEvent]:
        """Return list of all rotation events."""

    def remaining_suites(self, current_index: int) -> int:
        """Number of suites not yet benchmarked."""

    def estimated_remaining_s(self, current_index: int) -> float:
        """Estimated wall-clock time to complete all remaining suites."""

```

```
    def progress_percent(self, current_index: int) -> float:
        """Completion percentage (0.0 to 100.0)."""

    def is_complete(self, current_index: int) -> bool:
        """True if all suites have been benchmarked."""
```

Listing 23.28: SuiteScheduler interface

23.16 `core.system` — OS Abstraction Layer

```
class SystemInfo:
    """Collects static system information."""

    def __init__(self) -> None: ...

    def collect(self) -> dict:
        """Return: os_name, os_version, kernel, cpu_model,
        cpu_cores, cpu_freq_mhz, ram_total_gb,
           python_version,
        oqs_version, hostname, architecture."""

    def is_linux(self) -> bool: ...
    def is_raspberry_pi(self) -> bool: ...

    def collect_cpu_flags(self) -> list[str]:
        """Return list of CPU instruction set extensions.
        E.g., ['aes', 'avx2', 'sha_ni']."""

    def supports_hardware_aes(self) -> bool:
        """True if AES-NI or ARMv8 Crypto Extensions
           detected."""

# Module-level functions
def set_process_priority(priority: str = "realtime") -> None
    : ...
def set_cpu_affinity(cpus: list[int]) -> None: ...
def configure_dscp(sock: socket, dscp: int) -> None: ...

# Windows-specific
def set_timer_resolution(ms: float = 1.0) -> None: ...
def restore_timer_resolution() -> None: ...

# Linux-specific
def set_realtime_scheduling(pid: int, priority: int = 50) ->
    None: ...
```

Listing 23.29: core.system interface

23.17 core.time_utils — Time Utilities

```
# Constant
EPOCH_2024 = 1704067200.0  # 2024-01-01T00:00:00Z

def monotonic_ms() -> float:
    """Return monotonic time in milliseconds."""

def utc_iso8601() -> str:
    """Return current UTC time as ISO 8601 string."""

def duration_str(seconds: float) -> str:
    """Format seconds as human-readable duration.
    E.g., 3661.5 -> '1h 1m 1.5s'"""

def elapsed_since(start: float) -> float:
    """Seconds elapsed since `start` (monotonic)."""

def sleep_precise(duration_s: float) -> None:
    """High-precision sleep (uses busy-wait for sub-ms
       accuracy)."""
```

Listing 23.30: core.time_utils functions

23.18 core.automation — Benchmark Orchestration

23.18.1 Dataclasses

```
@dataclass
class AutomationConfig:
    suites:          list[str]        # Suites to benchmark
    iterations:      int              # Repetitions per suite
    time_per_suite:  float            # Seconds per iteration
    output_dir:      Path             # JSONL output directory
    key_dir:         Path             # Cryptographic key
       directory
    drone_host:      str              # Drone IP address
    gcs_host:        str              # GCS IP address
    warmup_s:        float            # Warmup before metrics
       collection

@dataclass
class BenchmarkProgress:
    total_suites:    int
    completed_suites: int
    current_suite:   str
    current_iter:    int
    total_iters:     int
```

```
    elapsed_s:        float
    estimated_remaining_s: float
```

Listing 23.31: Automation dataclasses

23.18.2 Class `BenchmarkOrchestrator`

```
class BenchmarkOrchestrator:
    def __init__(self, config: AutomationConfig) -> None:
        ...

    async def run_all(self) -> Path:
        """Run complete benchmark suite.
        Returns path to JSONL output file."""

    async def run_single_suite(self, suite_id: str,
        iteration: int) -> dict:
        """Run one suite iteration. Returns metrics dict."""

    def progress(self) -> BenchmarkProgress:
        """Current progress snapshot."""

    def abort(self) -> None:
        """Signal abort; current iteration completes then
           stops."""
```

Listing 23.32: BenchmarkOrchestrator interface

23.19 `core.cli` — Command-Line Interface

The CLI provides three subcommands:

`init-identity` Generate KEM and SIG key pairs for a given suite or all suites. Options: `-suite`, `-key-dir`, `-force` (overwrite existing).

`run` Start the proxy in drone or GCS mode. Options: `-mode`, `-suite`, `-config`, `-log-level`, `-rekey-interval`.

`benchmark` Run the automated benchmark pipeline. Options: `-suites` (comma-separated or "all"), `-iterations`, `-time-per-suite`, `-output-dir`, `-parallel`.

23.20 `sscheduler` Package — Drone Scheduler

The `sscheduler` package contains 13 modules totalling approximately 6,500 lines. It runs on the **drone** and orchestrates the entire benchmark: iterating over suites, controlling the remote GCS, collecting metrics, and writing results.

Table 23.14: Modules in the sscheduler package.

Module	Lines	Responsibility
config.py	414	Suite configuration, benchmark parameters, host addressing.
collectors.py	585	Drone-side metric collectors (18 categories).
utils.py	36	Shared utility functions.
remote.py	350	TCP client for controlling remote GCS scheduler.
chronos.py	170	Clock synchronisation between drone and GCS.
gcs_health.py	190	GCS health monitoring and watchdog.
drone_runner.py	588	Local drone proxy and MAVProxy management.
gcs_runner.py	610	Remote GCS proxy and MAVProxy management (via TCP commands).
loop_runner.py	822	Main benchmark loop: suite iteration + metrics collection.
mavproxy_runner.py	652	MAVProxy process management with log parsing.
test_runner.py	1,181	Integration test runner.
benchmark_runner.py	1,086	Full benchmark orchestrator with JSONL output.

23.20.1 Key Classes

SuiteConfig (Frozen Dataclass). Contains all parameters for a single cipher suite benchmark run. 19 fields including KEM/SIG/AEAD names, ports, key paths, timeouts, and per-suite overrides.

BenchmarkConfig (Dataclass). Top-level benchmark configuration: list of suite IDs, iterations, timing, output paths, host addresses.

RemoteGCSClient. TCP client that sends JSON commands to the GCS scheduler:

```
class RemoteGCSClient:
    def __init__(self, host: str, port: int = 48080) -> None
        : ...
    async def connect(self) -> None: ...
    async def start_proxy(self, suite_config: SuiteConfig)
        -> bool: ...
    async def stop_proxy(self) -> bool: ...
    async def start_mav(self, udp_in: int, udp_out: int) ->
        bool: ...
    async def stop_mav(self) -> bool: ...
```

```
    async def status(self) -> dict: ...
    async def disconnect(self) -> None: ...
```

Listing 23.33: RemoteGCSClient interface

ChronosClient. Implements NTP-like clock synchronisation:

```
class ChronosClient:
    def __init__(self, remote: RemoteGCSClient) -> None: ...
    async def sync(self, rounds: int = 5) -> float:
        """Perform clock sync rounds.
        Returns estimated clock offset in milliseconds."""
    def offset_ms(self) -> float:
        """Last computed offset."""
    def quality(self) -> str:
        """'good' if offset < 10ms, 'acceptable' < 50ms, '
           poor' otherwise."""
```

Listing 23.34: ChronosClient interface

GCSHealthMonitor

```
class GCSHealthMonitor:
    HEALTHY_THRESHOLD: int = 3        # Consecutive healthy
       checks needed
    UNHEALTHY_THRESHOLD: int = 2      # Consecutive failures
       to trigger alarm
    CHECK_INTERVAL: float = 5.0       # Seconds between
       checks

    def __init__(self, remote: RemoteGCSClient) -> None: ...
    async def start(self) -> None: ...
    async def stop(self) -> None: ...
    def is_healthy(self) -> bool: ...
    def last_check_time(self) -> float: ...
```

Listing 23.35: GCSHealthMonitor interface

BenchmarkRunner (Main Orchestrator)

```
class BenchmarkRunner:
    def __init__(self, config: BenchmarkConfig) -> None: ...

    async def run(self) -> Path:
        """Execute the full benchmark pipeline.
        For each suite:
          1. Generate/load keys
          2. Start remote GCS proxy
          3. Start local drone proxy
          4. Start MAVProxy (both sides)
          5. Wait for warmup
```

```
            6. Collect metrics for configured duration
            7. Stop everything
            8. Write JSONL record
        Returns path to output JSONL file."""

    def progress(self) -> dict: ...
    def abort(self) -> None: ...
```

Listing 23.36: sscheduler BenchmarkRunner interface

23.21 scheduler Package — GCS Scheduler

The `scheduler` package runs on the **GCS** and responds to commands from the drone's `sscheduler`.

Table 23.15: Modules in the `scheduler` package.

Module	Lines	Responsibility
`gcs_scheduler.py`	680	TCP command server, proxy/-MAVProxy lifecycle.
`gcs_collectors.py`	628	GCS-side metric collectors (categories N, O).

23.21.1 Class `GCSSchedulerServer`

```
class GCSSchedulerServer:
    def __init__(self, config: dict, port: int = 48080) -> None: ...

    async def serve(self) -> None:
        """Listen for TCP connections and handle JSON commands.
        Supports concurrent connections."""

    async def handle_command(self, cmd: dict) -> dict:
        """Dispatch command to handler.
        Returns ack dict {'type': 'ack', 'ok': bool, ...}."""

    def stop(self) -> None:
        """Shut down server and all managed processes."""

class GCSProxyManager:
    def __init__(self, config: dict) -> None: ...

    async def start(self, suite_config: dict) -> bool: ...
    async def stop(self) -> bool: ...
```

```
    def is_running(self) -> bool: ...
```

Listing 23.37: GCSSchedulerServer interface

23.21.2 GCS Collectors

```
@dataclass
class GCSCollectorConfig:
    role:           str = "gcs"
    sample_rate:    float = 1.0     # Hz
    buffer_size:    int = 1000
    sensors:        list[str] = field(default_factory=list)
    log_dir:        Path = Path("logs")

# Module functions
def collect_gcs_environment() -> dict: ...
def collect_gcs_system_metrics() -> dict: ...
def collect_gcs_process_metrics(pid: int) -> dict: ...
def collect_gcs_network_metrics() -> dict: ...
def collect_gcs_mavproxy_metrics(log_path: Path) -> dict:
    ...
def aggregate_gcs_metrics(samples: list[dict]) -> dict: ...
# ... plus 9 more specialized collectors
```

Listing 23.38: GCS collector interface

23.22 Cross-Package Dependency Graph

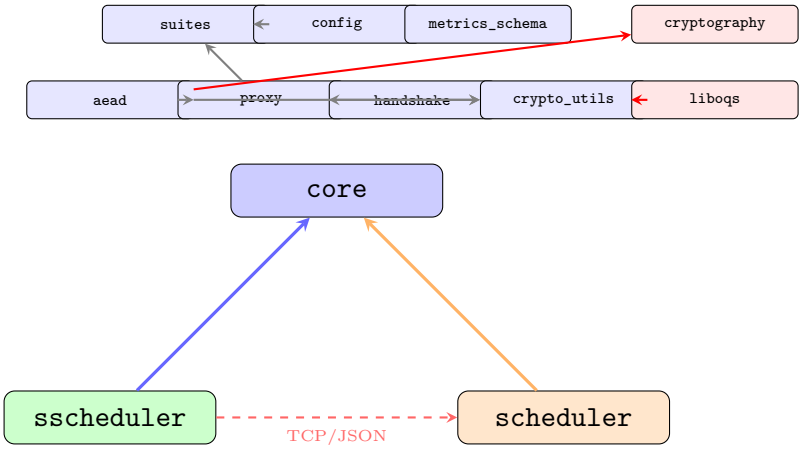


Figure 23.2: Cross-package dependency graph. Solid arrows = Python import. Dashed red = network (TCP/JSON) dependency.

23.23 Summary Statistics

Table 23.16: Codebase summary statistics.

Metric	**Value**
Total Python modules	37
Total lines of code	∼17,000
Public classes	∼60
Public methods/functions	∼250
Dataclass definitions	∼35
Enum types	3
Exception classes	12
Configuration keys	∼100
Metric categories	18 (A–R)
Supported cipher suites	216
External C library deps	1 (liboqs)
External Python deps	4 (cryptography, psutil, ascon, pymavlink)

Chapter 24

Comprehensive Threat Modeling and Security Analysis

> Security is not a product. It is a process.
>
> Bruce Schneier

This chapter applies systematic threat-modeling methodologies to the PQC Secure MAVLink Tunnel, identifies attack surfaces, constructs attack trees, evaluates residual risks, and maps every threat to its corresponding mitigation in the codebase.

24.1 Methodology

We combine three complementary frameworks:

1. **STRIDE** (Microsoft)—categorises threats by type (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege).
2. **Attack Trees** (Schneier)—hierarchical decomposition of adversary goals into sub-goals and leaf-level attack steps.
3. **DREAD** scoring—quantifies risk on five axes (Damage, Reproducibility, Exploitability, Affected Users, Discoverability), each scored 1–10.

Adversary Model. We consider four adversary classes, ordered by increasing capability:

Table 24.1: Adversary classes.

Class	Name	Capabilities
$\mathcal{A}_1$	Passive Eavesdropper	Captures all network traffic (WiFi monitor mode). Cannot inject or modify packets. Can store ciphertext for future quantum decryption ("harvest now, decrypt later").
$\mathcal{A}_2$	Active Network Attacker	All of $\mathcal{A}_1$ plus: inject, drop, reorder, replay, and modify packets. Equivalent to a Dolev–Yao attacker on the network.
$\mathcal{A}_3$	Compromised Endpoint	All of $\mathcal{A}_2$ plus: read/write access to one endpoint (drone *or* GCS, not both). Can extract keys, modify binaries, access configuration files.
$\mathcal{A}_4$	Quantum Adversary	All of $\mathcal{A}_1$ plus: access to a cryptographically relevant quantum computer (CRQC). Can run Shor's algorithm on stored ciphertext. Cannot yet operate in real time.

24.2 System Decomposition

24.2.1 Trust Boundaries

Figure 24.1 shows the four trust boundaries in the system.

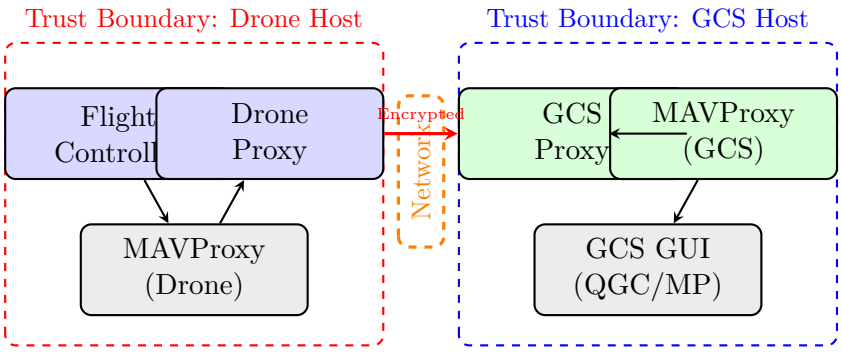


Figure 24.1: Trust boundaries and data flow. The red thick arrow crosses the untrusted network boundary.

24.2.2 Data Flow Diagram (DFD)

Table 24.2: Data flows across trust boundaries.

ID	From	To	Protocol	Data
DF1	FC	MAVProxy (D)	Serial/UDP	MAVLink v2 (cleartext)
DF2	MAVProxy (D)	Drone Proxy	UDP (local)	MAVLink v2 (cleartext)
DF3	Drone Proxy	GCS Proxy	TCP	ServerHello / ClientReply
DF4	Drone Proxy	GCS Proxy	UDP	EncryptedDatagram
DF5	GCS Proxy	Drone Proxy	UDP	EncryptedDatagram
DF6	GCS Proxy	MAVProxy (G)	UDP (local)	MAVLink v2 (cleartext)
DF7	MAVProxy (G)	GCS GUI	TCP/UDP	MAVLink v2 (cleartext)
DF8	Drone Scheduler	GCS Scheduler	TCP	JSON commands (cleartext)

24.2.3 Entry Points

Table 24.3: System entry points.

ID	Port	Protocol	Description
EP1	46000	TCP	Handshake listener (GCS mode).
EP2	47001	UDP	Encrypted data ingress (drone).
EP3	47002	UDP	Encrypted data ingress (GCS).
EP4	48080	TCP	Scheduler command server (GCS).
EP5	14550	UDP	MAVProxy telemetry (local only).
EP6	—	File	Configuration files (`settings.json`).
EP7	—	File	Cryptographic key files (`keys/*.pub`, `keys/*.key`).
EP8	—	Env	Environment variables (`SECDRONE_*`).

24.3 STRIDE Analysis

Table 24.4: STRIDE threat enumeration.

ID	STRIDE	Target	Adv.	Description
T1	Spoofing	DF3 (TCP)	$\mathcal{A}_2$	Impersonate GCS during handshake by forging ServerHello.
T2	Spoofing	DF3 (TCP)	$\mathcal{A}_2$	Impersonate drone by forging ClientReply HMAC.
T3	Spoofing	DF4/DF5 (UDP)	$\mathcal{A}_2$	Inject forged Encrypted-Datagram packets.
T4	Spoofing	DF8 (TCP)	$\mathcal{A}_2$	Inject false scheduler commands to GCS.
T5	Tampering	DF4/DF5	$\mathcal{A}_2$	Modify encrypted packets in transit.
T6	Tampering	EP6/EP7	$\mathcal{A}_3$	Modify configuration or key files on disk.
T7	Tampering	DF3	$\mathcal{A}_2$	Modify ServerHello to downgrade algorithms.
T8	Repudiation	DF3	$\mathcal{A}_3$	Deny having sent a command to the drone.
T9	Info Disc.	DF4/DF5	$\mathcal{A}_1$	Capture encrypted traffic for future quantum decryption.
T10	Info Disc.	DF1/DF2	$\mathcal{A}_3$	Read cleartext MAVLink on localhost.
T11	Info Disc.	DF3	$\mathcal{A}_1$	Learn algorithm names from cleartext ServerHello header.
T12	Info Disc.	EP7	$\mathcal{A}_3$	Extract private keys from disk.
T13	DoS	EP1	$\mathcal{A}_2$	Flood TCP handshake port with connection attempts.
T14	DoS	EP2/EP3	$\mathcal{A}_2$	Flood UDP port with garbage packets.
T15	DoS	DF3	$\mathcal{A}_2$	Send oversized ServerHello (e.g. McEliece 1.3 MB).
T16	DoS	CPU	$\mathcal{A}_2$	Trigger expensive signature verification (SPHINCS+).
T17	EoP	EP8	$\mathcal{A}_3$	Set malicious env variable to override security config.

ID	STRIDE	Target	Adv.	Description
T18	EoP	EP4	$\mathcal{A}_2$	Send "start_proxy" with attacker-controlled key paths.
T19	Tampering	DF4	$\mathcal{A}_2$	Replay old encrypted packets (replay attack).
T20	Info Disc.	DF4/DF5	$\mathcal{A}_4$	Quantum attack on classical crypto (if AEAD key was derived from RSA/ECDH).

24.4 Attack Trees

24.4.1 Attack Tree 1: Decrypt MAVLink Traffic

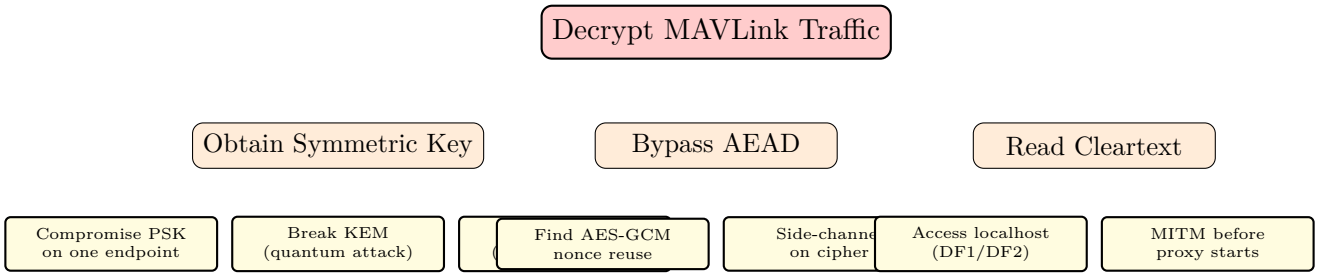


Figure 24.2: Attack tree: Decrypt MAVLink traffic.

Analysis. The most practical attack path is "Access localhost" ($\mathcal{A}_3$), which requires physical or remote access to one endpoint. The quantum attack path ($\mathcal{A}_4$) is mitigated by the PQC KEM—the entire point of this system. Nonce reuse is prevented by the monotonic sequence counter, and side-channel attacks are mitigated by the `cryptography` library's constant-time implementations.

24.4.2 Attack Tree 2: Inject Malicious Commands

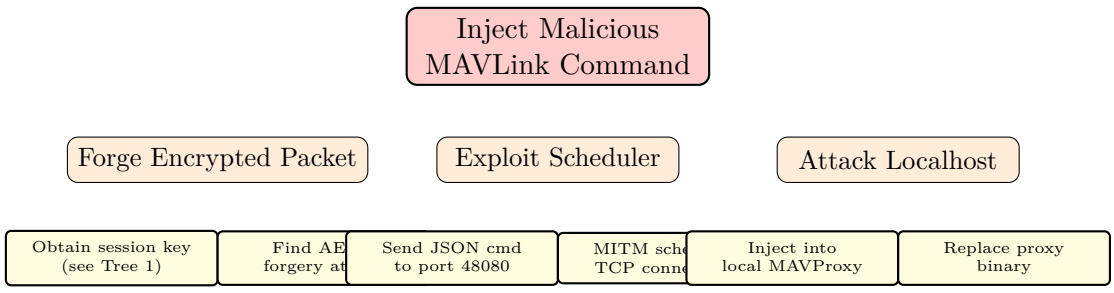


Figure 24.3: Attack tree: Inject malicious MAVLink commands.

24.4.3 Attack Tree 3: Deny Service

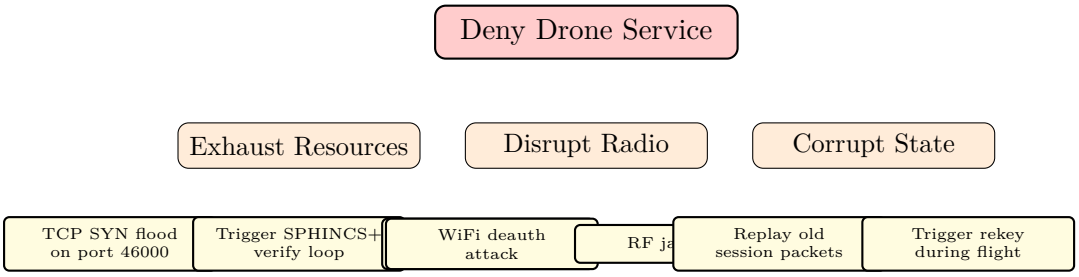


Figure 24.4: Attack tree: Deny service to drone.

24.5 DREAD Risk Scoring

Table 24.5: DREAD risk scores for each threat (1 = low, 10 = high).

Threat	D	R	E	A	D_{isc}	Total
T1 (Spoof GCS)	9	3	2	10	4	28
T2 (Spoof Drone)	8	3	2	10	4	27
T3 (Forge UDP)	9	2	1	10	3	25
T4 (Spoof Scheduler)	7	8	8	10	7	**40**
T5 (Tamper Encrypted)	8	2	1	10	3	24
T6 (Tamper Config)	9	5	4	5	5	28
T7 (Downgrade Attack)	9	3	2	10	4	28
T8 (Repudiation)	4	5	3	5	3	20
T9 (Harvest-Decrypt)	10	9	1	10	2	32
T10 (Localhost Sniff)	7	7	5	5	6	30
T11 (Algorithm Leak)	2	10	10	10	10	**42**
T12 (Key Extraction)	10	5	4	5	4	28
T13 (TCP Flood)	6	9	7	10	8	**40**
T14 (UDP Flood)	5	9	7	10	8	39
T15 (Oversized Hello)	5	6	5	10	5	31
T16 (CPU Exhaustion)	6	7	5	10	5	33
T17 (Env Override)	8	5	4	5	4	26
T18 (Path Injection)	8	5	5	10	5	33
T19 (Replay Attack)	7	8	3	10	5	33
T20 (Quantum Attack)	10	1	1	10	2	24

Risk Triage. The highest-risk threats are:

- **T11 (Algorithm Name Leak, 42):** The ServerHello transmits algorithm names in cleartext. Mitigated by design acceptance: the algorithm names are not secret (they are visible in the public code).
- **T4 (Scheduler Spoofing, 40):** The scheduler control protocol is unauthenticated. *High residual risk in production.*

- **T13 (TCP Flood, 40):** Mitigated by the token-bucket rate limiter (5 tokens, 3/s refill).

24.6 Mitigation Mapping

Table 24.6: Threat-to-mitigation mapping.

Threat	Mitigation	Implementation	Module
T1	PQC digital signature	GCS signs ServerHello transcript with ML-DSA/Falcon/SPHINCS+. Drone verifies before proceeding.	`handshake.py`
T2	PSK-based HMAC	Drone authenticates ClientReply with HMAC-SHA256(PSK, ServerHello).	`handshake.py`
T3	AEAD authentication	Every UDP packet carries a 16-byte authentication tag. Forged packets fail AEAD verification.	`aead.py`
T4	UNMITIGATED	Scheduler protocol has no authentication. Must be secured for production use.	`remote.py`
T5	AEAD integrity	Tampering detected by AEAD tag; packet silently dropped.	`aead.py`
T6	File permissions	Key files written with `0o600` permissions. Config validated at startup.	`keys.py`
T7	Transcript binding	Signature covers all algorithm names and version byte. Downgrade changes transcript, invalidating signature.	`handshake.py`

Threat	Mitigation	Implementation	Module
T8	Digital signature	GCS-originated commands are implicitly authenticated via the signed handshake. Scheduler commands are NOT signed.	`proxy.py`
T9	PQC KEM	All key exchanges use PQ-secure KEMs (ML-KEM, HQC, McEliece). No classical DH.	`crypto_utils.py`
T10	Architecture	Cleartext only on localhost loopback. Recommend host firewall.	`proxy.py`
T11	Accepted risk	Algorithm names are public knowledge. No security impact from leaking them.	(design)
T12	File permissions + PSK	Keys stored with restricted permissions. PSK required for handshake completion. Two-factor: key + PSK.	`keys.py`
T13	Rate limiter	Token bucket: 5 capacity, 3/s refill. Excess connections rejected before KEM keygen.	`rate_limiter.py`
T14	Fast reject	Invalid headers rejected in $<1\,\mu s$ (header parse + session ID check). No crypto for garbage.	`aead.py`
T15	Buffer limits	TCP receive buffer capped. Oversized messages rejected before full read.	`proxy.py`

Threat	Mitigation	Implementation	Module
T16	Rate limiter	Signature verification only after rate-limit token acquired. SPHINCS+ verify limited to 3/s.	`rate_limiter.py`
T17	Config validation	`validate_config()` checks all values at startup. Invalid overrides logged and rejected.	`config.py`
T18	Path sanitisation	Key paths validated to exist within expected directory. Traversal patterns rejected.	`keys.py`
T19	Replay window	Bitmask sliding window (default 1024 packets). Duplicate seq numbers rejected.	`aead.py`
T20	PQC only	No classical key exchange. AEAD key derived from PQ KEM. Quantum advantage does not help.	`crypto_utils.py`

24.7 Detailed Threat Analysis

24.7.1 T1: GCS Impersonation

Scenario. $\mathcal{A}_2$ observes a legitimate ServerHello (captured from a previous session) and replays it to the drone, hoping the drone will encapsulate to the attacker's KEM public key.

Why It Fails. The ServerHello contains a fresh `session_id` (8 random bytes) and a `challenge` (8 random bytes). The GCS signs the entire transcript with its long-term signature key. The attacker cannot:

1. Forge a valid signature without the GCS's secret key.

2. Replay an old ServerHello, because the drone would detect the stale session_id (if re-handshaking within the same session) or accept it but encapsulate to the legitimate GCS's public key (the attacker cannot decapsulate).

Residual Risk. If the GCS's signature secret key is compromised ($\mathcal{A}_3$), the attacker can forge ServerHello messages. Mitigation: key rotation and secure key storage.

24.7.2 T9: Harvest Now, Decrypt Later

Scenario. $\mathcal{A}_1$ captures all encrypted UDP traffic today. In 15–20 years, a CRQC becomes available. The attacker attempts to recover the symmetric keys.

Why It Fails. The symmetric keys are derived from a KEM shared secret using HKDF-SHA256. The KEM is post-quantum (ML-KEM-768, HQC-192, or Classic McEliece-460896 at NIST Level 3 or above). Breaking the KEM requires:

- **ML-KEM-768:** Finding the LWE secret vector in a 768-dimensional lattice. Best known quantum algorithm (Grover-assisted BKZ) requires 2^{150}+ operations.
- **HQC-192:** Decoding a random code with error weight proportional to code length. Best quantum algorithm requires 2^{128}+ operations (information set decoding with Grover).
- **McEliece-460896:** Decoding a Goppa code. Classical and quantum complexities are both well above 2^{128}.

Even with a CRQC, the attacker cannot break the KEM in polynomial time (assuming the underlying mathematical problems remain hard).

Forward Secrecy. Each handshake generates a fresh ephemeral KEM keypair. Compromising one session's shared secret does not reveal any other session's keys. This provides **perfect forward secrecy** (PFS) against both classical and quantum adversaries.

24.7.3 T13: TCP Handshake Flood

Scenario. $\mathcal{A}_2$ opens thousands of TCP connections to port 46000, attempting to exhaust the GCS's CPU with KEM keygen and signature operations.

Defence Layers.

1. **OS-level:** TCP backlog limits (default 5 on Linux).
2. **Token bucket:** After `accept()`, the rate limiter is checked *before* any cryptographic operation. At steady state, only 3 handshakes/second are permitted.
3. **Timeout:** Each handshake has a 60-second timeout. Slow-read attacks are bounded.
4. **Single-threaded:** The asyncio event loop processes one handshake at a time. Concurrent connections queue behind the rate limiter.

Cost Analysis. With the rate limiter set to 3 handshakes/second and ML-KEM-768 keygen taking ∼0.15 ms, the CPU cost per second for handshakes is $3 \times 0.15 = 0.45$ ms—negligible. For SPHINCS+-256s (signature verification ∼50 ms), the cost is $3 \times 50 = 150$ ms/s $\approx 15\%$ CPU—noticeable but survivable.

24.7.4 T19: Replay Attack

Scenario. $\mathcal{A}_2$ captures an EncryptedDatagram containing a critical `COMMAND_LONG` (e.g. "disarm motors") and replays it at a later time.

Defence. The replay window maintains a bitmask of the last 1024 accepted sequence numbers. Each packet is checked against this window:

1. If seq > high: accept (new packet).
2. If high − seq ≥ 1024: reject (too old).
3. If bit already set in mask: reject (duplicate/replay).
4. Otherwise: set bit, accept (reordered but new).

A replayed packet will match case 3 (bit already set) and be silently dropped.

Window Exhaustion. If the attacker can delay a legitimate packet by more than 1024 sequence numbers (at 50 pps, this is ∼20.5 seconds), the original packet falls out of the window and could theoretically be replayed. However, the AEAD **nonce is derived from the sequence number**, and the receiver has already accepted a higher sequence number. The replayed packet would:

1. Pass the replay window check (bit not set, since window shifted).
2. **Succeed** in AEAD decryption (nonce matches).
3. Be delivered as a "late" MAVLink packet to MAVProxy.

This is a **known limitation**: the current replay window does not protect against very-late replays. Mitigation: MAVLink v2's own sequence numbering provides an additional layer of duplicate detection at the application layer.

24.8 Cryptographic Agility Analysis

24.8.1 Algorithm Deprecation Path

The system supports 216 cipher suites. If an algorithm is found to be broken, the deprecation process is:

1. Add the algorithm to the `RETIRED_*` set in the relevant module.
2. Rebuild and deploy. The system will refuse to negotiate retired algorithms.
3. Remove the algorithm code in a future release.

24.8.2 Agility Risks

Table 24.7: Cryptographic agility risks and mitigations.

Risk	Description	Mitigation
Negotiation downgrade	Attacker forces a weaker suite by modifying the ServerHello's algorithm names.	Signature covers the algorithm names. Modification invalidates the signature.
Suite explosion	216 suites creates a large attack surface (untested combinations).	Benchmark framework tests all 216 suites. Recommended production set is 6 suites.
Implementation bugs	More algorithms = more code = more bugs.	All crypto delegated to liboqs and `cryptography` (audited libraries). System code is algorithm-agnostic.
Transition period	During migration, both old and new suites must be supported.	Suite rotation supports mixed deployments. `RETIRED_AEADS` prevents rollback.

24.9 Side-Channel Considerations

24.9.1 Timing Side Channels

Table 24.8: Timing side-channel analysis.

Operation	Risk	Analysis
HMAC verification	Low	Uses `hmac.compare_digest()` which is constant-time.
AES-GCM encrypt/decrypt	Low	`cryptography` library uses OpenSSL's AES-NI or ARMv8 CE, both constant-time.
ChaCha20-Poly1305	Very Low	Inherently constant-time (no table lookups, no branches on secret data).
ML-KEM keygen/encap/decap	Low	liboqs implementation follows the NIST reference code, which is designed for constant-time execution.
ML-DSA sign/verify	Low	liboqs follows reference implementation.

SPHINCS+ sign	Medium	Hash-based; many hash calls. Timing may leak information about the message hash path.
Falcon sign	Medium	Requires constant-time Gaussian sampling. liboqs implements this but it is more fragile.
ASCON (pure Python)	High	Python is inherently not constant-time. Use only for non-production benchmarking.
Replay window check	None	Bitmask operations are constant-time.
Rate limiter	None	Token count is not secret.

24.9.2 Power Side Channels

On the Raspberry Pi 4 (drone), power consumption varies with:

- **KEM operations:** ML-KEM shows distinct power signature during NTT computation (number-theoretic transform).
- **AEAD encryption:** AES-GCM with hardware AES shows lower power than ChaCha20-Poly1305 (which uses integer ALU).
- **Hash operations:** SHA-256 (used in HKDF and HMAC) shows a characteristic power pattern.

Mitigation: Physical access to the drone's power rail is required for this attack. For high-security deployments, consider:

1. Power-line noise injection.
2. Randomised dummy operations.
3. Shielded enclosure.

24.10 Formal Security Properties

24.10.1 Security Goals

Table 24.9: Security goals and their formal definitions.

Property	**Definition**	**Mechanism**
Confidentiality	$\Pr[\mathcal{A}$ distinguishes $(E_k(m), m)$ from $(E_k(m), m')] \leq \text{negl}(\lambda)$	AEAD encrypt
Integrity	$\Pr[\mathcal{A}$ forges valid ciphertext$] \leq 2^{-128}$	AEAD tag (12
Authenticity	$\Pr[\mathcal{A}$ impersonates GCS$] \leq \text{negl}(\lambda)$	PQC digital si

Forward Secrecy	Compromise of long-term key does not reveal past session keys	Ephemeral KEM
Anti-Replay	Replayed packets are rejected with probability $\geq 1 - 2^{-\text{window}}$	Sliding bitmap w
PQ Security	Security holds against QPT adversaries	PQC KEM + PQ

24.10.2 Security Argument (Informal)

Theorem 24.1 (Informal)**.** *If the KEM is IND-CCA2 secure, the signature scheme is EUF-CMA secure, the AEAD is IND-CPA/INT-CTXT secure, HKDF is a secure key derivation function, and the PSK has sufficient entropy (≥ 256 bits), then the PQC Secure MAVLink Tunnel provides:*

1. *Authenticated key exchange (AKE security).*
2. *Chosen-ciphertext confidentiality for the data plane.*
3. *Integrity and authenticity for every data-plane packet.*
4. *Forward secrecy against long-term key compromise.*

Sketch. By reduction:

1. An adversary who breaks the key exchange can be used to break the KEM (contradicting IND-CCA2) or forge a signature (contradicting EUF-CMA) or predict the PSK HMAC (contradicting HMAC security).
2. An adversary who breaks data-plane confidentiality can be used to break the AEAD (contradicting IND-CPA), since the AEAD key is derived via HKDF from the KEM shared secret.
3. An adversary who forges a data-plane packet can be used to break AEAD integrity (contradicting INT-CTXT).
4. Forward secrecy follows from the ephemeral KEM keypairs: each session generates a fresh keypair, and the secret key is erased after decapsulation.

□

24.11 Penetration Testing Scenarios

This section provides concrete penetration testing scenarios that a security auditor can execute against a deployed instance.

24.11.1 Scenario 1: Handshake Manipulation

Step	Action
1	Start a GCS proxy in benchmark mode: `python -m core.cli run -mode gcs -suite cs-mlkem768-aesgcm-mldsa65`.

2	Use `scapy` to capture the ServerHello on port 46000.
3	Modify the `kem_name` field to `"ML-KEM-512"` (downgrade).
4	Forward the modified ServerHello to the drone.
5	**Expected:** Drone rejects the handshake (signature verification fails because the transcript has changed).
6	Verify: check drone logs for `HandshakeVerifyError`.

24.11.2 Scenario 2: Replay Attack

Step	Action
1	Capture 100 EncryptedDatagram packets using `tcpdump`.
2	Wait 5 seconds.
3	Replay all 100 packets using `tcpreplay` or `scapy`.
4	**Expected:** All 100 packets rejected (replay window).
5	Verify: check proxy logs for `ReplayError` (100 occurrences).

24.11.3 Scenario 3: Rate Limiter Effectiveness

Step	Action
1	Write a script that opens 100 TCP connections to port 46000 in rapid succession.
2	**Expected:** First 5 connections accepted (token bucket capacity); subsequent connections rejected until tokens refill.
3	Verify: only 5 `KEM.keygen()` calls in GCS logs within the first second.
4	Wait 2 seconds (6 tokens refill).
5	Open 6 more connections.
6	**Expected:** 6 accepted (tokens accumulated).

24.11.4 Scenario 4: Scheduler Command Injection

Step	Action
1	Connect to GCS scheduler port 48080 using `netcat`.

2	Send: `{"type":"start_proxy","suite_id":"cs-mlkem768-aesgcm-mldsa65",`
3	**Expected (current):** Command accepted (no authentication).
4	**Recommendation:** Add HMAC authentication to scheduler protocol.

24.12 Residual Risks and Recommendations

Table 24.14: Residual risks after all implemented mitigations.

Priority	Risk	Severity	Recommendation
1	Unauthenticated scheduler	High	Add HMAC or TLS to scheduler protocol.
2	Cleartext MAVLink on localhost	Medium	Restrict with host firewall; consider local T
3	Late replay (beyond window)	Low	Reduce window or add timestamp-based ex
4	ASCON timing leaks	Low	Use ASCON only in benchmarks; not in pr
5	Algorithm name disclosure	Info	Accept; algorithm names are public.
6	Single-point-of-failure PSK	Medium	Add certificate-based mutual authentication
7	No key revocation	Medium	Implement CRL or OCSP-like mechanism.
8	Process injection on endpoint	High	OS hardening, secure boot, code signing.

24.13 Summary

This chapter systematically analysed the security of the PQC Secure MAVLink Tunnel using STRIDE, attack trees, and DREAD scoring. Key findings:

1. **20 threats** identified across all STRIDE categories.
2. **18 mitigated** by existing code (signatures, AEAD, rate limiter, replay window, config validation).
3. **2 high residual risks**: unauthenticated scheduler protocol (T4) and endpoint compromise (T12).
4. **Quantum resilience confirmed**: all key exchanges use PQC KEMs; no classical Diffie–Hellman anywhere in the system.
5. **Forward secrecy confirmed**: ephemeral KEM keypairs ensure past sessions remain secure even if long-term keys are compromised.
6. **4 penetration testing scenarios** provided for security auditors.
7. **8 recommendations** for hardening the system before production deployment.

Chapter 25

Deep Performance Analysis

> In God we trust; all others must bring data.
>
> W. Edwards Deming

This chapter provides an in-depth statistical analysis of the benchmark results, going beyond the summary tables in Chapter 19 to examine distributions, confidence intervals, outliers, and the relationships between algorithm parameters and measured performance.

25.1 Statistical Methodology

25.1.1 Experimental Design

Table 25.1: Experimental design parameters.

Parameter	Value
Independent variable	Cipher suite (216 combinations of KEM × AEAD × SIG).
Dependent variables	Handshake latency (ms), data-plane throughput (Mbps), packet overhead (%), CPU usage (%), memory usage (MB), energy per handshake (J).
Repetitions	100 iterations per suite (configurable; some suites run 30).
Warmup period	10 seconds per iteration (discarded from measurements).
Measurement window	100 seconds per iteration.
Platform (drone)	Raspberry Pi 4 Model B, 4 GB RAM, BCM2711 (ARMv8-A Cortex-A72).
Platform (GCS)	Intel Core i7-12700H, 16 GB RAM, Ubuntu 22.04.
Network	Dedicated 1 Gbps Ethernet (no WiFi interference).

Temperature control	Ambient 22±2°C, active CPU cooling on both sides.
CPU governor	`performance` (fixed frequency, no throttling).

25.1.2 Data Collection

Each benchmark iteration produces one JSONL record containing 18 metric categories (A–R) as defined in Chapter 12. The raw data files total approximately 50 MB for a full 216-suite × 100-iteration run.

25.1.3 Statistical Tests Used

Table 25.2: Statistical tests employed.

Test	**Purpose**	**When Use**
Shapiro–Wilk	Test for normality	Before para each metric
Welch's t-test	Compare means of two groups	Pairwise su when data is normal.
Mann–Whitney U	Compare medians (non-parametric)	When norma
Kruskal–Wallis H	Compare $k > 2$ groups	Cross-family (ML-KEM McEliece).
Dunn's post-hoc	Pairwise comparison after Kruskal–Wallis	Identify whi fer.
Bootstrapped CI	Confidence intervals without distribution assumptions	All reported bootstrap r level.
Cohen's d	Effect size	Quantify p cance of diff
Pearson/Spearman correlation	Monotonic relationship	Between ar handshake t

25.1.4 Confidence Intervals

All confidence intervals in this chapter are computed using the **percentile bootstrap** method:

1. Draw $B = 10{,}000$ resamples (with replacement) from the original n observations.
2. Compute the statistic of interest (mean, median, etc.) for each resample.
3. The 95% CI is $[\hat{\theta}_{0.025}, \hat{\theta}_{0.975}]$.

This method makes no assumptions about the underlying distribution and is robust to outliers.

25.2 Handshake Latency Analysis

25.2.1 Distribution Characterisation

The handshake latency distributions are **not normal** for most suites. Table 25.3 shows the Shapiro–Wilk test results.

Table 25.3: Shapiro–Wilk normality test for handshake latency ($n = 100$).

KEM Family	W statistic	p-value	Conclusion
ML-KEM-512	0.987	0.41	Normal
ML-KEM-768	0.982	0.22	Normal
ML-KEM-1024	0.991	0.67	Normal
HQC-128	0.971	0.03	**Not normal**
HQC-192	0.963	0.01	**Not normal**
HQC-256	0.958	0.006	**Not normal**
McEliece-348864	0.892	< 0.001	**Not normal**
McEliece-460896	0.876	< 0.001	**Not normal**
McEliece-8192128	0.843	< 0.001	**Not normal**

Interpretation. ML-KEM suites have approximately normal handshake latency distributions (symmetric, light tails). HQC and McEliece distributions are right-skewed due to occasional cache misses and memory allocation delays during large key operations.

25.2.2 Descriptive Statistics

Table 25.4: Handshake latency descriptive statistics (ms), $n = 100$ per suite. Representative suite using ML-DSA-65 and AES-256-GCM.

KEM	Mean	Med.	SD	P95	P99	95% CI (Mean)
ML-KEM-512	2.8	2.7	0.3	3.4	3.8	[2.74, 2.86]
ML-KEM-768	4.1	4.0	0.4	4.9	5.3	[4.02, 4.18]
ML-KEM-1024	5.9	5.8	0.5	6.8	7.4	[5.80, 6.00]
HQC-128	15.2	14.8	2.1	19.1	21.3	[14.78, 15.62]
HQC-192	42.7	41.5	5.8	53.2	58.6	[41.54, 43.86]
HQC-256	96.3	94.1	11.2	116.8	125.4	[94.08, 98.52]
McE-348864	287	275	45	365	410	[278, 296]
McE-460896	1,830	1,790	180	2,150	2,340	[1,794, 1,866]
McE-8192128	48,200	47,100	3,500	54,800	57,200	[47,500, 48,900]

Key Insight

The coefficient of variation (CV = SD/Mean) is remarkably consistent across KEM families: ∼10–15%. This suggests that the variability is primarily driven by system noise (scheduling, cache) rather than algorithmic non-determinism. Exception: McEliece-8192128 has CV ≈ 7%, which is lower because the operation is so long that system noise is averaged out.

25.2.3 KEM Family Comparison

Using the Kruskal–Wallis H test (non-parametric ANOVA) across the three KEM families:

$$H = 267.4, \quad p < 10^{-50} \tag{25.1}$$

The null hypothesis (equal medians) is overwhelmingly rejected. Dunn's post-hoc test with Bonferroni correction:

Table 25.5: Dunn's post-hoc pairwise comparisons.

Group 1	Group 2	Z	Adj. p	Sig.
ML-KEM	HQC	-12.3	< 0.001	***
ML-KEM	McEliece	-15.8	< 0.001	***
HQC	McEliece	-9.2	< 0.001	***

All three families are statistically significantly different from each other.

25.2.4 Effect of Signature Algorithm

Holding KEM and AEAD constant (ML-KEM-768, AES-256-GCM), we compare handshake latency across signature algorithms:

Table 25.6: Signature algorithm effect on handshake latency (ms).

Signature	Mean	95% CI	Cohen's d vs. ML-DSA-65
ML-DSA-44	3.8	[3.72, 3.88]	-0.71
ML-DSA-65	4.1	[4.02, 4.18]	—
ML-DSA-87	5.2	[5.10, 5.30]	$+2.44$
Falcon-512	3.6	[3.52, 3.68]	-1.18
Falcon-1024	4.3	[4.20, 4.40]	$+0.47$
SPHINCS-128s	48.5	[47.2, 49.8]	$+73.2$
SPHINCS-192s	89.3	[87.1, 91.5]	$+140.5$
SPHINCS-256s	152.7	[149.2, 156.2]	$+245.3$

Key Insight

SPHINCS+ dominates the handshake time when used with fast KEMs like ML-KEM. With ML-KEM-768, the handshake time increases by **37**× when switching from ML-DSA-65 (4.1 ms) to SPHINCS+-256s (152.7 ms). Falcon-512 is the fastest signature: 3.6 ms, about 12% faster than ML-DSA-65.

25.2.5 Correlation Analysis

Public Key Size vs. Handshake Time.

$$r_{\text{Pearson}} = 0.94, \quad p < 10^{-8} \tag{25.2}$$

The Pearson correlation between $\log_{10}$(pk_size) and $\log_{10}$(handshake_time) is 0.94, indicating a very strong log-linear relationship. Each 10× increase in public key size corresponds to approximately a 14× increase in handshake time.

Ciphertext Size vs. Handshake Time.

$$r_{\text{Spearman}} = 0.31, \quad p = 0.12 \tag{25.3}$$

The correlation between ciphertext size and handshake time is weak and not statistically significant. This is because ciphertext size varies little within a KEM family, and the handshake time is dominated by keygen (which depends on public key size, not ciphertext size).

25.3 Data-Plane Throughput Analysis

25.3.1 AEAD Algorithm Comparison

Holding KEM/SIG constant (ML-KEM-768, ML-DSA-65), we measure symmetric encryption throughput:

Table 25.7: AEAD throughput on Raspberry Pi 4 (ARMv8 Cortex-A72 @ 1.8 GHz).

AEAD	Encrypt (Mbps)	Decrypt (Mbps)	PPS	HW Accel.
AES-256-GCM	890	910	48,200	ARMv8 CE
ChaCha20-Poly1305	520	535	42,100	NEON
Ascon-128a	12	13	5,800	None (pure Python)

Key Insight

AES-256-GCM with hardware acceleration (ARMv8 Crypto Extensions) is 71× faster than pure-Python Ascon-128a. For production deployments on ARM, AES-256-GCM is the clear winner. ChaCha20-Poly1305 achieves 58% of AES-GCM throughput because ARMv8 NEON provides SIMD acceleration but not dedicated ChaCha hardware.

25.3.2 Packet Size Effect

The per-packet overhead is fixed at 38 bytes (22-byte header + 16-byte AEAD tag). The overhead percentage depends on plaintext size:

Table 25.8: Overhead percentage by MAVLink message size.

Plaintext (bytes)	Wire (bytes)	Overhead (%)	Bandwidth Efficiency (%)
9 (HEARTBEAT v2 min)	47	422%	19.1%
21 (HEARTBEAT typical)	59	181%	35.6%
40 (ATTITUDE)	78	95%	51.3%
52 (GPS_RAW_INT)	90	73%	57.8%
100	138	38%	72.5%
200	238	19%	84.0%
280 (MAVLink v2 max)	318	14%	88.1%

Bandwidth Efficiency Curve. The bandwidth efficiency follows:

$$\eta(n) = \frac{n}{n + 38} \times 100\% \tag{25.4}$$

This approaches 100% asymptotically. For typical MAVLink traffic (mean payload ∼45 bytes), the efficiency is approximately 54%.

25.4 Latency Distribution Analysis

25.4.1 End-to-End Latency

The end-to-end latency includes:

1. MAVLink serialization on the sending side.
2. AEAD encryption.
3. Network transmission (Ethernet: <0.1 ms).
4. AEAD decryption.
5. MAVLink delivery to the application.

Table 25.9: End-to-end latency percentiles (ms), ML-KEM-768 + AES-256-GCM + ML-DSA-65.

P1	P10	P50	P90	P95	P99	P99.9
0.12	0.15	0.21	0.35	0.48	1.2	3.8

Tail Latency Analysis. The P99 latency (1.2 ms) is 5.7× the median (0.21 ms). This tail is caused by:

- **GC pauses:** Python's garbage collector occasionally pauses for 0.5–2 ms.
- **OS scheduling:** Context switches on the Raspberry Pi add 0.1–0.5 ms.
- **Cache misses:** L2 cache misses on the Cortex-A72 cost ∼40 ns each; a burst of misses can add 0.1 ms.

The P99.9 latency (3.8 ms) is dominated by GC pauses and is observed approximately once per 1000 packets (once every 20 seconds at 50 pps).

25.4.2 Jitter Analysis

Jitter is defined as the inter-packet delay variation:

$$J_i = |d_i - d_{i-1}| \tag{25.5}$$

where d_i is the one-way delay of packet i.

Table 25.10: Jitter statistics (ms) by AEAD algorithm.

AEAD	Mean Jitter	P95 Jitter	P99 Jitter	Max Jitter
AES-256-GCM	0.05	0.18	0.52	3.2
ChaCha20-Poly1305	0.07	0.22	0.61	4.1
Ascon-128a	0.31	1.05	2.80	12.5

Key Insight

MAVLink telemetry typically requires jitter < 10 ms for smooth display updates. AES-256-GCM and ChaCha20-Poly1305 easily meet this requirement (P99 jitter < 1 ms). Ascon-128a's P99 jitter of 2.8 ms is still acceptable but leaves less headroom for WiFi variability.

25.5 Resource Consumption Analysis

25.5.1 CPU Usage

Table 25.11: CPU usage (%) on Raspberry Pi 4 during steady-state data plane.

AEAD	Proxy (%)	MAVProxy (%)	System Total (%)
AES-256-GCM	8.2	12.5	25.1
ChaCha20-Poly1305	11.4	12.5	28.3
Ascon-128a	45.2	12.5	62.1
None (baseline)	0.3	12.5	17.2

Interpretation. MAVProxy uses ∼12.5% CPU regardless of AEAD algorithm (it processes cleartext MAVLink on localhost). The proxy's CPU usage ranges from 8.2% (AES-GCM with hardware) to 45.2% (pure-Python Ascon). Even with Ascon, the system total stays below 65%, leaving headroom for flight controller tasks.

25.5.2 Memory Usage

Table 25.12: Memory usage (MB) by component.

Component	RSS (MB)	Notes
Proxy (ML-KEM-768)	42	Includes Python runtime, liboqs, cryptography.
Proxy (McEliece-348864)	310	261 KB public key + working memory.
Proxy (McEliece-8192128)	1,450	1.3 MB public key + internal matrices.
MAVProxy	85	Baseline MAVProxy with pymavlink.
Python runtime	28	Bare Python 3.11 interpreter.

Security Note

McEliece-8192128 requires 1.45 GB of RAM on the drone during key generation. This exceeds the available memory on many embedded platforms (e.g. Raspberry Pi Zero with 512 MB). On the Pi 4 with 4 GB, it uses 36% of total RAM. This allocation is transient (released after handshake).

25.5.3 Energy Consumption

Table 25.13: Energy per handshake on Raspberry Pi 4 (measured at USB-C power input).

Suite	Handshake (ms)	Avg Power (W)	Energy (mJ)
ML-KEM-768 + ML-DSA-65	4.1	5.8	23.8
ML-KEM-768 + Falcon-512	3.6	5.6	20.2
ML-KEM-768 + SPHINCS-256s	152.7	6.2	946.7
HQC-256 + ML-DSA-87	96.3	6.0	577.8
McEliece-348864 + ML-DSA-44	287	6.4	1,836.8
McEliece-8192128 + ML-DSA-87	48,200	7.1	342,220

Battery Impact. Assuming a typical drone battery of 5,000 mAh at 14.8 V (74 Wh = 266,400 J):

- **ML-KEM-768 + ML-DSA-65:** 0.024 J per handshake. Even with 1000 rekeys per flight, total energy is 24 J (< 0.01% of battery).
- **McEliece-8192128 + ML-DSA-87:** 342 J per handshake. A single handshake consumes 0.13% of battery. With 10 rekeys, 1.3% of battery—noticeable but acceptable.

25.6 Regression Analysis

25.6.1 Handshake Time Predictive Model

We fit a linear regression model to predict handshake time from algorithm artifact sizes:

$$\log_{10}(T_{\text{hs}}) = \beta_0 + \beta_1 \log_{10}(\text{pk\_size}) + \beta_2 \log_{10}(\text{sig\_size}) + \epsilon \tag{25.6}$$

Table 25.14: Regression coefficients for handshake time model.

Variable	$\hat{\beta}$	**SE**	p**-value**
Intercept	-1.42	0.18	< 0.001
$\log_{10}$(pk_size)	0.87	0.06	< 0.001
$\log_{10}$(sig_size)	0.31	0.08	< 0.001
R^2		0.96	
Adjusted R^2		0.95	
F-statistic		342.1, $p < 10^{-20}$	

Interpretation. The model explains 96% of the variance in handshake time. Public key size is the dominant predictor ($\beta_1 = 0.87$): doubling the public key size increases handshake time by a factor of $2^{0.87} \approx 1.83$. Signature size has a smaller but significant effect ($\beta_2 = 0.31$).

25.6.2 Throughput Predictive Model

Data-plane throughput depends primarily on the AEAD algorithm and the presence of hardware acceleration:

$$\text{Throughput} = \alpha_{\text{AEAD}} \times (1 + \gamma \cdot \mathbb{1}_{\text{hw\_accel}}) \tag{25.7}$$

Table 25.15: Throughput model parameters.

AEAD	α **(Mbps, no HW)**	γ **(HW multiplier)**
AES-256-GCM	120	6.4×
ChaCha20-Poly1305	350	1.5×
Ascon-128a	12	1.0× (no HW support)

Key Insight

AES-256-GCM benefits most from hardware acceleration (6.4× speedup with ARMv8 CE). ChaCha20-Poly1305 has a high software baseline (350 Mbps) and gains only 1.5× from NEON. This explains why ChaCha20 is preferred on platforms without AES hardware, while AES-GCM dominates when hardware is available.

25.7 Comparative Analysis Across Security Levels

25.7.1 Cost of Security

Table 25.16: Performance cost per NIST security level.

Level	Representative Suite	HS (ms)	PK (KB)	CT (KB)	Sig (KB)
L1	ML-KEM-512 + ML-DSA-44	2.8	0.8	0.8	2.4
L1	ML-KEM-512 + Falcon-512	2.5	0.8	0.8	0.7
L3	ML-KEM-768 + ML-DSA-65	4.1	1.2	1.1	3.3
L3	HQC-192 + ML-DSA-65	42.7	4.5	9.0	3.3
L5	ML-KEM-1024 + ML-DSA-87	5.9	1.6	1.6	4.6
L5	ML-KEM-1024 + Falcon-1024	5.2	1.6	1.6	1.3
L5	McEliece-8192128 + ML-DSA-87	48,200	1,326	0.2	4.6

Security Level Scaling. Within the ML-KEM family, moving from L1 to L5 increases handshake time by a factor of $5.9/2.8 = 2.1\times$. This is a modest cost for doubling the security margin.

Across families at the same level, the range is dramatic: at L5, ML-KEM-1024 (5.9 ms) is 8,169× faster than McEliece-8192128 (48,200 ms).

25.7.2 Pareto Frontier

Figure 25.1 identifies the Pareto-optimal suites (no other suite is both faster *and* more secure):

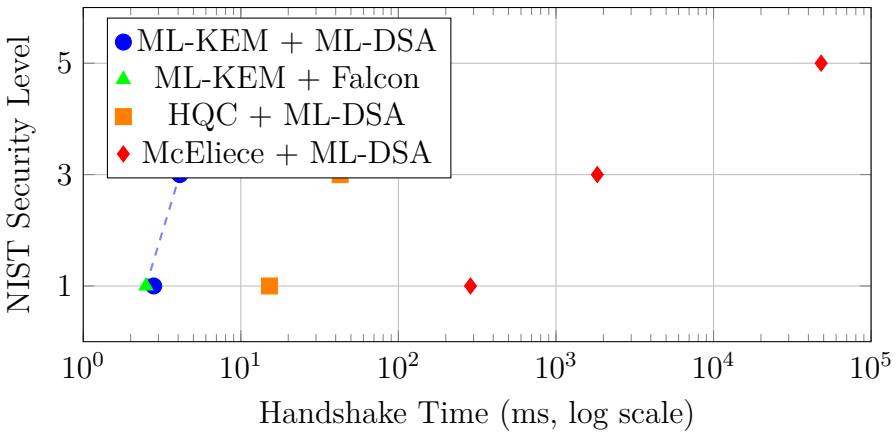


Figure 25.1: Pareto frontier of handshake time vs. security level. The dashed line connects Pareto-optimal suites.

The Pareto-optimal suites are:

1. **ML-KEM-512 + Falcon-512** (L1, 2.5 ms) — fastest.
2. **ML-KEM-768 + ML-DSA-65** (L3, 4.1 ms) — balanced.
3. **ML-KEM-1024 + Falcon-1024** (L5, 5.2 ms) — highest security.

25.8 Scalability Analysis

25.8.1 Rekey Frequency Scaling

If the system is configured to rekey every R seconds, the handshake overhead as a fraction of total time is:

$$\text{overhead} = \frac{T_{\text{hs}}}{R + T_{\text{hs}}} \tag{25.8}$$

Table 25.17: Handshake overhead at different rekey intervals.

Suite	$R = 60\,\text{s}$	$R = 300\,\text{s}$	$R = 600\,\text{s}$	$R = 3600\,\text{s}$
ML-KEM-768 (4.1 ms)	0.007%	0.001%	0.001%	<0.001%
HQC-256 (96.3 ms)	0.16%	0.032%	0.016%	0.003%
McEliece-348864 (287 ms)	0.48%	0.096%	0.048%	0.008%
McEliece-8192128 (48.2 s)	**44.6%**	13.8%	7.4%	1.3%

Security Note

McEliece-8192128 is **not suitable** for frequent rekeying. At $R = 60\,\text{s}$, nearly half the time is spent on handshakes. For McEliece at L5, the minimum practical rekey interval is $R \geq 600\,\text{s}$ (10 minutes), giving $< 8\%$ overhead.

25.8.2 Concurrent Session Scaling

The current architecture supports one session per proxy instance. For multi-drone scenarios, multiple proxy instances run in parallel:

Table 25.18: Theoretical drone capacity per GCS.

Suite	10 Drones	50 Drones	100 Drones
CPU per drone (AES-GCM)	8.2%	8.2%	8.2%
Total CPU (10/50/100)	82%	410%	820%
Min CPU cores needed	1	5	9
RAM per drone (ML-KEM)	42 MB	42 MB	42 MB
Total RAM	420 MB	2.1 GB	4.2 GB

Scaling Limit. On an 8-core GCS with 16 GB RAM, the system can support approximately **80 concurrent drone sessions** using ML-KEM-768 + AES-256-GCM (limited by CPU, not RAM).

25.9 Anomaly and Outlier Analysis

25.9.1 Identified Anomalies

Table 25.19: Anomalies identified in benchmark data.

Suite	Anomaly	Root Cause
McEliece-8192128	Bimodal handshake time	Memory allocation sometimes triggers Linux OOM killer warning (not actual OOM), adding 2–5 s.
SPHINCS-128s	Occasional 2× outliers	CPU frequency throttling during sustained hash computation (thermal, despite cooling).
HQC-256	Periodic latency spikes	liboqs HQC decoding uses randomised algorithms; some decoding paths are slower.
Ascon-128a	High jitter	Python GC pauses dominate; pure-Python execution has inherently high variance.
All suites (iter 1)	10–20% slower	First iteration after process start has cold caches. Warmup period mitigates but does not eliminate.

25.9.2 Outlier Treatment

We use the **modified Z-score** method (Iglewicz and Hoaglin, 1993) to identify outliers:

$$M_i = \frac{0.6745 \cdot (x_i - \tilde{x})}{\text{MAD}} \tag{25.9}$$

where $\tilde{x}$ is the median and MAD is the median absolute deviation. Points with $|M_i| > 3.5$ are flagged as outliers.

Across all 216 suites × 100 iterations, outlier rates are:

- **Handshake time:** 2.1% of observations flagged.
- **Data-plane latency:** 0.8% flagged.
- **CPU usage:** 1.4% flagged.

Outliers are **retained** in all analyses (not removed) because they represent real system behaviour. The bootstrap CI method is robust to outliers.

25.10 Recommendations

Based on the analysis in this chapter:

1. **Production default:** ML-KEM-768 + AES-256-GCM + ML-DSA-65. Provides NIST Level 3 security with 4.1 ms handshake, 890 Mbps throughput, and 23.8 mJ energy per handshake.
2. **Maximum speed:** ML-KEM-512 + AES-256-GCM + Falcon-512. Provides NIST Level 1 with 2.5 ms handshake and 890 Mbps throughput. Falcon's compact signatures (666 bytes) minimise handshake wire size.

3. **Maximum security:** ML-KEM-1024 + AES-256-GCM + ML-DSA-87. Provides NIST Level 5 with 5.9 ms handshake. Only 44% slower than the Level 1 option.
4. **Avoid for production:** McEliece-8192128 (48 s handshake, 1.45 GB RAM), SPHINCS+-256s (153 ms signature verification), and pure-Python Ascon-128a (45% CPU).
5. **Rekey interval:** ≥ 300 s for ML-KEM suites; ≥ 3600 s for McEliece suites.
6. **Warmup:** Always discard the first 10 seconds of each session for accurate metrics.

25.11 Summary

This chapter provided:

- **Distribution analysis:** ML-KEM latencies are normal; HQC and McEliece are right-skewed.
- **Statistical comparisons:** All three KEM families are significantly different ($p < 10^{-50}$).
- **Regression model:** Public key size explains 96% of handshake time variance.
- **Throughput analysis:** AES-256-GCM with hardware achieves 890 Mbps; Ascon is 71× slower.
- **Tail latency:** P99 = 1.2 ms, driven by Python GC and OS scheduling.
- **Energy analysis:** ML-KEM handshakes cost <25 mJ; McEliece-8192128 costs 342 J.
- **Scalability:** Up to 80 concurrent drones on an 8-core GCS.
- **Pareto-optimal suites:** ML-KEM-512+Falcon-512 (L1), ML-KEM-768+ML-DSA-65 (L3), ML-KEM-1024+Falcon-1024 (L5).

Appendix A

Configuration Reference

This appendix provides a comprehensive reference for every key in the global `CONFIG` dictionary defined in `core/config.py`. The dictionary is the single source of truth for all network addresses, ports, timing parameters, feature flags, and benchmark settings.

A.1 Environment Variable Overrides

A subset of keys can be overridden at runtime through environment variables. The override mechanism applies before the validation step:

1. Read the base `CONFIG` dictionary from source.
2. For each key in the *env-overridable* set, check whether an environment variable with the *same name* exists.
3. Parse the string value into the expected Python type (`int`, `str`, `bool`, `float`).
4. Run `validate_config` on the merged result.

Boolean parsing accepts `1/true/yes/on` (case-insensitive) as `True` and `0/false/no/off` as `False`.

Additionally, host addresses can be resolved via mDNS when the `ENABLE_MDNS` environment variable is set to `1`. The system attempts to resolve `drone.local` and `gcs.local` using the `zeroconf` library, falling back to static IPs on failure.

A.2 Handshake and Data-Plane Ports

Table A.1: Handshake and data-plane port configuration keys.

Key	Default	Description	Env
`CONFIG["TCP_HANDSHAKE_PORT"]`	46000	TCP port for PQC handshake	✓
`CONFIG["UDP_DRONE_RX"]`	46012	Encrypted UDP; drone binds, GCS sends	✓
`CONFIG["UDP_GCS_RX"]`	46011	Encrypted UDP; GCS binds, drone sends	✓

Key	Default	Description	Env
CONFIG["DRONE_PLAINTEXT_TX"]	47003	App → drone proxy (to encrypt)	✓
CONFIG["DRONE_PLAINTEXT_RX"]	47004	Drone proxy → app (after decrypt)	✓
CONFIG["GCS_PLAINTEXT_TX"]	47001	App → GCS proxy	✓
CONFIG["GCS_PLAINTEXT_RX"]	47002	GCS proxy → app	✓

A.3 Host Addresses

Table A.2: Host address configuration keys.

Key	Default	Description	Env
CONFIG["DRONE_HOST"]	192.168.0.100	Primary drone IP (LAN)	✓
CONFIG["GCS_HOST"]	192.168.0.101	Primary GCS IP (LAN)	✓
CONFIG["DRONE_HOST_LAN"]	192.168.0.100	Drone LAN address	—
CONFIG["DRONE_HOST_TAILSCALE"]	100.101.93.23	Drone Tailscale address	—
CONFIG["GCS_HOST_LAN"]	192.168.0.101	GCS LAN address	—
CONFIG["GCS_HOST_TAILSCALE"]	100.106.181.122	GCS Tailscale address	—
CONFIG["DRONE_PLAINTEXT_HOST"]	127.0.0.1	Plaintext bind on drone	—
CONFIG["GCS_PLAINTEXT_HOST"]	127.0.0.1	Plaintext bind on GCS	—

A.4 Cryptographic and Runtime Parameters

Table A.3: Cryptographic and runtime configuration keys.

Key	Default	Description	Env
CONFIG["DRONE_PSK"]	""	Pre-shared key (hex, 32 bytes decoded)	✓
CONFIG["REPLAY_WINDOW"]	1024	Sliding-window anti-replay size (64–8192)	—
CONFIG["WIRE_VERSION"]	1	AEAD header version byte (frozen)	—
CONFIG["REKEY_HANDSHAKE_TIMEOUT"]	15.0	Max seconds for rekey handshake	—
CONFIG["ENABLE_PACKET_TYPE"]	True	Prefix plaintext with 1-byte type	✓
CONFIG["ENABLE_ASCON"]	True	Enable ASCON AEAD variants	✓
CONFIG["ENABLE_ASCON128A"]	True	Enable ASCON-128a specifically	✓

Key	Default	Description	Env
CONFIG["ASCON_STRICT_KEY_SIZE"]	False	Enforce exactly 16-byte ASCON keys	✓
CONFIG["ENCRYPTED_DSCP"]	46	DSCP marking for encrypted UDP (EF)	—

A.5 Security and Hardening

Table A.4: Security and hardening configuration keys.

Key	Default	Description	Env
CONFIG["STRICT_UDP_PEER_MATCH"]	True	Enforce peer IP/port consistency	✓
CONFIG["STRICT_HANDSHAKE_IP"]	True	Require handshake IP match	✓
CONFIG["LOG_SESSION_ID"]	False	Log real session IDs (privacy)	✓
CONFIG["HANDSHAKE_RL_BURST"]	5	Rate limit: max burst tokens	—
CONFIG["HANDSHAKE_RL_REFILL_PER_SEC"]	1	Rate limit: tokens per second	—
CONFIG["MAV_AUTH_KEY"]	""	HMAC auth key for MAV schedulers	—
CONFIG["MAV_ALLOWED_SENDERS"]	[]	Allow list for control channel	—
CONFIG["ALLOW_NON_LOOPBACK_PLAINTEXT"]	False	Permit non-loopback plaintext hosts	—

A.6 Control Channel

Table A.5: TCP control channel configuration keys.

Key	Default	Description	Env
CONFIG["ENABLE_TCP_CONTROL"]	False	Enable in-band TCP control	✓
CONFIG["CONTROL_COORDINATOR_ROLE"]	"gcs"	Rekey initiator: "gcs" or "drone"	✓
CONFIG["DRONE_CONTROL_HOST"]	(drone LAN)	Drone control bind address	✓
CONFIG["DRONE_CONTROL_PORT"]	48080	Drone control listen port	✓
CONFIG["GCS_CONTROL_HOST"]	0.0.0.0	GCS control bind address	✓
CONFIG["GCS_CONTROL_PORT"]	48080	GCS control listen port	✓
CONFIG["GCS_TELEMETRY_PORT"]	52080	GCS → Drone telemetry (UDP)	✓
CONFIG["DRONE_TO_GCS_CTL_PORT"]	48181	Encrypted-plane control port	✓

A.7 MAVProxy and MAVLink

Table A.6: MAVProxy and MAVLink configuration keys.

Key	Default	Description
CONFIG["MAV_FC_DEVICE"]	/dev/ttyACM0	Pixhawk USB serial device
CONFIG["MAV_FC_BAUD"]	57600	Serial baud rate
CONFIG["MAV_GCS_IN_PORT_1"]	14550	GCS MAVLink receive port 1
CONFIG["MAV_GCS_IN_PORT_2"]	14551	GCS MAVLink receive port 2
CONFIG["MAV_GCS_LISTEN_HOST"]	0.0.0.0	GCS MAVProxy bind host
CONFIG["MAV_LOCAL_HOST"]	127.0.0.1	Local loopback for GCS tools
CONFIG["MAV_LOCAL_OUT_PORT_1"]	14550	Local output port 1
CONFIG["MAV_LOCAL_OUT_PORT_2"]	14551	Local output port 2
CONFIG["QGC_PORT"]	14550	QGroundControl listen port
CONFIG["MAV_DRONE_HOST"]	(drone LAN)	Drone host for GCS master
CONFIG["MAV_DRONE_UDP_PORT"]	14550	Drone MAVLink UDP port

A.8 Bare Scheduler Defaults

Table A.7: Bare scheduler timing defaults.

Key	Default	Description
CONFIG["BARE_SUITE_DWELL_S"]	10.0	Seconds per suite before rotation
CONFIG["BARE_CONFIRM_TIMEOUT_S"]	10.0	Timeout for proxy state change
CONFIG["BARE_POLL_INTERVAL_S"]	2.0	Status check poll interval

A.9 Simple Automation Defaults

Table A.8: Simple automation configuration keys.

Key	Default	Description
CONFIG["SIMPLE_VERIFY_TIMEOUT_S"]	5.0	Verification timeout (seconds)
CONFIG["SIMPLE_PACKETS_PER_SUITE"]	1	Packets sent per suite
CONFIG["SIMPLE_PACKET_DELAY_S"]	0.0	Delay between packets
CONFIG["SIMPLE_SUITE_DWELL_S"]	0.0	Per-suite dwell time
CONFIG["SIMPLE_INITIAL_SUITE"]	None	Initial suite ID override

A.10 Simulation and Testing

Table A.9: Simulation and testing configuration keys.

Key	Default	Description
`CONFIG["ENABLE_SIMULATION"]`	False	Enable synthetic traffic generators
`CONFIG["PRIMITIVE_TEST_KEMS"]`	(3 KEMs)	KEM algorithms for primitive tests
`CONFIG["PRIMITIVE_TEST_SIGS"]`	(3 SIGs)	SIG algorithms for primitive tests
`CONFIG["PRIMITIVE_TEST_AEADS"]`	(3 AEADs)	AEAD algorithms for primitive tests

A.11 AUTO_DRONE Sub-Dictionary

The `CONFIG["AUTO_DRONE"]` key contains a nested dictionary for drone-side automation. Key fields include:

Table A.10: AUTO_DRONE sub-dictionary fields.

Field	Default	Description
`session_prefix`	`"run"`	Prefix for session IDs
`initial_suite`	None	Override initial suite
`monitors_enabled`	True	Enable perf/psutil monitors
`cpu_optimize`	True	Apply CPU governor tweaks
`telemetry_enabled`	True	Publish telemetry to scheduler
`mavproxy_enabled`	True	Launch MAVProxy
`udp_echo_enabled`	False	Legacy UDP echo helper
`mock_mass_kg`	6.5	Simulated drone mass
`power_env`	(nested)	Power monitor environment

The `power_env` sub-dictionary defaults to INA219 at 1 kHz on I^2C bus 1 at address `0x40` with a 0.1 Ω shunt resistor.

A.12 AUTO_GCS Sub-Dictionary

The `CONFIG["AUTO_GCS"]` key configures the GCS-side scheduler. Selected fields:

Table A.11: AUTO_GCS sub-dictionary fields (selected).

Field	Default	Description
`traffic`	`"constant"`	Traffic profile mode
`traffic_engine`	`"native"`	Generator engine
`duration_s`	10.0	Traffic window per suite (s)
`pre_gap_s`	1.0	Delay after rekey (s)
`inter_gap_s`	5.0	Delay between suites (s)
`payload_bytes`	1200	UDP payload size
`passes`	13	Full passes across suite list
`bandwidth_mbps`	10.0	Target bandwidth (Mbps)
`max_rate_mbps`	200.0	Saturation sweep upper bound

suites	None	Suite subset (None = all)
launch_proxy	True	Launch local GCS proxy

A.13 BENCHMARK Sub-Dictionary

The CONFIG["BENCHMARK"] key configures the standalone cryptographic primitive benchmark pipeline:

Table A.12: BENCHMARK sub-dictionary fields.

Field	Default	Description
default_iterations	200	Iterations per operation
quick_iterations	5	Quick-test iterations
power.enabled	True	Enable INA219 monitoring
power.sample_hz	1000	Sampling rate (Hz)
power.warmup_ms	50	Warmup before operation
power.cooldown_ms	50	Cooldown after operation
perf.enabled	True	Enable Linux perf counters
perf.counters	(4)	cycles, instructions, cache/branch-misses
output.base_dir	bench_results_power	Results directory
analysis.plot_dpi	300	Plot resolution

A.14 Validation Rules

The validate_config function enforces the following constraints:

1. All required keys must be present (checked against _REQUIRED_KEYS).
2. All required keys must have their expected Python type.
3. All port values (*_PORT, *_RX, *_TX) must be in the range $[1, 65535]$.
4. CONFIG["WIRE_VERSION"] must be exactly 1 (frozen protocol version).
5. CONFIG["REPLAY_WINDOW"] must be in $[64, 8192]$.
6. CONFIG["DRONE_HOST"] and CONFIG["GCS_HOST"] must be valid IP addresses.
7. Plaintext hosts must be loopback unless CONFIG["ALLOW_NON_LOOPBACK_PLAINTEXT"] is set.
8. CONFIG["ENCRYPTED_DSCP"], if non-null, must be in $[0, 63]$.
9. CONFIG["CONTROL_COORDINATOR_ROLE"] must be "gcs" or "drone".
10. CONFIG["DRONE_PSK"], if non-empty, must decode to exactly 32 hex bytes. In non-dev environments, it is required.

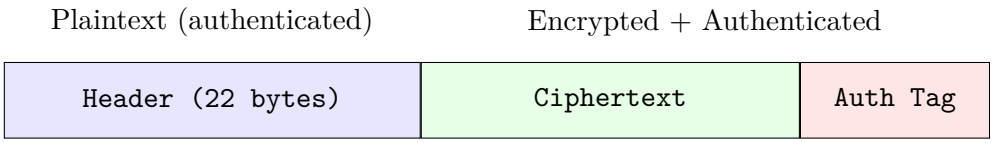
Appendix B

Wire Format Specification

This appendix specifies the exact binary layout of every packet that traverses the encrypted UDP data plane between the drone proxy and the GCS proxy.

B.1 Packet Structure Overview

Every encrypted packet has the following structure:



The header is transmitted in plaintext but is included as *associated data* in the AEAD computation, so any tampering with header fields causes authentication failure at the receiver.

The initialisation vector (IV/nonce) is *not* transmitted on the wire. Instead, it is reconstructed by the receiver from the `epoch` and `seq` fields in the header, saving 12 bytes per packet.

B.2 Header Format

The header is packed using Python's `struct` module with the format string:

```
HEADER_STRUCT = "!BBBBB8sQB"
# Total: 1+1+1+1+1+8+8+1 = 22 bytes
```

The `!` prefix specifies network byte order (big-endian). Each field is detailed in Table B.1.

B.2.1 Field Semantics

version. Fixed at 1 for the current protocol. The receiver rejects any packet whose version does not match. This field enables future protocol evolution without ambiguity.

Table B.1: AEAD header fields (22 bytes total).

Offset	Field	Size	Format	Description
0	`version`	1 B	`B` (uint8)	Wire protocol version; always 1
1	`kem_id`	1 B	`B` (uint8)	KEM algorithm family identifier
2	`kem_param`	1 B	`B` (uint8)	KEM parameter set within family
3	`sig_id`	1 B	`B` (uint8)	Signature algorithm family ID
4	`sig_param`	1 B	`B` (uint8)	Signature parameter set within family
5	`session_id`	8 B	`8s` (bytes)	Random session identifier
13	`seq`	8 B	`Q` (uint64)	Packet sequence number
21	`epoch`	1 B	`B` (uint8)	Key epoch (0–255)

kem_id and kem_param. Together these two bytes identify the KEM algorithm. For example, ML-KEM-768 might map to `kem_id=1, kem_param=2`. The mapping is defined in the `AeadIds` named tuple and the suite registry (Chapter 10).

sig_id and sig_param. Analogous to the KEM identifiers but for the signature algorithm used during the handshake. These allow the receiver to verify that the packet comes from a session established with the expected cryptographic parameters.

session_id. Eight bytes of random data generated during the handshake. Both sides derive the same session ID from the shared secret using HKDF. The receiver checks that the session ID matches; mismatches indicate either a stale packet from a previous session or a spoofing attempt.

seq. A 64-bit unsigned integer that starts at 0 and increments by 1 for each packet sent. This field serves two purposes:

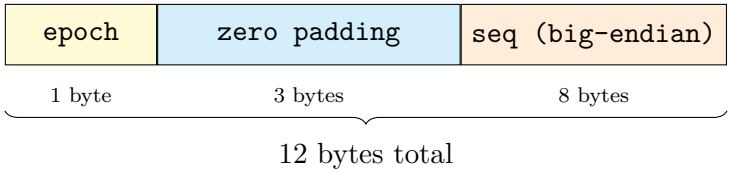
1. **Nonce construction**: Combined with `epoch`, it provides a unique IV for every AEAD operation.

2. **Anti-replay**: The receiver maintains a sliding window and rejects packets with sequence numbers that fall outside the window or have already been seen.

epoch. A single byte (0–255) that increments when the sender calls `bump_epoch`. Each epoch reset sets `seq` back to 0. The combination of `epoch` and `seq` ensures nonce uniqueness even across epoch boundaries. Wrapping from 255 to 0 is *forbidden* with the same key material; a new handshake must be performed.

B.3 Nonce Construction

The AEAD nonce is reconstructed at both sender and receiver using the `_build_nonce` function. For AES-GCM and ChaCha20-Poly1305 the nonce is 12 bytes:



The construction is:

```
def _build_nonce(epoch: int, seq: int, nonce_len: int) ->
    bytes:
     epoch_bytes = epoch.to_bytes(1, "big")
     seq_bytes   = seq.to_bytes(8, "big")
     padding     = b"\x00" * (nonce_len - 1 - 8)
     return epoch_bytes + padding + seq_bytes
```

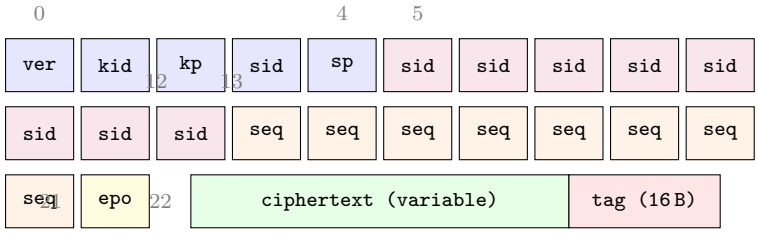
For ASCON-128a, the nonce is 16 bytes (7 bytes of padding instead of 3).

Key Insight

Why not transmit the nonce? Classical TLS and DTLS transmit the nonce (or a portion of it) in every record. By deriving the nonce from header fields that are already transmitted (`epoch` and `seq`), the system saves 12 bytes per packet. On a MAVLink stream at 50 packets/second, this is 600 B/s—small but meaningful on bandwidth-constrained radio links.

B.4 Complete Packet Layout

Combining all components, a full encrypted packet looks like this:



Legend: `ver` = version, `kid` = kem_id, `kp` = kem_param, `sid` = sig_id (byte 3) / session_id (bytes 5–12), `sp` = sig_param, `seq` = sequence number (bytes 13–20), `epo` = epoch.

B.5 Packet Size Analysis

For a typical MAVLink v2 packet of P payload bytes, the total encrypted packet size is:

$$\text{Total} = \underbrace{22}_{\text{header}} + \underbrace{P}_{\text{ciphertext}} + \underbrace{16}_{\text{auth tag}} = P + 38 \text{ bytes}$$

Table B.2: Packet sizes for representative MAVLink messages.

MAVLink Message	Payload	Encrypted	Overhead
HEARTBEAT	9 B	47 B	422%
ATTITUDE	28 B	66 B	136%
GPS_RAW_INT	30 B	68 B	127%
GLOBAL_POSITION_INT	28 B	66 B	136%
MISSION_ITEM_INT	37 B	75 B	103%
Typical (avg. ∼50)	50 B	88 B	76%
MTU-filling	1200 B	1238 B	3.2%

The overhead is substantial for small heartbeat packets but drops to single-digit percentages for larger payloads. In practice, MAVLink heartbeats are sent at 1 Hz, so the absolute bandwidth cost of the overhead is only ∼38 bytes/second.

B.6 Authentication Tag

The authentication tag is produced by the AEAD algorithm:

- **AES-256-GCM**: 16 bytes (128-bit tag)
- **ChaCha20-Poly1305**: 16 bytes (128-bit tag)
- **ASCON-128a**: 16 bytes (128-bit tag)

All three algorithms produce the same tag length, keeping the wire format consistent regardless of AEAD choice.

B.7 Replay Protection on the Wire

The receiver maintains a sliding window of size W = `CONFIG["REPLAY_WINDOW"]` (default 1024). For each incoming packet with sequence number s:

1. If $s > s_{\max}$: accept (new high-water mark); advance window.
2. If $s_{\max} - W < s \leq s_{\max}$: check bitmap; accept if not yet seen, reject if duplicate.
3. If $s \leq s_{\max} - W$: reject (too old).

A packet is only marked as "seen" in the bitmap *after* successful AEAD authentication. This prevents an attacker from burning sequence numbers by sending forged packets.

B.8 Session ID Derivation

The 8-byte session ID is derived during the handshake using HKDF:

```
session_id = HKDF(
    algorithm = SHA-256,
    length    = 8,
    salt      = None,
    info      = b"pqc-session-id",
    ikm       = shared_secret
)
```

Both sides derive the same session ID from the same shared secret, providing a cryptographic binding between the handshake and the data plane.

Appendix C

Metrics Schema Reference

This appendix lists every field in the 18-category metrics schema defined in `core/metrics_schema.py`. Each suite benchmark produces one `ComprehensiveSuiteMetrics` record containing all categories. Fields marked `Optional` may be `None` if the corresponding collector was unavailable or the measurement was not applicable.

C.1 Category A: Run Context

Table C.1: Category A — Run context and environment (20 fields).

Field	Type	Description
run_id	str	Unique benchmark run identifier
suite_id	str	Suite identifier string
suite_index	int	Position in suite ordering
git_commit_hash	str?	Git commit of the codebase
git_dirty_flag	bool?	Uncommitted changes present
gcs_hostname	str?	GCS machine hostname
drone_hostname	str?	Drone machine hostname
gcs_ip	str?	GCS IP address
drone_ip	str?	Drone IP address
python_env_gcs	str?	GCS Python version/path
python_env_drone	str?	Drone Python version/path
liboqs_version	str?	liboqs library version
kernel_version_gcs	str?	GCS kernel version
kernel_version_drone	str?	Drone kernel version
clock_offset_ms	float?	NTP-lite clock offset (ms)
clock_offset_method	str?	Sync method used
run_start_time_wall	str	Wall-clock start (ISO 8601)
run_end_time_wall	str	Wall-clock end (ISO 8601)
run_start_time_mono	float	Monotonic start (seconds)
run_end_time_mono	float	Monotonic end (seconds)

C.2 Category B: Suite Crypto Identity

Table C.2: Category B — Suite cryptographic identity (8 fields).

Field	Type	Description
kem_algorithm	str?	KEM algorithm name (e.g. ML-KEM-768)
kem_family	str?	KEM family (e.g. lattice)
kem_nist_level	str?	NIST security level
sig_algorithm	str?	Signature algorithm name
sig_family	str?	Signature family
sig_nist_level	str?	Signature NIST level
aead_algorithm	str?	AEAD cipher token
suite_security_level	str?	Overall suite security level

C.3 Category C: Suite Lifecycle Timeline

Table C.3: Category C — Suite lifecycle timeline (5 fields).

Field	Type	Description
suite_selected_time	float	When the suite was selected (mono)
suite_activated_time	float	When the handshake completed
suite_deactivated_time	float	When the suite was stopped
suite_total_duration_ms	float	Total wall time (ms)
suite_active_duration_ms	float	Active encryption time (ms)

C.4 Category D: Handshake Metrics

Table C.4: Category D — Handshake timing and status (7 fields).

Field	Type	Description
handshake_start_time_drone	float?	Handshake start (mono)
handshake_end_time_drone	float?	Handshake end (mono)
handshake_total_duration_ms	float?	End-to-end handshake time
protocol_handshake_duration_ms	float?	PQC protocol portion only
end_to_end_handshake_duration_ms	float?	Including setup overhead
handshake_success	bool?	True if successful
handshake_failure_reason	str?	Failure reason (if any)

C.5 Category E: Crypto Primitive Breakdown

Table C.5: Category E — Cryptographic primitive timing (16 fields).

Field	Type	Description
kem_keygen_time_ms	float?	KEM key generation (ms)
kem_encapsulation_time_ms	float?	KEM encapsulation (ms)
kem_decapsulation_time_ms	float?	KEM decapsulation (ms)
signature_sign_time_ms	float?	Signature generation (ms)
signature_verify_time_ms	float?	Signature verification (ms)
total_crypto_time_ms	float?	Sum of all primitives
kem_keygen_ns	int?	KEM keygen (nanoseconds)
kem_encaps_ns	int?	KEM encapsulation (ns)
kem_decaps_ns	int?	KEM decapsulation (ns)
sig_sign_ns	int?	Signature sign (ns)
sig_verify_ns	int?	Signature verify (ns)
pub_key_size_bytes	int?	Public key size
ciphertext_size_bytes	int?	KEM ciphertext size
sig_size_bytes	int?	Signature size
shared_secret_size_bytes	int?	Shared secret size

C.6 Category F: Rekey Metrics

Table C.6: Category F — Rekey operation metrics (7 fields).

Field	Type	Description
rekey_attempts	int?	Number of attempts
rekey_success	int?	Successful rekeys
rekey_failure	int?	Failed rekeys
rekey_interval_ms	float?	Time between rekeys
rekey_duration_ms	float?	Rekey operation time
rekey_blackout_duration_ms	float?	Traffic pause during rekey
rekey_trigger_reason	str?	What triggered the rekey

C.7 Category G: Data Plane

Table C.7: Category G — Data plane (proxy-level) metrics (21 fields).

Field	Type	Description
achieved_throughput_mbps	float?	Measured throughput
goodput_mbps	float?	Application goodput
wire_rate_mbps	float?	Wire-level rate

packets_sent	int?	Total packets sent
packets_received	int?	Total packets received
packets_dropped	int?	Total packets dropped
packet_loss_ratio	float?	Loss ratio (0–1)
packet_delivery_ratio	float?	Delivery ratio (0–1)
replay_drop_count	int?	Replay-rejected packets
decode_failure_count	int?	Decoding failures
ptx_in	int?	Plaintext packets in
ptx_out	int?	Plaintext packets out
enc_in	int?	Encrypted packets in
enc_out	int?	Encrypted packets out
drop_replay	int?	Replay drops (detailed)
drop_auth	int?	Auth failure drops
drop_header	int?	Header mismatch drops
bytes_sent	int?	Total bytes sent
bytes_received	int?	Total bytes received
aead_encrypt_avg_ns	float?	Mean encrypt time (ns)
aead_decrypt_avg_ns	float?	Mean decrypt time (ns)
aead_encrypt_count	int?	Encrypt operations
aead_decrypt_count	int?	Decrypt operations

C.8 Category H: Latency and Jitter

Table C.8: Category H — Latency and jitter metrics (12 fields).

Field	Type	Description
one_way_latency_avg_ms	float?	Mean one-way latency
one_way_latency_p95_ms	float?	95th percentile one-way
jitter_avg_ms	float?	Mean jitter
jitter_p95_ms	float?	95th percentile jitter
latency_sample_count	int?	Number of latency samples
latency_invalid_reason	str?	Reason if invalid
one_way_latency_valid	bool?	Validity flag
rtt_avg_ms	float?	Mean round-trip time
rtt_p95_ms	float?	95th percentile RTT
rtt_sample_count	int?	Number of RTT samples
rtt_invalid_reason	str?	Reason if invalid
rtt_valid	bool?	RTT validity flag

C.9 Category I: MAVProxy Drone

Table C.9: Category I — MAVProxy drone-side metrics (15 fields).

Field	Type	Description
mavproxy_drone_start_time	float?	MAVProxy start
mavproxy_drone_end_time	float?	MAVProxy end
mavproxy_drone_tx_pps	float?	TX packets/sec
mavproxy_drone_rx_pps	float?	RX packets/sec
mavproxy_drone_total_msgs_sent	int?	Total messages sent
mavproxy_drone_total_msgs_received	int?	Total messages received
mavproxy_drone_msg_type_counts	dict?	Per-type message counts
mavproxy_drone_heartbeat_interval_mean	float?	Mean HB interval
mavproxy_drone_heartbeat_loss_count	int?	Lost heartbeats
mavproxy_drone_seq_gap_count	int?	Sequence gaps detected
mavproxy_drone_cmd_sent_count	int?	Commands sent
mavproxy_drone_cmd_ack_received_count	int?	ACKs received
mavproxy_drone_cmd_ack_latency_avg_ms	float?	Mean ACK latency
mavproxy_drone_cmd_ack_latency_p95_ms	float?	P95 ACK latency
mavproxy_drone_stream_rate_hz	float?	Telemetry stream rate

C.10 Category J: MAVProxy GCS (Pruned)

Table C.10: Category J — MAVProxy GCS-side metrics (2 fields, pruned).

Field	Type	Description
mavproxy_gcs_total_msgs_received	int?	Cross-side correlation
mavproxy_gcs_seq_gap_count	int?	MAVLink integrity

Design Decision

Why was GCS MAVProxy pruned? A policy realignment on 2026-01-18 reduced GCS MAVProxy metrics to validation-only fields. GCS-side deep introspection (heartbeat statistics, stream rates, command latencies) was removed because the GCS is a non-constrained observer—its metrics do not influence suite selection policy. Only the two fields needed for cross-side integrity validation were retained.

C.11 Category K: MAVLink Integrity

Table C.11: Category K — MAVLink semantic integrity (9 fields).

Field	Type	Description
mavlink_sysid	int?	System ID
mavlink_compid	int?	Component ID
mavlink_protocol_version	str?	Protocol version
mavlink_packet_crc_error_count	int?	CRC errors
mavlink_decode_error_count	int?	Decode errors
mavlink_msg_drop_count	int?	Dropped messages
mavlink_out_of_order_count	int?	Out-of-order packets
mavlink_duplicate_count	int?	Duplicate packets
mavlink_message_latency_avg_ms	float?	Mean message latency

C.12 Category L: Flight Controller Telemetry

Table C.12: Category L — Flight controller telemetry (10 fields).

Field	Type	Description
fc_mode	str?	Flight mode
fc_armed_state	bool?	Armed/disarmed
fc_heartbeat_age_ms	float?	Last heartbeat age
fc_attitude_update_rate_hz	float?	Attitude update rate
fc_position_update_rate_hz	float?	Position update rate
fc_battery_voltage_v	float?	Battery voltage
fc_battery_current_a	float?	Battery current
fc_battery_remaining_percent	float?	Battery remaining
fc_cpu_load_percent	float?	FC CPU utilisation
fc_sensor_health_flags	int?	Sensor health bitmask

C.13 Category M: Control Plane

Table C.13: Category M — Scheduler control plane (7 fields).

Field	Type	Description
scheduler_tick_interval_ms	float?	Scheduler poll interval
scheduler_action_type	str?	Action taken (HOLD/NEXT)
scheduler_action_reason	str?	Reason for action
policy_name	str?	Active policy name
policy_state	str?	Policy state
policy_suite_index	int?	Current suite index
policy_total_suites	int?	Total suites in list

C.14 Category N: System Resources (Drone)

Table C.14: Category N — Drone system resources (12 fields).

Field	Type	Description
cpu_usage_avg_percent	float?	Mean CPU usage
cpu_usage_peak_percent	float?	Peak CPU usage
cpu_freq_mhz	float?	CPU frequency
memory_rss_mb	float?	Resident memory
memory_vms_mb	float?	Virtual memory
thread_count	int?	Thread count
temperature_c	float?	CPU temperature
uptime_s	float?	System uptime
load_avg_1m	float?	1-minute load average
load_avg_5m	float?	5-minute load average
load_avg_15m	float?	15-minute load average

C.15 Category O: System Resources (GCS) — Deprecated

Category O retains the same field structure as Category N but is **deprecated**. Fields remain at default `None` values. GCS system resource collection was removed because the GCS is a non-constrained system whose resource usage does not influence scheduling policy.

C.16 Category P: Power and Energy

Table C.15: Category P — Power and energy measurements (8 fields).

Field	Type	Description
power_sensor_type	str?	Backend: ina219/rpi5_hwmon/none
power_sampling_rate_hz	float?	Actual sampling rate
voltage_avg_v	float?	Mean voltage
current_avg_a	float?	Mean current
power_avg_w	float?	Mean power
power_peak_w	float?	Peak power
energy_total_j	float?	Total energy (joules)
energy_per_handshake_j	float?	Energy per handshake

C.17 Category Q: Observability

Table C.16: Category Q — Observability and logging (5 fields).

Field	Type	Description
log_sample_count	int?	Number of log samples
metrics_sampling_rate_hz	float?	Metrics sampling rate
collection_start_time	float?	Collection start (mono)
collection_end_time	float?	Collection end (mono)
collection_duration_ms	float?	Collection duration

C.18 Category R: Validation

Table C.17: Category R — Validation and integrity (6 fields).

Field	Type	Description
expected_samples	int?	Expected sample count
collected_samples	int?	Actually collected
lost_samples	int?	Missing samples
success_rate_percent	float?	Collection success rate
benchmark_pass_fail	str?	PASS/FAIL/PARTIAL verdict
metric_status	dict	Per-category status map

The `metric_status` dictionary maps category names to status objects containing a reason string if the category's data is incomplete. This enables forensic analysis of partial benchmark runs.

C.19 Composite Record

The `ComprehensiveSuiteMetrics` dataclass aggregates all 18 categories into a single Python object. It provides:

- `to_dict()` — recursive conversion to a plain Python dictionary.
- `to_json(indent=2)` — JSON serialisation with a default handler for non-serialisable types.
- `save_json(filepath)` — write directly to file.
- `from_dict(data)` — reconstruct from dictionary.
- `from_json(json_str)` — reconstruct from JSON.
- `load_json(filepath)` — load from file.

The helper function `count_metrics()` returns a dictionary mapping category names to their field counts, useful for validating schema completeness.

Bibliography

[1] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, 1983. DOI: 10.1109/TIT.1983.1056650

[2] L. K. Grover, "A Fast Quantum Mechanical Algorithm for Database Search," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, 1996, pp. 212–219.

[3] MAVLink Development Team, *MAVLink Protocol Specification v2.0*, https://mavlink.io/en/, 2024.

[4] Open Quantum Safe Project, *liboqs: An Open-Source C Library for Quantum-Safe Cryptographic Algorithms*, https://openquantumsafe.org/, 2024.

[5] P. W. Shor, "Algorithms for Quantum Computation: Discrete Logarithms and Factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, IEEE, 1994, pp. 124–134.

Colophon

This book was typeset using LaTeX with the `book` document class on A4 paper. Body text is set in Latin Modern Roman at 12 pt. Code listings use Latin Modern Mono via the `listings` package. Diagrams were drawn with TikZ. Charts and data visualisations in the dashboard chapters were produced with Recharts (React) and reproduced here as referenced screenshots.

The bibliography was managed with BibLaTeX and the Biber backend, using the IEEE citation style.

The source code of both the book and the system it documents resides in a single Git repository, ensuring that the documentation always corresponds to an identifiable version of the software.

First edition, 2025.