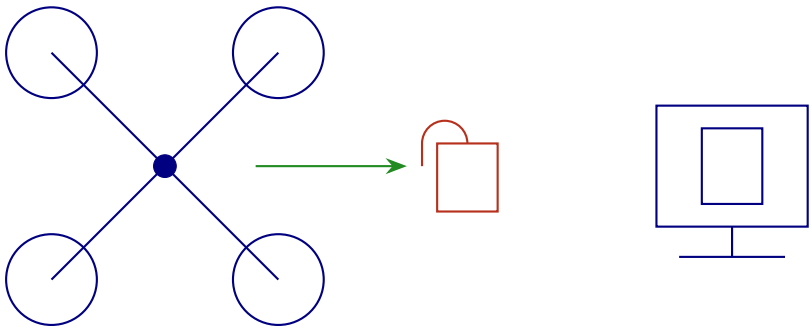
Post-Quantum Secure MAVLink Tunnel

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.

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 **mavlink-spec**, 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 **shor1994**, 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.

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) **liboqs**, 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. By the time you finish reading, you will understand:

- How computer networks work (Chapter **??**).
- What cryptography is and how it provides confidentiality, integrity, and authentication (Chapter **??**).
- Why quantum computers threaten current cryptography and how post-quantum algorithms work (Chapter **??**).
- How drones communicate using MAVLink and MAVProxy (Chapter **??**).
- The complete architecture of the secure tunnel system (Chapter **??**).
- How the handshake protocol establishes post-quantum keys (Chapter **??**).
- How AEAD framing protects individual packets (Chapter **??**).
- How the proxy engine moves packets between plaintext and encrypted worlds (Chapter **??**).
- How 70+ cipher suites are defined, registered, and selected (Chapter **??**).
- How the benchmark scheduler orchestrates test runs (Chapter **??**).
- How metrics are collected, aggregated, and validated (Chapter **??**).
- How the dashboard visualizes results (Chapter **??**).
- The engineering trade-offs, limitations, and lessons learned (Chapter **??**).

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 **??**) and refer back to the Glossary and foundational chapters as needed.

If you are a researcher interested specifically in the post-quantum performance measurements, you may focus on Chapters **??** and **??**, but we strongly recommend at least skimming Chapter **??** for an honest assessment of limitations.

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.

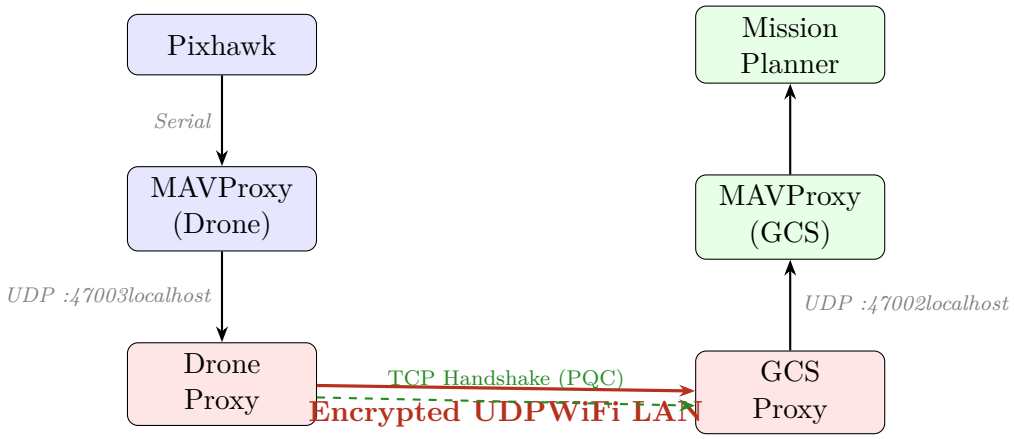


Figure 2.1: Data flow through the PQC tunnel system

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 **grover1996** 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 **??**).

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 **??**.

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 **??** 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 **??**–**??**) 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.

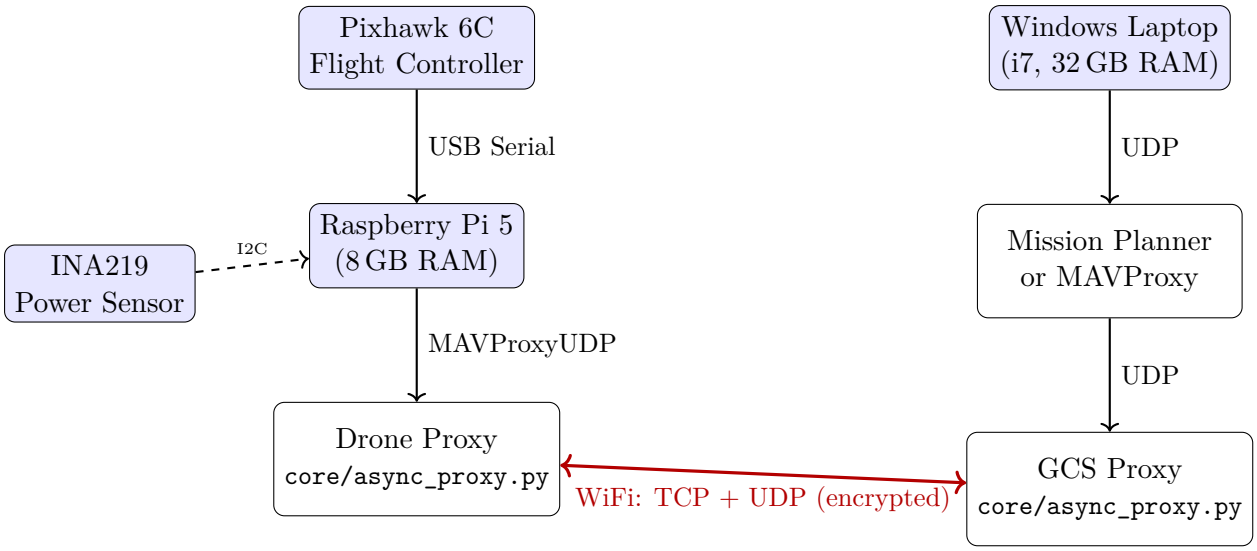


Figure 6.1: Physical hardware topology of the secure tunnel system.

- **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.

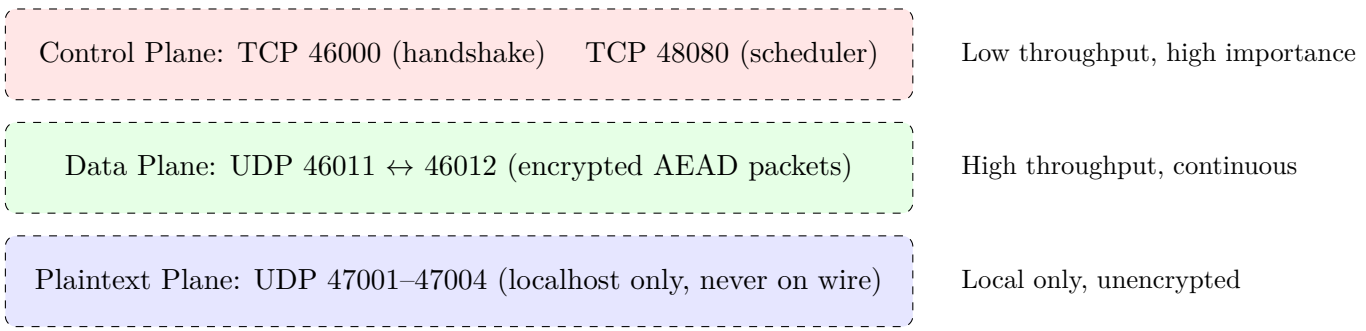


Figure 6.2: The three communication planes of the secure tunnel.

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 **??**.

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 **??** 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 **??** 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`.

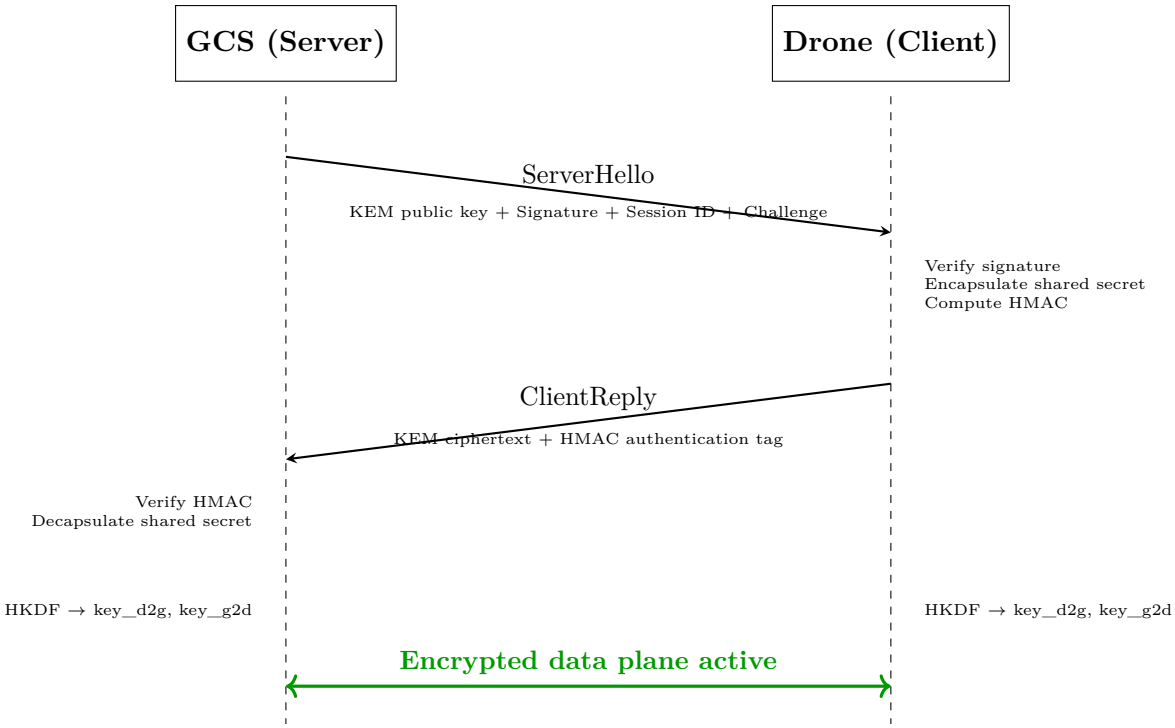


Figure 7.1: The two-message PQC handshake protocol.

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 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 (**??**).
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 ?? through ?? 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 ?? shows the high-level interaction.

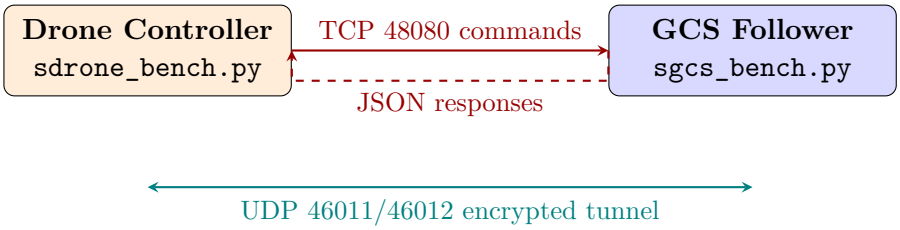


Figure 11.1: Controller–follower architecture. The drone sends commands over a plaintext TCP control channel; the encrypted tunnel carries MAVLink data.

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 ?? 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 **??** 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:

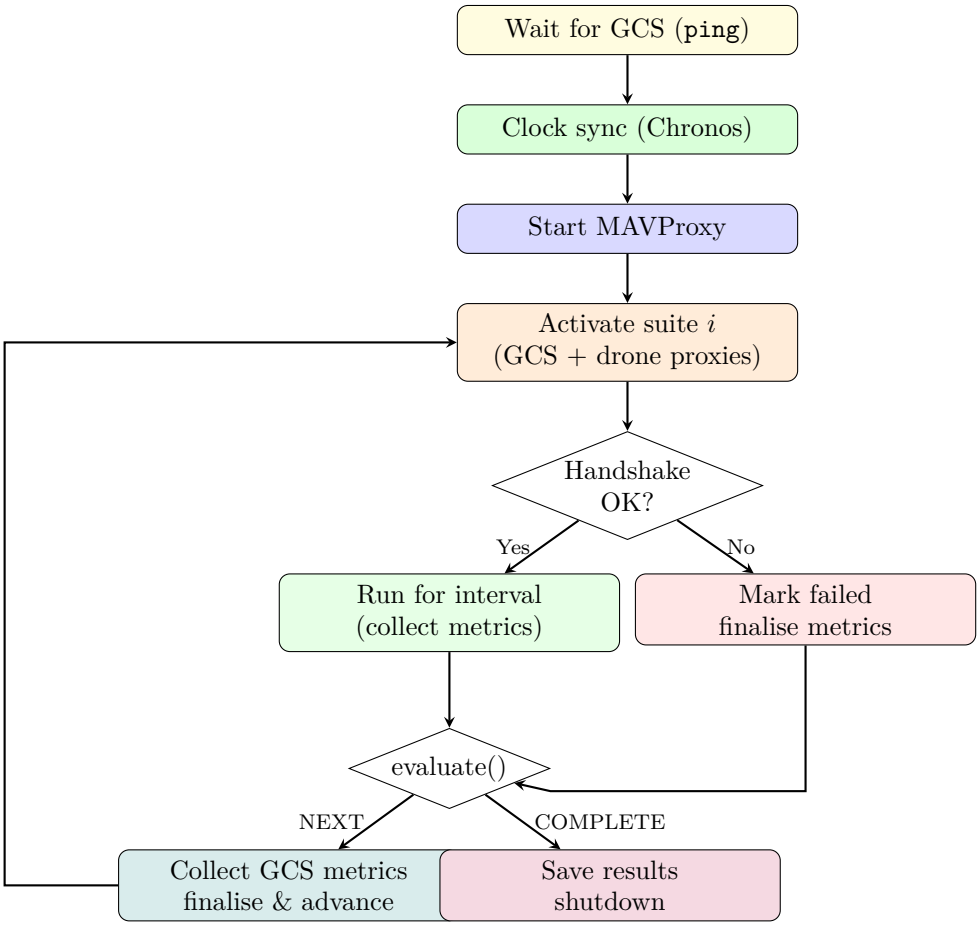


Figure 11.2: Lifecycle of the benchmark main loop. Each suite goes through activation, handshake, metric collection, and policy evaluation.

- 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 **??**).
   (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 **??** 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.

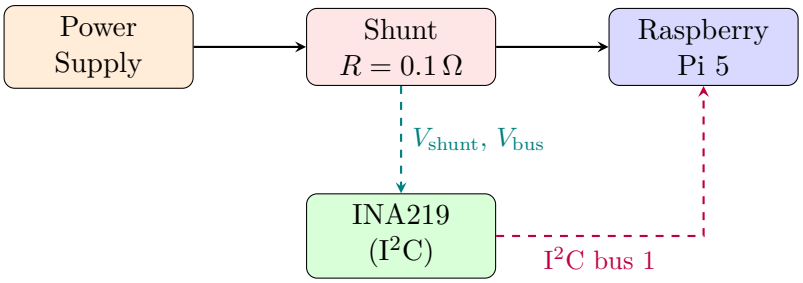


Figure 12.1: INA219 measurement topology. The sensor sits in-line between the power supply and the Pi, measuring the voltage drop across the shunt resistor.

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.

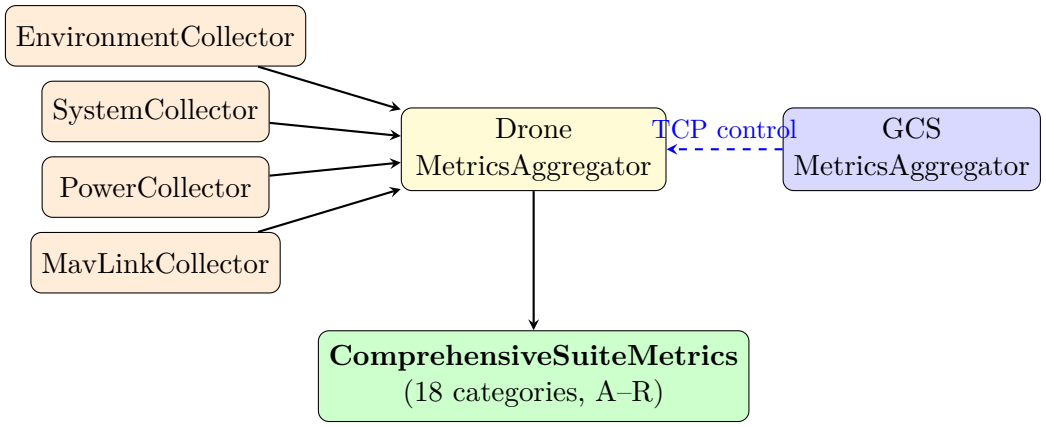


Figure 12.2: Metric flow: four drone-side collectors and the GCS aggregator merge into a single ComprehensiveSuiteMetrics object.

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 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 `ComprehensiveSuiteMet`

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.

12.7 Persistence Formats

The system produces several file formats:

Table 12.4: 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

12.8 Chapter Summary

- The metrics pipeline follows a three-stage architecture: **collectors** → **aggregator** → **persistence**.
- An 18-category schema (A–R) with ~ 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.
- 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 **??**) and metrics pipeline (Chapter **??**) 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 **??** 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

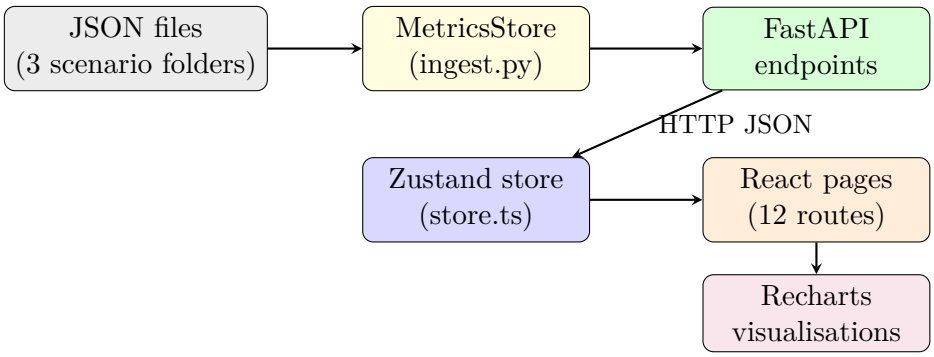


Figure 13.1: Data flow from benchmark output files through the backend API, Zustand state management, React pages, and into Recharts visualisations.

13.6 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.

13.7 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 ms	$\sim$ 50 ms	$\sim$ 2,000 ms
L3	$\sim$ 25 ms	$\sim$ 100 ms	$\sim$ 15,000 ms
L5	$\sim$ 40 ms	$\sim$ 200 ms	$\sim$ 45,000 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 **??**) 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 **??**) 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 Threat Model

The system assumes:

1. **Network attacker**: Can observe, inject, modify, replay, and drop packets on the LAN between drone and GCS.

2. **No physical access**: The attacker cannot tamper with the Raspberry Pi or the GCS hardware.
3. **Pre-shared identity keys**: Signing key pairs are generated offline and deployed to both machines before the benchmark. The private keys are stored on the filesystem (no HSM).
4. **Quantum adversary (future)**: The attacker may record ciphertext today and decrypt it later with a quantum computer ("harvest now, decrypt later").

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 **??**) 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 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, several findings emerge:

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 **??**) 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 **??**) 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 **??**) 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.

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.

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	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.

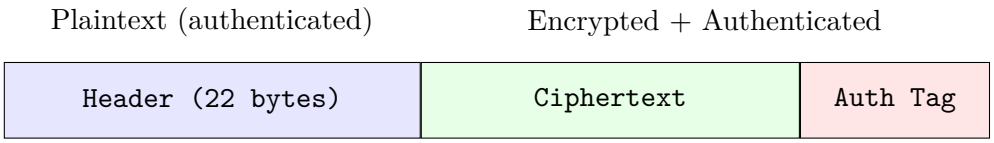
Appendix B

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.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 ??).

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:

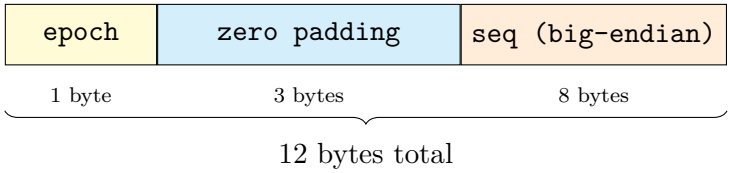
1. **Nonce construction**: Combined with `epoch`, it provides a unique IV for every AEAD operation.

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
```

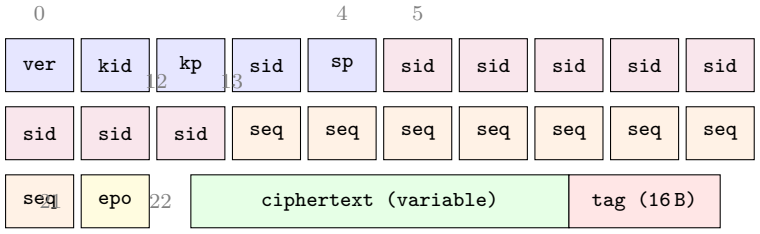
For ASCON-128a, the nonce is 16 bytes (7 bytes of padding instead of 3).

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.

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.