Secure Tunnel Thesis

Generated from codebase

January 29, 2026

Contents

Part I

System Overview

0.1 System Overview and Architectural Philosophy

0.1.1 Terminology and Definitions (Code-Grounded)

This thesis is self-contained and defines each term before use. All terms are grounded in code or configuration. Where a term appears only in configuration or logging, that is explicitly stated.

Ground Control Station (GCS). The Ground Control Station is the control-side host that accepts inbound handshake connections and runs the GCS benchmark server. In code, the GCS role is the server side of the handshake and binds the encrypted UDP receive port (core/async_proxy.py: _perform_handshake; _setup_sockets). GCS host addressing is defined in configuration (core/config.py: CONFIG). Its role affects benchmarking because GCS controls suite activation and returns validation metrics to the drone (sscheduler/sgcs_bench.py: GcsBenchmarkServer._handle_command).

Drone system. The drone system is the client-side host that initiates the handshake and runs the drone benchmark scheduler. In code, the drone role is the client side of the handshake and binds the drone encrypted UDP receive port (core/async_proxy.py: _perform_handshake; _setup_sockets). Drone host addressing is defined in configuration (core/config.py: CONFIG). It affects benchmarking because it orchestrates suite switching and consolidates metrics (sscheduler/sdrone_bench.py: BenchmarkScheduler.run, _finalize_metrics).

Secure tunnel. The secure tunnel is the composed transport formed by (1) a TCP handshake that establishes keys and (2) a UDP data plane that carries AEAD-framed packets. This is not a conceptual abstraction; it is the runtime of `run_proxy` bridging plaintext and encrypted sockets (core/async_proxy.py: run_proxy, _setup_sockets).

Proxy / async proxy. The proxy is the transport process started by `core.run_proxy` that performs the handshake, creates AEAD sender/receiver objects, and forwards UDP packets between plaintext and encrypted planes. The file name `async_proxy.py` is a historical name; the implementation uses selectors, not asyncio (core/async_proxy.py: module docstring).

Control plane. The control plane is the set of messages that drive rekey and scheduling decisions. In this system it exists in two forms: an in-band control plane over encrypted packets (core/policy_engine.py: handle_control) and an optional TCP JSON control plane for external schedulers (core/control_tcp.py: ControlTcpServer._handle_message). It impacts benchmarking because suite transitions and rekeys are encoded in control-plane metadata (core/metrics_aggregator.py: record_control_plane_metrics).

Data plane. The data plane is the encrypted UDP path that carries application data after the handshake. It is implemented as AEAD-framed UDP packets with explicit counters and replay tracking (core/aead.py: Sender.encrypt, Receiver.decrypt; core/async_proxy.py: _setup_sockets). Data-plane counters are the basis for throughput and loss metrics (core/metrics_aggregator.py: record_data_plane_metrics, finalize_suite).

Crypto plane. The crypto plane refers to the handshake and AEAD operations that derive keys and protect data. It is implemented in `core/handshake.py` (KEM + signature + HKDF) and `core/aead.py` (AEAD framing). Its cost is recorded as handshake and primitive metrics (core/metrics_aggregator.py: record_crypto_primitives).

Measurement plane. The measurement plane is the instrumentation path that collects environment, system, power, MAVLink, and proxy counters into a unified schema (core/metrics_schema.py; core/metrics_aggregator.py; core/metrics_collectors.py; core/mavlink_collector.py). It impacts reproducibility because it records git commit, environment metadata, and explicit `metric_status` flags when data is missing (core/metrics_collectors.py: EnvironmentCollector.collect; core/metrics_aggregator.py: _mark_metric_status).

Handshake. The handshake is the authenticated key establishment between GCS and drone, implemented as a TCP exchange that constructs and verifies a signed server hello, encapsulates/decapsulates a shared secret, and derives keys via HKDF (core/handshake.py: build_server_hello, parse_and_verify_server_hello, client_drone_handshake, server_gcs_handshake, derive_transport_keys). It directly impacts benchmark timing because its duration is recorded and used as a primary metric (core/metrics_aggregator.py: record_handshake_start, record_handshake_end).

Rekey. Rekey is the process of negotiating a new suite and keys after the proxy is running. It is implemented in the in-band control plane and triggers a fresh handshake inside `run_proxy` (core/policy_engine.py: request_prepare, handle_control; core/async_proxy.py: rekey worker logic in run_proxy). Rekey affects benchmarking because it can introduce blackout intervals and contributes to suite lifecycle metrics (core/async_proxy.py: counters.rekey_blackout_duration_ms; core/metrics_aggregator.py: record_data_plane_metrics).

Blackout. Blackout refers to the time during which no encrypted packets are forwarded during rekey. The proxy explicitly tracks blackout durations and rekey state (core/async_proxy.py: counters.rekey_blackout_duration_ms and rekey tracking). This impacts throughput and latency metrics because data-plane delivery is paused during blackout windows.

MAVLink. MAVLink is the UAV telemetry protocol observed by the system for integrity and latency measurements. The system does not implement MAVLink; it observes MAVLink traffic via UDP sniffing using pymavlink (core/mavlink_collector.py: MavLinkMetricsCollector.start_sniffing). MAVLink metrics influence latency/jitter and integrity reporting (core/metrics_aggregator.py: finalize_suite).

MAVProxy. MAVProxy is the MAVLink relay used to feed data into the proxy and duplicate telemetry for measurement. The drone scheduler starts MAVProxy with dual outputs, and the GCS scheduler starts MAVProxy to duplicate telemetry to sniff and QGC ports (sscheduler/sdrone_bench.py: start_mavproxy; sscheduler/sgcs_bench.py: GcsMavProxyManager.start).

pymavlink. pymavlink is the Python MAVLink library used by the collector to parse and interpret MAVLink messages (core/mavlink_collector.py: import mavutil). It affects metrics availability because MAVLink metrics are collected only if pymavlink is available.

SYSTEM_TIME. The term SYSTEM_TIME appears in the MAVLink context; the code references time alignment and latency extraction from timestamped messages in the collector logic (core/mavlink_collector.py: latency tracking fields). If SYSTEM_TIME messages are not present or timestamps cannot be aligned, latency validity flags are set as invalid (core/metrics_aggregator.py: finalize_suite).

One-way latency. One-way latency is derived from MAVLink timestamped messages and stored in the latency/jitter schema (core/metrics_schema.py: LatencyJitterMetrics). Its validity is determined by collector output and can be marked invalid when sample prerequisites are missing (core/metrics_aggregator.py: finalize_suite).

Round-trip time (RTT). RTT is computed from command/ack exchanges in MAVLink telemetry and recorded in the latency/jitter schema (core/metrics_schema.py: LatencyJitterMetrics; core/mavlink_collector.py: command tracking). It affects benchmark interpretation of link responsiveness.

Packet loss. Packet loss is derived from proxy counters (drops vs sent) and recorded in data-plane metrics (core/metrics_aggregator.py: record_data_plane_metrics). It affects throughput and goodput reporting.

Goodput. Goodput is computed from plaintext byte counters and suite active duration, representing effective payload throughput (core/metrics_aggregator.py: finalize_suite). It differs from wire rate, which uses encrypted byte counters.

Throughput. Throughput (achieved and wire rate) is computed from proxy counters and suite duration (core/metrics_aggregator.py: finalize_suite). It reflects encrypted transport capacity, not necessarily application payload delivery.

Chronos clock sync. Chronos clock synchronization is a scheduler-driven RPC that estimates clock offset between GCS and drone to align timestamps (sscheduler/sdrone_bench.py: BenchmarkScheduler.run; sscheduler/sgcs_bench.py: GcsBenchmarkServer._handle_command). This affects timing metrics that rely on synchronized clocks.

Suite. A suite is a specific combination of KEM, AEAD, and signature algorithms defined in the suite registry. Suites are enumerated and sorted by NIST level for benchmarking (core/suites.py: registry definitions; sscheduler/benchmark_policy.py: BenchmarkPolicy._build_suite_list).

Benchmark suite. A benchmark suite is a suite instance being executed under the benchmark scheduler with a fixed duration and metrics collection. It is not an abstract term; it corresponds to a specific activation interval in the scheduler run loop (sscheduler/sdrone_bench.py: BenchmarkScheduler._run_loop).

Scheduler. The scheduler is the control process that decides when to activate or switch suites. In this system, the benchmark scheduler is drone-driven and uses a deterministic policy (sscheduler/sdrone_bench.py: BenchmarkScheduler; sscheduler/benchmark_policy.py: BenchmarkPolicy).

MetricsAggregator. MetricsAggregator is the component that collects and merges metrics into the comprehensive schema and computes derived values such as throughput and validity flags (core/metrics_aggregator.py: MetricsAggregator).

Comprehensive JSON. Comprehensive JSON refers to the serialized form of the comprehensive metrics schema. It is produced by the aggregator and stored as JSON files in the logs directory (core/metrics_aggregator.py: finalize_suite; dashboard/backend/ingest.py: _parse_comprehensive_filename).

JSONL logs. JSONL logs are line-delimited JSON records written by schedulers and the GCS server for per-suite events (sscheduler/sdrone_bench.py: _log_result; sscheduler/sgcs_bench.py: stop_suite handler). The ingest pipeline uses them as fallback data when comprehensive JSON is missing (dashboard/backend/ingest.py: _build_minimal_suite).

Ingest pipeline. The ingest pipeline loads comprehensive JSON and JSONL fallback data into an in-memory store for the dashboard API (dashboard/backend/ingest.py: MetricsStore, _load_json, _build_minimal_suite). It affects reproducibility by preserving missing-data reasons.

Validation status. Validation status is the set of per-field flags that indicate whether a metric was collected, invalid, or missing, recorded in `metric_status` (core/metrics_aggregator.py: _mark_metric_status, finalize_suite; dashboard/backend/ingest.py: _build_minimal_suite).

Metric validity flags. Metric validity flags are stored in the validation schema and indicate missing or invalid measurements (core/metrics_schema.py: ValidationMetrics; core/metrics_aggregator.py: finalize_suite).

Policy engine. The policy engine implements the in-band rekey state machine and is invoked by the proxy when control messages are received (core/policy_engine.py: handle_control). It influences runtime suite changes and rekey behavior.

Suite registry. The suite registry defines algorithm mappings, IDs, and NIST levels for KEM and signature algorithms, enabling consistent suite selection (core/suites.py: registry definitions).

KEM (Key Encapsulation Mechanism). A KEM is the algorithm used to encapsulate a shared secret in the handshake. It is instantiated and used in the handshake code (core/handshake.py: build_server_hello, client_encapsulate, server_decapsulate).

SIG (Digital Signature). The signature algorithm authenticates the server hello. It is used to sign and verify the transcript during handshake (core/handshake.py: build_server_hello, parse_and_verify_server_hell

AEAD (Authenticated Encryption with Associated Data). AEAD is the encryption mode used for the data plane. It is instantiated in the sender/receiver and used to encrypt/decrypt UDP payloads with the header bound as associated data (core/aead.py: Sender.encrypt, Receiver.decrypt).

0.1.2 System Goals and Non-Goals (Derived from Code)

+The system's implemented goals are those exercised by the proxy, scheduler, and metrics components. This section does not infer intent beyond code paths. +

Goal: deterministic, dependency-light transport. The proxy explicitly avoids asyncio and relies on selectors to remain deterministic and dependency-light (core/async_proxy.py: module docstring). This supports reproducible experiments by reducing non-deterministic event-loop behavior. +

Goal: measurable cryptographic and transport costs. The handshake records per-primitive timings and artifact sizes, and the proxy records AEAD timing and drop counters (core/handshake.py: _finalize_handshake_metrics; core/async_proxy.py: ProxyCounters). These are later exposed through the metrics aggregator. +

Goal: explicit separation of planes. Distinct sockets are used for plaintext, encrypted data, and control plane operations (core/async_proxy.py: _setup_sockets; core/control_tcp.py: ControlTcpServer). + +

0.1.3 Architectural Overview

+The system consists of a transport core (proxy), a control/orchestration layer (schedulers), and a measurement layer (collectors and aggregator). The proxy is the only component that handles encryption and replay protection. The schedulers control suite activation and gather results. The measurement layer captures environment, system, power, and MAVLink metrics and merges them into a comprehensive schema. + +

0.1.4 Component Breakdown and File Responsibilities

+

core/config.py. Defines hosts, ports, and feature flags that govern runtime behavior across all components. It encodes strict peer matching, DSCP marking, and control-plane defaults (core/-config.py: CONFIG). +

core/async_proxy.py. Implements the transport proxy, including handshake orchestration, UDP socket setup, AEAD sender/receiver binding, replay enforcement, and status file emission (core/async_proxy.py: run_proxy, _setup_sockets, ProxyCounters). +

core/handshake.py. Implements server and client handshake logic, including signature verification, KEM encapsulation/decapsulation, PSK authentication tag verification, HKDF key derivation, and metrics flattening (core/handshake.py: build_server_hello, parse_and_verify_server_hello, client_drone_handshake, server_gcs_handshake, _finalize_handshake_metrics). +

core/aead.py. Implements AEAD framing with deterministic nonces, header binding, and replay window enforcement (core/aead.py: Sender.encrypt, Receiver.decrypt). +

core/policy_engine.py and core/control_tcp.py. Implement the in-band two-phase commit for rekey and an optional TCP JSON control plane for external schedulers (core/policy_engine.py: handle_control; core/control_tcp.py: ControlTcpServer). +

core/metrics_schema.py, core/metrics_aggregator.py, core/metrics_collectors.py. Define the metrics schema and implement collection, aggregation, and validation flags (core/metrics_schema.py; core/metrics_aggregator.py; core/metrics_collectors.py). +

core/mavlink_collector.py. Implements MAVLink integrity and latency observation using pymavlink and a UDP sniff loop (core/mavlink_collector.py: MavLinkMetricsCollector.start_sniffing). +

sscheduler/sdrone_bench.py. Drone benchmark scheduler that coordinates suite cycling, starts the proxy, reads status, and merges GCS metrics (sscheduler/sdrone_bench.py: BenchmarkScheduler). +

sscheduler/sgcs_bench.py. GCS benchmark server that accepts drone commands, starts the GCS proxy, collects system/MAVLink metrics, and returns suite records (sscheduler/sgcs_bench.py: GcsBenchmarkServer). + +

0.1.5 Control-Flow Narrative (High-Level Execution)

+

Startup. The GCS benchmark server starts and listens for TCP commands (sscheduler/sgcs_bench.py: GcsBenchmarkServer.start). The drone scheduler waits for the GCS `ping` response (sscheduler/sdrone_bench.py: wait_for_gcs). +

Suite activation. The drone sends `start_proxy` to the GCS, then starts its local proxy and waits for handshake confirmation (sscheduler/sdrone_bench.py: _activate_suite; sscheduler/sgcs_bench.py: GcsBenchmarkServer._handle_command). +

Measurement. During suite activation, the metrics aggregator begins background sampling and records control-plane metadata. MAVLink sniffing begins on role-specific ports (core/metrics_aggregator.py: start_suite). +

Suite finalization. The drone requests `stop_suite` from the GCS, reads proxy counters, finalizes metrics, and logs JSONL results (sscheduler/sdrone_bench.py: _collect_gcs_metrics, _finalize_metrics; core/metrics_aggregator.py: finalize_suite). + +

0.1.6 Data-Flow Narrative (Planes and Artifacts)

+

Plaintext to encrypted flow. MAVProxy emits plaintext UDP to the proxy's plaintext input port, which is encrypted and forwarded over the encrypted UDP socket to the peer (core/async_proxy.py: _setup_sockets; core/aead.py: Sender.encrypt). The reverse path decrypts and forwards to the local plaintext egress. +

Measurement flow. System and environment collectors run in the background, power sampling runs on the drone, and MAVLink sniffing runs on a separate port. These feeds are merged into a comprehensive schema (core/metrics_collectors.py: SystemCollector; core/metrics_aggregator.py: finalize_suite; core/mavlink_collector.py: MavLinkMetricsCollector). +

Control-plane flow. Rekey commands flow either in-band over encrypted packets (core/policy_engine.py: handle_control) or out-of-band via TCP JSON control (core/control_tcp.py: ControlTcpServer). + +

0.1.7 Benchmarking Correctness and Research Reproducibility

+

Correctness. Handshake completion is a gate condition; suite metrics are finalized only after explicit handshake confirmation and GCS `stop_suite` results. This ensures that suite records correspond to actual cryptographic activation (sscheduler/sdrone_bench.py: read_handshake_status, _finalize_metrics). +

Reproducibility. The environment collector records git commit hash, dirty state, and liboqs version per suite, providing a reproducibility anchor for each run (core/metrics_collectors.py: EnvironmentCollector.collect). The metrics aggregator records `metric_status` entries for missing values, which the ingest pipeline preserves (core/metrics_aggregator.py: _mark_metric_status; dashboard-/backend/ingest.py: _build_minimal_suite). + +

0.1.8 Not Implemented or Disabled in Current Codebase

+

Traffic generation in GCS benchmark server. The `start_traffic` command is explicitly disabled in the GCS benchmark server. Any autonomous traffic generation from the GCS benchmark server is therefore not implemented (sscheduler/sgcs_bench.py: GcsBenchmarkServer._handle_command). +

Simulation tooling. Simulation is disabled by default via configuration and must be explicitly enabled; no claims of simulation behavior are made beyond the configuration flags (core/config.py: ENABLE_SIMULATION; settings.json: dev_tools.enabled).

Part II

Handshake and Data Plane

0.2 Handshake, Rekey, and Data Plane Mechanics

0.2.1 Entry Point, Role Binding, and Key Material Loading

The proxy entrypoint is a thin CLI wrapper that binds an explicit role and suite, loads signing material for the GCS server, and delegates execution to the transport core. The GCS role requires a private signing key, while the drone role consumes a peer public key when provided (core/run_proxy.py: main). This entrypoint enforces role correctness before any network sockets are opened.

Role validation. The role is strictly limited to `gcs` or `drone` and is validated before any handshake logic runs (core/run_proxy.py: main).

Suite resolution. Suite identifiers are resolved to registry entries before the proxy starts, guaranteeing that the handshake and AEAD layers operate on a consistent suite definition (core/run_proxy.py: main; core/suites.py: get_suite).

0.2.2 Handshake Construction: GCS as Server

The GCS handshake path constructs a server hello that is explicitly bound to the negotiated suite and wire version. The server hello includes the KEM public key and a signature over a transcript that concatenates the protocol version, session identifier, algorithm names, public key bytes, and a random challenge (core/handshake.py: build_server_hello).

Why the transcript exists in code. The transcript construction is explicitly concatenated in the server implementation; this is not a conceptual claim. The transcript is the actual input to `Signature.sign` (core/handshake.py: build_server_hello).

Performance artifacts. KEM key generation and signature timing are recorded into the handshake metrics structure, including wall-clock and perf counter intervals (core/handshake.py: build_server_hello).

0.2.3 Handshake Verification: Drone as Client

The drone explicitly verifies the server hello signature and enforces suite consistency against the expected suite to prevent downgrade. Any mismatch raises a verification error, which aborts the handshake (core/handshake.py: parse_and_verify_server_hello; client_drone_handshake).

Signature verification is mandatory. The drone constructs a signature verifier and fails closed on verification failure (core/handshake.py: parse_and_verify_server_hello). This is enforced in code; there is no bypass path.

Suite downgrade protection. The client compares negotiated KEM and signature identifiers to the expected values in the suite configuration and explicitly raises `HandshakeVerifyError` on mismatch (core/handshake.py: client_drone_handshake).

0.2.4 PSK Authentication Tag

In addition to signature verification, the GCS validates a drone authentication tag computed from a pre-shared key (PSK). The PSK is expected to be injected via environment or configuration, and is required outside dev mode (core/handshake.py: _drone_psk_bytes; server_gcs_handshake). This tag is verified before decapsulation and key derivation, ensuring that only authenticated drones proceed to key agreement.

0.2.5 Key Derivation and Directional Keys

Both roles derive transport keys using HKDF with explicit context binding the session ID and algorithm identifiers. The derived output is split into two directional keys, and the return order is role-specific, ensuring that each side uses the correct send/receive keys (core/handshake.py: derive_transport_keys).

Role-specific mapping. The drone treats the first derived key as send-to-GCS, while the GCS treats the first derived key as send-to-drone; this asymmetry is encoded in the derivation function itself (core/handshake.py: derive_transport_keys).

0.2.6 Handshake Integration into Proxy Startup

The transport core performs the handshake before binding the AEAD sender/receiver and before entering the selector loop. The handshake result provides the session ID, negotiated algorithm identifiers, and peer address for encrypted UDP routing (core/async_proxy.py: run_proxy; _perform_handshake).

Handshake timeouts. The GCS listener enforces rate limiting and timeout logic when accepting the TCP connection, and rejects connections that fail allow-list checks or rate-limiting gates (core/async_proxy.py: _perform_handshake; _TokenBucket.allow).

0.2.7 AEAD Sender and Receiver Construction

Once keys and identifiers are available, the proxy constructs an AEAD sender and receiver that embed the wire version, suite identifiers, session ID, and replay window. The AEAD token is carried in configuration and is required for sender/receiver creation (core/async_proxy.py: _build_sender_receiver; core/aead.py: Sender, Receiver).

Header semantics. The sender packs a wire header containing version, KEM ID, signature ID, session ID, sequence, and epoch (core/aead.py: Sender.pack_header). The receiver validates these fields before attempting decryption (core/aead.py: Receiver.decrypt).

0.2.8 Wire Format and Deterministic Nonce

The AEAD design removes nonce bytes from the wire by deriving nonces locally from epoch and sequence. This reduces per-packet overhead and forces both ends to strictly track state (core/aead.py: Sender.encrypt; Receiver.decrypt; _build_nonce).

Sequence-based safety. The sender enforces a sequence threshold to avoid IV exhaustion and raises a specific exception that signals the need for rekey (core/aead.py: Sender.encrypt). This behavior has direct benchmarking implications for long-running tests.

0.2.9 Replay Protection and Error Taxonomy

The receiver maintains a replay window and rejects duplicate or stale packets. Each failure reason is explicitly classified for accurate drop accounting (core/aead.py: Receiver._check_replay; core/async_proxy.py: _parse_header_fields).

Header pre-validation. Before decryption, the proxy pre-classifies header errors (version mismatch, crypto ID mismatch, session mismatch) to separate authentication failures from structural mismatches (core/async_proxy.py: _parse_header_fields).

0.2.10 In-Band Rekey Control Flow

The in-band control plane implements a two-phase commit protocol for rekeying. The coordinator sends a prepare request, the peer responds with prepare_ok or prepare_fail, and a commit triggers an immediate handshake for the new suite (core/policy_engine.py: request_prepare, handle_control, record_rekey_result).

Rekey execution. Rekeying performs a fresh handshake, rebuilds AEAD IDs, reconstructs sender/receiver objects, updates the active context, and records counters and status messages (core/async_proxy.py: run_proxy, rekey worker logic).

0.2.11 TCP JSON Control Plane (Out-of-Band)

The optional TCP JSON control server exists to bridge external schedulers. It parses newline-delimited JSON, enforces allow-list checks, and forwards rekey requests into the in-band control state (core/control_tcp.py: ControlTcpServer._handle_message). This layer is intentionally small and dependency-free to avoid contaminating the transport core with orchestration complexity.

0.2.12 Control-Plane and Benchmarking Implications

Handshake and rekey events are critical to benchmarking correctness because they define the temporal bounds for suite activation and cryptographic cost. The metrics aggregator records handshake start/end, protocol duration, and primitive timing and can mark missing data explicitly (core/metrics_aggregator.py: record_handshake_start, record_handshake_end, record_crypto_primitives).

Scheduler dependency. The drone scheduler waits for the proxy status file to report `handshake_ok` or `running` before marking a suite as active, preventing premature measurement (sscheduler/sdrone_bench.py: read_handshake_status; _activate_suite).

0.2.13 Not Implemented or Disabled Paths

No additional handshake or rekey paths are implemented beyond the in-band control plane and the optional TCP JSON interface. Any other orchestration behavior must be treated as not implemented in the current codebase.

Part III

Schedulers and Suite Benchmarks

0.3 Benchmark Schedulers and MAVProxy Integration

0.3.1 Benchmark Scheduler Role and Scope

The benchmark scheduler is a drone-driven orchestration component that controls suite activation, proxy lifecycle, and metrics consolidation. It is designed for exhaustive, deterministic suite traversal and explicitly records the orchestration timeline for forensic auditing (sscheduler/sdrone_bench.py: BenchmarkScheduler).

Benchmark policy coupling. The scheduler instantiates a benchmark policy with a cycle interval and optional AEAD filter (sscheduler/sdrone_bench.py: BenchmarkScheduler.__init__). The policy enumerates suites using the registry and sorts them for predictable coverage (sscheduler/benchmark_policy.py: BenchmarkPolicy._build_suite_list).

0.3.2 Control-Flow Entry and Run Loop

The scheduler's run method performs GCS availability checks, clock synchronization, MAVProxy startup, and then enters a suite cycling loop (sscheduler/sdrone_bench.py: BenchmarkScheduler.run, _run_loop).

Clock synchronization. The scheduler calls the GCS `chronos_sync` command and records the offset in the metrics aggregator if available (sscheduler/sdrone_bench.py: BenchmarkScheduler.run). This is the only path that populates `run_context.clock_offset_ms` in the metrics schema.

Suite boundary control. The loop evaluates policy output and triggers suite changes on `NEXT_SUITE`, finishing by collecting GCS metrics and finalizing the suite before proceeding (sscheduler/sdrone_bench.py: BenchmarkScheduler._run_loop).

0.3.3 Suite Activation Control Flow

Suite activation is a multi-step sequence with explicit ordering to satisfy handshake requirements and logging integrity.

Metrics initialization. The scheduler starts the metrics aggregator for the suite and records control-plane metadata before any network activation, ensuring the suite has a measurement record even if activation fails (sscheduler/sdrone_bench.py: BenchmarkScheduler._activate_suite; core/metrics_aggregator.py: start_suite, record_control_plane_metrics).

GCS-first proxy start. The scheduler sends `start_proxy` to the GCS and aborts on rejection, preventing the drone proxy from starting without a ready server (sscheduler/sdrone_bench.py: BenchmarkScheduler._activate_suite; sscheduler/sgcs_bench.py: GcsBenchmarkServer._handle_command).

Drone proxy activation. The drone then starts its local proxy process and waits for handshake completion by reading the status file written by the proxy (sscheduler/sdrone_bench.py: DroneProxyManager.start; read_handshake_status).

0.3.4 Handshake Confirmation and Metrics Extraction

The scheduler treats a `handshake_ok` or `running` status as a successful handshake and extracts handshake metrics from the status file. Those metrics are then forwarded to the benchmark policy and the metrics aggregator (sscheduler/sdrone_bench.py: read_handshake_status; BenchmarkScheduler._activate_suite).

Primitive timing capture. The metrics aggregator records crypto primitives directly from the handshake metrics structure, bridging transport events into the schema (core/metrics_aggregator.py: record_crypto_primitives).

0.3.5 Data-Plane Metrics Collection and Finalization

The scheduler finalizes a suite by reading proxy counters from the status file and passing them to the metrics aggregator for throughput and loss calculations (sscheduler/sdrone_bench.py: BenchmarkScheduler._finalize_metrics; core/metrics_aggregator.py: record_data_plane_metrics, finalize_suite).

GCS metrics merge. The scheduler calls `stop_suite` on the GCS, which returns validation, latency, and system metrics that are merged into the suite record (sscheduler/sdrone_bench.py: BenchmarkScheduler._collect_gcs_metrics; sscheduler/sgcs_bench.py: GcsBenchmarkServer._handle_command).

0.3.6 MAVProxy Integration on the Drone

The drone scheduler launches MAVProxy with two output streams: a plaintext output to the proxy and a duplicate output to a sniff port for MAVLink metrics. This explicitly decouples application traffic from measurement traffic (sscheduler/sdrone_bench.py: start_mavproxy).

Sniff port isolation. The dedicated sniff port avoids collisions with the proxy's plaintext bindings and enables the MAVLink collector to operate without interfering with the transport (sscheduler/sdrone_bench.py: MAVLINK_SNIFF_PORT; core/metrics_aggregator.py: start_suite).

0.3.7 MAVProxy Integration on the GCS

The GCS benchmark server starts MAVProxy with a master bound to the proxy plaintext output and duplicates telemetry to both a sniff port and a local QGC port. These outputs are explicitly described in the GCS MAVProxy manager (sscheduler/sgcs_bench.py: GcsMavProxyManager.start).

0.3.8 GCS Benchmark Server: Command Handling

The GCS benchmark server is a TCP command server that executes a fixed command set for benchmark orchestration.

Command repertoire. The server supports `ping`, `get_info`, `prepare_rekey`, `start_proxy`, `stop_suite`, `shutdown`, and `chronos_sync` (sscheduler/sgcs_bench.py: GcsBenchmarkServer._handle_command). Each command is mapped to explicit runtime actions, not inferred behavior.

Suite lifecycle at the GCS. On `start_proxy`, the server resets MAVLink counters, starts system sampling, records handshake start, and starts the proxy. On `stop_suite`, it stops MAVLink sampling, stops the proxy, and writes a JSONL record (sscheduler/sgcs_bench.py: GcsBenchmarkServer._handle_command).

0.3.9 Traffic Generation Constraint and Its Implications

The GCS benchmark server explicitly disables `start_traffic`, returning a `traffic_generation_disabled` error. As a consequence, any benchmark run that depends solely on this server will have handshake and control-plane metrics but may have zero data-plane throughput unless an external traffic source is injected (sscheduler/sgcs_bench.py: GcsBenchmarkServer._handle_command).

0.3.10 Benchmark Correctness and Reproducibility

Correctness depends on strict sequencing and observable evidence.

Strict sequencing. The scheduler enforces GCS-first activation and handshake completion before recording suite success, aligning recorded metrics with actual cryptographic availability (sscheduler/sdrone_bench.py: BenchmarkScheduler._activate_suite).

Reproducibility hooks. Each suite is recorded with a run ID and suite index, and the metrics schema captures git commit state and environment metadata through the aggregator (core/metrics_aggregator.py: start_suite; core/metrics_collectors.py: EnvironmentCollector.collect).

0.3.11 Not Implemented or Disabled Paths

No in-process traffic generator is active in the benchmark server; the command exists but is explicitly disabled. Any discussion of automated traffic generation within `sgcs_bench.py` beyond this disabled stub is not implemented in the current codebase (sscheduler/sgcs_bench.py: GcsBenchmarkServer._handle_command).

0.4 Suite Benchmark Framework

0.4.1 Scope and Terminology

This part documents the suite benchmark framework implemented under `suite_benchmarks/framework`. The framework is an alternate, application-level harness that orchestrates suite runs by controlling the secure tunnel proxies and collecting end-to-end measurements. The key terms used below are:

- **Suite benchmark framework**: The pair of scripts `suite_bench_gcs.py` and `suite_bench_drone.py` that coordinate suite runs using a TCP control channel and proxy subprocess management.
- **GCS scheduler**: The GCS-side controller in `SuiteBenchmarkRunner` that selects suites, starts proxies, generates traffic, and aggregates results.
- **Drone control server**: The drone-side TCP server in `BenchmarkControlServer` that starts/stops the drone proxy and returns metrics.
- **Suite iteration**: One execution of a suite ID with a fixed duration; repeated `iterations` times per suite.

- **Suite metrics**: The structured metrics container `SuiteMetrics` collected on the drone, and `SuiteBenchResult` collected on the GCS.

1

0.4.2 High-Level Architecture and Control/Data Planes

The framework is split into a GCS-side scheduler and a drone-side control server. The GCS scheduler performs orchestration, while the drone server performs local proxy lifecycle management and collects hardware/system metrics.

- **Control plane**: A TCP command channel from GCS to drone. The GCS uses `DroneController_send_comma` to send JSON commands such as `start_suite`, `stop_suite`, `rekey`, `status`, and `shutdown`. The drone server parses these commands in `BenchmarkControlServer_handle_command()` and dispatches to handler methods such as `_start_suite()` and `_stop_suite()`.
- **Data plane**: The tunnel data plane is created by running `core.run_proxy` in subprocesses on both GCS and drone. The GCS side manages its proxy in `GCSProxyManager.start()` and the drone side manages its proxy in `ProxyManager.start()`.

2

0.4.3 GCS Scheduler: Configuration, Suite Selection, and Orchestration

The GCS scheduler declares a fixed local configuration for suites and runtime parameters, then allows override via CLI flags. The local defaults include `LOCAL_SUITES`, `LOCAL_ITERATIONS`, `LOCAL_DURATION_S`, and `LOCAL_TEST_REKEY`. If no suites are specified, the GCS enumerates suites with generated keys and limits to the first ten for quick runs. [3]

The main orchestration logic is implemented in `SuiteBenchmarkRunner.run()` and `SuiteBenchmarkRunner.ru` Each iteration performs the following steps:

1. Start the drone proxy by issuing `start_suite` to the drone control server and record `handshake_drone_ms`.
2. Start the GCS proxy subprocess via `GCSProxyManager.start()` and compute `handshake_gcs_ms` and `handshake_total_ms`.
3. Wait for tunnel establishment (a fixed sleep).
4. Generate UDP traffic via `TrafficGenerator` for `duration_s` seconds and compute throughput as packets/s and Mbps from sent bytes.
5. Optionally request rekey via `DroneController.rekey()` and record rekey timing fields returned by the drone.
6. Stop the drone suite and fetch aggregated metrics via `DroneController.stop_suite()`.

4

[1]Evidence: suite_benchmarks/framework/suite_bench_gcs.py (module docstring, `SuiteBenchmarkRunner`, `SuiteBenchResult`); suite_benchmarks/framework/suite_bench_drone.py (module docstring, `BenchmarkControlServer`, `SuiteMetrics`).

[2]Evidence: suite_benchmarks/framework/suite_bench_gcs.py (`DroneController`, `GCSProxyManager`); suite_benchmarks/framework/suite_bench_drone.py (`BenchmarkControlServer`, `ProxyManager`).

[3]Evidence: suite_benchmarks/framework/suite_bench_gcs.py (local configuration block, `get_available_suites()`, `main()`).

[4]Evidence: suite_benchmarks/framework/suite_bench_gcs.py (`SuiteBenchmarkRunner.run_suite_iteration()`, `TrafficGenerator`).

0.4.4 Traffic Generator Semantics

The traffic generator is a dedicated thread that sends UDP datagrams at a target rate. Each packet includes a 4-byte sequence number and an 8-byte timestamp; remaining bytes are padding to the configured payload length. Traffic is sent to the GCS plaintext host and port (defaults from `core.config.CONFIG`). The generator does not read responses; its role is strictly transmit-side load generation for throughput calculations. [5]

0.4.5 Drone Control Server: Proxy Lifecycle and Metrics

The drone-side control server is a long-running TCP server created by `BenchmarkControlServer.start()`. For each command, it decodes JSON, dispatches to handlers, and returns JSON responses. Key handler behaviors are:

- `_start_suite()`: Initializes a `SuiteMetrics` record, resets latency tracking, starts power collection, timestamps the proxy launch as a handshake window, and starts the drone proxy with `ProxyManager.start()`.
- `_stop_suite()`: Stops power collection and the proxy, aggregates power and latency statistics into `SuiteMetrics`, captures CPU/memory/temperature, and writes a per-iteration JSON record under `suite_benchmarks/raw_data/drone`.
- `_handle_rekey()`: Records rekey timing fields and (currently) simulates rekey by restarting the proxy; a TODO comment indicates that actual rekey logic is not yet implemented in this framework.

[6]

0.4.6 Metrics Structures and Aggregation

The drone defines the authoritative metrics schema for a suite run in `SuiteMetrics`, which includes handshake timing, optional rekey timing, packet counters, latency statistics, power summaries, and system resource measurements. The GCS uses `SuiteBenchResult` to store a superset of these fields along with GCS-side CPU/memory and throughput computed from the traffic generator. The GCS stores results as a session JSON file via `SuiteBenchmarkRunner._save_results()` in `suite_benchmarks/raw_data/gcs`. [7]

0.4.7 Power and System Telemetry on the Drone

Power collection is implemented as a background thread in `PowerCollector`. It performs raw INA219 I2C reads at approximately 1 kHz and stores `PowerSample` records with timestamps. On suite stop, the server computes mean and peak power and estimates energy by integrating over the sample interval. System telemetry is gathered with `psutil` (if available) and the Raspberry Pi thermal zone file. These values are injected into `SuiteMetrics` at suite finalization. [8]

[5]Evidence: suite_benchmarks/framework/suite_bench_gcs.py (`TrafficGenerator.run()`, `SuiteBenchmarkRunner.run_suite_iteration()`).

[6]Evidence: suite_benchmarks/framework/suite_bench_drone.py (`BenchmarkControlServer`, `_start_suite()`, `_stop_suite()`, `_handle_rekey()`).

[7]Evidence: suite_benchmarks/framework/suite_bench_drone.py (`SuiteMetrics`); suite_benchmarks/framework/suite_bench_gcs.py (`SuiteBenchResult`, `_save_results()`).

[8]Evidence: suite_benchmarks/framework/suite_bench_drone.py (`PowerCollector`, `get_system_metrics()`, `_stop_suite()`).

0.4.8 Latency Tracking Status in Current Code

The framework defines a `LatencyTracker` with `record_send()` and `record_receive()` methods and populates latency percentiles during `_stop_suite()`. However, in the current code, there are no call sites that invoke `record_send()` or `record_receive()` within `suite_bench_drone.py`. As a result, latency statistics remain at default values unless another component invokes these methods externally, which is not present in this module. This is a documented limitation of the current framework implementation. [9]

0.4.9 Outputs and Reproducibility Artifacts

The GCS scheduler persists session-level results under `suite_benchmarks/raw_data/gcs` with a session ID timestamp. The drone server persists per-iteration JSON records under `suite_benchmarks/raw_data/dron` also tagged by session. The framework also logs GCS proxy stdout/stderr to `suite_benchmarks/logs` using timestamped filenames to support post-hoc diagnosis. [10]

[9]Evidence: suite_benchmarks/framework/suite_bench_drone.py (`LatencyTracker` definition; no invocations in module; `_stop_suite()` uses `get_stats()`).

[10]Evidence: suite_benchmarks/framework/suite_bench_gcs.py (`OUTPUT_DIR`, `LOGS_DIR`, `_save_results()`, `GCSProxyManager.start()`); suite_benchmarks/framework/suite_bench_drone.py (`OUTPUT_DIR`, `_save_result()`).

Part IV

Metrics and Dashboard

0.5 Metrics Pipeline and Dashboard Backend

0.5.1 Scope and Terminology

This part documents the metrics pipeline and the dashboard backend that serves the forensic metrics. The relevant terms are:

- **Comprehensive metrics schema**: The canonical A–R metrics contract defined as dataclasses in `core/metrics_schema.py`.
- **Metrics aggregator**: The collector-orchestrator `MetricsAggregator` that builds `ComprehensiveSuiteMetr` for each suite.
- **Metric status**: The per-field status map used to encode missing, invalid, or unavailable measurements.
- **Ingest store**: The backend `MetricsStore` that loads metrics artifacts into memory and exposes query views.

11

0.5.2 Canonical Schema and Backend Model Parity

The metrics schema enumerates 18 categories (A–R) as dataclasses, covering run context, crypto identity, lifecycle, handshake, crypto primitives, rekey, data plane, latency/jitter, MAVProxy layers, MAVLink integrity, flight controller telemetry, control plane, system resources, power/energy, observability, and validation. The dashboard backend mirrors this schema using Pydantic models to enforce the same field structure and to validate API responses. [12]

0.5.3 Aggregator Lifecycle and Role-Specific Collection

The `MetricsAggregator` is instantiated with a role of `gcs`, `drone`, or `auto` and selects collectors accordingly. It creates a `ComprehensiveSuiteMetrics` object in `start_suite()`, populates run context fields using `EnvironmentCollector`, and normalizes missing values by nulling empty strings while recording `metric_status` entries. [13]

The aggregator also starts background system sampling at 2 Hz and, when running on the drone, starts power sampling at 1 kHz via the power collector backend. MAVLink sniffing is started if the optional collector is available. [14]

0.5.4 Data Plane Derivation and Missing-Counter Handling

Data plane throughput and wire-rate are computed during `finalize_suite()` using proxy byte counters and the suite active duration. If counters are missing, the aggregator marks the entire

[11]Evidence: core/metrics_schema.py (module docstring and dataclasses); core/metrics_aggregator.py (`MetricsAggregator`); dashboard/backend/ingest.py (`MetricsStore`).

[12]Evidence: core/metrics_schema.py (A–R categories); dashboard/backend/models.py (Pydantic models mirroring the schema).

[13]Evidence: core/metrics_aggregator.py (`MetricsAggregator.__init__`, `start_suite()`, `_mark_metric_status()`); core/metrics_collectors.py (`EnvironmentCollector`).

[14]Evidence: core/metrics_aggregator.py (`_start_background_collection()`, `start_suite()` power and MAVLink start logic); core/metrics_collectors.py (`PowerCollector` and system collection).

data plane section as `not_collected`, nulls dependent fields, and also nulls rekey metrics because they are derived from proxy counters. [15]

0.5.5 Power/Energy Accounting and Handshake Energy

For drone-side runs with power samples, `finalize_suite()` uses the power collector's energy summary to populate average and peak power, total energy, and voltage/current averages. It also computes energy-per-handshake using handshake duration. If no samples are present, it sets power fields to `None` and records a `no_power_samples` status with the backend name. [16]

0.5.6 MAVLink-Derived Latency, Integrity, and Telemetry

When the MAVLink collector is available, the aggregator stops the collector at suite finalization, populates MAVProxy metrics, integrity counters, and latency/jitter statistics from the collector's output, and marks invalid latency fields with explicit reasons. Flight controller telemetry is populated on the drone side. When the collector is unavailable, the aggregator marks MAVLink-related sections as `not_collected` and nulls their fields. [17]

0.5.7 Observability and Validation Fields

The observability section records the number of system samples, sampling rate, and collection window. The validation section computes expected samples from the duration and marks missing fields when samples are absent. It also sets `benchmark_pass_fail` and `success_rate_percent` from handshake success, or marks them as not collected if handshake status is missing. All accumulated field statuses are written into `validation.metric_status`. [18]

0.5.8 Persistence and Peer Merge

The aggregator saves each suite as `{run_id}_{suite_id}_{role}.json` under `logs/comprehensive_metrics` unless an alternate output directory is configured. The aggregator can merge peer data (GCS or drone) into the current suite record, enabling cross-side context enrichment. [19]

0.5.9 Dashboard Backend API Surface

The backend service is a FastAPI application that exposes a root health endpoint and two primary API routes: `/api/runs` for run summaries and `/api/runs/{run_id}/suites` for per-run suite metrics. The endpoints return Pydantic models defined in the backend schema module. [20]

0.5.10 Ingest Pipeline and Store Semantics

The ingest module loads comprehensive metrics from `logs/benchmarks/comprehensive`, parses suite/run identifiers from filenames, and builds a `MetricsStore` that supports run listing and suite

[15] Evidence: core/metrics_aggregator.py (`finalize_suite()` data plane block; rekey nulling and `proxy_counters_missing` status).

[16] Evidence: core/metrics_aggregator.py (power block in `finalize_suite()`); core/metrics_collectors.py (power collector backend).

[17] Evidence: core/metrics_aggregator.py (MAVLink block in `finalize_suite()` and unavailable-collector path).

[18] Evidence: core/metrics_aggregator.py (observability and validation blocks; `metric_status` aggregation).

[19] Evidence: core/metrics_aggregator.py (`_save_metrics()`, `_merge_peer_data()`).

[20] Evidence: dashboard/backend/main.py (FastAPI app and routes); dashboard/backend/models.py (response models).

filtering by crypto families and NIST level. The store constructs run summaries by grouping suites and exposes filters via `get_unique_values()`. [21]

0.5.11 GCS JSONL Merge and Scientific Validity Gate

If GCS JSONL metrics are found, the ingest layer merges GCS system metrics, latency/jitter summaries, MAVLink validation counters, and proxy counters into the comprehensive suite objects, recording the source of each category. It then evaluates a scientific validity predicate requiring at least one of handshake, latency, or throughput to be present; suites failing this check are marked with an `invalid_run` status and a validation reason. [22]

0.5.12 JSONL Fallback and Missing-Data Transparency

If comprehensive metrics are absent, the ingest module falls back to JSONL entries and builds a minimal suite record. It explicitly annotates missing fields using `validation.metric_status` with a `not_collected` status and a `missing_comprehensive_metrics` reason, preserving provenance for downstream UI queries. [23]

0.6 Dashboard Frontend

0.6.1 Scope and Terminology

This part documents the analyst-facing dashboard frontend implemented in `dashboard/frontend/src`. Key terms used in this section are:

- **Frontend shell**: The top-level React component `App.tsx` that defines routing, navigation, and global UI banners.
- **Dashboard store**: The Zustand store in `state/store.ts` that coordinates data fetching and filter state.
- **Suite summary**: The lightweight suite records displayed in tables and used for charts (loaded via `/api/suites`).
- **Suite detail**: The full forensic metrics record displayed in `SuiteDetail`, loaded via `/api/suite/{suiteKey}`

24

0.6.2 Routing, Navigation, and Global UI Contracts

The frontend shell defines routes for the overview, suite explorer, per-suite detail, comparison, bucket comparisons, power analysis, and integrity monitoring. The navigation bar is derived from a static list of route definitions. A disclaimer banner is always rendered and states that the dashboard is observational and does not imply causality. The shell also renders a global error display bound to the store's `error` field. [25]

[21] Evidence: dashboard/backend/ingest.py (`COMPREHENSIVE_DIR`, `_load_comprehensive()`, `MetricsStore.list_runs()`, `MetricsStore.list_suites()`, `MetricsStore.get_unique_values()`).

[22] Evidence: dashboard/backend/ingest.py (`_load_gcs_jsonl_entries()`, `_merge_gcs_metrics()`, `_is_suite_scientifically_valid()`, `build_store()`).

[23] Evidence: dashboard/backend/ingest.py (`_build_minimal_suite()`, JSONL fallback logic).

[24] Evidence: dashboard/frontend/src/App.tsx; dashboard/frontend/src/state/store.ts; dashboard/frontend/src/-pages/SuiteExplorer.tsx; dashboard/frontend/src/pages/SuiteDetail.tsx.

[25] Evidence: dashboard/frontend/src/App.tsx (`Navigation`, `DisclaimerBanner`, routes, and error display).

0.6.3 State Store and API Contract

The Zustand store centralizes the frontend data model and the API contracts. It manages suite summaries, run summaries, filters, selected suite detail, comparison suites, and filter state. It defines fetch methods for: `/api/suites`, `/api/runs`, `/api/suites/filters`, and `/api/suite/{suiteKey}`. Filter state is mapped into query parameters (KEM family, signature family, AEAD, NIST level, and run ID) for suite retrieval. [26]

0.6.4 Overview Page: Derived Aggregates and Health Signals

The overview page loads suites and runs, queries `/api/health`, and requests an aggregate endpoint `/api/aggregate/kem-family`. It renders cards for total suites and runs while marking pass-rate and failed-suite cards as `NOT AVAILABLE`. It also renders multiple bar charts that visualize aggregated handshake time, power, goodput, loss ratio, and latency metrics by KEM family. If the aggregate endpoint returns a warning, the UI displays the warning and disables the charts. [27]

0.6.5 Suite Explorer: Filters and Summary Table

The suite explorer binds filter dropdowns to the store and re-fetches suite summaries on filter changes. The summary table presents suite identifiers and key fields (KEM, signature, AEAD, security level, handshake duration, power, and energy). Missing values are displayed as `Not collected`. The status cell uses a badge that maps PASS and FAIL to distinct styles. [28]

0.6.6 Suite Detail: Reliability and Metric Status Rendering

The suite detail view loads a comprehensive suite record using a `suiteKey` of the form `run_id:suite_id`. It uses a reliability badge model and renders each field using a `MetricValue` component that interprets `metric_status`. This produces explicit labels for invalid, not-implemented, and not-collected fields. The view organizes metrics by schema section letters (A, B, D, M, G, H, and others), and annotates latency and packet counters with the recorded data source (e.g., `latency_source`, `packet_counters_source`). [29]

0.6.7 Comparison View: Side-by-Side Suite Metrics

The comparison view uses suite summaries to select two suites by key, then loads each suite's full metrics record. It filters out metrics with missing values before charting and displays a vertical bar chart comparing handshake, goodput, packet loss, power, energy, and CPU. A detailed table provides numeric values or `Not collected` for each metric. [30]

0.6.8 Bucket Comparison: Grouped Views and Hardcoded Endpoint

The bucket comparison view requests `http://localhost:8000/api/buckets` directly rather than using the store's `API_BASE`. It groups suites by multiple bucket types (KEM family, signature family, AEAD, and NIST level composites), allows a subgroup selection, and renders handshake

[26]Evidence: dashboard/frontend/src/state/store.ts (API methods and filter query construction).

[27]Evidence: dashboard/frontend/src/pages/Overview.tsx.

[28]Evidence: dashboard/frontend/src/pages/SuiteExplorer.tsx.

[29]Evidence: dashboard/frontend/src/pages/SuiteDetail.tsx.

[30]Evidence: dashboard/frontend/src/pages/ComparisonView.tsx; dashboard/frontend/src/state/store.ts (`fetchComparison`).

and power/energy comparisons as bar charts. This explicit endpoint choice means the page is coupled to a localhost backend configuration rather than relative API routing. [31]

0.6.9 Power Analysis: Aggregation, Scatter, and Ranking

The power analysis page computes energy aggregates by KEM family from suite summaries and renders a bar chart of average energy. It also renders a scatter plot of power versus handshake duration with bubble size proportional to energy, and a top-10 energy consumer table. The summary cards calculate overall averages and totals using only collected values and display `Not collected` when values are missing. [32]

0.6.10 Integrity Monitor: Metadata-Driven Alerts

The integrity monitor scans suite summaries and raises issues based on metadata-only rules: benchmark failure, handshake failure, missing power data, and unusually high handshake duration. It assigns severity tiers and provides links to the suite detail page for forensic review. The view emphasizes that it does not replace detailed MAVLink integrity analysis and defers that analysis to the suite detail view. [33]

[31]Evidence: dashboard/frontend/src/pages/BucketComparison.tsx.
[32]Evidence: dashboard/frontend/src/pages/PowerAnalysis.tsx.
[33]Evidence: dashboard/frontend/src/pages/IntegrityMonitor.tsx.

Part V

Tools, Telemetry, and Operations

0.7 Tools, Diagnostics, and Orchestration

0.7.1 Scope and Terminology

This part documents the standalone utilities and operator-facing tooling in `tools/` and `scripts/`. These utilities are not part of the core proxy runtime; they provide orchestration, diagnostics, verification, and data-transfer support. The key terms are:

- **Orchestration helper**: A wrapper that launches the GCS scheduler and the drone loop and streams logs.
- **Preflight diagnostics**: Network checks that validate bindings, UDP reachability, and control-plane TCP connectivity.
- **Truth verification**: Scripts that check metrics artifacts for missing-data transparency and forbidden field usage.
- **Data transfer utility**: A helper that copies benchmark artifacts from the drone to the GCS host.

[34]

0.7.2 End-to-End Orchestration Helper

The orchestration helper in `tools/orchestrate_run.py` starts the GCS benchmark server, waits for stabilization, then launches the drone benchmark loop via SSH. It streams stdout from both processes, monitors the drone return code, and shuts down the GCS server on completion. A non-zero drone exit code causes the helper to exit with that code, ensuring reproducible CI-like behavior. [35]

0.7.3 Network Diagnostics and Clock Skew Checks

The network diagnostic tool performs preflight validation without invoking the proxy. It detects role based on configured IPs, attempts to bind local UDP/TCP ports, runs ICMP reachability checks, sends UDP diagnostic packets, echoes responses, and reports clock skew if peer timestamps differ by more than 1 second. It can also test the TCP control port by binding on GCS or connecting from the drone. [36]

0.7.4 MAVProxy Process Management

The shared `MavProxyManager` encapsulates cross-platform MAVProxy launch logic. It builds a master/out configuration, resolves the MAVProxy entry point (explicit script, module invocation on Windows, or PATH binary on Linux), and launches the process through `ManagedProcess`. Logs are written to `logs/sscheduler/{role}` with timestamped filenames, and the manager exposes `start()`, `stop()`, and `is_running()`. [37]

[34]Evidence: tools/orchestrate_run.py; tools/net_diag.py; tools/verify_metrics_truth.py; tools/verify_dashboard_truth.py; scripts/transfer_benchmark_data.py.

[35]Evidence: tools/orchestrate_run.py (`GCS_CMD`, SSH launch, log streaming, shutdown logic).

[36]Evidence: tools/net_diag.py (role detection, bind checks, UDP echo, clock skew warnings, TCP control test).

[37]Evidence: tools/mavproxy_manager.py.

0.7.5 GCS Control Ping Utility

The TCP control ping utility sends JSON commands to a control server and prints the response. It supports an arbitrary command string and optional suite parameter, enabling deterministic liveness checks and manual command injection for tests. [38]

0.7.6 Metrics Truth Verification

Two verification scripts enforce transparency and correctness rules for metrics artifacts. `verify_metrics_truth.py` checks JSONL records against a reliability policy (imported from the dashboard backend) and flags any `NOT_COLLECTED` fields that contain non-default values. `verify_dashboard_truth.py` loads comprehensive metrics and asserts that displayed fields have matching `metric_status` entries when missing, while also enforcing consistency rules for latency and RTT samples. [39]

0.7.7 Comprehensive Metrics Availability Gate

The `wait_for_comprehensive_metrics.py` tool blocks until comprehensive A–R metrics are present and structurally complete. It requires both `_drone.json` and `_gcs.json` files for a run (unless disabled), validates expected field counts per schema category, and rejects runs where handshake success is false or data-plane counters are empty. [40]

0.7.8 Power and Rekey Diagnostics Helpers

The toolset includes small utilities for power and blackout diagnostics. `ina219_read.py` probes INA219 sensors across multiple I2C buses and addresses to verify power monitor availability. `power_utils.py` loads power CSV traces, normalizes field names, and integrates energy over time windows. `blackout_metrics.py` computes rekey blackout estimates by analyzing packet timing gaps and optionally correlating them to marked rekey intervals. [41]

0.7.9 Packet-Level MAVLink UDP Sniffing

The `mav_udp_sniff.py` tool performs a minimal UDP packet counter on specified ports for a given duration. It binds sockets, counts received datagrams, and reports per-port counts. This is useful for verifying traffic presence independent of proxy logic. [42]

0.7.10 Configuration and Suite Registry Dumps

Two simple utilities support reproducible configuration capture. `dump_config.py` prints selected configuration values (hostnames, control port, PSK length, environment flags). `dump_suites.py` writes the suite registry into `benchmark_plan.json` for inspection and offline audit. [43]

[38]Evidence: tools/gcs_control_ping.py.
[39]Evidence: tools/verify_metrics_truth.py; tools/verify_dashboard_truth.py.
[40]Evidence: tools/wait_for_comprehensive_metrics.py.
[41]Evidence: tools/ina219_read.py; tools/power_utils.py; tools/blackout_metrics.py.
[42]Evidence: tools/mav_udp_sniff.py.
[43]Evidence: tools/dump_config.py; tools/dump_suites.py.

0.7.11 Benchmark Data Transfer Utility

The data transfer utility `scripts/transfer_benchmark_data.py` copies benchmark artifacts from the drone to the GCS host. It discovers available runs by listing the remote benchmark directory over SSH, selects a run (latest by default), and transfers files via `rsync` when available or `scp` as a fallback. The script logs progress and verifies the presence of JSONL artifacts after transfer. [44]

0.7.12 RPC Connectivity Check

The `scripts/verify_rpc.py` script performs a Chronos sync RPC by calling `send_gcs_command` and validates that the response includes a timing field. It is a direct sanity check for the control-plane RPC path used by the scheduler stack. [45]

0.8 MAVLink Integrity and Power Telemetry

0.8.1 Scope and Terminology

This part documents two measurement subsystems: MAVLink integrity/latency telemetry and power monitoring. The key terms are:

- **MAVLink collector**: The `MavLinkMetricsCollector` responsible for parsing MAVLink messages and deriving integrity/latency metrics.
- **Integrity metrics**: Counters for sequence gaps, duplicates, out-of-order packets, decode errors, and CRC errors.
- **Latency/jitter metrics**: One-way latency and jitter derived from timestamped MAVLink messages, and RTT derived from command/ack pairs.
- **Power monitor**: The INA219-backed sampling engine in `core/power_monitor.py` and the higher-level `PowerCollector` in `core/metrics_collectors.py`.

[46]

0.8.2 MAVLink Collector Initialization and Sniffing

The MAVLink collector is instantiated with a role (GCS or drone) and starts sniffing via `start_sniffing(port, host)`. It requires `pymavlink` and uses a UDP connection string `udpin:host:port`. The collector resets internal counters for each suite to avoid cross-suite accumulation. [47]

0.8.3 Message-Level Tracking and Integrity Counters

Each MAVLink message is parsed and processed in `_handle_message()`. The collector tracks total RX/TX counts, per-message type counts, and estimated payload sizes. Sequence tracking detects gaps, duplicates, and out-of-order packets using per-system last-seen sequence numbers. CRC errors, decode errors, and message drops are tracked explicitly. [48]

[44]Evidence: scripts/transfer_benchmark_data.py.
[45]Evidence: scripts/verify_rpc.py.
[46]Evidence: core/mavlink_collector.py; core/power_monitor.py; core/metrics_collectors.py.
[47]Evidence: core/mavlink_collector.py (`__init__()`, `start_sniffing()`, `reset()`, `_sniff_loop()`).
[48]Evidence: core/mavlink_collector.py (`_handle_message()`, `_track_sequence()`, integrity counters).

0.8.4 Heartbeat and Command/ACK Telemetry

Heartbeat metrics include observed count, expected count (1 Hz), loss count, average interval, and the system/component identifiers. Command metrics track sent commands, pending acknowledgments, and ACK latency samples. RTT samples are derived from command/ack pairs, and invalidity reasons are recorded if commands are not sent or ACKs are missing. [49]

0.8.5 One-Way Latency and Jitter Derivation

Latency is derived from MAVLink timestamp fields. If `time_usec` is present and plausible, it is used directly; otherwise, `time_boot_ms` is aligned to Unix time using `SYSTEM_TIME` to establish a boot-to-unix offset. Samples are filtered for plausibility, jitter is computed as the absolute difference between consecutive latency samples, and invalidity reasons are surfaced when timestamps or sample counts are insufficient. [50]

0.8.6 Schema Population for MAVLink Metrics

The collector exposes `populate_schema_metrics()` to fill the MAVProxy metrics section and `populate_mavlink_integrity()` to fill integrity counters in the comprehensive schema. GCS role populates a validation-only subset, while the drone role includes message histograms and command-latency fields. [51]

0.8.7 Flight Controller Telemetry Extraction

The collector derives flight controller telemetry from MAVLink messages such as `GLOBAL_POSITION_INT`, `SYS_STATUS`, `BATTERY_STATUS`, and `ATTITUDE`. It computes update rates for attitude and position and captures battery voltage/current, remaining percentage, CPU load, and sensor health flags. These fields are exported through `get_flight_controller_metrics()`. [52]

0.8.8 Power Monitor Backends and Configuration

The power monitoring subsystem provides a dedicated INA219 sampler in `Ina219PowerMonitor` with configurable I2C bus, address, shunt resistance, and sample rate. It selects ADC profiles based on requested sample rates and supports sign handling (auto or fixed). Failure to access I2C or configuration errors result in `PowerMonitorUnavailable` exceptions. [53]

0.8.9 Power Capture and Energy Summary

The INA219 monitor captures samples to CSV at the requested sample rate, records per-sample timestamps, and returns a `PowerSummary` with average current/voltage/power, total energy, sample rate, and capture window bounds. This provides deterministic energy accounting for suite-level analysis. [54]

[49] Evidence: core/mavlink_collector.py (`_handle_heartbeat()`, `_handle_command_ack()`, RTT validity logic in `get_metrics()`).

[50] Evidence: core/mavlink_collector.py (`_track_message_latency()`, latency validity logic in `get_metrics()`).

[51] Evidence: core/mavlink_collector.py (`populate_schema_metrics()`, `populate_mavlink_integrity()`).

[52] Evidence: core/mavlink_collector.py (message handlers and `get_flight_controller_metrics()`).

[53] Evidence: core/power_monitor.py (configuration defaults, ADC profiles, `Ina219PowerMonitor`, `PowerMonitorUnavailable`).

[54] Evidence: core/power_monitor.py (`capture()`, `PowerSummary`).

0.8.10 Collector-Level Power Integration

The higher-level `PowerCollector` in `core/metrics_collectors.py` selects an available backend (INA219, RPi5 hwmon, or none), samples in a background thread, and computes energy using trapezoidal integration across samples. It returns summary statistics including total energy, average/peak power, and voltage/current averages. [55]

0.9 Process Management, Logging, and Error Semantics

0.9.1 Scope and Terminology

This part documents cross-platform process management, structured logging, and the explicit error taxonomy used throughout the system. Key terms include:

- **Managed process**: A subprocess wrapped by `ManagedProcess` with parent ownership and controlled termination semantics.
- **Job Object**: The Windows mechanism used to ensure child processes terminate when the parent exits.
- **PDEATHSIG**: The Linux mechanism (`prctl(PR_SET_PDEATHSIG)`) used to ensure child termination on parent death.
- **JSON log record**: The structured output format emitted by the logging utility with a fixed field schema.

[56]

0.9.2 Managed Process Lifecycle and Ownership

The `ManagedProcess` class is a unified wrapper around `subprocess.Popen` that ensures ownership and consistent shutdown across platforms. On Windows it uses Job Objects with the `KILL_ON_JOB_CLOSE` limit, and on Linux it sets `PR_SET_PDEATHSIG` and creates a new session via `setsid()` to support process group termination. All managed processes are registered in a global registry, and a shutdown hook calls `kill_all_managed_processes()` on interpreter exit. [57]

0.9.3 Termination Semantics and Cleanup

Shutdown uses a staged termination strategy. On Linux, the process group receives `SIGTERM` and is followed by `SIGKILL` if the timeout elapses. On Windows, `taskkill /F /T` is invoked to terminate the process tree; an additional kill call is issued as a fallback. Job handles are closed during cleanup to release OS resources. [58]

[55]Evidence: core/metrics_collectors.py (`PowerCollector`, `start_sampling()`, `stop_sampling()`, `get_energy_stats()`).

[56]Evidence: core/process.py; core/logging_utils.py; core/exceptions.py.

[57]Evidence: core/process.py (platform-specific setup, `ManagedProcess.start()`, registry and `atexit` hook).

[58]Evidence: core/process.py (`ManagedProcess.stop()` implementation).

0.9.4 Structured JSON Logging

The logging utility exposes `get_logger()` with a `JsonFormatter` that emits JSON records containing timestamp, level, logger name, and message. Exception details are serialized when present. The formatter also attempts to include extra attributes attached to the log record if they are JSON-serializable. [59]

0.9.5 File Logger Attachment and Handler Hygiene

The `configure_file_logger()` helper attaches a JSON file handler and returns the log path. It removes any previously attached file handlers to avoid duplicate outputs during tests or repeated initialization. Log files are written under `logs/` with a timestamped filename and the role prefix. [60]

0.9.6 Minimal In-Process Metrics Hooks

The logging utility also defines a minimal `Metrics` container with `Counter` and `Gauge` primitives, enabling lightweight instrumentation without external dependencies. These are used as in-process counters for components that want to track simple numeric state. [61]

0.9.7 Error Taxonomy and Semantic Clarity

The codebase defines a concise error taxonomy in `core/exceptions.py`. It includes configuration errors (`ConfigError`), sequence exhaustion (`SequenceOverflow`), handshake errors (`HandshakeError` and its format/verify variants), and AEAD errors (`AeadError`). These typed exceptions preserve semantic meaning for upstream error handling and observability. [62]

0.9.8 Proxy Error Attribution and Counters

The proxy maintains structured counters for packet drops and failure modes (replay drops, auth failures, header mismatches, session epoch errors) alongside handshake and primitive timing summaries. This enables error attribution in the data plane and provides the basis for higher-level metrics aggregation and validation. [63]

0.10 Suite Registry and Scheduling Policy

0.10.1 Scope and Terminology

This part describes how cryptographic suites are registered and how scheduling policies select suites at runtime. Key terms are:

- **Suite registry**: The canonical mapping of suite IDs to KEM/AEAD/SIG metadata in `core/suites.py`.
- **Suite ID**: The canonical identifier of the form `cs-<kem>-<aead>-<sig>`.

[59]Evidence: core/logging_utils.py (`JsonFormatter`, `get_logger()`).
[60]Evidence: core/logging_utils.py (`configure_file_logger()`).
[61]Evidence: core/logging_utils.py (`Counter`, `Gauge`, `Metrics`).
[62]Evidence: core/exceptions.py.
[63]Evidence: core/async_proxy.py (`ProxyCounters` fields and handshake primitive aggregation).

- **Benchmark policy**: A deterministic, exhaustive policy that cycles through suites for measurement.
- **Telemetry-aware policy**: A safety-oriented policy that upgrades/downgrades suites based on telemetry and hysteresis.

[64]

0.10.2 Registry Structure: KEM, Signature, and AEAD Catalogs

The suite registry is composed of three primary catalogs: KEMs, signatures, and AEADs. Each entry includes a token, OQS or display name, NIST level, embedded identifier fields (e.g., `kem_id`, `sig_id`), and a list of aliases. The registry explicitly maps ML-DSA-44 to NIST L1 for practical pairing with L1 KEMs. [65]

0.10.3 Alias Normalization and Suite ID Construction

Alias normalization removes punctuation and casing to resolve tokens consistently. `build_suite_id()` resolves aliases for KEM, AEAD, and signature names and constructs the canonical suite ID. A fixed suite alias table maps legacy Kyber/Dilithium and SPHINCS+ naming conventions into the canonical ML-KEM/ML-DSA suite identifiers. [66]

0.10.4 Level-Consistent Suite Matrix Generation

Suites are generated by pairing KEM and signature entries that share the same NIST security level. The registry produces a deterministic matrix sorted by KEM and signature token, then expands across a fixed AEAD order. Environment variables can exclude specific KEMs or AEADs from suite generation without removing them from the registry. [67]

0.10.5 Runtime AEAD Availability and Suite Pruning

AEAD availability is probed at runtime. AES-GCM is always included; ChaCha20-Poly1305 and Ascon-128a are optional based on installed dependencies and configuration flags. Ascon support is detected via a native module or the pure-Python fallback. A runtime pruning step removes suites whose signatures are not supported by the current OQS environment. [68]

0.10.6 Suite Accessors and Header ID Embedding

The registry exposes `list_suites()` and `get_suite()` to retrieve immutable suite configurations, resolving aliases and validating required fields. The helper `header_ids_for_suite()` returns embedded identifier tuples and `header_ids_from_names()` resolves IDs from algorithm names for runtime header validation. Suite IDs are also serialized into bytes for HKDF info via `suite_bytes_for_hkdf()`. [69]

[64]Evidence: core/suites.py; sscheduler/benchmark_policy.py; sscheduler/policy.py.

[65]Evidence: core/suites.py (`_KEM_REGISTRY`, `_SIG_REGISTRY`, `_AEAD_REGISTRY`, module docstring notes).

[66]Evidence: core/suites.py (`_normalize_alias()`, `_build_alias_map()`, `build_suite_id()`, `_SUITE_ALIASES`).

[67]Evidence: core/suites.py (`_generate_level_consistent_matrix()`, `_AEAD_ORDER`, `_generate_suite_registry()`).

[68]Evidence: core/suites.py (`_probe_aead_support()`, `available_aead_tokens()`, `unavailable_aead_reasons()`, `_prune_suites_for_runtime()`).

[69]Evidence: core/suites.py (`list_suites()`, `get_suite()`, `header_ids_for_suite()`, `header_ids_from_names()`, `suite_bytes_for_hkdf()`).

0.10.7 Benchmark Policy: Deterministic Exhaustive Coverage

The benchmark policy in `sscheduler/benchmark_policy.py` loads `settings.json` and reads the `benchmark_mode` section to configure cycling intervals and output directories. It constructs an ordered suite list, optionally filtering by AEAD token, and sorts suites by NIST level, KEM, and signature for organized reporting. It maintains a run ID, per-suite iteration metrics, and writes results to `logs/benchmarks`. [70]

0.10.8 Benchmark Metrics Capture Hooks

The benchmark policy includes a `SuiteMetrics` record that stores handshake, throughput, latency, and power metrics alongside cryptographic primitive timings and artifact sizes. Handshake metrics are populated from proxy handshake timing structures, and suite metadata (KEM, signature, AEAD, NIST level) is captured at suite start. [71]

0.10.9 Telemetry-Aware Policy: Settings and Filtering

The telemetry-aware policy `TelemetryAwarePolicyV2` loads thresholds and limits from `settings.json`. It filters the suite list by allowed AEAD and maximum NIST level, then sorts suites by a deterministic tier function that combines NIST level and AEAD/KEM sub-tiers. [72]

0.10.10 Telemetry-Aware Decision Logic

The policy evaluates a `DecisionInput` snapshot and applies a structured sequence: telemetry freshness gate, emergency downgrade on critical battery/thermal thresholds, rollback on blackouts shortly after a switch, cooldown gating, downgrade on sustained link degradation, downgrade on sustained thermal/battery stress, optional rekey when stable and within rate limits, and conservative upgrade only when disarmed and stable. The output encodes action, target suite, and reasons. [73]

0.10.11 Blacklist, Hysteresis, and Rekey Rate Limits

The policy maintains a blacklist with TTL to avoid oscillations after failure, uses per-condition hysteresis timers for downgrade and upgrade decisions, and records successful rekeys in a sliding window to enforce rekey rate limits. The rekey timestamp list is updated only after successful rekey execution. [74]

0.10.12 Simple Scheduler Policies

In addition to the telemetry-aware policy, the scheduler defines simpler suite selection policies: `LinearLoopPolicy` for round-robin sequencing, `RandomPolicy` for stochastic selection, and `ManualOverridePolicy`

[70] Evidence: sscheduler/benchmark_policy.py (`load_benchmark_settings()`, `BenchmarkPolicy.__init__()`, `_build_suite_list()`, `SuiteMetrics`)).

[71] Evidence: sscheduler/benchmark_policy.py (`SuiteMetrics`, `_start_suite_metrics()`, `record_handshake_metrics()`).

[72] Evidence: sscheduler/policy.py (`load_settings()`, `TelemetryAwarePolicyV2.__init__()`, `_filter_suites()`, `get_suite_tier()`); settings.json.

[73] Evidence: sscheduler/policy.py (`DecisionInput`, `PolicyOutput`, `TelemetryAwarePolicyV2.evaluate()`).

[74] Evidence: sscheduler/policy.py (`_add_blacklist()`, `_check_hysteresis()`, `record_rekey()` and rekey window logic).

to allow explicit suite pinning with a linear fallback. These policies implement minimal `next_suite()` and duration accessors. [75]

[75] Evidence: sscheduler/policy.py (`LinearLoopPolicy`, `RandomPolicy`, `ManualOverridePolicy`).