IoTGuard: AI-Based Intrusion Detection System for IoT Networks

Low-Level Design Report (Version 1)

Course: CMPE 492 – Graduation Project II

Supervisor: Prof. Dr. Gökçe Nur Yılmaz

Team Members:

• Ahmad Ismail – 99000332006

• Ateş Öztürk – 21145665572

• Esra Gürgen – 40654688056

Submission Date: October 28, 2025

1. Introduction

IoTGuard is a network intrusion detection and response system that ingests high-volume telemetry (Suricata eve.json, NetFlow/IPFIX, Syslog), aggregates flow into short windows, extracts statistical features, scores them with both supervised (LightGBM) and unsupervised (Isolation Forest/Autoencoder) models, and enforces policy via a decision engine (thresholds, grace/windowing, cooldown, instant block). A Flask dashboard exposes observability and remote tuning.

1.1 Object Design Trade-offs

A. Data pipeline vs. tight coupling

Choice: Keep collectors (Suricata tail), feature extraction, scoring, and decisioning as separate processes communicating via files/streams (CSV + JSONL) rather than a single monolith.

Why: Improves fault isolation, hot-swapping models and configs, and makes local debugging easy.

Trade-off: Slight I/O overhead (CSV/JSONL) vs. an in-memory bus; mitigated by small windows (e.g., 5–10 s) and bounded file sizes with rotation.

B. Feature schema stability vs. model flexibility

Choice: Fix a minimal, stable feature schema (flows, bytes_total, pkts_total, uniq_src, uniq_dst, syn_ratio, mean_bytes_flow).

Why: Stable schema simplifies the decision loop, dashboard, and documentation.

Trade-off: Fewer raw signals per window; we offset with unsupervised scoring and leave room for optional columns guarded by schema validation.

C. Supervised + Unsupervised fusion vs. single model

Choice: Use score fusion (e.g., weighted or calibrated combination).

Why: Supervised models capture known attacks; unsupervised models flag novel patterns.

Trade-off: More moving parts and calibration steps; mitigated with a simple linear fusion and thresholds exposed in config.

D. Deterministic policy vs. adaptive auto-tuning

Choice: Deterministic DecisionEngine (threshold, grace, window, cooldown, instant-block).

Why: Predictable blocking behavior for a security product; easier to explain to stakeholders.

Trade-off: Requires occasional human tuning; mitigated with a dashboard and runtime hot-reload of YAML config.

E. Simulated vs. real blocking hooks

Choice: Provide pluggable enforcement (Windows netsh advfirewall, Linux iptables/nft, or webhook).

Why: Lets teams adopt progressively (monitor → alert → enforce).

Trade-off: Platform-specific glue code; hidden behind a common BlockHook interface.

1.2 Interface Documentation Guidelines

To keep the LLD actionable and consistent, each interface is specified with:

Purpose: One-sentence intent.

Inputs / Outputs: Types, units, allowed ranges, and edge cases.

Contract & Errors: Pre/postconditions, failure modes, timeouts, retries, and idempotency.

Schema / Versioning: CSV headers, JSON fields, and version fields when applicable.

Security & Privacy: What's logged (and what's not), redaction rules, and least-privilege assumptions.

Examples:

- features.csv (Producer: FeatureExtractor; Consumer: DecisionLoop)
- Columns (required): flows:int, bytes_total:int, pkts_total:int, uniq_src:int, uniq_dst:int, syn_ratio:float (0-1), mean_bytes_flow:float
- Contract: No NaNs; all numeric; rows are time-ordered aggregates for a fixed window (e.g., 10s).
- Errors: If there are missing columns or non-numeric fields, consumer waits and logs a warning.
- alerts.jsonl (Producer: DecisionLoop; Consumer: Dashboard)
- Fields:
- ts: float (epoch seconds, UTC)
- index: int (row index in features.csv)
- score: float (0-1 fused)
- state: "ATTACK"|"benign"
- hits_in_window: int
- action: "BLOCK"|"NONE"
- Contract: One event per processed row; append-only; rotated when size exceeds a configured limit.
- Errors: Malformed lines are ignored by readers (lenient parse).
- /api/config (Flask)
- GET: Returns current decision parameters.
- POST: Accepts {decision: {threshold, grace, window, cooldown_sec, instant_block } } (types enforced).
- Contract: Changes persist to configs/model.yaml. Decision loop hot-reloads on file mtime change.

1.3 Engineering Standards

Modeling & Diagrams:

- UML/PlantUML: Component, sequence, class, deployment, and state diagrams (included in later sections).

- Naming: UpperCamelCase for classes, snake_case for functions/variables, kebab-case for CLI flags.
- Specification & Documentation
- IEEE 1016 / 29148 principles for LLD structure and requirements traceability.
- Docstrings & Type Hints: Python type annotations; Pydantic-style validation where applicable.
- Coding Standards
- Python: PEP 8, PEP 257; mypy-friendly typing for public APIs; black/ruff for formatting/linting.
- Logging: Structured logs where useful (JSON), otherwise standard logging with levels (INFO/WARN/ERROR).
- Data Formats
- CSV: UTF-8, header row required, comma delimiter, no empty columns.
- JSON Lines: UTF-8, one JSON object per line, lenient readers (ignore malformed).
- YAML: Configuration (configs/model.yaml), validated on load; unknown keys ignored with warning.
- APIs
- RESTful conventions: Nouns for resources, explicit verbs for actions; clear status codes (200/201, 400, 404, 500).
- Versioning: Config schema version and API minor version in responses when breaking changes are possible.
- Security & Privacy
- Principle of least privilege: Readers get read-only paths; block hooks run with necessary OS rights only.
- Data minimization: No PII persisted; IPs only if required for enforcement; redact were logged publicly.
- Defense in depth: Input validation, safe file handling, rotate logs, and restrict dashboard bind address in dev.
- Testing & CI
- Unit tests: Feature aggregation, schema validators, decision rules, API endpoints (happy & edge paths).

- Integration tests: Tail → features → decision → alerts end-to-end with a seeded eve.json.
- Reproducibility: Model artifacts stored with metadata (training set hash, feature list, metrics).

1.4 Definitions, Acronyms, and Abbreviations

Term	**Definition**
Suricata	Open-source IDS/IPS producing structured telemetry in eve.json.
eve.json	Suricata's line-delimited JSON log containing events (flow, dns, http, tls, alert, …).
Flow	Bidirectional tuple (5-tuple context) with counters such as bytes_toserver, pkts_toclient, state.
Window	Fixed time slice (e.g., 10s) over which flows are aggregated to produce one feature row.
Feature row	One CSV row summarizing window statistics used as ML input.
Supervised score (p)	Probability from a trained classifier (LightGBM) that the window is malicious.
Unsupervised score (a)	Anomaly score from IF/AE indicating deviation from normal behavior.
Fused score	Combined score of supervised & unsupervised signals (e.g., weighted or calibrated).
DecisionEngine	Deterministic policy using thresholds, grace, sliding window, cooldown, and instant-block.

Term	**Definition**
Grace	Minimum number of ATTACK flags within the sliding window to trigger a block (burst rule).
Cooldown	Minimum time between two block actions to avoid flapping.
Instant-block	High-confidence single-row threshold that triggers immediate block regardless of grace.
alerts.jsonl	Line-delimited JSON file of decisions emitted by the decision loop.
Dashboard	Flask web UI that shows counts, charts, recent events, and allows live parameter tuning.
Block hook	Pluggable enforcer (e.g., Windows netsh advfirewall, Linux iptables/nft, webhook).

2. Package Overview

The IoTGuard system uses a modular, layered package structure. This method is to make sure tasks are clearly separated for better scalability and easy maintenance. Every package contains a particular duty - starting from gathering and altering data to drawing conclusions from models, making decisions and displaying them. The communication of packages is managed by using lightweight, structured data formats such as CSV, JSONL and YAML. In addition to this, local message queues or streams are used when these are deployed in the production environment.

2.1 Packages

Package	**Description**	**Key Technologies**
collectors	Handles network telemetry ingestion from Suricata (eve.json), NetFlow/IPFIX, and Syslog sources. Each collector normalizes data into a unified event format.	Suricata, Python JSON parser
stream_buffer	Temporary message relay for real-time event forwarding between collectors and feature extraction. Can use MQTT or Redis Streams in production.	Redis, MQTT, Python asyncio
feature_extractor	Aggregates and summarizes raw flow data into statistical feature windows suitable for ML models. Implements rolling windows, schema validation, and CSV output.	Pandas, datetime, CSV
models	Hosts the machine learning models — supervised (LightGBM) and unsupervised (Isolation Forest or Autoencoder). Responsible for inference, calibration, and model persistence.	LightGBM, scikit-learn, joblib
decision_engine	Applies deterministic rule-based logic (threshold, grace, window, cooldown, instant-block) to the	Python core, JSONL

Package	Description	Key Technologies
	fused ML scores and emits final BLOCK or NONE actions.	
block_hooks	Contains OS-level or external integrations that enforce blocking actions or send alerts (e.g., firewall, webhook, SIEM).	Windows netsh / iptables / requests
api_dashboard	Flask-based dashboard providing a web interface for monitoring alerts, tuning thresholds, and visualizing feature data.	Flask, Chart.js, Bootstrap
configs	Stores runtime YAML configurations for models, decision parameters, and dashboard preferences. Supports hot-reloading.	PyYAML
data	Persistent data storage containing raw logs (eve.json), feature windows (features.csv), and alert logs (alerts.jsonl).	Local FS or mounted volume

2.2 Data Flow Between Packages

- collectors → stream_buffer
- Suricata outputs eve.json (JSON lines).
- The buffer (or file tailer) picks up new lines in real time.
- stream_buffer → feature_extractor
- The extractor parses flow events, aggregates them in rolling windows (e.g., every 10 seconds), and appends rows to features.csv.
- feature_extractor → models
- The ML models read new feature rows and output fused anomaly scores (0–1).
- models → decision_engine
- The decision engine classifies each row as ATTACK or benign and determines whether to block immediately or accumulate hits before action.
- decision_engine → block_hooks + api_dashboard

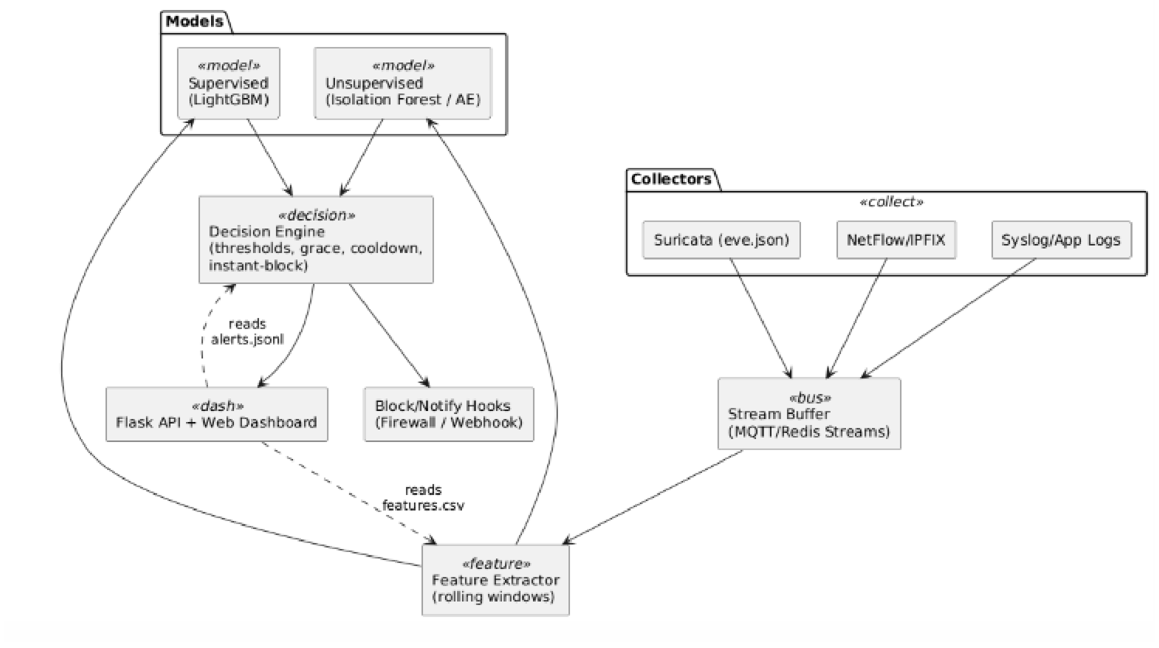
- Blocking actions are executed by OS-level scripts or hooks.
- Simultaneously, decisions are logged to alerts.jsonl for visualization in the dashboard.

2.3 Design Rationale

- Modularity: Each component can evolve independently (e.g., swapping Redis for MQTT or LightGBM for another model).
- Resilience: Failures in one stage (e.g., dashboard crash) do not affect core detection pipeline.
- Scalability: Feature extraction and model inference can be parallelized or containerized as microservices.
- Reproducibility: All inter-package data formats are text-based (CSV, JSONL) for transparency and auditability.
- Portability: Entire system runs on Windows (via WSL) or Linux with minimal modification.

2.4 Component and Package Diagram

Figure 1. IoTGuard - Component/Package Architecture


Models
«model» Supervised (LightGBM)
«model» Unsupervised (Isolation Forest / AE)
«decision» Decision Engine (thresholds, grace, cooldown, instant-block)
reads alerts.jsonl
«dash» Flask API + Web Dashboard
Block/Notify Hooks (Firewall / Webhook)
reads features.csv
«feature» Feature Extractor (rolling windows)
Collectors
«collect»
Suricata (eve.json)
NetFlow/IPFIX
Syslog/App Logs
«bus» Stream Buffer (MQTT/Redis Streams)

2.5 Future Extension Points

Extension	**Description**
Auto-tuning Policy Agent	Future module that automatically adjusts thresholds using feedback (TPR/FPR metrics).
Distributed Stream Pipeline	Replace CSV/JSONL with Kafka or Redis Streams for high-throughput deployments.
Signature Augmentation	Feed Suricata alerts directly into model retraining pipelines to keep up with new threats.
Cross-device Dashboard	Central management console aggregating logs from multiple IoTGuard nodes.

3. Class Interfaces

This section specifies the public interfaces, responsibilities, and contracts of IoTGuard's core components. Interfaces are written in Python-style pseudocode that mirrors the current implementation while allowing future refactors (e.g., moving from file-based CSV/JSONL to stream-based).

3.1 collectors.suricata

Purpose: Convert Suricata eve.json flow lines into normalized in-memory events and push them to the feature pipeline.

3.1.1 Public Data Model

```
@dataclass(frozen=True)

class FlowEvent:

   ts: datetime          # UTC

   src_ip: str
```

```
  dst_ip: str

  bytes_to_srv: int

  bytes_to_cli: int

  pkts_to_srv: int

  pkts_to_cli: int

  flow_state: str       # e.g., "new", "established", ...

  raw: dict             # full original line (for forensics)
```

3.1.2 Public API

```
class EveTail:

  def __init__(self, path: Path, offset_state: Path): ...

  def start(self) -> Iterator [FlowEvent]:

    Yields FlowEvent for each new 'flow' line found.

    - Resumes from last byte offset (offset_state) if present.

    - Ignores non-flow event_type.

    - Tolerates partial/truncated lines.
```

Contracts & Notes:

- Input: newline-delimited JSON (event_type == "flow").
- Resilience: Invalid JSON lines are skipped; parser never raises to caller for content issues.
- Persistence: Saves byte offset in offset_state to survive restarts.

3.2 feature_extractor.window_aggregator

Purpose: Aggregate raw events into rolling time windows and emit one ML-ready feature row per window.

3.2.1 Public Data Model

```
@dataclass(frozen=True)

class FeatureRow:

    flows: int

    bytes_total: int

    pkts_total: int

    uniq_src: int

    uniq_dst: int

    syn_ratio: float

    mean_bytes_flow: float
```

3.2.2 Public API

```
class RollingAggregator:

    def __init__(self, window_seconds: int = 10): ...

    def add(self, e: FlowEvent) -> Optional[FeatureRow]:

        //

        Add a single FlowEvent; returns a FeatureRow when a window boundary

        is crossed (time-based). Otherwise returns None.
    //

  def flush(self) -> Optional[FeatureRow]:

    //Force-flush current buffer into a FeatureRow (if non-empty).//

class FeatureSink:

  def __init__(self, csv_path: Path, schema: list[str]): ...

  def append(self, row: FeatureRow) -> None: ...

  def validate(self, df: pd.DataFrame) -> pd.DataFrame:

    //Coerce numeric dtypes; drop malformed rows; enforce schema order.//
```

Contracts & Notes:

- Schema: Exactly ["flows","bytes_total","pkts_total","uniq_src","uniq_dst","syn_ratio","mean_bytes_flow"].
- Determinism: For the same set of input events and windowing, feature rows are reproducible.

3.3 models.inference (LightGBM + optional unsupervised)

Purpose: Turn feature rows into anomaly probabilities.

3.3.1 Public API

```
class ModelServer:

  def __init__(self, lgbm_path: Path, features: list[str]): ...

  def predict_proba(self, batch: pd.DataFrame) -> np.ndarray:

    //

    Returns float array in [0,1] of shape (N,).

    batch must contain all self.features; raises ValueError otherwise.

    //

class ScoreFusion:

  def __init__(self, weights: dict[str, float] = {"supervised": 1.0}): ...
  def fuse(self, **scores: np.ndarray) -> np.ndarray:

    //

    Weighted average fusion (min-max aligned beforehand).

    Unknown channels ignored. Returns array len == len(any input).

    //
```

Contracts & Notes:

- Calibration: The LightGBM model is probability-calibrated during training (Platt/Isotonic) and saved via joblib.
- Versioning: The model file embeds a training manifest (features list, train date, metrics) in its metadata (recommended).

3.4 decision_engine.loop

Purpose: Convert scores into ATTACK/benign and decide BLOCK/NONE based on policy.

3.4.1 Configuration (hot-reload)

```
(YAML)

decision:

 threshold: 0.60       # classify attack when p >= threshold

 grace: 2              # min ATTACKs in window to allow block

 window: 4             # sliding window length (rows)

 cooldown_sec: 8       # min seconds between blocks

 instant_block: 0.95   # bypass grace+window on very high scores
```

3.4.2 Public API

```
class DecisionLoop:

  def __init__(self, features_csv: Path, alerts_log: Path, cfg_yaml: Path, state_json: Path): ...

  def run(self) -> None:

    //

    Tails features.csv and, for each new row:
      - validate schema & coerce numerics

      - score via ModelServer

      - apply policy (threshold, grace, window, cooldown, instant_block)

      - emit event to alerts.jsonl

      - optionally call hooks on BLOCK

@dataclass

class DecisionEvent:

  ts: float

  index: int

  score: float

  state: Literal["ATTACK","benign"]

  hits_in_window: int

  action: Literal["BLOCK","NONE"]
```

Contracts & Notes:

- Statefulness: Maintains CSV row offset and last_block_idx/time to avoid duplicate or bursty blocks.
- Idempotence: Re-running on the same CSV does not re-emit duplicate BLOCKs for the same row.

3.5 block_hooks (OS/Network actions)

Purpose: Enforce actions (firewall, webhook, SIEM). Pluggable per-OS.

3.5.1 Public API

```
class BlockHook:

  def on_block(self, evt: DecisionEvent) -> None: ...

Concrete Hooks:

class WebhookHook(BlockHook):

  def __init__(self, url: str, timeout: float = 2.0): ...
```

```
    def on_block(self, evt: DecisionEvent) -> None: ...


class WindowsFirewallHook(BlockHook):
    def __init__(self, rule_prefix: str = "IoTGuard", ttl_sec: int = 600): ...
    def on_block(self, evt: DecisionEvent) -> None: ...
    # Internally uses `subprocess.run(["netsh", "advfirewall", ...])`
class IptablesHook(BlockHook):
    def __init__(self, chain: str = "INPUT", ttl_sec: int = 600): ...
    def on_block(self, evt: DecisionEvent) -> None: ...
    # Internally uses `iptables -I` with comment; TTL via `at`/cron cleanup.
```

Contract: Hooks must be non-blocking for the loop (timeouts, exceptions swallowed with warning).

3.6 api_dashboard (Flask + Chart.js)

Purpose: Visualize decisions/metrics and provide a simple control plane.

3.6.1 REST Endpoints (stable)

```
GET /api/latest → {"latest": DecisionEvent, "server_time": now}

GET /api/events?since_ts=<float> → page new events

GET /api/counts?window_minutes=<int> → totals/attacks/blocks

GET /api/config → current decision config

POST /api/config → update decision config (persist YAML)

POST /api/clear → truncate alerts.jsonl

POST /api/clear_all → also reset data/state.json

GET /api/download.csv → export decisions as CSV
```

GET / → SPA dashboard (Chart.js table + line graph)

Contract: API is read-mostly. Config writes require the decision loop to hot-reload or restart.

3.7 Cross-Cutting Concerns

- Schema Validation: FeatureSink.validate() and DecisionLoop's schema gate keep pipeline stable.
- Logging: All decisions are line-delimited JSON in alerts.jsonl (auditable, SIEM-ingestible).
- Fault Tolerance: Readers tolerate partial/truncated lines; writers rotate logs after N MB.

3.8 Class Diagram

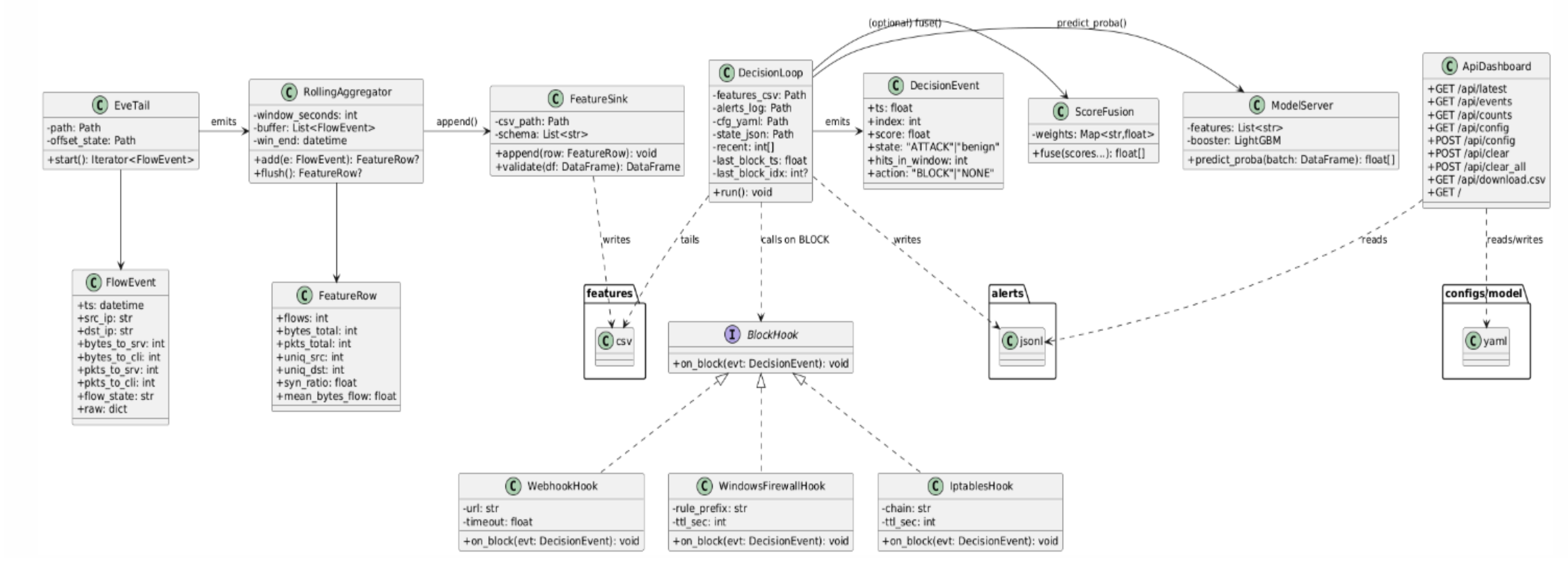


Figure 2. IoTGuard Core Class Interfaces

4. Glossary

Term / Acronym	Definition
IoTGuard	The intelligent Intrusion Detection and Response System designed to monitor IoT network traffic, detect malicious behavior using machine learning, and automatically apply mitigation measures.
IDS (Intrusion Detection System)	A network security tool that monitors and analyzes network packets to identify suspicious activity or known attack patterns.
IPS (Intrusion Prevention System)	An active extension of IDS that not only detects but also blocks or mitigates attacks in real time. IoTGuard can act as a hybrid IDS/IPS depending on configuration.
Suricata	An open-source high-performance network IDS/IPS and security monitoring engine that produces structured JSON logs (eve.json). It forms IoTGuard's primary data source.
eve.json	The standard Suricata JSON output file that contains detailed flow, DNS, HTTP, and TLS event data. It is line-delimited, meaning each line is a complete JSON event.
Flow	A sequence of packets sharing the same 5-tuple (source IP, destination IP, source port, destination port, protocol). Suricata aggregates packet-level information into per-flow records.
Feature Extraction	The process of converting raw flow data into numerical features (e.g., total bytes, packet counts, SYN ratios) suitable for machine learning models.
Rolling Window	A time-based aggregation method where features are computed over small time intervals (e.g., every 10 seconds) to capture dynamic changes in network behavior.

Term / Acronym	Definition
LightGBM	A gradient boosting framework that uses tree-based learning algorithms for high performance and efficiency. It serves as IoTGuard's supervised classifier.
Isolation Forest (IF)	An unsupervised machine learning algorithm that detects anomalies by randomly partitioning data; used in IoTGuard for zero-day or unlabeled scenarios.
Autoencoder (AE)	A neural network architecture trained to reconstruct normal data patterns and identify anomalies based on reconstruction error. Optional in the unsupervised path.
Threshold	The minimum model score (0–1) above which an observation is classified as an "ATTACK." Typically defined in model.yaml.
Grace Value	The number of consecutive detections required within a time window before issuing a block, preventing false positives from single anomalies.
Cooldown Period	A time interval that must pass after a block before a new block can occur, preventing redundant or bursty responses.
Instant Block	A policy that triggers immediate blocking if the score exceeds a critical value (e.g., 0.95), bypassing grace and window checks.
Decision Engine	The policy enforcement module that interprets model predictions, applies thresholds and rules, and generates final actions (BLOCK / NONE).
Alert Log (alerts.jsonl)	Line-delimited JSON log of all detections and blocking actions, used by the dashboard for visualization and auditing.

Term / Acronym	**Definition**
State File (state.json)	Persistent JSON structure maintaining offsets and timestamps to prevent duplicate reads across restarts.
Feature File (features.csv)	The output of the feature extraction pipeline containing numeric features aggregated from Suricata flows. The decision engine tails this file for new rows.
Hook	An external integration (e.g., firewall command, webhook, SIEM) triggered automatically on block events to enforce or report actions.
Webhook	An HTTP callback that sends detection data to an external endpoint (e.g., Slack, SIEM, or threat intelligence service) upon a block.
Dashboard	The web-based interface (Flask + Chart.js) providing live visualization of alerts, metrics, and configuration parameters.
Config File (model.yaml)	YAML file defining runtime parameters such as decision thresholds, window sizes, and model file paths. Supports hot reloading.
JSONL (JSON Lines)	A convenient data format where each line in a text file is a separate JSON object, used for streaming logs in IoTGuard.
CSV (Comma-Separated Values)	Lightweight tabular data format used by IoTGuard to exchange feature data between pipeline stages.
PlantUML	A text-based UML diagram generator used to model system components, classes, and interactions within this report.
IEEE / UML Standards	Standardized notations used for object-oriented design and documentation. UML (Unified Modeling Language) is applied in class and component diagrams.

Term / Acronym	**Definition**
IPv4 / IPv6	Internet Protocol versions 4 and 6, representing addressing formats used by network entities in flow records.
TCP / UDP	Transmission Control Protocol and User Datagram Protocol — common network transport layers represented in flow statistics.
False Positive (FP)	A benign event incorrectly classified as an attack. IoTGuard mitigates this risk through grace periods and cooldowns.
False Negative (FN)	A malicious event not detected by the system; IoTGuard reduces FN risk via multi-model fusion and adaptive thresholds.
Model Fusion	Combining predictions from multiple models (supervised and unsupervised) to achieve more robust detection accuracy.
Anomaly Score	The numeric probability output of the model representing the likelihood that the observed activity is malicious.
Visualization API	The REST interface of the dashboard that serves alerts and metrics to client-side charts and control panels.
System State Persistence	Mechanism for maintaining progress between executions (CSV offsets, timestamps) to ensure real-time continuity.
Hot Reloading	Runtime configuration reload mechanism triggered when model.yaml is updated without restarting the main loop.
Security Event	Any network observation classified by IoTGuard as potentially malicious or requiring further analysis.

.

5. References

Below are all references used or recommended to support the design, methodology, and engineering standards of IoTGuard.

They include textbooks, open-source documentation, and professional standards relevant to object-oriented design, cybersecurity, and machine learning integration.

- Core Design & Software Engineering References:

[1] Bruegge, B., & Dutoit, A. H. (2004). Object-Oriented Software Engineering: Using UML, Patterns, and Java (2nd ed.). Prentice Hall.

Primary textbook reference for object design patterns, UML documentation standards, and interface decomposition techniques applied throughout the system.

[2] IEEE Computer Society. (1990). IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990).

Defines key engineering terms used in the Glossary and overall report structure.

[3] Association for Computing Machinery (ACM). (2018). ACM Code of Ethics and Professional Conduct.

Guides ethical considerations in AI-driven security decisions and user-data handling.

Cybersecurity & Intrusion Detection References

[4] Open Information Security Foundation (OISF). (2024). Suricata User Guide v7.0.

Retrieved from https://docs.suricata.io/

Official Suricata documentation detailing eve.json structure, flow event fields, and output configuration used in IoTGuard's collectors.

[5] Scarfone, K., & Mell, P. (2007). Guide to Intrusion Detection and Prevention Systems (IDPS). NIST Special Publication 800-94.

Reference for IDS/IPS design taxonomy and classification methods.

[6] CrowdSec. (2023). CrowdSec Open-Source IPS Documentation.
https://doc.crowdsec.net/

Used as conceptual inspiration for IoTGuard's crowd-based blocking and alert propagation mechanisms.

[7] Cisco Systems. (2023). NetFlow Version 9 Protocol Specification.
https://www.cisco.com/

Referenced for flow aggregation concepts reused in feature extraction.

Machine Learning References

[8] Ke, G., Meng, Q., Finley, T., et al. (2017). LightGBM: A Highly Efficient Gradient Boosting Decision Tree. Advances in Neural Information Processing Systems (NIPS 2017).

Describes the algorithmic basis for IoTGuard's supervised anomaly-detection model.

[9] Liu, F. T., Ting, K. M., & Zhou, Z.-H. (2008). Isolation Forest. IEEE International Conference on Data Mining (ICDM 2008).

Cited for unsupervised anomaly detection design in optional model fusion.

[10] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.

Conceptual foundation for potential Autoencoder-based extensions of IoTGuard.

Data Engineering & Visualization References

[11] McKinney, W. (2010). Data Structures for Statistical Computing in Python. Proceedings of the 9th Python in Science Conference (SciPy 2010), 51–56.

Defines the Pandas data structures used in feature aggregation.

[12] Flask Documentation (2024). Flask User Guide. https://flask.palletsprojects.com/

Framework reference for IoTGuard's web dashboard and REST API implementation.

[13] Chart.js Developers. (2025). Chart.js Documentation. https://www.chartjs.org/docs/

Used for rendering live dashboards from alert logs.

[14] Redis Labs. (2024). Redis Streams Introduction. https://redis.io/docs/data-types/streams/

Referenced for future distributed-streaming support.

System & OS Integration References

[15] Microsoft Docs. (2024). Windows Firewall with Advanced Security Cmdlets. https://learn.microsoft.com/

Used for Windows blocking hook design.

[16] Iptables Project. (2024). iptables Manual. https://man7.org/linux/man-pages/man8/iptables.8.html

Used for Linux blocking hook implementation details.

[17] Open Networking Foundation (ONF). (2023). Software-Defined Networking (ONF White Paper).

Cited for conceptual SDN integration in future IoTGuard versions.

Additional Research and Reference Datasets

[18] Kaggle Inc. (2024). IoT Network Intrusion Dataset (ISCX / CICIDS / TON_IoT Collections).

https://www.kaggle.com/datasets/

Datasets used during IoTGuard's model training phase.

[19] Moustafa, N., & Slay, J. (2016). UNSW-NB15: A Comprehensive Data Set for Network Intrusion Detection Systems. IEEE Military Communications Conference (MILCOM 2015).

Served as part of the supervised training set for the LightGBM model.

Citation Note:

- All citations comply with IEEE format for technical documentation.
- Direct web links have been included for open-source frameworks to support verification and reproducibility.
- Academic references [8] and [9] anchor the machine-learning approach, while engineering standards [1] and [2] validate the documentation methodology.