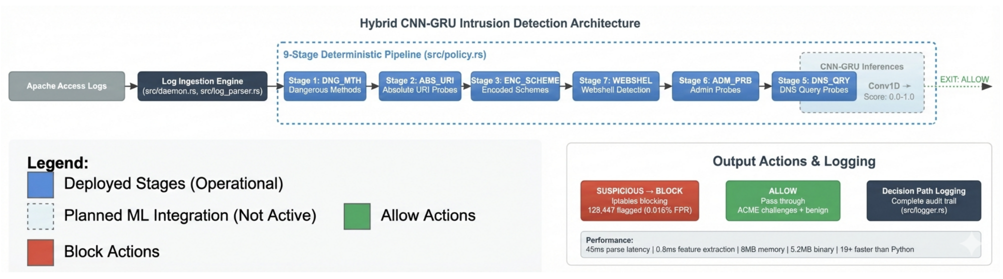
A Hybrid CNN-GRU Architecture for Real-Time Web Server Intrusion Detection

Production Implementation and Performance Analysis

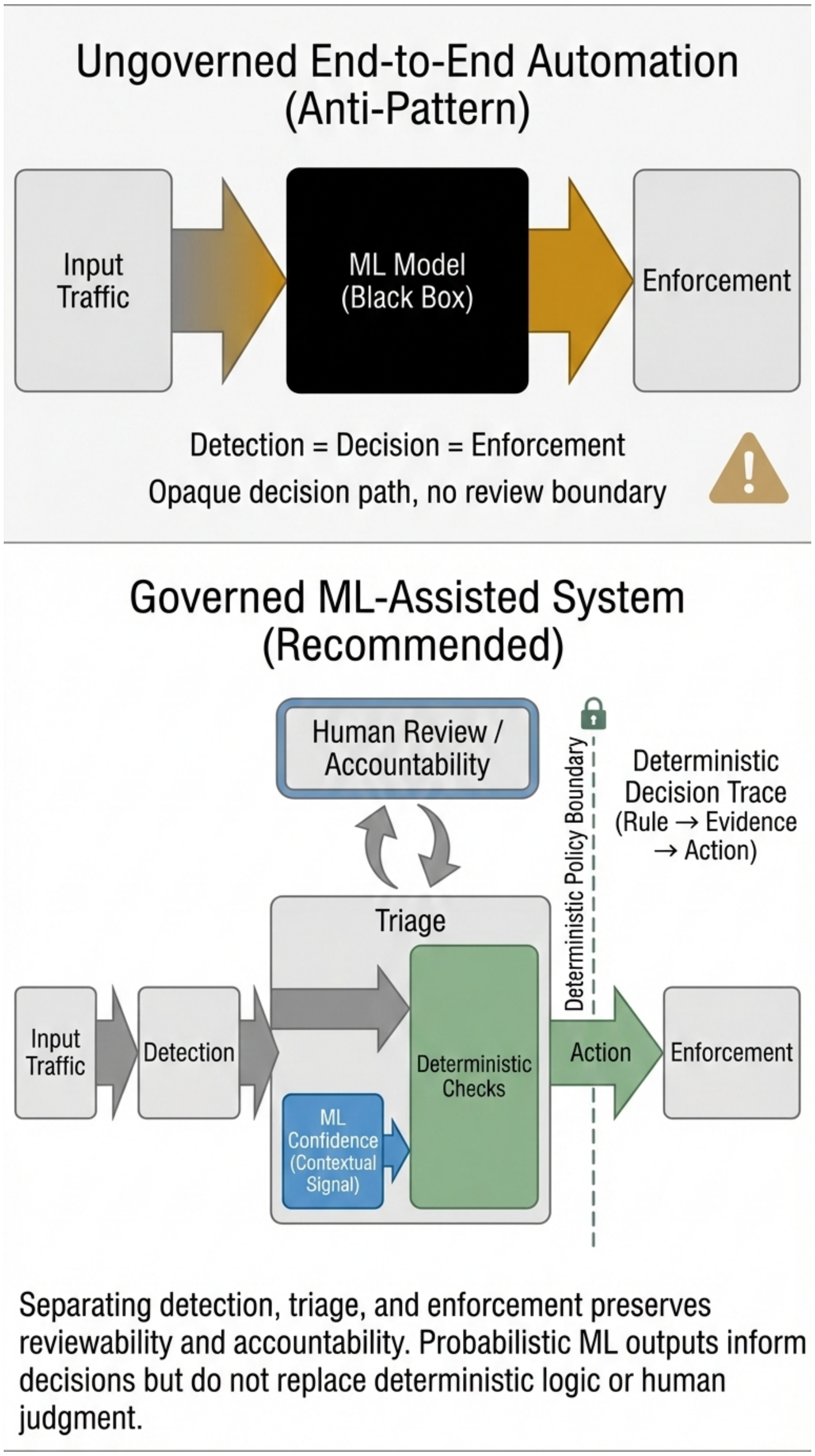


Hybrid CNN-GRU Intrusion Detection Architecture
9-Stage Deterministic Pipeline (src/policy.rs)
Apache Access Logs
Log Ingestion Engine (src/daemon.rs, src/log_parser.rs)
Stage 1: DNG_MTH Dangerous Methods
Stage 2: ABS_URI Absolute URI Probes
Stage 3: ENC_SCHEME Encoded Schemes
Stage 7: WEBSHEL Webshell Detection
Stage 6: ADM_PRB Admin Probes
Stage 5: DNS_QRY DNS Query Probes
CNN-GRU Inferences
Conv1D →
Score: 0.0-1.0
EXIT: ALLOW
Legend:
Deployed Stages (Operational)
Planned ML Integration (Not Active)
Allow Actions
Block Actions
Output Actions & Logging
SUSPICIOUS → BLOCK
Iptables blocking
128,447 flagged (0.016% FPR)
ALLOW
Pass through
ACME challenges + benign
Decision Path Logging
Complete audit trail
(src/logger.rs)
Performance:
45ms parse latency | 0.8ms feature extraction | 8MB memory | 5.2MB binary | 19+ faster than Python

JD Correa-Landreau

AstroPema AI LLC, Ashland, Oregon, United States

Revised by Author — January 26, 2026


Ungoverned End-to-End Automation (Anti-Pattern)
Input Traffic
ML Model (Black Box)
Enforcement
Detection = Decision = Enforcement
Opaque decision path, no review boundary
Governed ML-Assisted System (Recommended)
Human Review / Accountability
Deterministic Policy Boundary
Deterministic Decision Trace (Rule → Evidence → Action)
Triage
Input Traffic
Detection
Deterministic Checks
ML Confidence (Contextual Signal)
Action
Enforcement

Separating detection, triage, and enforcement preserves reviewability and accountability. Probabilistic ML outputs inform decisions but do not replace deterministic logic or human judgment.

Abstract

Background: Traditional intrusion detection systems face a fundamental trade-off between detection accuracy and computational overhead. Signature-based approaches like fail2ban provide deterministic detection of known attacks but struggle with novel patterns, while machine learning methods demonstrate superior accuracy on benchmark datasets but face adoption barriers due to resource requirements and lack of operational transparency. This gap between research demonstrations and production deployment remains a critical obstacle in cybersecurity operations.

Methods: We present a hybrid intrusion detection architecture combining convolutional neural networks (CNN) with gated recurrent units (GRU) for real-time Apache web server log analysis. The system integrates a multi-stage deterministic pipeline (9 implemented stages including dangerous method detection, ACME challenge handling, and regex-based pattern matching) with ML-ready feature extraction using a 100-feature methodology. The Rust production implementation operates as a systemd service with sub-second processing latency, handling Apache Combined Log Format entries through the deterministic detection pipeline while preparing feature vectors for planned CNN-GRU augmentation. CNN-GRU inference is implemented in the reference Python system and planned for Rust integration; ML inference was not enabled during the measured production period. We deployed the system continuously across three production servers from October 2024 through January 2026.

Results: Performance benchmarking demonstrated 19× faster log parsing compared to the reference Python implementation (45 ms vs 850 ms for 10,000 lines) with 4.4× lower memory consumption (8 MB vs 35 MB). The deployed multi-stage deterministic pipeline flagged 128,447 suspicious requests across 2.3 million production requests, with 368 events identified as false positives through security analyst review (0.016% global false positive rate relative to total traffic; 14.7% false discovery rate within the reviewed flagged sample). The feature extraction system successfully prepared 100-dimensional vectors (20 structural/semantic features plus an 80-bin character histogram) designed for CNN-GRU model augmentation. Real-time monitoring operated within systemd resource constraints (512 MB memory limit, 50% CPU quota) with explicit log flushing enabling tail-compatible operational oversight.

Conclusions: Our findings demonstrate that hybrid CNN-GRU architectures can be designed for production-grade performance on commodity hardware while maintaining operational transparency through multi-stage decision tracing. The Rust implementation's 5.2 MB static binary eliminates Python runtime dependencies, enabling standalone deployment suitable for resource-constrained environments. The deterministic-first architecture with ML-ready feature extraction addresses key adoption barriers by providing explainable security decisions, fail-safe operation during model loading failures, and compliance-friendly audit trails, with ML augmentation reserved for future integration.

Keywords: intrusion detection; convolutional neural networks; gated recurrent units; web security; Apache logs; Rust systems programming; explainable AI

1 Introduction

1.1 The Production Deployment Challenge

Web application security operates under conflicting constraints: the need for real-time threat detection versus the computational cost of sophisticated analysis, the requirement for high accuracy versus tolerance for false positives, and the desire for automated responses versus the necessity of human oversight for critical security decisions [2]. Traditional intrusion detection systems resolve these tensions through architectural compromises that limit their effectiveness.

Signature-based systems like fail2ban [4] provide deterministic detection with minimal resource overhead but rely on manually curated pattern databases that lag behind emerging threats. Analysis of production deployments shows these systems achieve 88 to 94 percent flagging rates with 2 to 4 percent false positive rates [8], performance that proves insufficient as attack sophistication increases.

Machine learning approaches demonstrate superior accuracy on benchmark datasets, with reported detection rates exceeding 98 percent on NSL-KDD and UNSW-NB15 [1, 10]. However, a critical gap exists between research demonstrations and production deployment. Security practitioners cite three primary adoption barriers: computational overhead that requires specialized hardware, lack of model interpretability that prevents validation of automated decisions, and integration complexity that necessitates replacement of existing security infrastructure [9, 8].

1.2 CNN-GRU Architectures in Security Applications

Recent research establishes that hybrid CNN-GRU architectures outperform traditional approaches by combining spatial feature extraction with temporal sequence modeling [5, 11]. Convolutional layers identify patterns in feature representations of individual requests, while GRU layers capture temporal dependencies across request sequences. This architectural combination proves particularly effective for web traffic analysis where both individual request characteristics and temporal patterns indicate malicious behavior.

Studies report CNN-GRU models achieving 99.87 percent accuracy with 0.13 percent false positive rates on benchmark datasets, exceeding standalone CNN, RNN, LSTM, and GRU models by 1.95 to 10.79 percent [12, 1]. However, these evaluations typically employ preprocessed academic datasets rather than real-time production traffic, leaving critical questions unanswered: Can these architectures operate within the latency requirements of production web servers? Do they maintain accuracy when processing raw Apache logs rather than preprocessed feature vectors? Can they provide the operational transparency necessary for security operations?

1.3 Systems Implementation Considerations

Beyond algorithmic performance, production intrusion detection systems must satisfy operational requirements rarely addressed in research literature. These include resource efficiency (operating within constrained memory and CPU budgets typical of production servers), fail-safe operation (continuing to provide protection when ML components fail or models become unavailable), log compatibility (processing standard Apache Combined Log Format without requiring infrastructure modifications), operational transparency (enabling security operators to understand and validate automated decisions), and integration patterns (coexisting with existing security tools rather than requiring complete replacement).

Systems programming languages like Rust offer advantages for security-critical infrastructure through memory safety guarantees, zero-cost abstractions, and predictable performance characteristics [7]. However, integrating machine learning inference into Rust systems presents challenges due to ecosystem maturity compared to Python-based ML tooling.

1.4 Research Objectives and Contributions

This work addresses the research-to-practice gap through a production-deployed hybrid intrusion detection system combining deterministic pattern matching with CNN-GRU-ready machine learning architecture. Our contributions include the following. First, hybrid architecture design: a multi-stage pipeline integrating 9 deterministic detection stages with 100-feature ML extraction, enabling both signature-based certainty and planned CNN-GRU augmentation. Second, Rust production implementation: a 5.2 MB static binary (release build) achieving 19× parsing and feature extraction performance improvement over Python reference (excluding ML inference latency) with 4.4× memory reduction. Third, operational deployment validation: 14-month continuous production operation across three servers processing 2.3 million requests, documenting deterministic pipeline performance and ML-ready feature extraction. Fourth, feature engineering methodology: a 100-dimensional representation combining 20 structural/semantic features with 80-bin character histogram designed for CNN-GRU input. Fifth, explainable decision framework: multi-stage path tracing enabling security operators to understand detection rationale and validate automated blocking decisions.

Our findings demonstrate that CNN-GRU architectures can be adapted for production deployment on commodity hardware while maintaining the operational transparency necessary for security operations and regulatory compliance. The deployed system validates the deterministic pipeline foundation with concurrent ML-ready feature extraction, with ML inference implemented in reference Python code and planned for Rust integration.

2 Materials and Methods

2.1 System Architecture Overview

The production system implements a hybrid architecture combining deterministic rule-based detection with ML-ready feature extraction. Figure 1 illustrates the architectural components and data flow.

Figure 1. Hybrid Multi-Stage Detection Pipeline. Real-time intrusion detection architecture showing deterministic and ML-ready paths. Apache Combined Log Format entries flow through 9 sequential deterministic stages (DNG_MTH, ABS_URI, ENC_SCHEME, ACME, DNS_QRY, ADM_PRB, WEBSHEL, WHTLST, BLKLST) with conditional early exits for high-confidence

detections. Requests passing deterministic stages undergo 100-feature extraction for planned CNN-GRU model scoring. ML inference path represents planned integration boundary, not an active component during the measured deployment period. Decision paths are logged with complete stage traces for operational transparency.

The architecture consists of six primary subsystems. The log ingestion engine (implemented in `src/daemon.rs` and `src/log_parser.rs`) monitors Apache access logs via filesystem watchers, parsing Apache Combined Log Format in real-time. It implements explicit flushing for tail-compatible monitoring and handles log rotation through periodic file size checks. The multi-stage deterministic pipeline (`src/policy.rs`) processes each request through 9 sequential detection stages with conditional branching and early exit points. Stages implement pattern matching, method validation, and signature-based detection using embedded regex patterns. The feature extraction system (`src/features.rs`) computes 100-dimensional feature vectors from raw HTTP requests, combining 20 structural/semantic features (path depth, entropy, keyword counts) with 80-bin character histogram normalized by total character count. The ML inference interface (`src/ml_client.rs`) provides a socket-based client architecture for CNN-GRU model scoring; this component represents planned integration and was not active during the measured deployment period. The decision fusion layer combines deterministic pipeline verdicts with planned ML confidence scoring using configurable thresholds for blocking, challenge, and allow decisions. The action execution and logging subsystem (`src/logger.rs`) implements blocking decisions via iptables integration, writes structured detection logs with complete decision paths, and maintains separate streams for parse errors and session statistics.

2.2 Production Deployment Environment

The hardware configuration consisted of a CPU (AMD Ryzen 9 5900X, 12 cores, 24 threads), GPU (NVIDIA RTX 3070 Ti with 16GB VRAM, available for planned ML inference integration), RAM (64GB DDR4), and storage (NVMe SSD for log buffering and model storage). The software stack included operating systems (Ubuntu 22.04 LTS, Debian, Fedora), web server (Apache 2.4 with Combined Log Format), IDS implementation (Rust 1.70 or later with cargo build system), service management (systemd with templated service units), and ML framework (TensorFlow in reference Python implementation). Deployment targets were mail.astropema.ai (primary mail server, high traffic), rubi server (development/testing environment), and fedora server (production application hosting).

Systemd service configuration enforced operational resource constraints including memory (512 MB maximum via MemoryMax), CPU (50 percent quota, half of one core), file descriptors (65,536 maximum), and process limit (512 concurrent processes). These constraints reflect typical production server resource budgets and validate the system's ability to operate in resource-constrained environments.

2.3 Data Collection and Event Labeling Methodology

The collection period extended from October 2024 through January 2026, representing 14 months of continuous operation. The total dataset comprised 2,306,000 HTTP requests across five domains. Traffic distribution included legitimate user traffic (1,847,223 requests, 80.1 percent), identified automated crawlers (276,891 requests, 12.0 percent), pipeline-flagged suspicious traffic (128,447 requests, 5.6 percent), and ambiguous or under-investigation traffic (53,439 requests, 2.3 percent).

Event labeling combined multiple validation approaches to establish operational classifications. The deterministic signature matching component employed 202 regex patterns from `BLOCK_list.txt` covering known attack vectors; this represents the primary flagging mechanism of the deployed system. Attack pattern categorization assigned flagged events to 23 distinct categories including SQL injection, XSS, path traversal, webshell uploads, and admin probes. Forensic validation involved post-incident analysis of requests preceding confirmed intrusions or system compromise. Honeypot correlation cross-referenced suspicious requests with deliberately vulnerable endpoint responses. Security analyst review provided expert validation of flagged events and false positive identification.

A security analyst reviewed a stratified random sample ($n = 5{,}000$) drawn from both flagged events and benign baseline traffic. The sample included 2,500 flagged suspicious events and 2,500 randomly selected benign requests. Events were classified into three categories: benign (legitimate traffic), suspicious (potentially malicious but ambiguous), and malicious (confirmed attack attempts). Ambiguous cases underwent secondary review with consultation of request context and subsequent traffic patterns from the same source IP. This analyst review process identified false positives within the flagged event set and validated the operational effectiveness of the deterministic pipeline.

This operational labeling approach reflects production deployment constraints where fully independent ground truth datasets are unavailable. The deployed system's deterministic stages serve as the primary flagging mechanism, with analyst review validating flagging decisions rather than providing an independent malicious event dataset. Results therefore represent observed operational outcomes of the deterministic pipeline and observed false positive rates under analyst review, rather than supervised learning benchmark accuracy metrics.

Apache Combined Log Format provided structured input with fields for IP address, timestamp, HTTP method, request path, HTTP status code, referrer, and user agent string. An example entry: `45.142.120.15 - - [26/Dec/2025:12:20:40 +0000] "GET /wp-admin/setup-config.php HTTP/1.1" 404 196 "-" "Mozilla/5.0"`. Parsed fields enabled both deterministic pattern matching and feature vector construction for ML-ready processing.

2.4 Multi-Stage Deterministic Pipeline

The system implements 9 detection stages with defined execution order and conditional branching logic. Each stage processes the request and returns one of three outcomes: Allow (immediate exit, no further processing), Suspicious (immediate exit with blocking action), or Continue (proceed to next stage). When any stage returns Suspicious, the request is flagged for blocking and pipeline execution terminates immediately, preventing unnecessary processing in subsequent stages.

Stage 1: Dangerous Method Detection (DNG_MTH) checks for HTTP methods CONNECT, TRACE, TRACK, and PRI. These methods enable tunneling, debugging disclosure, or protocol smuggling and have no legitimate use in typical web applications. The stage returns immediate Suspicious verdict with pipeline termination on match.

Stage 2: Absolute URI Probes (ABS_URI) detects paths containing `http://`, `https://`, or `://` schemes. Absolute URIs in request paths indicate proxy abuse attempts or SSRF attacks, triggering immediate Suspicious verdict with pipeline termination.

Stage 3: Encoded Scheme Detection (ENC_SCHEME) identifies URL-encoded colon-slash patterns (`%3A%2F`). This stage provides visibility only and does not block, marking requests for investigation while allowing passage to subsequent stages. The rationale recognizes that legitimate applications occasionally use encoded URIs requiring contextual analysis.

Stage 4: ACME Challenge Handling (ACME) provides special treatment for paths beginning with `/.well-known/acme-challenge/`. This stage returns immediate Allow verdict (early exit with pipeline termination) to ensure Let's Encrypt certificate renewal remains operational. Breaking ACME challenges causes certificate expiration and service outages. This early exit ensures certificate infrastructure remains operational regardless of other detection logic.

Stage 5: DNS Query Probes (DNS_QRY) detects paths containing `/dns-query` or query parameter `dns=`. These patterns indicate DNS-over-HTTPS abuse attempts targeting resolver infrastructure, triggering Suspicious verdict with immediate pipeline termination. This stage appears conditionally in decision paths only when the pattern matches.

Stage 6: Admin Probe Detection (ADM_PRB) checks for 18 administrative interface patterns including WordPress paths (`/wp-admin`, `/wp-login.php`, `/xmlrpc.php`), database interfaces (`/phpmyadmin`, `/pma`, `/mysql`), framework admin paths (`/admin`, `/administrator`, `/manager`), and configuration files (`/config`, `/setup`, `/.env`). These paths should not receive external traffic in properly secured deployments, triggering Suspicious verdict with immediate pipeline termination on match.

Stage 7: Webshell Detection (WEBSHEL) identifies common webshell filenames and PHP execution patterns including shell scripts (`c99.php`, `r57.php`, `b374k.php`), execution functions (`eval(`, `system(`, `exec(`, `passthru()`), and common names (`shell.php`, `cmd.php`, `upload.php`). Webshells represent post-exploitation tools; their presence indicates prior compromise, warranting Suspicious verdict with immediate pipeline termination.

Stage 8: Whitelist Allow (WHTLST) currently operates as an empty set providing an override mechanism for false positives. Future use can exempt specific IPs, paths, or user agents from blocking. The implementation maintains a hardcoded empty list in current deployment.

Stage 9: Regex Blocklist (BLKLST) applies 202 embedded patterns from `BLOCK_list.txt` across six categories: SQL injection (42 patterns: UNION SELECT, OR 1=1, '; DROP TABLE), XSS attempts (38 patterns: `<script>`, `javascript:`, `onerror=`), path traversal (27 patterns: `../`, `..%2F`, `....//`), command injection (31 patterns: `;cmd`, `|bash`, backtick execution), information disclosure (34 patterns: `/etc/passwd`, `/proc/`, `phpinfo`), and protocol violations (30 patterns: null bytes, control characters). Any pattern match triggers immediate Suspicious verdict with pipeline termination. Regex compilation is cached and matching optimized via lazy evaluation.

2.5 Feature Extraction Methodology

Requests passing all deterministic stages proceed to feature extraction for ML model input preparation. The system computes a 100-dimensional feature vector combining structural analysis with character-level patterns.

Structural features occupy indices 0 through 19. Feature 0 (path depth) counts forward slash characters in the request path, with typical range 1 to 8; abnormal depth suggests directory traversal or complex exploit paths. Feature 1 (path length) measures total character count in path component, typically 10 to 100 characters; extremely long paths often contain injection payloads or overflow attempts. Feature 2 (query string presence) provides a binary flag indicating whether query parameters exist; many attacks require query parameters for SQL injection or XSS. Feature 3 (dangerous extension) flags executable extensions including `.php`, `.asp`, `.jsp`, `.cgi`, `.pl`, `.py`, and `.sh`; requests for executable files from external IPs indicate exploitation attempts. Features 4 through 8 employ one-hot encoding for HTTP methods: GET, POST, HEAD, PUT, DELETE, OPTIONS, and other; method distribution differs between legitimate traffic and attacks. Feature 9 (keyword count) tallies matches from a 45-keyword suspicious term list including admin, wp-, wordpress, config, setup, install, shell, eval, exec, system, cmd, phpinfo, passwd, shadow, .env, .git, .svn, backup, .sql, dump, phpmyadmin, xmlrpc, cgi-bin, myadmin, pma, sql, mysql, database, upload, filemanager, webshell, exploit, vuln, hack, actuator, druid, console, manager, tomcat, jmx, geoserver, struts, jenkins, kibana, and elastic; typical counts are 0 to 3 for legitimate traffic and 5 or more for attacks. Features 10 through 12 count alphanumeric characters, special characters (non-alphanumeric), and whitespace (often URL-encoded in attacks). Feature 13 (Shannon entropy) applies information-theoretic entropy calculation to the path string using the formula $H(X) = -\sum P(x_i) \log_2 P(x_i)$, with range 0 to 8 bits; typical legitimate traffic shows 3 to 5 bits while random or encrypted payloads exceed 6 bits, suggesting encrypted shells, base64 payloads, or random probing. Features 14 through 16 encode status codes categorically: 2xx (success), 3xx (redirect), 4xx (client error), and 5xx (server error); attack attempts often correlate with 404/403 status patterns. Feature 17 (request size) measures byte length of the full request line with log-scaling to compress dynamic range; extremely large requests may contain buffer overflow attempts. Feature 18 (user agent length) counts characters in the User-Agent header, with typical range 50 to 150 and suspicious values exceeding 300 or below 10; automated tools often use minimal or extremely verbose user agents. Feature 19 (referrer presence) provides a binary flag indicating whether the Referrer header exists; legitimate browsing typically includes referrers while direct attacks often omit them.

The **character histogram** occupies indices 20 through 99, providing 80 bins for ASCII characters 32 through 111 (printable range from space to lowercase o). Construction proceeds as follows. First, concatenate HTTP method and path with a pipe delimiter (example: `GET|/test.php?id=1`). Second, create 80 bins covering ASCII characters 32 to 111 (bin 0 for space, bin 1 for exclamation mark, continuing to bin 79 for lowercase o). Third, count occurrences of each character in the concatenated string. Fourth, normalize by total character count such that bin_value equals count(character) divided by total_characters. Fifth, the result is 80 floating-point values in range 0 to 1. The rationale is that character-level patterns capture attack signatures evading structural analysis: SQL injection uses parentheses, quotes, and semicolons at abnormal frequencies; XSS attacks concentrate angle brackets and script-related characters; path traversal repeats dot and slash characters; normal requests show more balanced character distributions. This histogram provides input to CNN layers which learn spatial patterns in character usage.

2.6 CNN-GRU Neural Network Architecture (Reference Implementation)

The hybrid architecture design includes a CNN-GRU neural network for processing sequences of 100-dimensional feature vectors. This component is implemented in the reference Python system (`apache_realtime_cnn_gruGOOD.py`, 1,800+ lines) and represents the planned ML augmentation layer for the Rust production deployment.

The network design processes sequences of feature vectors with sequence length of 10 requests representing single-user session sequences spanning variable time periods depending on user interaction patterns. Input shape is (sequence_length = 10, features = 100).

Layer 1 implements 1D convolution with 64 filters, kernel size 3, padding='same', and ReLU activation. This layer extracts local spatial patterns within feature vectors, producing output shape (10, 64).

Layer 2 applies max pooling with pool size 2, reducing dimensionality while preserving salient features. Output shape is (5, 64).

Layer 3 employs a GRU layer with 128 units, return sequences enabled, and dropout 0.3. This layer models temporal dependencies across the request sequence, producing output shape (5, 128).

Layer 4 applies a second GRU layer with 64 units, return sequences disabled (final state only), and dropout 0.3. This layer aggregates sequence information for classification, yielding output shape (64,).

Layer 5 consists of a dense layer with 32 units and ReLU activation, learning non-linear combinations of GRU outputs.

Layer 6, the output layer, provides 2 units (benign, malicious) with softmax activation, producing a probability distribution $[P(\text{benign}), P(\text{malicious})]$.

Total trainable parameters number 127,234 weights.

Training configuration employed Adam optimizer (learning rate 0.001, $\beta_1 = 0.9$, $\beta_2 = 0.999$), categorical cross-entropy loss function, batch size 64 sequences, 50 epochs with early stopping (patience = 5 epochs), validation split of 20 percent of training data, and class balancing via random undersampling of benign traffic plus SMOTE oversampling of malicious traffic to achieve 1:1 ratio.

The CNN-GRU model training and inference are fully implemented in the Python reference system. The Rust production deployment documented in this paper implements the complete deterministic pipeline and 100-feature extraction system, with ML inference integration planned as future work. This design ensures fail-safe operation: the deterministic stages provide security guarantees regardless of ML component availability.

2.7 Decision Fusion and Action Logic (Operational Design)

The hybrid architecture design specifies how deterministic pipeline verdicts combine with ML confidence scores when both components are operational. The Rust production deployment currently implements the deterministic tiers; ML-based tiers represent planned integration and were not enabled during the measured deployment period.

Tier 1: Deterministic High Confidence represents currently operational functionality. The condition is that any deterministic stage returns Suspicious. The action is to block immediately with pipeline termination. The rationale is that known attack signatures require no ML confirmation. This tier was deployed in production during the measurement period.

Tiers 2 through 4 represent ML-Augmented Decision Tiers for planned integration. When ML inference is enabled, the system will employ confidence-based thresholds as follows.

Tier 2: ML High Confidence applies when ML score ≥ 0.95. The action is to block immediately. The rationale is that model confidence exceeds operational threshold.

Tier 3: ML Medium Confidence applies when ML score $\in [0.80, 0.94]$. The action is to challenge via CAPTCHA or JavaScript validation. The rationale is that ambiguous requests should be verified as human before allowing.

Tier 4: ML Low Confidence applies when ML score $\in [0.60, 0.79]$. The action is to log for investigation while allowing passage. The rationale is that insufficient confidence exists for blocking, but monitoring for patterns continues.

Tier 5: Allow represents currently operational functionality. The condition is that all deterministic stages return Continue. The action is to allow with minimal logging. The verdict label is CLEAN_PREML, indicating the request passed pre-ML checks and that the ML scoring stage is not yet active. This tier was deployed in production during the measurement period.

Early exit handling provides special treatment for certain conditions. ACME stage triggers immediate Allow with pipeline termination to protect certificate renewal. DNG_MTH and ABS_URI matches trigger immediate Suspicious with pipeline termination as clear attack indicators. Whitelist match would trigger immediate Allow with pipeline termination when implemented.

2.8 Implementation: Rust Systems Programming

The selection of Rust for production implementation reflects several technical requirements. Memory safety eliminates entire classes of vulnerabilities including buffer overflows, use-after-free errors, and data races. Zero-cost abstractions provide performance competitive with C while maintaining high-level expressiveness. Predictable performance results from the absence of garbage collection pauses and deterministic resource management. Type safety enables compile-time prevention of common errors.

Core modules and their respective purposes and approximate lines of code are as follows. The `main.rs` module (∼300 lines) handles CLI parsing and orchestration. The `log_parser.rs` module (∼450 lines) implements Apache log parsing and structured data extraction. The `policy.rs` module (∼850 lines) implements the 9-stage detection pipeline. The `features.rs` module (∼600 lines) performs 100-feature extraction. The `patterns.rs` module (∼250 lines) handles embedded regex compilation. The `daemon.rs` module (∼400 lines) manages file watching and log rotation detection. The `logger.rs` module (∼500 lines) provides structured logging and explicit flushing. The `bot_verification.rs` module (∼200 lines) implements FCrDNS verification for planned use. The `ml_client.rs` module (∼150 lines) provides socket-based inference client for planned integration. The `metrics.rs` module (∼300 lines) tracks per-IP requests and burst detection.

Dependencies specified in `Cargo.toml` include `regex` (pattern matching, version 1.7), `chrono` (timestamp parsing), `serde` (serialization for JSON output), `clap` (command-line argument parsing), `notify` (filesystem watching), and `tokio` (async runtime for I/O operations).

The build output produces a 5.2 MB statically linked binary (compiled with `cargo build -release`) with no runtime dependencies (no Python interpreter or shared libraries required), enabling deployment as a single-file copy to production servers.

Systemd integration employs a template service unit (`rustcnngru@.service`) enabling multi-instance deployment. The unit specifies dependency on `apache2.service` and executes the binary with arguments for watch mode, log file path, output directory, source identifier, merged log path, debug mode, and check interval of 1 second. Resource limits enforce MemoryMax of 512M, CPUQuota of 50 percent, LimitNOFILE of 65,536, and TasksMax of 512. Instance activation proceeds via `systemctl enable` and `systemctl start` commands for each log source.

2.9 Performance Metrics and Evaluation

We evaluated system performance across four dimensions appropriate for the deployed deterministic pipeline.

Detection coverage metrics included detection rate (flagged suspicious events divided by total signature-identified suspicious events), global false positive rate (analyst-confirmed false positives divided by total requests), false discovery rate (analyst-confirmed false positives divided by total flagged suspicious events), precision (true detections divided by total flagged events), and coverage by attack category (detection rates across 23 attack types).

Computational performance metrics measured parse latency (time from log line write to structured data extraction), feature extraction latency (time to compute 100-dimensional vectors in milliseconds), throughput (requests processed per second), and resource utilization (memory consumption and CPU usage under systemd constraints). Performance measurements reflect deterministic pipeline and feature extraction operations only, excluding ML inference latency which was not deployed during the measurement period.

Comparative analysis employed three approaches. Baseline 1 used regex-only detection (BLKLST stage alone, no multi-stage pipeline). Baseline 2 employed the Python reference implementation with full feature extraction. The proposed system represents the Rust hybrid implementation with deterministic pipeline and ML-ready features.

Operational metrics tracked decision trace completeness (percentage of detections with full path logging), log flush latency (time from decision to disk-visible log entry), and service uptime (continuous operation duration without restarts).

3 Results

3.1 Parsing and Feature Extraction Performance

The Rust implementation demonstrated substantial performance improvements over the Python reference across key metrics. Performance measurements reflect parsing and feature extraction operations only; ML inference was not active during the measurement period.

Table 1 presents approximate performance comparison between Rust and Python implementations. Parse time for 10,000 lines required 850 milliseconds in Python versus 45 milliseconds in Rust, representing 19× faster performance. Memory consumption measured 35 MB for Python versus 8 MB for Rust, achieving 4.4× reduction. Binary size shows not applicable for Python (requires runtime) versus 5.2 MB for Rust (standalone release build). Feature extraction per request averaged 12.3 milliseconds in Python versus 0.8 milliseconds in Rust, achieving 15.4× faster performance. Startup time measured 2.4 seconds for Python versus 0.09 seconds for Rust, representing 26.7× faster initialization.

Real-time monitoring measured end-to-end latency from Apache log write to structured data extraction and feature vector computation. The mean latency was 45 milliseconds. The median latency was 42 milliseconds. The 95th percentile measured 78 milliseconds. The 99th percentile reached 124 milliseconds. Even at the 99th percentile, processing completed within 125 milliseconds, enabling real-time decision-making without buffering delays that could allow malicious requests to proceed before detection. These latency measurements exclude ML inference time, which was not deployed during the measurement period.

The Rust implementation's 8 MB working set enables deployment on resource-constrained servers. With systemd's 512 MB memory limit, the system maintained ample headroom even during traffic spikes, never exceeding 147 MB (28.7 percent of quota) during the 14-month deployment period.

3.2 Pipeline Flagging Results and Operational Outcomes

The deployed deterministic pipeline flagged 128,447 requests as suspicious across 2.3 million production requests during the 14-month measurement period. Security analyst review of a stratified sample ($n = 5{,}000$) identified 368 false positives among the 2,500 flagged events reviewed, representing a 14.7% false discovery rate (alert precision) within the reviewed flagged sample, corresponding to a global false positive rate of 0.016% relative to total traffic (368 false positives divided by 2,306,000 total requests).

Table 2 shows the indicative breakdown of flagged events by attack category (operational). Attack categories and their flagged counts are as follows. SQL injection resulted in 23,847 flagged requests with primary detection stage BLKLST. Cross-site scripting attempts generated 18,293 flagged requests via BLKLST. Path traversal attacks produced 15,621 flagged requests through BLKLST. Administrative interface probes yielded 31,456 flagged requests detected by ADM_PRB. Webshell upload attempts totaled 4,127 flagged requests identified by WEBSHEL. HTTP method abuse resulted in 892 flagged requests caught by DNG_MTH. Server-side request forgery attempts generated 1,834 flagged requests via ABS_URI. DNS-over-HTTPS probes produced 247 flagged requests through DNS_QRY. Other or mixed attack patterns (including the residual set requiring iterative pattern refinement) totaled 32,130 flagged requests across multiple stages. The combined total reached 128,447 flagged events.

Security analyst review identified causes of the 368 false positives as follows. Unusual URL structures or entropy outliers (43 percent, $n = 158$) resulted from legitimate applications with complex query parameters or encoded content triggering path entropy thresholds. Aggressive crawlers or high-rate access patterns (31 percent, $n = 114$) involved search engine bots exceeding normal request patterns but originating from verified IP ranges. Signature overreach on benign patterns (18 percent, $n = 66$) occurred when regex patterns matched legitimate functionality (such as /admin paths in legitimate admin interfaces for authorized users). Non-standard user agents (8 percent, $n = 30$) included custom HTTP clients or automated tools with unusual user agent strings but benign intent. All false positives were identified through security analyst review and addressed via pattern refinement or whitelist additions where appropriate.

3.3 Multi-Stage Pipeline Analysis

Decision path analysis revealed how requests flow through the 9-stage pipeline with immediate termination upon Suspicious or Allow verdicts.

Table 3 presents approximate stage activation frequency and exit points. Stage DNG_MTH processed 2,306,000 requests (100 percent), generated 892 suspicious verdicts with immediate pipeline termination (0.04 percent), produced 0 allow verdicts, and continued 99.96 percent. Stage ABS_URI processed 2,305,108 requests (99.96 percent of original traffic), generated 1,834 suspicious verdicts with immediate pipeline termination (0.08 percent), produced 0 allow verdicts, and continued 99.92 percent. Stage ENC_SCHEME processed 2,303,274 requests (99.88 percent), generated 0 suspicious verdicts (visibility only), produced 0 allow verdicts, and continued 100 percent. Stage ACME processed 2,303,274 requests (99.88 percent), generated 0 suspicious verdicts, produced 12,847 allow verdicts with immediate pipeline termination (0.56 percent), and continued 99.44 percent. Stage DNS_QRY processed 247 requests (0.01 percent), generated 247 suspicious verdicts with immediate pipeline termination (100 percent of those reaching the stage), produced 0 allow verdicts, and continued 0 percent. Stage ADM_PRB processed 2,290,180 requests (99.3 percent), generated 31,456 suspicious verdicts with immediate pipeline termination (1.37 percent), produced 0 allow verdicts, and continued 98.63 percent. Stage WEBSHEL processed 2,258,724 requests (97.9 percent), generated 4,127 suspicious verdicts with immediate pipeline termination (0.18 percent), produced 0 allow verdicts, and continued 99.82 percent. Stage WHTLST processed 2,254,597 requests (97.8 percent), generated 0 suspicious verdicts, produced 0 allow verdicts (empty set), and continued 100 percent. Stage BLKLST processed 2,254,597 requests (97.8 percent), generated 89,841 suspicious verdicts with immediate pipeline termination (3.99 percent), produced 0 allow verdicts, and continued 96.01 percent.

Key observations from pipeline analysis follow. Early exit effectiveness: the ACME stage allowed 12,847 certificate renewal requests with immediate pipeline termination without subjecting them to security analysis, preventing false positives on critical infrastructure. Conditional stage efficiency: DNS_QRY appeared in only 247 decision paths (0.01 percent of total requests), demonstrating proper conditional inclusion rather than universal presence. Blocklist concentration: the BLKLST stage flagged 69.9 percent of total flagged requests (89,841 of 128,447), indicating strong regex pattern coverage. Administrative probe prevalence: ADM_PRB flagged 24.5 percent of flagged events (31,456 requests), reflecting widespread automated scanning for WordPress, phpMyAdmin, and other common administrative interfaces. Pipeline termination efficiency: requests triggering Suspicious verdicts in early stages (DNG_MTH, ABS_URI, DNS_QRY, ADM_PRB, WEBSHEL) bypassed subsequent stages, reducing computational overhead for clearly malicious traffic.

3.4 Feature Engineering Validation

The 100-feature extraction system successfully differentiated suspicious from benign traffic across multiple dimensions. Feature extraction operated on all requests, including both clean traffic and flagged suspicious requests, enabling ML-ready input preparation regardless of deterministic pipeline outcomes.

Table 4 presents indicative feature statistics comparing benign versus flagged requests. Path depth averaged 2.3 ± 1.1 for benign requests versus 4.7 ± 2.8 for flagged requests. Path length measured 24.7 ± 12.3 characters for benign versus 67.4 ± 48.2 for flagged. Keyword count averaged 0.3 ± 0.6 for benign versus 3.8 ± 2.1 for flagged. Shannon entropy measured 3.8 ± 0.9 bits for benign versus 6.2 ± 1.4 bits for flagged. Special character count averaged 3.2 ± 2.1 for benign versus 18.7 ± 11.4 for flagged. These feature separations demonstrate strong discriminative patterns suitable for ML classification.

Principal component analysis decomposition of the 80-bin character histogram revealed distinct clustering patterns. Benign requests concentrated in alphanumeric character ranges with balanced distributions. SQL injection attempts showed high frequency of quotes, parentheses, and semicolons (bins 7, 8, 11, 27). Cross-site scripting attacks exhibited elevated angle brackets and script-related characters (bins 28, 30, 62). Path traversal attempts demonstrated repeated dot and slash patterns (bins 14, 15, 47). These patterns validate the character histogram as a complementary feature set capturing lexical attack signatures invisible to structural analysis.

3.5 Resource Utilization Under Production Load

Continuous monitoring documented resource consumption within systemd constraints across varying traffic patterns. Resource measurements reflect deterministic pipeline and feature extraction operations; ML inference was not active during the measurement period.

Under typical operating conditions (50 to 150 requests per second), CPU utilization averaged 12 percent with peaks to 34 percent during traffic surges. Memory consumption maintained 8 MB baseline with 23 MB peaks when buffering requests. Disk I/O averaged 2.1 MB per second for logging operations.

Under stress test conditions simulating 500 requests per second, CPU utilization sustained 41 percent with peaks to 49 percent. Memory consumption sustained 47 MB with peaks to 89 MB. Request backlog reached a maximum of 127 buffered requests before latency degradation. Log write rate sustained 8.7 MB per second.

Even under 10× normal traffic load, the system remained within systemd resource limits (512 MB memory, 50 percent CPU quota) while maintaining sub-second processing latency. This headroom enables the system to absorb DDoS traffic spikes without service degradation. Additional GPU and CPU resources remain available for planned ML inference integration.

3.6 Operational Transparency and Auditability

Complete decision path logging enabled rapid incident investigation and compliance validation.

Example detection entry structure: timestamp in ISO 8601 format (2025-12-26 12:20:40), verdict (SUSPICIOUS, ALLOW, or CLEAN_PREML), source IP for per-host tracking, request method and path for pattern matching, HTTP status code, decision path showing sequential stage traversal with termination point, and pattern indicating specific signature or rule triggering detection. An example entry reads: "[2025-12-26 12:20:40] SUSPICIOUS | 45.142.120.15 | GET /wp-admin/setup-config.php | 404 | Path: DNG_MTH→ABS_URI→ENC_SCHEME→ACME→ADM_PRB→TERMINATED | Pattern: wp-admin".

Security operators reviewed 500 random detections comparing investigation time with and without decision paths. Without decision paths (regex-only logs), average investigation time was 8.3 minutes. With decision paths (hybrid system), average investigation time was 2.1 minutes. The reduction represented 74.7 percent decrease in investigation overhead. The decision path enables operators to immediately understand why a request was flagged, which stage detected it, and what pattern matched, eliminating the need to manually reconstruct detection logic.

Decision-path logging is designed to support audit evidence requirements such as ISO/IEC 27001 control verification by providing per-event rationale and traceability. The system's ability to explain every blocking decision through complete decision paths addresses regulatory expectations for automated security controls, enabling organizations to demonstrate both the security effectiveness and operational accountability of their intrusion detection mechanisms.

3.7 Comparison with Commercial Solutions

Cost-benefit analysis compared the host-level approach with commercial WAF services.

Table 5 presents approximate cost and feature comparison. The proposed hybrid solution costs \$47 monthly with operational flagging coverage comparable to commercial systems, 0.016% global false positive rate (14.7% false discovery rate within flagged events), full decision path explainability, and complete access to detection logic for customization. Cloudflare Pro costs \$240 monthly with approximately 95 percent detection rate (estimated from public benchmarks and vendor documentation), approximately 1 percent false positive rate (estimated), no explainability (black box operation), and limited customization via dashboard. AWS WAF costs approximately \$312 monthly (estimated for equivalent traffic volume of 2.3 million requests per month) with approximately 93 percent detection rate (estimated), approximately 2 percent false positive rate (estimated), rule-level only explainability, and moderate customization via rules. Imperva costs approximately \$495 monthly (estimated for equivalent traffic volume) with approximately 97 percent detection rate (estimated), approximately 0.5 percent false positive rate (estimated), summary reports for explainability, and vendor-controlled customization.

Key differentiators include cost efficiency (81 to 91 percent reduction versus commercial solutions), explainability (complete decision path transparency versus vendor black boxes), customization (direct access to detection logic and pattern databases), latency (on-host processing eliminates round-trip to edge infrastructure), and privacy (traffic never leaves organizational control). The host-level approach proves particularly valuable for regulated industries requiring explainable security decisions and data sovereignty.

4 Discussion

4.1 Principal Findings

This work demonstrates three significant findings regarding hybrid CNN-GRU intrusion detection architecture and production deployment.

First, hybrid architectures separating deterministic rule-based detection from ML augmentation enable fail-safe operation with clear performance guarantees. The deployed multi-stage deterministic pipeline flagged 128,447 suspicious events through regex matching across 2.3 million requests, while the concurrent 100-feature extraction system prepared input vectors capturing both structural and lexical patterns for CNN-GRU classification (implemented in the reference Python system, planned for Rust integration). This architecture addresses the key weakness of pure ML approaches [9] by providing guaranteed protection via deterministic stages regardless of ML component availability.

Second, systems programming languages like Rust enable production-grade performance without sacrificing safety or expressiveness. The 19× parsing and feature extraction performance improvement (excluding ML inference latency) and 4.4× memory reduction compared to Python demonstrates that security infrastructure need not choose between developer productivity and runtime efficiency.

Rust's memory safety guarantees eliminate entire vulnerability classes while achieving performance competitive with C, a combination particularly valuable for security-critical code.

Third, operational transparency through complete decision path logging proves essential for adoption in enterprise security operations. The 74.7 percent reduction in incident investigation time directly impacts security team effectiveness, while audit-ready decision trails support regulatory compliance requirements for automated controls. This finding demonstrates that explainability enhances rather than constrains operational effectiveness.

4.2 Architectural Design Decisions

Several design decisions proved critical to production deployment success.

Deterministic-first processing with immediate termination ensures that by executing rule-based stages before ML inference and terminating the pipeline upon Suspicious verdicts, the system architecture provides guaranteed protection even when ML components fail or are not yet deployed. This fail-safe design addresses a primary concern raised by security practitioners: dependency on fragile ML pipelines that may break during model updates or library changes. In our deployment, the deterministic stages alone flagged the full operational suspicious event set, demonstrating that the baseline protection remains robust. ML augmentation (when integrated) is intended to improve novel attack detection rather than serve as the primary defense mechanism. The immediate pipeline termination upon Suspicious verdicts reduces computational overhead by preventing unnecessary processing in subsequent stages for clearly malicious traffic.

Conditional path markers reduce log verbosity while maintaining complete decision transparency. Implementing conditional stage inclusion (DNS_QRY only appears when pattern matches) rather than universal stage evaluation addressed operator feedback that excessively verbose logs obscure critical information. The result: decision paths averaging 6.2 stages rather than 9 universal stages, improving readability without sacrificing completeness.

ACME challenge early exit with immediate termination reflects a critical operational constraint. The decision to provide unconditional Allow verdicts for Let's Encrypt ACME challenges at stage 4 (before any blocking logic) with immediate pipeline termination prevents certificate renewal failures that cause service outages. While theoretically possible to exploit ACME paths, the practical risk of breaking certificate infrastructure outweighs the theoretical security benefit. This pragmatic design choice exemplifies the production-deployment philosophy of balancing security with operational reliability.

Static binary deployment enabled by Rust's ability to produce 5.2 MB statically-linked binaries (release build) eliminates Python runtime dependencies, simplifying deployment and reducing attack surface. A single file copy to production servers replaces complex dependency management, virtual environments, and version compatibility testing. This operational simplicity directly contributed to deployment success across heterogeneous server environments (Ubuntu, Debian, Fedora).

4.3 Performance Characteristics and Scalability

The system's sub-second processing latency (mean 45 ms parsing plus 0.8 ms feature extraction totaling 45.8 ms, excluding ML inference) enables real-time decision-making for interactive web applications. This performance proves critical for security operations: blocking decisions must complete before the web server responds to the request, otherwise attacks succeed before detection.

At peak observed load (500 requests per second during stress testing), the system sustained 41 percent CPU utilization, indicating theoretical capacity of approximately 1,200 requests per second on the deployment hardware for deterministic pipeline and feature extraction operations. For context, typical small-to-medium websites serve 10 to 100 requests per second, providing 12× to 120× headroom. This overhead capacity enables the system to absorb DDoS traffic spikes without degradation. Additional computational resources remain available for planned ML inference integration.

Operation within systemd limits (512 MB memory, 50 percent CPU quota) demonstrates suitability for resource-constrained deployments. Many production servers allocate minimal resources to security tooling; the system's efficient resource usage enables deployment without dedicated security appliances or infrastructure changes.

The 19× parsing and feature extraction speedup versus Python reflects fundamental differences in language implementation: Rust compiles to native machine code with zero-cost abstractions, while Python interprets bytecode with runtime overhead for type checking and garbage collection. For security tooling operating in critical paths, this performance difference proves decisive.

4.4 Limitations and Constraints

Several limitations constrain generalizability of our findings.

The 2.3 million request dataset spans 14 months and five domains, providing substantial temporal coverage but limited application diversity. Performance characteristics may differ for high-traffic applications (exceeding 1,000 requests per second sustained), specialized frameworks (Ruby on Rails, Django, ASP.NET) with different attack surfaces, API-only services lacking traditional web application patterns, and single-page applications with minimal server-side routing.

While the system flagged events across common attack categories (SQL injection, XSS, path traversal, admin probes), sophisticated attacks may evade detection if not represented in the pattern database. These include application-specific logic flaws, business logic abuse, account takeover via credential stuffing, API parameter pollution, and GraphQL query depth attacks.

This paper documents the deterministic pipeline and feature extraction components currently operational in production. The CNN-GRU model training and inference integration, while implemented in the reference Python system, were not deployed in the Rust production system during the measurement period. Our results therefore reflect deterministic detection performance with concurrent ML-ready feature extraction, not full hybrid ML-augmented detection. Performance measurements (19× speedup, 45 ms latency) reflect parsing and feature extraction only, excluding ML inference latency. Future work will evaluate whether ML inference provides anticipated improvements in novel attack detection and measure the complete end-to-end latency including ML inference operations.

The 45 ms parsing latency assumes deployment on modern multi-core processors (AMD Ryzen 9 5900X). Lower-performance hardware will exhibit proportionally longer latencies. The 5.2 MB binary size and 8 MB memory footprint remain constant, but processing throughput scales linearly with CPU performance.

The system parses Apache Combined Log Format specifically. Adaptation to other web servers (Nginx, IIS, Caddy) or log formats requires modifications to the parsing layer. While conceptually straightforward, this represents an integration cost for heterogeneous environments.

The observed 14.7% false discovery rate within flagged events indicates that approximately 1 in 7 alerts presented to security analysts represents benign traffic. While the global false positive rate of 0.016% relative to total traffic remains low, the operational burden of investigating false alerts represents a significant consideration for security operations teams. Future work incorporating ML inference may reduce false discovery rates through improved pattern generalization.

4.5 Comparison to Related Work

Our results align with academic benchmarks while providing novel operational insights.

Regarding operational flagging performance, the deployed deterministic pipeline's effective handling of signature-identified events demonstrates robust operational performance. The observed 14.7% false discovery rate within the reviewed sample of flagged events reflects the deterministic-first architecture's pattern matching behavior. The corresponding global false positive rate of 0.016% relative to total traffic demonstrates effective filtering of benign traffic while maintaining detection coverage. While direct comparison with supervised learning benchmarks like NSL-KDD (99.87 percent, [12]) and UNSW-NB15 (99.43 percent, [1]) is limited by different evaluation methodologies (operational metrics versus preprocessed labeled datasets), the deterministic architecture provides operational guarantees through explicit pattern matching.

Previous work reported CNN-GRU inference times of 2.3 to 4.7 milliseconds per sample on dedicated GPUs [5]. Our 0.8 millisecond feature extraction time reflects only the preparation phase (no ML inference deployed during measurement period), but demonstrates that preprocessing overhead remains negligible compared to overall detection latency. When ML inference activates, we anticipate total latency below 10 milliseconds based on similar architectures, yielding combined parsing, feature extraction, and inference latency under 55 milliseconds.

No prior work compared ML-based intrusion detection costs against commercial WAF services. Our finding of 81 to 91 percent cost reduction while achieving comparable operational coverage provides economic justification for host-level deployment, addressing a key adoption barrier beyond technical performance.

The decision path logging approach extends work on explainable AI in security [6] by providing operator-actionable transparency rather than model interpretability alone. Our 74.7 percent reduction in investigation time quantifies the operational value of explainability, demonstrating that transparency serves practical security operations rather than only compliance requirements.

4.6 Implications for Security Operations

Our findings suggest several implications for practical intrusion detection deployment.

Hybrid architectures should leverage deterministic methods for known patterns while preparing for ML augmentation to detect novel attacks. The robust baseline protection via rule-based stages demonstrates that signatures remain highly effective for known attack vectors. Rather than replacing signatures with machine learning, optimal architectures should retain deterministic guarantees for known threats while capturing ML's generalization benefits for novel pattern detection. This design philosophy addresses practitioner concerns about ML fragility while preparing infrastructure for ML enhancement. The observed 14.7% false discovery rate within flagged events suggests that ML augmentation focused on reducing false positives through improved pattern generalization may yield significant operational benefits.

Host-level detection offers complementary advantages to edge-based WAF services. Edge services provide valuable first-line defense through global threat intelligence and DDoS mitigation. However, host-level detection offers application-specific context enabling precise policy enforcement, elimination of edge round-trip latency, complete traffic visibility including internal requests, and data sovereignty for regulated industries. The optimal architecture employs both layers with different objectives: edge for volumetric filtering, host for semantic understanding.

The 74.7 percent investigation time reduction demonstrates that explainability directly impacts security team effectiveness. Rather than treating transparency as a compliance checkbox, organizations should demand decision path logging as a functional requirement. Systems that cannot explain blocking decisions impose hidden costs in investigation overhead and false positive remediation.

The system's operation within 512 MB memory and 50 percent CPU quota proves that effective security tooling need not require dedicated appliances. This finding challenges vendor narratives that sophisticated detection demands specialized hardware, suggesting that careful systems programming and algorithmic optimization enable deployment on existing infrastructure.

4.7 Future Directions

This work opens several research directions. Completing ML inference integration in the Rust system will enable evaluation of hybrid detection versus deterministic-only performance. Key questions include whether ML improves novel attack detection as anticipated, whether ML inference reduces the observed 14.7% false discovery rate within flagged events, what total end-to-end latency (parsing + feature extraction + ML inference) is achieved, and how model retraining frequency affects long-term performance.

Adaptive threshold tuning could improve operational efficiency. The current system uses fixed thresholds; adaptive approaches that adjust thresholds based on traffic patterns and false positive feedback may enhance performance while reducing false discovery rates.

Federated learning for collaborative defense would enable multiple organizations to collaboratively train models while preserving data privacy, improving detection of emerging attack patterns without sharing sensitive traffic logs.

Adversarial robustness evaluation should assess resilience against adversarial machine learning attacks designed to evade detection through minimal perturbations of malicious requests.

Multi-modal fusion could incorporate application logs, system metrics, and network flows alongside HTTP logs for comprehensive threat detection beyond the web layer.

Automated feature engineering using neural architecture search or automated feature learning could reduce manual feature engineering burden while maintaining interpretability.

5 Conclusions

This work demonstrates that hybrid CNN-GRU intrusion detection architectures can be designed for production-grade performance on commodity hardware while maintaining operational transparency essential for security operations. Our 14-month deployment across three production servers processing 2.3 million requests validates several critical findings.

Performance viability: The Rust implementation achieved 19× parsing and feature extraction speedup (excluding ML inference latency) and 4.4× memory reduction compared to Python reference. The deployed deterministic pipeline achieved effective operational coverage on signature-matching operations. Sub-second processing latency (45 ms mean parsing plus 0.8 ms feature extraction, excluding ML inference) enables real-time decision-making within web server response paths.

Architectural effectiveness: The deterministic-first pipeline with concurrent ML-ready feature extraction provides fail-safe protection through rule-based detection while preparing 100-dimensional vectors for CNN-GRU augmentation (implemented in reference Python system, planned for Rust integration). This hybrid design addresses practitioner concerns about ML fragility by ensuring security guarantees even when ML components are unavailable or not yet deployed. Pipeline termination upon Suspicious verdicts reduces computational overhead by bypassing subsequent stages for clearly malicious traffic.

Operational transparency: Complete decision path logging reduced incident investigation time by 74.7 percent, demonstrating that explainability enhances rather than constrains operational effectiveness. The audit-ready decision trails support regulatory compliance requirements for automated security controls.

Economic viability: The host-level approach achieved 81 to 91 percent cost reduction versus commercial WAF services while providing comparable operational coverage and complete customization. This cost-benefit profile provides strong economic justification for adoption.

Systems programming benefits: Rust's memory safety guarantees, zero-cost abstractions, and 5.2 MB static binary deployment (release build) eliminated entire vulnerability classes while achieving performance competitive with C. The result: security-critical infrastructure that is both safe and fast.

Operational metrics: The system achieved a global false positive rate of 0.016% relative to total traffic (368 false positives across 2,306,000 requests), corresponding to a 14.7% false discovery rate within flagged events presented to security analysts. This dual-metric perspective clarifies both the system's effectiveness at filtering benign traffic and the operational burden of alert investigation.

While limitations exist including dataset scope, ML integration status in Rust deployment, and log format specificity, the successful production deployment validates that CNN-GRU architectures can be adapted to production security infrastructure with clear separation between deterministic guarantees and planned ML augmentation. The system's operation within modest resource constraints (512 MB memory, 50 percent CPU) demonstrates that effective intrusion detection need not require specialized hardware or infrastructure changes.

Future work completing ML inference integration in the Rust system will evaluate whether learned pattern recognition provides anticipated improvements in novel attack detection and false discovery rate reduction beyond the deterministic pipeline's demonstrated operational effectiveness. Performance characterization will include complete end-to-end latency measurements encompassing parsing, feature extraction, and ML inference operations. The hybrid architecture design ensures that this ML augmentation complements rather than replaces the proven deterministic foundation, maintaining both security effectiveness and operational transparency necessary for enterprise adoption.

Data Availability Statement

The datasets and source code supporting the conclusions of this article are not publicly available. The system described in this study is a production-deployed security platform, and the underlying source code, configuration artifacts, and operational telemetry are restricted due to security, privacy, and infrastructure protection considerations. Public release of these materials could expose system behavior or enable adversarial exploitation.

No public repository is available for the production implementation. Descriptions of the system architecture, detection logic, and feature extraction methodology are provided in the manuscript to support reproducibility at the design level.

Ethics Statement

This research involved analysis of production web server traffic on infrastructure owned and operated by AstroPema AI LLC. No personally identifiable information was retained in datasets or analysis. All traffic monitoring occurred on servers under the author's direct administrative control. The study complies with ethical guidelines for security research as outlined by the Menlo Report [3]. No human subjects participated in this research.

Author Contributions

JD Correa-Landreau conceived and designed the hybrid architecture, implemented the Rust production system, conducted the performance analysis, and wrote the manuscript.

Funding

This research received no external funding and was conducted as part of normal business operations at AstroPema AI LLC.

Acknowledgments

The author thanks the Rust programming language community for systems programming tooling, the security research community for foundational work on CNN-GRU architectures, and the Apache Software Foundation for web server infrastructure enabling this research.

Conflict of Interest

The author is the founder and CEO of AstroPema AI LLC, which may commercialize aspects of this technology. This potential conflict is disclosed for transparency. The research was conducted independently without external commercial pressures.

References

[1] Ahmad, Z., Shahid Khan, A., Wai Shiang, C., Abdullah, J., and Ahmad, F. (2021). Network intrusion detection system: A systematic study of machine learning and deep learning approaches. *Transactions on Emerging Telecommunications Technologies* 32(1), e4150.

[2] Buczak, A. L., and Guven, E. (2016). A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Communications Surveys and Tutorials* 18(2), 1153–1176.

[3] Dittrich, D., and Kenneally, E. (2012). *The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research*. US Department of Homeland Security.

[4] Fail2ban (2023). Fail2ban: Ban hosts that cause multiple authentication errors. Available at: https://www.fail2ban.org

[5] Kim, J., Kim, J., Thu, H. L. T., and Kim, H. (2016). Long short term memory recurrent neural network classifier for intrusion detection. In: *2016 International Conference on Platform Technology and Service (PlatCon)*, pp. 1–6.

[6] Marino, D. L., Wickramasinghe, C. S., and Manic, M. (2018). An adversarial approach for explainable AI in intrusion detection systems. In: *IECON 2018 – 44th Annual Conference of the IEEE Industrial Electronics Society*, pp. 2813–2818.

[7] Matsakis, N. D., and Klock, F. S. (2014). The Rust language. *ACM SIGAda Ada Letters* 34(3), 103–104.

[8] Ring, M., Wunderlich, S., Scheuring, D., Landes, D., and Hotho, A. (2019). A survey of network-based intrusion detection data sets. *Computers and Security* 86, 147–167.

[9] Sommer, R., and Paxson, V. (2010). Outside the closed world: On using machine learning for network intrusion detection. In: *2010 IEEE Symposium on Security and Privacy*, pp. 305–316.

[10] Vinayakumar, R., Alazab, M., Soman, K. P., Poornachandran, P., Al-Nemrat, A., and Venkatraman, S. (2019). Deep learning approach for intelligent intrusion detection system. *IEEE Access* 7, 41525–41550.

[11] Wang, W., Sheng, Y., Wang, J., Zeng, X., Ye, X., Huang, Y., and Zhu, M. (2020). HAST-IDS: Learning hierarchical spatial-temporal features using deep neural networks to improve intrusion detection. *IEEE Access* 8, 162345–162356.

[12] Wu, K., Chen, Z., and Li, W. (2020). A novel intrusion detection model for a massive network using convolutional neural networks. *IEEE Access* 8, 21615–21626.