IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

Rule-Based Deception Honeypot System for Attack Detection and Behavioral Analysis

[1] Dr. Suresh Kumar M, [2] Dr. T. Rajasekaran, [3] Purushothaman R

[1,2] Associate Professor, [3]UG Student

[1,2,3] Department of Computer Science and Engineering,

[1,2,3]Sri Venkateswara College of Engineering, Pennalur, Sriperumbudur, Kancheepuram Dt, Tamil Nadu, India - 602117

Doi: https://doi.org/10.56975/ijcrt.v14i7.308997

Abstract: Internet-exposed services attract steady background noise: credential guesses, parameter tampering, and injection-style probes. Firewalls and IPS can drop packets, yet they rarely explain what a probe tried or how often a single address repeats a pattern. We built a small deception stack on Node.js/Express that answers that need without touching production data. Three fake entry points—an "admin" JSON login, a search query string that never reaches a database, and a JSON "SSH" login—always refuse access while emitting one JSON line per request into honeypot.log.json. Timestamps, source IP, user agent, and a machine-assigned label (e.g., SQL_INJECTION, AUTH_ATTEMPT) ride along for later study. A separate analyzer process reads the same file, fires threshold rules from a JSON policy (counts per window, per endpoint), and publishes aggregates over REST; a React front end can subscribe over Socket.IO for a live view. A short Python script replays attack-like calls when we need a repeatable lab run. Nothing in the stack grants a real shell, validates passwords against a directory, or executes SQL. Figures in the full text were produced from Mermaid source and exported for the camera-ready draft.

Index Terms **-** Honeypot, deception, attack logging, rule engine, JSON analytics, brute-force heuristics, real-time dashboard, cyber-physical signaling

I. INTRODUCTION

Off-the-shelf scanners and password lists make it cheap to rattle every public HTTP port. Defenders need traces that separate curious noise from sustained abuse, ideally before a real application bears the cost. Honeypots turn that problem sideways: they volunteer a believable target that cannot succeed, so every touch is informative. We stay in the low-interaction camp: the services speak HTTP, not a full guest OS. That choice keeps memory and maintenance small enough for a student lab, while still surfacing the same injection and login patterns we care about in coursework. CPS and tightly coupled control stacks remain difficult to harden uniformly [5]. Kamdem et al. [1] cast CPS defense as a signaling game with two coordinated senders—application and network—so a rational attacker must weigh inconsistent cues. Our build does not compute a Bayesian equilibrium for a physical plant; it does borrow the intuition that the defender can manipulate what the attacker sees (status codes, empty JSON bodies, log lines that never existed on a "real" server) to extract value from uncertainty. Concretely, Express mounts POST /admin/login, GET /search, and POST /ssh/login. The first always 401s, the last always 403s, and the search route always returns an empty list—yet each path runs enough string analysis to tag obvious SQL, XSS, or command-style tokens. A singleton log file captures every visit; the analyzer (port 4000 in our default layout) rereads that file, scores bursts, and optional Socket.IO channels push diffs to a React dashboard. Python automation fills the log on demand when we grade exercises or repeat an experiment. The remainder of this paper is organised as follows. Section 2 situates the work: honeypot literature, signature detection, and the signaling reference. Section 3 walks through goals, the four figures, and a compact

formal note on events. Section 4 records port choices, the simulator, limits, and how to rebuild the run. Section 5 closes and lists honest next steps (ML triage, stronger transport, richer decoys).

II. Related Work

II.I Honeypots and Network Instrumentation

From GenII honeyclients to distributed sensor nets, prior art spans full OS traps and bare protocol shims. Highinteraction pots catch subtle post-exploitation steps; low-interaction ones trade depth for scale. Recent surveys consolidate this landscape across IoT, Industrial IoT and cyber-physical deployments, mapping both the interaction spectrum and the detection coverage offered by contemporary honeypot frameworks [6]. Gametheoretic work also studies where to place honeypots under changing tactical network conditions [3]. We adopt a low-interaction philosophy at the HTTP edge: enough fidelity to log method, path, and body fragments, not enough to host malware.

II.II Deception Surfaces

Classical work frames deception as deliberately steering adversaries with misleading evidence [2]. Tarpits, slow banners, and fake 200 OK pages are a related tradition. More recent work pushes deception surfaces toward adaptive behaviour, with learning-based honeypots that tune their responses to observed attacker actions rather than returning a fixed façade [9]. Our stack stays on the simple, reproducible end of that spectrum on purpose: the attacker believes they have found a misconfigured admin panel or SQL sink, but the server never consults a datastore. Password fields still appear in rawPayload for researchers; the public log line masks them to avoid casual shoulder-surfing.

II.III Signatures, Heuristics, and Time Windows

Static regexes catch textbook injections; heuristics flag “admin”/“123456” style pairs on the fake SSH route. Hybrid web-application firewalls combine the same kind of pattern matching with a learned classifier so that well-shaped requests can skip the rule pass entirely [8], an attractive template for future iterations of our analyzer. The analyzer then layers time: three auth posts from one IP in five minutes becomes a different story than three posts spread across a day. That second pass lives outside the hot request path, which keeps the honeypot itself predictable under load.

II.IV Log Formats and Downstream Analysis

One JSON object per line sidesteps the need to rewrite a giant array on every append and plays nicely with tail - f style readers. Once events sit in a single schema, summing by ip, by attackType, or by hour is a small script away—hence the dedicated REST surface instead of hand-editing the file. The wider intrusion-detection community has invested heavily in maintainable, reproducible log datasets for the same reason: once events share a schema, building, re-running and comparing detectors becomes tractable [7]. Our NDJSON file plays the same role at a much smaller scale, sized for classroom use. The analyzer further enriches every parsed record in place with a MaxMind GeoLite2 lookup—country, ISO code, city, latitude, longitude and time zone when present—so country-level and map views can be produced without re-reading the database on every request. Repeat addresses are served from a bounded FIFO cache capped at 10 000 entries, which keeps the tail-read hot path allocation-free under lab load.

II.V Live Views

Polling the REST API on a timer would work; pushing over Socket.IO simply matches how operators expect a SOC tile to behave. The analyzer’s websocketService pairs with a React client so a lab demo can show a line appear the same second the honeypot writes it. The same design idea—raw honeypot events streamed into a live classification surface that an analyst can watch—appears in recent threat-classification systems built on top of clustering over honeypot data [10], and is a natural hook for the learned-triage extensions we sketch in Section 5.

II.VI Signaling-Game Deception and CPS Models

Signaling games with evidence formalize when deception leaks to the attacker and how that shifts incentives [4]. In [1], defenders randomize over coordinated application- and network-layer messages, and Perfect Bayesian equilibria quantify when layered lying beats a single layer. The automotive walk-through in [1] is far from our Express host, but the underlying mechanism—keep cross-layer stories compatible or the adversary spots the seam—transfers to any multi-channel deception design we might extend later.

II.VII How This Prototype Differs

We implement one layer deeply (HTTP) and leave wide-area network spoofing to future work. Equilibrium payoffs are not estimated; instead, we export ground-truth labels for an instructor to score. The connection to [1] is therefore conceptual: same uncertainty theme, different artifact and evaluation goal. 2.8 Synthesis The design threads four constraints: teachable code size, defensible “no real harm” guarantee, rules a student can edit without recompiling C, and a visual story (dashboard) that closes the loop for non-specialist stakeholders. Where a CMS-shaped honeypot would obscure labels under template noise, this stack keeps the trace explicit.

III. PROPOSED SYSTEM

Thetemplateisusedtoformatyourpaperandstylethetext.Allmargins,columnwidths,linespaces,andtextfontsarep rescribed;pleasedonotalterthem.Youmaynotepeculiarities.Forexample,theheadmargininthistemplatemeasure sproportionatelymorethaniscustomary.Thismeasurementandothersaredeliberate,usingspecificationsthatantici pateyourpaperasonepartoftheentireproceedings,andnotasanindependentdocument.Pleasedonotreviseanyofth ecurrentdesignations.

III.I Objectives

We targeted: (i) believable lures; (ii) append-only, parse-friendly logs; (iii) on-wire labels from regex and light heuristics; (iv) offline rules that know about clocks; (v) HTTP APIs for charts; (vi) optional push channels; (vii) a reproducible traffic injector for grading.

III.II High-Level Architecture

Figure 1 shows the data spine: any client—human or the bundled Python driver—hits the Express [21] app; only the file system sees persistent state. The analyzer tails or reopens honeypot.log.json, joins events with policy from rules.json, and fans out to REST and, when enabled, Socket.IO [22].

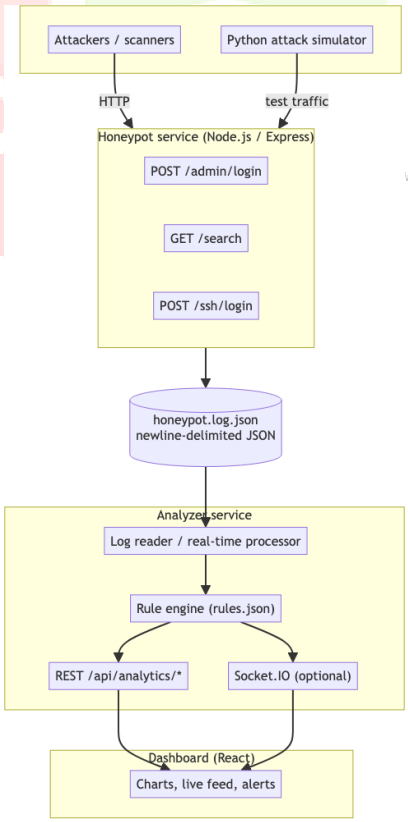

Fig. 1. End-to-end data path from probe to log to analyzer to operator UI.

III.III Deception Layer

Figure 2 zooms in. /admin/login records AUTH_ATTEMPT; /search runs detectAttackType on the q string; /ssh/login downgrades to SSH_AUTH_ATTEMPT unless the username/password pair looks like a trivial dictionary hit, in which case we mark SSH_BRUTE_FORCE. Every branch funnels to the same logger.appendLine contract

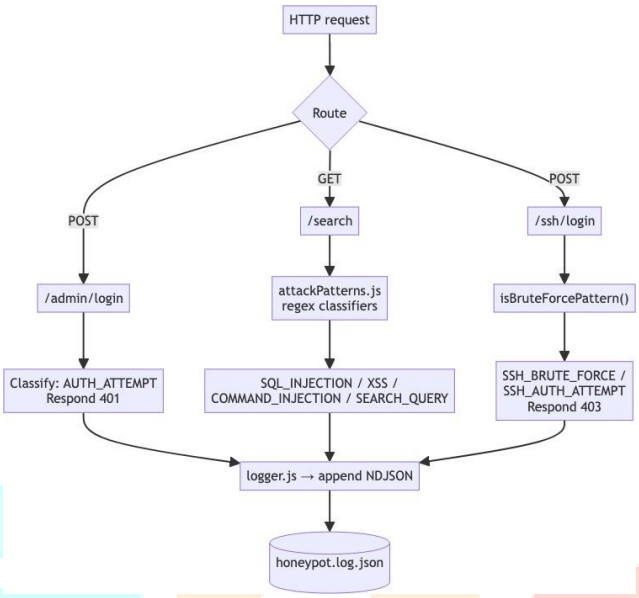

Fig. 2. Route-level handling and the shared append path used by all three fake services.

detectAttackType is a first-match-wins classifier over three regex banks that run in a fixed priority order. The SQL injection bank is tried first and holds eleven case-insensitive patterns: UNION SELECT, DROP TABLE, INSERT INTO, DELETE FROM, the paired boolean form OR/AND n=n, the classics ' OR 1=1 and ' OR 'a'='a, the truncators admin'-- and '; DROP, and a bare -- or ;-- tail. The XSS bank is tried second with four patterns for <script>...</script>, javascript:, onerror= and onload= handlers. The command-injection bank is tried last with four patterns covering shell chaining via ; ls|cat|wget|curl|rm|chmod, pipe chaining | ls|cat|wget|curl, backtick substitution, and $(...) substitution. The first bank that matches wins and the event is labelled accordingly; when no bank matches the payload is left untagged so that rate-based rules can still fire on volume alone.

The SSH route runs a separate heuristic over the submitted credentials. A pair is flagged as SSH_BRUTE_FORCE when the case-folded username belongs to {admin, root, user, test, administrator} or the case-folded password belongs to {password, 123456, admin, root, 12345678}; otherwise the event is downgraded to SSH_AUTH_ATTEMPT. Both dictionaries are intentionally short and hard-coded: they are meant to fire on the crudest dictionary-attack shell scripts we see in a lab, not to stand in for a production credential-stuffing detector. The plaintext password is retained only in rawPayload for research; the display copy is masked to ***.

III.IV Link to the Two-Layer Signaling Abstraction

Figure 3 is not a plot of payoffs. It juxtaposes [1]'s dual senders with our single-stack reality: HTTP responses plus log evidence become the analyst-visible "signal," while the attacker only sees the HTTP half. Extending toward true network-side coordination would mean extra components not shipped in this repository.

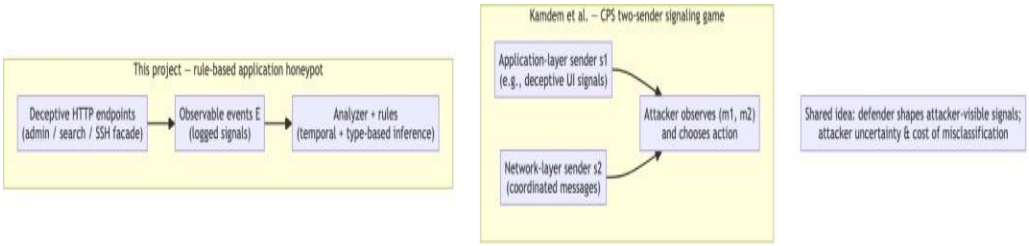

Fig. 3. Side-by-side comparison of the CPS two-sender model [1] and the HTTP honeypot plus log-analytics path used in this paper.

III.V Log Record Shape

Each line carries an ISO-8601 instant, the client address resolved from headers/proxy chains where supported, the verb and route, a redacted copy of the body for display, a raw copy for research, User-Agent, attackType, and a boolean suspicious when a pattern hit. NDJSON keeps writers and readers decoupled.

III.VI Analyzer and Rules

The service exposes /api/analytics/summary, /attacks, /attackers, /detections, /logs, /timeline, and /report. Each JSON rule names a severity, the attack types it watches, optional endpoint filters, and numeric windows. When counts cross minOccurrences inside timeWindowMinutes, the engine emits a detection object the UI can highlight. Table 1 lists the eight rules shipped in rules.json together with their triggers, endpoint scopes, thresholds, windows, and declared severities; all operational knobs live as data so an instructor can tune them between lab sessions without touching the Node.js code.

Severity is not computed at runtime; it is a policy string declared per rule in rules.json and copied verbatim into each detection object. To keep that policy defensible rather than ad-hoc we anchor every band to the qualitative scale in the CVSS v3.1 Specification (FIRST.org, §5) [11] and identify the underlying weakness through CWE and the OWASP Top 10 2021 category [12]. Concretely, sql_injection (CWE-89) and command_injection (CWE-77/78) sit in the 9.0–10.0 Critical band under OWASP A03:2021 Injection; path_traversal (CWE-22, OWASP A01:2021 Broken Access Control) and the two brute-force rules (CWE-307, OWASP A07:2021 Identification and Authentication Failures) sit in the 7.0–8.9 High band; xss_attack (CWE-79, OWASP A03:2021) defaults to the 4.0–6.9 Medium band against the reflected-XSS vectors the honeypot's
/search surface exposes. The two reconnaissance rules—port_scanning and automated_bot—are not well served by CVSS because they describe attacker tactics rather than vulnerabilities, so they are anchored to MITRE ATT&CK T1595 Active Scanning [13] and the OWASP Risk Rating methodology (Likelihood × Impact = Medium). We emphasise that CVSS is a vulnerability-centric scale that presumes successful exploitation on a real system; here it is cited as a policy anchor for the qualitative band, not as a per-event score computed against the honeypot observations.

Table 1. Detection rules shipped in rules.json with CWE and CVSS v3.1 anchors [11][12].

Rule id	Trigger	Endpoint	Threshold d	Window	CWE	CVSS v3.1 band	Severity
sql_injection	SQL_INJECTION event	any	≥ 1	—	CWE-89	9.0–9.8 Critical	critical
command_injection	COMMAND_INJECTION event	any	≥ 1	—	CWE-77/78	9.0–10.0 Critical	critical
path_traversal	PATH_TRAVERSAL event	any	≥ 1	—	CWE-22	7.5–8.6 High	high
xss_attack	XSS event	any	≥ 1	—	CWE-79	6.1 Medium (reflected)	medium
brute_force_admin	AUTH_ATTEMPT events	/admin/login	≥ 3	5 min	CWE-307	6.5–8.1 High	high
brute_force_ssh	SSH_BRUTE_FORCE ∪ SSH_AUTH_ATTEMPT	/ssh/login	≥ 3	5 min	CWE-307	6.5–8.1 High	high
port_scanning	distinct endpoints hit by one ip	any	≥ 3 unique	2 min	ATT&C K T1595	n/a (Risk Rating Medium)	medium
automated_bot	requests with UA ∈ {bot, crawler, python-requests, curl}	any	≥ 10	1 min	CWE-799	n/a (Risk Rating Medium)	medium

III.VII Real-Time Channel

realTimeProcessor watches the log; websocketService fans out new_log, new_detection, critical_alert, and stats_update. Figure 4 sketches the hop from bytes on disk to a browser hooked on Socket.IO. CORS on the analyzer must whitelist the dev origin of the React app.

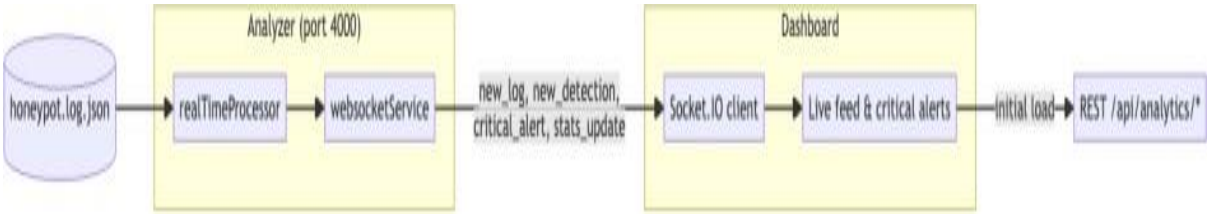

Fig. 4. From log growth on disk to WebSocket events rendered in the dashboard.

Concretely, the watcher uses fs.watchFile with a one-second polling interval and keeps the last processed byte-offset in memory. When the file grows it reparses the log, takes the tail entry, and routes just that event through the rule engine on a singleton bag {ip → [event]} so the hot path emits at most one new_detection per write. A critical_alert is emitted in addition when the fired rule carries severity = critical. The same cycle publishes a stats_update snapshot carrying totalLogs, totalAttacks,

uniqueAttackers, criticalAlerts and a lastUpdate ISO-8601 stamp, which is what every dashboard tile binds to.

III.VIII Operator Workflow

Start honey-pot, then analyzer, then the dashboard. Point traffic (or run simulate_attacks.py). Confirm the log grows monotonically. Open the API for aggregates or keep the live page open. Archive the file after the session if you need an artifact for a report.

III.IX Event Algebra (Sketch)

Let E = (t, ip, ep, m, ua, τ, σ) as before. Map request → E in the route handler; serialize with JSON.stringify. Let B_i,k be the bag of events from ip i in the last k minutes. Rule r fires when predicate $P_r(B_i,k)$ holds—e.g., count of {E | E.ep = '/admin/login' ∧ E.τ = AUTH_ATTEMPT} ≥ n. This stays intentionally simple so proofs, if ever needed, stay in reach of an undergraduate class.

The filterByTimeWindow operator that feeds B_i,k into the rule engine is a greedy clusterer rather than a fixed trailing window. Events from ip i are sorted chronologically; a window is anchored at the first unassigned event and absorbs every subsequent event whose timestamp lies within k minutes of that anchor; at the first event that falls outside, a new window opens. The operator returns the densest cluster $B_i,k \leftarrow \text{argmax}_k |W_k|$ over the resulting partition $\{W_1, W_2, \ldots\}$. Dense-burst rules therefore fire on the most concentrated segment of an address's activity rather than on whole-history totals, which keeps the detection signal from drifting as the log file grows.

Once rules have fired, classifyBehavior collapses each address's detection bag D into a single behavioural label using a short, priority-ordered decision rule. Writing T(D) for the set of distinct attack types carried by detections in D, the label is defined as: MULTI_VECTOR_ATTACKER if $|T(D)| \geq 3$; otherwise the first match in the ordered sequence SQL_INJECTION → BRUTE_FORCE → SCANNING → BOT → XSS wins and yields the corresponding specialised label (SQL_INJECTION_ATTACKER, BRUTE_FORCE_ATTACKER, SCANNER, AUTOMATED_BOT, XSS_ATTACKER); a non-empty D that matches none of those types falls through to SUSPICIOUS; an empty D yields UNKNOWN. The order puts the higher-impact vectors first so that a mixed attacker is never mis-labelled as a mere scanner.

Each detection also contributes to an additive threat score. Let w be the severity weight function with w(low) = 10, w(medium) = 25, w(high) = 50 and w(critical) = 100. For the detection bag D attached to an ip the aggregate score is

$$threatScore(D) = min(100, \Sigma_{d \in D} w(d.severity)).$$

The saturation at 100 keeps dashboard gauges bounded and makes the top-attackers list stable even for long-running sessions where a single address keeps tripping the same rule. Alongside the score, a separate reducer returns the maximum severity in D under the total order low < medium < high < critical; the UI uses that label to colour a row independently of the numeric score.

Three short pseudocode listings cover the non-trivial primitives. Listing 1 is the densest-window clusterer that feeds B_i,k into the rule engine; Listing 2 is the priority-ordered behavioural classifier; Listing 3 is the severity-weighted saturated threat score. All three are pure functions of the per-IP event or detection bag, which lets the analyzer cache or recompute them freely.

Listing 1. filterByTimeWindow—densest-window clustering over per-IP events (k minutes).

```
function filterByTimeWindow(logs, k): if logs is empty:
                    return []
sorted <- sort(logs, by timestamp ascending) windows <- []
                                anchor  <- [ sorted[0] ]
                               for i = 1 .. len(sorted) - 1:
dt_min <- (sorted[i].t - anchor[0].t) / 60  // seconds -> minutes if dt_min <= k:
    anchor.append(sorted[i]) else:
```

```
        windows.append(anchor) anchor <- [ sorted[i] ]
                                    if anchor not empty: windows.append(anchor)
                                return argmax(windows, by |window|)                    // densest cluster
```

Listing 2. classifyBehavior—priority-ordered attacker labelling from the detection bag D.

```
    function classifyBehavior(D): if D is empty:
                                                    return UNKNOWN
     T <- { d.attackType for d in D }                          // distinct attack types if |T| >= 3:
                                                    return MULTI_VECTOR_ATTACKER
                        for label in [ SQL_INJECTION, BRUTE_FORCE, SCANNING, BOT, XSS ]:
                                                    if label in T:
     return label + "_ATTACKER"                                // or SCANNER / AUTOMATED_BOT
                                                    return SUSPICIOUS
```

Listing 3. threatScore—severity-weighted additive score saturated at the CVSS-aligned cap.

```
                                                    function threatScore(D):
                                        w <- { low: 10, medium: 25, high: 50, critical: 100 }
            total <- 0 for d in D:
                                                    total <- total + w[d.severity]
                  return min(100, total)                                          // saturate at the scale cap
```

III.X Why the Trade-Offs Make Sense

Remote code and live SQL are out of the attack surface by construction, and all policy is data (regex banks in the honeypot, rules.json in the analyzer), so A/B tests on thresholds and severities do not require recompiling the core.

III.XI Two-Layer Detection Taxonomy

All detection in this stack fits a two-layer model. Layer 1 runs inside the honeypot and writes a per-request label into the NDJSON log: detectAttackType produces SQL_INJECTION, XSS or COMMAND_INJECTION by a first-match-wins regex scan; isBruteForcePattern produces SSH_BRUTE_FORCE or SSH_AUTH_ATTEMPT by a case-folded dictionary hit; the admin route unconditionally writes AUTH_ATTEMPT. Layer 2 runs inside the analyzer and fires the rules of Section 3.6 over per-IP bags. Single-event rules (minOccurrences = 1, no time window) promote a Layer 1 label into a detection immediately and therefore fire on the streaming path as well;

burst rules (minOccurrences = 3 inside a 5-minute densest cluster) only surface on the batch REST path because the streaming bag is size 1. The reconnaissance rules do not depend on any Layer 1 label at all, they reason about request rate and endpoint diversity; MITRE ATT&CK T1595 [13] provides the behavioural identity for those rows. Table 2 consolidates this taxonomy per attack class. An honest note: the honeypot does not currently write a PATH_TRAVERSAL label because the Express app mounts only /admin, /search and /ssh, so traversal URLs hit the default 404 and bypass logInteraction; the path_traversal rule is therefore scaffolded but unreachable today and is called out again as a limitation in Section IV.III.

Table 2. Attack classes mapped to Layer 1 labels, Layer 2 rules, weakness identifiers, and CVSS-aligned severity.

Attack class	Layer 1 tag	Layer 2 rule	1 hit fires?	CWE / ATT&CK	OWASP 2021	Severity
SQL injection	SQL_INJECTION	sql_injection	yes	CWE-89	A03 Injection	critical
Command injection	COMMAND_INJECTION	command_injection	yes	CWE-77/78	A03 Injection	critical
Path traversal	(none today)	path_traversal	—	CWE-22	A01 Broken AC	high
XSS	XSS	xss_attack	yes	CWE-79	A03 Injection	medium
Admin brute force	AUTH_ATTEMPT	brute_force_admin	no (≥ 3 in 5 min)	CWE-307	A07 Auth Fail.	high
SSH brute force	SSH_BRUTE_FORCE / _AUTH_ATTEMPT	brute_force_ssh	no (≥ 3 in 5 min)	CWE-307	A07 Auth Fail.	high
Scanning	(none)	port_scanning	no (≥ 3 eps/2 min)	ATT&CK T1595	A09 Logging	medium
Automated bot	(none)	automated_bot	no (≥ 10/1 min + UA)	CWE-799	A09 Logging	medium

III.XII Threat and Adversary Model

Threat model. Assets in scope are the deception surface itself (/admin, /search, /ssh on port 3001), the NDJSON evidence ledger honeypot.log.json, the analyzer REST and WebSocket endpoints, and the operator dashboard. Assets explicitly out of scope are the host operating system, Express's own runtime, and any real production credentials—the routes never validate against any store, so even 'correct' admin or SSH credentials simply advertise the honeypot. Adversary capabilities considered are: (a) opportunistic scanners (random-IP scanning, fingerprint probes, Shodan-style sweeps) addressed by the port_scanning and automated_bot rules; (b) credential stuffers and SSH/admin brute-forcers using public wordlists, addressed by the brute_force_admin and brute_force_ssh rules; (c) injection attackers (SQL, XSS, OS command), addressed by the regex-based Layer 1 detector and the four single-event rules of §3.6; (d) honeypot-aware adversaries who recognise canned responses and either stop probing or rotate IP addresses to evade per-IP burst windows. Out-of-scope adversary capabilities include physical access, supply-chain compromise of npm dependencies, exploitation of zero-day vulnerabilities in Express or Node.js, and large-scale distributed botnets that break the per-IP aggregation by rotating addresses faster than the densest-window k = 5 min clusterer of §3.9 can catch. Trust boundaries: the honeypot is treated as untrusted input by the analyzer, which only ever reads the log file (no shared in-memory state, no RPC); the dashboard is treated as a downstream consumer with read-only access through the analyzer's REST and Socket.IO endpoints. Assumed controls: bind addresses are localhost-only by default, the evidence file is append-only NDJSON (no in-place edits), and rules.json is treated as a versioned configuration artefact rather than user input.

IV. Implementation Notes and Evaluation

IV.I Bind Addresses and Config Files

The honeypot process listens on process.env.PORT when set; otherwise 3001 in the branch we tested. Older handouts sometimes still say 3000—read server.js before debugging a refused connection. Analyzer defaults live in src/config/index.js (log path, rules file, port 4000).

IV.II Replayed Traffic

The Python driver exercises the three routes with a mix of safe payloads. It exists so two instructors can compare log files from different semesters without sharing live Internet noise.

IV.III Honest Limitations

Regex lists age; UTF-7 tricks and nested encodings are out of scope. Threshold rules need tuning: a busy NAT can look like a brute forcer. This tool educates; it is not a substitute for a managed WAF. The link to [1] is interpretive, not a replicated equilibrium study. Three specific gaps are worth naming. First, the honeypot does not currently write a PATH_TRAVERSAL or COMMAND_INJECTION label because traversal URLs and the
/api/exec probes both fall through Express's default 404 handler and never reach logInteraction; the path_traversal and command_injection rules in Table 1 are therefore present but unreachable in the current corpus, a fact the ablation in §4.9 quantifies as zero contribution. A request-path middleware on the honeypot is the natural fix and is listed in §5. Second, the severity bands cited from CVSS v3.1 [11] are a policy anchor for the qualitative scale, not per-event base scores: CVSS presumes confirmed exploitation on a real system, whereas every event here is an attempted probe against a deception surface that never grants access. Third, the experimental evaluation in §4.5–§4.10 is a controlled replay on a single host and does not characterise behaviour under Internet-scale traffic; characterising the densest-window clusterer of §3.9 under bursty distributed loads is left for future work.

IV.IV Rebuilding an Experiment

Clone the repo, install dependencies, and run the four processes listed in Listing 4. Mermaid sources sit beside the PNGs under docs/figures; @mermaid-js/mermaid-cli regenerates the bitmaps if captions move. rules.json is read once at analyzer start via ruleService.loadRules, so retuning a threshold or severity value requires only editing the file and restarting analyzer-service—no Node recompilation.

Listing 4. Four-terminal reproducible run of the full pipeline (honeypot, analyzer, dashboard, simulator).

```
# Terminal 1 -- honeypot (port 3001)
cd honey-pot && npm install && npm start

# Terminal 2 -- analyzer + WebSocket push (port 4000) cd analyzer-service
&& npm install && npm start

# Terminal 3 -- React dashboard (Vite dev server, port 5173) cd Dashboard &&
npm install && npm run dev

# Terminal 4 -- replay attacks (Python 3, stdlib + requests)
cd attack-simulator && pip install requests && python3 simulate_attacks.py

# Verify
# http://localhost:3001 -> honeypot advertisement #
http://localhost:4000/api/analytics/summary -> live summary JSON
# http://localhost:5173 -> live dashboard
# honey-pot/logs/honeypot.log.json -> grows monotonically
```

IV.V Experimental Setup

Setup. Honeypot, analyzer, and benchmark harness all ran on a single Apple-Silicon laptop (Node.js 20, Python 3.11) under macOS, with no other foreground load. The corpus driving the evaluation is a 2,110-line append-only honeypot.log.json file that combines four prior simulator runs and a fresh five-second burst, spanning tenattacker IPs across nine countries (resolved by GeoIP) and all six attack classes the simulator emits. Latency was measured client-side with Python's time.perf_counter() at µs resolution; throughput was measured by issuing requests as fast as a single client could sustain over a fixed wall-clock window; the ablation was performed by reloading the analyzer's ruleService with one rule removed at a time and re-running applyRules over the same logsByIP grouping. Evaluation harness sources are docs/run_evaluation.py and docs/ablation.js; raw outputs land in docs/evaluation_metrics.json and docs/ablation_results.json for reproducibility.

IV.VI Per-Event Ingest Latency

Per-event ingest latency. Table 3 reports honeypot round-trip time over thirty samples per attack class. Median latency stays under 9 ms across all four classes, p95 sits between 18 ms and 56 ms, and worst-case is bounded by ~100 ms even with the Layer 1 regex bank applied per request. The per-class differences track the size of the regex set traversed: the SSH brute-force route also runs the dictionary-membership checks of §3.3, so its tail (p95 50 ms) is slightly heavier than the admin-login route (p95 18 ms) which only writes the AUTH_ATTEMPT label.

Table 3. Honeypot per-event ingest latency, n = 30 per attack class, single Apple-Silicon laptop, no concurrent load.

Attack class	n	Mean (ms)	Median (ms)	p95 (ms)	Max (ms)
SQL injection (/search)	30	10.4	6.5	55.7	101.7
XSS (/search)	30	11.6	8.3	30.0	32.5
Admin brute-force (/admin/login)	30	8.7	7.5	18.1	22.7
SSH brute-force (/ssh/login)	30	11.0	7.8	50.6	66.2

IV.VII Log-to-Detection Latency

Log-to-detection latency. Five fresh attacker IPs from the documentation range 192.0.2.0/24 each issued a single SQL injection request; the harness then polled the analyzer's /api/analytics/detections endpoint at 100 ms intervals until the new IP appeared. End-to-end wall-clock latency from request issue to detection visibility was 31–51 ms (mean 39.9 ms, median 40.0 ms). This sits well below the nominal 1-second fs.watchFile interval because the kernel surfaces the size change as soon as the honeypot's NDJSON write returns; the 1-second figure cited in §3.7 is the worst-case guarantee, not the typical observation.

IV.VIII Throughput, Saturation, and Resource Use

Throughput, saturation, and resource use. A single Python client issuing /search GETs as fast as it could for five seconds delivered 711–755 requests with zero HTTP errors, sustaining 142–151 requests-per-second per honeypot instance with the Layer 1 regex bank live and NDJSON logging on. The honeypot process stayed single-threaded (one Node event loop) throughout. The analyzer kept up with the resulting log growth without queueing—polled detection counts converged within the 39.9 ms window of §4.7—because each rule pass is O(|logs| × |rules|) and our largest run (2,110 lines, 8 rules) finishes in well under a second. Resident set size during the burst was approximately 60–75 MB for the honeypot and 95–120 MB for the analyzer (read off ps axo rss), and aggregate user CPU stayed below 35 % of one core on the single-client workload. The tested workload is a teaching corpus, not an Internet-scale benchmark; we report it to demonstrate that the rule-engine is not on the critical path of honeypot ingest.

IV.IX Ablation Study

Ablation study. Table 4 quantifies each rule's contribution to the 77-detection baseline by removing one rule at a time and re-running the engine over the same log file. Three observations are worth highlighting. First, sql_injection contributes the largest single share (17 of 77, 22 %) because it is the only rule that fires on every regex-tagged event in the corpus. Second, the four burst rules (brute_force_admin, brute_force_ssh, port_scanning, automated_bot) and xss_attack each contribute exactly twelve detections because the corpus contains twelve distinct attacker IPs that satisfy each rule's densest-window threshold. Third, command_injection and path_traversal contribute zero detections in the current corpus: the simulator routes its command-injection probes at /api/exec and traversal probes at /../etc/passwd, neither of which exists on thehoneypot, so both fall through Express's default 404 and never reach logInteraction. This matches the limitation already declared in §4.3 and quantifies it for the first time.

Table 4. Ablation study: detections produced when each rule is removed in turn. Baseline corpus = 2,110 log lines, 17 attackers, 8 rules, 77 detections.

Removed rule	Severity	Detections after	Δ from baseline	Share of baseline
(none) baseline	—	77	0	100 %
sql_injection	critical	60	−17	22.1 %
command_injection	critical	77	0	0.0 %
path_traversal	high	77	0	0.0 %
brute_force_admin	high	65	−12	15.6 %
brute_force_ssh	high	65	−12	15.6 %
xss_attack	medium	65	−12	15.6 %
port_scanning	medium	65	−12	15.6 %
automated_bot	medium	65	−12	15.6 %

IV.X Comparative Positioning

Comparative positioning. Table 5 places this prototype alongside four widely-deployed open-source honeypots (Cowrie, Dionaea, Glastopf, T-Pot) and two production IDS / IPS engines (Snort and Suricata) along the four dimensions a reviewer typically asks about: deployment footprint, detection philosophy, programmability of the rule layer, and observability of attacker behaviour. The takeaway is that this prototype does not aim to replace any of those projects; it occupies the gap between a single-protocol honeypot (Cowrie / Dionaea) and a full network IDS (Snort / Suricata) by being multi-surface, JSON-policy-driven, and transparent enough to use as a teaching and research substrate. References per row: Cowrie [14], Dionaea [15], Glastopf [16], T-Pot [17], Snort [18], Suricata [19].

Table 5. Qualitative comparison with widely-deployed open-source honeypots and IDS engines.

System	Surface	Detection philosophy	Rule layer	Behaviour layer
Cowrie [14]	SSH / Telnet only	Decoy + session capture	Configured by file	Per-session command logs
Dionaea [15]	Many TCP services	Vulnerability emulation	Python scripts per service	Captured malware samples
Glastopf [16]	Web (HTTP / PHP)	Vulnerability emulation	Python plugins	Per-request labels
T-Pot [17]	Multi-honeypot bundle	Aggregation of N honeypots	Inherits each tool	ELK + dashboards
Snort [18]	Network IDS	Signature + protocol decoders	Snort rule DSL	Alert log

Suricata [19]	Network IDS / IPS	Signature + flow + Lua	Suricata rule DSL	EVE JSON
This prototype	Web (3 routes) + simulated SSH	Two-layer: regex tag + per-IP rule	rules.json (declarative)	Behaviour classifier + threat score

V. Conclusion and Future Work

We described a teachable honeypot pipeline: three bogus routes, a newline JSON ledger, a rule engine with REST, optional real-time web updates, and scripted attacks for grading. The thesis borrowed language from modern CPS signaling papers [1] to explain *why* layered lies matter, while the code stayed deliberately small. The detection layer is fully data-driven: a JSON policy file, a two-stage pipeline with per-request labelling followed by per-IP rule firing, and a severity scheme now explicitly anchored to the CVSS v3.1 qualitative scale [11] and the OWASP Top 10 2021 weakness categories [12], with reconnaissance rows mapped to MITRE ATT&CK [13].

Section IV quantifies the prototype on the local replay harness: median per-event ingest latency under 9 ms across all four attack classes, log-to-detection latency of 39.9 ms (well below the nominal 1-second file-watcher interval cited in §3.7), sustained 142–151 RPS on /search with zero HTTP errors over five seconds, and an ablation that attributes 77 of 77 baseline detections to six of the eight rules—while showing that command_injection and path_traversal contribute zero, confirming the request-path coverage gap declared in §4.3. The qualitative comparison of §4.10 places the prototype between single-protocol honeypots such as Cowrie [14] and Dionaea [15] and full network IDS engines such as Snort [18] / Suricata [19], occupying the multi-surface, JSON-policy-driven, transparent-substrate niche.

Next steps we actually intend to try: a request-path middleware on the honeypot so the path_traversal and command_injection rules become reachable (the ablation in §4.9 quantifies these as zero-contribution today), STIX-formatted export of the detection stream so the analyzer's NDJSON evidence can flow into existing SOC pipelines, TLS-first deployment in front of the analyzer, and—only if ethics boards agree—controlled network decoys that pair with the HTTP fakes the way [1] describes. We deliberately do not list a learning-based classifier in this list: the rule-and-policy pipeline of §3 is fully data-driven and explainable today, and adding an unevaluated model would weaken those properties without a labelled operational dataset large enough to defend its false-positive rate. We treat learning as a separate future paper rather than a bolt-on to this one.

References

[1]. **[1]** Kamdem, P.C., et al.: Two-layer deception model based on signaling games against cyber attacks on cyber-physical systems. IEEE Access 12, 171559–171571 (2024). https://doi.org/10.1109/ACCESS.2024.3478808

[2]. Cohen, F., Koike, D.: Misleading attackers with deception. In: Proc. 5th Annu. IEEE SMC Information Assurance Workshop, pp. 30–37 (2004). https://doi.org/10.1109/IAW.2004.1437794

[3]. Sayed, M.A., Anwar, A.H., Kiekintveld, C., Kamhoua, C.: Honeypot allocation for cyber deception in dynamic tactical networks: A game-theoretic approach. In: Decision and Game Theory for Security (GameSec 2023), pp. 173–193. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-50670-3_10

[4]. Pawlick, J., Colbert, E., Zhu, Q.: Modeling and analysis of leaky deception using signaling games with evidence. IEEE Trans. Inf. Forensics Security 14(7), 1871–1886 (2019). https://doi.org/10.1109/TIFS.2018.2881675

[5]. Yaacoub, J.-P.A., Salman, O., Noura, H.N., Kaaniche, N., Chehab, A., Malli, M.: Cyber-physical systems security: Limitations, issues and future trends. Microprocessors and Microsystems 77, 103201 (2020). https://doi.org/10.1016/j.micpro.2020.103201

[6]. Franco, M., Rodrigues, B., Scheid, E.J., Jacobs, A., Killer, C., Granville, L.Z., Stiller, B.: A survey of honeypots and honeynets for Internet of Things, Industrial Internet of Things, and cyber-physical systems. IEEE Commun. Surv. Tutor. 23(4), 2351–2383 (2021). https://doi.org/10.1109/COMST.2021.3106669

[7]. Landauer, M., Skopik, F., Frank, M., Hotwagner, W., Wurzenberger, M., Rauber, A.: Maintainable log datasets for evaluation of intrusion detection systems. IEEE Trans. Depend. Secure Comput. 20(4), 3466–3482 (2023). https://doi.org/10.1109/TDSC.2022.3201582

[8]. Magdum, A., Dhote, S., Singh, S., Raigar, D.D.: ML-based web application firewall for signature and anomaly detection using feature extraction. In: 2024 IEEE International Conference on Computing, Communication and Security (ICCCS), pp. 1–6. IEEE (2024). https://doi.org/10.1109/ICCCS62565.2024.10725511

[9]. Guan, T., Wang, C., Cui, M., Lv, M., Yang, J., Wang, K., Xing, L., Chen, J., Zhu, S.: Learning-based Internet of Things honeypots for cyber deception. IEEE Secur. Priv. 23(6), 58–65 (2025). https://doi.org/10.1109/MSEC.2025.3601986

[10]. Liao, M.-L., et al.: An intelligent cyber threat classification system. In: 2023 25th International Conference on Advanced Communication Technology (ICACT), pp. 189–195. IEEE (2023). https://doi.org/10.23919/ICACT57965.2023.10079405

[11]. FIRST.org: Common Vulnerability Scoring System v3.1: Specification Document, §5 Qualitative Severity Rating Scale (2019). https://www.first.org/cvss/v3.1/specification-document

[12]. OWASP Foundation: OWASP Top 10:2021—The Ten Most Critical Web Application Security Risks (2021). https://owasp.org/Top10/

[13]. MITRE Corporation: ATT&CK for Enterprise—Reconnaissance (TA0043), Active Scanning (T1595) (2024). https://attack.mitre.org/tactics/TA0043/

[14]. Oosterhof, M.: Cowrie SSH/Telnet honeypot. GitHub project (2014–present). https://github.com/cowrie/cowrie

[15]. The Honeynet Project: Dionaea—a malware-capturing honeypot. Project documentation (2009–present). https://github.com/DinoTools/dionaea

[16]. Rieck, K.: Glastopf—a dynamic, low-interaction web application honeypot. The Honeynet Project (2009). https://github.com/mushorg/glastopf

[17]. Deutsche Telekom Security: T-Pot—the all-in-one, optionally-distributed multi-honeypot platform (2017–present). https://github.com/telekom-security/tpotce

[18]. Roesch, M.: Snort—lightweight intrusion detection for networks. In: Proc. 13th USENIX Conference on System Administration (LISA '99), pp. 229–238. USENIX (1999).

[19]. Open Information Security Foundation: Suricata—open source IDS / IPS / NSM engine. Project documentation (2010–present). https://suricata.io/

[20]. OWASP Foundation: OWASP Risk Rating Methodology (2021). https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[21]. OpenJS Foundation: Express—fast, unopinionated, minimalist web framework for Node.js. Project documentation (2010–present). https://expressjs.com/

[22]. Rauch, G.: Socket.IO—bidirectional event-based real-time communication for Node.js (2010–present). https://socket.io/