WSHawk: Stateful Security Assessment of WebSocket Applications through Adaptive Payload Mutation and Browser-Assisted Validation

Regaan R
Independent Researcher
Chennai, India
regaan.rothackers.com orcid.org/0009-0006-3683-7824

***Abstract*—Most automated web-application scanners are built around the request/response model of HTTP and do not translate cleanly to WebSocket endpoints, where a single long-lived, bidirectional connection carries application state and where server reactions may arrive asynchronously and out of band. This paper presents an implementation study of WSHawk, an open-source toolkit (v4.0.1, AGPL-3.0) for authorized security assessment of WebSocket and related realtime web applications. Working strictly from the source code, this study describes how WSHawk keeps connections open across a learning phase and per-vulnerability probing phases; how a two-layer payload subsystem—a strategy-scored mutation engine and a genetic "Smart Payload Evolution" pipeline (context-aware generator, response-classifying feedback loop, and crossover/mutation evolver)—adapts attack strings to observed server behaviour; how a Playwright-backed browser pool converts heuristic cross-site scripting (XSS) findings into sandboxed execution evidence and records/replays single sign-on flows; and how out-of-band callbacks detect blind XML External Entity (XXE) and Server-Side Request Forgery (SSRF). This paper further documents a project-backed store and a set of parallel HTTP and WebSocket attack services (replay, authorization diffing, race testing, subscription abuse) unified by a variable-templating workflow engine with six built-in playbooks; a protocol-graph subsystem that fingerprints six realtime protocol families (GraphQL-WS, Phoenix Channels, ActionCable, SignalR, Socket.IO, and a generic/binary fallback) and recommends attacks; a binary-message handler with format auto-detection and mutation; a WebSocket endpoint-discovery module; a session-hijacking tester and a blue-team defensive-validation suite (DNS exfiltration, bot detection, CSWSH, and WSS/TLS posture); an Ed25519-signed, hash-chained evidence bundle with redaction; a process-isolated plugin system; an adaptive token-bucket rate limiter and an HTTP resilience layer with retry and circuit breaking; and integrations for Jira, DefectDojo, and webhooks. The current repository focuses on engineering validation through a unit/integration test suite and three local realtime-application validation labs; a comprehensive empirical evaluation of detection accuracy against a benchmark corpus remains future work.**

***Index Terms*—WebSocket security, stateful scanning, payload mutation, genetic algorithms, browser-assisted validation, out-of-band testing, protocol inference, race conditions, authorization testing, CVSS, tamper-evident evidence, penetration testing.**

I. Introduction

The WebSocket protocol [1] establishes a persistent, full-duplex channel over a single TCP connection after an HTTP upgrade handshake. It underpins chat systems, collaboration platforms, trading interfaces, and other realtime Software-as-a-Service (SaaS) applications. From a security-testing standpoint, WebSockets differ from classical HTTP in three ways that matter to tooling. First, the connection is stateful: authentication, subscription, and application context are established through an ordered message sequence and must be preserved for the duration of testing. Second, communication is asynchronous and bidirectional: a server reaction to an injected message may be delayed, may arrive interleaved with unrelated push traffic, or may not be reflected on the same channel at all. Third, message payloads are frequently structured (JSON, XML, or binary encodings such as Protocol Buffers, MessagePack, or CBOR) so that naive injection of raw attack strings tends to be rejected before it reaches a vulnerable sink.

WSHawk is an open-source toolkit that targets this setting. This paper is an implementation study: its purpose is to describe what WSHawk actually implements, why each subsystem exists, and how the subsystems are organized, using only evidence available in the repository (source code, configuration, tests, documentation, and comments). Where a capability could not be verified from the reviewed source, it is stated explicitly rather than inferred.

The repository is organized as a Python package (`wshawk/`) providing the scanning engine, adaptive payload subsystems, attack services, protocol subsystem, evidence store, and a loopback daemon; an Electron desktop application (`desktop/`) that is the primary operator interface; a Manifest V3 browser companion (`extension/`); documentation guides (`docs/`); a unit and integration test suite (`tests/`); and three local validation labs (`validation/`). The package metadata declares version 4.0.1, Python 3.8+, and the AGPL-3.0 license [2].

Contributions. This paper (i) documents a stateful Web-

Socket scanning pipeline built on Python `asyncio` [3] and the `websockets` library that separates a passive learning phase from active per-vulnerability phases; (ii) describes a two-layer adaptive payload subsystem combining a strategy-scored mutation engine with a genetic-algorithm evolver and a response-classifying feedback loop; (iii) describes a Playwright [4] browser pool that supplies sandboxed execution evidence for XSS and records/replays SSO authentication flows; (iv) documents a protocol-graph subsystem that fingerprints six realtime protocol families and recommends attacks and playbooks; (v) documents parallel HTTP/WebSocket replay, authorization-diff, race, and subscription-abuse services unified by a variable-templating workflow engine; (vi) documents an Ed25519-signed, hash-chained evidence bundle with redaction and its verification path; and (vii) documents the surrounding platform: a binary-message handler, endpoint discovery, a session-hijacking tester, a blue-team defensive-validation suite, a process-isolated plugin system, an adaptive rate limiter, an HTTP resilience layer, and issue-tracker integrations. An honest account of the toolkit's limitations is also given, most notably the absence of a published empirical detection benchmark in the current repository.

II. Background

A. WebSockets and their attack surface

After the opening handshake, RFC 6455 [1] frames carry application data with no per-message HTTP semantics. Consequently, input-validation flaws that would otherwise appear on HTTP parameters (SQL injection, XSS, command injection, XXE, SSRF, NoSQL injection, path traversal) can reappear inside WebSocket messages, but reaching them requires establishing session state and matching the message schema. The handshake also introduces protocol-specific issues such as Cross-Site WebSocket Hijacking (CSWSH), in which insufficient `Origin` validation lets a malicious page open an authenticated socket on the victim's behalf [5]. Beyond injection, realtime applications are prone to *authorization* and *state* flaws: a message that is safe for one identity may leak data for another (cross-tenant exposure), and a state-changing action may be vulnerable to duplicate execution or stale-token reuse in a race window. WSHawk addresses the injection surface (scanner and payload subsystems), the authorization/state surface (attack services), and the protocol surface (defensive-validation module and session-hijacking tester).

B. Out-of-band testing

Blind vulnerabilities produce no on-channel signal. Out-of-band Application Security Testing (OAST) resolves this by injecting a payload that causes the target to contact an external collaborator; an observed callback confirms the flaw [6]. WSHawk integrates an OAST provider for blind XXE and SSRF.

C. Browser-assisted validation

String reflection is a weak XSS oracle: a payload echoed in a response may never execute. Rendering a candidate response inside an instrumented headless browser and observing script execution provides a stronger oracle. WSHawk uses Playwright-driven Chromium [4] for this purpose, an approach conceptually similar to interactive DOM-analysis tooling in the wider ecosystem [7].

D. Scoring and interchange

Findings are scored with CVSS v3.1 [8] and can be exported in the SARIF 2.1.0 interchange format [9] for ingestion by code-scanning platforms. Evidence-bundle integrity relies on Ed25519 signatures [10].

III. Related Work

A. WebSocket and realtime-application security

The WebSocket protocol is specified in RFC 6455 [1], and its deployment security has been examined both by practitioner standards and by the research community. The OWASP Web Security Testing Guide [11] and the OWASP Top Ten [12] enumerate the injection and access-control classes that WSHawk probes, while Schneider's analysis of Cross-Site WebSocket Hijacking [5] characterizes the `Origin`-validation weakness that WSHawk's defensive module tests. General-purpose interception suites such as OWASP ZAP [13] and Burp Suite [14] added WebSocket support primarily for manual message editing and replay, and injection-focused tools such as sqlmap [15] automate a single vulnerability class over HTTP. WSHawk differs from these in treating the socket as a stateful session and in coupling detection to a project-backed evidence trail.

B. Fuzzing and evolutionary input generation

WSHawk's payload subsystem is a domain-specific instance of feedback-guided fuzzing, a field surveyed comprehensively by Manès et al. [16]. The foundational observation that random and mutated inputs expose robustness failures dates to Miller et al. [17]; modern greybox fuzzers such as coverage-guided Markov-chain scheduling [18] formalize how effort should be redistributed toward productive inputs, which is analogous to WSHawk's per-category effectiveness scoring and its genetic reallocation of the payload population toward high-fitness individuals. Grammar-aware input generation [19] improves validity rates by respecting the input structure, mirroring WSHawk's context-aware generator that embeds payloads inside well-formed JSON/XML rather than emitting raw strings.

C. Stateful and protocol-aware testing

Because WebSocket sessions are stateful, WSHawk is more closely related to network-protocol fuzzers than to stateless web scanners. SNOOZE introduced explicit protocol state in fuzzing [20]; Pulsar inferred state machines for proprietary protocols to drive stateful black-box fuzzing [21]; and AFLNet combined greybox coverage feedback with protocol state for network services [22]. WSHawk's session state machine and its protocol-graph subsystem occupy the same design space, though they are used to guide authorized security probing rather than crash discovery. WSHawk's inference of message families, identifier fields, and framework-specific target packs

from captured traffic is related to automatic protocol reverse-engineering, exemplified by Discoverer's field inference from network traces [23] and Prospex's extraction of protocol state machines [24]; WSHawk performs a lighter-weight, heuristic form of this inference aimed at suggesting concrete attack templates.

D. Positioning

WSHawk's distinguishing engineering choices, as observed in the source, are: (i) treating the WebSocket connection as a stateful session with an explicit learning phase before active probing; (ii) combining a heuristic verifier with a sandboxed browser oracle for XSS; (iii) tying replay, authorization diffing, race testing, and subscription abuse to a single project-backed evidence store with signed, hash-chained export bundles; (iv) unifying WebSocket and HTTP workflows under the same project, identity, and variable-templating model; and (v) inferring the realtime protocol family from captured traffic and mapping it to concrete attack templates and playbooks. No claim of superiority over the systems above is made; the comparison is qualitative and scoped to what the repository demonstrates.

a) Formal model: Let Σ^* denote the set of finite payload strings and let $P \subseteq \Sigma^*$ be the current population with an associated fitness map $f : P \to [0,1]$. When a payload p is evaluated and receives an observed reward $r \in [0,1]$ (derived from the feedback signal, with interesting/error/reflected/delayed responses rewarded), its fitness is updated by the exponential moving average

$$f'(p) \;=\; \beta\, r \;+\; (1-\beta)\, f(p), \qquad \beta = 0.4, \tag{1}$$

and p is admitted to the hall of fame iff $r \geq \tau$ with $\tau = 0.7$. Selection is elitist: the parent set is

$$P_{\text{par}} \;=\; \operatorname*{top}_{k}(P, f), \qquad k = \max\Big(10,\; \Big\lfloor \tfrac{|P|}{2} \Big\rfloor\Big). \tag{2}$$

A *mutation operator* is a map $m : \Sigma^* \to \Sigma^*$ drawn uniformly from a finite set $\mathcal{M} = \{m_1, \ldots, m_9\}$ of character-level edits (swap, delete, insert, replace, case change, single-character encoding, segment duplication, segment reversal, null insertion). A *crossover operator* is a map $\chi : \Sigma^* \times \Sigma^* \to \Sigma^*$ drawn from $\mathcal{X} = \{\text{split}, \text{interleave}, \text{wrap}, \text{inject}\}$; for example the split operator, for parents a, b with midpoints $\mu_a = \lfloor |a|/2 \rfloor$ and $\mu_b = \lfloor |b|/2 \rfloor$, produces

$$\chi_{\text{split}}(a,b) \;=\; a_{[0:\mu_a]} \,\|\, b_{[\mu_b:]}, \tag{3}$$

where $\|$ is string concatenation. Given per-generation rates r_x (crossover) and r_m (mutation) and a draw $u \sim \mathcal{U}(0,1)$, each offspring c is produced by

$$c \;=\; \begin{cases} \chi(p_1, p_2), & u < r_x,\ |P_{\text{par}}| \geq 2, \\ m(p), & r_x \leq u < r_x + r_m, \\ \textsc{InjectBypass}(p), & \text{otherwise}, \end{cases} \tag{4}$$

with parents sampled uniformly from P_{par}. Offspring are accepted only if $\mathrm{md5}(c)$ is previously unseen, $ceq\varepsilon$, and $|c| < 2000$; after each generation the population is truncated to the N fittest individuals, bounding memory regardless of run length. The per-category effectiveness score used by the feedback loop is updated by an analogous EMA with rate $\alpha = 0.3$, and a category is retired once its score falls below 0.1 after at least five attempts.

IV. Problem Statement

The implementation is organized around four concrete problems that follow from the WebSocket setting:

1) **State preservation.** Reaching an injectable sink often requires connecting, authenticating, and subscribing in order. The tool must reproduce such sequences and keep the socket open across many probes.
2) **Schema-aware injection.** Raw payloads are frequently rejected by structural validation. The tool must learn the message format—including binary encodings—and embed injections in well-formed messages.
3) **Reliable oracles.** Asynchronous, blind, and reflection-only responses make naive detection error-prone. The tool must combine timing/error/reflection heuristics with out-of-band callbacks and sandboxed browser execution.
4) **Reproducible evidence.** Manual realtime testing is hard to reproduce and audit. The tool must persist identities, traffic, findings, and replay recipes, and must export them in a tamper-evident, redacted form.

V. System Design and Architecture

WSHawk v4 is a local, project-backed platform rather than a single scanner binary. According to the architecture guide and confirmed by the package layout, the runtime is split into service groups: a desktop operator interface (`desktop/`); a loopback bridge/daemon (`wshawk/gui_bridge.py`, `wshawk/daemon/`); transport, session, attack, protocol, store, and evidence layers under `wshawk/`; and a web penetration-testing package (`wshawk/web_pentest/`). The legacy command-line scanner remains available: the top-level `wshawk` and `wshawk-advanced` entry points are thin compatibility wrappers that delegate to `wshawk.legacy_core` and `wshawk.legacy_advanced_cli` respectively, as shown by `wshawk/__main__.py`.

Figure 1 shows the layered architecture. The daemon is deliberately split by responsibility into five route groups (Table XI): platform routes for projects, identities, replay, attacks, events, evidence and exports; transport routes for extension ingestion, DOM tooling, and interceptor operations; system routes for status, config, and pairing; scan routes for the WebSocket scan lifecycle; and web routes for the HTTP toolkit. The desktop application communicates with this daemon over loopback HTTP and Socket.IO. A Manifest V3 browser extension pairs with the daemon on `localhost` and ingests scoped WebSocket handshake context.

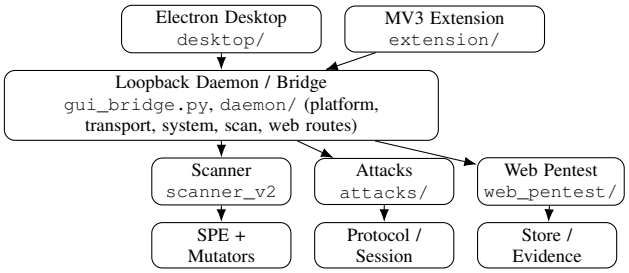

Fig. 1. Layered architecture of WSHawk v4. The desktop app and browser companion talk to a loopback daemon whose routes are split by responsibility; the daemon drives the scanner, attack services, and web-pentest toolbox, all backed by the protocol/session, store, and evidence layers.

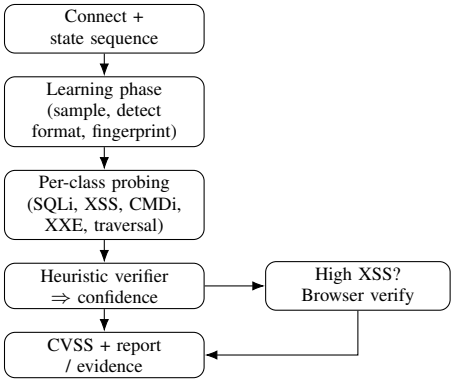

Fig. 2. Stateful scan pipeline (`WSHawkV2`). A passive learning phase precedes active per-class probing; only high-confidence XSS candidates are escalated to the browser oracle.

VI. Implementation

This section describes each subsystem in the order a reader would encounter it during an assessment. Declared dependencies in `pyproject.toml` include `websockets`, `playwright`, `aiohttp`, `PyYAML`, `flask`, `uvicorn`, `fastapi`, `python-socketio`, `dnspython`, `python-whois`, and `cryptography`. The `requirements.txt` file additionally lists `colorama`, `msgpack`, `cbor2`, `numpy`, and `scipy`; the binary handler uses `msgpack`/`cbor2` opportunistically (guarded by import checks), while the genetic evolver reviewed here uses only the standard `random` module, so any `numpy`/`scipy` usage is not present in the components analyzed.

A. Stateful scanning pipeline

The scanning engine is `wshawk/scanner_v2.py`, class `WSHawkV2`. It is constructed with a target URL, optional headers, an optional authentication sequence, a maximum request rate, and an optional event callback used to stream progress to the GUI. On construction it wires together the message analyzer, the vulnerability verifier, the server fingerprinter, the session state machine, an adaptive token-bucket rate limiter, the HTML reporter, the multi-format report exporter, the binary handler, and the three Smart Payload Evolution components.

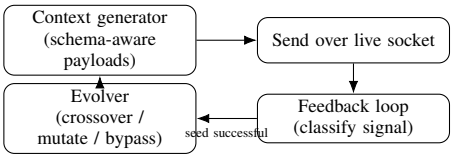

Fig. 3. Smart Payload Evolution. Context-aware payloads are sent, responses are classified by the feedback loop, and successful payloads are seeded into the genetic evolver whose offspring re-enter generation.

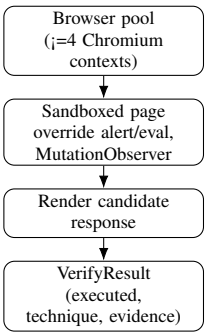

Fig. 4. Browser-assisted XSS validation (`dom_invader.py`). A pooled Chromium context renders the response in an instrumented sandbox to confirm actual script execution.

The scan proceeds in phases. The *learning phase* (`learning_phase`) keeps the socket open for a fixed duration (five seconds by default), collecting sample messages with a per-receive timeout. Collected samples are passed to the message analyzer to detect the wire format and injectable fields, to the server fingerprinter to infer language/framework/database, and (when Smart Payloads are enabled) to the context generator and the feedback-loop baseline. Only after learning do the active phases run: `test_sql_injection_v2`, `test_xss_v2`, `test_command_injection_v2`, `test_path_traversal_v2`, and `test_xxe_v2`. Each active phase injects payloads into a well-formed base message when the detected format is JSON, sends them over the live connection, awaits a bounded response, and asks the verifier to classify the result. Figure 2 depicts the flow, and Table III lists the classes and techniques.

Two design details are visible in the code and worth highlighting. First, timing safety: the engine uses `time.monotonic()` rather than wall-clock time, so timing-based detection is robust to system clock changes (a fix recorded in the changelog). Second, escalation: in `test_xss_v2`, only findings the heuristic verifier rates `HIGH` are escalated to the headless browser for execution confirmation; a confirmed execution upgrades the description to note sandboxed browser evidence and sets a `browser_verified` flag that later propagates into reports and CVSS treatment. Optional AI payload assistance (`ai_engine.py`) is gated behind a `scanner.features.ai_fuzzing` configuration flag that is off by default; when enabled, generated payloads are

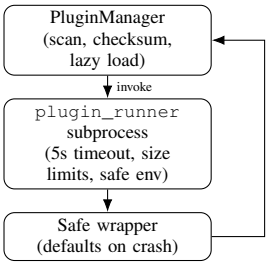

Fig. 5. Process-isolated plugin execution (`plugin_system.py`, `plugin_runner.py`).

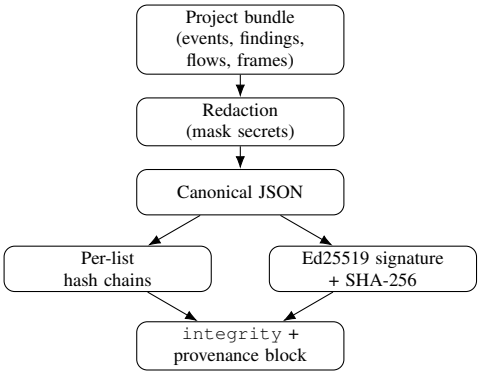

Fig. 6. Tamper-evident evidence export (`evidence/redaction.py`, `evidence/integrity.py`). Content tampering and reordering are both detectable at verification time.

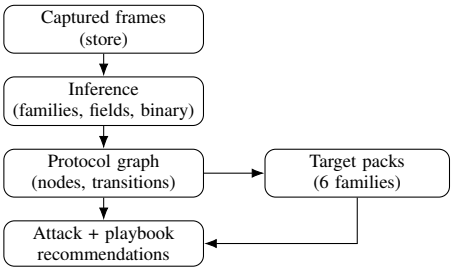

Fig. 7. Protocol subsystem (`wshawk/protocol/`). Captured frames are inferred into families and a graph, matched against six framework target packs, and mapped to attack and workflow-playbook recommendations.

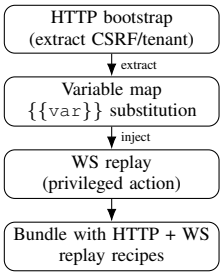

Fig. 8. Cross-protocol `ws_privilege_escalation` workflow (`attacks/workflows.py`): an HTTP bootstrap feeds extracted variables into a privileged WebSocket replay, producing a bundle with recipes spanning both protocols.

prepended to the built-in payload list for the relevant phase.

B. Message intelligence and injection

The message analyzer (`message_intelligence.py`) exposes a `MessageFormat` enumeration (JSON, XML, binary, Base64, plaintext, Protobuf, MessagePack) and detects the format of a sample by attempting JSON parsing first, then XML delimiters, then a Base64 pattern with a decode-and-reparse check, then a non-printable-character test for binary, defaulting to plaintext. For JSON it infers a schema by walking every message and recording, per dotted field path, the observed type, an occurrence count, and up to five sample values. Payload injection (`inject_payload_into_message`) places a payload into each learned string field by navigating the dotted path (including array indices), returning one mutated message per injectable field so that the structure around the injection remains valid.

C. Server fingerprinting

The server fingerprinter (`server_fingerprint.py`) accumulates responses and scores them against regular-expression signature banks for seven backend languages (Python, Node.js, Java, PHP, Ruby, Go, C#), nine frameworks (Express, Socket.IO, Django, Flask, Spring, Rails, SockJS, Tornado, FastAPI), and seven databases (MySQL, PostgreSQL, MongoDB, Redis, MSSQL, Oracle, SQLite), selecting the highest-scoring candidate in each category and computing an overall confidence as the ratio of matched to possible signatures. Crucially, `get_recommended_payloads` maps the detected stack to technology-specific payloads (for example MySQL `SLEEP(5)`, MSSQL `WAITFOR DELAY`, Flask/Jinja {{7*7}}), which the scanner prepends to the generic payload list for the SQL, NoSQL, command-injection, and template phases.

D. Session state machine

Authentication and subscription sequences are handled by `state_machine.py`. The `SessionStateMachine` models a session as a progression over the states DISCONNECTED, CONNECTED, AUTHENTICATING, AUTHENTICATED, SUBSCRIBED, READY, and ERROR. A sequence can be declared in YAML and executed with `execute_sequence`, which sends templated messages, waits for expected substrings, and applies inter-step delays. Message templates support `${VAR}` substitution over a supplied variable map, so captured tokens can be injected into later steps. The companion `AuthenticationFlow` provides helpers that construct JWT, basic, API-key, and session-based authentication messages, and `detect_auth_message` heuristically recognizes authentication fields (e.g., `token`, `auth`, `api_key`) in observed traffic.

E. Payload mutation engine

WSHawk contains two complementary payload subsystems. The first is the strategy-scored mutation engine in `payload_mutator.py`, class `PayloadMutator`. It defines eight mutation strategies (Table IV): encoding, case

variation, comment injection, whitespace substitution, concatenation, generic filter-bypass, tag breaking, and polyglot. Each strategy is a deterministic transformation family; for example, the encoding strategy emits URL, double-URL, decimal/hex HTML-entity, unicode, UTF-7, UTF-16, Base64, mixed, and overlong-UTF-8 variants of the base payload.

The engine is adaptive in two ways. It maintains a weighted score $s(\sigma)$ per strategy σ, initialized to 1.0; on a successful (unblocked) payload it multiplicatively boosts the score of every strategy the payload appears to use by a factor of 1.2 and records the successful pattern. It also learns from blocking signals: `learn_from_response` scans responses for WAF signatures (Cloudflare, Akamai, Imperva, F5, AWS WAF, and ModSecurity) and for generic filter indicators (e.g., "blocked", "forbidden", "policy violation"), recording detected filter types so future mutation can steer around them. A second, class-based mutator package (`wshawk/mutators/`) provides a `BaseMutator` interface with a `MutatorConfig` of feature toggles and concrete `EncodingMutator`, `CommentMutator`, `TagBreakMutator`, and `PolyglotMutator` implementations assembled by `create_default_mutators`.

F. Smart Payload Evolution

The second subsystem, referred to in the code and documentation as Smart Payload Evolution (SPE), is a three-stage adaptive pipeline in `wshawk/smart_payloads/`. Figure 3 shows the loop.

Context-aware generation. `ContextAwareGenerator` learns the target message format (JSON, XML, key/value, or text) from samples, recursively records field types, sample values, and simple constraints (length, and regex-inferred patterns such as email, URL, date, UUID, or numeric), and then emits payloads that match the learned structure. For JSON it builds a template from learned fields and injects an attack string into each string field individually and into all string fields at once; it additionally emits type-confusion payloads including boolean, null, empty-array, and a `__proto__` prototype-pollution object. Injection strings are drawn from a per-class database (SQLi, XSS, command injection, SSTI, NoSQL, and traversal) and are filtered against patterns previously marked as blocked.

Feedback classification. `FeedbackLoop` first establishes a baseline from at least three normal responses, recording average length and response time. For each probe it classifies the response into one of several signals—ERROR, BLOCKED, REFLECTED, DELAYED, DIFFERENT, or NORMAL—using regular-expression banks for database/stack-trace error strings and for WAF/block strings, substring reflection checks (including HTML-escaped variants), a timing test that flags delays when the ratio $\rho = t_r/t_0$ exceeds 3.0 (high confidence) or $1.5+\theta$ (lower confidence), and a size-anomaly test at 50% deviation from baseline. It tracks per-category effectiveness with an exponential moving average (learning rate $\alpha = 0.3$) and exposes `should_continue_category`, which stops testing a category once its effectiveness falls below 0.1 after at least five attempts. It can also generate encoding, case, whitespace, comment, and concatenation mutations of a payload that produced an interesting response.

Algorithm 1 One generation of the payload evolver (from `payload_evolver.py`).

parents $\leftarrow$ top $\max(10, |P|/2)$ payloads of P by f
$O \leftarrow \emptyset$; *attempts* $\leftarrow 0$
while $|O| <$ *count* **and** *attempts* $< 5 \cdot$ *count* **do**
  $r \leftarrow \text{uniform}(0, 1)$
  if $r < r_x$ **and** $|$*parents*$| \geq 2$ **then**
    $c \leftarrow$ CROSSOVER(p_1, p_2) for random parents p_1, p_2
  else if $r < r_x + r_m$ **then**
    $c \leftarrow$ MUTATE(random parent)
  else
    $c \leftarrow$ INJECTBYPASS(random parent)
  end if
  if $\text{md5}(c)$*otinseen* **and** $ceq\emptyset$ **and** $|c| < 2000$ **then**
    $O \leftarrow O \cup \{c\}$; $P[c] \leftarrow 0.5$; mark c seen
  end if
  attempts $\leftarrow$ *attempts* $+ 1$
end while
trim P to the N fittest payloads
return O

Genetic evolution. `PayloadEvolver` maintains a population P mapping each payload p to a fitness score $f(p) \in [0, 1]$; the scanner instantiates it with population size $N = 100$. Successful payloads are seeded and their fitness updated via an exponential moving average, $f'(p) = 0.4\,\text{score} + 0.6\,f(p)$, with any payload scoring ≥ 0.7 appended to a hall of fame. Each generation selects the top half of the population as parents (elitism) and produces offspring by one of three operators chosen by rate: crossover (split, interleave, wrap, or inject), mutation (nine character-level operators such as swap, delete, insert, replace, case change, single-character encoding, segment duplication, segment reversal, and null insertion), or targeted WAF-bypass injection (comment insertion, encoding wrap, null padding, concatenation break, or newline injection). Offspring are MD5-deduplicated against all previously seen payloads and the population is trimmed back to N by fitness. Algorithm 1 summarizes one generation.

G. Vulnerability verification

The heuristic oracle is `vulnerability_verifier.py`, class `VulnerabilityVerifier`, which returns an (*is_vulnerable*, *confidence*, *description*) triple with a four-level `ConfidenceLevel` (LOW/MEDIUM/HIGH/CRITICAL). SQL injection is confirmed by a bank of database-specific error signatures (MySQL, PostgreSQL, MSSQL, Oracle, SQLite, and generic syntax errors); mere payload reflection yields only LOW confidence. XSS verification performs context analysis, distinguishing a payload inside a `<script>` tag (CRITICAL), inside an event handler (HIGH), inside an attribute or element content (MEDIUM), and recognizing HTML/unicode-escaped reflections as non-exploitable. Command injection matches

shell-error and id-output patterns (uid=...gid= yields CRITICAL). Path traversal matches file-content markers such as the /etc/passwd prefix, win.ini sections, and private-key headers. The module also provides time-based oracles (verify_sql_timing, verify_command_timing) that send SLEEP/WAITFOR and sleep payloads and compare elapsed time against a threshold.

H. Browser-assisted validation

The browser oracle is wshawk/dom_invader.py. A BrowserPool launches headless Chromium through Playwright with hardening flags and maintains up to four reusable browser contexts, avoiding a cold start per verification; contexts are cleared (pages closed, cookies cleared) on release for isolation. XSSVerifier.verify renders a candidate response inside a sandboxed page whose instrumentation overrides alert, confirm, prompt, and eval, installs a MutationObserver to count injected nodes and detect injected script tags, and exposes a console beacon. The method returns a structured VerifyResult recording whether execution occurred, the evidence string, the inferred technique (reflected, DOM-based, or mutation), any dialog message, DOM mutation count, and elapsed time. A batch_verify variant confirms many results concurrently under an asyncio.Semaphore (default concurrency three). Figure 4 shows this flow.

The same module implements AuthFlowRecorder, which opens a *visible* browser so an operator can perform an SSO/OAuth login while the recorder captures cookies, localStorage, and tokens seen in responses and Authorization headers (matched against a bank of success patterns); the recorded flow can later be *replayed* headlessly to mint fresh session material for long fuzzing runs. An older headless_xss_verifier.py remains as an import-guarded fallback within the scanner.

I. Out-of-band testing

The OAST provider (oast_provider.py) can register with the public interact.sh/oast.fun service—obtaining a correlation identifier and unique subdomain, polling for interactions, and deregistering on shutdown—or fall back to a DNS-only domain when registration fails. It generates OAST payloads for XXE (including parameter-entity and file variants), SSRF, and command execution (curl/wget/ping), and matches received interactions by test identifier. A bundled SimpleOASTServer provides a local aiohttp callback listener; in the reviewed scanner path the XXE phase instantiates the provider against this local server rather than the public service, which is appropriate for self-contained and lab testing.

J. Binary message handling

binary_handler.py adds support for non-text frames through the BinaryMessageHandler. It auto-detects format from raw bytes via magic bytes and heuristics—zlib/gzip compression, Avro, BSON (length-prefix and trailing-null check), Protocol Buffers (wire-type/field-number heuristic), FlatBuffers, MessagePack, CBOR, and a Shannon-entropy test that classifies high-entropy payloads (> 7.5 bits/byte) as likely encrypted. It reassembles fragmented frames per connection, parses protobuf and BSON heuristically (extracting varint, fixed, and length-delimited fields), decompresses and recursively parses compressed frames, and produces a full analysis (hex dump, MD5, SHA-256, entropy, printable and null ratios, and a list of injectable string fields). generate_binary_payloads builds attack frames by raw text injection (append/prepend/middle), format-specific mutations (re-packing MessagePack maps with injected values or numeric overflow values; rebuilding protobuf fields with injected length-delimited values; recompressing mutated deflate content, including a compression "bomb"), and boundary payloads (empty, single-null, truncated, oversized, reversed, bit-flipped, null-padded). The binary-analysis metadata it emits is consumed downstream by the protocol subsystem (Section VI-L).

K. Attack framework

The wshawk/attacks/ package exposes parallel HTTP and WebSocket services (Table V) that all write into the project store. Shared helpers in common.py and http_common.py build cookie/identity headers, serialize/normalize messages, and summarize authorization diffs; identity-backed handshake headers are constructed by build_ws_headers and merge_http_identity.

Replay. replay_websocket_message opens a socket with optional identity-backed headers, drains prelude frames, sends a single frame, and records status, timing, response preview, and the header keys used. WebSocketReplayService wraps this in a project-aware attack run that opens a stored WS connection, records outbound and inbound frames, and closes the run with a summary. HTTPReplayService is the HTTP analogue and additionally can build a reusable request *template* from a stored HTTP flow, with baseline status/length metadata for later comparison, and render templates with {{variable}} substitution.

Authorization diffing. authz_diff_websocket replays the same frame across a list of identities and summarizes behaviour drift; WebSocketAuthzDiffService and HTTPAuthzDiffService persist each identity's connection and frames so that cross-role and cross-tenant differences are auditable.

Race testing. The WebSocket race service (WebSocketRaceService) launches concurrency coroutines per wave across waves waves; all attempts in a wave block on a shared asyncio.Event barrier so they are released as close to simultaneously as possible, with optional per-attempt stagger. Each attempt opens its own socket, optionally sends setup payloads, sends the target payload, and records status, timing, and response preview. The service then classifies the run: it counts distinct behavioural groups and effective successes, and flags a *suspicious race*

Algorithm 2 Race-wave barrier (from `attacks/race.py`).

for $w \leftarrow 1$ **to** *waves* **do**
  barrier $\leftarrow$ new `asyncio.Event`
  spawn *concurrency* attempts, each awaiting *barrier*
  barrier.SET() {release all attempts together}
  gather attempt results
  if $w <$ *waves* **and** *wave_delay* > 0 **then**
    sleep *wave_delay*
  end if
end for
suspicious $\leftarrow$ (*accepted* > 1) $\wedge$ (state-changing mode $\vee$ behaviour groups > 1)

window when more than one attempt succeeds in a state-changing mode (duplicate action, replay-before-invalidation, or stale-token window). A recommended severity of HIGH, MEDIUM, or INFO is derived from whether later waves also succeed. `HTTPRaceService` mirrors this over HTTP request templates, grouping behaviour by a SHA-256 of status/preview/error and treating any < 400 status as success. Algorithm 2 sketches the wave-barrier structure.

Subscription abuse. `WebSocketSubscriptionAbuseService` targets realtime targets where access depends on a topic/tenant/object field. `generate_subscription_mutations` walks a JSON payload, identifies fields whose names hint at channels, tenants, or identifiers, and produces candidate values (e.g., `admin`, `*`, `global`, adjacent integer IDs). Each mutation is replayed against a per-identity baseline; a mutation is flagged suspicious when it is accepted and either changes behaviour relative to baseline or uses a sensitive candidate value, and severity is escalated when a sensitive candidate is accepted.

Workflow engine. `WorkflowExecutionService` composes the above into multi-step, project-backed workflows. Steps are typed (`http`, `ws`, `http_authz_diff`, `http_race`, `ws_authz_diff`, `sleep`, `set`, `note`) and share a variable map; values propagate via {{`variable`}} substitution, and per-step `extract` rules pull follow-on values from a response body, header, or JSON path via regex or dotted path. Six built-in playbooks (Table VI) encode common cross-protocol chains such as login bootstrap, CSRF/session capture, WS privilege escalation, stale-token reuse, and tenant hopping. The red-team benchmark test exercises a `ws_privilege_escalation` playbook that bootstraps an HTTP request to obtain a CSRF/tenant value and then replays a WebSocket subscription with it, producing a bundle whose replay recipes span both HTTP and WebSocket.

L. Protocol subsystem

The `wshawk/protocol/` package turns captured traffic into an actionable protocol map. `ProtocolInferenceService` (`inference.py`) normalizes frames, learns the format via the message analyzer, and derives per-field profiles (types, sample values, nested keys), message families keyed by action/type/event, and sets of auth fields, identifier fields, subscription fields, server-issued fields (from inbound frames), and observed binary formats. `ProtocolTemplateService` (`templates.py`) builds reusable, editable message templates per family, marking id/tenant/channel/token fields as editable and suggesting variable names.

`ProtocolGraphService` (`graph.py`) assembles a graph whose nodes are correlation groups, connections, and message families and whose edges capture frame direction and family-to-family transitions. From this it derives attack recommendations (subscription abuse, identifier tampering, authorization diffing, workflow chaining, race windows) and maps them to concrete workflow playbook candidates. Six target-pack adapters (`target_packs.py`) fingerprint specific realtime protocols and surface protocol-aware attack templates (Table VII): GraphQL-WS (subscribe/next/complete, operation and variable extraction), Phoenix Channels (topic/event/payload/join_ref), ActionCable (command/identifier with embedded identifier JSON), SignalR (record-separator framing, invocation IDs), Socket.IO/Engine.IO (numeric prefixes, namespaces, event arrays), a generic structured-JSON fallback, and a binary adapter that consumes the binary-analysis metadata from Section VI-J. The red-team benchmark test asserts that GraphQL traffic yields the `graphql_ws` pack and a `ws_privilege_escalation` playbook candidate and that a protobuf frame yields a `binary_realtime` pack with injectable identifiers.

M. WebSocket endpoint discovery

`ws_discovery.py` finds sockets behind an HTTP target through three phases: probing a list of common WebSocket paths with HTTP upgrade headers and treating an HTTP 101 as a confirmed endpoint (with lower confidence for upgrade-header or error-message signals); parsing HTML for `ws://`/`wss://` URLs and following same-domain links to a bounded depth; and analyzing linked JavaScript for `WebSocket` constructors, string concatenation, and library signatures (Socket.IO, SockJS, SignalR, GraphQL subscriptions). Results are deduplicated and ranked by a four-level confidence scale (CONFIRMED/HIGH/MEDIUM/LOW). When available, discovery runs over the resilient session (Section VI-P).

N. Session-hijacking tester

`session_hijacking_tester.py` runs six focused session-security tests (Table VIII): token reuse after session termination, subscription spoofing against privileged channels, user impersonation via client-supplied identifiers, channel-boundary violations, session fixation with an attacker-chosen session ID, and privilege escalation via role/permission fields. Each test returns a `SessionTestResult` with a confidence level, evidence, a fixed CVSS score, and a remediation string, and authentication is configurable rather than hard-coded (a fix noted in the changelog).

O. Defensive validation (blue team)

`defensive_validation.py` and `wss_security_validator.py` provide a blue-team suite driven by the `wshawk-defensive` CLI. It validates DNS-exfiltration egress filtering (via XXE- and SSRF-triggered callbacks), anti-bot effectiveness (a basic headless Playwright browser and an evasion variant that patches `navigator.webdriver` and related properties), CSWSH `Origin` enforcement (attempting connections with malicious origins loaded from `payloads/malicious_origins.txt`) and a CSRF-token requirement check, and—for `wss://` targets—TLS posture: it probes for deprecated protocol versions (SSLv2/3, TLS 1.0/1.1), weak cipher suites and missing forward secrecy, certificate expiry/self-signing, chain integrity, and secure renegotiation, attaching CVSS scores to findings.

P. Adaptive rate limiting and resilience

`rate_limiter.py` implements `TokenBucketRateLimiter`, a token bucket with refill rate λ and capacity B plus a multiplicative `backoff_multiplier` b that scales the effective rate. `acquire`/`done` bracket each request and enforce a maximum in-flight concurrency. Two feedback paths adjust b: `report_response_time` halves the rate after three consecutive slow windows (mean latency above threshold) and gradually recovers it ($\times 1.1$, capped at 1.0) when responses are fast; `report_server_feedback` inspects WebSocket close codes (treating 1013 as a strong rate-limit signal and 1008/1011 as soft signals requiring repetition), close reasons, and message text for throttling keywords, halving the rate on a strong signal. Reductions and increases are separately rate-limited with cooldowns.

Complementing this, `resilience.py` provides an HTTP reliability layer used by discovery and the integrations: a `RetryConfig` with exponential backoff that honours HTTP 429 `Retry-After` headers and retries on 5xx and connection errors; a three-state `CircuitBreaker` (closed/open/half-open) that trips after a failure threshold and self-tests after a reset timeout; and a `ResilientSession` wrapper around `aiohttp` that combines both.

Q. Vulnerability scoring and reporting

CVSS scoring is implemented in `cvss_calculator.py`, which computes a v3.1 base score from per-vulnerability-type base metrics. The attack vector is fixed to NETWORK and scope to UNCHANGED; impact metrics vary by class (e.g., SQL injection, command injection, and NoSQL injection map to high confidentiality/integrity impact, while XSS defaults to low impact and REQUIRED user interaction, escalating to high impact when the finding is browser-verified). The implementation notes in comments that it is a simplified CVSS v3.1 computation.

Report generation spans several modules (Table XIV). The multi-format exporter (`report_exporter.py`) supports JSON, CSV, and SARIF 2.1.0, enriching each finding with a CVSS score, vector, and severity, assigning `WSHAWK-nnnn` identifiers, and mapping severities to SARIF levels. The enhanced HTML reporter (`enhanced_reporter.py`) produces styled HTML. The evidence exporters (`evidence/exporters.py`) render project bundles in JSON, Markdown, and HTML. The desktop web-pentest package includes its own report generator (`web_pentest/report_gen.py`). PDF output is described in the documentation and desktop guide; the specific PDF generation backend was not confirmed in the source modules reviewed, so PDF is reported as a documented format rather than a code-verified one.

R. Persistence layer

All project state is persisted by `db_manager.py` in a single SQLite database that unifies the legacy scanner with the v4 platform. The database location is resolved defensively across development, packaged, and sandboxed environments by probing a series of candidate directories (`WSHAWK_DATA_DIR`, `~/.wshawk`, the working directory, and `/tmp`) for writability. Connections are opened in write-ahead-logging mode with foreign keys enforced.

The schema comprises five tables. A legacy `scans` table stores historical scan reports (target, options, per-severity counts, message counts, timings, and a findings JSON blob) and supports a scan-to-scan diff that reports fixed and newly introduced findings. The platform tables are `projects` (unique name, target URL, metadata), `identities` (project-scoped alias with cookies, headers, tokens, browser storage, and notes), `traffic_events` (typed events with direction, connection identifier, target, and payload), and `evidence` (title, category, severity, payload, and an optional link to a triggering event). Foreign keys cascade on project deletion, and indexes are declared for the common access paths (status, recency, and project scoping).

Two security properties are visible in the code. First, sensitive columns—identity cookies/headers/tokens/storage and project/event/evidence payloads—are written through a `SensitiveDataCipher`, and identity notes are encrypted at rest; the row-to-object helpers transparently decrypt on read. Second, the legacy `update` method validates every column name against a fixed allow-list (`SCAN_COLUMNS`) before interpolating it into SQL, preventing column-name injection. The higher-level `ProjectStore` builds on these primitives to expose the target, connection, frame, attack-run, and correlation-group operations used by the attack and evidence layers.

S. Evidence store and tamper-evident bundles

Project state is persisted through `db_manager.py` and the `wshawk/store/` package (`ProjectStore`), which records targets, identities, findings, notes, events, HTTP flows, WebSocket connections and frames, and attack runs. The `wshawk/evidence/` package builds bundles (`bundles.py`), constructs timelines (`timeline.py`), redacts sensitive material (`redaction.py`), exports

(exporters.py), and attaches integrity metadata (integrity.py).

TimelineService materializes the store into a project-centric timeline: targets, HTTP flows, WS connections and frames, browser artifacts, attack runs, findings, correlation groups, and—usefully for reproduction—*replay recipes*, including generated curl commands for HTTP flows and per-connection outbound-frame lists for WebSocket connections, all passed through the redactor. It also builds correlation chains that tie a correlation identifier to its attack runs and findings. EvidenceBundleBuilder assembles the exportable bundle and additionally *derives* evidence and findings from replay events—for example detecting cross-tenant invoice exposure, cross-tenant team-message leakage, or approval-token replay by comparing an identity's alias tenants against the tenant in the response—so that successful abuse is captured as first-class evidence.

The redaction.py module masks secrets before export: it detects secret-bearing header/field names (authorization, cookie, token, session, JWT, secret, API key, password, CSRF/XSRF), masks bearer tokens and cookie values while preserving structure, and rewrites long tokens and secret-looking query/JSON values, recursing through nested structures.

The EvidenceIntegrityService then canonicalizes the bundle to sorted-key, separator-normalized JSON and computes an Ed25519 signature over it using a key held in a SecretStore; it also computes a linear hash chain over each ordered list (events, evidence, findings, HTTP flows, WebSocket connections, and frames), where each link hashes the previous root together with the current item. The attached integrity block records the scheme (ed25519-sha256), a key identifier, the public key, the content SHA-256, the signature, and the chain roots, alongside a provenance block (tool, version, operator, hostname, platform). The verify method recomputes the canonical hash, verifies the signature, and recomputes the chain roots, so any modification to the bundle contents or ordering is detectable. Figure 6 illustrates this, and Table XV lists the integrity fields.

T. Notification and tracker integrations

The wshawk/integrations/ package contains three connectors, each configurable from environment variables and each using the resilient session and a per-service circuit breaker (Section VI-P) when available, falling back to raw aiohttp otherwise.

Jira (jira_connector.py) maps WSHawk severities to Jira priorities, filters findings by a configurable minimum severity, and builds a wiki-markup description containing a property table, the payload and response in code blocks, remediation, and reproduction steps. It creates issues either individually or through the bulk API, with a graceful fall-back to individual creation if the bulk call fails.

DefectDojo (defectdojo.py) targets the DefectDojo vulnerability management platform. It resolves (or creates) a product, auto-creates a CI/CD-typed engagement, converts findings to DefectDojo's schema—including a per-class CWE mapping (SQLi→89, XSS→79, command injection→78, path traversal→22, XXE→611, NoSQL→943, SSRF→918, SSTI→1336, open redirect→601, and CSRF/CSWSH→352)—and uploads them through the multipart import-scan endpoint with the CVSS vector and score attached.

Webhook (webhook.py) delivers scan summaries to Slack (Block Kit), Discord (embeds with severity-based colour), Microsoft Teams (MessageCard with facts), or a generic JSON endpoint, honouring a notify_on mode and a minimum severity.

a) AI-assisted payload generation.: The optional AI engine (ai_engine.py), gated behind the scanner.features.ai_fuzzing flag, generates context-aware payloads from the observed message samples. It supports a local Ollama endpoint and OpenAI-compatible APIs, builds a role-primed prompt that requests a raw JSON array of payloads tailored to the detected message format and target vulnerability class, and defensively parses the model output—stripping markdown fences and falling back to line splitting if JSON parsing fails. Generated payloads are prepended to the built-in payload list for the relevant scanning phase rather than replacing it.

U. Plugin architecture

plugin_system.py defines an abstract PluginBase and three extension points: PayloadPlugin (custom payload packs), DetectorPlugin (custom detectors returning a (*is_vulnerable*, *confidence*, *description*) triple), and ProtocolPlugin (custom message handlers). The PluginManager scans a plugin directory, computes a SHA-256 checksum per plugin file, and lazily loads plugins on first use. Security controls are explicit in the code: plugins are executed in a separate process through wshawk.plugin_runner with a five-second timeout, a 2 MB maximum plugin file size, a 128 KB maximum request size, and a sanitized environment; method calls are wrapped so that a crashing plugin returns a safe default rather than aborting the scan. Metadata validation enforces semantic-version formatting and a minimum-version compatibility check, and duplicate registration is rejected unless an override policy is set. Payload retrieval is memoized with an LRU cache. The plugin isolation behaviour is covered by tests/test_plugin_isolation.py. Figure 5 sketches the isolation path.

V. Configuration

config.py implements a hierarchical YAML configuration (WSHawkConfig) with a documented precedence of CLI arguments, environment variables, config file, and defaults. It searches ./wshawk.yaml, ./wshawk.yml, ~/.wshawk/config.yaml, and /etc/wshawk/config.yaml, deep-merges the file over defaults, and applies a fixed map of environment overrides with type coercion. Secret references are resolved lazily: env:VAR reads an environment variable, file:/path

reads a file, and `secret:ns:key` resolves through the platform `SecretStore`. Sections cover `scanner` (rate limit, timeout, learning duration, payload count, SSL verification, and feature toggles for Playwright, OAST, binary analysis, smart payloads, and AI fuzzing), `ai`, `reporting`, `integrations`, and `web`; `scanner.verify_ssl` defaults to true.

W. Desktop application and daemon

The desktop application (`desktop/`) is an Electron front end (`index.js`, `preload.js`, `renderer.js`) with feature modules for the API bridge, attacks, evidence, identities, protocol, traffic, team mode, and a large web-pentest module. It wraps the Python sidecar and, per the desktop guide and security model, enables Electron renderer isolation and sandboxing and treats the bridge as loopback-only. Three operating modes (Standard, Advanced, Web Pentest) expose progressively more tooling, including a frame-by-frame WebSocket interceptor, a payload blaster, an endpoint map, an auth builder, and an evidence vault. Packaged builds for Linux, Windows, and macOS are produced via `electron-builder`, and the Python sidecar is bundled via a PyInstaller spec (`wshawk-bridge.spec`). Desktop features are enumerated in Table XVIII.

X. Command-line interfaces

Four console entry points are declared in `pyproject.toml` (Table X): `wshawk` (compatibility scanner CLI), `wshawk-interactive` (interactive terminal workflow), `wshawk-advanced` (compatibility advanced CLI), and `wshawk-defensive` (defensive validation CLI). As noted in Section V, `wshawk` and `wshawk-advanced` delegate to legacy modules.

VII. Web Penetration-Testing Tool Design

While WebSocket testing is the core focus, the desktop Web Pentest mode is backed by a substantial HTTP toolbox in `wshawk/web_pentest/`. Each tool is an `asyncio`-based class that either issues requests directly through `aiohttp` or, when a project is active, routes them through the shared `WSHawkHTTPProxy` so that every request is recorded as a project flow with a correlation identifier. This section documents the design of the most substantial tools; all details are drawn directly from their modules.

A. Crawler

The crawler (`crawler.py`) is a breadth-first spider built on a custom `HTMLParser` subclass, `_LinkExtractor`, that pulls anchors, forms (with their method and input fields), script sources, CSRF tokens from `<meta>` tags, and media/link references from a page. Before crawling, it probes four sensitive paths—`/robots.txt`, `/sitemap.xml`, `/.git/config`, and `/.env`—and seeds the queue with any `Allow`/`Disallow` entries and sitemap URLs it finds. The BFS loop is bounded by `max_depth` and `max_pages`, honours a `same-host` or `same-domain` scope, throttles between requests, and runs batches of up to ten fetches concurrently under a semaphore. A regular expression additionally harvests inline API endpoints (paths containing `/api/`, `/v1/`, `/graphql`, `/ws`, and similar) from HTML and JavaScript. The result aggregates pages, forms, scripts, API endpoints, sensitive files, CSRF tokens, and per-fetch errors.

B. HTTP fuzzer

The fuzzer (`fuzzer.py`) substitutes a `$FUZZ$` marker in the URL, body, or headers with each wordlist entry, applying an optional encoder (Base64, URL, hex, or JSON-string). Wordlists are either built-in (`common`, `sqli`, `xss`), loaded from the payload files by a name-to-file map, or supplied from disk under a path-traversal-guarded loader capped at 50,000 entries. Its key design feature is a *baseline-differential* detector: it first sends an invalid canary value to record baseline status, length, cookies, and response time, and then flags each fuzzed response as interesting when it matches a grep regular expression, exhibits an authentication-bypass signal (an unexpected redirect, a new cookie, or a response-length delta above 500 bytes), shows a time delay more than four seconds over baseline, reflects an XSS payload, or contains LFI/command-output markers (`root:x:0:0:`, `uid=...gid=`, and similar).

C. Directory scanner

The directory scanner (`dir_scanner.py`) brute-forces paths from a default or custom wordlist, appending extension permutations to words that do not already carry one. Two design details stand out. First, a *soft-404 calibration* phase issues three random probe paths and records response fingerprints (status, normalized content type, length, and a SHA-256 of the first 128 KB of the body); a fingerprint seen at least twice becomes a fallback signature, and matching brute-force hits are suppressed to control single-page-application noise. Second, a *variant-grouping* pass collapses near-identical hits such as `/api`, `/api.php`, and `/api.js` into a single representative finding by comparing the parent path, stem, status, content type, and either fingerprint identity or a length-similarity test (within 64 bytes or 5%). Recursive scanning re-enqueues discovered directories.

D. SSRF prober

The SSRF prober (`ssrf_prober.py`) auto-detects URL-like parameters (either by value shape or by a list of ∼30 parameter names such as `url`, `redirect`, `callback`, `next`) and injects a payload set organized into five categories: cloud metadata endpoints for AWS, GCP, Azure, Alibaba, and Oracle; internal-network addresses including octal/decimal/hex encodings of `127.0.0.1`; protocol smuggling (`file://`, `dict://`, `gopher://`, `ftp://`); DNS rebinding hosts; and parser-confusion bypasses (`@`-authority, fragment, and CRLF tricks). Responses are matched against

indicator banks for cloud-metadata fields, internal-service output (/etc/passwd, Redis redis_version), and internal-address reflection; a hit is rated HIGH, while a bare 200 with a non-trivial body is rated LOW.

E. Open-redirect scanner

The redirect hunter (redirect_scanner.py) tests redirect-like parameters (drawn from a set of ∼30 names) with 25+ bypass payloads spanning absolute and protocol-relative URLs, backslash tricks, single/double URL-encoding, unicode-dot and @-authority confusion, path-escape variants, CRLF header injection, and javascript:/data: URIs. It classifies a finding by inspecting the response for an external Location header (HTTP 3xx, HIGH), a meta refresh, or a JavaScript redirect via four sink patterns (window.location, location.href, location.replace, window.open), with a host-comparison helper deciding whether the destination is off-site.

F. CORS tester

The CORS tester (cors_tester.py) sends a baseline request and then six crafted Origin values—arbitrary origin, null, subdomain-suffix, domain-prefix injection, subdomain reflection, and HTTP downgrade—and flags a misconfiguration whenever the Access-Control-Allow-Origin response reflects the sent origin or a wildcard. It specifically escalates severity to HIGH when a wildcard or reflected origin is combined with Access-Control-Allow-Credentials: true, and reduces the six findings to a single HIGH/MEDIUM/LOW/SAFE risk score.

G. Prototype-pollution tester

The prototype-pollution tester (proto_polluter.py) injects __proto__ and constructor.prototype payloads through two vectors: query parameters (bracket and dotted notations, filter-bypass and URL-encoded variants) and JSON bodies (deep-merge objects that attempt to set isAdmin, role, or a canary key). It records a baseline response and flags pollution when a canary indicator appears in the response (for example a reflected wshawk_pp_test value or an isAdmin:true escalation) or when the response length diverges significantly from baseline.

H. WAF detector

The WAF detector (waf_detector.py) fingerprints 15 firewalls from a signature database keyed on response headers, cookies, and body patterns (Cloudflare, AWS WAF, Akamai, ModSecurity, Sucuri, Imperva/Incapsula, F5 BIG-IP ASM, Barracuda, FortiWeb, Wordfence, DenyAll, Citrix NetScaler, Radware, StackPath, Varnish). It operates in two phases: a passive phase that matches a normal response, and an active phase that sends four malicious probe strings (XSS, SQLi, traversal, command injection) and treats a 403/406/429/503 response as a block signal while re-matching signatures on the block page.

VIII. Experimental Methodology

The repository's validation strategy is engineering-oriented and local. Two artifacts constitute it. First, a unit and integration test suite under tests/ (approximately 28 test modules in the reviewed tree) exercises the binary handler, CVSS calculator, platform database and routes, HTTP attack services, integrations, message intelligence, offensive attacks, the payload mutator, the plugin isolation path, the project store, protocol and export paths, report export, the secret store, server fingerprinting, session services, the vulnerability verifier, the WAF detector, and the web pentest platform. The changelog additionally references a suite of "90+ unit and integration tests."

Second, three local validation labs under validation/ provide repeatable regression targets: full_stack_realtime_saas, socketio_saas, and graphql_subscriptions_lab. Each lab ships a runnable application, a scenario definition, and an expected-baseline JSON file under validation/expected/; a harness (validation/run_validation.py) runs the labs and writes artifacts. The red-team benchmark test (tests/test_redteam_lab_benchmarks.py) asserts, for example, that a GraphQL subscription connection yields the graphql_ws target pack and a ws_privilege_escalation playbook candidate, that a cross-protocol workflow produces a bundle with both HTTP and WebSocket replay recipes, and that a binary protobuf frame surfaces a binary_realtime pack with injectable identifiers.

Crucially, these artifacts validate *behavioural correctness and reproducibility*, not detection accuracy against a labelled vulnerability corpus. The repository does not contain a benchmark measuring true/false positive rates, throughput comparisons, or head-to-head results against other scanners. Consistent with the source, it is therefore stated: the current implementation focuses on engineering validation, and a comprehensive empirical evaluation remains future work.

IX. Worked Example: GraphQL Subscription Lab

To make the end-to-end workflow concrete, this section traces the graphql_subscriptions_lab validation lab, whose intentionally vulnerable ASGI application and scenario are shipped in the repository. The lab exercises GraphQL-over-WebSocket testing using the graphql-transport-ws subprotocol and defines four seeded accounts across two tenants (tenant-alpha and tenant-beta); the intended vulnerabilities, per the lab README, are cross-tenant subscription exposure and privileged refund replay via a leaked approval token.

The scenario (scenario.py) proceeds as follows, authenticating as alice (a tenant-alpha user) and then operating against tenant-beta resources:

1) An HTTP POST /api/auth/login obtains a bearer token.

2) A WebSocket connection is opened and a `connection_init` frame carries the bearer token; the server replies `connection_ack`.
3) A `whoami` query confirms the acting identity.
4) An `invoiceUpdates` subscription for `inv-beta-9001` returns a foreign-tenant invoice, including its `approval_token`.
5) A `tenantMessages` subscription for `tenant-beta` returns foreign-tenant message rows.
6) An `approveRefund` mutation *without* a token is rejected ("Approval requires…").
7) The same mutation *with* the leaked `approve-beta-9001` token succeeds, and the response reports `approval_token_reused: true` for `tenant-beta`.

The scenario encodes these as five boolean checks (`graphql_acknowledged`, the two cross-tenant subscription checks, the token-less denial, and the token-replay success). The validation harness (`run_validation.py`) runs the scenario against a live in-process ASGI server, compares the result to the expected baseline under `validation/expected/`, and writes `result.json`, `evaluation.json`, and a combined `bundle.json` per lab, emitting an overall pass/fail.

This lab also demonstrates the protocol subsystem end to end. When the same captured frames pass through `ProtocolGraphService`, the `graphql-transport-ws` subprotocol and the `subscribe`/`next`/`complete` message types cause the GraphQL target-pack adapter to fire, yielding the `graphql_ws` pack with variable- and subscription-id attack templates and a `ws_privilege_escalation` playbook candidate; the corresponding assertions are made in `tests/test_redteam_lab_benchmarks.py`. The leaked-token replay is precisely the pattern the evidence builder (Section VI-S) recognizes when it derives a `websocket_token_replay` finding from replay events, so a single lab run exercises the capture, inference, attack-recommendation, and evidence-derivation path together.

Listing 1. Crossover operators from `smart_payloads/payload_evolver.py` (abridged).

```
def _crossover(self, p1, p2):
    strategy = random.choice(
        ['split', 'interleave', 'wrap', 'inject'])
    if strategy == 'split':
        m1, m2 = len(p1)//2, len(p2)//2
        return (p1[:m1] + p2[m2:] if random.random()
     < 0.5
                else p2[:m2] + p1[m1:])
    if strategy == 'interleave':
        k = random.randint(2, 8); out = []
        for i in range(0, max(len(p1), len(p2)), k):
            out.append(p1[i:i+k] if i % (2*k) < k
    else p2[i:i+k])
        return ''.join(out)
    if strategy == 'wrap' and len(p1) > 4:
        m = len(p1)//2
        return p1[:m] + p2 + p1[m:]
    # inject: splice p2 at a random offset in p1
    pos = random.randint(0, len(p1))
    return p1[:pos] + p2 + p1[pos:]
```

Listing 2. Per-list hash chain from `evidence/integrity.py`. Each link folds the previous root with the next item, so any reorder or edit changes the root.

```
@classmethod
def _hash_chain(cls, items):
    previous = ""
    count = 0
    for item in items:
        previous = cls._sha256_hex(
            {"previous": previous, "item": item})
        count += 1
    return {"count": count, "root": previous}
```

Listing 3. SignalR record-separator parsing from `protocol/target_packs.py`. SignalR frames are split on the 0x1e record separator before per-record JSON parsing.

```
_RECORD_SEPARATOR = "\x1e"

@classmethod
def _parse_records(cls, payload_text):
    records = []
    for chunk in payload_text.split(cls.
    _RECORD_SEPARATOR):
        if not chunk:
            continue
        parsed = cls._json_loads(chunk)
        if isinstance(parsed, dict):
            records.append(parsed)
    return records
```

X. Implementation Analysis

Several properties follow directly from the implementation. The separation of a passive learning phase from active phases means the tool adapts its payloads to the observed schema before spending its request budget, which is consistent with the stated goal of bypassing structural validation (Sections VI-B, VI-F). The layering of oracles—heuristic verifier, then out-of-band callback for blind classes, then sandboxed browser for XSS—reflects a deliberate cost/precision trade-off: the expensive browser oracle is invoked only on high-confidence XSS candidates (Section VI-A).

The adaptive controls form two feedback loops. The payload loop uses the feedback classifier and the strategy scores to steer generation toward categories and transformations that produce interesting responses, while the transport loop uses latency and close-code signals to back off when the server shows stress (Section VI-P). Because the evolver deduplicates by MD5 and caps population size, its memory footprint is bounded regardless of run length.

The attack framework's symmetry between HTTP and WebSocket—parallel replay, authorization-diff, and race services sharing identities and a variable-templating workflow engine—is what lets a single playbook chain an HTTP bootstrap into a WebSocket action (Section VI-K). The protocol subsystem closes the loop from observation to action: it converts raw frames into families, templates, and framework-specific attack templates, and recommends the playbooks that the workflow engine can then execute (Section VI-L).

On the evidence side, the combination of an Ed25519 signature over canonical JSON with per-list hash chains means

that both content tampering and reordering are detectable, and that verification is self-contained because the public key and chain roots travel inside the bundle; redaction runs before export so shared bundles do not leak secrets (Section VI-S). The plugin system's process isolation, size/timeout limits, and safe wrappers indicate that third-party plugin code is treated as untrusted (Section VI-U).

XI. Discussion

WSHawk's design suggests a view of realtime-application testing in which the connection, not the individual message, is the unit of work, and in which findings are only as useful as the evidence trail that accompanies them. The project-backed model—identities, traffic, findings, notes, and replay recipes living in one signed record—is the clearest expression of this view. The unification of WebSocket and HTTP workflows under a shared identity, variable, and evidence model is a practical response to modern applications that split state across a browser-authenticated HTTP surface and a realtime socket, and the protocol subsystem's framework-aware target packs reflect that realtime traffic is rarely "raw" but rather one of a handful of recognizable families.

XII. Limitations

Only limitations observable from the implementation are reported. (i) *No accuracy benchmark.* As noted, the repository validates behaviour and reproducibility but does not publish detection accuracy, false-positive rates, or comparative performance numbers. (ii) *Browser oracle dependency.* Execution confirmation and auth-flow recording require Playwright/Chromium; when Playwright is absent the verifier degrades to reporting unavailability, and the scanner falls back to heuristic-only XSS detection. (iii) *Simplified CVSS.* The scoring code implements a simplified v3.1 base-score computation with fixed vector components for some metrics, so scores are indicative rather than a full CVSS calculator. (iv) *OAST default.* In the reviewed scanner path the XXE phase targets a local OAST server; using the public collaborator for real blind detection requires enabling that provider, and the defensive module's DNS-callback check is an integration placeholder. (v) *Heuristic verifier bounds.* Non-browser classes rely on error/timing/reflection heuristics, which are inherently susceptible to false positives and negatives, mitigated but not eliminated by the feedback loop. (vi) *Heuristic binary parsing.* Binary format detection and protobuf/BSON parsing are heuristic and schema-less, so field extraction is best-effort. (vii) *Dependency discrepancy.* `numpy`/`scipy` appear in `requirements.txt` but were not used by the components reviewed. (viii) *Legacy surfaces.* Some CLI paths are explicitly compatibility wrappers over legacy code, so the CLI is not the most complete interface.

XIII. Security Considerations

WSHawk is an offensive tool; the repository's `SECURITY.md`, responsible-use notice, and license make authorized use a precondition. The implementation nonetheless protects its own trust boundaries: the desktop enables Electron renderer isolation and sandboxing and treats the bridge as loopback-only; the browser companion uses explicit, session-scoped pairing rather than unrestricted capture; secrets and signing keys are held in a platform-aware secret store; plugin code runs process-isolated with resource limits; exported bundles carry integrity and provenance metadata and are redacted before sharing; and the defensive-validation suite is explicitly framed for authorized blue-team use. A release security-check script (`scripts/release_security_checks.py`) is included for pre-publication review.

XIV. Future Work

The most significant gap is empirical. A controlled evaluation that measures detection accuracy and false-positive rates across a labelled realtime-application corpus, together with throughput measurements under the adaptive rate limiter, would substantiate the design claims made here; extending the existing validation labs into such a benchmark, and reporting per-class oracle precision (heuristic vs. browser-verified vs. OAST-confirmed), are natural next steps consistent with the current harness.

Beyond evaluation, several extensions follow directly from the architecture. First, the protocol subsystem's target-pack registry is open-ended, so *additional realtime protocols* (for example MQTT over WebSocket, STOMP, or WAMP) could be added as new adapters without changing the graph or workflow layers. Second, the context-aware generator currently learns fields and types heuristically; moving toward *protocol-aware grammars*, in the spirit of grammar-based fuzzing [19] and inferred protocol specifications [23], [24], would let the generator produce structurally valid messages for more complex schemas and would tighten the coupling between inference and payload synthesis. Third, the engine is single-node and `asyncio`-bound; a *distributed scanning* mode that partitions payload populations or target surfaces across workers, while preserving the token-bucket and circuit-breaker controls, would improve throughput against large targets. Fourth, the evidence layer already builds correlation chains and derives findings from replay events, so *richer evidence correlation*—linking HTTP flows, WebSocket frames, browser artifacts, and attack runs into end-to-end causal narratives, and scoring the confidence of a derived finding—is a natural deepening of the existing timeline and bundle model. These directions are extensions of subsystems that already exist in the source rather than speculative redesigns.

XV. Conclusion

This paper presented an implementation study of WSHawk, an open-source toolkit for stateful security assessment of WebSocket and realtime web applications. Grounded entirely in the repository source, the study described its phased stateful scanner and message/fingerprint intelligence; its two-layer adaptive payload subsystem (a strategy-scored mutation engine and a genetic Smart Payload Evolution pipeline); its

heuristic and Playwright-backed browser oracles and auth-flow recorder; its out-of-band testing and binary-message handling; its symmetric HTTP/WebSocket attack framework (replay, authorization diffing, race, subscription abuse) unified by a variable-templating workflow engine with six playbooks; its protocol-graph subsystem with six framework-aware target packs; its endpoint discovery, session-hijacking tester, and defensive-validation suite; its adaptive rate limiter, resilience layer, and issue-tracker integrations; its simplified CVSS scoring and multi-format reporting; and its project store with Ed25519-signed, hash-chained, redacted evidence bundles and a process-isolated plugin system. The toolkit's current validation is engineering-focused—a substantial test suite and three local validation labs—while a comprehensive empirical evaluation of detection performance remains future work.

References

[1] I. Fette and A. Melnikov, "The WebSocket Protocol," RFC 6455, IETF, 2011.

[2] Regaan, "WSHawk: Websocket security testing and web penetration testing toolkit (v4.0.1)," https://github.com/regaan/wshawk, 2026, aGPL-3.0. Accessed 2026.

[3] Python Software Foundation, "asyncio — Asynchronous I/O," https://docs.python.org/3/library/asyncio.html, 2024.

[4] Microsoft, "Playwright: Fast and Reliable End-to-End Testing for Modern Web Apps," https://playwright.dev/, 2024.

[5] C. Schneider, "Cross-Site WebSocket Hijacking (CSWSH)," https://christian-schneider.net/CrossSiteWebSocketHijacking.html, 2013.

[6] ProjectDiscovery, "Interactsh: An OOB Interaction Gathering Server and Client Library," https://github.com/projectdiscovery/interactsh, 2021.

[7] PortSwigger, "DOM Invader," https://portswigger.net/burp/documentation/desktop/tools/dom-invader, 2024.

[8] FIRST.org, "Common Vulnerability Scoring System v3.1: Specification Document," https://www.first.org/cvss/v3.1/specification-document, 2019.

[9] OASIS, "Static Analysis Results Interchange Format (SARIF) Version 2.1.0," https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html, 2020.

[10] S. Josefsson and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)," RFC 8032, IETF, 2017.

[11] OWASP Foundation, "OWASP Web Security Testing Guide," https://owasp.org/www-project-web-security-testing-guide/, 2020.

[12] ——, "OWASP Top 10," https://owasp.org/Top10/, 2021.

[13] ——, "OWASP Zed Attack Proxy (ZAP)," https://www.zaproxy.org/, 2024.

[14] PortSwigger, "Burp Suite," https://portswigger.net/burp, 2024.

[15] B. Damele and M. Stampar, "sqlmap: Automatic SQL Injection and Database Takeover Tool," https://sqlmap.org/, 2024.

[16] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 4, p. 2312–2331, oct 2019.

[17] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[18] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. Association for Computing Machinery, 2016, p. 1032–1043.

[19] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Association for Computing Machinery, 2017, p. 50–59.

[20] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "Snooze: toward a stateful network protocol fuzzer," in *Proceedings of the 9th international conference on Information security*, ser. ISC'06. Springer-Verlag, 2006, p. 343–358.

[21] H. Gascon, C. Wressnegger, F. Yamaguchi, K. Rieck, and L. Ortiz, "Pulsar: stateful black-box fuzzing of proprietary network protocols," in *Proceedings of the 10th International Conference on Security and Privacy in Communication Networks*, ser. SecureComm'14. Springer-Verlag, 2015, p. 330–347.

[22] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: a greybox fuzzer for network protocols," in *Proceedings of the 13th IEEE Conference on Software Testing, Validation and Verification*, ser. ICST '20. IEEE Press, 2020, p. 460–465.

[23] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: automatic protocol reverse engineering from network traces," in *Proceedings of the 16th USENIX Security Symposium*, ser. SS'07. USENIX Association, 2007, p. 14.

[24] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, ser. SP '09. IEEE Computer Society, 2009, p. 110–125.

Appendix A
Acronyms

TABLE I
Acronyms and abbreviations.

Acronym	Expansion
CORS	Cross-Origin Resource Sharing
CSWSH	Cross-Site WebSocket Hijacking
CSRF	Cross-Site Request Forgery
CVSS	Common Vulnerability Scoring System
DOM	Document Object Model
EMA	Exponential Moving Average
GA	Genetic Algorithm
HSTS	HTTP Strict Transport Security
JSON	JavaScript Object Notation
JWT	JSON Web Token
LFI	Local File Inclusion
MitM	Man-in-the-Middle
MV3	(Chrome) Manifest V3
OAST	Out-of-band Application Security Testing
RCE	Remote Code Execution
SaaS	Software as a Service
SARIF	Static Analysis Results Interchange Format
SPE	Smart Payload Evolution
SQLi	SQL Injection
SSO	Single Sign-On
SSRF	Server-Side Request Forgery
SSTI	Server-Side Template Injection
TLS	Transport Layer Security
WAF	Web Application Firewall
WS	WebSocket
WSS	WebSocket Secure
XSS	Cross-Site Scripting
XXE	XML External Entity

Appendix B
Notation

Appendix C
Data Tables

Appendix D
Plugin Interfaces

The plugin system (`plugin_system.py`) exposes three abstract extension points. Payload plugins return payloads for a vulnerability type; detector plugins return a $(\textit{is\_vulnerable}, \textit{confidence}, \textit{description})$ triple; protocol plugins asynchronously transform a message. The abbreviated interface is:

TABLE II
NOTATION USED FOR THE ADAPTIVE SUBSYSTEMS.

Symbol	Meaning
P	Population of candidate payloads in the evolver
$f(p)$	Fitness of payload p, $f(p) \in [0, 1]$
α	EMA learning rate (fitness 0.4; category 0.3)
N	Max population size (`population_size`, 100 in scanner)
r_x	Crossover rate (`crossover_rate`)
r_m	Mutation rate (`mutation_rate`)
$s(\sigma)$	Weighted score of mutation strategy σ
b	Adaptive rate multiplier (`backoff_multiplier`)
λ	Token refill rate (`tokens_per_second`)
B	Token-bucket capacity (`bucket_size`)
t_0	Baseline response time (learning phase)
t_r	Observed response time for a probe
ρ	Time ratio t_r/t_0 for delay detection
θ	Feedback delay threshold (`response_threshold`)

TABLE III
VULNERABILITY CLASSES PROBED BY THE WEBSOCKET SCANNER AND DETECTION TECHNIQUES, FROM `SCANNER_V2.PY`, `VULNERABILITY_VERIFIER.PY`, AND THE PAYLOAD FILES.

Class	Technique (as implemented)
SQL Injection	DB-specific error signatures; time-based (SLEEP/WAITFOR/pg_sleep); reflection = low confidence
XSS	Reflection/context heuristics (script/handler/attr), then sandboxed browser execution confirmation
Command Injection	Shell-error and `id`-output patterns; time-based sleep oracle
XXE	OAST callback detection (entity / parameter-entity / file variants)
SSRF	OAST callbacks and internal/metadata target payloads
NoSQL Injection	Operator-injection payloads (`$gt`,`$ne`,`$regex`,`$where`)
Path Traversal / LFI	File-content markers (`/etc/passwd`, `win.ini`, key headers)

```python
class PayloadPlugin(PluginBase):
    def get_payloads(self, vuln_type: str) -> List[
    str]: ...

class DetectorPlugin(PluginBase):
    def detect(self, response: str, payload: str,
               context: Dict = None) -> tuple[bool,
    str, str]: ...

class ProtocolPlugin(PluginBase):
    async def handle_message(self, message: str,
                             context: Dict = None)
    -> str: ...
```

Each plugin supplies `PluginMetadata` (name, version, description, author, requirements, minimum WSHawk version, checksum). The manager validates semantic-version formatting and major/minor compatibility, computes a SHA-256 file checksum, and invokes methods in a separate `plugin_runner` process bounded by a five-second timeout and message/file size limits, wrapping calls so a failing plugin returns a safe default.

TABLE IV
PAYLOAD MUTATION STRATEGIES IN `PAYLOAD_MUTATOR.PY` (`MUTATIONSTRATEGY`).

Strategy	Representative transformations
Encoding	URL, double-URL, HTML entity, unicode, UTF-7/16, Base64, overlong UTF-8
Case variation	upper, lower, alternating, randomized case
Comment injection	`/**/`, `--`, MySQL `/*!*/`, keyword splitting
Whitespace	tab, newline, vertical tab, form feed, encoded space
Concatenation	JS/SQL/Python string concatenation, `fromCharCode`
Bypass filter	null bytes, double encoding, unicode normalization
Tag breaking	split tags, encoded brackets, broken event handlers
Polyglot	multi-context XSS/SQL/command polyglots

TABLE V
ATTACK SERVICES IN `WSHAWK/ATTACKS/`.

Service	Module
WebSocket replay	`replay.py`
WebSocket AuthZ diff	`authz_diff.py`
WebSocket race	`race.py`
WebSocket subscription abuse	`subscription_abuse.py`
HTTP replay (+ templates)	`http_replay.py`
HTTP AuthZ diff	`http_authz_diff.py`
HTTP race	`http_race.py`
Cross-protocol workflows	`workflows.py`
Shared helpers	`common.py`, `http_common.py`

APPENDIX E
CONFIGURATION SURFACE

Configuration is YAML-based (`config.py`) with environment overrides and secret references. Documented search paths include `./wshawk.yaml`, `~/.wshawk/config.yaml`, and `/etc/wshawk/config.yaml`. Sections cover `scanner` (rate limit, timeout, payload count, SSL verification, feature flags), `ai` (optional payload assistance), `reporting` (output directory and formats), `integrations` (Jira, DefectDojo, webhooks), and `web` (host, port, auth, database). `scanner.verify_ssl` defaults to true. Relevant environment variables include `WSHAWK_BRIDGE_PORT`, `WSHAWK_WEB_PASSWORD`, and `WSHAWK_API_KEY`.

TABLE VI
BUILT-IN WORKFLOW PLAYBOOKS (ATTACKS/WORKFLOWS.PY, WORKFLOWEXECUTIONSERVICE.PLAYBOOKS).

Playbook ID	Purpose
login_bootstrap	Fetch login page, submit credentials, extract CSRF/session
csrf_session_capture	Fetch authed page, extract CSRF/session for reuse
http_replay	Replay an authed HTTP request with variables
ws_privilege_escalation	HTTP bootstrap then privileged WS replay
stale_token_reuse	Prime then race a stale replay window
tenant_hopping	HTTP + WS AuthZ diff across tenants

TABLE VII
REALTIME PROTOCOL TARGET PACKS (PROTOCOL/TARGET_PACKS.PY).

Pack ID	Protocol / signals
graphql_ws	GraphQL subprotocol; subscribe/next/complete; operations, variables
phoenix_channels	topic/event/payload/join_ref frames
actioncable	command/identifier frames with embedded identifier JSON
signalr	record-separator framing; type/target; invocation IDs
socket_io	Engine.IO prefixes; namespaces; event arrays
generic_realtime	structured-JSON fallback (action/type keys)
binary_realtime	consumes binary-analysis metadata (protobuf/msgpack/CBOR)

TABLE VIII
SESSION-SECURITY TESTS (SESSION_HIJACKING_TESTER.PY).

Test	Checks
Token reuse	Token accepted after session close
Subscription spoofing	Subscribe to privileged/other channels
Impersonation	Client-supplied user IDs honoured
Channel violation	Read other users' private channels
Session fixation	Attacker-chosen session ID accepted
Privilege escalation	Client-supplied role/permission fields honoured

TABLE IX
DEFENSIVE-VALIDATION TESTS (DEFENSIVE_VALIDATION.PY, WSS_SECURITY_VALIDATOR.PY).

Module	Validates
DNS exfiltration	Egress filtering via XXE/SSRF callbacks
Bot detection	Basic + evasion headless-browser detection
CSWSH	Origin enforcement (216+ origins), CSRF-token requirement
WSS/TLS	Deprecated versions, weak ciphers, cert, chain, renegotiation

TABLE X
CONSOLE ENTRY POINTS DECLARED IN PYPROJECT.TOML.

Command	Target / purpose
wshawk	wshawk.__main__:cli (compat → legacy core)
wshawk-interactive	wshawk.interactive:cli
wshawk-advanced	wshawk.advanced_cli:cli (compat)
wshawk-defensive	wshawk.defensive_cli:cli

TABLE XI
DAEMON ROUTE GROUPS (WSHAWK/DAEMON/).

Group	File / responsibility
Platform	platform_routes.py: projects, identities, replay, attacks, evidence, exports
Transport	transport_routes.py: extension ingestion, DOM tooling, interceptor
System	system_routes.py: status, config, pairing
Scan	scan_routes.py: WebSocket scan lifecycle
Web	web_routes.py: HTTP toolkit and reporting

TABLE XII
CONFIGURATION SECTIONS (CONFIG.PY, DEFAULT_CONFIG).

Section	Keys (selected)
scanner	rate_limit, timeout, learning_duration, max_payload_count, verify_ssl, features
ai	provider, model, base_url, api_key
reporting	output_dir, formats, auto_open
integrations	defectdojo, jira, webhook
web	host, port, debug, auth, database

TABLE XIII
FINDING INTEGRATIONS (WSHAWK/INTEGRATIONS/).

Integration	Behaviour
Jira	Severity→priority, min-severity filter, single + bulk create
DefectDojo	Product/engagement import of findings
Webhook	Slack / Discord / Teams / generic notifications

TABLE XIV
REPORT AND EVIDENCE OUTPUT FORMATS AND THEIR SOURCE MODULES.

Format	Producing module
JSON, CSV, SARIF 2.1.0	report_exporter.py
HTML (scanner)	enhanced_reporter.py
JSON, Markdown, HTML (bundle)	evidence/exporters.py
HTML (+documented PDF)	web_pentest/report_gen.py

TABLE XV
EVIDENCE-BUNDLE INTEGRITY METADATA (EVIDENCE/INTEGRITY.PY).

Field	Meaning
scheme	ed25519-sha256
key_id	Truncated SHA-256 of the public key
public_key	Base64 Ed25519 public key
content_sha256	Hash of canonical bundle JSON
signature	Ed25519 signature over canonical JSON
chain_roots	Hash-chain roots per ordered list
provenance	tool, version, operator, hostname, platform

TABLE XVI
BINARY FORMATS AUTO-DETECTED (BINARY_HANDLER.PY).

Format	Detection cue
Compressed	zlib/gzip magic bytes; recursive parse
Avro	`Obj\x01` magic
BSON	4-byte LE length + trailing null
Protobuf	wire-type/field-number heuristic
FlatBuffers	root-table offset range
MessagePack	fixmap/map prefix or library parse
CBOR	map prefix or library parse
Encrypted	Shannon entropy > 7.5 bits/byte

TABLE XVII
WEB PENETRATION-TESTING MODULES (WSHAWK/WEB_PENTEST/).

Module	Function
`crawler.py`	BFS crawler / endpoint discovery
`fuzzer.py`	HTTP parameter fuzzer
`dir_scanner.py`	Directory brute-forcing
`header_analyzer.py`	Security-header analysis
`tech_fingerprint.py`	Technology fingerprinting
`subdomain_finder.py`	Subdomain enumeration
`dns_lookup.py`	DNS / WHOIS lookup
`port_scanner.py`	Async TCP port scanner
`waf_detector.py`	WAF fingerprinting
`cors_tester.py`	CORS misconfiguration tests
`ssl_analyzer.py`	TLS/certificate analysis
`ssrf_prober.py`	SSRF probing
`redirect_scanner.py`	Open-redirect scanning
`proto_polluter.py`	Prototype-pollution testing
`csrf_forge.py`	CSRF PoC generation
`attack_chainer.py`	Multi-step attack chaining
`proxy_ca.py`	Root CA generation for interception
`sensitive_finder.py`	Secret/sensitive-data detection
`vuln_scanner.py`	Multi-phase orchestrator
`report_gen.py`	Report generation
`http_proxy.py`, `platform.py`	Proxy shim / platform glue

TABLE XVIII
SELECTED DESKTOP FEATURES (DESKTOP/ AND THE DESKTOP GUIDE).

Feature	Notes
WebSocket interceptor	Frame-by-frame forward/drop/edit
Payload blaster	High-volume WS fuzzing, SPE + DOM-verify toggle
Endpoint map	WS endpoint discovery
Auth builder	Multi-step auth with variable extraction
Evidence vault	Findings, recipes, timeline review
Project files	`.wshawk` save/reopen
Modes	Standard / Advanced / Web Pentest
Builds	Linux, Windows, macOS (electron-builder)