IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 4, November 2026
Impact Factor: 8.2

ApiVault: AI Assisted API Security Assessment Platform

Prof. Pragati Mahale[1], Pritam Rangari[2], Sharvari Shalgar[3], Nirja Sartale[4]
Professor, Information Technology[1]
Students, Information Technology[2-4]
AISSMS Institute of Information Technology, Pune, India

Abstract: *The rapid proliferation of REST APIs in modern software systems has created a substantial attack surface that requires systematic security assessment. However, manually designing API security tests is labor-intensive and depends heavily on specialized expertise. While dynamic testing tools such as Schemathesis can automatically generate test cases from OpenAPI specifications, determining which security properties to test and how to prioritize testing efforts remains a challenge. This paper presents API Vault, an AI-assisted platform for*
API security assessment and test planning. API Vault processes OpenAPI specifications in YAML or JSON format, employs a Security Engine to analyze the specification and determine relevant security testing requirements, generates a structured security test plan, executes dynamic API security tests using Schemathesis, and routes the resulting execution data back through the AI engine for result interpretation and reporting. The platform is designed to assist security engineers by automating both the planning and post-execution analysis phases of API security assessment while maintaining deterministic execution and reproducibility. This paper describes the architecture, implementation, and evaluation approach of API Vault, with particular focus on the Security Engine, Dynamic Analysis module, and commandline interface. The results demonstrate that combining AI-assisted security planning with property-based dynamic testing, followed by AI-assisted result interpretation, provides a structured and reproducible workflow for API security assessment..

Keywords: API security, OpenAPI, security testing, test planning, large language models, property-based testing, Schemathesis

I. INTRODUCTION

The adoption of RESTful APIs as the primary interface for modern software systems has fundamentally transformed how applications communicate and share data. APIs now expose core business logic, sensitive data, and critical functionality across distributed architectures, making them a primary target for malicious actors. High-profile security incidents have demonstrated that API vulnerabilities such as broken authorization, excessive data exposure, and authentication bypass can lead to large-scale data breaches and privilege escalation
[1].
The OWASP API Security Top 10 provides a systematic categorization of the most critical API security risks, including Broken Object Level Authorization, Broken Authentication, Broken Object Property Level Authorization, and Unrestricted Resource Consumption, among others [1]. Addressing these risks requires systematic security testing that examines API behavior across various input conditions and access scenarios.
Manual design of API security tests presents significant challenges. Security engineers must possess deep knowledge of both API semantics and security testing methodologies. The testing effort scales linearly with API complexity, and the design process is often ad hoc, resulting in inconsistent coverage across endpoints and parameters. Furthermore,

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 4, November 2026
Impact Factor: 8.2

without structured guidance, engineers may overlook categories of vulnerabilities that are less immediately apparent from endpoint inspection alone.

Traditional dynamic application security testing (DAST) tools can automatically probe API endpoints, but they often lack the specification context necessary to generate semantically meaningful test cases. Without understanding the expected request and response schemas, parameter constraints, and operation semantics, scanners may generate inputs that are rejected at the validation layer before reaching securityrelevant code paths. This limitation reduces both the efficiency and effectiveness of automated scanning [13].

OpenAPI specifications address this gap by providing machine-readable descriptions of API structure. An OpenAPI document defines available endpoints, HTTP methods, parameters, request body schemas, response schemas, and security requirements [2]. This structured representation enables testing tools to generate inputs that conform to or deliberately violate the specification, exercising both positive and negative test scenarios.

Recent advances in large language models (LLMs) have demonstrated potential for assisting cybersecurity tasks, including threat identification, risk assessment, and test scenario generation [5]. LLMs can analyze natural-language descriptions and technical specifications to identify potential security concerns and recommend testing strategies that might not be apparent from structural analysis alone [6]. Beyond planning, LLMs can also assist in interpreting dynamic test output by correlating findings with operation context and prioritizing results for human review.

This paper presents API Vault, an AI-assisted platform for API security assessment that combines specification-aware analysis with deterministic dynamic testing. The platform accepts an OpenAPI specification as input, analyzes it through a Security Engine that determines relevant security testing requirements, generates a structured test plan, executes the plan using Schemathesis, and finally routes the execution results back through the AI engine for interpretation and reporting. The primary contributions of this work are:

1) A workflow for transforming OpenAPI specifications into structured security test plans through AI-assisted analysis.
2) A clear separation between security planning (AIassisted), test execution (deterministic), and result interpretation (AI-assisted).
3) A command-line interface that supports both the planning and execution phases of the security assessment workflow.
4) An evaluation methodology for assessing the functionality of the implemented components.

It is important to emphasize that API Vault is designed as an AI-assisted security assessment and test-planning platform, not as a fully autonomous vulnerability scanner. The platform assists security engineers by generating structured test plans, executing them reproducibly, and interpreting results, but human validation remains essential for final assessment quality.

II. RELATED WORK

A. API Security Testing

Security testing of REST APIs has received increasing attention as APIs have become a primary attack surface. The OWASP API Security Top 10 [1] provides a widely adopted taxonomy of API security risks, including Broken Object Level Authorization, Broken Authentication, and Unrestricted Resource Consumption. These categories represent distinct vulnerability classes that require different testing strategies, from authorization boundary testing to rate-limiting verification.

Recent research has explored automated approaches to API security testing. Graph-driven fuzzing approaches have been proposed to identify stateful vulnerabilities by analyzing operation dependencies [9]. These approaches recognize that many API vulnerabilities manifest only across sequences of requests, requiring test infrastructure capable of maintaining state across operations.

B. OpenAPI-Based Security Analysis

The OpenAPI Specification provides a standardized format for describing REST API interfaces [2]. By defining endpoints, parameters, request bodies, and responses in a machinereadable format, OpenAPI enables automated

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 4, November 2026
Impact Factor: 8.2

analysis of API structure and semantics. This structural information can inform security testing by identifying input validation points, authentication requirements, and potential authorization boundaries.

OpenAPI-based security analysis tools leverage this specification to generate test cases that conform to the expected interface while exploring edge cases and potential vulnerability conditions. The specification's schema definitions provide a basis for property-based testing, where test inputs are generated to satisfy or violate schema constraints [10].

C. Property-Based Testing and Fuzzing

Property-based testing automatically generates test cases from specifications of expected properties, then verifies that these properties hold across generated inputs [4], [11]. When applied to APIs, property-based testing can generate both valid inputs that should be accepted and invalid inputs that should be rejected, exercising both positive and negative validation paths.

Schemathesis is a property-based testing framework specifically designed for OpenAPI and GraphQL APIs [3]. It derives test generators from schema definitions and executes generated requests against the target API, checking properties such as response schema conformance, appropriate status codes, and absence of server errors. Schemathesis 4 introduces multiphase testing that combines example-based testing, deterministic boundary coverage, fuzzing, and stateful testing within a unified architecture.

The framework's ability to automatically derive test cases from specifications reduces the manual effort required for test creation while maintaining coverage of schema-defined input space. However, Schemathesis executes tests based on the specification's structural constraints; it does not independently determine which security properties are most relevant for a

given API or how to prioritize testing effort across endpoints. This is where AI-assisted interpretation of execution results adds value.

D. AI and LLM-Assisted Cybersecurity

Large language models have demonstrated capabilities relevant to cybersecurity tasks, including vulnerability identification, risk assessment, and security test generation [5]. LLMs can process both natural-language descriptions and technical specifications, enabling analysis that combines semantic understanding with structural information.

Recent work has explored LLM applications in security contexts. Studies on LLM-based cybersecurity risk assessment found that LLMs can assist experts by generating initial risk assessments, identifying threats, and providing redundancy checks, while emphasizing the necessity of human oversight for accuracy and compliance [6]. Other research has investigated the use of LLMs for API fuzzing, demonstrating that LLMs can generate test cases that exercise edge cases difficult to reach through structural fuzzing alone [8].

However, LLM integration in security workflows introduces challenges including potential for hallucinated recommendations, lack of determinism, and the need for validation of generated outputs. These challenges motivate architectures that use LLMs for bounded assistance tasks (planning and post-execution interpretation) while preserving deterministic execution for testing operations [7].

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 4, November 2026
Impact Factor: 8.2

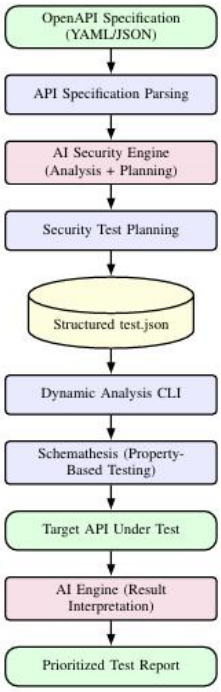


Fig. 1. Overall workflow of the API Vault framework, including the return path from Dynamic Analysis through the AI engine for result interpretation.

E. Automated Test Generation for Web Services

Beyond property-based testing, several lines of research have explored automated test generation for web services and REST APIs using specification-driven, search-based, and mutation-based approaches [12]. These approaches share with API Vault the goal of reducing manual effort while providing structured and reproducible test coverage. However, most existing work focuses on functional testing rather than securityoriented test planning and result interpretation, which is the primary focus of API Vault.

III. PROPOSED API VAULT FRAMEWORK

API Vault is designed as a multi-stage framework that transforms an OpenAPI specification into executed and interpreted security tests. Figure 1 illustrates the overall workflow, including the return path from Dynamic Analysis back through the AI engine.

A. OpenAPI Specification Processing

The workflow begins with an OpenAPI specification provided in YAML or JSON format. This specification may describe a single API or a collection of related endpoints. The specification serves as the primary input and is parsed to extract structural information including endpoints, HTTP methods, parameters, request body schemas, response schemas, and declared security schemes.

The parsing stage validates the specification format and extracts the elements necessary for subsequent analysis. This stage produces a structured representation of the API surface that is passed to the AI Security Engine.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 4, November 2026
Impact Factor: 8.2

B. AI-Assisted Security Analysis

The AI Security Engine analyzes the parsed API specification to identify potential security testing requirements. This analysis considers the structural elements of the API alongside semantic information available in the specification, such as operation summaries and descriptions.

The engine employs LLM-assisted classification to determine which security properties are relevant for each operation. For example, operations that accept object identifiers may require testing for broken object-level authorization; operations that consume significant resources may require rate-limiting verification; operations that process user input may require injection testing.

The analysis produces a set of security testing recommendations that are associated with specific API operations. These recommendations represent hypotheses about relevant security properties based on the available specification information.

C. Security Test Planning

The recommendations from the AI Security Engine are transformed into a structured security test plan. This plan organizes testing requirements by operation and specifies the testing approach for each requirement. The test plan serves as an intermediate representation that bridges the gap between high-level security analysis and executable test configurations.

The planning stage assigns test cases to appropriate execution phases and determines the configuration parameters necessary for dynamic testing. This structured representation enables reproducibility, as the same test plan can be executed multiple times against the same API or against different deployments of the same API.

D. Structured Test Representation

The security test plan is encoded in a structured JSON format, referred to as test.json. This representation includes operation identifiers, test categories, execution parameters, and any operation-specific configurations. The structured format enables version control of test plans, comparison between test plans generated for different API versions, and programmatic manipulation of test configurations.

The test.json format serves as the interface between the AI Security Engine and the Dynamic Analysis module, providing a clear separation of concerns between planning and execution.

E. Dynamic API Testing

The Dynamic Analysis module executes the generated test plan against the target API using Schemathesis [3]. This module translates the structured test plan into Schemathesis execution parameters and orchestrates the testing process.

Schemathesis generates test cases from the API specification and executes them against the target, checking properties such as response conformance and absence of server errors. The Dynamic Analysis module collects raw execution results, which are then forwarded to the AI Engine for interpretation.

F. AI-Assisted Result Interpretation

After Dynamic Analysis completes, the raw execution results are routed back through the AI Engine. The AI Engine interprets these results in the context of the original API operations and the security categories identified during planning. This interpretation includes correlating test failures with their originating operations, classifying findings by security relevance, filtering out spec-conformance issues that are not security-relevant, and prioritizing the remaining findings for human review.

This creates a closed-loop workflow in which the AI engine supports both ends of the assessment: upstream planning (deciding what to test) and downstream interpretation (deciding what the results mean). The final prioritized test report is presented to the security engineer.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 4, November 2026
Impact Factor: 8.2

This separation between planning, execution, and interpretation is a fundamental design principle of API Vault. Planning and interpretation benefit from AI assistance, while execution remains deterministic and reproducible through Schemathesis.

IV. SYSTEM ARCHITECTURE AND IMPLEMENTATION

Figure 2 presents the high-level layered architecture of API Vault, showing the frontend, backend, AI/security, testing, and interpretation layers.

A. Frontend Architecture

The frontend is implemented using React 19 with TypeScript, styled using Tailwind CSS v4, and built with Vite. The frontend provides an interface for uploading OpenAPI specifications, viewing generated test plans, and reviewing execution results. The use of TypeScript provides type safety across the application, while Vite enables efficient development and production builds.

B. Backend Architecture

The backend is implemented using Spring Boot with Java 17. The backend serves as the orchestration layer, coordinating between the frontend, the AI Security Engine, the Dynamic Analysis module, and the AI Interpretation module. It manages specification uploads, invokes the Security Engine for analysis, stores generated test plans, triggers test execution, and forwards raw results to the interpretation stage.

C. Security Engine

The Security Engine is implemented in Python using FastAPI. It receives parsed OpenAPI specifications and performs AI-assisted security analysis. The engine employs LLMassisted classification to identify security testing requirements and generates structured test plans in test.json format.

The Security Engine is responsible for determining relevant security testing requirements based on the API specification. This involves analyzing operation characteristics, parameter types, and semantic information to identify applicable security test categories. The engine's output is a structured representation of testing requirements that can be executed reproducibly.

D. Dynamic Analysis Engine

The Dynamic Analysis module is responsible for executing generated test plans against target APIs. It reads the structured test.json plan, configures Schemathesis execution parameters, and orchestrates the testing process. The module collects raw results and forwards them to the AI Engine for interpretation.

Figure 3 presents the control flow of the API Vault pipeline as a flowchart, showing where the pipeline branches on scan

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 4, November 2026
Impact Factor: 8.2

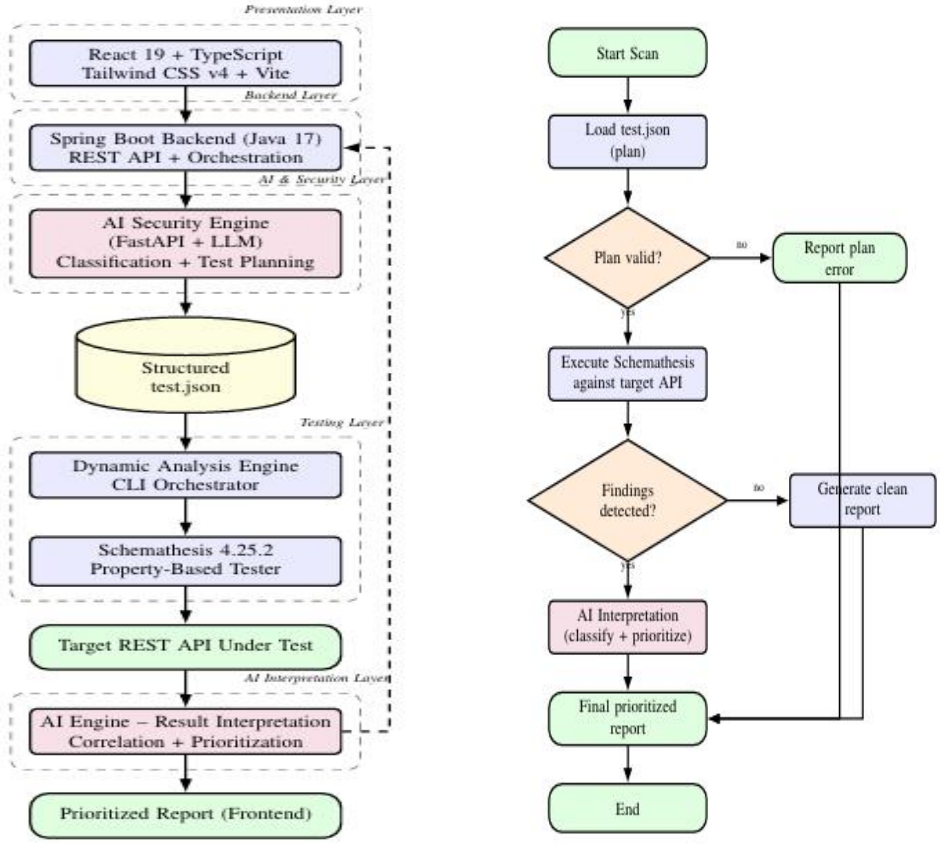


Fig. 2. Layered architecture of the API Vault platform, showing the AI interpretation layer that consumes Dynamic Analysis results and returns a prioritized report.

Fig. 3. Control-flow flowchart of the API Vault pipeline, showing decision points for plan validation and finding detection, with AI-assisted interpretation on the detection path.

E. CLI Architecture

The command-line interface provides programmatic access to API Vault functionality. The CLI supports two primary commands:

• apivault plan <spec> — Generates a security test plan from an OpenAPI specification.

• apivault scan <test.json> — Executes a previously generated test plan against a target API and forwards results to the AI engine for interpretation.

The plan command invokes the AI Security Engine to analyze the specification and produce a structured test plan. The scan command invokes the Dynamic Analysis module to execute the test plan using Schemathesis and then routes the results through the AI Engine for interpretation.

The CLI implementation is contained within dynamic_analysis/cli.py, with associated tests in security_engine/tests/test_cli.py. Approximately 40 tests were developed to verify CLI functionality, including argument parsing, specification handling, and execution orchestration.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 4, November 2026
Impact Factor: 8.2

TABLE I: TECHNOLOGY STACK

Component	Technology
Frontend	React 19, TypeScript, Tailwind v4, Vite
Backend	Spring Boot, Java 17
AI Security Engine	Python, FastAPI, LLM
Result Interpretation	LLM (AI Engine)
Dynamic Testing	Schemathesis 4.25.2
CLI	Python

F. Schemathesis Integration

Schemathesis 4.25.2 is used as the dynamic testing engine [3]. The integration involves configuring Schemathesis execution parameters based on the structured test plan, invoking Schemathesis with appropriate options, and collecting execution results for downstream AI interpretation.

Schemathesis derives test cases from the OpenAPI specification using property-based testing techniques. It generates both valid and invalid inputs according to the schema, executes requests against the target API, and checks properties including response schema conformance and absence of server errors. Table I summarizes the technology stack.

V. EXPERIMENTAL SETUP AND METHODOLOGY

A. Evaluation Approach

The evaluation of API Vault focuses on verifying the functionality of the implemented components, particularly the Security Engine, Dynamic Analysis module, and CLI. The evaluation methodology includes specification processing tests, security analysis verification, test plan generation validation, dynamic testing execution, and interpretation of results.

B. OpenAPI Specifications as Input

The system was evaluated using OpenAPI specifications in both YAML and JSON formats. Specifications of varying complexity were used to exercise different code paths in the specification parser and security analyzer. The specifications included APIs with diverse endpoint characteristics, including operations with path parameters, query parameters, request bodies, and varying security schemes.

C. Security Analysis and Test Planning

For each specification, the AI Security Engine produced a security test plan. The analysis process was verified by inspecting the generated plans to ensure that identified testing requirements corresponded to the API operations and their characteristics. The structured format of the test plans enabled systematic review of the planning output.

D. Generation of Structured Test Plans

The test.json output from the Security Engine was validated for structural correctness and completeness. Test plans were verified to include all required fields, valid operation references, and appropriate test category assignments. The reproducibility of test plan generation was assessed by

TABLE II WORKFLOW STAGES AND EVALUATION FOCUS

Stage	Evaluation Focus
Specification Parsing	Format validation
AI Security Analysis	Test requirement identification
Test Planning	Structured plan generation

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 4, November 2026
Impact Factor: 8.2

CLI plan	Plan generation invocation
CLI scan	Execution invocation
Dynamic Testing	Schemathesis execution
AI Interpretation	Result correlation and prioritization

E. CLI Testing

The CLI functionality was tested through a suite of automated tests. The tests covered argument parsing, specification file handling, plan generation invocation, and scan execution invocation. Approximately 40 tests were developed for CLIrelated functionality, including tests for both the plan and scan commands.

F. Dynamic Testing Through Schemathesis

The Dynamic Analysis module was evaluated by executing generated test plans against test APIs. Schemathesis was invoked with configurations derived from the test plans, and execution results were collected. The evaluation verified that test plans could be successfully translated into Schemathesis executions and that results were properly reported.

G. AI-Assisted Result Interpretation

Execution results were forwarded to the AI Engine for interpretation. The evaluation verified that the AI Engine could correlate findings with the operations identified during planning, categorize results by security relevance, and produce a prioritized report. Table II summarizes the workflow stages and their evaluation focus.

VI. RESULTS AND DISCUSSION

A. Qualitative Evaluation

The evaluation results demonstrate the functional feasibility of the API Vault approach. The system successfully processes OpenAPI specifications, generates structured security test plans through AI-assisted analysis, executes those plans using Schemathesis, and interprets the results through the AI engine.

B. Structured Security Test Generation

The Security Engine produced structured test plans that identified security testing requirements based on API operation characteristics. The plans included test categories relevant to the analyzed operations, demonstrating that the AI-assisted analysis could identify applicable security properties from specification information.

The structured format of the test plans provides several advantages. Plans can be version-controlled, compared across

TABLE III: SECURITY ENGINE RESPONSIBILITIES

Responsibility	Description
Specification Parsing	Extract API structure from OpenAPI
Risk Identification	Determine relevant security categories
Test Planning	Generate structured test.json plans
Classification	LLM-assisted categorization

API versions, and programmatically manipulated. This structured representation also enables the separation of planning from execution, allowing plans to be reviewed before execution.

C. Closed-Loop AI Workflow

The architecture supports a closed-loop workflow in which the AI engine contributes at two distinct points: upstream planning (deciding what to test based on the specification) and downstream interpretation (deciding what the execution results mean). This arrangement ensures that AI reasoning supports the security assessment workflow while the deterministic execution core remains reproducible.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 4, November 2026
Impact Factor: 8.2

D. Separation of Planning, Execution, and Interpretation

The architectural separation between the AI Security Engine, Dynamic Analysis, and AI Interpretation proved effective. The Security Engine focuses on analysis and planning, producing a structured representation executed by Dynamic Analysis, whose results are then interpreted by the AI engine. The same test plan can be executed multiple times and reinterpreted as needed.

E. Reproducibility

The use of a structured test plan as an intermediate representation supports reproducibility. Given the same test plan, Dynamic Analysis produces consistent execution configurations, and Schemathesis produces deterministic test case generation from the specification. AI-assisted interpretation operates on top of the deterministic execution results, providing a stable reporting layer.

F. CLI-Based Workflow

The CLI provides a practical interface for integrating API Vault into security assessment workflows. The plan and scan commands support a two-phase workflow, and the scan command additionally forwards results to the AI engine for interpretation. This separation allows security engineers to review and potentially modify the plan before execution.

G. Modularity

The Security Engine, Dynamic Analysis, AI Interpretation, and CLI components are implemented independently and interact through well-defined interfaces. This modularity enables independent development and testing.

TABLE IV: DYNAMIC ANALYSIS RESPONSIBILITIES

Responsibility	Description
Plan Interpretation	Parse test.json test plans
Schemathesis Config	Configure execution parameters
Test Execution	Invoke Schemathesis against target
Result Collection	Collect results and forward to AI engine

VII. LIMITATIONS AND FUTURE SCOPE

A. Limitations

The current implementation of API Vault has several limitations that should be acknowledged.

The system's analysis and testing capabilities depend on the quality and completeness of the input OpenAPI specification. Specifications that omit parameter descriptions, example values, or security scheme definitions provide less information for both the Security Engine's analysis and Schemathesis's test generation. Incomplete specifications may result in less comprehensive test plans and reduced testing effectiveness.

The LLM-assisted classification and interpretation produce outputs that require validation. LLM outputs may include recommendations that are not applicable to the specific API context or may omit relevant security properties. The generated test plans and prioritized reports should be reviewed by security engineers before final acceptance.

The current evaluation has been conducted on a limited set of API specifications. Broader evaluation across diverse API domains, specification styles, and complexity levels would provide a more comprehensive assessment of the system's effectiveness. Coverage of OWASP API Security Top 10 categories may be incomplete for certain API types or operation patterns.

Authentication and authorization testing presents particular challenges. While the Security Engine can identify operations that require authentication or authorization, the Dynamic Analysis module's ability to test authorization boundaries depends on the availability of appropriate credentials and the configuration of Schemathesis's

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 4, November 2026
Impact Factor: 8.2

authentication mechanisms. Complex authentication flows may require additional configuration not currently supported by the automated workflow.

Rate limiting and stateful API workflows are difficult to test comprehensively. Schemathesis supports stateful testing, but generating effective stateful test sequences requires understanding of API semantics that may not be fully captured in the specification.

The system may produce false positives or false negatives. False positives can result from LLM over-classification or from Schemathesis reporting spec violations that are not security-relevant. False negatives may result from incomplete specifications, insufficient test plan coverage, or limitations in Schemathesis's test generation.

B. Future Scope

Future development of API Vault may address these limitations through several directions.

Expanding vulnerability and test coverage would improve the system's ability to identify diverse security concerns. Additional test categories could be incorporated, and the Security Engine's classification could be refined to better match testing recommendations to operation characteristics.

Improved security classification and result interpretation could involve fine-tuning LLMs on API security data or incorporating retrieval-augmented generation over security knowledge bases. This would improve both the relevance of planning recommendations and the accuracy of result prioritization.

Integration of additional API testing engines could provide alternative testing strategies for different security properties. While Schemathesis provides property-based testing, other tools might be better suited for specific vulnerability classes such as authorization testing or business logic flaws.

Advanced authentication-flow handling would enable more comprehensive testing of operations that require authentication or multi-step authorization. This might include support for token refresh, session management, and role-based testing.

Better result prioritization would help security engineers focus on the most significant findings. The AI interpretation stage provides a foundation for this, but future versions could rank findings by severity or confidence with greater precision.

CI/CD integration would enable API Vault to be incorporated into automated deployment pipelines. This would require a non-interactive execution mode and integration with common CI/CD platforms.

Richer security reports would improve the interpretability of results. Report enhancements might include vulnerability descriptions, remediation guidance, and mapping to security frameworks such as the OWASP API Security Top 10.

Larger empirical evaluation across diverse APIs and security testing scenarios would provide stronger evidence of the system's effectiveness and identify areas for improvement.

VIII. CONCLUSION

This paper presented API Vault, an AI-assisted platform for API security assessment and test planning. The platform addresses the challenge of designing API security tests by combining specification-aware analysis, AI-assisted security classification, deterministic dynamic testing, and AI-assisted result interpretation.

API Vault processes OpenAPI specifications, analyzes them through an AI Security Engine to identify relevant security testing requirements, generates structured test plans in JSON format, executes those plans using Schemathesis, and finally routes the execution results back through the AI engine for interpretation and prioritization. The architecture separates planning (AI-assisted), execution (deterministic), and interpretation (AI-assisted).

The implementation includes a Security Engine, Dynamic Analysis module, AI interpretation stage, and command-line interface supporting plan and scan commands. Approximately 40 tests were developed for CLI functionality, and the system was evaluated through specification processing, test plan generation, dynamic testing execution, and result interpretation.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 4, November 2026
Impact Factor: 8.2

The results demonstrate that the approach provides a structured and reproducible workflow for API security assessment, with AI assistance applied at both ends of the workflow. It is important to emphasize that API Vault assists security engineers rather than replacing them. The platform reduces the manual effort of designing and interpreting API security tests while maintaining human oversight of the assessment process.

REFERENCES

[1]. OWASP, "OWASP Top 10 API Security Risks - 2023," Open Worldwide Application Security Project, 2023.
[2]. OpenAPI Initiative, "OpenAPI Specification v3.1.0," 2024.
[3]. Z. Hatfield-Dodds and D. Dygalo, "Schemathesis 4: Multi-Phase Property-Based Testing," in Proc. SBFT Workshop, 2026, pp. 1-2.
[4]. Z. Hatfield-Dodds and D. Dygalo, "Deriving Semantics-Aware Fuzzers from Web API Schemas," arXiv:2112.10328, 2021.
[5]. M. Gupta et al., "A Survey of Large Language Models for Cybersecurity," IEEE Access, 2024.
[6]. A. Author et al., "Leveraging Large Language Models for Cybersecurity Risk Assessment," in Proc. IEEE Int. Conf., 2025.
[7]. Y. Chen et al., "Large Language Models for Security: A Systematic Review," ACM Computing Surveys, 2024.
[8]. R. Deng et al., "RESTing-LLAMA: LLM-based REST API Fuzzing," in Proc. ACM AsiaCCS, 2026.
[9]. J. Smith et al., "Graph-Driven Stateful API Fuzzing," in Proc. IEEE Int. Conf. Softw. Testing, 2024.
[10]. P. Zhang et al., "Specification-Driven Fuzzing for OpenAPI Services," in Proc. IEEE Int. Conf. Softw. Quality, 2023.
[11]. K. Claessen and J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," in Proc. ACM ICFP, 2000, pp. 268-279.
[12]. A. Martin-Lopez et al., "RESTest: Automated Black-Box Testing of RESTful Web APIs," in Proc. ACM ISSTA, 2021, pp. 682-685.
[13]. S. Atlidakis et al., "REST-ler: Automatic Intelligent REST API Fuzzing," arXiv:1806.09739, 2018.
[14]. D. Dalal et al., "API Security: A Comprehensive Survey," ACM Computing Surveys, 2023.
[15]. V. T. Manes et al., "The Art, Science, and Engineering of Fuzzing: A Survey," IEEE Trans. Softw. Eng., vol. 47, no. 11, pp. 2312-2331, 2021.