International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 04 | April - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

SQLiShield:

A Multi-Layer SQL Injection Detection and Prevention Framework

Combining Deterministic Regex Matching with Adaptive Machine Learning for Real-Time Web Application Protection

SQLiShield Framework — Student Authors

Patel Vishwa Mehulkumar | **Chetan Ravindra Vashiste** | **Heet Patel**

B.Tech Computer Science and Engineering

Parul Institute of Technology, Parul University, Gujarat, India

SQLiShield Framework — Project Guides

Dr. Vivek Tiwari | **Mr. Dinesh Kumar Cholkar** | **Ms. Riddhi Atulkumar Mehta**

Department of Computer Science and Engineering, Parul Institute of Technology, Parul University, Gujarat, India

Abstract

SQL Injection (SQLi) continues to rank among the most exploited vulnerabilities in web applications globally, appearing in the OWASP Top 10 for over two decades. This paper presents **SQLiShield**, a novel multi-layer detection and prevention middleware engineered for Python/Flask web applications. SQLiShield fuses three hierarchical protection layers: a deterministic engine of 53 pre-compiled regular expression patterns (Layer 1), an adaptive Random Forest ML classifier trained on 6,941 labeled payload samples (Layer 2), and a structural input sanitization fallback (Layer 3). The framework is deployed as a Flask before_request hook, transparently intercepting all GET parameters, POST form fields, JSON bodies, and HTTP cookies before they reach application logic.

Empirical evaluation using the Kaggle SQL Injection Dataset demonstrates that the combined multi-layer architecture achieves **99.2% detection accuracy**, **99.5% precision**, **98.8% recall**, and a **0.5% false positive rate** — substantially outperforming standalone regex (87.0%, 18% FPR) and standalone ML (97.3%, 1.8% FPR). End-to-end detection latency averages **3.7 ms** per request. Automated penetration testing with SQLMap across 100 obfuscated payloads yielded **zero successful bypasses**. A companion Flask demonstration application exposes four intentionally vulnerable endpoints — authentication bypass, UNION-based extraction, secondorder injection, and a real-time admin dashboard — enabling controlled side-by-side comparison of protected and unprotected execution paths.

Keywords: *SQL Injection, SQLiShield, Random Forest, Web Application Firewall, Flask Middleware, OWASP, Intrusion Detection, Regex Pattern Matching, Parameterized Queries, Defense-in-Depth, Parul University*

1. Introduction

SQL Injection is a code injection technique in which a malicious actor embeds crafted SQL syntax within user-supplied input, causing the backend database engine to interpret attacker-controlled content as executable code. The OWASP Foundation has classified injection flaws as a critical web application risk continuously since 2003. The 2023 Verizon Data Breach Investigations Report identifies injection-

based techniques as contributors to hundreds of breach incidents annually, collectively resulting in billions of dollars of financial damage.

Despite this enduring threat, a significant fraction of web applications remain vulnerable. A 2023 Immuniweb scan of Fortune 500 digital assets found 18% susceptible to some form of injection attack. The persistence of this vulnerability is attributable to several compounding factors: developer unawareness of secure query construction, sprawling legacy codebases written before modern security best practices existed, and increasingly sophisticated attack obfuscation that defeats traditional signature-based Web Application Firewalls (WAFs).

This paper introduces **SQLiShield**, an open-source middleware framework developed during a cybersecurity internship at Parul University, Gujarat. SQLiShield addresses the core weaknesses of existing single-paradigm defenses by combining the sub-millisecond speed of regex pattern matching with the contextual adaptability of machine learning, wrapped in a practical Flask integration layer requiring no changes to application database queries.

1.1 Motivation

Parameterized queries remain the theoretically ideal defense against SQL injection. However, empirical evidence indicates that near 60% of enterprise legacy systems cannot fully migrate to parameterized patterns across their entire SQL surface area due to complexity, cost, and architectural constraints. Furthermore, parameterized queries are passive — they silently neutralize attacks without logging, alerting, or classifying the event. Organizations deploying only parameterized queries are effectively operating blind to active attack campaigns. SQLiShield was designed to fill this gap: active detection, logging, and blocking with near-zero deployment friction.

1.2 Research Objectives

1. Design a multi-layer SQLi detection architecture combining deterministic and adaptive detection strategies.
2. Implement a 53-pattern optimized regex engine for rapid elimination of known attack signatures.
3. Train and integrate a Random Forest classifier on 6,941 labeled payload samples to detect obfuscated attacks.
4. Build a Flask middleware hook that inspects all request surfaces (GET, POST, JSON, cookies) transparently.
5. Validate the framework's effectiveness using automated penetration testing (SQLMap) and standard ML metrics.
6. Demonstrate the framework against four distinct SQLi attack classes in a controlled demo application.

1.3 Scope and Limitations

SQLiShield is designed for Python/Flask applications using MySQL and SQLite backends. While the framework's architecture is generalizable, the regex and ML feature set were optimized and validated against MySQL, SQLite, and MSSQL dialects. PostgreSQL and Oracle-specific vendor syntax is partially covered. The framework does not address XSS, command injection, CSRF, or other non-SQL injection attack classes.

2. Background and Related Work

2.1 SQL Injection: Mechanism and Taxonomy

SQL Injection exploits the failure to separate executable SQL code from user-supplied data. When an application interpolates raw input directly into a query string, an attacker can alter the query's logical structure. Three primary anomaly categories exist in this context:

- **Point Anomalies:** A single malicious input causes immediate unauthorized query execution (e.g., authentication bypass via ' OR '1'='1).
- **Contextual Anomalies:** Input that is structurally normal globally but semantically anomalous for a given user or endpoint (e.g., an unusually structured search term for a known benign user).
- **Collective Anomalies:** A series of individually innocuous inputs that constitute a coordinated extraction campaign (e.g., sequential boolean-blind inference probes).

2.2 Limitations of Existing Defenses

Regex-only WAFs (e.g., ModSecurity CRS): Static pattern libraries achieve approximately 8587% detection accuracy but suffer from 10-18% false positive rates requiring intensive manual tuning. Critically, regex cannot model query semantics — attackers routinely bypass rules through URL encoding (%27 for apostrophe), hex encoding (0x61646d696e for 'admin'), SQL comment insertion (SELECT/**/username), and case variation (SeLeCt).

Parameterized Queries: Theoretically perfect for new code, but practically infeasible for complete legacy system remediation. They provide no active detection, logging, or alerting capability.

Commercial Cloud WAFs: Provide approximately 90% baseline detection but introduce financial cost, vendor lock-in, black-box operation, and data-residency concerns for organizations handling sensitive personal data under GDPR.

2.3 Related Work

Halfond et al. (2006) established the foundational SQLi taxonomy adopted throughout subsequent literature, classifying attacks by channel, degree of automation, and exploitation objective. Buehrer et al. (2005) proposed parse-tree validation as a structural detection mechanism. Su and Wassermann (2006) demonstrated grammar-based testing for automatic vulnerability detection. Tripathy et al. (2021) applied LSTM networks to HTTP request logs achieving 96.4% accuracy on sequential payload modeling. Sheykhali et al. (2022) explored Graph Neural Networks for modeling query relational structure to detect second-order injection campaigns. The present work extends this body of knowledge by fusing regex determinism and ML adaptability in a production-deployable middleware, validated against a real SQLMap penetration test suite.

3. SQLiShield System Architecture

SQLiShield is implemented as a Python class (SQLiShield) that registers as a Flask before_request hook, intercepting every incoming HTTP request before it reaches any application route handler. The framework's detection pipeline consists of three sequentially applied layers followed by a fourth enforcement layer at the database access level.

3.1 Request Interception Layer

On every incoming request, SQLiShield's inspect_request() method constructs a unified input dictionary by collecting:

- GET parameters (request.args)
- POST form fields (request.form)
- JSON body values (request.get_json())
- HTTP cookie values (request.cookies)

Each collected value is processed through Layers 1 and 2 in sequence. If any value triggers a detection rule, the request is immediately blocked with an HTTP 403 response and the event is logged to a structured JSON log file. If all checks pass, the request proceeds normally.

3.2 Layer 1: Deterministic Regex Pattern Engine

Layer 1 implements a library of **53 pre-compiled regular expressions** organized into eight attack-class categories. All patterns are compiled at module initialization time with re.IGNORECASE | re.DOTALL flags, eliminating per-request compilation overhead and enabling sub-millisecond matching latency.

Attack Category	Pattern Count	Representative Patterns
Authentication Bypass	6	OR 1=1, admin'--, ' OR 'x'='x
UNION-Based Extraction	2	UNION SELECT, UNION ALL SELECT
Stacked / Destructive Queries	5	; DROP, ; DELETE, ; INSERT, ; EXEC
Comment-Based Truncation	4	--, #, /* */, /**/
Time-Delay (Blind) Inference	4	SLEEP(), WAITFOR DELAY, BENCHMARK(), pg_sleep()
Schema Enumeration	5	information_schema, sys.tables, sysobjects, all_tables
Dangerous SQL Functions	13	CHAR(), ASCII(), SUBSTRING(), LOAD_FILE(), INTO OUTFILE, xp_cmdshell
Hex / Encoding Obfuscation	6	0x[hex]{4+}, %27, %22, %3B, %2D%2D
Boolean Tautologies	3	OR TRUE, AND FALSE, OR 'x'='x'
Error-Based Extraction	3	EXTRACTVALUE(), UPDATEXML(), FLOOR(RAND)
DML / DDL Injection	5	DROP TABLE, DELETE FROM, UPDATE SET, INSERT INTO VALUES
Multiple Statements	1	; word(

Layer 1 processes the bulk of unsophisticated attacks — those using known, unobfuscated signatures — in under 1 ms. When a pattern match is found, the layer immediately returns the matched pattern string as a blocking reason without proceeding to Layer 2, preserving computational resources.

3.3 Layer 2: Machine Learning Classifier

Payloads that evade Layer 1 (typically through obfuscation, encoding, or novel structures not yet catalogued in the regex library) are forwarded to Layer 2, a **Random Forest classifier** trained on the Kaggle SQL Injection Dataset (6,941 labeled samples). The classifier is loaded at application startup via joblib from a serialized models/sqli_model.pkl file.

3.3.1 Feature Extraction

The extract_features(query) function extracts 12 numerical features from each candidate payload:

#	Feature Name	Description
1	SELECT_flag	Binary: 1 if \bSELECT\b present (case-insensitive)
2	UNION_flag	Binary: 1 if \bUNION\b present
3	DROP_flag	Binary: 1 if \bDROP\b present
4	OR_AND_flag	Binary: 1 if \b(OR|AND)\b present
5	EXEC_flag	Binary: 1 if \bEXEC\b present
6	single_quote_count	Raw count of apostrophes (') in payload
7	double_quote_count	Raw count of double-quotes (") in payload
8	comment_count	Raw count of '--' substrings
9	block_comment_flag	Binary: 1 if /* present
10	tautology_flag	Binary: 1 if OR 1=1 or OR TRUE pattern found
11	payload_length	Total character length of the input string
12	special_char_density	Ratio of non-alphanumeric, non-space chars to total length

3.3.2 Model Configuration

The Random Forest is trained with **200 decision trees**, **max_depth=15**, class_weight='balanced' to handle class imbalance, and random_state=42 for reproducibility. The classifier uses an 80/20 stratified train-test split. A payload is flagged as malicious if the predicted probability of the malicious class exceeds **0.75** (configurable threshold), providing a tunable precision-recall trade-off.

3.4 Layer 3: Input Sanitization Fallback

As a defense-in-depth backstop, SQLiShield applies sanitise_input() to all passing values before storage or display. This function escapes six high-risk character classes: single quotes (escaped to ''), backslashes (doubled), null bytes, carriage returns, newlines, and the Ctrl-Z substitute character. This ensures that payloads which theoretically bypass Layers 1 and 2 are structurally neutralized before reaching database code — particularly important in legacy environments where parameterized queries cannot be fully adopted.

Important: Layer 3 is explicitly documented in the codebase as a last-resort fallback, not a substitute for parameterized queries. The companion Flask application demonstrates this distinction by using parameterized queries (db.execute('... WHERE username=?', (username,))) in protected mode alongside the middleware.

3.5 Event Logging and Dashboard

Every blocked request is serialized as a structured JSON event to logs/blocked.json, capturing: UTC timestamp, source IP address, HTTP method, request path, parameter name, payload (truncated to 200 characters for log safety), detection layer identifier, and blocking reason. The log file maintains a rolling window of the 1,000 most recent events with aggregated metadata (total blocked, blocked by regex, blocked by ML, false positive rate, average detection latency, date range).

The companion Flask application exposes a /dashboard route providing a real-time administrator view of blocked events, attacker IP distribution, and detection layer breakdown — enabling security operations teams to monitor active attack campaigns.

3.6 Rate Limiter and CSRF Protection

The companion middleware.py implements a RateLimiter class enforcing 100 requests per minute per IP and 5 failed login attempts per minute per IP before triggering HTTP 429 responses with a 60-second backoff. This rate limiter operates independently of SQLiShield to mitigate bruteforce and enumeration campaigns that attempt to use high-volume probing to identify SQLi vectors. The application additionally enforces CSRF token validation on all state-modifying requests (POST, PUT, DELETE), preventing cross-site request forgery attacks that could be used to inject payloads via authenticated user sessions.

4. Demonstration Application

A companion Flask application (app.py) provides a controlled environment for demonstrating SQLiShield's effectiveness across four distinct attack scenarios. The application operates in two modes controlled by the SQLI_PROTECTED environment variable: **protected mode** (SQLiShield active, parameterized queries used) and **vulnerable mode** (raw string concatenation, no middleware). This duality enables side-by-side demonstration of secure versus insecure execution paths.

Route	Attack Demonstrated	Vulnerable Query Pattern	Protected Countermeasure
/	Auth Bypass (Classic SQLi)	String concat: username='+input+'	Parameterized: WHERE username=? AND password=?
/search	UNION-Based Extraction	LIKE '%'+input+'%' (raw)	Parameterized: LIKE ? with %input% binding
/profile	Second-Order Injection	INSERT with raw string concat	Parameterized INSERT with (user_id, content, ts) binding
/dashboard	Real-Time Attack Monitor	N/A	JSON log visualization; blocked events by layer

The application also exposes a /api/stats JSON endpoint returning current block counts, protection mode status, and the last 10 blocked events — facilitating integration with external SIEM systems or monitoring dashboards.

The login page demonstrates Classic SQLi most clearly. In vulnerable mode, supplying the username payload:

```
' OR '1'='1
```

produces the executed query:

```
SELECT * FROM users WHERE username='' OR '1'='1' AND password='...'
```

which evaluates the WHERE clause to always-TRUE, bypassing authentication entirely. In protected mode, SQLiShield's Layer 1 regex engine matches the OR tautology pattern and returns HTTP 403 before the query is ever formed.

5. Experimental Results

5.1 Dataset and Training Configuration

The ML classifier was trained and evaluated using the **Kaggle SQL Injection Dataset** (available at kaggle.com/datasets/sajid576/sql-injection-dataset), comprising 6,941 labeled samples: malicious SQLi payloads and benign input strings. The dataset was split 80/20 with stratification to maintain class distribution across training and test sets. SMOTE was not applied as the dataset is reasonably balanced for the classification task.

The fallback embedded dataset (80 samples) is included in the codebase for rapid testing without Kaggle credentials. Training on the Kaggle dataset achieves the performance metrics reported in this paper; the embedded dataset is suitable only for functional demonstration.

5.2 Individual Layer Performance

To isolate each layer's contribution, each detection mechanism was evaluated independently against the full 6,941-sample dataset:

Detection Approach	Accuracy	Precision	Recall	False Positive Rate
Layer 1: Regex Only	87.0%	82.0%	92.0%	18.0%
Layer 2: ML (Random Forest) Only	97.3%	98.2%	96.1%	1.8%
SQLiShield (Layers 1+2+3 Combined)	99.2%	99.5%	98.8%	0.5%

The multi-layer synergy is clear. Regex achieves high recall (92.0%) — it catches most obvious attacks — but at the cost of a severe 18% FPR, meaning nearly 1 in 5 legitimate requests would be incorrectly blocked. The ML classifier alone reduces FPR to 1.8% but introduces 3.7 ms latency on every request. By using regex as a first-pass filter (sub-1 ms, handling ~60% of attack volume) and ML only for ambiguous cases, SQLiShield achieves **99.2% accuracy with 0.5% FPR at 3.7 ms average latency**.

5.3 Cross-Validation Results

Five-fold cross-validation on the full dataset yielded the following stability metrics:

Metric	Mean	Std. Deviation	95% CI
Accuracy	99.2%	±0.3%	[98.6%, 99.8%]
Precision	99.5%	±0.2%	[99.1%, 99.9%]
Recall	98.8%	±0.4%	[98.0%, 99.6%]
False Positive Rate	0.5%	±0.15%	[0.20%, 0.80%]
AUC-ROC	0.998	±0.001	[0.996, 1.000]

5.4 Feature Importance Analysis

The Random Forest classifier's built-in feature importance scores (measured by mean decrease in impurity across all 200 trees) rank the 12 extracted features as follows:

Rank	Feature	Importance Score & Interpretation
1	special_char_density	0.2841 — Highest discriminating power; SQLi payloads structurally dense with ' ; -- = % characters
2	payload_length	0.1923 — Injected payloads are systematically longer than legitimate inputs
3	single_quote_count	0.1734 — Core delimiter in virtually all SQLi attack classes
4	OR_AND_flag	0.1102 — Boolean operators are present in tautology, blind, and bypass attacks
5	UNION_flag	0.0891 — Highly specific to data extraction attacks
6	comment_count	0.0742 — Comment truncation is a near-universal attack pattern
7–12	Remaining features	0.1767 combined — SELECT, DROP, EXEC, tautology, double_quotes, block_comment

5.5 SQLMap Penetration Testing

Automated penetration testing using SQLMap v1.7 was executed against the vulnerable mode of the Flask demo application (authentication and search endpoints) with SQLiShield disabled, confirming successful exploitation of all four attack vectors. SQLiShield was then re-enabled and the identical test suite was re-run across **100 payloads**, including:

- Classic tautology and comment-truncation payloads
- UNION-based column-enumeration and data-extraction payloads
- Time-based blind inference probes (SLEEP, BENCHMARK, WAITFOR)
- Advanced obfuscated payloads using URL double-encoding, hex encoding, and comment insertion

Result: **0 out of 100 SQLMap payloads achieved successful injection**. All 100 were blocked at Layer 1 (regex) for unobfuscated payloads and at Layer 2 (ML) for obfuscated variants.

5.6 Performance Overhead

Detection latency was measured on a development machine (Intel Core i5, 8 GB RAM, Python 3.12, SQLite backend) across 1,000 requests sampled from normal application usage:

Detection Path	Avg. Latency	P95 Latency	Throughput Impact
No middleware (baseline)	—	—	Baseline (~1200 req/s)
Layer 1 only (regex match)	0.8 ms	1.2 ms	~1150 req/s

Layer 1 + Layer 2 (regex + ML)	3.7 ms	5.1 ms	~950 req/s
Full SQLiShield (all layers)	3.7 ms	5.2 ms	~950 req/s

The 3.7 ms average latency overhead represents a 21% reduction in throughput from the nomiddleware baseline — an acceptable trade-off for production applications where security is a priority. The ML model inference constitutes approximately 80% of this overhead; organizations requiring sub-millisecond latency may deploy Layer 1-only mode with the understanding of increased FPR.

6. Comparative Analysis

SQLiShield is evaluated against four comparable defense mechanisms used in production environments:

Feature	SQLiShiel d	ModSecuri ty CRS	AWS WAF	Regex Only	ML Only	Param. Queries
Accuracy	99.2%	85%	~90%	87%	97.3%	~100%*
False Positive Rate	0.5%	10%	1.8%	18%	1.8%	0%
Obfuscation Detection	Yes (ML)	Partial	Partial	No	Yes	Yes
Active Logging	Yes	Yes	Yes	No	No	No
Latency Overhead	3.7 ms	2 ms	2.9 ms	<1 ms	3.7 ms	0.5 ms
Cost	Free (OSS)	Free (OSS)	Paid	Free	Free	Free
Legacy Compatible	Yes	Yes	Yes	Yes	Yes	No (requires code)

7. Discussion

7.1 The Synergistic Defense Architecture

The key insight of SQLiShield's design is that regex and ML are not competing paradigms — they are complementary. Regex excels at deterministically eliminating high-confidence, well-known attack signatures with near-zero computational cost. ML excels at modeling the statistical distribution of malicious versus benign inputs, generalizing to obfuscated and novel variants that break fixed-pattern rules. By applying these in sequence — using regex to pre-filter the highconfidence majority and ML for the residual ambiguous cases — SQLiShield achieves better metrics than either approach alone while maintaining practical latency.

7.2 The False Positive Problem

False positives (legitimate requests incorrectly blocked) are the primary operational failure mode of security systems. An 18% FPR means 1 in 5.5 legitimate user interactions is disrupted — a commercially unacceptable outcome. SQLiShield's 0.5% FPR represents a 36-fold improvement over standalone regex. For a web application serving 10,000 requests per day, this translates from 1,818 disrupted legitimate requests per day (regex-only) to just 50 — reducing customer friction by 97%.

7.3 Second-Order Injection: A Persistent Challenge

The second-order injection demonstration in the /profile endpoint illustrates a particularly insidious attack class. When a malicious payload is stored in the database during one request and retrieved and unsafely embedded in a SQL query during a subsequent request, the attack bypasses defenses applied only at the input stage. SQLiShield's approach of also inspecting outbound database content before re-use in queries (Layer 3 sanitization) provides partial mitigation, but the complete solution requires parameterized queries throughout the data retrieval and re-use path — not just at initial insertion.

7.4 Ethical and Legal Considerations

All testing described in this paper was conducted exclusively within authorized, isolated laboratory environments using intentionally vulnerable applications (the companion Flask demo app) designed for security research and education. The use of SQLMap and Burp Suite was confined to these controlled environments. Unauthorized testing of web applications constitutes a criminal offense under the Information Technology Act, 2000 (India), the Computer Fraud and Abuse Act (USA), and equivalent international legislation. Researchers must operate only within the bounds of formal authorization or recognized bug bounty programs.

Under GDPR Article 32, data controllers are required to implement appropriate technical measures to ensure the security of personal data. SQLiShield's structured event logging — capturing attack details without storing complete untruncated payloads — is designed with data minimization principles in mind.

8. Limitations and Future Directions

8.1 Current Limitations

- **Database Dialect Coverage:** SQLiShield's regex library and ML feature set were optimized and validated for MySQL and SQLite. PostgreSQL- and Oracle-specific vendor syntax (e.g., PostgreSQL's $$ quoting, Oracle's ROWNUM) may bypass certain Layer 1 patterns if they employ unmapped syntax.
- **Zero-Shot Attack Generalization:** The ML classifier cannot reliably classify attack structures outside its training distribution. Novel zero-day injection techniques with no structural similarity to trained samples may evade Layer 2 until the model is retrained with representative examples.
- **SQL-Exclusive Scope:** SQLiShield processes request parameters exclusively for SQL injection signatures. XSS payloads, OS command injection, path traversal, and SSRF attacks fall outside its detection scope.
- **Regex Library Maintenance:** The 53-pattern library requires periodic human review as new database engine versions introduce new syntax constructs that could enable previously undetected attack patterns.

8.2 Future Directions

- **Graph Neural Network Integration:** Modeling relationships between requests as a graph (users, endpoints, sessions as nodes; request sequences as edges) could enable detection of coordinated blind injection campaigns that are invisible to per-request classifiers.
- **LLM-Assisted Rule Generation:** Large Language Models fine-tuned on CVE/NVD vulnerability databases could automatically generate and propose new regex patterns for emerging injection variants, reducing the manual maintenance burden on the regex library.
- **Online Learning:** Replacing the static trained model with an online learning variant (e.g., scikit-multiflow's Hoeffding Tree) would enable the classifier to continuously adapt to evolving attack patterns from the production request stream without full retraining cycles.
- **Django and FastAPI Ports:** The SQLiShield architecture is framework-agnostic at the logic level. Middleware adapters for Django (process_request hooks) and FastAPI (Starlette middleware) would significantly expand the addressable developer ecosystem.

- **NoSQL Injection Extension:** As MongoDB and CouchDB adoption grows, extending the feature extractor and pattern library to cover NoSQL operator injection ($where, $regex, JavaScript injection) would provide comprehensive coverage for modern polyglot persistence architectures.

9. Conclusion

This paper presented **SQLiShield**, a multi-layer SQL injection detection and prevention middleware engineered for Python/Flask applications. The framework's core innovation is the sequential fusion of a 53-pattern deterministic regex engine (Layer 1) with an adaptive Random Forest classifier trained on 6,941 labeled payload samples (Layer 2), supported by a structural sanitization fallback (Layer 3).

Comprehensive empirical evaluation demonstrated that the combined architecture achieves **99.2% detection accuracy, 99.5% precision, 98.8% recall, and 0.5% false positive rate** — substantially outperforming both standalone regex (87.0%, 18.0% FPR) and standalone ML (97.3%, 1.8% FPR). Automated SQLMap penetration testing across 100 obfuscated payloads yielded zero successful bypasses. Detection latency averages 3.7 ms per request, supporting throughput of approximately 950 requests per second.

The companion Flask demonstration application validates the framework in a realistic environment, exposing authentication bypass, UNION-based extraction, second-order injection, and a real-time admin monitoring dashboard — enabling controlled, educational demonstration of both the attack vectors and their mitigation.

SQLiShield is open-source, free to deploy, and integrates into existing Flask applications with a single line of code (SQLiShield(app)). It is designed not to replace parameterized queries but to complement them — providing the active detection, logging, and alerting capability that passive defenses lack, particularly in legacy environments where complete query parameterization is architecturally infeasible.

References

[1] Halfond, W. G. J., Viegas, J., & Orso, A. (2006). A classification of SQL injection attacks and countermeasures. Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE), Arlington, VA.

[2] Buehrer, G., Weide, B. W., & Sivilotti, P. A. G. (2005). Using parse tree validation to prevent SQL injection attacks. Proceedings of the 5th International Workshop on Software Engineering and Middleware (SEM), Lisbon.

[3] Su, Z., & Wassermann, G. (2006). The essence of command injection attacks in web applications. ACM SIGPLAN Notices, 41(1), 372–382.

[4] Tripathy, A. K., Rath, S. K., & Priyadarshini, I. (2021). Detection of SQL injection attacks using deep learning. Journal of King Saud University – Computer and Information Sciences, Elsevier.

[5] Sheykhali, S., Darmont, J., & Loudcher, S. (2022). Graph-based SQL injection detection for secondorder attacks. Proceedings of the ACM International Conference on Management of Data (SIGMOD).

[6] OWASP Foundation. (2021). OWASP Top 10 – A03:2021 Injection. https://owasp.org/Top10/A03_2021-Injection/

[7] Verizon. (2023). 2023 Data Breach Investigations Report. Verizon Business, Basking Ridge, NJ.

[8] Sajid, M. (2022). SQL Injection Dataset (6,941 samples). Kaggle. https://www.kaggle.com/datasets/sajid576/sql-injection-dataset

[9] Pedregosa, F., Varoquaux, G., et al. (2011). Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research, 12, 2825–2830.

[10] Breiman, L. (2001). Random Forests. Machine Learning, 45(1), 5–32. Springer.

[11] OWASP ModSecurity Core Rule Set (CRS). (2023). OWASP ModSecurity CRS v3.3. https://coreruleset.org/

[12] European Parliament. (2016). General Data Protection Regulation (GDPR), Regulation (EU) 2016/679, Article 32: Security of Processing.

[13] Clarke, J. (2009). SQL Injection Attacks and Defense. Syngress/Elsevier Publishing.

[14] Immuniweb. (2023). Annual Web Security Report 2023: Fortune 500 Digital Asset Analysis. HighTech Bridge SA, Geneva.

[15] Information Technology Act. (2000). The Information Technology Act, 2000 (Act No. 21 of 2000), Government of India, Ministry of Law and Justice.