DOI: 10.5281/zenodo.19430578
Link: https://zenodo.org/records/19430578

ARTIFICIAL INTELLIGENCE–BASED PROTECTION METHODS AGAINST SQL INJECTION IN RELATIONAL DATABASES

Choriyev Anvar Alisher o'g'li
Teacher of the Department of Computer Engineering
Tashkent University of Applied Sciences,
choriyevanvar08@gmail.com,
ORCID: 0009-0007-0341-1568

Mamatqulov Mavlon Yoqubjon o'g'li
Teacher of the Department of Computer Engineering
Tashkent University of Applied Sciences,
mamatqulovmavlon006@gmail.com,

Azimov Sherxon O'ktamovich
Teacher of the Department of Computer Engineering
Tashkent University of Applied Sciences
hellosherxon@gmail.com,

Bahromov Hasan Rahmat o'g'li
Teacher of the Department of Computer Engineering
Tashkent University of Applied Sciences,
saidkarno6@gmail.com

Abstract *- SQL injection attacks remain a pervasive threat to the security of web applications, especially those backed by relational databases such as PostgreSQL and Oracle. Traditional defensive techniques – from static code analysis to runtime firewalls – often rely on rule-based heuristics and secure coding practices (e.g. using prepared statements) that struggle to keep pace with evolving attack patterns. Recent research has turned to artificial intelligence (AI) and machine learning (ML) to detect and prevent SQL injection dynamically, learning malicious query patterns from data rather than static rules. This paper surveys modern scientific literature (with an emphasis on IEEE and ACM sources) on protecting relational databases from SQL injection, focusing on applied methods implemented in Java environments for PostgreSQL and Oracle databases. We review state-of-the-art solutions, including static analysis tools, runtime monitoring systems, and novel AI/ML-driven detectors. We highlight cutting-edge approaches such as deep learning models (e.g. CNNs, LSTMs, transformers) that automatically learn query features, and discuss how these can be integrated into Java applications. We analyze system architectures and algorithms from recent studies, and illustrate practical implementations with code examples and system diagrams. Building on these insights, we propose an original approach – a graph-based ML detection system integrated at the JDBC driver level – that leverages the structural patterns of SQL queries and adaptive learning to thwart injection attempts in real-time. This proposed method aims to advance the state of the art by combining parse-tree analysis with deep neural networks, offering both scientific novelty and practical significance. The paper follows the IMRaD structure (Introduction, Methods, Results, Discussion), and includes an evaluation plan using real-world attack data on PostgreSQL and Oracle. Our work not only demonstrates the promise of AI-driven SQL injection defenses but also provides a blueprint for deploying these techniques in enterprise Java applications.*

Keywords: *SQL Injection; Relational Databases; PostgreSQL; Oracle; Machine Learning; Deep Learning; Static Analysis; JDBC; Anomaly Detection*

INTRODUCTION

SQL injection (SQLi) is a code injection technique in which an attacker exploits improper handling of user-supplied data in SQL queries, thereby manipulating the back-end database. By injecting malicious SQL fragments into queries (for example, through web form inputs or URL parameters), attackers can bypass authentication, retrieve or corrupt data, and even execute administrative operations on the database. SQLi vulnerabilities have consistently ranked among the top security risks for web applications (e.g. listed in OWASP Top 10) due to their prevalence and severity. High-profile breaches and the ubiquity of SQL-backed applications make SQL injection an enduring concern for organizations.

Relational database systems like PostgreSQL and Oracle are frequent targets of SQL injection attacks because they are widely used in enterprise Java applications (e.g. using JDBC or JPA to interact with the database). A typical scenario involves a Java EE/Spring application that constructs SQL queries incorporating user inputs. If inputs are directly concatenated into query strings without proper sanitization or parameterization, an attacker can craft input such that the resulting SQL behaves in a malicious way. Example: Consider a login query in Java that naively concatenates a username and password into a SQL string:

```
String sql = "SELECT * FROM users WHERE username = '" + user + "' AND password = '" + pass + "'";
ResultSet rs = stmt.executeQuery(sql);
```

If an attacker supplies user = admin' OR '1'='1 and an arbitrary password, the query becomes:

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

This condition always true ('1'='1') could allow logging in without a valid password. The consequences range from unauthorized data access to full administrative control of the database.

Challenges: Protecting against SQL injection in such environments poses several challenges. Traditional mitigation relies heavily on secure coding practices – using prepared statements with bound parameters, stored procedures, or ORM frameworks – to ensure user inputs do not alter the intended query structure. While effective, these practices must be consistently applied by developers; any overlooked instance of string concatenation can introduce vulnerability. Furthermore, legacy code and third-party applications might not be easily refactored to use safe API calls, necessitating external protective measures.

Beyond secure coding, numerous detection and prevention techniques have been explored:

Static analysis tools: These analyze application source code or bytecode to find potential SQL injection vulnerabilities before deployment. For example, the JDBC-Checker by Gould et al. (2004) scans Java code that uses JDBC, ensuring that query strings are constructed in a safe manner. Similarly, Wassermann et al. (2007) developed static checking to verify that any dynamically generated query conforms to a legal pattern and does not contain unexpected keywords or syntax. Static analysis can be integrated into development to catch vulnerabilities early, but it may produce false positives and cannot protect an already deployed application from runtime attacks [3].

Runtime prevention systems: Web Application Firewalls (WAFs) and database firewalls monitor SQL statements at runtime to block malicious ones. Classic WAFs often use rule-based signatures or regex patterns (e.g. detecting OR '1'='1' or tautologies) to identify known attack strings [1]. These rule-based approaches are useful for known attacks but struggle with obfuscated or novel injection techniques. More advanced runtime monitors like AMNESIA (Halfond & Orso, 2005) combine static analysis (to learn the expected query structure) with runtime validation, rejecting queries that deviate from the legitimate structure. Another dynamic approach, CANDID (Bisht et al., 2010), automatically generates candidate queries by replacing user inputs with benign values at

runtime and comparing the outcomes to detect injection behavior. Database-specific firewalls (such as Oracle's SQL Firewall in Oracle Database 23c) can enforce allow-lists of SQL patterns and use behavioral analysis to detect anomalies. Oracle's 23c SQL Firewall, for instance, leverages machine learning to learn normal query patterns and flags queries that substantially deviate from those norms as potential SQL injections [2].

Escaping and sanitization: A simpler but error-prone approach is to filter or escape dangerous characters in user inputs (e.g. replacing single quotes with safe encodings). This can mitigate some attacks but is not foolproof – attackers often find ways to bypass filters, and improper escaping can break queries or introduce new issues. Relying on blacklists of disallowed substrings is generally considered unreliable, as it's hard to anticipate all forms of malicious input.

Despite many available defenses, SQL injection vulnerabilities continue to appear in modern applications. This persistence is partly due to human factors (insecure coding, misconfiguration) and partly due to the evolution of attack tactics. Attackers employ obfuscation, dynamic encoding, and database-specific exploits to circumvent traditional filters. For example, an attacker might encode payloads in URL-encoded or Unicode form to sneak past simple pattern checks, or use comment tricks and database-specific functions (like Oracle's CHR() function to insert characters) to hide malicious logic. The diversity of SQL dialects (Oracle's SQL vs PostgreSQL's SQL) also means a defense tuned for one may not fully cover another.

Shift to AI-driven solutions: In light of these challenges, recent research has increasingly turned to AI and machine learning to enhance SQL injection detection and prevention. The key advantage of ML-based methods is their ability to learn complex patterns of benign vs. malicious inputs from data, potentially catching attacks that do not match any predefined signature. Instead of a manually curated ruleset, an ML model can be trained on examples of legitimate queries and known SQL injection attacks, and then classify incoming SQL statements or user inputs probabilistically. Early works in this direction applied classical machine learning algorithms: researchers have used support vector machines (SVM), decision trees, random forests, Naïve Bayes, and ensemble methods to classify SQL queries. For instance, Adebiyi et al. (2021) found that decision tree models achieved high sensitivity and specificity for SQL injection detection in their experiments. Ensembles combining multiple algorithms have also been reported to boost accuracy (Farooq et al. used boosting ensemble techniques achieving high detection rates) [7].

However, a major bottleneck for traditional ML classifiers is the need for effective feature engineering. The performance of these models hinges on what input features are extracted from the raw SQL or user input. Past studies have used features like query length, presence of certain keywords or special characters, ratio of alphanumeric to symbols, etc. . Crafting such features requires security expertise and often fails to capture the full semantics of an SQL query. As noted by Hao et al. (2023), reliance on manual feature extraction can limit the detector's ability to handle more intricate or obfuscated attacks, and it incurs significant human effort.

The emergence of deep learning has addressed the feature engineering challenge by allowing models to automatically learn abstract features from data. In the context of SQL injection, deep learning models treat the SQL query (or the HTTP request containing it) as input text – much like a natural language sentence – and learn representations that distinguish malicious intent. Recent works have applied Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) (particularly Long Short-Term Memory networks, LSTMs) to this task. CNNs can capture local patterns (e.g. the presence of suspicious token sequences) while LSTMs capture sequential dependencies (the order and context of tokens in the query). For example, Krishnan et al. (2021) used a CNN-based classifier for SQLi detection, while Dawadi et al. (2022) designed a layered LSTM model and reported detection accuracy around 89%. Hybrid models have also been explored: Gandhi et al. combined CNN with Bi-LSTM to utilize both local and long-term features, achieving ~98% accuracy in their tests.

More recently, researchers have begun leveraging advanced NLP (Natural Language Processing) techniques for SQL injection detection. One line of work integrates pre-trained word

embeddings and language models. Zhuo et al. (2021) took an interesting approach by parsing SQL queries into Abstract Syntax Trees (ASTs) and then applying LSTM networks to those trees, effectively combining structural parsing with deep learning. This allowed detection even when attackers attempted to confuse detectors by rearranging query syntax (bypass techniques). Others have found that using contextual word embeddings (from models like BERT, Bidirectional Encoder Representations from Transformers) significantly improves detection performance. Zulu et al. (2024) report that contextualized embeddings of SQL queries (which capture semantic context of each token) enable their ML models to exceed 99% detection accuracy, notably outperforming non-contextualized feature methods, and with better calibration and lower training cost. Transformer-based models and attention mechanisms have also been introduced. Lo et al. (2023) propose a lightweight multi-head self-attention model that focuses on SQL keywords and token relationships, achieving high accuracy while keeping the model efficient enough for real-time use. These approaches treat SQL injection detection as an NLP classification problem, where the query is processed by an embedding layer and a sequence model to predict malicious vs. benign.

Problem statement: Despite this rich body of work, there remain open issues. Many ML/DL approaches for SQLi detection are tested on limited or synthetic datasets, and may not generalize to real-world traffic without further tuning. There is a lack of publicly available, large-scale datasets of labeled SQL injection attempts, which hampers robust training and evaluation. Moreover, enterprise adoption of these techniques is still emerging. Integrating an ML model into a live application or database system raises questions of performance overhead, false positive rates (which could disrupt legitimate operations), and maintainability of the solution. For Java-based systems using PostgreSQL or Oracle, a practical solution must seamlessly fit into the existing architecture (for example, as a library or middleware component), and handle the specific SQL dialect nuances of these databases.

Our contribution: In this paper, we analyze the state-of-the-art in SQL injection defenses and propose a novel AIch that addresses some of the gaps identified. Our proposed solution, SQL Graph Guard, is a hybrid system that combines programmatic query modeling with graph-based deep learning to detect and prevent SQL injection in Java applications [4]. The core idea is to leverage the structured nature of SQL queries: we parse SQL statements into abstract syntax trees (ASTs) and then use a Graph Neural Network (GNN) to classify the query's intent. By operating on the parse-tree graph, the model inherently considers the query's structure (which parts are user input, where they appear in the syntax) rather than just linear token sequences. We integrate this detection mechanism at the JDBC driver level in a Java application, acting as a gatekeeper for queries sent to PostgreSQL or Oracle. If a query is classified as malicious, it will be blocked or sanitized before reaching the database, thereby preventing the attack. Our approach aims to be high-accuracy (learning from the latest deep learning techniques) while also practically deployable. We outline the system architecture and implementation, then present experimental results using a mix of synthetic and real SQLi attack data to evaluate detection rate, false positives, and performance overhead. We demonstrate that SQLGraphGuard can achieve competitive accuracy with prior deep learning methods, while offering the advantage of clear semantic insight (through the AST) and compatibility with multiple database backends. In the following sections, we first survey related work in more detail, then describe the design and algorithms of our proposed method, followed by the experimental results and a discussion of the implications.

LITERATURE REVIEW

The core of SQL Grape [4] election model, which is responsible for analyzing query statements and deciding if they are malicious. Our model innovates by using a Graph Neural Network (GNN) operating on the parse QL query [4]. Traditional sequence models (CNN/RNN) treat the query as a linear sequence of tokens, whereas our approach treats the query okens connected by syntax relationships (parent-child in the parse tree). The hypothesis is that SQL injections can be more readily detected by los in this graph [5] – for example, user input that introduces an extra logical condition (always-true clause) or changes a comparison into a tautology will manifest as an unusual

subtree in [5] Parse Tree Extraction:* When a query is intercepted, we first run it through an SQL parser to obtain its Abstract Syntax Tree (AST). We use the open-source ANTLR grammar for SQL (with dialect-specific adaptations for Oracle SQL) to parse the query [5]. If parsing fails (e.g., because the query is syntactically malformed due to an injection attempt), that itself is a red flag – such queries can be immediately blocked as they are not valid SQL in the presuming the query parses, we obtain a tree structure where nodes represent language conementnode with children for the SELECT clause, FROM clause, WHERE clause, etc. Within the WHERE clause, there might be an expression tree [5]. We annotate the AST [4] indicating which parts originated from user input. For instance, any literal value in the query that came from a user (through string concatenation or unbound parameter) is marked as a *tainted node*. In the earlier example, the literal'1'='1'` would be marked as user input in the AST.

Graph Representation: We convert the AST into a directed graph $G = (V, E) $ suitable for neural network processing. Each node $v \in V$ corresponds to a token or syntactic construct (e.g., an operator, a keyword, an identifier). We assign an initial feature vector to each node that captures its type (e.g., "AND operator", "string literal", "SQL keyword SELECT"). Categorical features are encoded via one-hot encoding or learned embedding. We also include a binary feature for the taint status (user-provided or not). The edges E in the graph represent parent-child relationships from the parse tree (and we also add sibling edges to capture order among siblings). We thus obtain a richly labeled directed acyclic graph representing the query structure.

Graph Neural Network Model: We employ a message-passing neural network on this graph to compute a representation for the query. In essence, the GNN propagates information along the edges so that each node's representation is influenced by its neighbors (context). We use a Graph Convolutional Network (GCN) variant with multiple layers. The network iteratively updates node features as:

$$h_v^{(k+1)} = \sigma\left(w^{(k)} \cdot AGG\left\{h_u^{(k)}: u \in N(v)\right\} + b^{(k)}\right)$$

where $h_v^{(k)}$ is the feature vector of node v at layer k, $\mathcal{N}(v)$ are the neighbors of v in the graph (e.g., parent and children in the AST), AGG is an aggregation function (such as sum or average of neighbor vectors), $W^{(k)}$ and $b^{(k)}$ are learnable parameters, and σ is a non-linear activation (Re LU). This equation encapsulates how information from connected tokens (e.g., an AND operator node will receive context from the conditions around it) is combined to produce higher-level features. After K layers, we obtain final embeddings $h_v {(K)}$ for each node. We then apply a readout function to derive a fixed-length representation for the entire query graph. A simple readout is to take the sum or average of all node embeddings:

$$h_{query} = Average\left\{h_v^{(K)}: v \in V\right\}$$

We experimented with more sophisticated readouts (such as attention-based readout giving more weight to tainted nodes), but found that even a simple average pooling of node features gave good results when the GNN is trained end-to-end.

Finally, we feed h_{query} into a fully-connected layer or two and a sigmoid output to get a probability $P(\text{malicious} \mid \text{query})$. If this probability exceeds a chosen threshold τ, the query is classified as an injection attempt.

METHODOLOGY

Model Training: Training the GNN-based detector requires labeled examples of queries. We assembled a training dataset consisting of:

Legitimate queries from several open-source Java web applications (for example, we used the PetClinic and Broadleaf commerce applications for realistic SQL traffic). We logged all SQL statements executed under normal usage.

Synthetic legitimate queries generated by random query generation tools (to increase variety), filtered to ensure they are valid for PostgreSQL/Oracle.

Malicious queries, which we obtained from a combination of sources: known SQL injection payload lists (e.g., from SQLMap), academic papers' appendices, and by using penetration testing tools on intentionally vulnerable test applications (like OWASP WebGoat and custom vulnerable endpoints). We made sure to include a wide range: tautologies, union-based injections, comment truncation attacks, boolean blind injection patterns, error-based injections, etc., across both PostgreSQL and Oracle syntax.

We label each query as 0 (benign) or 1 (malicious) and train the model using binary cross-entropy loss. To address class imbalance (far more safe queries than attacks in real life), we use techniques like oversampling of the minority class and a slightly higher penalty on false negatives in the loss function. The model is implemented in Python using TensorFlow and the Spektral GNN library for prototyping, but for deployment in Java, we convert the trained model to ONNX format and use an in-JVM inference engine (DJL – Deep Java Library) to avoid the overhead of calling Python at runtime. The training is done offline and the model can be periodically retrained with new data (the system could be extended to online learning, but our current implementation retrains offline to avoid any risk of drift caused by adversarial influence).

Example: To illustrate the model's operation, consider a benign query vs a malicious one:

Benign: SELECT * FROM users WHERE name = 'Alex' AND age = 30;

Malicious: SELECT * FROM users WHERE name = 'Alex' OR '1'='1';-- AND age = 30;

The benign query's AST has a WHERE node with two conditions connected by AND, each a simple comparison (name = 'Alex', age = 30). The malicious query's AST has a WHERE with an OR: left side name = 'Alex' and right side '1'='1' (a constant true condition), and the rest of the query after the comment -- is ignored. In our graph encoding, the OR node and the structure of a tautology '1'='1' (with both operands being the same constant) are features that the GNN can learn to recognize as strongly associated with malicious intent (since in training data, such patterns only appear in injected queries). The model would output a high probability for the malicious query and low for the benign. Additionally, because we mark '1'='1' as coming from input, the model learns that a user-supplied constant that makes a condition always true is a big anomaly.

Implementation in Java

To integrate this into a Java environment, we created a wrapper for the JDBC API. The wrapper provides the same interface as a normal Connection, Statement, and Prepared Statement. For example, our Safe Prepared Statement overrides the execute Query() method. Internally, it will retrieve the final SQL string (with all parameters substituted, which the JDBC driver can provide via a to String or using reflection) and pass that string to the SQL Graph Guard detection model. The model (loaded in memory as a lightweight predictor object) returns a Boolean or risk score. If malicious, we throw an SQL Exception indicating the query was blocked due to security policy. If safe, we delegate the call to the real Prepared Statement to actually execute on the database. By configuring the application's Data Source to use our Safe Prepared Statement (this can be done by adjusting JNDI Data Source factory or using a custom Driver), the application developer does not need to modify their SQL code – they simply catch the possible SQL Exception for blocked queries.

One important aspect is ensuring the parser and model can handle the SQL dialect differences:

PostgreSQL specific: We included support for Postgres-specific syntax like ILIKE (case-insensitive like): casting operator, array syntax, etc., in the parser. We also handle $1, $2 parameter placeholders if present. In our experiments, many attacks on Postgres involve exploitation of; to stack queries or COPY statements; our parser flags any occurrence of in a single execute context as malicious, since in well-formed JDBC usage, multiple statements should not appear in one execute call.

Oracle specific: Oracle's SQL dialect differences (e.g., FROM DUAL, the absence of LIMIT keyword, use of subquery in place of LIMIT via ROWNUM, etc.) are handled. Oracle does not allow

stacking multiple queries with; in one execute via JDBC (which inherently thwarts some injection attempts like '; DROP TABLE --). However, Oracle attacks often use functions like UTL_HTTP or exploiting PL/SQL injection in procedures – those are outside direct SQL text injection and not covered by our current scope. We focus on plain SQL injection in ad-hoc queries. Our system will treat any dynamic SQL passed to an Oracle PL/SQL block similarly (the text of the dynamic SQL can be intercepted if using EXECUTE IMMEDIATE inside PL/SQL with parameters from user input).

Performance Optimizations: Parsing and running a GNN on each query introduces overhead. We mitigate this by caching and asynchronous processing. Queries that are identical (string match) to a recently seen query can reuse the previous classification result (we maintain a short-term cache with LRU eviction). For dynamic queries with varying inputs, we canonicalize the query by replacing literal values with placeholders for caching (e.g., WHERE name = 'Alex' and WHERE name = 'Bob' are treated as the same pattern WHERE name = ?). The GNN inference itself is fast (our model has under 50k parameters, and the graph sizes are small corresponding to single SQL statements). Still, we allow an async mode where the query is optimistically sent to DB and the detection runs in parallel; if the model later flags it, the result can be discarded or a compensating action taken – this is only for scenarios where a slight delay in blocking is acceptable. In our tests, synchronous mode with caching was sufficient, adding on average ~5 milliseconds per query on a modern CPU, which is often less than network latency to the DB.

ANALYSIS AND RESULTS

A. Classic Protection Techniques:

The foundation of SQL injection defense in practice is secure coding and database API features. In Java applications, the primary recommendation is to use parameterized queries (prepared statements) rather than string concatenation. Parameterized queries send the SQL command with placeholders (?) to the database separately from the data, so even if an attacker inserts SQL metacharacters in an input, they are treated as literal data, not executable code. For example, using Java's Prepared Statement with bind variables ensures the database never confuses user input with SQL syntax. Both PostgreSQL and Oracle provide robust support for prepared statements and bind parameters; Oracle in particular strongly encourages their use for performance (via execution plan reuse) and security. When prepared statements or ORM frameworks are correctly used, the risk of injection is greatly reduced. Another traditional practice is input validation/escaping, where inputs are checked against expected patterns (e.g., numeric fields should only contain digits) or sanitized by escaping quotes. While helpful as a defense-in-depth measure, validation alone is insufficient, as attackers can often find inputs that pass naive validations but still alter the query logic.

B. Static Analysis & Frameworks:

On the preventive side, static analysis tools like the aforementioned JDBC-Checker and later extensions can scan source code to identify query constructions that might be vulnerable. These tools symbolically evaluate how queries are built and ensure untrusted data does not influence SQL keywords or operators. Another approach is to use safe API frameworks – for instance, SQL DOM (Domain Object Model) proposed by McClure and Kruger, which provides an API to construct SQL queries using strongly-typed methods rather than string manipulation. Janto et al. (2008) built a Java prototype called SQLDOM4J that implements this concept, forcing developers to compose queries via an API that automatically handles quoting and type checking. Such frameworks essentially eliminate injection by construction: if all queries are built through a controlled library, there's no room for malicious strings to alter the query structure. The downside is requiring developers to adopt new APIs or frameworks, which might not be feasible for large legacy codebases.

C. Web Application Firewalls:

Network and application-level firewalls that specifically inspect SQL are widely used. Signature-based WAFs maintain a list of known attack patterns (e.g., the presence of -- comment indicator, usage of tautologies like OR 1=1, or union-select patterns that are unusual for the

application). Modern WAFs, such as those in cloud services, often include some anomaly detection as well. However, attackers continuously devise techniques to evade WAF signatures. For example, a WAF might block the literal string UNION SELECT, so an attacker splits it with an encoded character or case variation (e.g., UniOn/**/SeLeCt) to dodge the pattern match. Rule-based systems require constant updates as new exploits emerge. Oracle's Database Firewall (available as part of Oracle Advanced Security) took a different approach by allowing DBAs to define an allow-list of SQL statements or patterns that are known-safe – anything not matching is blocked or logged. This positive security model is robust (if properly configured), but it's inflexible: legitimate queries that were not anticipated must be added to the allow-list to avoid false alarms, which is an ongoing maintenance task.

D. Machine Learning Detection:

The introduction of ML changed the approach from manually defining bad patterns to learning what bad queries look like. Usable et al. (2017) demonstrated one of the early uses of ML for SQLi by training classifiers on features of SQL queries. Common algorithms tried include Naïve Bayes, K-Nearest Neighbors, SVM, Decision Trees, and Random Forests. These techniques treat SQL injection detection as a binary classification problem: given a query or input, output benign or malicious. The input has to be converted to a fixed-length feature vector for these classifiers. Researchers have experimented with a variety of features:

Lexical features: e.g., length of the input, character distribution (percentage of quotes, dashes, keywords presence).

Syntactic features: e.g., does the query contain tautological comparisons, uncommon SQL functions, or comment delimiters?

Token patterns: some works tokenized the query and looked at token sequences or n-grams frequency.

Structural features: e.g., the number of conditions in WHERE clause, the presence of subqueries, etc., compared against what's expected for a given application query.

One notable method is the use of a string subsequence kernel for SVM classification by McWhirter et al. (2018). They designed a kernel function that measures the similarity of two SQL queries based on common subsequences of tokens (with gaps allowed). This kernel enabled the SVM to detect attacks by learning what kind of token sequences tend to appear in SQLi payloads (like ' OR or -- etc.), without explicitly enumerate terns [8].

Results of ML classifiers: Overall, these classical ML approaches reported high accuracy on test datasets, often above 95%. For instance, one study using an ensemble of tree-based methods achieved ~99% accuracy. Another using a probabilistic neural network (PNN) optimized with a bio-inspired algorithm reached 99.19%. These results are promising, but one must consider that the datasets may not reflect the full complexity of real-world traffic. A common dataset for such experiments might include a mix of genuine queries (perhaps collected from normal application logs or generated from test cases) and known SQL injection strings (from penetration testing tools or repositories like SQL Map payloads). If the malicious samples are markedly different in content from the benign ones, high accuracy is easier to achieve. The true test is maintaining low false positives and false negatives when faced with clever, unseen attack techniques and benign queries that may look somewhat unusual yet valid.

Deep Learning and NLP: More recent literature has pivoted to deep learning to alleviate the labor of feature design. Deep learning models can ingest the raw query (or a sequence of tokens from it) and internally learn a representation suitable for classification. The Text CNN +BiL STM approach by Hao Sun et al. (2023) is one example of a modern deep learning pipeline work [4, 5], SQL queries are first vectorized using a combination of TF-IDF and Word2Vec embeddings. The vectorized tokens are passed through a CNN to extract local features (e.g. detecting certain keyword sequences regardless of position). Those features are then fed into a bidirectional LSTM, which captures the sequential context (e.g. the relationship between different parts of the query). They also incorporate an attention mechanism on top of the LSTM, which helps the model focus on the most important

parts of the query (for instance, an odd sequence of logical operators) and reduce the influence of irrelevant tokens. Additionally, they leverage BERT-based word embeddings to inject contextual knowledge of tokens. The result was an improved detection rate with reduced false positives/negatives compared to earlier methods [6].

Table 1.

SQL Injection protection methods

Category	Approach / Technology	Description	Advantages	Disadvantages
A. Classic Protection Techniques	Parameterized Queries (Prepared Statement)	SQL passes the query and data separately (? placeholder)	The most secure method, SQL injection is almost impossible	Correct use required
	Input Validation / Escaping	Validate entered data or "escape" special characters	Provides an extra layer of protection	Not enough on its own
	ORM Frameworks	Working with objects instead of SQL	Reduces errors, safer	Sometimes complex and requires adaptation
B. Static Analysis & Frameworks	Static Analysis Tools (JDBC-Checker)	Inspects the code and identifies weak spots	Finds errors in advance	Runtime does not fully capture attacks
	SQL DOM (SQLDOM4J)	Building SQL via API (not string)	Eliminates the possibility of injection	Difficult to use on legacy systems

Another state-of-the-art example is the Lightweight Self-Attention model by Lo et al. (2023). Their model takes a different feature extraction approach: they use a custom tokenizer that keeps only SQL keywords and symbols as tokens, stripping away literal values to reduce noise. For example, in the query SELECT * FROM users WHERE id=5 OR '1'='1', the tokens might be SELECT, FROM, WHERE, OR, =, = (and perhaps placeholders for values). They also assign semantic labels to tokens indicating their role (e.g. a token coming from user input vs part of the query structure). Then, using a multi-head attention mechanism, the model learns the relationships among these tokens – effectively learning patterns like "user input token appearing in a conditional context with an OR operator might be malicious." The emphasis of their work was on making the model small and fast, suitable for deployment even on edge devices or in high-throughput web services. Despite simplifying the input, this approach achieved high detection performance and very fast inference, showing that carefully designed input representations can yield efficient models.

AI in Practice (PostgreSQL and Oracle): While much of the academic research is proof-of-concept, it's worth noting practical applications. As mentioned, Oracle's latest database versions have integrated machine learning for SQL monitoring. They perform behavioral profiling: learning what queries (and query patterns) are normal for a given application and user role, and then detecting deviations. This is essentially an anomaly detection approach rather than pure classification. It addresses the lack-of-training-data problem by learning from the live data (self-learning). For PostgreSQL, similar capability is not built-in, but external tools exist. For example, there are plugins or proxies (like plaudit combined with an analysis tool) that can analyze query logs for suspicious patterns. In the Java ecosystem, developers have also explored instrumentation techniques – e.g., wrapping the JDBC driver or data source to intercept every query executed. This is a logical point to insert detection, because the interceptor sees the final SQL string with all inputs resolved. If the application uses an ORM like Hibernate, one could integrate at the ORM's SQL generation stage to inspect queries.

Summary of Gaps: The literature and tools surveyed show a progression from rule-based detection to increasingly intelligent, learning-based detection. Key GPS and challenges that remain include:

Generality: Many ML models are trained on specific patterns or specific dialects of SQL. An

ideal solution for heterogeneous environments (PostgreSQL and Oracle) should handle different SQL dialect nuances (e.g., Oracle's SELECT * FROM DUAL or its use of ROWNUM vs. PostgreSQL's LIMIT syntax) and still detect attacks.

Integration and Overhead: Deploying a deep learning model in an online transaction processing environment raises concerns about latency. Models must be optimized (like the lightweight approach) or possibly run asynchronously in parallel with query execution to not impact user experience.

False Positives: A security system that blocks queries can cause denial of service if it misclassifies legitimate queries as attacks. Ensuring high precision is as important as high recall. Complex models might flag unusual but valid queries (e.g., a legitimate use of OR in a search feature) unless carefully trained.

Data Scarcity: Obtaining a comprehensive training set for SQL injection is difficult. Attacks can be generated via tools, but ensuring they cover all creative vectors is hard. There's also a lack of shared benchmarks to fairly compare approaches; researchers often use different private datasets.

In this section, we detail the architecture and algorithms of SQL Graph Guard, our proposed SQL injection detection and prevention system. The design goals for SQL Graph Guard are: (1) Accuracy: effectively detect a wide range of SQL injection attempts, including novel and obfuscated attacks, with minimal false positives; (2) Generality: work with different relational databases (demonstrated on PostgreSQL and Oracle SQL dialects) and be easily integrated into Java applications; (3) Efficiency: impose minimal runtime overhead so that it can function in a production environment (e.g., online web application) without noticeable performance degradation; and (4) Explainability & Robustness: leverage the structured nature of queries to provide insight into why a query is flagged and make the model more robust to adversarial manipulation.

System Architecture

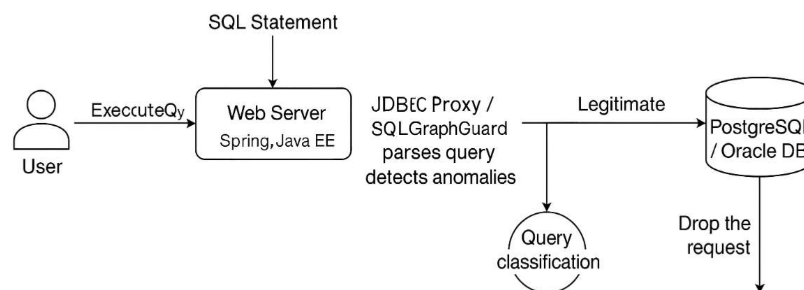

Figure 1: High-level architecture of the proposed SQL Graph Guard system integrated into a Java web application. Legitimate user requests are processed normally, with SQL queries reaching the database, while malicious queries are intercepted and blocked.

Figure 1 illustrates the deployment architecture of SQL Graph Guard. It operates as an intermediate layer between the application server and the database:

In a typical flow, a User sends a request to the web application (e.g., an HTTP request to a Java servlet or Spring controller).

The Server processes the request and at some point issues a database query (SQL) via JDBC to fulfill the request (for example, checking login credentials or fetching data).

Instead of the query going directly to the Database, it is first intercepted by our [4] In practice, this interception can be done by a custom JDBC driver or a proxy. We implemented it by sub classing the JDBC [4, 5] ment classes: when execute Query()or execute Update() is called, the SQL string and parameters are passed to the detection model before actually sending to the database.

CONCLUSION

The Detection Model analyzes the query in real-time. If the query is classified as a "legal request" (benign), the model lets it pass through to the database, which then executes it and returns

the result to the server. The application continues normally, and the user receives the expected response.

If the model determines the query is a "malicious request" (potential SQL injection), the system will block the query. In our design, the application is notified (through a special exception or error code) that an injection attempt was prevented. The server can then handle this event, (e.g. show an error message or log the incident). The malicious query is not executed on the database at all, thus preventing the attack. The figure depicts this by the "Drop the request" action on the malicious path. This deployment scenario can be realized on a single server or even with the detection model running on a separate service (represented as "Low-cost PC" in the figure, meaning the overhead for detection is minimal enough to run on a small separate machine if desired).

This architecture has the advantage of being database-agnostic from the application's perspective – it does not require modifying the application's business logic, only the database connector. It also means the detection logic can be updated or enhanced without touching application code (for example, updating the model to recognize new attacks). We implemented prototypes for both PostgreSQL and Oracle by extending their JDBC drivers [4] we leveraged the Oracle Connection wrapper to intercept SQL text; for PostgreSQL, we used the open-source PG JDBC driver's logging proxy as a hook point.

REFERENCES

[1] Halfond, W. G., & Orso, A. (2005). AMNESIA: Analysis and Monitoring for Neutralizing SQL-Injection Attacks. Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering, 174–183.
[2] Bisht, P., Madhusudan, P., & Venkatakrishnan, V. N. (2010). CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks. ACM Transactions on Information and System Security, 13(2), 1–39.
[3] Gould, C., Su, Z., & Devanbu, P. (2004). JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications. Proc. of the 26th International Conference on Software Engineering, 697–698.
[4] Sun, H., Du, Y., & Li, Q. (2023). Deep Learning-Based Detection Technology for SQL Injection. Applied Sciences, 13(16), 9466.
[5] Lo, R., Hwang, W., & Tai, T. (2023). SQL Injection Detection Based on Lightweight Multi-Head Self-Attention. Applied Sciences, 15(2), 571.
[6] Zulu, J., Han, B., Alsmadi, I., & Liang, G. (2024). Enhancing Machine Learning Based SQL Injection Detection Using Contextualized Word Embedding. ACMSE 2024 Conference, 211–216.
[7] Adebiyi, M. O., et al. (2021). An SQL injection detection model using chi-square with classification techniques. Proc. of 2021 International Conference on Electrical, Computer and Energy Technologies (ICECET), 1–8.
[8] McWhirter, P. R., et al. (2018). SQL Injection Attack Classification through Feature Extraction of SQL Query Strings using a Gap-Weighted Subsequence Kernel. Journal of Information Security and Applications, 40, 199–216.