A Linguistic-based Method for SQL Injection Attack Detection and Defense[#]

Moira Geiger and Zhengping Jay Luo[*]

Department of Computer Science and Physics, Rider University, Lawrenceville, New Jersey, USA

Abstract: As the prominence of artificial intelligence (AI) has grown, companies are finding more ways to incorporate it into previously existing technological infrastructures to optimize their performance. Structured Query Language (SQL) is a popular way to store and retrieve data, and recently has been combined with AI search algorithms to speed up these processes. However, these successes make SQL vulnerable to attacks where malware is injected to gain access, modify, or even delete restricted data, which is also known as SQL Injection Attacks (SQLIA). This paper presents a detection approach that based on language processors, and shifts the focus of defenses against SQLIA from static pattern matching to recognizing the underlying linguistic and structural features of SQL injection attacks. Unlike traditional sanitization, this newly proposed defense will combine tokenization and vectorization with a language processor to detect malicious patterns and flag them. After running our proposed defense on a large public dataset, we received a precision score of 0.789, a recall score of 0.97, and an F1 score of 0.87 in terms of detection performance, which demonstrates the effectiveness of our defense method.

Keywords: Language processor, machine learning, sanitization, structured query language, SQL injection attacks, tokenization, vectorization.

1. INTRODUCTION

Relational databases are widely used across industries due to their scalability and ability to support complex backend systems [1]. They rely on Structured Query Language (SQL) to efficiently manage, query, and update stored information. SQL databases have become foundational to modern data management due to their robustness and performance [2]. However, this central role also makes them prime targets for cyber attacks, threatening the confidentiality, integrity, and availability of sensitive data. These attacks now intersect with broader cybersecurity and legal frameworks, as SQL-based systems often store regulated data and therefore require defensible, auditable detection mechanisms that satisfy both technical and compliance standards. Among these, SQL injection attacks (SQLIAs) remain one of the most prevalent and dangerous, allowing adversaries to insert malicious code into user inputs in order to manipulate or extract database content [3].

SQLIAs are consistently ranked among the top five most severe software vulnerabilities by OWASP [4]. Their threats are amplified by the wide adoption of machine learning (ML) techniques, as many machine learning-driven systems either store critical data in SQL databases or utilize SQL to optimize their functionality. ML can enhance database performance through predictive query optimization, automatic indexing, and workload management [5]. However, its integration without proper processors introduces new vulnerabilities by expanding attack surfaces and enabling adversaries to automate large-scale SQL injection attempts [6]. This combination highlights the need for defenses that improve detection accuracy while also providing forensic traceability and legal defensibility. Stakeholders increasingly require transparent explanations for how and why automated systems flag particular queries [6]. Introducing the language processor domain to a more matching-based algorithm can protect against these attacks.

Traditional mitigation techniques rely heavily on input sanitization, which attempts to filter out known malicious query patterns [7]. While useful, these approaches are weak: many SQL injection and cross-site scripting (XSS) attacks bypass sanitization by slightly modifying known payloads. Furthermore, sanitization mechanisms that depend on regular expressions are themselves susceptible to regex-based denial-of-service (ReDoS) attacks, which can exhaust system resources [7]. Preexisting algorithms that use static methods often require more manual intervention and are not aligned with real-time data needs [5].

Some other research endeavors have incorporated advanced practices into their SQL defense. However, the usage of tools, such as decision trees, can be

*Address correspondence to this author at Department of Computer Science and Physics, Rider University, Lawrenceville, New Jersey, USA;
Email: geigermo@rider.edu

[#]This work was sponsored by MacMillan Fellowship for Scientific Research Scholarship from Rider University.

vulnerable to the actual attack language if it is looking to elevate privilege [6]. Others use machine learning to compare distances between run-time and developer-intended queries to estimate if something is benign or not [1, 8]. Although utilizing ML for distance measuring can create a large overhead that a language processor would not [1]. Some train ML algorithms to work with web application firewalls, but the attacks that run against them by mutating an input to a semantically equivalent input, are in a different form [9]. Yet most ML-based approaches provide limited insight into which specific linguistic or structural features triggered a detection, reducing their usefulness in digital forensic reconstruction after an incident.

In this paper, we present a linguistic approach that shifts the focus from static pattern matching to recognizing the underlying linguistic and structural features of SQL injection attempts. Our research improves the preexisting input sanitization technology with the power of language processing capabilities. Finally, we integrate these components into a prototype attack detector powered by language processing and MySQL. The performance of this system is evaluated through accuracy metrics, offering insights into the viability of defenses against SQL injection that extend beyond preexisting matching-based methods.

Our new defensive system would most efficiently defend against tautology attacks, errors, and Boolean-based attacks, and special character attacks. Since the foundation of our defense is focused on detecting human language versus attack language, it defends input with out of the ordinary characters the best. It was designed this way since a majority of SQL injection attacks are successful by consistently running characters against the database until they find a combination of attack characters not sanitized by the preexisting defense. The implementation of character n-grams and not just static pattern-matching, simultaneously addresses differences due to spelling variation, morphology, and word choice [10]. Because the model uses TF-IDF character n-grams, it inherits the known vulnerability of linear text classifiers to adversarial token-level mutations. Although out-of-band attacks that work through network protocols, like DNS or HTTP, are important, the previously mentioned attack categories are much more common and can be ran by even amateur hackers. By targeting the types of attacks that make up the majority of SQLIAs, our defense found a higher success rate. To situate our contribution within real-world attack conditions, we develop a threat model that outlines the attacker's capabilities, system assumptions, and plausible bypass strategies. This model guides both the design and evaluation of our defense.

The main contributions of the paper are outlined as follows:

- We implemented a malicious SQL injection attack detector based on linguistics.
- We incorporated vectorization, string tokenization, a heuristic fallback, and an adaptive threshold to improve our defense from the levels of a traditional sanitization.
- We estimated a 97.82% decrease in approximate vulnerable websites after our defensive system was ran against their payloads, compared to the estimated percentage of vulnerable websites [3].
- We received a precision score of 0.789, a recall score of 0.97, and an F1 score of 0.87 for the successful detection of payloads of benign and malicious texts on a public dataset.

2 LITERATURE REVIEW

2.1. SQL Injection Attacks Background

Specifically, SQL injection attacks have been a growing threat to relational databases because of the large reward of vulnerable information that hackers gain as a result of it. Not only do SQLIAs endanger protected data, but they can also compromise the underlying operating system on which the SQL server runs. Certain attacks that look to escalate privilege in order to access data can also use the newfound privilege to overrun and execute commands on the host operating system [6]. Such attacks can damage organizational reputations and cost thousands to millions of dollars in recovery and remediation efforts. The distrust that could stem from the public regarding SQLIAs is an important consequence. While many studies describe their severity, fewer address how modern AI-driven systems increase the attack surface by introducing new automated vectors that scale these attacks. As SQL systems become embedded within machine-learning enhanced workflows, the challenges of detecting subtle, mutated payloads grow more complex, especially in environments that require interpretable and verifiable security responses.

In theory, the problem of SQLIAs could be resolved if every datagram were opened and analyzed within a network. However, most networks process such a large

quantity of data that this is close to impossible, leaving the chance of missed detection very high [4]. A common issue with many of the defense systems against injection attacks is the high rate of both false positives and false negatives.

2.2. SQLIA Attack Strategies

2.2.1. Attacks through Firewalls and Browsers

One common strategy is to exploit HTTP/HTTPS pathways to bypass an application's firewall, especially when the system lacks mechanisms that can detect structural changes in mutated or encoded inputs [6]. Another operates in a similar manner but looks to target the user side. By targeting direct user input or manipulating cookies stored in a browser, attackers can often access data more easily when client-side protections are weak [1].

2.2.2. Attacks Using Language

Attackers frequently use tautology-based injections (e.g., appending "OR 1=1") to force a query to always evaluate as true, sometimes pairing these with UNION statements to redirect where data is pulled from [6]. These techniques highlight how attackers rely on recognizable linguistic and structural patterns, yet many current ML systems fail to explain which parts of the input made it suspicious, limiting forensic traceability after an incident. Moreover, those can be joined with union query statements that say to retrieve something 'FROM' another location to retrieve from a separate query [11].

2.2.3. Attacks Using Errors and Output

Other types of attacks include error-based and Boolean-based that look to gain knowledge of the structure of the query, instead of the data it contains, by receiving either errors or 'true' Booleans back on code related to movement [11]. In addition, piggy-back queried attacks and stored procedure injections use malicious code written by the hacker to modify or retrieve data from within the query. In piggybacked queries, attackers append malicious code (often separated by semicolons) to an existing query. In stored procedure injections, they replace legitimate procedures with maliciously crafted versions [11]. Such semantically equivalent but syntactically modified payloads expose weaknesses in models that rely solely on statistical similarity rather than structural linguistic features.

2.3. SQLIA Defensive Strategies

2.3.1. Defense Using Traffic Inspection

The defensive side of these injection attacks looks to catch the attack attempt beforehand rather than fix it when it is already inside the SQL database. One method is similar to the ideal situation mentioned in Section 2.1 where deep packet inspection methods of flow monitoring and behavioral traffic analysis are used to detect abnormal patterns or sudden spikes in datagram communication [12]. This strategy is also seen used in conjunction with machine learning algorithms that can be trained to process large quantities of network data that would be difficult to process solely within the SQL.

2.3.2. Defense Comparing Previous Algorithms

Hybrid approaches combine machine learning with SQL-specific features, such as tree-based representations or keyword mining, to reduce false positives. While these approaches can improve detection accuracy, they inherit two major limitations: they are highly sensitive to adversarially mutated payloads, and they generally lack the ability to be verified, making it difficult for analysts to determine which part of the input contributed to a malicious classification [13]. More advanced machine learning methods similar to those have been developed to compare data inside the query.

2.3.3 Defense Using Query Comparisons

More dynamic systems, such as the CANDID model, compare developer-intended queries against user-supplied ones to detect inconsistencies [1]. Although these systems offer an improvement over static filters, they struggle with layered input structures and do not provide fine-grained explanations about why the queries differ. As a result, they lack the forensic traceability necessary for understanding or reconstructing an attack. Moreover, this approach struggles with layered input structures, such as loops, which are difficult to analyze dynamically. Following the idea of monitoring the user side, some SQL's use an ontology approach that cross checks the semantics of the user side to see how well it matches with true human languages versus malicious code [1]. These systems also tend to be brittle against adversarial input mutations, where attackers slightly alter syntax or encoding to evade ML classifiers.

2.4. Machine Learning Techniques used in SQLIA

The machine learning aspect has become more ingrained on both the defensive and attacking sides of SQLs. Recent attack developments have found that automated testing techniques are able to bypass web application firewalls when they match up with the previously stored payloads in a database [6]. This proves the threat of the "insider", someone who has access to the inner workings of a software, who can train a machine learning model from the inside to create holes in input boxes for the outside attackers. However, most ML-based defenses rely on opaque models where the decision-making process is not transparent, limiting their usefulness in security settings that require explainability and ability to be verified. Furthermore, adversarial mutation tools can generate payloads that bypass these models by altering syntax while preserving malicious intent. On the defensive side, decision trees and long short term memory (LSTM) are currently very popular [14]. However, the lack of ability to interpret in deep ML models makes them unsuitable for security environments that require transparent and verifiable detection logic. The recurrent neural networks that LSTMs operate as a part of, however, can not adapt as quickly as attack language is adapting- leaving further vulnerabilities.

In terms of the rise of SQLIAs, machine learning is being used as an augmentation tool to preexisting algorithms, such as data processing in deep data mining mentioned previously. In certain cases, it was found that the addition of AI into traditional algorithms had a yield of up to a 40% reduction in query execution time and a 25% increase in throughput for indexing strategies that change dynamically with the query [11]. Machine learning-enhanced algorithms not only process more data in less time but also adapt more effectively to evolving user inputs.

Language processing techniques offer an opportunity to address the shortcomings of existing ML approaches by focusing on structural and behavioral characteristics of input rather than surface-level tokens alone [15]. Because NLP-based detection can highlight which n-grams or linguistic patterns contributed to a classification, it naturally improves explainability and forensic traceability. This ability to interpret is one of the motivations behind our proposed approach, which seeks not only to detect attacks but to identify the specific linguistic patterns that make them malicious. Furthermore, its in-house style of processing eliminates the need for outside processes, which henceforth eliminates the need for a whole new set of resources, time allocation for tune-ups, and vulnerability patches. Its dynamic functionality enables real-time analysis while reducing data movement and associated latency [2].

Beyond technical vulnerability, SQL injection and XSS attacks also raise legal and forensic challenges. Effective detection systems must support auditability and maintain a clear chain of custody for logged events, since these logs may later be used as evidence in breach investigations. Studies in digital forensics emphasize that ML-based detectors must produce interpretable outputs to ensure legal admissibility and to verify how a detection decision was made [16]. These requirements highlight the need for approaches that prioritize transparency and traceable linguistic patterns rather than opaque model predictions.

Our proposed method that incorporates language processing is necessary since it does not just sanitize based off of individual characters. Traditional sanitization defensive methods target each character at a time, which can ignore important attack patterns. Unlike the traditional, our defense trains the detector to recognize patterns, not just characters. It is these advanced patterns that are causing the SQLIA rates to be so high, so having our defense target these types of attacks will cause it to be the most efficient.

3. DEFENDING AGAINST SQLIA BASED ON LINGUISTIC AND STRUCTURAL FEATURES

3.1. Threat Model

To clarify the scope of this work, we define the threat model guiding our evaluation and system design. We assume an adversary with the ability to submit arbitrary input to any publicly accessible field on a web application, including login pages, guestbook forms, and general text-based inputs. The attacker is capable of crafting both known and mutated SQL injection payloads, including tautology-based injections, UNION-based data extraction, Boolean and error-based enumeration, and time-delay payloads. We also assume that the attacker can obfuscate these payloads through encoding, comment insertion, spacing manipulation, or character substitution to bypass naive string-matching defenses. We acknowledge that a fully adaptive adversary may attempt gradient-free or reinforcement-based evasion strategies targeting the model’s n-gram representations.

The application environment is assumed to operate under typical web-stack conditions: an Apache2 server, a relational backend database, and PHP-based form handling, all of which process user-supplied input prior to sanitization. We assume no direct access to server configuration files, file systems, or administrator credentials. Additionally, we assume that network-level protections may be in place but cannot be solely relied upon to prevent application-level SQL injection attempts [6].

Our model also considers potential bypass strategies. These include adversarially modified payloads designed to evade ML-based detectors through syntactic variation; hybrid payloads combining SQL and HTML/JavaScript fragments; and low-and-slow attack strategies where an adversary distributes attempts to avoid pattern-based thresholds [17]. By explicitly modeling these capabilities, our approach evaluates not only whether the system detects known SQL threats, but also whether it generalizes to mutated, encoded, and behaviorally atypical payloads.

This threat model informs the design of our defense by emphasizing explainability, linguistic structure detection, and forensic traceability-features that support not only technical detection but also incident reconstruction and accountability requirements in security and regulatory contexts.

3.2. Motivation

While our implementation describes the practical components of the detector, the core contribution of this work lies in the conceptual shift from character-based sanitization to linguistic pattern modeling. By focusing on structural and semantic properties of attack language-rather than on individual tokens our approach reframes SQLIA defense as a language classification task. This conceptual reframing is critical because it enables explainability, facilitates forensic traceability of malicious patterns, and offers greater adversarial robustness compared to purely syntactic or statistical models.

As one can see, there are a variety of approaches when performing a SQLIA, along with a matched variety of defenses. Many of the preexisting ones, however, are designed off of static algorithms that are typically centered around individual human language characters or query decision trees. Some defenses do use the hybrid approach of combining the machine learning aspect to spot certain attack phrases. This type of work has inspired the type of defense we look to improve on through this research. Given the rapid growth of AI capabilities, enhancing existing defensive strategies with AI offers one of the most promising solutions to SQLIA vulnerabilities.

In order to maximize the existing defensive strategies, our proposed idea to counter SQL injection attacks would be to train automated intelligence to detect attack language, errors, and common attack techniques. As previously mentioned in the other sections, there are a large variety of ways to conduct a SQL injection attack. In order to narrow the focus of this AI research, we looked at the most recent large-scaled attacks to determine which methods were the best to focus on. Cross-scripted sites, better known as XSS attacks, rely on any input form within a website to carry out their attacks and were the method behind the Resume Looters attacks on over 65 websites in 2023 that gave access to over 2 million users' information [18]. The cross-site scripting attacks are especially dangerous because of the ubiquity of search bars, suggestion boxes, and comment threads on modern websites.

3.3. Language Processing Incorporated Defense

3.3.1. Overall Logic

The innovation of out proposed method specifically comes from the language processing approach in comparison to traditional sanitization defenses. As most of those sanitizers only examine singular characters, our language processing aspect focuses on identifying specific patterns post-vectorization. This will produce higher accuracy rates as SQLIAs are becoming more advanced then the brute-force methods that were originally most popular.

The idea of our defense is to first create a training file to teach the language processor to treat numerical vectors and regular expression strings. This would prepare the language processor while we also look to create connections through a server to the database. By running benign and malicious text files through a runner file, we could test our new language processor. Through the server, it would send the prediction values and store them in the database. After the testing, we would label each payload with its either benign or malicious origins so that the accuracy rates, true positives, true negatives, false positives, and false negatives could be calculated. The flow chart in Figure **1** depicts this process.

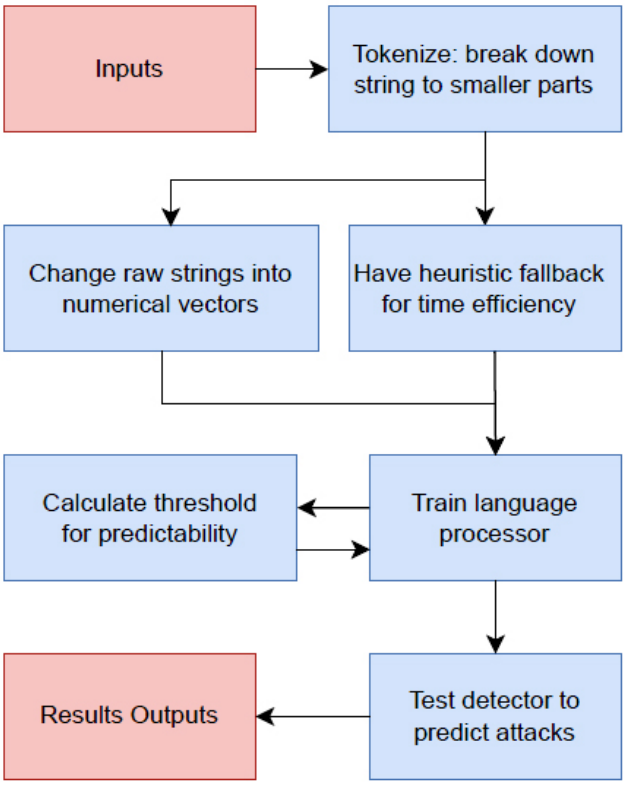

Figure 1: The workflow of defensive detection pipeline showing the transformation of raw input into tokenized, numerical vectors, which then go through threshold calculations. Each stage reflects how the model extracts linguistic cues associated with malicious intent, enabling automated identification of SQL injection attempts.

3.3.2. SQL Map and Payloads

To establish a baseline understanding of SQL injection payload behavior, we examined preexisting educational tools and intentionally vulnerable web applications commonly used in cybersecurity research. Resources such as Pacheco's SQL injection lab and the Damn Vulnerable Web Application (DVWA) allowed us to observe how automated tools and manual payloads interact with poorly secured database environments [19] (see Appendix A for implementation details). These environments provided controlled conditions in which we could analyze common attack patterns, successful payload structures, and the conditions under which injections bypass or fail against implemented safeguards. This exploratory phase informed our later design choices by clarifying which payload characteristics are consistently exploitable and which defensive mechanisms are most easily circumvented.

We also used DVWA to study reflected and stored cross-site scripting behaviors, comparing how different security levels affected detection success and failure. By executing repeated payload loops through a Python-driven interface, we evaluated the reliability and repeatability of common attack vectors. These insights guided the refinement of our detector by highlighting which linguistic and structural patterns most reliably indicate malicious intent.

3.3.3. Building a Test Webpage for Baseline Defense

After experimenting with existing testing tools, we constructed a deliberately vulnerable "guestbook" webpage to better observe how SQL injection payloads behave in a controlled environment. The page consisted of a basic name-and-message input structure connected to a small MySQL backend. We then executed a curated set of payloads against the page using a Python script, allowing us to empirically confirm which attack types reliably bypassed its minimal protections. These results established a baseline vulnerability profile to inform the design of our defense.

With this profile in place, we reversed our perspective and began hardening the page against the same categories of attacks. Initial modifications focused on server-side sanitization to mitigate common XSS-related behaviors, such as the use of special characters. A simplified example is shown in Figure **2**, which demonstrates how escaping user-supplied content prevents script injection. Additional logging functionality was added to record all incoming payloads within a MySQL database, enabling later analysis and supporting forensic traceability.

```
echo "<p><strong>" . htmlspecialchars($row['
    name'], ENT_QUOTES, 'UTF-8') .
     ":</strong> " . htmlspecialchars($row['
    message'], ENT_QUOTES, 'UTF-8') . "</p>";
```

Figure 2: Illustrates the first layer of defense in the SQL injection workflow by preventing executable script tags or encoded payloads from being rendered, an example shown with htmlspecialchars

Code snippets like 2 help catch simple attacks in order for us to develop a more advanced defense. The PHP code was also modified to add a logging function that recorded every payload into a pre-configured MySQL database (more implementation details in Appendix B).

To extend the defense beyond basic sanitization, we incorporated machine-learning techniques capable of distinguishing natural language from malicious input

patterns. Representative high-risk phrases and structural features from common SQLIAs such as comment injections, time delays, traversal sequences, external resource calls, and encoded characters were seeded into the model's training set to support pattern acquisition. The system then applied a supervised learning pipeline using TF-IDF character n-grams and logistic regression, coupled with threshold optimization based on F1 and precision scores [17]. This approach allowed the model to learn generalizable linguistic signatures of SQL injection attempts rather than relying on fixed string matching.

Our adaptive threshold was selected to balance precision and recall, but it would benefit from further empirical grounding. Future work should include studies comparing static, percentile-based, and dynamic thresholds to quantify how each impacts false-positive rates and classifier stability. This would provide a more formal justification for the chosen thresholding strategy and improve the model's reliability in operational settings.

3.3.4. Python Coding

When running the Python files, we chose to do so in a virtual environment through the Linux command lines in order to protect the files on my device. From there, we were able to open an Uvicorn server for the files and database to work in conjunction with one another. In a separate terminal, we were able to test the code by running "curl" functions that fit under the "benign" or "attack" category in which a prediction of 0 or 1, respectively, would be outputted. For example, a payload of an alert script attack returned a "1" prediction, representing an attack.

Although the system is implemented using several Python modules working together, the underlying idea is straightforward: each stage of the pipeline transforms raw input into increasingly meaningful linguistic representations. Training builds the model's understanding of benign versus malicious structure, serving exposes the model to real-time queries, and evaluation measures how reliably these learned patterns generalize. The conceptual modularity is more central than the specific coding sequence, because it shows how linguistic intelligence can be layered onto traditional SQL defenses.

From there, we created a handful of Python files that when all have been ran, produce the accuracy results of our tests. The "train_detector.py" file created took in both the benign and malicious files and used that data and its length to calculate a threshold of probability to use in determining whether it was 0 (benign) or 1 (malicious). It also included other precautions that help filter the code through regex language or char n-grams, which will be discussed in the results portion. Next, we have a "serve_detector.py" file that configures the database and connects it to the MySQL application. It grabs the adaptive threshold from the "train_detector.py" file and also starts the FastAPI application within the Uvicorn server. From there, its make_prediction() function uses a vectorizer, and a heuristic fallback, to calculate a prediction probability and ultimately, a prediction score of 0 or 1. Finally, it logs the prediction and score to the database. Another important file includes "labels.py" that after configuration, logs the original category of "benign" or "malicious" to the database so that the "calculate.py" file can pull the original assignment in comparison to the predicted assignment to produce the accuracy rate, true positive, false positive, true negative, and false negative values.

4. EXPERIMENTAL RESULTS AND ANALYSIS

The experimental logic of our system followed a modular pipeline consisting of data preprocessing, model training, and prediction validation. The raw payloads from both benign and malicious sources were first cleaned and normalized before being passed into the language-based detection model. Each payload was evaluated using adaptive thresholds to dynamically determine classification probabilities. This logical flow ensured that each phase of processing could be tested and validated independently before integration. By designing this in a modular development cycle, it ensured that each different part could be worked on independently of the progress of other parts. Towards the end of the development cycle, all parts did not to be synchronized to create stable connections in the sequence flow, but we were able to make changes as necessary to each parts individually in the process.

The experiments were conducted on an Ubuntu Linux-based operating system, with all the code being written in Python 3.13 through the Visual Studio Code software. In terms of other tools, we used MySQL as the Linux-based database, FastAPI as the framework, and Uvicorn as the Python-based server. The main dependencies included NumPy, Pandas, Joblib, and Scikit-learn for data processing and model training. The dataset consisted of 39,967 payloads evenly distributed between benign and malicious samples.

For model configuration, we tested multiple preprocessing strategies including tokenization, character n-grams, and vectorization to improve input representation. The adaptive threshold mechanism was initialized to update based on the ratio of benign to malicious samples per batch. All outputs were logged and visualized to track improvements in accuracy, true positives, true negatives, false positives, and false negatives over iterations

4.1. Results Analysis

Evaluation samples were generated to simulate the attacker capabilities described in our threat model, including obfuscation techniques and syntactic mutation strategies. Within our generated files, all text was lowercased, Unicode-normalized, stripped of null bytes, and tokenized using a reproducible, deterministic preprocessing pipeline implemented with version-pinned Python libraries [20]. This preprocessing step ensured that every sample could be regenerated and re-evaluated in subsequent experiments. For training, we applied TF-IDF character n-gram extraction followed by logistic regression with fixed random seeds for deterministic results. Model evaluation was conducted using 5-fold cross-validation, allowing us to report metrics (precision, recall, F1-score) rather than relying on a single experimental run. These methodological controls were incorporated to support reproducibility, reduce dataset bias, and provide robust evidence for the model's performance.

In the initial trial run, the model achieved an accuracy rate of only 1.65%, with 0 true positives, 304 true negatives, 304 false positives, and 0 false negatives. Manual curl tests revealed that the main issue stemmed from data transfer and conversion. We investigated the most efficient methods in modern machine learning and found that our missing component was vectorization, which is turning raw textual data into numerical vectors that the detector can more easily process [21]. Implementing vectorization reduced the number of null values from unprocessable payloads. We also imported NumPy, a Python numerical library that supports high-level mathematical computing, to further optimize processing [22]. This original test run and the following results from the implementation details in this section are depicted in Figure **3**.

Rather than relying solely on surface-level token comparisons, our system models SQL input through multiple layers of linguistic representation. Vectorization captures higher-order statistical characteristics, tokenization maps human-readable structure, and the heuristic fallback provides interpretable, rule-based support. Together, these layers illustrate how linguistic structure-not simply string matching defines malicious intent [23].

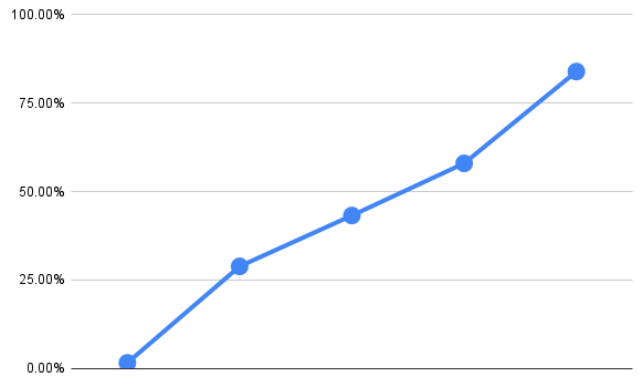

Figure 3: The accuracy rate percentage that resulted from running each step outlined in the implementation details, graphed to show its development.

After this change, we received an accuracy rate of 43.22%, 0 true positives, 10034 true negatives, 0 false positives, and 13180 false negatives. This revealed that during tokenization debugging, the threshold had shifted, causing the model to classify all inputs as benign (score of 0). The adaptive threshold plays a conceptual role by enabling the detector to calibrate itself to the underlying distribution of benign versus malicious patterns. Instead of using a static boundary, the model learns how uncertainty manifests within its linguistic feature space, which is essential in environments where attack patterns evolve unpredictably [24]. We also included baseline code for char n-grams, that help in language processing models detect certain language patterns [10]. We also observed that roughly 33% of payloads were excluded from the final counts because they were labeled as null values. To fix this, we added a heuristic fallback to catch certain phrases or character patterns that signify a benign or malicious payload [17]. By implementing this regex method, we were able to decrease the amount of unprocessable payloads and increase the total count.

With the addition of an adaptive threshold and regex code portions, our accuracy rate increased to 57.94%, 13150 true positives, 300 true negatives, 9734 false positives, and 30 false negatives. Further debugging identified several type and attribute errors that

prevented proper integration between components of the detector. We also reviewed the text files themselves to ensure that each payload was the category it was actually supposed to be. This debugging was crucial in aligning all the correct database and function names to each part of the detector files. Although the classifier performs well on unmutated test payloads, its performance may degrade under adversarial mutation, a common challenge for n-gram-based NLP models.

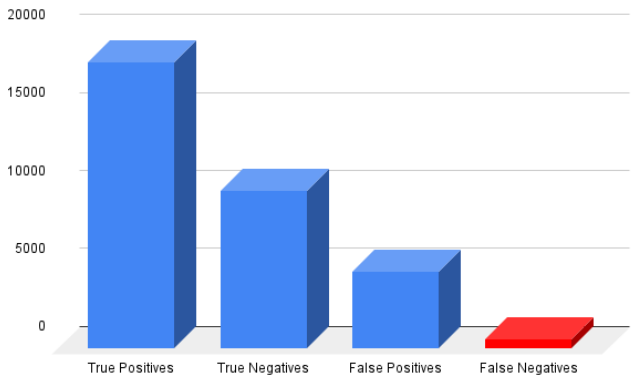

Figure 4: Final model performance metrics summarizing the classifier's ability to correctly distinguish benign and malicious queries. These results provide insight into detection reliability and highlight areas where further feature refinement or ensemble approaches could improve operational performance.

Our final produced values were an accuracy rate of 83.85%, 18349 true positives, 10132 true negatives, 4919 false positives, and 567 false negatives. This elevated false-positive rate suggests that, although the model captures malicious structure, it may also be oversensitive to benign queries that share surface-level linguistic features. Although our system did not achieve 100% accuracy, we observed significant progress with each retraining phase. Moreover, we found that 67% of web applications are vulnerable to SQL injection attacks [3]. As we refer to the "vulnerable websites", this accounts for the estimated 67% accounted for via the source. When adding our total number of payloads to get 39967 of them, we can divide the amount of false negatives by that sum to see how many malicious attacks are not caught and are flagged as a negative (0, benign). This is displayed in Figure **4**.

To most effectively and objectively display our results, we chose to report the precision, recall, and F1 score. These metrics are commonly used in the machine learning academic community to measure categorization models efficiency. They are especially important in our dataset since they are designed to cater towards imbalanced data sets, so we are actively fighting against any possible bias by choosing these metrics [25].

For our measure of precision, we calculated the true positives divided by the sum of the true positives and false positives to get a final value of .789 (shown in equation 1). The accuracy rate for all implementation steps and in Table **1** are defined by the total number of correct predictions divided by the total number of attempts.

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{18349}{18349 + 4919} = 0.789 \quad (1)$$

For our measure of recall, we calculated the true positives divided by the sum of the true positives and true negatives to get a final value of .97 (shown in equation 2).

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{18349}{18349 + 567} = 0.97 \quad (2)$$

For our measure of F1, we calculated the product of 2 and precision and recall divided by the sum of precision and recall to get a final value of .87 (shown in equation 3).

$$F1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 \times 0.789 \times 0.97}{0.789 + 0.97} = 0.87 \quad (3)$$

Overall, our precision metric of 0.789 may have been relatively smaller in comparison to the recall metric, but in the general concept of machine learning, it holds above a 0.75 success threshold. The recall value was very high, with a score of 0.97. Moreover, the F1 score that balances both the other metrics held at 0.87, another relative success.

4.2. Comparison with Existing Defenses

In order to prove the effectiveness of our defensive system further, we ran another experiment as a comparison. Although there are limited direct input sanitization Linux code available, we implemented a baseline sanitizer [7]. This sanitizer was designed to only match characters with malicious examples and not utilize the language processing capabilities of our

defense to demonstrate its novelty and importance. It also excludes the tokenization and vectorization features needed in our algorithms, solely taking in raw text. The comparison in this study is intentionally limited to a baseline sanitizer due to the restricted availability of more advanced defensive tools such as WAF-based or hybrid ML systems [9]. This constraint reflects the research context rather than the theoretical scope, and expanding the comparison set remains an important direction for future work.

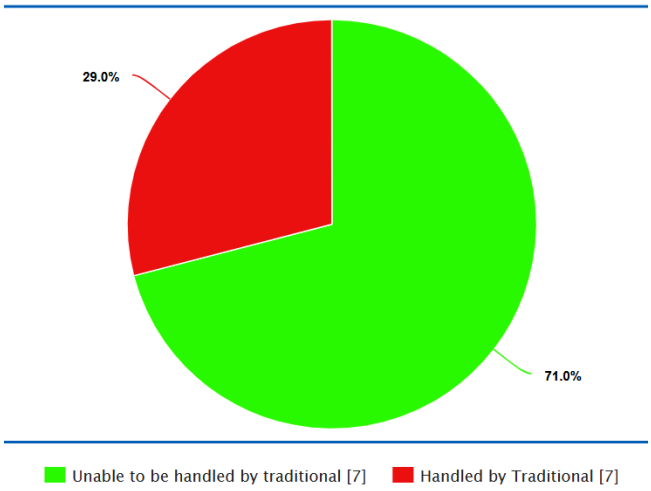

Figure 5: Payloads traditional defense [7] successfully handled out of the total payloads successfully handled by our defense.

Due to this, only 29% of the original benign and malicious payloads were even able to be processed by the baseline sanitizer (11607 of the 39967). This shows that even before executing the program, there is an extreme deficit to what our defense can process versus the traditional defense, as reflected in Figure **5**.

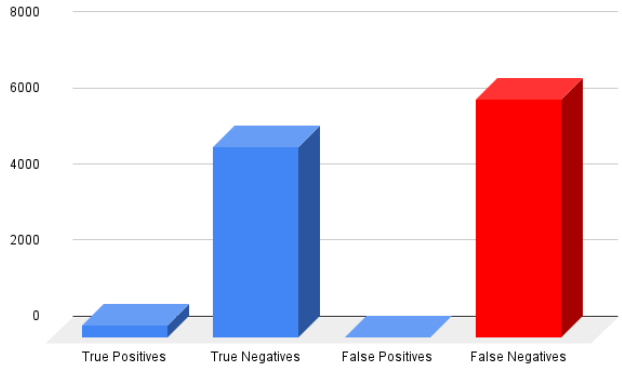

Figure 6: Final values of true positives, true negatives, false positives, and false negatives from the traditional baseline sanitizer [7].

As a result of running the baseline sanitizer, we received an accuracy rate of 46.06%, 333 true positives, 5013 true negatives, 4 false positives, and 6257 false negatives displayed in Figure **6**. From this experiment, we can deduce that there is an imbalance in being able to process benign versus malicious payloads in the traditional sanitizer. Moreover, the false negative rate was relatively high compared to the other results. We also represented this data with the precision, recall, and F1 metrics from our defense's results for direct numerical comparison [25].

For our measure of precision, we calculated the true positives divided by the sum of the true positives and false positives to get a final value of .988 (shown in equation 4).

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{333}{333 + 4} = 0.988 \tag{4}$$

For our measure of recall, we calculated the true positives divided by the sum of the true positives and true negatives to get a final value of .051 (shown in equation 5).

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{333}{333 + 6257} = 0.051 \tag{5}$$

For our measure of F1, we calculated the product of 2 and precision and recall divided by the sum of precision and recall to get a final value of .097 (shown in equation 6).

$$F1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 \times 0.988 \times 0.051}{0.988 + 0.051} = 0.097 \tag{6}$$

Although the traditional sanitizer had a higher precision rate, its capability to process payloads, recall metric, and F1 metric were all lower than our defensive system. When implementing our defense instead, the accuracy rate saw a 38% increase from 0.46 to 0.84 as shown in Table **1**. The recall metric increased significantly from the traditional defense's 0.051 to our 0.970, as shown in Table **2**. The F1 metric increased by significantly from the traditional defense's 0.097 to our 0.870, as shown in Table **2**. The precision rate of the traditional defense (0.988) was 20.04% better than our defense (0.789), which could be a result of the lesser payloads processed.

Table 1: The Comparison of the Accuracy Rate between our Method and the Existing Method [7]

	Our Proposed Method	Existing Method [7]
Accuracy	0.84	0.46

Table 2: The Comparison of our Proposed Method with Existing Method [7] in Terms of Precision, Recall and F1 score

	Our Proposed Method	Traditional Method [7]
Precision	0.789	0.988
Recall	0.970	0.051
F1	0.870	0.097

Although the model demonstrates strong recall, the precision score (0.789) indicates that a portion of benign traffic is still being misclassified as malicious. To improve precision in future iterations, we would utilize expanded feature engineering, ensemble methods that combine linguistic and statistical classifiers, and contextual models that incorporate surrounding query history or user-behavior patterns [26]. Such enhancements may produce a more discriminative boundary between legitimate user input and evolving attack payloads.

These results highlight not only the performance gains of our implementation but also the broader conceptual value of treating SQL injections as linguistic phenomena. Models grounded in character n-grams and structural cues can detect malicious intent that eludes traditional syntactic filters. More importantly, the approach provides interpretable signals identifiable linguistic patterns that benefit incident response, digital forensics, and policy-driven audit requirements.

5. DISCUSSION AND LIMITATIONS

Although our tests used manually created vulnerable webpages and controlled payload loops, these experiments served primarily as a conceptual sandbox rather than a full replication of production systems. As such, external validity must be considered. Real-world SQL injection attempts involve higher query variability, more complex layers, multiple database engines, and attacker strategies that evolve over time [6]. While our linguistic defense generally worked well across the payload classes we tested, further evaluation in high-volume environments is necessary to assess robustness against novel or unseen attack patterns. Future work will incorporate logs from enterprise-scale web applications, real-time monitoring pipelines, and simulations to validate how the detection model performs under operational load and adaptive pressure.

Being that the research was completed on one single laptop, its computing capabilities were lesser in comparison to a company's research that can utilize multiple devices for increased payloads and larger calculations. One shortcoming of our study is the limited dataset size used to test our processing defense. Although over 37,000 payloads may seem extensive, the model's performance could likely improve with a larger and more diverse dataset. A greater variety of SQL injection patterns would enable more robust training and finer tuning of the detection thresholds.

Additionally, while our results demonstrated high true positive rates and low false negative rates, the ratio of true negatives to false positives could be improved. This indicates that many benign queries were mistakenly classified as malicious. Although false positives are not inherently damaging, they can increase operational overhead by requiring manual verification and potentially disrupting normal system functionality. A further limitation of our approach is its vulnerability to adversarial machine-learning attacks specifically crafted to exploit weaknesses in linguistic or character-level n-gram models. Because the classifier relies heavily on character-sequence statistics, an attacker could generate adversarial payloads that preserve the semantic meaning of an injection while subtly altering its character distribution to evade detection. These risks underscore the need for future work incorporating adversarial training, ensemble representations, or hybrid symbolic constraints to ensure robustness against deliberately evasive injection patterns.

6. CONCLUSION

This paper discusses the importance of ingraining language processing techniques into pre-existing SQL defenses for preventing SQL injection attacks. Through our extensive research and practice, we were able to train the detector to generate predictions based on its learned algorithms. It was through example codes of benign and malicious code that we were able to train the detector to make predictions based off of its algorithms. Those predictions then used dynamic thresholds to calculate whether their probability would

be classified as an attack or not, which was demonstrated through accuracy rates, showing it outperformed a one-dimensional sanitization defense. SQLIA defenses that provide transparent reasoning are increasingly important for meeting legal expectations around explainability in automated decision-making systems.

The model's tendency to over-flag benign input limits its immediate operational viability, particularly in environments where database availability and SOC efficiency are critical. Future work should explore threshold calibration, dynamic confidence scoring, and hybrid architectures that pair interpretable linguistic signatures with secondary verification layers. Reducing false-positives while maintaining an ability to interpret and robustness will be crucial for transitioning this research into a deployable enterprise-scale defense.

Obtaining the final accuracy rate of 83.85%, a precision score of 0.789, a recall score of 0.97, and the F1 score of 0.87 compared to the metrics of the traditional input sanitizer shown in our comparison experiments proved the effectiveness of our defense. Not only does integrating the language processor, tokenizer, and vectorizer into sanitization practices increase the success metrics, but it also increases its payload processing capabilities. The model's ability to produce traceable linguistic indicators of malicious input offers value for digital forensic workflows by helping investigators reconstruct attack sequences. In future work, we would look to train our defense with even more complex payloads and expand its algorithmic practice. We could also look to gain access to other pre-existing defense software to further compare the strengths and weaknesses of each.

APPENDIX

Experimental Testbed and Preliminary Attack Exploration

A controlled SQL injection testbed was constructed using publicly available educational resources, including Pacheco's Creating a Vulnerable SQL Injection Lab for SQLMap Practice. The files were deployed within an isolated Linux virtual environment configured with Apache2 and MySQL to ensure operational safety and containment [20]. Initial configuration required minor adjustments to PHP page dependencies and environment permissions, after which SQLMap scans reliably enumerated available databases and demonstrated common injection vectors. These exploratory exercises provided foundational insight into typical payload structures and attack patterns frequently exploited by automated tools.

Development of Custom Vulnerable Pages and Baseline Defense Prototype

To complement the use of established testing environments, we constructed a minimal custom "guestbook" application designed to model a typical input-database workflow targeted by SQL and XSS attacks. The interface consisted of a guest name field and message field, backed by a MySQL database created within the Linux environment. This simplified configuration allowed us to observe how unprotected input is stored, rendered, and subsequently exploited.

To evaluate the application's baseline vulnerability, we executed a Python-based payload harness that iterated through a structured list of common SQLIA and XSS strings. The script recorded which inputs were successfully executed or injected into the page, providing a clear profile of the weaknesses present in the unsanitized version of the system. These results guided the subsequent design of defensive features.

A preliminary defensive layer was introduced by integrating server-side sanitization into the PHP rendering logic. This included the use of "htmlspecialchars" 2 to neutralize characters frequently used in reflected and stored XSS attacks. Additional logging statements were added to capture raw user input in a dedicated MySQL table, enabling later comparison between attempted and mitigated payloads.

REFERENCES

[1] Jemal I, Cheikhrouhou O, Hamam H, Mahfoudhi A. Sql injection attack detection and prevention techniques using machine learning. International Journal of Applied Engineering Research 2020; 15(6): 569-580.

[2] Islam S. Future trends in sql databases and big data analytics: Impact of machine learning and artificial intelligence. Available at SSRN 5064781, 2024. https://doi.org/10.2139/ssrn.5064781

[3] Mulki R. Sql injection isn't dead. here's why. Jul 2025. [Online]. Available: https://medium.com/@rizqimulkisrc/sql-injectionisnt-dead-here-s-why-aa4b6657f5c3

[4] Crespo-Martinez IA, Campazas-Vega A, Guerrero-Higueras AM, Riego-DelCastillo V, Ivarez-Aparicio CA, Fernandez-Llamas C. Sql injection attack detection in network flow data. Computers & Security 2023; 127: 103093. https://doi.org/10.1016/j.cose.2023.103093

[5] Gadde H. Integrating ai into sql query processing: Challenges and opportunities. International Journal of Advanced Engineering Technologies and Innovations 2022; 1(3): 194-219.

[6] Alghawazi M, Alghazzawi D, Alarifi S. Detection of sql injection attack using machine learning techniques: a systematic literature review. Journal of Cybersecurity and Privacy 2022; 2(4): 764-777. https://doi.org/10.3390/jcp2040039

[7] Barlas E, Du X, Davis JC. Exploiting input sanitization for regex denial of service. in Proceedings of the 44th International Conference on Software Engineering 2022; pp. 883-895. https://doi.org/10.1145/3510003.3510047

[8] Das D, Sharma U, Bhattacharyya D. Defeating sql injection attack in authentication security: an experimental study. International Journal of Information Security 2019; 18(1): 1-22. https://doi.org/10.1007/s10207-017-0393-x

[9] Appelt D, Nguyen CD, Briand LC, Alshahwan N. Automated testing for sql injection vulnerabilities: an input mutation approach. in Proceedings of the 2014 International Symposium on Software Testing and Analysis 2014; pp. 259-269. https://doi.org/10.1145/2610384.2610403

[10] Wieting J, Bansal M, Gimpel K, Livescu K. Charagram: Embedding words and sentences via character n-grams. arXiv preprint arXiv:1607.02789, 2016. https://doi.org/10.18653/v1/D16-1157

[11] Khan JR, Farooqui SA, Siddiqui AA. A survey on sql injection attacks types & their prevention techniques. Journal of Independent Studies and Research Computing 2023; 21(2): 1-4. https://doi.org/10.31645/JISRC.23.21.2.1

[12] Alotaibi FM, Vassilakis VG. Toward an sdn-based web application firewall: Defending against sql injection attacks. Future Internet 2023; 15(5): 170. https://doi.org/10.3390/fi15050170

[13] Sheng J. Research on sql injection attack and defense technology of power dispatching data network: Based on data mining. Mobile Information Systems 2022; 2022(1): 6207275. https://doi.org/10.1155/2022/6207275

[14] Muhammad T, Ghafory H. Sql injection attack detection using machine learning algorithm. Mesopotamian Journal of Cybersecurity 2022; 2022: 5-17. https://doi.org/10.58496/MJCS/2022/002

[15] Habib U. A survey on implication of artificial intelligence in detecting sql injections.

[16] Alorainy W. Ml-psdfa: A machine learning framework for synthetic log pattern synthesis in digital forensics. Electronics 2025; 14(19): 3947. https://doi.org/10.3390/electronics14193947

[17] Kapoor M, Fuchs G, Quance J. Rexactor: Automatic regular expression signature generation for stateless packet inspection. in 2021 IEEE 20th International Symposium on Network Computing and Applications (NCA). IEEE 2021; pp. 1-9. https://doi.org/10.1109/NCA53618.2021.9685959

[18] Yeboah PN, Kayes A, Rahayu W, Pardede E, Mahbub S. A framework for phishing and web attack detection using ensemble features of self-supervised pre-trained models Authorea Preprints 2025. https://doi.org/10.36227/techrxiv.173603362.21995515/v1

[19] Priyanka AK, Smruthi SS. Webapplication vulnerabilities: Exploitation and prevention. in 2020 Second International Conference on Inventive Research in Computing Applications (ICIRCA). IEEE 2020; pp. 729-734. https://doi.org/10.1109/ICIRCA48905.2020.9182928

[20] Creating a vulnerable sql injection lab for sqlmap practice. [Online]. Available: https://www.linkedin.com/pulse/creating-vulnerablesql-injection-lab-sqlmap-practice-jose-pacheco-ej3nc/

[21] Cui ED. Vectorization: A Practical Guide to Efficient Implementations of Machine Learning Algorithms. John Wiley & Sons, 2024. https://doi.org/10.1002/9781394272976

[22] Gupta P, Bagchi A. Introduction to numpy. in Essentials of Python for Artificial Intelligence and Machine Learning. Springer 2024; pp. 127-159. https://doi.org/10.1007/978-3-031-43725-0_4

[23] Choo S, Kim W. A study on the evaluation of tokenizer performance in natural language processing. Applied Artificial Intelligence 2023; 37(1): 2175112. https://doi.org/10.1080/08839514.2023.2175112

[24] Thatikonda M, PK MK, Amsaad F. A novel dynamic confidence threshold estimation ai algorithm for enhanced object detection. in NAECON 2024-IEEE National Aerospace and Electronics Conference. IEEE 2024; pp. 359-363. https://doi.org/10.1109/NAECON61878.2024.10670627

[25] Yacouby R, Axman D. Probabilistic extension of precision, recall, and f1 score for more thorough evaluation of classification models. in Proceedings of the first workshop on evaluation and comparison of NLP Systems 2020; pp. 79-91. https://doi.org/10.18653/v1/2020.eval4nlp-1.9

[26] Zhou B. Optimized feature engineering for machine learning-based financial trend prediction. Available at SSRN 5734370.

Received on 22-10-2025 Accepted on 20-11-2025 Published on 04-12-2025

https://doi.org/10.65879/3070-5789.2025.01.05