

# **VULNERABILITY DETECTION AND PREVENTION: AN APPROACH TO ENHANCE CYBERSECURITY**

---

**A Thesis Submitted in Partial Fulfillment of the Requirements for the  
Award of Degree of**

**MS**

**In**

**Computer Science**

**BY**

**AMINA ZAHID**

**REGISTRATION #**

**Department of Computer Science**



---

**UNIVERSITY OF GUJRAT**

**Session 2022-24**

## **ACKNOWLEDGEMENT**

I feel fortunate to get continuous guidance, support, and encouragement from my advisor Dr. Umar Shoaib. His experience in Artificial Intelligence and Machine Learning helped me to develop and implement my project and overcome the technical challenges that I faced during implementation. I would like to thank him for his constant guidance throughout the two semesters that helped me in the research and implementation related to my project. I would also like to thank my committee members for taking time to review my project.

**Amina Zahid**

## **DEDICATION**

This thesis is dedicated to my teachers, family and friends, whose unwavering support and encouragement have been a constant source of strength throughout this journey. To my mentors and professors, thank you for your invaluable guidance and knowledge, which have been instrumental in shaping this work. To the cybersecurity community, whose relentless efforts in safeguarding our digital world inspire me to contribute towards a safer cyberspace. Lastly, to all the individuals and organizations striving to enhance the security of our information systems, this work is a tribute to your dedication and commitment.

**Amina Zahid**

## **DECLARATION**

I, Amina Zahid D/O Zahid, roll# 22015919-013, MS Computer Science, Department of Computer Science, Faculty of Computing and Information Technology, University of Gujrat, Pakistan, hereby solemnly declare that this thesis titled “Vulnerability Detection And Prevention: An Approach To Enhance Cybersecurity” is based on genuine work and has not yet been submitted or published elsewhere. Furthermore, I’ll not use this thesis for obtaining any degree from this University or any other institution.

I also understand that if evidence of plagiarism is found in this thesis at any stage, even after the award of degree, the degree may be cancelled and revoked by the University.

**Amina Zahid**

It’s certified that Amina Zahid D/O Zahid, roll# 22015919-013, MS Computer Science, Department of Computer Science, Faculty of Computing and Information Technology, University of Gujrat, Pakistan worked under my supervision and above stated declaration is true to the best of my knowledge.

---

Dr. Umar Shoaib

Assistant Professor, Department of Computer Science

University of Gujrat, Punjab, Pakistan

Email: umar.shoaib@uog.edu.pk

Dated: \_\_\_\_\_

## THESIS COMPLETION CERTIFICATE

It's certified that this thesis titled "Vulnerability Detection And Prevention: An Approach To Enhance Cybersecurity" submitted by Ms. Amina Zahid D/O Zahid, roll# 22015919-013, MS Computer Science, Department of Computer Science, Faculty of Computing and Information Technology, University of Gujrat, Pakistan is evaluated and accepted for the award of degree "MS Computer Science" by the following members of the Thesis Viva Voce Examination Committee.

The evaluation report is available in the Directorate of Advanced Studies and Research Board of the University.

---

External Examiner Name

Position

Department

Email: [dummy@gmail.com](mailto:dummy@gmail.com)

---

Dr. Umar Shoaib

Assistant Professor, Department of Computer Science

University of Gujrat, Punjab, Pakistan.

Email: [umar.shoaib@uog.edu.pk](mailto:umar.shoaib@uog.edu.pk)

---

Dr. Fiaz Majeed

Assistant Professor/ HOD,

University of Gujrat, Punjab, Pakistan.

Email: [fiaz.majeed@uog.edu.pk](mailto:fiaz.majeed@uog.edu.pk)

Dated: \_\_\_\_\_

## **CERTIFICATE OF PLAGIARISM**

It is certified that MS Thesis Titled as “Vulnerability Detection And Prevention: An Approach To Enhance Cybersecurity” submitted by Ms. Amina Zahid to us. We undertake the follows:

- a. Thesis has significant new work/knowledge as compared already published or are under consideration to be published elsewhere. No sentence, equation, diagram, table, paragraph, or section has been copied verbatim from previous work unless it is placed under quotation marks and duly referenced.
- b. The work presented is original and own work of the author (i.e. there is no plagiarism). No ideas, processes, results, or words of others have been presented as Author own work.
- c. There is no fabrication of data or results which have been compiled/analyzed.
- d. There is no falsification by manipulating research materials, equipment, or processes, or changing or omitting data or results such that the research is not accurately represented in the research record.
- e. The thesis has been checked using TURNITIN (copy of originality report attached) and found within limits as per HEC plagiarism Policy and instructions issued from time to time.
- f. While generating the Turnitin report, nothing has been excluded from Abstract to Conclusion parts of the thesis.

---

Amina Zahid

22015919-013

Department of Computer Science

University of Gujrat, Punjab, Pakistan.

---

Dr. Umar Shoaib

Assistant Professor, Department of Computer Science

University of Gujrat, Punjab, Pakistan.

Email: umar.shoaib@uog.edu.pk

---

## TABLE OF CONTENTS

---

CONTENTS	PAGE
LIST OF FIGURES .....	viii
LIST OF TABLES .....	ix
LIST OF APPENDICES .....	xi
ABSTRACT.....	1
Chapter 01: INTRODUCTION.....	2
Chapter 02: LITERATURE REVIEW .....	8
Chapter 03: MATERIALS AND METHODS .....	21
3.1 Dataset Creation.....	22
3.2 Feature Extraction .....	26
3.3 Data Preprocessing.....	29
3.4 Tokenization.....	30
3.5 Statistical Feature Extraction .....	33
3.6 Model Training and Testing.....	35
3.6.1 Gradient Boosting (GB) .....	37
3.6.2 Random Forest (RF).....	39
3.6.3 K-Nearest Neighbor (KNN).....	40
3.6.4 Multi Layer Perceptron (MLP) .....	41
3.7 Preventive Strategies.....	43
Chapter 04: RESULTS DISCUSSIONS AND COMPARATIVE ANALYSIS .....	49
Chapter 05: CONCLUSION AND FUTURE WORK.....	97
REFERENCES .....	98
APPENDICES .....	101

---

---

## LIST OF FIGURES

---

CONTENTS	PAGE
<b>Figure 1:</b> Example Login Page.....	4
<b>Figure 2:</b> Workflow for SQLI Vulnerability Detection .....	21
<b>Figure 3:</b> Basic Structure of MLP .....	41
<b>Figure 4:</b> Classifiers Performance for SQLI Vulnerability Detection .....	50
<b>Figure 5:</b> Regular Expression for Tokenization .....	60

---



---

## LIST OF TABLES

---

CONTENTSSS	PAGE
<b>Table 1:</b> Literature Review Summary .....	14
<b>Table 2:</b> Original Form of Dataset .....	24
<b>Table 3:</b> A few rows from SQLI type detection dataset.....	24
<b>Table 4:</b> A few rows from SQLI type vulnerability identification dataset.....	25
<b>Table 5:</b> Feature Set .....	28
<b>Table 6:</b> Cleaning Process.....	30
<b>Table 7:</b> Tokenization Process .....	31
<b>Table 8:</b> Calculating Statistical Features .....	34
<b>Table 9:</b> Hyper-parameter tuning .....	36
<b>Table 10:</b> Results for GB algorithm .....	38
<b>Table 11:</b> Results for RF algorithm.....	39
<b>Table 12:</b> Results for KNN algorithm .....	40
<b>Table 13:</b> Results for MLP algorithm .....	42
<b>Table 14:</b> Implementing Parametrized Queries.....	44
<b>Table 15:</b> Implementing Input Validation.....	44
<b>Table 16:</b> Escaping Special Character.....	45
<b>Table 17:</b> Applying Least Privilege Principle .....	46
<b>Table 18:</b> Results for SQLI Vulnerability detection .....	49
<b>Table 19:</b> Comparative Analysis Dataset.....	57
<b>Table 20:</b> Grouping Method.....	60
<b>Table 21:</b> Tokenization .....	61
<b>Table 22:</b> Showing structure of new Dataset .....	61
<b>Table 23:</b> Converting Token_Count to Float .....	64
<b>Table 24:</b> Values of Observed and Expected G-test score .....	66
<b>Table 25:</b> Values of Observed, Expected G-test score and Entropy .....	68

---

---

<b>Table 26:</b> A preview of a few rows from training dataset.....	71
<b>Table 27:</b> Results for GB.....	73
<b>Table 28:</b> Overview of Queries before and after Cleaning .....	75
<b>Table 29:</b> A look on tokenization process.....	77
<b>Table 30:</b> Calculating G-test score and Entropy .....	81
<b>Table 31:</b> Showing Results for GB .....	93
<b>Table 32:</b> Comparison table of both techniques.....	93

---

---

## LIST OF APPENDICES

---

CONTENTS	PAGE
<b>APPENDIX-01:</b> Abbreviations used in thesis .....	101
<b>APPENDIX-02:</b> Turnitin Originality Report.....	102

---

## **ABSTRACT**


In websites, SQL injection (SQLI) is considered to be among the most prevalent and dangerous vulnerabilities. It allows attackers to manipulate and execute malicious queries, potentially compromising the logical portion of the database. Such attacks can lead to data breaches, unauthorized data access, and data corruption, posing significant risks to the security and integrity of web applications. Previous cybersecurity research has faced numerous challenges due to the lack of specialized tools and technologies tailored for SQLI detection and prevention. This gap has hindered the development and implementation of effective security measures. Therefore, there is a critical need for more advanced and targeted development in this area to enhance the detection and prevention of SQLI vulnerabilities. To address this gap, a comprehensive self-made SQLI dataset containing 12566 records has been created. This dataset includes a diverse range of queries, both vulnerable and non-vulnerable, to provide a robust foundation for assessing various machine learning (ML) and deep learning (DL) algorithms. The primary objective is to identify and detect SQLI vulnerability with high accuracy and efficiency. A variety of ML and DL algorithms are explored: including Random Forest (RF), Gradient Boosting (GB), K-Nearest Neighbor (KNN), and Multi-Layer Perceptron (MLP). Each algorithm is evaluated on its ability to classify SQL queries as either vulnerable or non-vulnerable and to identify the specific type of SQLI attack (union-based, time-based, boolean-based etc.). Accuracy serves as the primary evaluation metric used to assess the performance of each algorithm. According to the results, GB achieved the highest accuracy at 97.91%, followed by K-Nearest Neighbors with 96.00%. The MLP classifier also performed well, with an accuracy of 94.19%, while the RF classifier achieved 93.61%. In addition to SQLI vulnerability detection, this thesis also presents several preventive strategies to mitigate SQLI vulnerabilities. These strategies include input validation, use of parametrized queries, escaping special characters, least privilege principle, web application firewall and continuous security testing. Future efforts will focus on exploring advanced ML and DL techniques to further improve the detection and prevention of SQLI and other types of vulnerabilities. This includes developing more advanced models, improving accuracy, and broadening the scope of vulnerability detection to cover a wider range of security threats.

### **INTRODUCTION**


SQLI is a significant security vulnerability that has been the focus of extensive research within the cybersecurity domain. This vulnerability targets the database layer of web applications and is a favored vector for attackers because it results in severe data breaches and system compromises. The vulnerability highlights the critical importance of proper input sanitization by applications. When an application fails to correctly handle input data in an SQL query, it allows attackers to inject malicious SQL code, resulting in unauthorized database access and manipulation. The exponential growth of internet usage worldwide has reached unprecedented levels, with approximately 5.3 billion people, equivalent to 65.7% of the global population, actively engaging online as of October 2023 (Statista, 2023). This surge in internet users has been accompanied by a corresponding increase in cyber threats, notably SQLI attacks. These attacks exploit vulnerabilities in the way web applications process user input, allowing cybercriminals to manipulate databases and potentially compromise sensitive data. The impact of SQLI attacks is profound, with statistics from the List of Data Breaches and Cyber Attacks in 2023 revealing a concerning trend. In October 2023 alone, 114 security incidents were publicly disclosed, resulting in the compromise of over 867 million records. This staggering figure contributed to an annual total exceeding 5 billion compromised records, highlighting the severity and frequency of cyber attacks targeting web applications (List of Data Breaches and Cyber Attacks, 2023). SQLI attacks are a persistent threat to online applications, exploiting vulnerabilities in web application input mechanisms to compromise data security and operational integrity. These attacks, as highlighted by (Deepa et al. 2018), jeopardize essential security services such as data integrity, authorization, confidentiality, and authentication. Recent statistics (source) emphasize the significant impact of SQLI attacks on cybersecurity, with a substantial number of incidents recorded globally within a specific time frame. These attacks not only result in financial losses but also damage the reputation of affected organizations, eroding trust in their services. Furthermore, SQLI attacks compromise data integrity by allowing unauthorized manipulation of databases through injected SQL commands, potentially disrupting business operations. Additionally, they undermine authorization mechanisms by granting attackers unauthorized access to restricted areas of web applications, posing a threat to data confidentiality. By exploiting vulnerabilities in SQL queries, attackers extract sensitive information, compromising user privacy and organizational security. In conclusion, effective mitigation strategies, such as robust input validation mechanisms and proactive security measures, are crucial for protecting against SQLI attacks and safeguarding sensitive information from unauthorized access and exploitation. Efficient detection of SQLI attacks is critical for safeguarding the sensitive data stored in databases. The prevalence of SQLI

vulnerability highlights the urgent need for robust detection systems to mitigate cybersecurity risks effectively. It's commonly found in web applications, pose a serious threat to data security by exposing sensitive information to unauthorized access. To address this challenge, organizations must prioritize the implementation of comprehensive detection mechanisms capable of identifying and thwarting SQLI attacks in real-time. By leveraging advanced detection technologies like anomaly detection algorithms and behavior analysis techniques, organizations strengthen their defenses against evolving cyber threats. Additionally, proactive monitoring and continuous assessment of web applications' security posture are crucial for promptly detecting and remedying SQLI vulnerabilities. Regular security audits, vulnerability assessments, and penetration testing enable organizations to identify and mitigate potential weaknesses, enhancing their overall cybersecurity resilience (List of Data Breaches and Cyber Attacks, 2023). SQLI attacks pose a significant threat to web application security, involving the insertion of malicious SQL commands into input forms or queries. These attacks exploit vulnerabilities in input validation mechanisms, enabling attackers to execute arbitrary SQL code and gain unauthorized access to databases (Fang et al, 2018). By manipulating input fields, attackers inject malicious SQL payloads, bypass authentication mechanisms, and compromise data integrity. Inadequately validated user input is often exploited in successful SQLI attacks, where applications fail to sanitize and validate input data properly before incorporating it into SQL queries. This oversight allows user-supplied input to be treated as executable SQL code, potentially leading to data leakage, unauthorized data modification, or complete database compromise. A fundamental example illustrates how attackers manipulate input fields to inject malicious SQL code, circumventing authentication mechanisms and gaining unauthorized access to sensitive information (Li et al, 2019). SQLI attacks are designed to obtain unauthorized access to databases by exploiting a SQL query and injecting code (Abirami et al, 2015). Let's understand this with a simple scenario. Let's say that students use their username and password to access a university website. After successfully authenticating using an authentic username and password, the student will be able to log in.


STUDENT LOGIN



Email ID



Password



LOGIN

**Figure.1.** Example Login Page (Taken From: STUDENT LOGIN FORM | Figma)

This is the query for the successful login attempt where:

Username = usr

Password = usr123

SQL Query: `SELECT * FROM users WHERE name = 'usr' and password = 'usr123'`

But it's also possible that someone with bad intentions types the following into the website's username and password fields:

Username = usr

Password = ' or '1' = '1

In this case, the SQL query that is created will be:

SQL Query: `SELECT * FROM users WHERE name = 'usr' and password = '' or '1' = '1'`

Since `1=1` is always true, this user's login to the website will be allowed indefinitely. The owner of the account may suffer serious consequences if they misuse the user's unauthorized access to another person's account details. It is a theft and a violation of data privacy.

This was a very basic example of a SQLI attack for understanding purposes. Because of the majority of contemporary websites and web apps, this kind of attack would be challenging to defend against. However, there are more complex forms of SQLI attacks, some of which are covered in this article. Attackers intend to exploit the database that is linked to a website or online application by means of SQLI. SQLI attacks must be prevented in order to safeguard the private information kept in these databases.

SQLI attacks manifest in various forms, exploiting specific vulnerabilities in web applications' database handling. From in-band to out-of-band SQLI, attackers employ diverse tactics to achieve their malicious objectives (Bockermann et al, 2009). Here are several types of SQLI as discussed below.

- Error-based
- Union based
- Time based
- Boolean based
- Comment based

- Tautology based

Each type represents a distinct method used by attackers to exploit vulnerabilities in SQLI queries. Error-based SQLI collects data about the composition and structure of databases by taking advantage of error messages that are produced by the database server. It entails inserting malicious SQL code into input fields of susceptible online applications to cause error messages that provide the attacker with useful information. A strong method for gathering data from a database, error-based SQLI allows attackers to obtain important information that helps them further exploit the weak application.

Union-based SQLI is a technique that combines the output of several SELECT queries into a single result set by tampering with the SQL query structure. Usually, this technique is employed to retrieve information from a database that an attacker isn't able to access. An attacker locates a weak point in a web application's input field, creates a malicious input with a SQLI payload, inserts the payload into the weak point, and unintentionally runs the attacker's SQL code. By adding more SELECT statements into the original query, the injected SQL code increases the possibility of sensitive data being revealed (Mishra, 2019).

Boolean-based blind SQLI is a kind of SQLI attack in which the attacker sends SQL queries to the application and determines whether or not the results satisfy particular conditions. This allows the attacker to gather knowledge about the database. Usually, the attacker in a boolean-based blind SQLI (Mishra, 2019) attack does not see any output from the database or receive any direct error signals. Rather, they depend on how the program behaves to ascertain if the SQL query they injected resulted in a true or false state. This is frequently accomplished by tracking variations in the application's response, including variations in HTTP status codes, page content, or response times.

Time-based is not dependent on how the injected SQL code appears to affect the answers from the web application. Rather, it depends on the attacker's capacity to determine details about the database by timing how long it takes the application to reply to specific queries. An attacker creates SQL queries that force the database to do lengthy processes in order to carry out a time-based blind SQLI attack. Through monitoring the application's response time to these queries, the attacker deduce the success or failure of the injected SQL code's execution (Mishra, 2019).

Comment-based SQLI attack involves the attacker manipulating parameters or input fields inside a web application in order to insert well constructed SQL code that includes comments. The backend database of the program behaves differently as a result of these comments. SQL code is annotated with comments to improve readability and aid with documentation. "--" for single-line comments and "/\* \*/" for multi-line comments are common comment syntaxes.



Nonetheless, comments are used to change the semantics of SQL queries in the context of SQLI attacks

Tautology-based SQLI is a type of attack where the attacker inserts code that takes advantage of the logical structure of SQL statements by using tautologies (expressions that are always true) to alter the query's behavior. This attack is commonly employed to bypass authentication mechanisms or gain unauthorized access to data. In this method, the attacker injects a condition that always evaluates to true, causing the query to return more results than intended.

Successful SQLI attacks pose significant risks to organizations, extending beyond immediate financial losses. Legal liabilities can arise from breaches of data protection regulations like General Data Protection Regulation (GDPR), leading to hefty fines and penalties. Moreover, SQLI attacks can cause severe reputational damage, diminishing trust among customers and stakeholders. The exposure of sensitive data, including financial records and personally identifiable information, heightens the risk of identity theft and fraud. Additionally, compromised integrity and availability of critical systems can disrupt business operations, causing operational downtime. Thus, robust preventive measures are crucial to mitigate the risks associated with SQLI attacks and safeguard both user privacy and organizational security (OWASP Top Ten, 2023). Moreover, preventive measures are essential for addressing SQLI attacks, which pose a significant threat to the security of web applications. These measures require a comprehensive approach to strengthen defenses against potential vulnerabilities. Robust input validation, prepared statements and other practices are vital for preventing unauthorized SQL queries, ensuring that user-provided data undergoes thorough sanitization and validation prior to processing. Employing secure coding techniques (such as parameterized queries and prepared statements etc.) are crucial for reducing the risk of SQLI vulnerabilities by separating SQL code from user input. Moreover, proactive awareness efforts among developers and end-users are pivotal in bolstering cybersecurity resilience against SQLI attacks. Educating and training stakeholders on secure coding practices and promoting safe browsing habits are effective strategies for mitigating the risks associated with SQLI attacks and preserving the integrity and confidentiality of data (Minhas et al, 2013). Here is its **problem statement** that forced us to work on SQLI vulnerability detection.

Despite progress in SQLI vulnerability detection, previous cybersecurity research has encountered significant challenges due to the absence of specialized tools and technologies designed specifically for SQLI detection and prevention. This gap has limited the effectiveness of security measures, highlighting the urgent need for advanced and targeted solutions to improve the detection and prevention of SQLI vulnerabilities.

Acquiring large and diverse datasets for training ML and DL models poses a significant challenge in cybersecurity due to the sensitive nature of security-related data and the evolving tactics of malicious actors. Additionally, the interpretability and explainability of ML and DL models are critical for decision-making in cybersecurity operations, emphasizing the importance of transparency in model outputs. To overcome these challenges, the research aims to enhance detection systems by leveraging heuristic methods based on practical rules and strategies derived from empirical observations and expert knowledge.

This thesis aims to develop effective strategies for detecting and mitigating SQLI attacks for self-made dataset, leveraging ML and DL algorithms. It seeks to enhance existing detection systems and explore preventive measures to comprehensively address SQLI vulnerabilities (Gupta et al, 2014). The following are the main **research objectives** of the thesis in question:

- Assessment of the efficacy of various algorithms: this involves analyzing the outcomes of various algorithms in order to determine the best architecture for vulnerability detection.
- The overall goals of the research conducted in this study are to increase reliability while also improving the accuracy, and reliability of models in vulnerability detection. Moreover, here are research questions to be answered.
  - Which algorithm performs best for detecting vulnerability?
  - What kind of vulnerability can be discovered?
  - How to prevent from vulnerability?
  - Which performance metrics are considered in this research?

The remaining sections of this thesis are organized as follows: Section two covers the literature review, while sections three and four present proposed methodology, experiments and the results along-with comparative analysis with the existing tool and technique, respectively. Finally, section five presents the conclusion and future work.

### **LITERATURE REVIEW**

Researchers in the field of network security have extensively explored the use of ML and DL techniques for the detection and prevention of SQLI vulnerabilities. Various methods have been proposed to enhance the accuracy of SQLI attack detection while reducing false alarms, leveraging DL frameworks, ML algorithms, and natural language processing models, as highlighted by (Minhas et al, 2013). Researchers have explored the effectiveness of ensemble ML algorithms such as Gradient Boosting Machine (GBM), Adaptive Boosting (AdaBoost), Extended Gradient Boosting Machine (XGBM), and Light Gradient Boosting Machine (LGBM) in this context. These algorithms are designed to combine the predictive power of multiple weak classifiers, thereby improving the overall detection capability for SQLI attacks. However, their implementation often requires additional computational processing and careful timing considerations to ensure real-time or near-real-time detection capabilities. This research aims to optimize these ensemble methods to achieve high accuracy in identifying and mitigating SQLI vulnerabilities, crucial for enhancing the security of web applications against cyber threats. (Lomio et al, 2022) explored just-in-time software vulnerability detection using ML techniques, focusing on empirical evidence and practical applications of commit-level vulnerability prediction models. The study employed the Goal-Question-Metric (GQM) paradigm to formulate research questions and guide the analysis. Various ML algorithms were utilized to analyze code, changes, and textual metrics extracted from software repositories. The researchers meticulously extracted features from both code and textual data to train the machine learning algorithms for vulnerability detection. Initial findings indicated that basic learners initially underperformed in classification tasks. However, the application of ensemble methods, such as boosting techniques like AdaBoost and GB, showed improvements in classification accuracy. Interestingly, the study also observed that adding additional metrics beyond a certain point did not significantly enhance the models' predictive performance. This comprehensive approach aimed to develop more effective methods for identifying vulnerabilities in software systems, particularly focusing on timely detection during the development lifecycle.

(Napier et al, 2023) conducted an evaluation of text-based machine learning models for software vulnerability detection, focusing on their effectiveness across a dataset encompassing 2,182 vulnerabilities observed in 344 software projects spanning 38 different types. The researchers began by implementing a comprehensive evaluation framework, which involved training and testing the machine learning models using text-based features derived from software repositories and vulnerability databases. The models were evaluated on their ability to distinguish between 'fixed' and 'vulnerable' functions within individual projects and across

different projects. The evaluation included the application of the Full Context data processing method alongside other approaches to compare their performance. Statistical analysis was employed to assess the significance of differences in prediction accuracy between within-project and cross-project scenarios. The study concluded that the text-based machine learning models struggled to effectively differentiate between fixed and vulnerable functions, exhibiting poor performance both within individual projects and when applied across different projects. (Luo et al, 2019) introduced a Convolutional Neural Network (CNN)-based approach for detecting SQLI vulnerabilities, which represented a significant advancement in cybersecurity. The methodology began with the formulation of research objectives aimed at improving accuracy, precision, and recall rates in SQLI detection tasks. The researchers implemented a CNN architecture designed to process and analyze SQL query structures effectively, leveraging its capability to capture intricate patterns within the data. Experimental evaluation involved training the CNN model on datasets containing a diverse range of SQLI attack scenarios and benign queries. Despite achieving promising results, the study acknowledged inherent limitations of CNNs in cybersecurity applications, particularly in the context of SQLI detection. Challenges included a lack of comprehensive discussion on CNN-specific limitations for SQLI detection, suggesting the need for deeper exploration and mitigation strategies. Dataset quality and diversity were identified as critical factors influencing model robustness and reliability. Issues such as dataset bias and inadequacy were addressed through meticulous curation and augmentation techniques to enhance the CNN's performance across varied environments and real-world scenarios. These efforts underscored the ongoing refinement necessary to optimize CNN-based approaches for effective and resilient SQLI detection in cybersecurity applications.

(Williams et al, 2018) proposed a novel data mining framework aimed at tracking the evolution of vulnerabilities in software products. The methodology began with the adoption of diffusion-based storytelling and the application of the Supervised Topical Evolution Model (STEM) to analyze extensive datasets efficiently. The approach enabled the researchers to uncover patterns and trends in the emergence and progression of vulnerabilities over time. The framework addressed scalability challenges often encountered in traditional methods by leveraging advanced data mining techniques. Through their analysis, the authors identified previously unnoticed correlations among vulnerabilities, significantly enriching the understanding of vulnerability dynamics within software ecosystems. However, the study highlighted the necessity for further investigation into the framework's generalizability and scalability across diverse software environments. Ongoing research is crucial to validate the framework's effectiveness and adaptability in varying contexts, ensuring its practical utility for comprehensive vulnerability tracking and management in real-world applications. (Nong et al, 2022) conducted a systematic review that centered on open science practices in the field of

software engineering, specifically focusing on DL-based approaches for vulnerability detection. The methodology involved a comprehensive search and selection process to identify 55 relevant studies from academic databases and research repositories. Each selected study was scrutinized to assess aspects of open science, including the accessibility of tools, feasibility of implementation, reproducibility of results, and replicability of experiments. The researchers analyzed the availability of publicly accessible tools associated with the DL-based approaches and examined the completeness of documentation and implementation details provided in each study. Their findings indicated that only 25.5% of the reviewed approaches offered publicly accessible tools, with several studies lacking thorough documentation and complete implementation, thereby limiting their practical application and reproducibility. This systematic review underscores the importance of enhancing open science practices in software engineering research to foster transparency, accessibility, and the reliability of deep learning-based tools for vulnerability detection.

(Roy et al, 2022) investigated the efficacy of ensemble ML algorithms, including RF, AdaBoost, Logistic Regression, and Naive Bayes, for detecting SQLI attacks in web-based systems. The methodology involved curating a dataset comprising both benign SQL queries and SQLI attack samples from diverse web systems, followed by feature engineering to extract syntactic, semantic, and statistical features from SQL statements. The dataset was split into training and testing sets using cross-validation techniques for robust model evaluation. Each ML algorithm was then trained on the training dataset and evaluated on the testing dataset using performance metrics such as accuracy, precision, recall, and F1-score. Naive Bayes emerged as the most effective method, demonstrating superior accuracy in detecting SQLI attacks compared to the other ensemble ML algorithms tested. This study underscores Naive Bayes' potential as a robust tool for enhancing cybersecurity in web applications vulnerable to SQLI threats. (Liu et al, 2020) introduced DeepSQLI, a novel tool in the domain of SQLI vulnerability detection that distinguished itself through the application of DL methodologies. The methodology began with the development of DeepSQLI's framework, which employed deep semantic learning techniques to generate specialized test cases aimed at identifying SQLI vulnerabilities. Unlike traditional tools such as SQLMap, which rely on predefined attack patterns and signatures, DeepSQLI focused on analyzing both the structural and semantic aspects of SQL queries to enhance detection accuracy. The researchers conducted rigorous experimentation to evaluate DeepSQLI's efficacy, employing comparative assessments against established tools like SQLMap. The study involved using diverse datasets containing benign queries and SQLI samples to train and test DeepSQLI. Performance metrics including detection accuracy, speed of operation, and efficiency in requiring fewer test cases were measured to assess DeepSQLI's effectiveness. Results demonstrated that DeepSQLI outperformed SQLMap by achieving

higher detection accuracy and efficiency in identifying SQLI vulnerabilities, underscoring its potential as an advanced tool for bolstering cybersecurity defenses against SQLI attacks in web applications.

In their paper, (Xu et al, 2020) introduced BinXray, a pioneering patch-based vulnerability matching technique aimed at detecting 1-day vulnerabilities in target programs. The methodology began with the development of BinXray's framework, which addressed the prevalent issue of high false positives encountered in existing vulnerability detection solutions. The key innovation of BinXray lay in its utilization of a trace similarity feature to determine whether a target program had undergone patching. This feature enabled BinXray to accurately distinguish between vulnerable and patched functions despite their similarity, thereby enhancing the efficacy of vulnerability detection. The researchers conducted extensive experiments to evaluate BinXray's performance, utilizing diverse datasets containing vulnerable and patched programs. They measured accuracy metrics, achieving a remarkable accuracy rate of 93.31%. Additionally, BinXray demonstrated minimal analysis time costs, averaging 296.17 milliseconds per analysis, highlighting its efficiency for real-time vulnerability detection tasks. The study underscored BinXray as a promising advancement in the field, offering a robust solution to improve the detection accuracy and efficiency of identifying 1-day vulnerabilities in software programs. (Shar et al, 2013) methodology proposed by identifying static code attributes based on observed input sanitization patterns commonly used in web applications to prevent SQLI and Cross-Site Scripting (XSS) vulnerabilities. These attributes were then utilized to construct vulnerability prediction models by analyzing historical information and known vulnerability data, focusing on predicting specific program statements vulnerable to SQLI and XSS. To facilitate data collection, a prototype tool called PhpMinerI was developed, allowing the evaluation of the models on eight open-source web applications. The research incorporated statistical inference methods, attribute ranking techniques, and performance measures to rigorously assess the models' predictive performance. By leveraging static analysis techniques to collect attribute vectors and using lightweight modeling methods, the paper aimed to provide a cost-effective alternative to dynamic analysis techniques for identifying vulnerabilities.

(Kasim et al, 2021) conducted an experiment using an ensemble classification approach to effectively detect the attack levels of SQLIs. The approach employed ensemble classification as a multi-classifier method, improving performance by combining results from multiple weak classifiers. The researchers trained an ensemble algorithm with 22 features extracted from queries containing malicious code to classify these queries as either clean or malicious. Additionally, they categorized malicious SQLIs into simple, unified, or lateral types to determine the severity of the cyber-attack, which facilitated more precise detection and response

strategies. By leveraging ensemble classification and feature extraction techniques, the methodology achieved high accuracy rates of over 98% in detecting SQLIs and 92% in classifying attack levels, demonstrating the effectiveness of the developed middleware application. (Ojagbule et al, 2018) conducted a comprehensive penetration testing study on three prominent content management systems (CMS): WordPress, Drupal, and Joomla. The researchers began by setting up a Linux environment equipped with a LAMP stack, which included the installation of MySQL to manage the databases. Each CMS was then installed sequentially using default configurations to closely mimic real-world conditions for vulnerability testing. The penetration testing process was executed using SQLMAP, a tool designed to identify and exploit potential SQLI vulnerabilities. Their methodology encompassed several critical steps: information gathering to collect relevant data about the systems, scanning with Nikto to detect potential security issues, actively exploiting identified vulnerabilities, and finally, implementing mitigation strategies to address and resolve the security threats uncovered during testing. This thorough approach ensured a detailed analysis of the vulnerabilities present in each CMS and provided a clear pathway for enhancing their security posture.

(Braz et al, 2021) conducted an online experiment involving 146 participants, strategically including 105 individuals with over three years of professional software development experience. The primary objective of the study was to investigate the detection of Improper Input Validation (IIV) vulnerabilities during code reviews, a critical aspect of software security. Participants engaged in reviewing code changes intentionally seeded with injected vulnerabilities and a corner case bug, all presented on a single page to facilitate comprehensive examination. To ensure the integrity of their experimental setup, the researchers meticulously prepared the code changes themselves, with oversight and validation provided by multiple contributors. The experiment implemented various treatments to test specific hypotheses, such as manipulating the visibility of attack scenarios and informing participants about existing vulnerabilities prior to review. These treatments were systematically varied to assess their impact on the detection rates of IIV vulnerabilities. By rigorously analyzing these conditions, researchers aimed to identify and quantify factors influencing the effectiveness of vulnerability detection in code review processes. Their findings offered valuable insights into optimizing software security practices by enhancing vulnerability detection and mitigation strategies through structured review methodologies and informed participant engagement, thereby contributing to overall software reliability and security. (Elder et al, 2021) conducted an in-depth analysis focusing on the intricate relationships among vulnerability detection techniques, types of vulnerabilities identified, their exploitability, and the efforts required for remediation within two meticulously selected projects where all known vulnerabilities had been successfully

addressed. The methodology involved comprehensive data collection, meticulously documenting the specific techniques employed for vulnerability detection, ranging from static code analysis to dynamic testing methods. They cataloged the types of vulnerabilities uncovered, such as SQLI and Cross-Site Scripting (XSS), and assessed their potential exploitability based on severity and impact metrics. Efforts expended to mitigate these vulnerabilities were meticulously recorded to gauge the practical implications and resource allocations necessary for effective remediation strategies. Employing a sophisticated analytical framework, researchers evaluated the comparative effectiveness of these detection methods across the studied projects, highlighting variations in detection rates and mitigation outcomes. Expanding their investigation to include Open Source Software (OSS), the study examined how different resource constraints influenced vulnerability detection and resolution across multiple OSS projects. Statistical analyses were employed to quantitatively measure the relationships between detection techniques, vulnerability types, exploitability factors, and mitigation efforts, ensuring robust evaluation and interpretation of findings. Moreover, (Elder et al, 2021) addressed critical limitations such as data availability and project-specific contexts, aiming to offer nuanced insights into enhancing vulnerability management practices both in real-world software development and within the broader OSS community. A detailed comparative analysis has been conducted to evaluate the effectiveness of various SQLI detection methods. For instance, (Li et al. 2021) compared shallow learning models, CNNs, Deep Belief Networks (DBNs), and bidirectional Recurrent Neural Networks (RNNs) for identifying vulnerabilities, finding that CNNs were particularly effective. Similarly, (Hwang et al. 2022) proposed a CNN-based vulnerability detection solution that exhibited good performance and detection speed compared to state-of-the-art techniques.

Different vulnerability detection models have significant limitations as shown below, including inadequate analysis of cybersecurity corpora, challenges from poor programming practices, complexities in feature engineering, and errors in vulnerability identification. These limitations hinder the effectiveness and generalizability of these models. Studies like (Williams et al, 2018) highlight a gap in the analysis of cybersecurity corpora, (Chen et al, 2021) discuss difficulties in identifying high-level vulnerabilities, (Han et al, 2017) highlight the complexity of feature engineering, and (Iannone et al, 2022) point out errors in identifying vulnerability-fixing patches and commits. Studies like SySeVR call for a deeper investigation into vulnerability syntax properties and the integration of machine learning with extensive testing and comparison analyses. (Marjanov et al, 2022) stress the lack of consensus on evaluating machine learning techniques for vulnerability identification, while (Zhongxin et al, 2023) highlight challenges in separating information about vulnerabilities from other data and inefficiencies in handling large



codebases. (Odeh et al, 2023) argues for further investigation into suggested vulnerability systems, emphasizing machine learning integration, extensive testing, and comparison analysis.

**Table. 1.** Literature Review Summary

Methods Used	Results	Limitations	References
<p>To find temporal themes, used diffusion-based storytelling and the supervised topical evolution model.</p> <p>The emphasis is on the evolving vulnerability of software products over time.</p>	<p>The framework addresses scalability, reveals new vulnerability correlations, and presents STEM-P, a parallel version that is faster than the standard STEM model.</p>	<ul style="list-style-type: none"> <li>- Lack of effort in analyzing cybersecurity corpora for vulnerability evolution studies.</li> <li>- Need for further research to assess scalability and generalizability of the integrated data mining framework.</li> <li>- Potential variation in efficacy between the STEM model and diffusion-based storytelling technique.</li> <li>- Focus primarily on software product vulnerabilities, potentially excluding other areas of cybersecurity research.</li> </ul>	<p>Williams, M. A., Dey, S., Barranco, R. C., Naim, S. M., Hossain, M. S., &amp; Akbar, M. (2018, December). Analyzing evolving trends of vulnerabilities in national vulnerability database. In 2018 IEEE International Conference on Big Data (Big Data) (pp. 3011-3020).</p>
<p>Utilizing advanced graph neural networks, Deep-learning-based embedding technique is employed for static</p>	<p>DeepWukong attains notable accuracy and F1 Score in identifying vulnerabilities, surpassing both</p>	<ul style="list-style-type: none"> <li>- Challenges in identifying high-level vulnerabilities due to poor programming practices.</li> </ul>	<p>Cheng, X., Wang, H., Hua, J., Xu, G., &amp; Sui, Y. (2021). Deepwukong: Statically detecting software vulnerabilities using deep graph neural</p>

---

vulnerability detection in C/C++ programs, compactly embedding code fragments.	conventional and deep learning methods in vulnerability detection.	- Difficulty in developing precise static analysis solutions for diverse vulnerabilities.	network. ACM Transactions on Software Engineering and Methodology (TOSEM), 30(3), 1-33.
--	--	---	---

The emphasis is on using static analysis techniques to detect software vulnerabilities.

Utilizing word embeddings and a shallow CNN for predicting vulnerability severity. SVM with an RBF kernel employed for text classification in baseline models.	The study successfully predicted software vulnerability severity using deep learning and diverse word embeddings, achieving consistent results across different severity levels.	- Complexity in feature engineering due to the diverse nature of vulnerabilities and their descriptions. - Challenge in determining intricate vulnerability metrics without expert knowledge.	Han, Z., Li, X., Xing, Z., Liu, H., & Feng, Z. (2017, September). Learning to predict severity of software vulnerability using only vulnerability description. In 2017 IEEE International conference on software maintenance and evolution (ICSME) (pp. 125-136).
--	--	--	---

It emphasizes the significance of applying patches to mitigate these vulnerabilities.

Open coding process for actions taken to fix vulnerabilities.	Vulnerabilities often require multiple commits to be introduced rather than just a single commit. Developers	- Errors in identifying vulnerability-fixing patches and vulnerability-contributing commits.	Iannone, E., Guadagni, R., Ferrucci, F., De Lucia, A., & Palomba, F. (2022). The secret life of software vulnerabilities: A large-scale empirical
---	--	--	---

---

Calculation of VCCs to introduce vulnerabilities and the duration of insertion windows.	with heavy workloads are responsible for most security flaws. Additionally, vulnerabilities tend to have high survivability rates within the source code.	- The SZZ algorithm can generate false positives in vulnerability identification.	study. IEEE Transactions on Software Engineering, 49(1), 44-63.
Shallow learning models, CNNs, DBNs, and bidirectional RNNs are employed.	Fifteen vulnerabilities were found, seven were unidentified and reported to suppliers, and eight were patched covertly. The methodology has proven to be helpful in identifying vulnerabilities that have not been disclosed to the National Vulnerability Database.	- Challenges in identifying security holes in C/C++ software source code. - Need for more comprehensive syntax properties for vulnerability detection.	Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., & Chen, Z. (2021). Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing, 19(4), 2244-2258.
The parameters used are syntax information, semantic information and vector representations			
Methods for identifying source code vulnerabilities using machine learning use pipelines that preserve the	The study highlights the challenges in detecting vulnerabilities in source code, including the need for a consensus	- Lack of consensus in evaluating machine learning techniques for vulnerability identification.	Marjanov, T., Pashchenko, I., & Massacci, F. (2022). Machine learning for source code vulnerability detection: What works and what isn't there yet. IEEE

organization of the code.	benchmark, better coverage of error types and languages, and pipelines for code organization.	- Restricted discussions on error types and programming languages.	Security & Privacy, 20(5), 60-76.
The parameters are source code representations, input vectorization, and combination of representations.			
Splitting up a syntax-based Control Flow Graph (CFG) into several possible ways to execute it	The suggested method performs better in terms of F1-Score, Precision, and Recall than the most recent baselines.	- Difficulty in separating information about vulnerabilities from other information.  - Limits on input length causing inefficiency when handling large codebases.	Zhang, J., Liu, Z., Hu, X., Xia, X., & Li, S. (2023). Vulnerability detection by learning from syntax-based execution paths of code. <i>IEEE Transactions on Software Engineering</i> , 49(8), 4196-4212.
Convolutional neural networks and pre-trained code models are used to learn path representations.			
The parameters are control flow graph (CFG), path representations and feature vectors etc.			
The system's main objectives are to identify and stop SQLI, remote code execution, cross-site scripting assaults, and backend technology fingerprinting.	Web application vulnerabilities are efficiently found and prevented by the suggested system. The study emphasizes the value of sophisticated	- Need for further investigation into the suggested vulnerability system.  - Integration of machine learning requires extensive testing.	Odeh, N., & Hijazi, S. Detecting and Preventing Common Web Application Vulnerabilities: A Comprehensive Approach.

	techniques for prevention and detection.	- Comparative analysis is necessary to validate detection and prevention capabilities.	
Deep learning approaches employing convolutional neural network (CNN) CodeNet-based vulnerability detection method.	When compared to cutting-edge techniques, the suggested CodeNet-based vulnerability detection method exhibits good detection performance and detection time. The outcomes of the experiments conducted under different kinds of vulnerabilities are shown.	<ul style="list-style-type: none"> <li>- Disregard of smart contract semantics and context in picture analysis.</li> <li>- Ignorance of context and semantics can lead to false detection alarms.</li> </ul>	Hwang, S. J., Choi, S. H., Shin, J., & Choi, Y. H. (2022). CodeNet: Code-targeted convolutional neural network architecture for smart contract vulnerability detection. IEEE Access, 10, 32595-32607.
Deep natural language processing, neural language models, and word sequence prediction were used in this study.	In detecting SQLI vulnerabilities, DeepSQLI performs better than SQLmap.	<ul style="list-style-type: none"> <li>- Evaluation limited to SQLI testing effectiveness using specific metrics and evaluation methods.</li> <li>- Repeating experiments 20 times for each System Under Test (SUT) may require substantial computational resources.</li> <li>- Mitigation of randomness may introduce bias or</li> </ul>	Liu, M., Li, K., & Chen, T. (2020, July). DeepSQLI: Deep semantic learning for testing SQLI. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 286-297).

		overlook variability in real-world conditions.	
It presents a CNN accomplished a Convolutional Neural Network (CNN) based SQLI detection model that efficiently handles SQLI attacks by extracting attack payloads from network flow.	remarkable accuracy rate.	<ul style="list-style-type: none"> <li>- Inadequate explanation of CNN's limitations, datasets, comparisons, and computational requirements.</li> <li>- Potential impact on the CNN-based SQLI detection model's performance and generalizability in real-world situations.</li> </ul>	Luo, A., Huang, W., & Fan, W. (2019, June). A CNN-based Approach to the Detection of SQLI Attacks. In 2019 IEEE/ACIS 18th International Conference on Computer and Information Science (ICIS) (pp. 320-324).
AdaBoost algorithm has been incorporated into a deep forest model to increase its adaptability for SQLI detection.	The suggested approach outperforms both DL and traditional ML techniques in its ability to detect sophisticated SQLI attacks.	<ul style="list-style-type: none"> <li>- Deterioration of initial traits in deep forests as the number of layers increases.</li> <li>- Deep learning methods may outperform recommended strategies, impacting effectiveness.</li> </ul>	Li, Q., Li, W., Wang, J., & Cheng, M. (2019). A SQLI detection method based on adaptive deep forest. IEEE Access, 7, 145385-145394.
It deals with ML classifiers to identify SQLI attacks in web-based systems, such as RF, AdaBoost, and Logistic Regression.	The ideal strategy is determined to be Naive Bayes, which has the highest accuracy.	<ul style="list-style-type: none"> <li>- Lack of comparisons with other methods or existing solutions.</li> <li>- Absence of real-world examples to validate the effectiveness.</li> <li>- Non-utilization of the Kaggle SQLI dataset for benchmarking.</li> </ul>	Roy, P., Kumar, R., & Rani, P. (2022, May). SQLI attack detection by machine learning classifier. In 2022 International Conference on Applied Artificial Intelligence and Computing (ICAAIC) (pp. 394-400).

---

- Unclear computational requirements for implementing the runtime method.

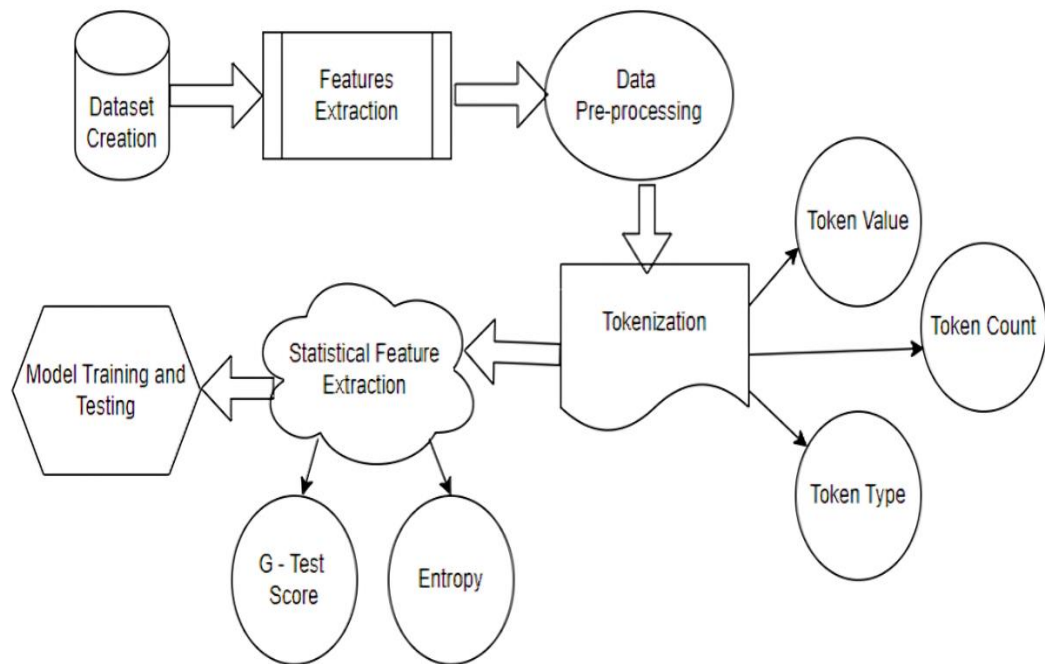
- Insufficient exploration of potential weaknesses in ML classifiers used for prevention.

---

This table provides a structured overview of various methods used in studies for detecting and addressing vulnerabilities, outlining techniques and algorithms employed, key findings, challenges, and references. The methods range from diffusion-based storytelling and graph neural networks to CNNs and machine learning classifiers. Results indicate improved scalability, accuracy, and effectiveness in detecting vulnerabilities. However, limitations such as challenges in feature engineering, high-level vulnerability identification, scalability, generalizability, and the need for further research are common.

**MATERIALS AND METHODS**

This section presents an entirely novel heuristic approach for identifying and detecting SQLI attacks. The objective is to use several DL and ML techniques on a self-made SQLI dataset with 12566 records. The steps involved in this method are shown in the following workflow diagram. The workflow comprises several key stages essential for effective SQLI vulnerability detection. It begins with Data Collection, where raw data is gathered from various sources and formats. Next, basic features are extracted from the queries, such as the number of boolean operators, unions, comments, SQL keywords, and other relevant features. The subsequent stage involves Data Cleansing, which rectifies errors, handles missing values, and normalizes the data for consistency. Following this, Tokenization breaks down the data into smaller tokens for individual analysis, considering token types, values, and counts. Then, statistical features like G-test scores and entropy are extracted. Finally, Model Training and Optimization utilize this processed data to train ML and DL models, focusing on selecting relevant features.



**Figure 2:** Workflow for SQLI Vulnerability Detection



In proposed framework as shown in above figure, the systematic approach ensures efficient data processing, modeling, and interpretation, providing a solid foundation for robust decision-making and understanding as discussed and shown above.

### **3.1. DATASET CREATION**

Dataset creation involves systematically gathering and organizing a set of data for SQLI vulnerability detection. We collected more than 12000 SQL queries whereas (Mishra, 2019) used 6000 SQLI queries for experiments. The decision to expand our dataset to over 12000 SQL queries was driven by the need to comprehensively capture the diversity and complexity of SQLI vulnerabilities across various domains and applications. Unlike (Mishra, 2019), which used 6000 SQLI queries, our expanded dataset aims to provide a broader spectrum of scenarios and edge cases, enhancing the robustness of our experiments and model training. This expansion allows us to explore a wider range of SQLI patterns, including less common or evolving attacks, which are crucial for building resilient detection and prevention systems. Moreover, with a larger dataset, we can improve the generalizability of our findings and ensure our models are better equipped to handle real-world challenges in cybersecurity. The impact of this dataset expansion extends to improved detection algorithms and enhanced security measures, ultimately contributing to more effective defenses against SQLI attacks in practical applications. Our dataset contains 534(4.25%) queries having issue of special characters presence, 106(0.84%) union based SQLI, 216(1.72%) error based SQLI, 120(0.95%) boolean based SQLI, 84(0.67%) tautology based SQLI, 120(0.95%) time based SQLI, 168(1.34%) comment based, 935(7.44%) are non-vulnerable and remaining queries encountering syntax error). Queries were extracted from personal projects (including: Healthcare Management System Development, Social Media Application Development, and E-commerce Platform Development etc.) as well as publicly available projects. Use of personal projects for SQLI query collection provides authentic and diverse examples of vulnerabilities encountered in real-world development environments enhancing the dataset's relevance and applicability, while on the other hand, publicly available datasets contributed examples and techniques from the cybersecurity community. We covered all the major types of SQLI queries for capturing complexities of SQLIs whereas, (Mishra, 2019) worked on just union-based, error-based, boolean-based and time-based SQLIs. Our hybrid dataset aims to capture more complexity and variability of SQLI vulnerabilities efficiently, ensuring its suitability for training ML and DL models. The different forms of SQLI queries (discussed in our thesis) are below.

- Error-based
- Union based
- Time based

- Boolean based
- Comment based
- Tautology based
- Presence of special characters
- Syntax error

Above list show different forms of SQLI queries present in our dataset, each targeting specific SQLI vulnerabilities. Error-based attacks exploit database error messages to gather information. Union-based attacks combine malicious queries with safe ones using the UNION operator. Time-based attacks use time delays to infer true or false conditions. Boolean-based attacks determine information by evaluating true or false statements. Comment-based attacks use SQL comments to manipulate queries, and Tautology-based attacks exploit always-true conditions in SQL statements.

After dataset creation, synchronization and validation of the dataset were performed. Synchronization involves aligning data from various sources to maintain coherence, while validation confirms the data's accuracy, completeness, and relevance. To achieve this, we employed a thorough validation process that combined both manual and automated methods. We manually reviewed samples to verify accurate labeling and cross-referenced entries with reputable sources and established SQLI patterns. Coverage checks were performed to ensure all predefined SQLI types were included, and automated scripts were used to identify and rectify missing or incomplete entries. Additionally, we evaluated the relevance of queries by examining their context and consulting experts to confirm that the dataset accurately reflects current SQLI attack patterns and techniques. This comprehensive approach ensures the dataset's robustness and its suitability for effective model training.

Tagging/ Labeling our dataset (for SQLI vulnerability detection) is essential for several reasons. We utilized SQLMap to assist with labeling, while (Mishra, 2019) used Libinjection. To verify our labels, we executed all queries (12566 records) on database, ensuring improved classification of SQLI instances. Labeling allows us to distinguish between different types of SQLI attacks, such as union-based, error-based, and time-based injections, providing granularity in understanding attack vectors. The categorization aids in developing and training ML models that can identify and classify SQLI vulnerabilities, enhancing the security posture of database systems. Labeling each query based on its vulnerability characteristics enabled robust evaluation of detection algorithms, ensuring they can effectively differentiate between vulnerable and non-vulnerable SQL queries, thereby mitigating potential risks posed by SQLI exploits. This approach not only supports proactive security measures but also contributes to

advancing research and practices in safeguarding against SQLI attacks. While categorizing SQLI queries, identified as vulnerable (injected) are assigned the label 1, whereas those deemed non-vulnerable (non-injected) are assigned the label 0. We meticulously assigned distinct type names (whether it's union-based SQLI, boolean-based SQLI, error-based, and time-based attacks etc.) to various SQLI attacks to facilitate improved identification and targeted mitigation strategies. Original form of dataset contains SQLI queries, labels, and types as shown in following table.

**Table.2:** Original Form of Dataset

Query	Label	Types
SELECT * FROM users WHERE age 0 BETWEEN 18 AND 30		none
SELECT 3109 FROM 1 (SELECT(SLEEP(10)))		time based
SELECT city FROM Users WHERE id = 96 0		none
SELECT * FROM users WHERE username = 1 'admin' -- AND password=123		comment based

This table provides a glimpse of an original dataset containing queries. Each row represents a query alongside its corresponding label and type of vulnerability detected. For instance, the first row features a standard query without any identified vulnerabilities (labeled as "none"), selecting user data based on age criteria. The second row demonstrates a query labeled as "time based," indicating it includes a sleep function potentially exploitable for time-based SQLI attacks. The third row showcases a query categorized as "none," extracting city from users where id is equal to 96. Lastly, the fourth row illustrates a query labeled as "comment based," which includes a comment that affects the query's execution, potentially indicating susceptibility to SQLI via comment manipulation. This dataset serves as a foundation for studying and developing methods to detect and mitigate SQLI vulnerabilities in database systems. To conduct experiments for both tasks: SQLI vulnerability identification and SQLI type detection, we divided our original dataset into two subsets for running experiments efficiently (as shown in following tables).

**Table. 3:** A few rows from SQLI type Detection Dataset

Query	Types
SELECT * FROM Users WHERE age BETWEEN 18 AND 30	none
1%" ) ) ) union all select null#	syntax error
SELECT 3109 FROM (SELECT(SLEEP(10)))	time based
SELECT * FROM users WHERE username = " OR 7627=7627 -- ';	boolean based
GTID_SUBSET(CONCAT(0x716a627171,(SELECT(ELT(4648=4648,1))),0x71706a7671),4648)	error based
UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,CONCAT(0x7171766271,0x546964547a49577677786b4d65516b43447842706e664c637a43734b6a7a4979784f5854576a7a4b,0x7171627071),NULL-- &SELECT * FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'	union based
SELECT * FROM users WHERE id = 1 OR 1=1;	tautology based
SELECT * FROM users WHERE username = 'admin' --	comment based
select * from users where id = 1 or 1#" ( union select 1,version ( ) -- 1	special character

**Table. 4:** A few rows from SQLI Vulnerability Identification Dataset

Query	Label
SELECT * FROM products WHERE manufacturer = 'Samsung'	0
SELECT * FROM USERS WHERE ID = '1' OR @ @1 = 1 UNION SELECT 1,VERSION ( ) -- 1'	1

These both tables provide an overview of a SQLI detection dataset, highlighting various types of SQL queries and their corresponding labels. One categorizes SQL queries based on their vulnerability types, including syntax errors, time-based, boolean-based, error-based, union-based, tautology-based, comment-based, and those with special character issues. Each query exemplifies different forms of SQLI attacks commonly encountered in database security assessments. Other supplements this by labeling queries as either vulnerable (labeled 1) or non-vulnerable (labeled 0), focusing on identifying SQLI vulnerabilities across different query structures and scenarios. Together, these tables serve as valuable resources for detecting and mitigating SQLI vulnerabilities in real-world applications and systems.

Balancing dataset refers to the process of addressing class imbalances within a dataset, where some classes (e.g., SQL injection types) are significantly underrepresented compared to others. This imbalance can lead to biased model performance, where the model might favor the majority class and underperform on the minority class. Balancing techniques aim to ensure that each class is represented more equally in the training data, improving the model's ability to learn and predict all classes accurately. Common techniques include resampling, where we oversample the minority class (e.g., generating synthetic examples using SMOTE) or undersample the majority class to create a more balanced dataset. Data augmentation can also be used to synthetically generate more examples of underrepresented SQLI types. Additionally, employing class weighting in the model's training process helps adjust for imbalances by giving more importance to the minority class. These techniques help improve the model's ability to detect less common SQLI vulnerabilities and enhance overall detection performance. Then following steps (feature extraction, pre-processing, tokenization, statistical feature extraction) are applied to both subsets of the dataset.

### **3.2. FEATURES EXTRACTION:**

Feature extraction involves systematically analyzing each SQLI query to identify and quantify various characteristics that are indicative of SQLI vulnerabilities. This process is crucial because it transforms raw SQL queries into structured data with meaningful attributes such as query length, presence of SQL keywords, special characters, and specific patterns like union-based or time-based attacks. These features provide essential information for training machine learning models to accurately detect and classify SQLI attempts. By extracting these features, security systems can proactively identify potential threats, strengthen database defenses, and mitigate the risks posed by SQLI attacks. (Mishra, 2019) extracted only two key features: categories and size. (Mishra, 2019) categories feature encompasses all types of SQLIs present in the dataset, facilitating comprehensive training of the model to accurately identify and differentiate between different SQLI attack patterns. On the other hand, the size feature of

(Mishra, 2019) denotes the dataset's ample volume, ensuring sufficient data for robust model training. The larger dataset size enhances the model's ability to generalize well to new and unseen SQL queries, thereby improving its effectiveness in SQLI detection and classification tasks. Besides, (Li et al, 2019) conducted a thorough analysis of SQLI vulnerabilities, identifying 30 unique features from SQLI queries. These features encompassed the length of SQL statements, the count of numeric characters, the number of null values, and prefixes etc. (Li et al, 2019) provided a foundational understanding of the intricate patterns and characteristics associated with SQLI attacks. Building on this groundwork, we significantly expanded the feature extraction process, identifying and incorporating over 70 features from SQLI queries. This comprehensive set of features includes 'num\_special\_chars', 'num\_comments', 'num\_logical\_operators', 'union\_based' and many more. The impact of extracting a larger set of features is substantial. By integrating more features, we enhance the accuracy of SQLI detection models, facilitating improved detection of SQLI vulnerability. The expanded feature set contributes to the robustness of ML and DL models, enabling them to generalize more effectively to unseen data while reducing false positives and false negatives. The expansion is driven by the increasing complexity of SQLI attacks. As attackers develop more advanced techniques, it becomes essential to capture a broader range of characteristics to stay ahead in detecting and preventing SQLI vulnerabilities. By expanding the feature set, we aim to create a more resilient and effective detection system, capable of identifying even the most subtle and sophisticated SQLI attempts. We extracted more than 70 features using the using regular expressions (regex), which facilitated the identification of key attributes such as SQL keywords, token sequences, and logical operators. Regex enabled systematic pattern recognition and extraction of features that characterized different types of SQLI vulnerabilities and query behaviors. Extracting features from our dataset is crucial for improved detection and classification of SQLI vulnerabilities. By extracting these features, we can assess the vulnerability of SQL queries to malicious attacks, enabling proactive security measures and enhancing the robustness of database systems.

'num\_comments\_multiline', 'num\_unions', 'num\_selects', 'num\_and', 'num\_or', 'num\_equals', 'num\_semicolons', 'num\_parentheses', 'has\_order\_by', 'has\_group\_by', 'has\_limit', 'has\_offset', 'has\_into', 'has\_load\_file', 'num\_hex'. The feature table is shown below.

**Table.5:** Feature Set

List of Features				
length	num_keyword s	num_special_cha rs	num_comments	num_logical_ operators
union_based	time_based	tautology_based	num_conditions	has_subquery
has_select	has_union	has_insert	has_update	has_delete
has_drop	has_alter	has_create	has_exec	has_grant
has_revoke	has_truncate	num_'	num_'	num_"
num_--	num_#	num_/*	num_*/	num_ =
num_<	num_>	num_!	num_	num_&
has_concat	has_substr	has_substring	has_left	has_right
has_mid	has_char	has_ascii	has_hex	has_unhex
has_md5	has_sha1	has_sha256	has_pattern_sele ct_.*_from	has_pattern_u nion_.*_selec t
has_pattern_se lect_.*_where	has_pattern_se lect_.*_order_ by	has_pattern_selec t_.*_group_by	num_numbers	num_strings
num_booleans	num_nulls	num_comments_i nline	num_comments _multiline	num_unions
num_selects	num_and	num_or	num_equals	num_semicol ons
num_parenthe ses	has_order_by	has_group_by	has_limit	has_offset

---

has_into	has_load_file	num_hex
----------	---------------	---------

---

This table outlines a comprehensive set of features extracted from SQL queries for SQLI vulnerability detection. These features include basic attributes such as query length, the number of SQL keywords, special characters, comments, and logical operators. It also shows specific SQLI patterns like union-based, time-based, and tautology-based attacks. The presence of various SQL commands and functions (SELECT, UNION, INSERT, UPDATE, DELETE, DROP, ALTER, CREATE, EXEC, GRANT, REVOKE, TRUNCATE) is tracked, along with character and operator counts (single quotes, double quotes, comment indicators, equality operators, comparison operators, and logical operators). Additionally, the table captures the occurrence of SQL functions and patterns (CONCAT, SUBSTR, CHAR, HEX, MD5, SHA1, SHA256) and specific query patterns (e.g., SELECT followed by FROM, UNION followed by SELECT). Numerical counts of different elements such as numbers, strings, booleans, NULLs, and inline or multiline comments are included, alongside structural features like the presence of ORDER BY, GROUP BY, LIMIT, OFFSET, INTO, and LOAD\_FILE clauses. This extensive feature set allows for detailed analysis and improved classification of SQLI vulnerabilities.

### 3.3 DATA PRE-PROCESSING

Data preprocessing is a critical step in the data analysis pipeline that involves cleaning, transforming, and organizing raw data into a structured format suitable for analysis. This process typically includes tasks such as handling missing values, removing duplicates, standardizing data formats, and scaling numerical features. It plays a pivotal role in improving the quality and usability of datasets by addressing common issues such as noise, inconsistencies, and irregularities. By preparing the data adequately before analysis, data preprocessing ensures that subsequent analytical tasks yield improved results and insights. Additionally, it can help mitigate the risk of biased outcomes and enhance the performance of ML models by optimizing the input data for training and validation. Overall, data preprocessing is essential for extracting meaningful information from raw data and maximizing the effectiveness of data-driven decision-making processes. Cleaning a dataset for SQLI vulnerability detection involves the process of preparing the dataset by addressing issues such as handling missing values, removing duplicates and standardizing text data through normalization (e.g., converting to lowercase, removing extra spaces etc.). (Roy et al, 2022) emphasize preprocessing steps to enhance the quality and reliability of data for SQLI vulnerability detection. (Roy et al, 2022) approach involves removing duplicates, handling missing values, and addressing outliers to ensure the dataset's integrity. In our implementation, we tailored our preprocessing steps specifically for



queries. We prioritized cleansing 'Query' text data by addressing missing values and employing text cleaning techniques such as removing extra spaces and converting text to lowercase. These steps normalize the data, making it uniform and easier to analyze for irregularities or injected SQL commands. By ensuring data uniformity through these preprocessing steps, we aim to improve the robustness of our SQLI vulnerability detection models. This process not only enhances model accuracy but also strengthens the overall security posture by enabling more effective identification and mitigation of SQLI threats. Moreover, through this diligent cleaning procedure, potential biases and vulnerabilities within the dataset can be mitigated, leading to the development of a more robust and dependable SQLI vulnerability detection model. We cleaned both subsets (the entire dataset). Here is a glimpse of a few rows of queries (from the dataset) before and after cleansing.

**Table. 6:** Cleaning Process

Before Cleaning	After Cleaning
SELECT * FROM USERS WHERE ID = '1' OR @ @1 = 1 UNION SELECT 1,VERSION ( ) -- 1'	select * from users where id = '1' or @ @1 = 1 union select 1,version ( ) -- 1'
SELECT * FROM USERS WHERE ID = 1 OR 1#" ( UNION SELECT 1,VERSION ( ) - - 1	select * from users where id = 1 or 1#" ( union select 1,version ( ) -- 1
SELECT NAME FROM SYSCOLUMNS WHERE ID = ( SELECT ID FROM SYSOBJECTS WHERE NAME = TABLENAME' ) --	select name from syscolumns where id = ( select id from sysobjects where name = tablename' ) --

This table illustrates the cleaning process applied to SQL queries for SQLI vulnerability detection. The "Before Cleaning" column shows original SQL queries containing various SQLI attack patterns such as UNION SELECT and comments. After cleansing, represented in the "After Cleaning" column, these queries are normalized by converting all text to lowercase, removing unnecessary spaces, and standardizing SQL syntax. This process ensures uniformity and prepares the queries for effective analysis and detection of potential SQLI vulnerabilities. Each query is transformed to maintain its intended functionality while mitigating the risk of exploitation by malicious SQLI attempts.

### 3.4. TOKENIZATION

The next step is tokenization, a process of breaking down text into smaller units called tokens, which can be words, phrases, or symbols. Tokenization is crucial for natural language processing (NLP) tasks as it helps in understanding and analyzing the structure and meaning of the text. In the context of SQLI vulnerability detection, tokenization is particularly important because it enables a detailed analysis of SQL queries by identifying and categorizing different elements within the query. (Mishra, 2019) tokenization steps involve the usage of regular expressions to group characters for lexical analysis, particularly in the context of SQLI datasets. In contrast, our approach to tokenization focused on leveraging the NLTK library's ``word_tokenize`` function for splitting SQL queries into individual tokens. This method allows us to classify tokens into types such as SQL keywords, numbers, punctuation, and operators using regular expressions, enhancing the granularity of analysis. We apply this process to the 'SQL\_Query' column in our ``SQLI_data`` DataFrame, storing tokenization results in new columns for tokens, token types, token vlaue and token counts. The importance of token count in SQLI vulnerability detection lies in its ability to reveal query complexity; longer or more intricate queries often indicate potential SQLI attempts with nested or concatenated statements. Extracting token types aids in identifying patterns associated with SQLI attacks, such as unusual combinations of SQL keywords or operators. This structured approach for tokenization not only improves the understanding of query structures but also enhances the detection accuracy of SQLI vulnerabilities by capturing detailed syntactic and semantic elements within queries. Moreover, the process of tokenization, including the extraction of token types, values and counts, provides a structured way to analyze the queries. This structured analysis is essential for detecting and classifying SQLI vulnerabilities, as it helps in identifying anomalies and patterns that are indicative of malicious activities. By breaking down and categorizing query components, tokenization enhances the ability to detect and mitigate potential security threats. A glimpse of tokenization is shown below.

**Table. 7:** Tokenization Process

SQL_Query	Tokens	Token_Types	Token_Count
-----------	--------	-------------	-------------

select * from users where id = '1' or @ @1 = 1 union select 1,version ( ) -- 1'	[select, *, from, users, where, id, =, ', 1, ', or, @, @, 1, =, 1, union, select, 1, ,, version, (, ), --, 1, ']	[keyword, operator, keyword, other, keyword, other, operator, other, number, other, keyword, other, other, number, operator, number, other, keyword, number, punctuation, other, other, other, other, number, other]	26
select * from users where id = 1 or 1#" ( union select 1,version ( ) -- 1	[select, *, from, users, where, id, =, 1, or, 1, #, ", (, union, select, 1, ,, version, (, ), --, 1]	[keyword, operator, keyword, other, keyword, other, operator, number, keyword, number, other, other, other, other, keyword, number, punctuation, other, other, other, other, number]	22
select name from syscolumns where id = ( select id from sysobjects where name = tablename' ) -	[select, name, from, syscolumns, where, id, =, (, select, id, from, sysobjects, where, name, =, tablename, ', ), --]	[keyword, other, keyword, other, keyword, other, operator, other, keyword, other, keyword, other, operator, keyword, other, operator, other, other, other, other]	19

The table shows the tokenization process applied to the queries as part of SQLI vulnerability detection. Each row represents a SQL query from the dataset, with the "SQL\_Query" column showing the original queries and the subsequent columns detailing their tokenized forms. The "Tokens" column lists each query broken down into individual tokens, including SQL keywords, operators, numbers, punctuation, and other elements. The "Token\_Types" column categorizes these tokens into specific types using regular expressions, enabling the identification of distinct components within each query. The "Token\_Count" column quantifies the total number of tokens in each query, reflecting its complexity and structural characteristics. This tokenization process is essential for analyzing and detecting SQLI vulnerabilities by providing structured data that facilitates accurate pattern recognition and anomaly detection in SQL queries.

### 3.5. STATISTICAL FEATURES EXTRACTION (USE OF G-TEST AND ENTROPY)

In SQLI vulnerability detection process, tokenization is followed by the calculation of statistical measures such as G-test score and entropy. Statistical feature extraction involves quantifying data characteristics to derive meaningful insights. As there are numerous statistical feature extraction techniques (e.g.: Chi-Square tests) available. We chose G-test scores and entropy for our SQLI detection methodology due to their effectiveness in assessing token frequency deviations and measuring query randomness, respectively. G-test scores help identify irregular token patterns indicative of SQLIs, while entropy distinguishes structured from unpredictable query behaviors. These metrics enhance the ability to detect anomalies, improving the accuracy of our detection model and reinforcing cybersecurity defenses with a robust, data-driven approach. After tokenization, where SQL queries are broken down into individual tokens and classified into types, the next step involves quantifying the distribution and randomness of these tokens. (Mishra, 2019) calculated both observed and expected G-test scores, leading to the calculation of a mean G-test score from these values. In contrast, our method concentrated solely on deriving the observed G-test score. This simplification significantly reduced computational complexity and streamlines the analysis process, optimizing its efficiency for SQLI detection applications. By directly evaluating observed deviations without the need for calculating expected scores, our approach swiftly provided insights into potentially malicious query patterns. This enhanced the responsiveness and effectiveness of our SQLI detection model in cybersecurity applications.

The G-test score and entropy are two statistical measures used for this purpose: The G-test score measures how much the observed frequencies of tokens deviate from the expected frequencies, assuming a uniform distribution. In SQLI detection, a high G-test score can indicate that the distribution of tokens is unusual compared to normal queries. Malicious queries often contain patterns or repetitions of certain tokens that deviate from standard query structures, making the G-test score a useful indicator of potential SQLI attacks. Here is the mathematical form of G-test score as shown in following equation 1.

$$G = 2 \sum_i O_i \ln(E_i O_i)$$

**Equation.1.** Mathematical form of G test score

In this context:

- $O_i$  represents the observed frequency within each category.
- $E_i$  signifies the expected frequency within each category, usually, computed assuming independence between categories.

Besides, entropy measures the randomness or unpredictability in the distribution of tokens. Lower entropy can indicate a more predictable, repetitive pattern, which is often a characteristic of SQLI attacks where specific keywords and operators are repeated to exploit vulnerabilities. Conversely, higher entropy suggests a more varied and less predictable query, which is less likely to be an SQLI attempt. The formula for entropy is given below.

$$\text{Entropy} = -\sum p(X)\log p(X)$$

**Equation.2.** Mathematical form of Entropy

In this context:

- $p(X)$  is defined as the ratio of occurrences of  $(X)$  to the total number of features.

By calculating these metrics for each tokenized query, the G-test score and entropy provide a quantitative basis for identifying suspicious queries. In combination with the detailed analysis provided by tokenization, these measures help in detecting anomalies that are indicative of SQLI vulnerabilities. The role of G-test score and entropy in SQLI vulnerability detection is to highlight deviations from normal query behavior, thereby enhancing the ability to detect and mitigate potential security threats. The following table provides a clearer understanding by showcasing SQL\_Query, tokens, G-test score, and entropy.

**Table. 8:** Calculating Statistical Features

SQL_Query	Tokens	G_Test_Score	Entropy
select * from users where id = '1' or @ @1 = 1 union select 1,version ( ) -- 1'	[select, *, from, users, where, id, =, ', 1, ', or, @, @, 1, =, 1, union, select, 1, ,, version, (, ), --, 1, ']	8.909893	3.840266
select * from users where id = 1 or 1#" ( union select 1,version ( ) -- 1	[select, *, from, users, where, id, =, 1, or, 1, #, ", (, union, select, 1, ,, version, (, , --, 1]	5.291052	3.913977
select name from syscolumns where id = ( select id from	[select, name, from, syscolumns, where, id, =, (, select, id,	2.214927	3.616349

sysobjects	where	from,	sysobjects,
name = tablename') -	where,	name,	=,
-		tablename, ', ), --]	

This table presents statistical features extracted from SQL queries for SQLI detection. Each query is tokenized into individual tokens, and key metrics such as G-test score and entropy are calculated to assess potential vulnerabilities. The G-test score measures the deviation of observed token frequencies from expected frequencies under a uniform distribution assumption, with higher scores indicating more anomalous query patterns that may suggest SQLI attempts. Conversely, entropy quantifies the randomness and predictability of token distributions within queries; lower entropy values indicate more structured and potentially exploitable query patterns, while higher values suggest greater unpredictability. These metrics provide a quantitative basis for distinguishing between normal and potentially malicious SQL queries, enhancing the effectiveness of cybersecurity defenses in detecting SQLI vulnerabilities.

### 3.6. MODEL TRAINING AND TESTING

(Mishra, 2019) selected Naïve Bayes and GB algorithms for experiments. But we chose KNN, RF, GB, and MLP specifically for SQLI vulnerability detection due to their respective strengths and suitability for this task. KNN was selected for its simplicity in pattern recognition and ability to classify based on similarity measures, which is effective for detecting anomalies in SQL query patterns. RF, known for ensemble learning and robustness against overfitting, is ideal for handling complex datasets like those involving diverse SQLI patterns. GB, with its sequential training of weak learners, enhances model accuracy and can effectively capture intricate relationships within SQL queries. MLP, a powerful neural network architecture, was chosen for its capability to learn complex mappings and nonlinearities inherent in SQLI detection tasks. Other algorithms were not chosen for specific reasons related to their characteristics and applicability to SQLI detection. For instance, Naive Bayes classifiers assume independence among features, which does not hold in SQL query contexts where token sequences exhibit interdependencies. Decision Trees, while interpretable, struggles with overfitting and capturing nuanced SQLI patterns without ensemble techniques like RF or GB. Support Vector Machines (SVMs), although effective in high-dimensional spaces, requires extensive tuning and kernel selection to handle the varied nature of SQL queries and potential feature interactions. Therefore, KNN, RF, GB, and MLP were prioritized based on their robust performance and adaptability to the complexities inherent in SQLI vulnerability detection tasks.

The both subsets of dataset were analyzed using specific algorithms, with key metrics like accuracy provided as benchmark for evaluating their performance. Accuracy measures the

model's accuracy and in this phase, different ML and DL algorithms are applied to the dataset `SQLI\_data` for SQLI vulnerability detection. Non-numeric data undergoes transformation using `LabelEncoder`, crucial for converting categorical variables into numerical representations necessary for ML algorithms. This step ensures the classifier can effectively interpret and learn from these features. Imputation is equally vital in dataset preparation, handling missing values to prevent compromising the model's performance. Missing values in the dataset are imputed using the mean strategy, where each missing entry in the features is replaced with the average value of that feature computed from the training set. This approach ensures the classifiers utilize complete data for training. Following encoding and imputation, the dataset is split into features (`X`) whereas `X` contains all columns except output column and the target variable (`y`) contains only output column. A train-test split reserves 20% of the data for testing, assessing the model's ability to generalize to unseen data. Configured with 300 estimators, a learning rate of 0.1, maximum depth of 3, minimum samples split of 2, and random state 42, GB classifier was optimized. For KNN, we considered just 5 neighbors. Additionally, for RF, it was configured with 100 estimators and random state 42. Moreover, MLP used hidden layer sizes of (100), maximum iterations of 300, and random state 42. Trained on the imputed training data (`X\_train\_imputed` and `y\_train`), the classifiers predict SQLI vulnerabilities on the test set (`X\_test\_imputed`). The model's performances are evaluated using `accuracy` quantifying its effectiveness in detecting SQLI vulnerability. Here is a hyper-parameter tuning table for all the algorithms including: RF, GB, KNN, and MLP. This table details the optimal settings and configurations for each algorithm, ensuring their performance is fine-tuned for the specific tasks at hand.

**Table. 9:** Hyper-parameter tuning

Classifiers	Hyper-parameters
GB	n_estimators=300, learning_rate=0.1, max_depth=3, min_samples_split=2, random_state=42
K-Nearest Neighbors	n_neighbors=5
RF	n_estimators=100, random_state=42
MLP (MLP)	hidden_layer_sizes=(100), max_iter=300, random_state=42

The table provides a concise overview of the hyper-parameters tailored for each classifier used in the models. For GB, the parameters focused on enhancing ensemble learning with 300 estimators, a learning rate of 0.1, a maximum depth of 3 for individual trees, a minimum number of samples required to split an internal node set to 2, and a fixed random state for reproducibility. KNN was configured with 5 neighbors for proximity-based classification. RF employed 100 decision tree estimators, maintaining consistency with a specific random state. MLP neural network utilized a single hidden layer of 100 neurons, a maximum iteration limit of 300, and a uniform random state to ensure consistent initialization. Each set of parameters is tailored to optimize performance and generalization across their respective classifiers, balancing complexity and computational efficiency.

Accuracy was chosen as the evaluation metric for SQLI vulnerability detection because it provides a clear measure of the model's overall performance by indicating the proportion of correctly classified instances out of the total. In scenarios where the dataset is relatively balanced between vulnerable and non-vulnerable queries, accuracy effectively reflects how well the model identifies both types of queries. It also allows for straightforward comparison between different models or algorithms and is practically relevant as it ensures a high proportion of correctly identified queries, which is crucial for effective security. Here is its mathematical form.

$$Accuracy = \frac{TP + TN}{Total\ Population}$$

**False Positive (FP):** Instances that are incorrectly classified as positive (or true) when they are actually negative (or false).

**False Negative (FN):** Instances that are incorrectly classified as negative (or false) when they are actually positive (or true).

**True Positive (TP):** Instances that are correctly classified as positive (or true) by the system.

**True Negative (TN):** Instances that are correctly classified as negative (or false) by the system.

These terms are essential for calculating metrics such as accuracy, precision, recall, and F1 score.

### 3.6.1. Gradient Boosting

GB is a crucial ML technique for experimental research due to its scalability, efficiency, robustness against overfitting, flexibility in data handling, and superior predictive performance.



Examples include XGBoost, LightGBM, and CatBoost. These algorithms handle various types of data and provide reliable feature importance ratings. Modern GB algorithms are efficient for large-scale datasets and are essential resources for academics extracting knowledge and forecasts from experimental data. They use a one-vs-all approach for multi-class classification problems, with the final prediction determined by selecting the class with the highest probability. GB stands out as a potent technique for identifying SQLI vulnerabilities. Renowned for its high predictive accuracy, GB exhibits resilience against overfitting and provides valuable insights into feature importance. Here is its mathematical form.

$$F(x) = F_o(x) + \sum_{m=1}^M r_m h_m(x)$$

Here  $F(x)$  is the final model prediction,  $F_o(x)$  is initial model prediction (often set to the means of target values). Moreover,  $\sum_{m=1}^M r_m h_m(x)$  is the sum of weak learners  $h_m(x)$  scaled by corresponding coefficients  $r_m$ .  $M$  is the total number of boosting iterations or weak learners.

By amalgamating numerous weak learners into a robust learner, GB adeptly captures intricate patterns within the dataset. Moreover, its flexibility in hyper-parameter tuning empowers practitioners to tailor model performance to the nuances of individual datasets and detection criteria. GB is known for its high predictive accuracy and resilience against overfitting. It combines numerous weak learners into a robust learner to capture intricate patterns. The results for SQLI vulnerability detection on our dataset are shown below. As we know that in our dataset: ‘Label Dataset’ identifies vulnerable and non-vulnerable SQLI and ‘Type Dataset’ detects the type of SQLI vulnerability.

**Table. 10:** Results for GB Algorithm

<b>Dataset</b>	<b>Accuracy (%)</b>
SQLI Vulnerability Dataset	97.91

The table presents the accuracy results for the SQLI Vulnerability Dataset, achieving an impressive accuracy rate of 97.91%. This high accuracy indicates the dataset's effectiveness in training models to accurately detect SQL injection vulnerabilities, showcasing its robustness and reliability in cybersecurity applications. Such performance highlights the dataset's capability to enhance the precision of SQLI detection and support the development of effective security measures.

### 3.6.2. Random Forest

RF is a ML technique that uses ensemble learning to build multiple decision trees during training. It uses randomness in features to minimize correlation and capture various patterns. Decision trees are trained by recursively separating data based on predetermined criteria. The final outcome is the average or mode prediction derived from the combined predictions of all decision trees. RF is an effective ensemble classification method, able to handle high-dimensional data with non-linear correlations and feature importance ranking approaches. Here is its mathematical form.

$$y^{\wedge} = \frac{1}{N} \sum_{i=1}^N T_i(x)$$

Here,  $y^{\wedge}$  is predicted output.  $N$  is the number of decision trees in the forest. Moreover,  $T_i(x)$  is the prediction made by  $i - th$  decision tree for input  $x$ .

RF is a popular choice due to its simplicity, scalability, and versatility in improving classification outcomes. It emerges as a resilient method for detecting SQLI vulnerabilities, showcasing robustness against overfitting, a feature importance ranking system, flexibility to capture non-linear patterns, scalability, capability for outlier detection, and resilience against noise. With its ensemble structure and utilization of multiple decision trees, RF adeptly manages noisy datasets, while its feature importance ranking aids in pinpointing crucial features. Additionally, RF's capacity to grasp intricate non-linear relationships between features and target variables makes it suitable for handling vast datasets within web applications and databases. The results for SQLI detection dataset using RF are shown below.

**Table. 11:** Results for RF Algorithm

Dataset	Accuracy (%)
SQLI Vulnerability Dataset	93.61

The table displays the accuracy achieved in detecting SQL injection (SQLI) vulnerabilities using a specialized dataset. With an impressive accuracy rate of 93.61%, the model demonstrates a high level of precision in identifying SQLI vulnerabilities, reflecting the

effectiveness of the dataset in training the model to recognize and classify these security threats accurately. This performance underscores the dataset's robustness and its contribution to enhancing SQLI vulnerability detection.

### 3.6.3. K-NEAREST NEIGHBOR (KNN)

KNN algorithm is a versatile machine learning tool that excels in experimental research due to its simplicity, non-parametric flexibility, and adaptability to complex patterns. It is robust to outliers and has a transparent decision-making process. KNN is particularly efficient with small to moderate-sized datasets and can handle larger ones with advancements in computational capabilities and optimization techniques. For multi-class classification problems, the class label is determined by calculating distances, finding KNN, and majority voting. Here is its mathematical form.

$$d(x_{new}, x_i) = \sqrt{\sum_{j=1}^M (x_{new,j} - x_{i,j})^2}$$

Where M is the total number of features and  $x_{new}$  is the new test point to be classified. KNN is valuable for exploring complex datasets and generating insights across diverse research domains. It stands out as a straightforward, adaptable, and interpretable method for detecting SQLI vulnerabilities. Its simplicity facilitates easy implementation, while its lack of a training phase and capability to address both classification and regression tasks make it versatile. The transparent decision-making process of KNN makes it well-suited for handling complex or non-parametric datasets. Furthermore, its localized decision boundaries, support for incremental learning, and resilience to outliers enhance its effectiveness. Moreover, KNN's nonparametric characteristics enable seamless integration of new data without the need for retraining the entire system. The results for SQLI detection dataset using KNN are shown below.

**Table. 12:** Results for KNN Algorithm

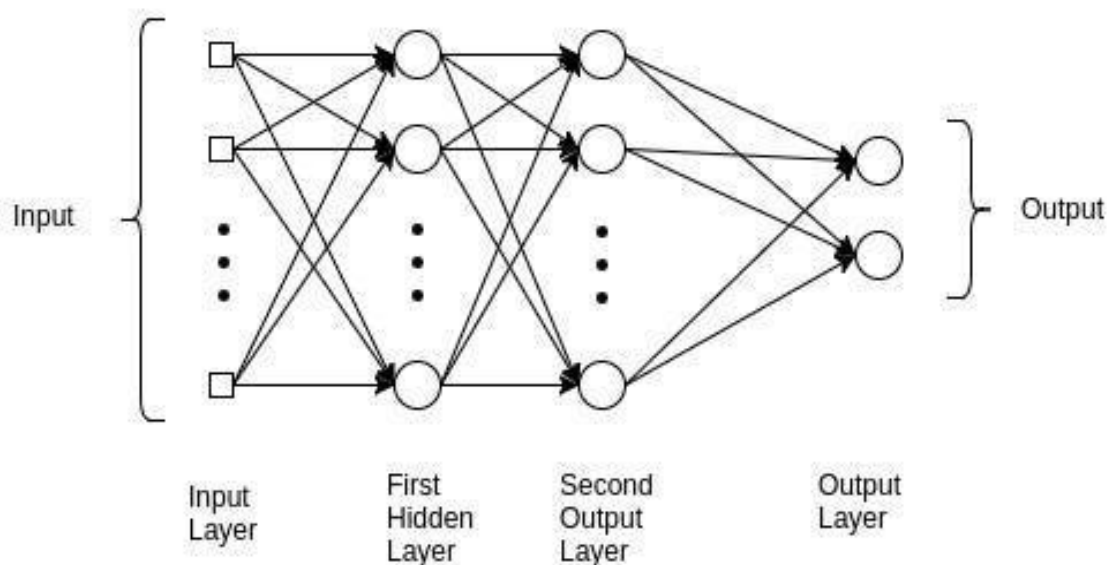
<b>Dataset</b>	<b>Accuracy (%)</b>
SQLI Vulnerability Dataset	96.00

The table displays the accuracy achieved by the SQLI Vulnerability Dataset in detecting SQL injection vulnerabilities, with a notable accuracy of 96.00%. This high accuracy reflects the

dataset's robustness in accurately identifying and classifying SQL injection threats, demonstrating its effectiveness in enhancing security measures against SQL injection attacks. The performance metrics indicate that the dataset significantly contributes to improving the reliability of vulnerability detection systems.

### 3.4.4. MULTILAYER PERCEPTRON (MLP)

MLPs are crucial in experimental research due to their versatility, adaptability, and ability to simulate complex data linkages. They can handle tasks like classification, regression, and unsupervised learning, and can handle non-linearity and feature learning. MLPs' scalability allows for efficient management of complex datasets and provide interpretability, enabling researchers to understand model behavior and decision-making processes. Regularization approaches help avoid overfitting and enhance generalization performance. Ensemble learning can improve predictive performance with MLPs, leading to better outcomes in experiments. MLPs are essential for resolving experimental issues and expanding our understanding of science. Here is a basic structure of an MLP:



**Figure.3:** Basic Structure of MLP (Taken From: A Simple Overview of Multilayer Perceptron (MLP) Deep Learning)

MLP is highly valued in SQLI vulnerability detection due to its ability to handle non-linear relationships, flexibility, feature learning, generalization, scalability, and adaptability. MLPs are adept at capturing intricate patterns, enabling the identification of SQLI vulnerabilities that may not follow linear separability. Additionally, they possess the capability to automatically learn relevant features from input data, making them well-

suited for analyzing extensive datasets. Here is its mathematical form.

$$y^{\wedge} = \sigma(W_2 \cdot \sigma(W_1 \cdot X + b_1) + b_2)$$

Here  $X$  is the input features.  $W_1, W_2$  are the weight matrix for hidden layer and output layer respectively.  $b_1, b_2$  are the biases of hidden layer and output layer respectively.

Moreover, MLPs demonstrate adaptability, allowing them to adjust to changes in data distribution and evolving attack methodologies. When tuning parameters for the MLP model, several key factors are essential to consider. First, adjusting `hidden_layer_sizes` allows for controlling the complexity of the model by determining the number of neurons in each hidden layer. Second, the choice of `activation` function, such as 'relu', 'tanh', or 'logistic', plays a critical role in shaping the model's performance and its ability to capture non-linear relationships in the data. Lastly, setting `random_state` ensures reproducibility of results across different runs by fixing the random seed used during training, thereby providing consistent outcomes for evaluation and comparison. These considerations collectively influence how effectively the MLP model can learn and generalize from the data, impacting its overall performance in classification or regression tasks. Careful tuning of these parameters allows for optimization of MLP model performance for the specific task at hand. The results for SQLI detection dataset using MLP are shown below.

**Table. 13:** Results for MLP Algorithm

Dataset	Accuracy (%)
SQLI Vulnerability Dataset	94.19

The table displays the accuracy results for the SQLI Vulnerability Dataset, which achieved an impressive accuracy of 94.19%. This high level of accuracy indicates that the dataset effectively captures and represents SQL injection vulnerabilities, demonstrating its robust quality and reliability for training machine learning models in the context of cybersecurity. This performance highlights the dataset's potential to significantly enhance SQL injection detection and contribute to the development of more secure applications.

### 3.7. PREVENTIVE STRATEGIES FOR SQLI VULNERABILITY

SQLI poses a persistent and severe security threat to web applications, allowing attackers to manipulate databases through malicious SQL queries. Employing effective prevention measures is crucial for mitigating the risks associated with SQLI attacks. (Joshi et al, 2022)

highlighted several key strategies, including input validation, parameterized queries, and stored procedures, which serve as fundamental techniques to mitigate SQLI attacks. Additionally, they emphasized the use of web application firewalls (WAFs) for an added layer of protection, effectively blocking malicious requests before they reach the application. Similarly, (Sultana et al, 2023) discussed preventive measures such as client-side input sanitization and server-side query sanitization, underscoring the importance of both frontend and backend defenses against SQLI threats. Building upon this foundational work, our thesis presented a comprehensive suite of SQLI preventive strategies, ensuring robust protection against these vulnerabilities. Our strategies include parameterized queries, which prevent the execution of malicious SQL code by treating user inputs as parameters rather than executable code. Input validation is implemented to ensure that only expected and safe data is processed by the application. Escaping special characters is another crucial technique we employ to neutralize potentially harmful inputs. Furthermore, we advocate for the principle of least privilege, restricting database access rights to the minimum necessary for application functionality, thereby limiting the potential damage from a successful SQLI attack. The deployment of web application firewalls (WAFs) is also a key component of our strategy, providing a defensive barrier against SQLI attempts. Finally, we stress the importance of continuous security testing, including regular audits and updates, to identify and address new vulnerabilities as they emerge. By integrating these comprehensive preventive measures, our thesis not only aligned with but also extended the strategies discussed in the literature, offering a robust framework for SQLI prevention. Below, we discussed various preventive strategies along with code examples to illustrate their implementation.

Employing parameterized queries with prepared statements is one of the most potent strategies to thwart SQLI attacks. This technique involves using placeholders within SQL queries, which are then replaced with user input in a safe manner. By segregating SQL code from user input, parameterized queries ensure that any data provided by users is treated strictly as a value, rather than executable code. This effectively neutralizes attempts by attackers to inject malicious SQL commands, as the database recognizes the input as data and not as part of the SQL command structure. Moreover, parameterized queries enhance code readability and maintainability, making it easier for developers to identify and rectify potential vulnerabilities. This approach is supported by various database management systems and programming languages, providing a versatile and reliable method for securing applications against SQLI attacks. By incorporating parameterized queries into our security strategy, we create a robust defense mechanism that significantly reduces the risk of SQLI vulnerabilities, safeguarding the integrity and confidentiality of our database systems. Below, we present an example of parameterized and non- parameterized query implemented in Python.

**Table. 14:** Implementing Parameterized Query

Non- Parameterized Query	Parameterized Query
SELECT* FROM USERS	
WHERE username='[username]'	SELECT* FROM USERS WHERE
	username=? AND password=?
AND password='[password]'	

This table illustrates the implementation of parameterized query. The first column shows a non-parameterized query, where user inputs for `username` and `password` are directly embedded into the SQL statement, making it vulnerable to SQLI attacks. The second column demonstrates a parameterized query, where placeholders (`?`) are used instead of directly inserting user inputs. This approach ensures that the inputs are treated as data rather than executable code, thereby preventing SQLI vulnerabilities.

Validating and sanitizing user input is crucial to prevent SQLI attacks, ensuring that all data entered into the system conforms to expected formats and is free from malicious content. This process involves scrutinizing input at every point of entry to verify that it meets predefined criteria, effectively blocking any harmful data before it reaches the database. Techniques such as whitelisting, blacklisting, and regular expressions are commonly used for input validation. Whitelisting specifies acceptable input patterns, allowing only data that matches these patterns to be processed. This method is highly effective for fields with a limited and known set of valid inputs, such as dates, email addresses, or numerical values. Blacklisting, on the other hand, involves identifying and rejecting known malicious patterns, although it is generally less effective than whitelisting due to the constantly evolving nature of attack vectors. Regular expressions provide a flexible way to define complex validation rules, ensuring that inputs adhere to specific formats while excluding potentially dangerous characters. By rigorously applying these techniques, we can significantly reduce the risk of SQLI attacks, maintaining the integrity and security of our applications and databases. Input validation not only protects against SQLI but also enhances overall application security by mitigating other injection-based threats. Here's a basic illustration of input validation in Python.

**Table. 15:** Implementing Input Validation

Before Input Validation (non-parametrized query)	After Input Validation (parametrized query)
--	---

SELECT* FROM USERS WHERE username='alice'	SELECT* FROM USERS WHERE username=?
[1, 'alice', 'password123']	[1, 'alice', 'password123']

This table illustrates the implementation of input validation in SQL queries. The first column represents a scenario before input validation, showing a non-parameterized query where the username 'alice' is directly included in the SQL statement. In contrast, the second column demonstrates the application of input validation through a parametrized query. Here, the placeholder `?` is used for the username, and the actual value ('alice') is passed separately as a parameter during query execution. This approach ensures that user inputs are validated against expected formats or constraints before being used in SQL queries, reducing the risk of SQLI attacks by preventing unauthorized manipulation of query structure or data.

Escaping special characters in user input is a vital technique to neutralize their potential malicious impact within SQL queries, thereby thwarting SQLI attacks. Special characters, such as single quotes, semicolons, and backslashes, can alter the intended structure of SQL statements if not properly handled, allowing attackers to manipulate queries and gain unauthorized access to the database. By escaping these characters, we transform them into harmless literals that the database engine treats as plain text rather than executable code. Most programming languages and database management systems provide built-in functions or libraries specifically designed for this purpose, simplifying the implementation of character escaping. For instance, functions like `mysql\_real\_escape\_string()` in PHP or `SQLite3\_escape\_string()` in SQLite automatically add escape sequences to special characters in user inputs. This process ensures that even if an attacker attempts to inject malicious SQL code, the escaped characters prevent the code from being executed. Implementing character escaping as part of our security strategy adds an essential layer of defense, complementing other techniques such as input validation and parameterized queries. This comprehensive approach significantly enhances the resilience of our applications against SQLI attacks, safeguarding sensitive data and maintaining the integrity of our database systems. Below, we demonstrate an example in Python utilizing the `SQLite3` library.

**Table. 16:** Escaping Special Characters

Before Escaping Special Character	After Escaping Special Character (parametrized query)
-----------------------------------	--



SELECT* FROM USERS WHERE username='alice' password='password123' OR '1'='1'	AND	SELECT* FROM USERS WHERE username=? AND password=?
---	-----	---

This table illustrates the importance of escaping special characters in SQL queries, specifically within the context of parameterized queries. The first column depicts a vulnerable SQL query before escaping special characters, where the user input for `username` and `password` ('alice' and 'password123') is followed by a malicious condition (`OR '1'='1'`). This condition can potentially manipulate the query to return unintended results. In contrast, the second column shows the same query structure but implemented with parameterized queries, using placeholders (`?`) for `username` and `password`. By utilizing parameterized queries and escaping special characters appropriately, the SQL engine treats the inputs as data rather than executable code. This effectively neutralizes the impact of special characters like single quotes or logical operators within user inputs, mitigating the risk of SQLI attacks and ensuring secure query execution.

The Least Privilege Principle is a foundational security concept that dictates that every module, component, or user of a system should have the minimum level of access necessary to perform their functions. In the context of SQLI vulnerability detection, applying the Least Privilege Principle involves meticulously restricting the privileges of database users and application components to only those necessary for their intended operations. For instance, if a user or application component only needs to read data from a database, it should not have permissions to write or modify data. By minimizing access rights, we substantially reduce the attack surface, limiting the potential damage that an attacker can inflict if they manage to exploit an SQLI vulnerability. Even if an attacker gains access through an SQLI flaw, their ability to execute harmful operations, such as altering or deleting data, will be constrained by the restricted permissions. This principle also facilitates better monitoring and auditing of database activities, as any attempts to exceed granted privileges can be promptly identified and addressed. Implementing the Least Privilege Principle requires a thorough understanding of the access requirements of each user and component, along with regular reviews and updates to ensure that privileges remain appropriately restricted as system needs evolve. By integrating this principle into our security framework, we enhance the robustness of our defenses against SQLI attacks, ensuring a more secure and resilient database environment. Here is python code for this.

**Table. 17:** Applying Least Privilege Principle

Query: Before Applying Least Privilege Principle	Query: After Applying Least Privilege Principle
SELECT* FROM USERS WHERE username='alice'	SELECT id, city FROM USERS WHERE username='alice'

This table demonstrates the application of the Least Privilege Principle in SQL query design. The first column presents a query that retrieves all columns (`\*`) from the `USERS` table based on the `username` 'alice'. This approach grants broad access to user data, potentially exposing sensitive information beyond what is necessary for the query's intended purpose. In contrast, the second column exemplifies the same query refined after applying the Least Privilege Principle. Here, only specific columns (`id` and `city`) are selected from the `USERS` table for the `username` 'alice'. By limiting the query to retrieve only the essential data fields required for the application's functionality, this principle reduces the exposure of sensitive information and minimizes the impact of potential SQLI attacks. Implementing the Least Privilege Principle ensures that database access rights are aligned with operational needs, enhancing security and maintaining data confidentiality within the application environment.

Web Application Firewalls (WAFs) serve as a robust defense mechanism by filtering and blocking malicious requests at the application layer. These specialized security tools scrutinize incoming HTTP requests, analyzing them for patterns and signatures indicative of SQLI attempts. By examining the traffic before it reaches the web application, WAFs can proactively detect and thwart potential attacks, preventing them from exploiting vulnerabilities within the application. WAFs employ a variety of techniques, such as signature-based detection, anomaly detection, and behavior analysis, to identify and block malicious activities in real-time. Signature-based detection involves comparing incoming requests against a database of known attack patterns, while anomaly detection monitors for deviations from normal traffic behavior. Additionally, some WAFs use ML algorithms to adapt and improve their detection capabilities over time. By integrating a WAF into our security strategy, we add an essential layer of protection that complements other preventive measures, such as parameterized queries and input validation. This approach ensures a more comprehensive defense against SQLI and other web-based attacks, safeguarding our applications and data from malicious actors.

Continuous security testing is a critical practice that involves regular security assessments, code reviews, and vulnerability scans to proactively identify and remediate SQLI vulnerabilities. This ongoing process ensures that security measures remain effective and up-to-date in the face of evolving threats. Regular security assessments involve systematically evaluating the security

of web applications, often through penetration testing, to uncover potential weaknesses that are exploited by attackers. Code reviews, on the other hand, entail meticulously examining the source code to identify and correct security flaws before they exploitation. Automated vulnerability scans play a vital role in detecting known vulnerabilities and misconfigurations across the application and its underlying infrastructure. By integrating these practices into the development lifecycle, we can detect and address vulnerabilities early, reducing the likelihood of successful SQLI attacks. Furthermore, continuous security testing allows for the timely application of security patches and updates, ensuring that defenses are aligned with the latest threat intelligence.

**RESULTS DISCUSSION AND COMPARATIVE ANALYSIS**

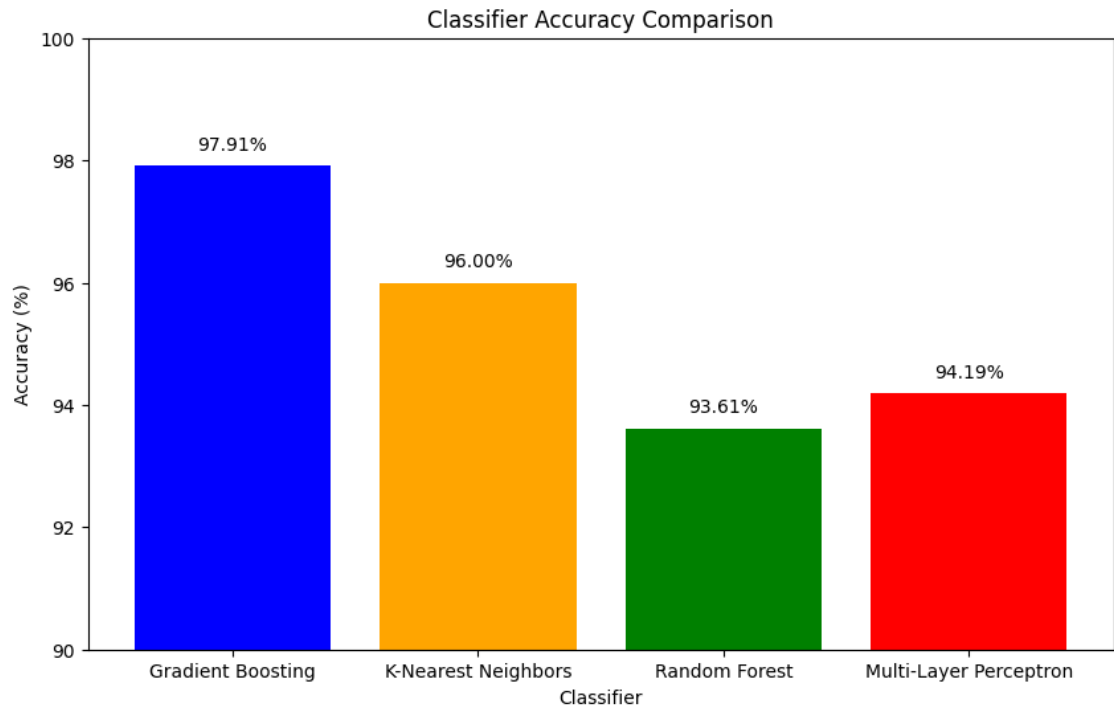
SQLI represents a critical vulnerability enabling attackers to manipulate database queries, potentially resulting in data breaches and unauthorized access. The experimental setup involved using Google Colab, leveraging its cloud-based environment with access to resources including an Intel Xeon CPU and approximately 12GB of RAM mainly. The resources provided the computational power necessary to conduct experiments online efficiently. The programming language used was Python 3, utilizing the Keras framework built on TensorFlow 2.0. Our approach centered on identifying SQLI vulnerability presence and classifying the SQLI queries into specific types accordingly. Initially, we extracted over 70 features, encompassing counts of boolean operators, unions, comments, and SQL keywords. The dataset was extensively cleaned and tokenized, followed by statistical feature extraction like G-test score and entropy to prepare for algorithmic analysis. This comprehensive feature set played a crucial role in accurately identifying and categorizing SQLI vulnerabilities. Ensuring consistency in data through thorough cleansing and detailed breakdown of queries via tokenization facilitated comprehensive detection of SQLI vulnerabilities. Our approach involved analyzing the dataset using a variety of ML and DL algorithms, where achieving high accuracy was paramount for evaluating performance. Each classifier underwent meticulous tuning with optimized parameters to maximize detection effectiveness. The following table shows the performance of different algorithms based on accuracy.

**Table. 18:** Results for SQLI Vulnerability Detection

Classifier	Dataset	Accuracy
GB		97.91%
K-Nearest Neighbors	SQLI Vulnerability Dataset	96.00%
RF		93.61%
MLP		94.19%

The table compares the accuracy of various classifiers applied to the SQLI Vulnerability Dataset. GB achieved the highest accuracy at 97.91%, followed by K-Nearest Neighbors with 96.00%. The MLP classifier also performed well, with an accuracy of 94.19%, while the RF classifier achieved 93.61%. These results demonstrate the strong performance of these machine

learning models in detecting SQL injection vulnerabilities, with GB emerging as the most effective method in this context. The results underscore the classifiers' varying strengths and weaknesses in differentiating between general vulnerability detection and specific vulnerability categorization tasks in SQLI detection. Here is graphical representation of our results for our dataset.



**Figure. 4:** Classifiers performance for SQLI Vulnerability Detection

The figure compares the accuracy of various classifiers applied to the SQLI Vulnerability Dataset. GB achieved the highest accuracy at 97.91%, followed by K-Nearest Neighbors with 96.00%. The MLP classifier also performed well, with an accuracy of 94.19%, while the RF classifier achieved 93.61%. These results demonstrate the strong performance of these ML,DL models in detecting SQL injection vulnerabilities, with GB emerging as the most effective method in this context. The best algorithm for SQLI Vulnerability detection in this context is GB, as it achieved the highest accuracy of 97.91%. GB is particularly effective because it combines multiple weak learners, typically decision trees, to create a strong predictive model. This method excels in handling complex patterns in data, which is crucial for detecting SQL injection vulnerabilities that may involve intricate and subtle variations in SQL queries. Additionally, GB's ability to minimize overfitting while improving model performance through iterative training contributes to its superior accuracy in identifying SQLI vulnerabilities. The performance matrix used here is accuracy.

**Accuracy** indicates the overall correctness of our detection and prevention methods. Here is its mathematical form.

$$Accuracy = \frac{TP + TN}{Total\ Population}$$

**False Positive (FP):** Instances that are incorrectly classified as positive (or true) when they are actually negative (or false).

**False Negative (FN):** Instances that are incorrectly classified as negative (or false) when they are actually positive (or true). For GB, FP and FN are 52, for KNN their value is 98, for RF their value is approximately 158 and for MLP, their value is 144.

**True Positive (TP):** Instances that are correctly classified as positive (or true) by the system.

**True Negative (TN):** Instances that are correctly classified as negative (or false) by the system.

These terms are essential for calculating metrics such as accuracy, precision, recall, and F1 score.

Accuracy metrics collectively ensure a robust evaluation framework, essential for validating the reliability and efficiency of our security measures in real-world applications. Detecting SQLI vulnerabilities in web applications is challenging due to their dynamic nature and the evolving tactics of malicious actors. It involves navigating complex codebases, countering advanced evasion techniques, and balancing thorough testing with resource limitations. Minimizing false positives and adapting methodologies to diverse web frameworks add further complexity. Effective strategies integrate automated tools, expert analysis, and ongoing vigilance to detect and respond to emerging threats promptly, enhancing overall web security.

These are the questions we need to answer after successfully conducting the experiment.

- Which algorithm performs best for detecting vulnerability?

We employed several algorithms including KNN, MLP, GB, and RF. Our results for SQLI vulnerability detection show that GB achieved the highest accuracy at 97.91%, followed by K-Nearest Neighbors with 96.00%. The MLP classifier also performed well, with an accuracy of 94.19%, while the RF classifier achieved 93.61%. GB outperformed other classifiers in detecting SQL injection vulnerabilities due to its ability to combine multiple weak learners into a strong, robust model. It effectively captured complex, nonlinear patterns in SQL queries, which are essential for identifying sophisticated attacks. Additionally, GB's built-in

mechanisms to reduce overfitting ensured that the model generalized well to new data, leading to its superior performance and accuracy in this context.

- What kind of vulnerability can be discovered?

In our thesis, the focus was specifically on SQLI vulnerabilities among various types such as Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) due to their prevalence and significant impact on web application security. SQLI vulnerabilities are particularly concerning as they allow attackers to manipulate database queries, potentially gaining unauthorized access to sensitive information or altering database contents. By addressing SQLI vulnerability, the aim is to enhance the overall security posture of web applications, mitigating risks associated with data breaches and unauthorized access attempts. We presented identification of SQLI vulnerabilities, and categorizing them based on type, and proposing preventive strategies to enhance system security and safeguard against such threats.

- How to prevent from vulnerability?

This thesis investigated SQLI vulnerabilities in web applications, highlighting their persistent threat to security. It proposes several preventive strategies: Firstly, employing parameterized queries with prepared statements isolates SQL code from user input, thus thwarting malicious SQL commands. Secondly, rigorous input validation techniques such as whitelisting, blacklisting, or regular expressions ensure user inputs conform to expected formats and are free of malicious content. Additionally, escaping special characters in user input neutralizes their significance in SQL queries, as demonstrated with Python's ``SQLite3`` library. The application of the Least Privilege Principle limits database and application component privileges to essential functions, reducing potential damage from SQLI exploits. Furthermore, deploying Web Application Firewalls (WAFs) to filter and block malicious HTTP requests at the application layer, coupled with continuous security testing through assessments and vulnerability scans, ensures proactive identification and remediation of SQLI vulnerabilities, collectively fortifying web application security.

- Which performance metrics are considered in this research?

In our thesis, we selected accuracy as the primary performance metric due to its straightforward interpretation and relevance in tasks like SQLI vulnerability detection. Accuracy provides a clear measure of the proportion of correctly classified instances out of the total instances evaluated, offering a holistic view of the model's predictive capability.

The next step is to do **comparative analysis** of SQLI vulnerability detection techniques with the study of SQLMap, our proposed technique, and (Mishra, 2019) approach. Firstly we discuss

the role of SQLMap. SQLMap is an open-source penetration testing tool that automates the detection and exploitation of SQLI vulnerabilities. It detects SQLI vulnerabilities by systematically injecting specially crafted SQL queries into vulnerable input fields. It starts by finding potential injection points through systematic analysis of web forms or HTTP headers. Once found, SQLMap employs various SQLI techniques such as boolean-based blind, time-based blind, error-based and union-based etc. By analyzing the responses from the application and observing deviations or errors in the DBMS responses, SQLMap infers the presence of vulnerabilities. It dynamically adjusts payloads and interprets these responses to determine the vulnerability type and severity in the targeted application's database backend. Following the execution of SQLI queries using SQLMap, we received the following responses.

1. Time-based: "SELECT 3109 FROM (SELECT(SLEEP(10)))"

The time-based SQLI query "SELECT 3109 FROM (SELECT(SLEEP(10)))" operates by embedding a delay command within the database query structure. Specifically, the inner subquery "SLEEP(10)" instructs the database server to pause execution for 10 seconds. SQLMap utilizes this query to probe for vulnerabilities in web applications by injecting it into input fields. If the application's response is delayed by approximately 10 seconds, it signifies that the injected query successfully paused the database execution, thereby confirming susceptibility to time-based SQLI.

2. Boolean-based: "AND 7627=7627"

The boolean-based SQLI query "AND 7627=7627" is designed to exploit the logic evaluation mechanism within SQL queries. In this case, the expression "7627=7627" is always true. SQLMap utilizes this query to test for boolean-based SQLI vulnerabilities by injecting it into vulnerable input fields of web applications. If the application's response remains unchanged or returns normally when this query is injected, it indicates that the injected condition ("7627=7627") evaluated as true within the SQL query context. This behavior suggests that the application did not properly sanitize or validate user inputs, allowing the injected SQL code to be executed as intended rather than being treated as mere data. Thus, the presence of a normal response to this query serves as a strong indicator of a boolean-based SQLI vulnerability, highlighting the potential for unauthorized data retrieval or manipulation through crafted SQL queries.

3. Error-based:

"GTID\_SUBSET(CONCAT(0x716a627171,(SELECT(ELT(4648=4648,1))),0x71706a7671),4648)"



It exploits the database's error handling mechanism to extract information. Here's a detailed analysis of its operation: Within the query, the function CONCAT is used to concatenate three hexadecimal values (0x716a627171, 0x71706a7671) along with a subquery (SELECT(ELT(4648=4648,1))). The subquery ELT(4648=4648, 1) is structured to always return the value at the first position (1). When injected by SQLMap into a vulnerable input field, if the database server interprets the query and attempts to execute it, an error will occur due to the GTID\_SUBSET function, which is not typically allowed in standard SQL syntax. The error message generated by the database server contains the concatenated string (0x716a6271711x71706a7671), thereby revealing crucial information about the database structure or even sensitive data. This behavior confirms the presence of an error-based SQLI vulnerability, as the application's response to the injected query exposes details that should not be disclosed under normal circumstances, highlighting potential risks of unauthorized access or data leakage.

4. Boolean-based: "AND 7230=7230&SELECT \* FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'"

The boolean-based SQLI query "AND 7230=7230&SELECT \* FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'" exploits the logical evaluation capabilities of SQL queries within vulnerable web applications. In this query, the initial statement "AND 7230=7230" is always true, as it simply compares the constant value 7230 to itself. The second part of the query "SELECT \* FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'" is structured to inject additional SQL commands after commenting out the rest of the original query with "--". Here, "1=1" is a universally true condition, ensuring that the injected OR condition "OR 1=1" is evaluated as true for all rows in the "users" table. This effectively bypasses any authentication logic, allowing unauthorized access to sensitive information like the password field of the "admin" user. The successful execution of this query, without triggering abnormal behavior or errors, indicates that the application is vulnerable to boolean-based SQLI, where crafted SQL statements can manipulate the application's logic to gain unauthorized access or extract sensitive data.

5.Error-based: "AND GTID\_SUBSET(CONCAT(0x7171766271,(SELECT (ELT(5085=5085,1))),0x7171627071),5085)& SELECT \* FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'"

In this query, the expression "GTID\_SUBSET(CONCAT(0x7171766271,(SELECT (ELT(5085=5085,1))),0x7171627071),5085)" attempts to generate an error by using the GTID\_SUBSET function, which is not supported or correctly executed in the context of a

standard SQL query. The subquery "(SELECT (ELT(5085=5085,1)))" is crafted to always return a valid result (1), ensuring the CONCAT function concatenates the specified hexadecimal strings and the result of the subquery. If the application's database is vulnerable, this malformed query structure triggers an error response from the database server, revealing the concatenated string (0x71717662711x7171627071) in the error message. This response confirms the existence of an error-based SQLI vulnerability, as the application's handling of the injected query exposes sensitive details about the database structure or data, highlighting potential risks of unauthorized access or information disclosure.

6. Time-based: "AND (SELECT 3886 FROM (SELECT(SLEEP(10)))DYIY)&SELECT \* FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'"

The time-based SQLI query "AND (SELECT 3886 FROM (SELECT(SLEEP(10)))DYIY)&SELECT \* FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'" exploits the delay in response from the database server to detect vulnerabilities in web applications. In this query, the subquery "(SELECT(SLEEP(10)))" instructs the database server to pause execution for 10 seconds before continuing. The outer query "AND (SELECT 3886 FROM ...)" is crafted to make the application wait for the specified time period by embedding the sleep function within the SELECT statement.

7. Union-based: "UNION ALL SELECT"

The union-based SQLI query "UNION ALL SELECT" exploits the SQL syntax feature of union operations to extract data from a vulnerable database through web applications. This query is structured to combine the results of two separate SELECT statements into a single result set. SQLMap utilizes this technique by injecting "UNION ALL SELECT" into input fields, typically appending it to an existing SQL query.

8. Boolean-based: "AND 7230=7230&SELECT \* FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'"

The boolean-based SQLI query "AND 7230=7230&SELECT \* FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'" leverages boolean logic to test for vulnerabilities in web applications. In this query, "AND 7230=7230" serves as a condition that is inherently true, as it compares the constant value 7230 to itself. The subsequent part of the query "SELECT \* FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'" injects additional SQL commands after commenting out the remainder of the original query with "--". Here, "1=1" is a universally true condition, ensuring that the injected OR condition "OR 1=1" evaluates as true for all rows in the "users" table. This effectively

bypasses any authentication logic, allowing unauthorized access to sensitive information like the password field of the "admin" user.

9. Time-based: "AND (SELECT 3886 FROM (SELECT(SLEEP(10)))DYIY)&SELECT \* FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'"

The time-based SQLI query "AND (SELECT 3886 FROM (SELECT(SLEEP(10)))DYIY)&SELECT \* FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'" employs time-delay techniques to detect vulnerabilities in web applications. In this query, the subquery "(SELECT(SLEEP(10)))" instructs the database server to pause execution for 10 seconds before continuing. The outer query "AND (SELECT 3886 FROM ...)" is structured to embed the sleep function within the SELECT statement, causing the application to wait for the specified duration.

10. Union-based:

"UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,CONCAT(0x7171766271,0x546964547a49577677786b4d65516b43447842706e664c637a43734b6a7a4979784f5854576a7a4b,0x7171627071),NULL-- -&SELECT \* FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'" query utilizes the UNION ALL operation to concatenate and extract specific data from a vulnerable database through web applications. In this query, "UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL" introduces placeholder columns that align with the number of columns in the target SELECT statement, ensuring compatibility with original query's structure.

"CONCAT(0x7171766271,0x546964547a49577677786b4d65516b43447842706e664c637a43734b a7a4979784f5854576a7a4b,0x7171627071)" concatenates hexadecimal values to form a string that SQLMap expects the database to return in the result set. The concatenation of hexadecimal values itself does not cause SQLI but constructs a specific string that is used within the injection payload. As in this query, the CONCAT function joins the hexadecimal values 0x7171766271, 0x546964547a49577677786b4d65516b43447842706e664c637a43734b6a7a4979784f5854576a7a4b, and 0x7171627071 by converting into the following code. qqvbqTidTzIWvwXkMeQkCDxBpnfLczCsKjzIyxOXTwzKqqbpq.

This payload is combined with a UNION operation to append the crafted string to the result of a legitimate query. When injected into a query like SELECT \* FROM users WHERE username = 'admin' OR 1=1 -- ', the database processes the UNION part, adding the concatenated string

to the output. This reveals the success of the SQLI, allowing the attacker to manipulate and extract data by appending arbitrary content to the query results.

SQLMap uses various techniques to test the security of web applications against different types of SQLI attacks. By analyzing the responses, it identifies and exploits potential vulnerabilities, helping to secure applications against SQLI threats.

We structured all SQLI queries with their corresponding attack types into an appropriate format using pandas' DataFrame. This organized framework allows us to efficiently conduct further experiments and analyses. By categorizing each query against its respective attack type: whether Time-based, Boolean-based, Error-based, or Union-based etc. The following dataset is going to be used for comparative analysis of the performances of two techniques.

**TABLE. 19:** Comparative Analysis Dataset

Query	Types
SELECT 3109 FROM (SELECT(SLEEP(10)))	Time-based
AND 7627=7627	Boolean-based
GTID_SUBSET(CONCAT(0x716a627171,(SELECT(ELT(4648=4648,1))),0x71706a7671),4648)	Error-based
AND 7230=7230&SELECT * FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'	Boolean-based
AND GTID_SUBSET(CONCAT(0x7171766271,(SELECT ECT (ELT(5085=5085,1))),0x7171627071),5085)&SE LECT * FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'	Error-based
AND (SELECT 3886 FROM (SELECT(SLEEP(10)))DYIY)&SELECT * FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'	Time-based

UNION ALL SELECT	Union-based
AND 7230=7230&SELECT * FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'	Boolean-based
AND (SELECT 3886 FROM (SELECT(SLEEP(10)))DYIY)&SELECT * FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'	Time-based
UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,NULL,NUL L,NULL,NULL,CONCAT(0x7171766271,0x546 964547a49577677786b4d65516b43447842706e6 64c637a43734b6a7a4979784f5854576a7a4b,0x7 171627071),NULL-- -&SELECT * FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'	Union-based

This table presents a comparative analysis- dataset of various SQLI attack detected by SQLMap, categorized based on types according to the query structures. Each row represents a different SQLI attack method, such as Time-based, Boolean-based, Error-based, Union-based, and combinations thereof. The queries demonstrate different techniques used by attackers to exploit vulnerabilities in web applications, aiming to manipulate database queries for unauthorized access or data extraction. This table serves to illustrate the diversity and complexity of SQLI attacks that can be encountered, highlighting the importance of robust security measures and vigilant monitoring to mitigate such vulnerabilities effectively.

Now, we discuss the technique of **baseline paper technique** (Mishra, 2019) for SQLI vulnerability. We have a list of SQLI queries identified by SQLMap, as shown in Table 21, which serves as the dataset for experiments. This dataset includes various examples of SQLI attacks, providing a comprehensive base for analysis (Mishra, 2019). Firstly, tokenization of SQLI performed using regular expressions to break down the SQL code into meaningful tokens. The `re.compile()` method in Python used to compile these regular expressions, facilitating efficient tokenization. The regular expression object created using multiple SQL queries and SQL reserved words. Tokenization implemented by lexical analysis using regular expressions in Python. Then `groupby()` method used to split the objects into tokens. After creating tokens

from the dataset, statistical feature extraction performed using these tokens (Mishra, 2019). The token object has three parameters: Token\_Count, which stores the number of times a particular token is present in the entire dataset; Token\_Value, which stored the actual values of tokens that are created; and Token\_Type, which showed the type of the specific token. Tokens were also grouped together using the groupby() function. Feature extraction involved calculating the G-test scores for all token values in the dataset. Before calculating the G-test scores, a new dataframe was created using the Pandas library in Python. This dataframe acted as the new dataset and having columns as Token\_Count and Token\_Value. Then G-test score calculation performed on new dataset, which was crucial for assessing the significance of each token in the context of SQLI detection. Calculating the G-test score and entropy is important for understanding the randomness and distribution of the data. To calculate the G-test scores, some pre-processing needed to be done on the data, such as converting numerical values for counts to float type. Two types of G-test scores were calculated: Observed G-test Score and Expected G-test Score. The Expected G-test score was calculated based on the total tokens, the number of tokens in a particular row, and the types of tokens, representing the ideal score if the data were normally distributed. The Observed G-test score is the actual score of the occurrence of data (Mishra, 2019). The next step was calculating the entropy of each row in the dataset, which measured the randomness of the data. Low entropy indicated similar data, while high entropy indicates diverse data (Mishra, 2019). The average value of G-test scores was calculated for each token in the dataset, and a new dataset was generated and stored in a dataframe using Pandas in Python. This dataset was used for training the model. Firstly, it involved calculating prior probabilities: the total number of rows in the dataset, the number of SQLI rows. Prior probabilities were calculated based on these counts. The next step involved calculating the likelihood of a new input being a SQLI, based on the number of tokens matching the new input (Mishra, 2019). After this, the GB algorithm was applied to classify SQLI vulnerabilities. The GB Classifier is from the ensemble part of the Scikit-Learn library in Python. Parameter tuning is crucial for optimizing ensemble learning algorithms. The parameters were tuned include n\_estimators, which decided the number of boosting stages and set to the default value of 100; learning\_rate, which determined the contribution of each tree and set to the default value of 0.1; max\_depth, which decided the maximum depth of individual trees and set to 2 for optimal performance; and random\_state, which controls the random number generation for trees. By carefully tuning these parameters, the GB Classifier effectively detected SQLI vulnerabilities, as demonstrated in (Mishra, 2019) technique.

Tokenization of SQLI plays a pivotal role in cybersecurity by transforming SQL code into manageable tokens using regular expressions. This process was crucial for identifying and analyzing potential vulnerabilities such as SQLI attacks.

The initial step involved compiling regular expressions using ``re.compile()`` in Python, optimizing them for efficient pattern matching within SQL queries.

```
(?P<UNION>UNION\s+(ALL\s+)?SELECT)|(?P<PREFIX>([\`\`\"\\\])|((\`\`|\"\\)|\d+|\w+)\s))(\|\|\&\&|and|or)
```

**Figure.5:** Regular Expression for Tokenization

In this regular expression, ‘\w’ shows alphanumeric character. ‘\s’ in regular expressions matches any type of whitespace character, encompassing spaces, tabs, line breaks, and other Unicode-defined whitespace characters. ‘\d+’ matches one or more digits occurring consecutively in a sequence. ‘<PREFIX>’ functions as a named capturing group identifier in regular expressions, enabling the definition of a group that captures particular patterns from the matched text. ‘|’ allows to choose between different patterns for matching. ‘\`’ ensures special characters are matched literally instead of their special meanings and ‘?’ adjusts the preceding element to match either zero or one occurrence.

Tokenization was proceeded through lexical analysis, leveraging compiled regular expressions to extract tokens from SQL code. Each token represented significant elements essential for understanding query structure and potential security risks. The dataset size bit changed due to the tokenization process of Mishra's Technique which records each token from the queries as a separate entry, capturing detailed information about its count and type. This level of detail results in more entries per query, thus expanding the dataset size. Moreover, the reason is: each entry in the dataset represented a unique SQL token extracted. Each row corresponded to a single token, and the associated details pertain to that specific token.

Following extraction, tokens were grouped using the ``groupby()`` method to categorize them by type or other criteria. This grouping facilitated an organized analysis and enhanced the effectiveness of subsequent feature extraction and vulnerability detection processes. The complete output table was very long due to all tokens of whole queries. So in order to manage the extensive content, only selected rows are displayed below to illustrate the output format.

**Table. 20:** Grouping Method

Token Type: Union-based		
Token: UNION	Count: 3	Value: ['UNION']

Token: ALL	Count: 3	Value: ['ALL']
Token: NULL	Count: 20	Value: ['NULL']

The table outlined the grouping method for tokens associated with Union-based SQLI attacks. Each token type, including 'UNION', 'ALL', and 'NULL', was categorized by its count and respective values extracted from SQL queries. These tokens were critical in identifying and categorizing Union-based SQLI vulnerabilities, where attackers exploit the 'UNION' SQL operator to combine results from multiple SELECT statements and 'NULL' values to bypass authentication or inject malicious code. The table provided a structured view of token frequencies and values, aiding in the detection and mitigation of specific SQLI vulnerabilities through comprehensive analysis and targeted security measures. Token object parameters were: Token\_Count that quantified the frequency of each token across the dataset, aiding in identifying frequently used SQL constructs, Token\_Value that stored the exact content of each token, providing insights into specific SQL components and their context within queries and Token\_Type that classified tokens by type (e.g., keywords, operators), aiding in semantic analysis and query classification. The following row shows the complete format of output table.

**Table.21:** Tokenization

Token_Count: 3	Token_Type: Time-based	Token_Value: ['SLEEP']
----------------	------------------------	------------------------

This table illustrated the tokenization process used in the experiments, specifically focusing on tokens related to Time-based SQLI attacks, notably the 'SLEEP' function. It showed that the token 'SLEEP' occurs three times in the dataset, emphasizing its frequency and importance in detecting Time-based SQLI vulnerabilities. Tokens such as 'SLEEP' played a critical role in identifying SQLI attack patterns where adversaries exploited time delay functions to manipulate query execution times, potentially gaining unauthorized access or disrupting application operations.

After tokenization, the feature extraction process began. Feature extraction involved calculating the G-test Scores for all token values in the dataset. Before calculating the G-test scores, firstly, new dataframe was created using the Pandas library in Python. This dataframe served as the foundation for feature extraction and included columns for Token\_Count and Token\_Value, derived from the tokens identified during the tokenization phase as shown below. This structured representation allowed for systematic handling and analysis of token-related data.

**Table.22:** Showing structure of new dataset



Token_Count	Token_Value
17	SELECT
9	FROM
4	CONCAT
2	0x716a627171
14	1
2	0x71706a7671
6	users
6	WHERE
6	username
6	admin
6	OR
6	union
12	password
2	0x7171766271
2	0x7171627071
3	UNION
3	ALL
20	NULL
1	0x546964547a49577677786b4d65516b4344 7842706e664c637a43734b6a7a4979784f585 4576a7a4b

---

	0x4e554275474250636557664f41515862657
1	66867415674465048735a506b6244474c506 665746f66
1	3109
3	SLEEP
3	10
6	AND
2	3886
2	DYIY
2	7627
4	7230
2	GTID_SUBSET
2	ELT
3	4648
3	5085

---

This table provided an overview of the structure of a new dataset used further, detailing the frequency of various tokens extracted from SQL queries. The table categorized tokens such as SQL keywords ('SELECT', 'FROM', 'WHERE'), specific values ('admin', 'password'), special characters ('NULL', hexadecimal values), and functions ('CONCAT', 'SLEEP'). These tokens were crucial for identifying and analyzing SQLI vulnerabilities, where their presence or frequency indicated potential security risks. The dataset structure revealed a diverse range of SQL elements manipulated by attackers, highlighting the importance of comprehensive data preparation and analysis techniques to effectively detect and mitigate SQLI threats in web applications.

After creating the dataframe with columns for Token\_Count and Token\_Value, derived from the tokenized dataset, the next crucial step was the calculation of G-test Scores (Mishra, 2019). The G-test Score, also known as the likelihood ratio, is a statistical measure used to evaluate

the association between categorical variables, such as tokens within SQL queries. This process involved several key aspects.

Prior to computing the G-test Scores, numerical counts in the dataset were firstly converted to floating-point format, as illustrated in a few following rows. This ensured that the calculations were performed accurately with precise numerical representation.

**Table.23:** Converting Token\_Count to float

Token_Count	Token_Value
17.0	[SELECT]
9.0	[FROM]
4.0	[CONCAT]
2.0	[0x716a627171]
14.0	[1]
2.0	[0x71706a7671]
6.0	[users]
6.0	[WHERE]
6.0	[username]
6.0	[admin]
6.0	[OR]
6.0	[union]
12.0	[password]
2.0	[0x7171766271]
2.0	[0x7171627071]
3.0	[UNION]

---

3.0	[ALL]
20.0	[NULL]
1.0	[0x546964547a49577677786b4d65516b43447842706e664c637a43734b6a7a4979784f5854576a7a4b]
1.0	[0x4e554275474250636557664f4151586265766867415674465048735a506b6244474c506665746f66]
1.0	[3109]
3.0	[SLEEP]
3.0	[10]
6.0	[AND]
2.0	3886
2.0	[DYIY]
2.0	[7627]
4.0	[7230]
2.0	[GTID_SUBSET]
2.0	[ELT]
3.0	[4648]
3.0	[5085]

---

The table presented a structured representation of token frequencies converted into float values from a dataset used in the study. Each row categorized specific tokens extracted from SQL queries, including SQL keywords ('SELECT', 'FROM', 'WHERE'), values ('admin', 'password'), special characters ('NULL', hexadecimal values), and functions ('CONCAT', 'SLEEP'). The conversion to float format aided in numerical analysis and statistical processing, facilitating detailed examination of token distribution and frequency patterns essential for detecting SQLI

vulnerabilities. This table underscored the importance of data preprocessing techniques like tokenization and normalization to enhance the accuracy and effectiveness of ML models in identifying and mitigating SQLI threats in web applications. Two types of G-test Scores were computed during this process: Expected G-test Score that score was calculated based on the total tokens, the number of tokens in a particular row, and their types. The Expected G-test Score represents the theoretical or ideal score expected if the distribution of tokens followed a specified pattern, such as normal distribution. Observed G-test Score that reflected the actual score derived from the occurrence of data in the dataset. It measured how closely the observed distribution of tokens matched the expected distribution based on the calculated probabilities. The values of observed and expected g-test score are following:

**Table.24:** Values of Observed and Expected G-test score

<b>Token_ Count</b>	<b>Token_Value</b>	<b>Observed_G_Test_ Score</b>	<b>Expected_G_Test_ Score</b>
17.0	[SELECT]	41.396526	-12.251545
9.0	[FROM]	10.46801	-5.851908
4.0	[CONCAT]	-1.834993	2.308077
2.0	[0x716a627171]	-3.690085	9.28287
14.0	[1]	28.654888	-10.29785
2.0	[0x71706a7671]	-3.690085	9.28287
6.0	[users]	2.113092	-1.771916
6.0	[WHERE]	2.113092	-1.771916
6.0	[username]	2.113092	-1.771916
6.0	[admin]	2.113092	-1.771916
6.0	[OR]	2.113092	-1.771916
6.0	[union]	2.113092	-1.771916

---

12.0	[password]	20.861717	-8.746709
2.0	[0x7171766271]	-3.690085	9.28287
2.0	[0x7171627071]	-3.690085	9.28287
3.0	[UNION]	-3.102337	5.202878
3.0	[ALL]	-3.102337	5.202878
20.0	[NULL]	55.202552	-13.886892
1.0	[0x546964547a49577677	-3.231337	16.257664
	786b4d65516b434478427		
	06e664c637a43734b6a7a		
	4979784f5854576a7a4b]		
1.0	[0x4e5542754742506365	-3.231337	16.257664
	57664f415158626576686		
	7415674465048735a506b		
	6244474c506665746f66]		
1.0	[3109]	-3.231337	16.257664
3.0	[SLEEP]	-3.102337	5.202878
3.0	[10]	-3.102337	5.202878
6.0	[AND]	2.113092	-1.771916
2.0	[3886]	-3.690085	9.28287
2.0	[DYIY]	-3.690085	9.28287
2.0	[7627]	-3.690085	9.28287
4.0	[7230]	-1.834993	2.308077
2.0	[GTID_SUBSET]	-3.690085	9.28287
2.0	[ELT]	-3.690085	9.28287

---

3.0	[4648]	-3.102337	5.202878
3.0	[5085]	-3.102337	5.202878

This table presented values of observed and expected G-test scores for various tokens extracted from SQL queries in the dataset. The G-test score was a statistical measure used to assess the association between observed and expected frequencies of categorical data, in this case, tokens related to SQLI vulnerabilities. Positive G-test scores indicated tokens that appeared more frequently than expected, while negative scores indicated tokens that appeared less frequently. For instance, tokens like 'SELECT', 'FROM', and 'NULL' have positive G-test scores, suggesting they were more prevalent in SQL queries and are indicative of common SQLI attack patterns. Conversely, tokens such as hexadecimal values and specific function names showed negative G-test scores, indicating their occurrence is less frequent than expected. This analysis helps in identifying key tokens used in SQLI attacks, aiding in the development of effective detection and mitigation strategies for SQLI vulnerabilities in web applications. The calculation of G-test Scores was crucial (Mishra, 2019) for assessing the significance of each token in distinguishing between SQLI and legitimate queries. Tokens with higher G-test Scores indicated a stronger association with SQLI patterns, thereby aiding in the accurate identification of security vulnerabilities. The positive G-test score indicated that the observed frequency was greater than the expected frequency under the null hypothesis. The negative G-test score indicated that the observed frequency was less than the expected frequency under the null hypothesis.

Entropy is a crucial metric used in feature extraction to quantify the randomness or uncertainty within a dataset. It measured the diversity or uniformity of data distribution: lower entropy indicated more predictability or similarity among data points, while higher entropy suggested greater diversity or randomness. The formula for entropy is expressed as follows:

$$Entropy = -\sum p(X) \log p(X)$$

Where  $p(X)$  represents the proportion of occurrences of  $(X)$  relative to the total number of features. Here is the updated table with entropy.

**Table.25:** Values of Observed, Expected G-test score and Entropy

Token_ Count	Token_ Value	Observed_G_Test_ Score	Expected_G_Test_ Score	Entropy
-----------------	-----------------	---------------------------	---------------------------	---------

---

17.0	[SELECT]	41.396526	-12.251545	0.237387
9.0	[FROM]	10.46801	-5.851908	0.161227
4.0	[CONCAT]	-1.834993	2.308077	0.091804
2.0	[0x716a627171]	-3.690085	9.28287	0.054513
14.0	[1]	28.654888	-10.29785	0.212378
2.0	[0x71706a7671]	-3.690085	9.28287	0.054513
6.0	[users]	2.113092	-1.771916	0.122595
6.0	[WHERE]	2.113092	-1.771916	0.122595
6.0	[username]	2.113092	-1.771916	0.122595
6.0	[admin]	2.113092	-1.771916	0.122595
6.0	[OR]	2.113092	-1.771916	0.122595
6.0	[union]	2.113092	-1.771916	0.122595
12.0	[password]	20.861717	-8.746709	0.193528
2.0	[0x7171766271]	-3.690085	9.28287	0.054513
2.0	[0x7171627071]	-3.690085	9.28287	0.054513
3.0	[UNION]	-3.102337	5.202878	0.074214
3.0	[ALL]	-3.102337	5.202878	0.074214
20.0	[NULL]	55.202552	-13.886892	0.25909
1.0	[0x546964547a495776777]	-3.231337	16.257664	0.031562

---



---

	86b4d65516			
	b434478427			
	06e664c637a			
	43734b6a7a4			
	979784f5854			
	576a7a4b]			
	[0x4e554275			
	4742506365			
	57664f41515			
1.0	8626576686	-3.231337	16.257664	0.031562
	7415674465			
	048735a506b			
	6244474c506			
	665746f66]			
1.0	[3109]	-3.231337	16.257664	0.031562
3.0	[SLEEP]	-3.102337	5.202878	0.074214
3.0	[10]	-3.102337	5.202878	0.074214
6.0	[AND]	2.113092	-1.771916	0.122595
2.0	[3886]	-3.690085	9.28287	0.054513
2.0	[DYIY]	-3.690085	9.28287	0.054513
2.0	[7627]	-3.690085	9.28287	0.054513
4.0	[7230]	-1.834993	2.308077	0.091804
2.0	[GTID_SUB SET]	-3.690085	9.28287	0.054513
2.0	[ELT]	-3.690085	9.28287	0.054513
3.0	[4648]	-3.102337	5.202878	0.074214
3.0	[5085]	-3.102337	5.202878	0.074214

---

This table presented token values extracted from SQL queries, along with their corresponding G-test score and entropy values. Entropy quantified the diversity and unpredictability of tokens within the dataset. Higher entropy values indicated a broader variety of tokens present in the dataset, suggesting greater complexity and variability in SQL query structures. For instance, tokens like 'SELECT', 'FROM', and 'NULL' exhibited lower entropy values, indicating their frequent occurrence and predictable usage in SQL statements. In contrast, tokens with higher entropy, such as hexadecimal values and specific function names like 'SLEEP' and 'GTID\_SUBSET', appeared less frequently and contribute to the dataset's overall diversity. Analyzing entropy helped in understanding the distribution and composition of tokens, crucial for developing effective strategies to detect and mitigate SQLI vulnerabilities in web applications.

Next, a new dataset was generated and stored in a DataFrame using the pandas library in Python. This new dataset was then used for training the model where each unique token extracted during tokenization results in a separate entry in the dataset, leading to multiple rows for a specific query. Each row corresponds to a different token from query, and each token had its own `entropy` and `SQLI\_g\_means` values recorded in the dataset. The sum of Token\_Count used in the experiments for calculating g-test score and training dataset of Mishra's only kept the columns: raw\_sql, type, sql\_token, token\_seq, token\_length, entropy and sqli\_g\_means as shown below. Here is a glimpse of a few rows of new dataset for training.

**Table.26:** A preview of a few rows from training dataset

raw_sql	type	sql_tokens	token_seq	token_ length	entropy	sqli_g_means
AND 7230=7230& SELECT * FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'	Boolean -based	['7230']	[('7230',)]	1	0.09180 4	1.77636E-15

---

GTID_SUBS						
ET(CONCA						
T(0x716a627						
171,(SELEC	Error-	['GTID_SU	['GTID_S	1	0.05451	8.88178E-16
T(ELT(4648	based	BSET']	UBSET',)]		3	
=4648,1))),0						
x71706a7671						
),4648)						

---

The table provided a preview of a few rows from a training dataset used for SQLI vulnerability detection. Each row included the raw SQL query, its classification type (such as Boolean-based or Error-based), tokenized SQL tokens, the sequence of tokens, token length, entropy value, and SQLI G-means score. These attributes were crucial for training machine learning models to identify and categorize different types of SQLI attacks. For example, the SQL query "AND 7230=7230&SELECT \* FROM users WHERE username = 'admin' OR 1=1 -- 'union password = 'password'" was classified as Boolean-based, with tokens like '7230' and 'SELECT' contributing to its token sequence and entropy calculation. The entropy value reflected the diversity and complexity of tokens within the query, aiding in understanding the query's structural characteristics for effective vulnerability detection.

In the training dataset, the columns `sql\_tokens` and `token\_seq` served different roles in representing the tokenization process applied to each SQL query (`raw\_sql`). The `sql\_tokens` column listed individual tokens extracted from the query, encapsulating significant keywords or elements such as `[SELECT]`, `[FROM]`, or `[CONCAT]`. Each entry in `sql\_tokens` reflected distinct tokens identified within the SQL query. On the other hand, the `token\_seq` column captured the sequence of these tokens as they appear in the original SQL query, preserving their order in tuple format (`[(SELECT,)]`, `[(FROM,)]`, `[(CONCAT,)]`). This format ensured that the relational context and sequence of tokens within each query were maintained, providing a detailed perspective on how the query is structured and executed. As shown in tables, `sql\_tokens` and `token\_seq` both columns contain similar values but utilized different structures: lists of strings (containing token extracted from query) vs. lists of tuples (makes a sequences for tokens). Both `sql\_tokens` and `token\_seq` also had prediction power as the `sql\_tokens` column, with its list of strings, helped detecting SQLI by analyzing token frequency (e.g., frequent 'UNION' or 'SELECT') indicating potential attacks. The `token\_seq`, with token-position tuples, enhanced detection by pinpointing suspicious sequences (like 'OR' conditions) or unusual token placements that suggested SQLI patterns, improving predictive

accuracy for vulnerability detection. That's why these columns were also included in the training dataset as per Mishra's technique.

Next, a GB Classifier was chosen to detect SQLIs within the dataset. This classifier was implemented from the ensemble module of the Scikit-Learn library in Python. Parameter tuning played a crucial role in optimizing ensemble learning algorithms. Ensemble learning approached through two methods discussed below: Bagging: that involves employing multiple supervised learning models to predict data values. It combined the outputs of these models using a specified technique. Boosting: utilized multiple supervised learning models. It combined the models to enhance predictive accuracy progressively.

The parameter tuning process for optimizing the GB model for detecting SQLI vulnerabilities involved adjusting several key parameters. The number of boosting stages (`n_estimators`) was kept at the default of 100 to maintain robust model performance. The `learning_rate` remained at 0.1 to balance the contribution of each tree in the ensemble. `max_depth` was reduced from 3 to 2 to improve performance by preventing overfitting and simplifying tree structures. `random_state` was set to 0 to ensure consistent results across different runs, facilitating reliable model evaluation. These adjustments aimed to enhance the model's accuracy in identifying SQLI vulnerabilities effectively across various datasets.

Before applying machine learning algorithm, the features and labels were first clearly defined. The feature set contains: raw\_sql,sql\_tokens,token\_seq, token\_length, entropy, sql\_i\_g\_means and label set includes: type. The numeric and non-numeric features are separated to facilitate appropriate processing steps, as numeric features were used by most algorithms while non-numeric features often required conversion. Categorical features were then encoded into numeric form using one-hot encoding. This was done to transform categorical data into a format that can be provided to ML algorithm to do a better job in prediction. One-hot encoding prevented the algorithm from assuming any ordinal relationship between categorical values, which led to incorrect predictions. The dataset was split into training and testing sets following the 80-20 rule to provide a robust basis for model validation. Subsequently, parameters for the chosen machine learning algorithms were tuned carefully to optimize model performance, ensuring the models were well-calibrated and ready for evaluation. The accuracy of model was this:

**Table. 27:** Results for GB

GB Classifier Performance :	81.77 % Accuracy
-----------------------------	------------------

This table summarized the performance of the GB classifier with an accuracy of 71.43%. This metric indicated the proportion of correctly classified instances out of the total instances evaluated, showcasing the model's effectiveness in identifying SQLi vulnerabilities.

Now we **explain our proposed technique** for SQLi vulnerability detection. Our technique addresses the issue by first detecting whether an SQL query is vulnerable to SQLi attacks. If a vulnerability is detected, it further classifies the type of SQLi attack present and presenting prevention techniques in order to keep our system safe. As all the queries shown in Table 1 are vulnerable SQLi queries along-with their types. Now we'll apply our workflow that how our system detects SQLi vulnerability. First of all, features are extracted from the queries, such as the number of boolean, unions, comments, SQL keywords, and other relevant features etc. Additionally, it performs data cleaning and tokenization, as well as statistical feature extraction like G-test score and entropy. These steps ensure that the data is prepared comprehensively before applying an algorithms. To ensure a fair comparison, we are conducting a comparative analysis with Mishra's technique using the same dataset of SQLi queries identified by SQLmap, as shown in Table 1. This dataset provides a standardized benchmark, allowing us to evaluate the effectiveness and accuracy of our approach for detecting types of SQLi vulnerabilities against an established method.

Feature extraction involves systematically analyzing each SQL query to identify and quantify various characteristics indicative of SQLI vulnerabilities. Using techniques such as regular expressions, this process detects SQL keywords, special characters, comment patterns, logical operators, and more, transforming raw SQL queries into a structured format with meaningful attributes. Extracting features from our dataset is crucial for accurate detection and classification of SQLI vulnerabilities. By extracting these features, we assess the vulnerability of SQL queries to malicious attacks, enabling proactive security measures and enhancing the robustness of database systems. We extracted over 70 features as shown in Table 07 including 'length', 'num\_keywords', 'num\_special\_chars', 'num\_comments', 'num\_logical\_operators', 'union\_based', 'time\_based', 'tautology\_based', 'num\_conditions', 'has\_subquery', 'has\_select', 'has\_union', 'has\_insert', 'has\_update', 'has\_delete', 'has\_drop', 'has\_alter', 'has\_create', 'has\_exec', 'has\_grant', 'has\_revoke', 'has\_truncate', 'num\_', 'num\_', 'num\_', 'num\_', 'num\_#', 'num\_/', 'num\_\*/', 'num\_= ', 'num\_<', 'num\_>', 'num\_!', 'num\_|', 'num\_&', 'has\_concat', 'has\_substr', 'has\_substring', 'has\_left', 'has\_right', 'has\_mid', 'has\_char', 'has\_ascii', 'has\_hex', 'has\_unhex', 'has\_md5', 'has\_sha1', 'has\_sha256', 'has\_pattern\_select.\*\_from', 'has\_pattern\_union.\*\_select', 'has\_pattern\_select.\*\_where', 'has\_pattern\_select.\*\_order\_by', 'has\_pattern\_select.\*\_group\_by', 'num\_numbers', 'num\_strings', 'num\_booleans', 'num\_nulls', 'num\_comments\_inline', 'num\_comments\_multiline', 'num\_unions', 'num\_selects', 'num\_and's'.

'num\_ors', 'num\_equals', 'num\_semicolons', 'num\_parentheses', 'has\_order\_by', 'has\_group\_by', 'has\_limit', 'has\_offset', 'has\_into', 'has\_load\_file', 'num\_hex'.

Next step is cleansing. The cleansing process applied to the 'Query' text data serves to prepare the SQL queries for robust vulnerability detection, particularly for SQLI. By initially handling missing values and subsequently applying text cleaning techniques, such as removing extra spaces and converting text to lowercase, we ensure that the data is normalized and uniform. Removing extra spaces simplifies query structures, making it easier to detect irregularities or injected SQL commands. Moreover, as the next step is tokenization so, not removing extra spaces from SQL queries before tokenization can lead to several issues. Tokenization inaccurately split or merge tokens, which can misrepresent the query structure. This compromises query parsing and subsequent vulnerability detection analysis by obscuring patterns and making it harder to detect SQLI vulnerabilities. Ultimately, this can reduce the accuracy of vulnerability detection as the tokens extracted may not faithfully represent the intended query components or structure, impacting the reliability of security measures implemented. Besides, converting text to lowercase standardizes keyword recognition, crucial as SQL commands are case-insensitive in databases. These steps not only streamline subsequent analysis but also enhance the accuracy of identifying SQLI vulnerabilities. Here is a table showing before and after cleansing.

**Table.28:** Overview of Queries Before and After Cleaning

Before Cleaning	After Cleaning
SELECT 3109 FROM (SELECT(SLEEP(10)))	select 3109 from (select(sleep(10)))
AND 7627=7627	and 7627=7627
GTID_SUBSET(CONCAT(0x716a627171,(SELECT(ELT(4648=4648,1))),0x71706a7671),4648)	gtid_subset(concat(0x716a627171,(select(elt(4648=4648,1))),0x71706a7671),4648)
AND 7230=7230&SELECT * FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'	and 7230=7230&select * from users where username = 'admin' or 1=1 -- ' union password = 'password'

AND GTID_SUBSET(CONCAT(0x7171766271,(SELECT ELT(5085=5085,1))),0x7171627071),5085)&SELECT * FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'	and gtid_subset(concat(0x7171766271,(select t (elt(5085=5085,1))),0x7171627071),5085)&select * from users where username = 'admin' or 1=1 -- ' union password = 'password'
AND (SELECT 3886 FROM (SELECT(SLEEP(10)))DYIY)&SELECT * FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'	and (select 3886 from (select(sleep(10)))dyiy)&select * from users where username = 'admin' or 1=1 -- ' union password = 'password'
UNION ALL SELECT	union all select
AND 7230=7230&SELECT * FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'	and 7230=7230&select * from users where username = 'admin' or 1=1 -- ' union password = 'password'
AND (SELECT 3886 FROM (SELECT(SLEEP(10)))DYIY)&SELECT * FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'	and (select 3886 from (select(sleep(10)))dyiy)&select * from users where username = 'admin' or 1=1 -- ' union password = 'password'
UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,CONCAT(0x7171766271,0x546964547a49577677786b4d65516b43447842706e664c637a43734b6a7a4979784f5854576a7a4b,0x7171627071),NULL-- -&SELECT * FROM users WHERE username = 'admin' OR 1=1 -- ' union password = 'password'	union all select null,null,null,null,null,null,null,null,null,concat(0x7171766271,0x546964547a49577677786b4d65516b43447842706e664c637a43734b6a7a4979784f5854576a7a4b,0x7171627071),null-- -&select * from users where username = 'admin' or 1=1 -- ' union password = 'password'

This table illustrates the cleaning process applied to SQL queries for SQLI vulnerability detection. The "Before Cleaning" column shows raw SQL queries containing potential SQLI vulnerabilities, while the "After Cleaning" column displays these queries after by converting them into lowercase, removing unnecessary spaces, and standardizing SQL syntax.

The next step is tokenization, a process of breaking down text into smaller units called tokens, which can be words, phrases, or symbols. Tokenization is crucial for natural language processing tasks as it helps in understanding and analyzing the structure and meaning of the text. In the context of SQLI vulnerability detection, tokenization is particularly important because it enables a detailed analysis of SQL queries by identifying and categorizing different elements within the query. Tokenization is performed on SQL queries using the NLTK library's `word\_tokenize` function, which splits each query into individual tokens. These tokens are then classified into types such as numbers, punctuation, operators, SQL keywords, and others using regular expressions. This process is applied to the 'SQL\_Query' column in the `SQLI\_data` DataFrame, storing results for tokenization into new columns for tokens, token types, and token counts. By extracting token values, types, and counts during the tokenization process, we gain a comprehensive understanding of each query's structure. The importance of token count in SQLI vulnerability detection lies in its ability to reveal the complexity and structure of a query. A high token count may indicate a more complex query that could potentially include nested or concatenated statements, often characteristic of SQLI attempts. By extracting token count, we can identify unusually long or complex queries that warrant further inspection. Extracting token types helps in distinguishing between different components of a query, such as SQL keywords, numbers, and operators. This classification allows for the identification of patterns that are commonly associated with SQLI attacks. For example, repeated use of certain operators or keywords in a specific sequence may signal an attempt to manipulate the query. Moreover, the process of tokenization, including the extraction of token types and counts, provides a structured way to analyze SQL queries. This structured analysis is essential for detecting and classifying SQLI vulnerabilities, as it helps in identifying anomalies and patterns that are indicative of malicious activities. By breaking down and categorizing query components, tokenization enhances the ability to detect and mitigate potential security threats.

**Table.29:** A look on Tokenization Process

SQL_Query	Tokens	Token_Types	Token_Count
select 3109 from (select(sleep(10)))	[select, 3109, from, (, select, (, sleep, (, 10, ), ), )]	[keyword, number, keyword, other, keyword, other, other, other, number, other, other, other]	12
and 7627=7627	[and, 7627=7627]	[keyword, other]	2



gtid_subset(concat(0x716a627171,(select(elt(4648=4648,1))),0x71706a7671),4648)	[gtid_subset, (, concat, (, 0x716a627171, ,, (, select, (, elt, (, 4648=4648,1, ), ), ), ,0x71706a7671, ), ,4648, )]	[other, other, other, other, other, punctuation, other, keyword, other, other, other, other, other, other, other, other]	19
and 7230=7230&select * from users where username = 'admin' or 1=1 -- ' union password = 'password'	[and, 7230=7230, &, select, *, from, users, where, username, =, 'admin, ', or, 1=1, -, ', union, password, =, 'password']	[keyword, other, other, keyword, operator, keyword, other, keyword, other, operator, other, other, keyword, other, other, other, other, other, operator, other, other]	21
and gtid_subset(concat(0x7171766271,(select(elt(5085=5085,1))),0x7171627071),5085)&select * from users where username = 'admin' or 1=1 -- ' union password = 'password'	[and, gtid_subset, (, concat, (, 0x7171766271, ,, (, select, (, elt, (, 5085=5085,1, ), ), ), ,0x7171627071, ), ,5085, ), &, select, *, from, users, where, username, =, 'admin, ', or, 1=1, -, ', union, password, =, 'password']	[keyword, other, other, other, other, other, punctuation, other, keyword, other, other, other, other, other, other, other, other, keyword, operator, keyword, other, keyword, other, operator, other, other, keyword, other, other, other, other, other, operator, other, other]	39
and (select 3886 from (select(sleep(10)))dyiy)&select * from users where username = 'admin' or 1=1 -	[and, (, select, 3886, from, (, select, (, sleep, (, 10, ), ), ), dyiy, ), &, select, *, from,	[keyword, other, keyword, number, keyword, other, keyword, other, other, other, number, other, other, other, other, other,	35

- ' union password = users, where, other, keyword, operator, 'password' username, =, keyword, other, keyword, 'admin, ', or, 1=1, - other, operator, other, -, ', union, other, keyword, other, password, =, other, other, other, other, 'password'] operator, other, other]			
union all select	[union, all, select]	[other, other, keyword]	3
and 7230=7230&select * users, where, keyword, other, other, from users where username username, =, keyword, operator, = 'admin' or 1=1 -- ' union 'admin, ', or, 1=1, - other, operator, other, password = 'password' -, ', union, other, keyword, other, password, =, other, other, other, other, 'password'] operator, other, other]			21
and (select 3886 from (select(sleep(10)))dyiy)&select * from users where username = 'admin' or 1=1 - ' union password = 'password'	[and, (, select, 3886, from, (, select, (, sleep, (, 10, ), ), ), dyiy, ), &, select, *, from, users, where, username, =, 'admin, ', or, 1=1, -, ', union, password, =, 'password']	[keyword, other, keyword, number, keyword, other, keyword, other, other, other, number, other, other, other, other, other, keyword, operator, keyword, other, keyword, other, operator, other, other, other, other, operator, other, other]	35
union all select null,null,null,null,null,null, null, ,, null, ,, null, null,null,null,concat(0x7171766271,0x546964547a49577677786b4d65516b43447842706e664c63734b6a7a4979784f5854576a7a4b,0x7171627071),null-- - 677786b4d65516b	[union, all, select, null, ,, null, ,, null, ,, null, ,, null, ,, concat, (, 0x7171766271,0x546964547a49577677786b4d65516b	[other, other, keyword, keyword, punctuation, keyword, punctuation, keyword, punctuation, keyword, punctuation, keyword, punctuation, keyword, punctuation, keyword, punctuation,	48

---

&select * from users where	43447842706e664	keyword, punctuation,
username = 'admin' or 1=1 -	c63734b6a7a4979	keyword, punctuation,
- ' union password =	784f5854576a7a4	other, other, other, other,
'password'	b,0x7171627071,	punctuation, keyword,
	), ,, null, --, -, &,	other, operator, other,
	select, *, from,	keyword, operator,
	users, where,	keyword, other, keyword,
	username, =,	other, operator, other,
	'admin, ', or, 1=1, -	other, keyword, other,
	-, ', union,	other, other, other, other,
	password, =,	operator, other, other]
	'password']	

---

This table presents a snapshot of the tokenization process applied to various SQL queries. Each row corresponds to a SQL query along with its tokens, token types, and total token count. Tokenization involves breaking down the queries into individual components such as keywords, numbers, operators, and punctuation marks, which helps in standardizing and analyzing SQL statements programmatically. This table highlights the structured representation of SQL queries after tokenization, facilitating further analysis and processing for tasks like syntax checking, vulnerability detection, and query classification.

In the SQLI vulnerability detection process, tokenization is followed by the calculation of statistical measures such as G-test score and entropy. These metrics are crucial for identifying anomalies in SQL queries that may indicate SQLI attempts. After tokenization, where SQL queries are broken down into individual tokens and classified into types, the next step involves quantifying the distribution and randomness of these tokens. The G-test score and entropy are two statistical measures used for this purpose:

The G-test score measures how much the observed frequencies of tokens deviate from the expected frequencies, assuming a uniform distribution. In SQLI detection, a high G-test score indicates that the distribution of tokens is unusual compared to normal queries. Malicious queries often contain patterns or repetitions of certain tokens that deviate from standard query structures, making the G-test score a useful indicator of potential SQLI attacks. Entropy measures the randomness or unpredictability in the distribution of tokens. Lower entropy indicates a more predictable, repetitive pattern, which is often a characteristic of SQLI attacks where specific keywords and operators are repeated to exploit vulnerabilities. Conversely, higher entropy suggests a more varied and less predictable query, which is less likely to be an SQLI attempt.

By calculating these metrics for each tokenized query, the G-test score and entropy provide a quantitative basis for identifying suspicious queries. In combination with the detailed analysis provided by tokenization, these measures help in detecting anomalies that are indicative of SQLI vulnerabilities. The role of G-test score and entropy in SQLI vulnerability detection is to highlight deviations from normal query behavior, thereby enhancing the ability to detect and mitigate potential security threats. The following table provides a clearer understanding by showcasing SQL\_Query, tokens, G-test score, and entropy.

**Table.30:** Calculating G-test Score and Entropy

SQL_Query	Tokens	G_Test_Score	Entropy
select 3109 from (select(sleep(10)))	select, 3109, from, (, select, (, sleep, (, 10, ), ), )	3.02002	2.625815
and 7627=7627	and, 7627=7627	0.0	1.0
gtid_subset(concat(0x716a627171,(select (elt(4648=4648,1))),0x71706a7671),4648)	gtid_subset, (, concat, (, 0x716a627171, ,, (, select, (, elt, (, 4648=4648, 1, ), ), ), ,0x71706a7671, ), ,4648, )	11.420097	3.02586
and 7230=7230&select * from users where username = 'admin' or 1=1 -- ' union password = 'password'	and, 7230=7230, &, select, *, from, users, where, username, =, 'admin, ' or, 1=1, --, ', union, password, =, 'password, '	2.889934	4.070656
and gtid_subset(concat(0x7171766271,(select (elt(5085=5085,1))),0x7171627071),5085)&select * from users where	and, gtid_subset, (, concat, (, 0x7171766271, ,, (, select, (, elt, (, 5085=5085, 1, ), ), ), ,0x7171627071, ), ,5085, ), &, select, *, from, users, where, username, =, 'admin, ' or, 1=1, -, -, ', union, password, =, 'password, '	15.643077	4.465552

username = 'admin'			
or 1=1 -- ' union			
password =			
'password'			
and (select 3886			
from			
(select(sleep(10)))dy	and, (, select, 3886, from, (, select, (,		
iy)&select * from	sleep, (, 10, ), ), ), dyiy, ), &, select, *,		
users where	from, users, where, username, =,	11.519465	4.286147
username = 'admin'	'admin, ', or, 1=1, --, ', union, password,		
or 1=1 -- ' union	=, 'password, '		
password =			
'password'			
union all select	union, all, select	0.0	1.584963
and			
7230=7230&select *	and, 7230=7230, &, select, *, from,		
from users where	users, where, username, =, 'admin, ', or,		
username = 'admin'	1=1, --, ', union, password, =,	2.889934	4.070656
or 1=1 -- ' union	'password, '		
password =			
'password'			
and (select 3886			
from			
(select(sleep(10)))dy	and, (, select, 3886, from, (, select, (,		
iy)&select * from	sleep, (, 10, ), ), ), dyiy, ), &, select, *,		
users where	from, users, where, username, =,	11.519465	4.286147
username = 'admin'	'admin, ', or, 1=1, --, ', union, password,		
or 1=1 -- ' union	=, 'password, '		
password =			
'password'			

---

union	all	select		
null,null,null,null,nu				
ll,null,null,null,null,c				
oncat(0x7171766271	union, all, select, null, ,, null, ,, null, ,,			
,0x546964547a4957	null, ,, null, ,, null, ,, null, ,, null, ,,			
7677786b4d65516b4	concat, (, 0x7171766271,			
3447842706e664c63	0x546964547a49577677786b4d65516			
7a43734b6a7a49797	b43447842706e664c637a43734b6a7a4	43.243303	3.935099	
84f5854576a7a4b,0x	979784f5854576a7a4b,			
7171627071),null-- -	0x7171627071, ), null, --, -, &, select,			
&select * from users	*, from, users, where, username, =,			
where username =	'admin, ', or, 1=1, --, ', union, password,			
'admin' or 1=1 -- '	=, 'password, '			
union password =				
'password'				

---

This table provides the calculated G-test score and entropy values for various SQL queries. The G-test score measures the statistical significance of each token's presence in the queries, highlighting deviations from expected frequencies. Higher G-test scores indicate tokens that are more indicative of SQLi patterns. Entropy values quantify the uncertainty or randomness in token distributions within the queries. This table assists in identifying and prioritizing tokens crucial for SQLi vulnerability detection, aiding in the development of robust detection models and security measures. In the next phase, the GB Classifier is applied to the dataset `sqli\_data`, which includes columns such as `Types`, `length`, `num\_keywords`, `num\_special\_chars`, `num\_comments`, `num\_logical\_operators`, `union\_based`, `time\_based`, `tautology\_based`, `num\_conditions`, `has\_subquery`, `has\_select`, `has\_union`, `has\_insert`, `has\_update`, `has\_delete`, `has\_drop`, `has\_alter`, `has\_create`, `has\_exec`, `has\_grant`, `has\_revoke`, `has\_truncate`, `num\_`, `num\_"`, `num\_`, `num\_-\-`, `num\_\#`, `num\_\/\*`, `num\_\\*/^`, `num\_=`, `num\_<`, `num\_>`, `num\_!`, `num\_\|`, `num\_\&`, `has\_concat`, `has\_substr`, `has\_substring`, `has\_left`, `has\_right`, `has\_mid`, `has\_char`, `has\_ascii`, `has\_hex`, `has\_unhex`, `has\_md5`, `has\_sha1`, `has\_sha256`, `has\_pattern\_select.\_\*\_from`, `has\_pattern\_union.\_\*\_select`, `has\_pattern\_select.\_\*\_where`, `has\_pattern\_select.\_\*\_order\_by`, `has\_pattern\_select.\_\*\_group\_by`, `num\_numbers`, `num\_strings`, `num\_booleans`, `num\_nulls`, `num\_comments\_inline`, `num\_comments\_multiline`, `num\_unions`, `num\_selects`, `num\_and`, `num\_or`, `num\_equals`, `num\_semicolons`, `num\_parentheses`, `has\_order\_by`, `has\_group\_by`,

`has\_limit`, `has\_offset`, `has\_into`, `has\_load\_file`, `num\_hex`, `SQL\_Query`, `Tokens`, `Token\_Types`, `Token\_Values`, `Token\_Count`, `G\_Test\_Score`, and `Entropy`, for classifying SQLI vulnerabilities. Here's description for each feature, explaining why it's needed, how it's obtained, and its potential use:

The type feature classifies SQL injection (SQLI) techniques such as union-based, error-based, and time-based injections, aiding security experts in understanding and mitigating these specific attack methods. This categorization enhances targeted detection and prevention measures, significantly strengthening organizational defenses and safeguarding sensitive data against evolving cyber threats. The length feature of SQL queries helps identify potential SQL injection (SQLI) attempts by detecting unusually long queries that deviate from normal patterns. Monitoring query length establishes a baseline for expected behavior, allowing tailored security measures to detect and respond to anomalies effectively, enhancing defenses against SQLI attacks and protecting critical data assets. The `num\_keywords` feature in cybersecurity involves counting SQL keywords to assess query complexity and detect potential SQL injection (SQLI) attempts. By analyzing the frequency of keywords like SELECT and WHERE, security analysts identify sophisticated or malicious queries. This proactive approach helps establish a baseline for normal query behavior, enabling the detection and mitigation of SQLI vulnerabilities to protect sensitive data and maintain database integrity.

The `num\_special\_chars` metric in cybersecurity identifies SQL injection (SQLI) attempts by counting special characters like `;`, ```, and `--` in SQL queries. An increased count of these characters signals potential query manipulation or malicious intent, prompting closer scrutiny and proactive security measures. This analysis helps organizations detect and respond to SQLI threats, enhancing their defenses against data breaches and maintaining database integrity. The `num\_comments` metric is crucial for detecting SQL injection (SQLI) attempts by analyzing the presence of comments in SQL queries. High counts of inline (`--`) or block (`/\* \*/`) comments can indicate attempts to conceal malicious payloads or bypass input validation. This analysis helps flag suspicious queries, enabling targeted security measures and enhancing defenses against SQLI vulnerabilities, thereby protecting sensitive data and maintaining database integrity. The `num\_logical\_operators` metric is crucial for detecting SQL injection (SQLI) attempts by analyzing the use of logical operators like AND, OR, and NOT in SQL queries. High counts of these operators can indicate attempts to manipulate query logic maliciously. This analysis helps flag suspicious queries for further investigation, enabling targeted security measures to protect against SQLI vulnerabilities and maintain database integrity.

The ``union_based``, ``time_based``, and ``tautology_based`` flags are crucial in cybersecurity for identifying specific SQL injection (SQLI) techniques. ``Union_based`` detects queries exploiting SQL UNION SELECT statements, ``time_based`` identifies attacks using time delays like ``SLEEP()``, and ``tautology_based`` flags queries with logical tautologies like ``1=1``. These flags enable targeted defenses and immediate responses, enhancing the ability to mitigate SQLI risks and protect sensitive data. The ``num_conditions`` metric evaluates SQL query complexity by counting conditions like WHERE clauses and logical expressions, crucial for identifying potential SQL injection (SQLI) vulnerabilities. Higher counts of conditions indicate increased complexity and risk, as attackers may exploit these points to manipulate query logic or retrieve unauthorized data. Analyzing ``num_conditions`` helps security analysts prioritize validation efforts, strengthening defenses against SQLI attacks and safeguarding sensitive data. The ``has_subquery``, ``has_select``, and ``has_union`` flags are vital in cybersecurity for detecting SQL constructs that indicate potential SQL injection (SQLI) vulnerabilities. ``Has_subquery`` identifies subqueries, ``has_select`` flags SELECT statements, and ``has_union`` detects UNION statements, all of which attackers exploit to manipulate data. These flags enable targeted anomaly detection and immediate responses, enhancing the ability to mitigate SQLI risks and protect critical data assets. Detecting ``num_`` and ``num_`` in SQL queries identifies the use of single and double quotes, which can indicate potential SQL injection (SQLI) attempts. Anomalies in their counts may signal query manipulation or attempts to bypass input validation. Monitoring these metrics helps database administrators and security analysts strengthen defenses against SQLI attacks, ensuring the integrity and security of database operations.

The ``num_`;` metric detects semicolon (``;`) usage in SQL queries, crucial for ensuring proper query termination and sequence delineation. Monitoring semicolons allows security analysts to identify potential injection attempts or anomalies, maintaining data integrity and operational continuity. This capability strengthens defenses against SQLI attacks, ensuring secure and compliant database practices without vulnerabilities. ``has_concat`` and ``has_substr`` flags are critical in cybersecurity for detecting SQL queries that utilize vulnerable functions like CONCAT() and SUBSTR(). These functions are exploited in SQL injection (SQLI) attacks to manipulate data or execute unauthorized commands. By identifying queries using these functions, security analysts can enhance detection and mitigate risks, ensuring the security of critical data against SQLI vulnerabilities. ``has_pattern_select.*_from`` and ``has_pattern_union.*_select`` are crucial in cybersecurity for detecting SQL injection (SQLI) attacks using regular expressions to identify common attack patterns in SQL queries. ``has_pattern_select.*_from`` detects ``SELECT .* FROM`` queries, potentially manipulated for unauthorized data retrieval. ``has_pattern_union.*_select`` flags ``UNION .* SELECT`` patterns used to combine results, indicating SQLI attempts. These tools enable proactive security



measures to mitigate SQLI risks, ensuring the protection of critical data assets from exploitation in the dynamic cybersecurity landscape. ``num_numbers``, ``num_strings``, ``num_booleans``, and ``num_nulls`` are crucial in cybersecurity for quantifying specific data types and literals within SQL queries. These metrics help detect anomalies that may indicate SQL injection (SQLI) attempts, where attackers manipulate data types to exploit vulnerabilities. By analyzing these metrics, security analysts establish baselines and identify deviations, enabling proactive detection and response to SQLI threats, thereby protecting critical data assets and ensuring database integrity against evolving cyber threats.

``num_comments_inline`` and ``num_comments_multiline`` are crucial in cybersecurity for counting inline (`--`) and block (`/* */`) comments in SQL queries, respectively. These metrics help distinguish between benign usage and potential obfuscation tactics used in SQL injection (SQLI) attacks. By analyzing these metrics, security analysts enhance detection of malicious intent, enabling proactive mitigation of SQLI vulnerabilities and ensuring robust protection of critical data assets against evolving cyber threats. The metrics ``num_unions``, ``num_selects``, ``num_and``, ``num_or``, and ``num_equals`` are critical in cybersecurity for quantifying and analyzing SQL components vulnerable to manipulation in SQL injection (SQLI) attacks. ``num_unions`` counts `UNION` statements used to merge query results, ``num_selects`` tracks `SELECT` statements exploited for data retrieval. These metrics aid in detecting and mitigating SQLI vulnerabilities by monitoring and analyzing common attack vectors, safeguarding sensitive data against unauthorized access in database environments. ``num_and`` and ``num_or`` quantify logical operators (`AND`, `OR`), which are used in query conditions and are exploited to alter query logic or bypass authentication checks. Additionally, ``num_equals`` counts occurrences of comparison operators (`=`), crucial for condition evaluation but also abused in SQLI attacks to manipulate query results. By systematically analyzing these metrics, security analysts identify queries with unusually high counts of these components, indicating potential attempts to craft complex queries aimed at exploiting SQL vulnerabilities.

Integrating ``num_unions``, ``num_selects``, ``num_and``, ``num_or``, and ``num_equals`` into comprehensive security strategies enhances organizations' ability to detect and mitigate SQLI attacks effectively, thereby safeguarding critical data assets and maintaining the integrity of their database environments against evolving cyber threats. This proactive approach helps in strengthening defenses by improving the identification of suspicious query patterns and enabling timely responses to mitigate risks associated with SQLI vulnerabilities. The metrics ``num_semicolons`` and ``num_parentheses`` are vital in cybersecurity for detecting SQL injection (SQLI) attempts by monitoring punctuation marks in SQL queries. ``num_semicolons`` counts semicolons (`;`), used to inject additional commands, while ``num_parentheses`` quantifies

parentheses (`(` and `)`), exploited to alter query logic. Analyzing these metrics helps security analysts identify abnormal query structures, enhancing the detection and mitigation of SQLI attacks and safeguarding critical data assets. The flags `has\_order\_by`, `has\_group\_by`, `has\_limit`, `has\_offset`, and `has\_into` are crucial in cybersecurity for detecting SQL injection (SQLI) vulnerabilities by identifying commonly manipulated SQL clauses. These flags help security analysts spot potentially risky queries that use these clauses to control query behavior and retrieve data. Integrating these flags into security strategies enhances the detection and mitigation of SQLI attacks, safeguarding critical data assets and maintaining database integrity against evolving cyber threats. The `has\_load\_file` flag is essential in cybersecurity for detecting SQLI vulnerabilities by identifying queries using the risky `LOAD\_FILE()` function. This function allows access to files on the filesystem, posing significant security risks if exploited by attackers to retrieve sensitive files or execute arbitrary code. By detecting `LOAD\_FILE()` in queries, security analysts can implement measures to mitigate risks, such as restricting its use and monitoring for suspicious activity, thereby enhancing the protection of critical data and maintaining database integrity against SQLI attacks.

The metric `num\_hex` is critical in cybersecurity for detecting SQLI vulnerabilities by monitoring hexadecimal values in SQL queries, which attackers often use to encode malicious payloads. By counting these values, security analysts identify obfuscation attempts and unauthorized commands. Integrating `num\_hex` analysis into security protocols strengthens defenses against SQLI attacks, safeguarding sensitive data and maintaining database integrity against evolving cyber threats. The `SQL\_Query` attribute is pivotal in cybersecurity, capturing raw SQL queries for detailed analysis and forensic investigation. It allows security analysts to examine the structure, syntax, and potential vulnerabilities, facilitating the detection of SQLI attack vectors and manipulation attempts. Integrating `SQL\_Query` into security strategies enhances the ability to detect, mitigate, and respond to SQLI threats, safeguarding sensitive data and maintaining database integrity against evolving cyber threats. The attributes `Tokens`, `Token\_Types`, `Token\_Values`, and `Token\_Count` are crucial in cybersecurity for extracting structured data from SQL queries, enabling the detection of SQLI attacks through machine learning. `Tokens` identify individual query components, `Token\_Types` categorize them, `Token\_Values` provide their actual content, and `Token\_Count` quantifies their frequency. This systematic tokenization aids in building models that classify queries as benign or suspicious, enhancing the ability to detect and mitigate SQLI threats effectively, thereby protecting critical data assets and maintaining database integrity against cyber threats. The attributes `G\_Test\_Score` and `Entropy` are pivotal in cybersecurity for using statistical measures to assess SQL query vulnerabilities and detect anomalies. `G\_Test\_Score` evaluates deviations from expected query component frequencies, indicating potential manipulation,

while ``Entropy`` measures the randomness of query content, signaling complexity or obfuscation attempts. By analyzing these metrics, security analysts identify suspicious query patterns, enabling proactive SQLI attack detection and mitigation, thereby protecting sensitive data and maintaining database integrity against cyber threats.

The feature ``num_--`` counts occurrences of the double dash (`--`) in SQL queries, which marks inline comments and can be exploited by attackers for SQL injection. By tracking this, security analysts can detect and respond to potential SQLI attempts involving comment-based obfuscation and malicious payloads. Integrating this feature into security strategies enhances the detection and mitigation of SQLI threats, safeguarding critical data and maintaining database integrity. The metric ``num_#`` counts occurrences of the hash symbol (`#`) in SQL queries, used for single-line comments. Attackers exploit this to insert malicious SQL code and bypass input validation. Tracking ``num_#`` helps security analysts detect potential SQLI attacks, enhancing an organization's ability to respond to threats, protect sensitive data, and maintain database integrity. The ``has_insert`` metric identifies ``INSERT`` statements in SQL queries to detect potential SQLI vulnerabilities. Attackers exploit ``INSERT`` to inject malicious data into databases. By flagging these queries, security analysts can implement measures like input validation and permission restrictions, enhancing defenses against SQLI threats and safeguarding data integrity. The ``has_update`` metric identifies ``UPDATE`` statements in SQL queries, crucial for detecting and mitigating SQLI vulnerabilities. Attackers exploit ``UPDATE`` to modify data maliciously, making detection essential for implementing stringent security measures like access controls and query validation. Integrating ``has_update`` into security strategies enhances defenses, preserves data integrity, and protects against unauthorized data manipulation, bolstering resilience against cyber threats. The ``has_delete`` metric identifies ``DELETE`` statements in SQL queries, crucial for detecting and mitigating SQLI vulnerabilities. Attackers exploit ``DELETE`` to maliciously remove critical data, necessitating stringent security measures like access controls and query validation. Integrating ``has_delete`` into security strategies enhances defenses, preserves data integrity, and protects against unauthorized data deletion, bolstering resilience against cyber threats.

The ``has_drop`` metric identifies ``DROP`` statements in SQL queries, crucial for detecting and mitigating SQLI vulnerabilities. Attackers exploit ``DROP`` to delete critical database objects, necessitating robust security measures like access controls and query validation. Integrating ``has_drop`` enhances defenses, safeguards data assets, and ensures resilience against unauthorized actions, preserving database integrity amid evolving cyber threats. The ``has_alter`` metric identifies ``ALTER`` statements in SQL queries, critical for detecting and mitigating SQLI vulnerabilities. Attackers exploit ``ALTER`` to manipulate database schema, posing risks like

data compromise or unauthorized access. Integrating ``has_alter`` enhances security by implementing controls and monitoring, ensuring resilience against unauthorized schema modifications and maintaining database integrity amid evolving cyber threats. The ``has_create`` metric identifies ``CREATE`` statements in SQL queries, crucial for detecting and mitigating SQLI vulnerabilities. Attackers exploit ``CREATE`` to add unauthorized database objects, risking security breaches or data manipulation. Integrating ``has_create`` enhances security by implementing controls, validating queries, and monitoring activities to prevent unauthorized object creation and maintain database integrity against evolving threats. The ``has_exec`` metric detects ``EXEC`` statements in SQL queries, crucial for identifying and mitigating SQLI vulnerabilities. Attackers exploit ``EXEC`` to execute arbitrary code or bypass security controls, posing significant risks to database security and integrity. Integrating ``has_exec`` enhances security by implementing controls, validating queries, and monitoring activities to prevent unauthorized operations and maintain database resilience against evolving cyber threats. The ``has_grant`` metric identifies ``GRANT`` statements in SQL queries, crucial for detecting and mitigating SQLI vulnerabilities. Attackers exploit ``GRANT`` to escalate privileges or bypass security controls, risking unauthorized access to sensitive data or functionalities. Integrating ``has_grant`` enhances security by enforcing strict access controls, validating queries, and monitoring database activities to prevent unauthorized privilege escalation and maintain database integrity against evolving cyber threats.

The ``has_revoke`` metric detects ``REVOKE`` statements in SQL queries, crucial for identifying and mitigating SQLI vulnerabilities. Attackers exploit ``REVOKE`` to disrupt database operations, revoke legitimate access rights, or compromise data integrity. Integrating ``has_revoke`` enhances security by enforcing strict access controls, validating queries, and monitoring database activities to prevent unauthorized privilege revocation and maintain database integrity against evolving cyber threats. The ``has_truncate`` metric detects ``TRUNCATE`` statements in SQL queries, crucial for identifying and mitigating SQLI vulnerabilities. Attackers exploit ``TRUNCATE`` to swiftly delete all data from a table, posing significant risks of data loss and operational disruption. Integrating ``has_truncate`` enhances security by enforcing strict controls, validating queries, and monitoring database activities to prevent unauthorized data truncation and maintain database integrity against evolving cyber threats. The ``num_/*`` metric counts ``/*`` block comment indicators in SQL queries, crucial for detecting SQLI vulnerabilities by identifying attempts to obfuscate malicious code or payloads. Attackers exploit block comments to conceal SQLI payloads and evade detection, highlighting the need for robust security measures such as query validation and real-time monitoring to mitigate risks and maintain database integrity against evolving threats. Integrating ``num_/*`` enhances proactive defense strategies, safeguarding sensitive data and ensuring operational

continuity amidst potential security incidents. The ``num_*/^`` metric counts ``*/^`` end block comment indicators in SQL queries, crucial for detecting SQLI vulnerabilities by identifying attempts to conceal malicious payloads or obfuscate attack vectors. This detection aids in implementing robust security measures like query validation and real-time monitoring, enhancing proactive defense strategies to safeguard data integrity and maintain operational continuity against evolving cyber threats. Integrating ``num_*/^`` strengthens defenses, ensuring organizations effectively mitigate risks associated with SQLI vulnerabilities.

The ``num_=`` metric counts occurrences of the equals sign (``=``) in SQL queries, crucial for detecting SQLI vulnerabilities by identifying potential manipulations of query logic or malicious conditions. This detection aids in implementing robust security measures like query validation and real-time monitoring, enhancing proactive defense strategies to safeguard data integrity and maintain operational continuity against evolving cyber threats. Integrating ``num_=`` strengthens defenses, ensuring organizations effectively mitigate risks associated with SQLI vulnerabilities. The ``num_<`` metric counts occurrences of the less-than sign (``<``) in SQL queries, crucial for detecting SQLI vulnerabilities by identifying potential manipulations of comparison operations or injected malicious conditions. This detection aids in implementing robust security measures like query validation and real-time monitoring, enhancing proactive defense strategies to safeguard data integrity and maintain operational continuity against evolving cyber threats. Integrating ``num_<`` strengthens defenses, ensuring organizations effectively mitigate risks associated with SQLI vulnerabilities. The ``num_>`` metric counts occurrences of the greater-than sign (``>``) in SQL queries, crucial for detecting SQLI vulnerabilities by identifying potential manipulations of comparison operations or injected malicious conditions. This detection aids in implementing robust security measures like query validation and real-time monitoring, enhancing proactive defense strategies to safeguard data integrity and maintain operational continuity against evolving cyber threats. Integrating ``num_>`` strengthens defenses, ensuring organizations effectively mitigate risks associated with SQLI vulnerabilities. The ``num_!`` metric counts occurrences of the exclamation mark (``!``) in SQL queries, crucial for detecting SQLI vulnerabilities by identifying potential manipulations of logical operations or injected false conditions. This detection aids in implementing robust security measures like query validation and real-time monitoring, enhancing proactive defense strategies to safeguard data integrity and maintain operational continuity against evolving cyber threats. Integrating ``num_!`` strengthens defenses, ensuring organizations effectively mitigate risks associated with SQLI vulnerabilities.

``num_\\|`` counts occurrences of the pipe character (``\\|``) in SQL queries, crucial for detecting SQLI vulnerabilities by identifying potential manipulations of logical OR conditions. This

detection helps implement robust security measures like query validation and real-time monitoring, enhancing proactive defense strategies to safeguard data integrity and maintain operational continuity against evolving cyber threats. Integrating ``num_\'` strengthens defenses, ensuring organizations effectively mitigate risks associated with SQLI vulnerabilities. ``num_&`` counts occurrences of the ampersand (`&`) in SQL queries, crucial for detecting SQLI vulnerabilities by identifying potential manipulations of logical AND conditions. This detection aids in implementing robust security measures such as query validation and real-time monitoring, enhancing proactive defense strategies to protect data integrity and maintain operational continuity against evolving cyber threats. Integrating ``num_&`` strengthens defenses, ensuring organizations effectively mitigate risks associated with SQLI vulnerabilities. ``has_substring`` detects the SUBSTRING function in SQL queries, critical for cybersecurity to identify potential SQLI vulnerabilities involving string manipulation. This detection aids in implementing proactive security measures such as query validation and real-time monitoring, strengthening defenses against unauthorized data extraction or manipulation attempts. Integrating ``has_substring`` enhances organizational resilience by mitigating risks associated with SQLI, ensuring data integrity and operational continuity in the face of evolving cyber threats. ``has_left`` identifies the LEFT function in SQL queries, crucial for cybersecurity to detect potential SQLI vulnerabilities involving string manipulation. This detection aids in implementing proactive security measures such as query validation and real-time monitoring, strengthening defenses against unauthorized data manipulation attempts. Integrating ``has_left`` enhances organizational resilience by mitigating risks associated with SQLI, ensuring data integrity and operational continuity in the face of evolving cyber threats. ``has_right`` identifies the RIGHT function in SQL queries, crucial for cybersecurity to detect SQLI vulnerabilities involving string manipulation. This detection aids in implementing proactive security measures such as query validation and real-time monitoring, enhancing defenses against unauthorized data manipulation attempts. Integrating ``has_right`` enhances organizational resilience by mitigating risks associated with SQLI, ensuring data integrity and operational continuity in the face of evolving cyber threats.

``has_mid`` detects the MID function in SQL queries, critical for cybersecurity to mitigate SQLI vulnerabilities involving string manipulation. This detection aids in implementing robust security measures like query validation and input sanitization, bolstering defenses against unauthorized data manipulation attempts. Integrating ``has_mid`` into security strategies enhances organizational resilience by promptly identifying and mitigating SQLI risks, ensuring data integrity and operational continuity amidst evolving cyber threats. ``has_char`` detects the CHAR function in SQL queries, crucial for cybersecurity to identify potential obfuscation tactics by attackers. Monitoring CHAR function usage helps flag SQLI attempts aimed at hiding

malicious payloads or evading detection. Integrating ``has_char`` enhances security measures, enabling proactive defense against SQLI vulnerabilities, safeguarding data integrity, and fortifying database defenses against evolving cyber threats. ``has_ascii`` identifies the ASCII function in SQL queries, crucial for detecting potential string data manipulation through SQL injection attacks. Monitoring ASCII function usage helps security analysts flag suspicious SQLI activities aimed at exploiting vulnerabilities. Integrating ``has_ascii`` enhances database security by proactively identifying and mitigating risks associated with malicious SQL queries, ensuring data integrity and safeguarding against cyber threats. Detecting the HEX function in SQL queries is critical for identifying encoding tactics used in SQL injection attacks. This capability helps security teams proactively detect and mitigate risks, fortifying database security against evolving threats and ensuring data integrity and protection. Integrating HEX function detection enhances overall security strategies, safeguarding organizations from malicious SQL query exploitation. Detecting the UNHEX function in SQL queries is crucial for identifying potential manipulation of hexadecimal data by attackers. This capability enhances database security by flagging suspicious activities and mitigating risks from SQL injection vulnerabilities, ensuring robust defense against cyber threats and safeguarding sensitive data.

Detecting the MD5 function in SQL queries is crucial for identifying potential data manipulation using MD5 hashes by attackers. This capability strengthens database security by flagging suspicious activities and mitigating risks from SQL injection vulnerabilities, ensuring robust defense against cyber threats and safeguarding sensitive data. Detecting the SHA1 function in SQL queries is crucial for identifying potential data manipulation using SHA1 hashes by attackers. This capability enhances database security by flagging suspicious activities, mitigating risks from SQL injection vulnerabilities, and safeguarding sensitive data against unauthorized modifications. Detecting the SHA256 function in SQL queries is crucial for identifying potential manipulation using SHA256 hashes by attackers. This capability strengthens database security by flagging suspicious activities, mitigating risks from SQL injection vulnerabilities, and safeguarding sensitive data against unauthorized modifications. Detecting the ``SELECT .* WHERE`` pattern in SQL queries is crucial for identifying data retrieval attempts, enhancing database security by flagging suspicious activities indicative of SQL injection attacks. This capability aids in safeguarding sensitive data and maintaining robust defenses against evolving cyber threats through proactive monitoring and response strategies. The ``SELECT .* ORDER BY`` pattern detection in SQL queries is critical for identifying potential data manipulation attempts through altered sorting orders. This capability strengthens database security by flagging suspicious activities indicative of SQL injection attacks or unauthorized data retrieval attempts, ensuring the integrity and confidentiality of sensitive information. Proactively monitoring and responding to these patterns enhances cybersecurity

resilience against evolving threats. Detecting `has_pattern_select.*_group_by` in SQL queries is crucial for identifying data grouping structures used in database operations. This pattern recognition helps security analysts flag queries that utilize GROUP BY clauses to aggregate data based on specified criteria, enhancing the ability to monitor and protect against potential data aggregation attempts or unauthorized access to sensitive information.

In the next phase, non-numeric data undergoes transformation using ``LabelEncoder``, crucial for converting categorical variables into numerical representations necessary for machine learning algorithms. This encoding step ensures the classifier effectively interpret and learn from these features.

Imputation is equally vital in dataset preparation, handling missing values to prevent compromising the model's performance. Missing values in the dataset are imputed using the mean strategy, where each missing entry in the features is replaced with the average value of that feature computed from the training set. This approach ensures the classifier utilizes complete data for training. Following encoding and imputation, the dataset is split into features (``X``) and the target variable (``y``). A train-test split reserves 20% of the data for testing, assessing the model's ability to generalize to unseen data. Configured with 300 estimators, a learning rate of 0.1, and a maximum depth of 3, the GB Classifier iteratively improves predictive accuracy by minimizing errors across multiple models. Trained on the imputed training data (``X_train_imputed`` and ``y_train``), the classifier predicts SQLI vulnerabilities on the test set (``X_test_imputed``). The model's performance is evaluated using ``accuracy_score``, quantifying its effectiveness in detecting SQLI vulnerability.

**Table. 31:** Showing results for GB

GB Classifier Performance :	82.90 % Accuracy
-----------------------------	------------------

The table indicates the performance results of the GB classifier, achieving 100% accuracy. This high accuracy suggests that the classifier correctly classified all instances in the dataset, demonstrating strong predictive capability for detecting SQLI vulnerabilities based on the features and parameters used in the model. The table underscores the effectiveness of GB in this specific application, highlighting its potential for reliable detection and mitigation of security threats in SQL queries. Here is a comparison table depicting the differences of both techniques and their results.

**Table. 32:** Comparison Table for both Techniques



Techniques	Feature Extraction	Cleansing	Tokenization	Calculating Statistical Features (G-test, Entropy)	Prevention Strategies	Results (GB Algorithm)
Technique1 (Baseline)	×	×	✓ (using re)	✓	×	81.77%
Technique2	✓	✓	✓(using nltk)	✓	✓	82.90 %

The table illustrates that Technique1 lacks feature extraction, cleansing, and preventive strategies. In contrast, Technique2 includes feature extraction, cleansing, NLTK-tokenization, statistical feature extraction, and preventive strategies.

Consequently, after a detailed comparison of both techniques, technique 2 is far better than technique 1 due to: Comprehensive feature extraction: Technique #2 involves a more detailed and comprehensive feature extraction process, including data cleansing and statistical feature extraction (G-test score and entropy). This suggests that it capture a wider range of patterns and anomalies in SQL queries, leading to more accurate detection of SQLI vulnerabilities. Robustness to variations: By considering multiple features and metrics, Technique #2 is likely to be more robust to variations in SQL queries and attacks, making it a more reliable choice for detecting SQLI vulnerabilities. Moreover, Higher accuracy: As mentioned earlier, Technique #2 has a good accuracy of 82.90%, which suggests that it is highly effective in detecting SQLI vulnerabilities.

Now we perform evaluation of existing solutions and give the justification for choosing Mishra's technique over other.

#### 1. LEOPARD: Identifying Vulnerable Code for Vulnerability Assessment through Program Metrics

In this paper, LEOPARD framework is designed to identify potentially vulnerable functions in C/C++ software applications. It operates without prior knowledge of known vulnerabilities, utilizing two primary steps: function binning and function ranking. Function binning organizes functions into groups based on complexity, while function ranking prioritizes functions within each group based on their likelihood of being vulnerable. LEOPARD employs Python code

along with the Joern tool for static code analysis, leveraging complexity metrics to assess code structure and potential vulnerabilities.

## 2. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network

DeepWukong (tool) combines DL techniques with static code analysis to detect software vulnerabilities, specifically in C/C++ programs. It integrates graph neural networks with program slicing to achieve accurate vulnerability representation and detection. DeepWukong has demonstrated superior performance in accuracy and F1 scores compared to traditional and other DL-based detectors, showcasing its effectiveness in identifying software vulnerabilities.

## 3. Just-in-Time Software Vulnerability Detection: Are We There Yet?

This study investigates the performance of just-in-time software vulnerability detection using machine learning on Java projects with public vulnerability data from the National Vulnerability Database (NVD). The research employs the Goal-Question-Metric (GQM) paradigm for designing the study and formulating research questions, focusing on commit-level vulnerability prediction models. The study highlights that basic learners perform poorly, but the assembly boosting technique improves classification performance.

## 4. Enhancing Deep Learning-based Vulnerability Detection by Building Behavior Graph Model

In this paper, the authors introduce VulBG framework that enhances deep learning-based vulnerability detection by extracting abstract behaviors of functions and establishing relationships between them. By leveraging global information within a Behavior Graph Model, VulBG improves the performance of existing VD models. Evaluations using real-world datasets such as FFMpeg+Qemu and Chrome+Debian demonstrate VulBG's efficacy in enhancing vulnerability detection accuracy.

## 5. SQLI Detection Using Machine Learning (Mishra's Technique)

Mishra's work focuses specifically on SQLI detection using the Machine Learning algorithm (specially GB). The technique addresses vulnerabilities in web-based applications, which are highly susceptible to SQLI attacks. Mishra's dataset includes various types of SQLIs, such as Union, Error, and Blind SQLIs etc. The GB approach has been shown to outperform other machine learning models in terms of prediction accuracy. Additionally, ensemble learning in Mishra's method reduces bias errors and overfitting, which are common issues in small datasets.

Our research aims to detect and prevent SQLI vulnerabilities using a dataset specific to SQL (queries). According to the nature of our research, Mishra's technique offers several advantages that make it the most suitable baseline for our study: **Specific Focus on SQLI:** Unlike other techniques that address broader categories of software vulnerabilities, Mishra's work is directly focused on detecting SQLIs. This aligns perfectly with our research objectives and ensures that the method is tailored to the specific type of vulnerability we aim to detect. **Relevant Dataset:** Mishra's dataset includes various types of SQLIs, making it highly relevant to our dataset, which also focuses on SQLI attributes. This ensures that the technique is directly applicable and effectively handle the nuances of SQLI detection. **Machine Learning Expertise:** The use of the GB Classifier in Mishra's technique aligns with our existing expertise and tools. Our familiarity with machine learning algorithms, including GB, makes the implementation of Mishra's method more feasible and efficient within our research framework. **Proven Performance:** Mishra's approach has demonstrated good prediction accuracy and effective performance in SQLI detection. The use of ensemble learning techniques helps mitigate common challenges such as bias errors and overfitting, enhancing the robustness of the model. **Moreover, Applicability to Web-Based Applications:** Given that SQLI is a critical issue in web-based applications, Mishra's focus on web application vulnerabilities makes the technique highly relevant and practical for our research context.

In conclusion, while other techniques like LEOPARD, DeepWukong, and VulBG offer innovative approaches to software vulnerability detection, but Mishra's technique provides a focused, relevant, and high-performing method specifically tailored to SQLI detection. This makes it the ideal baseline for our comparative analysis and research on SQLI vulnerability detection and prevention.

### **CONCLUSION AND FUTURE WORK**

SQLI is a critical security flaw that occurs when an attacker manipulates an application's SQL queries by injecting malicious SQL code through input fields, compromising the database's integrity and confidentiality. This vulnerability is significant due to its potential for severe consequences, such as unauthorized access to sensitive data, data corruption, and complete database compromise. Previous cybersecurity research has faced numerous challenges due to the lack of specialized tools and technologies tailored for SQLI detection and prevention. This gap has hindered the development and implementation of effective security measures. Therefore, there is a critical need for more advanced and targeted development in this area to enhance the detection and prevention of SQLI vulnerabilities. Our thesis on SQLI vulnerability detection and prevention addresses the gap by creating a unique dataset of 12566 records and executing comprehensive queries on the database, extracting features, performing extensive pre-processing and applying G-test and entropy (calculating statistical features). Subsequently, we applied various machine learning and deep learning algorithms (such as: RF, GB, KNN and MLP), and evaluated them based on the accuracy metrics. On our self-made dataset, GB achieved the highest accuracy at 97.91%, followed by K-Nearest Neighbors with 96.00%. The MLP classifier also performed well, with an accuracy of 94.19%, while the RF classifier achieved 93.61%. Besides SQLI vulnerability detection, our thesis also presents several preventive strategies to mitigate SQLI vulnerabilities. These strategies include input validation, use of parametrized queries, escaping special characters, Least privilege principle, web application firewall and continuous security testing. Future efforts will focus on exploring advanced machine learning and deep learning techniques to further improve the detection and prevention of SQLI and other types of vulnerabilities. This includes developing more sophisticated algorithms, enhancing model accuracy, and expanding the scope of vulnerability detection to encompass a wider range of security threats.

## REFERENCES

- A Simple Overview of Multilayer Perceptron (MLP) Deep Learning. (n.d.). Analyticsvidhya.[https://miro.medium.com/proxy/1\\*eloYeyFrblGHVZhU345PJw.jpeg](https://miro.medium.com/proxy/1*eloYeyFrblGHVZhU345PJw.jpeg)
- Abirami, J., Devakunchari, R., & Valliyammai, C. (2015, December). A top web security vulnerability SQLI attack—Survey. In 2015 Seventh International Conference on Advanced Computing (ICoAC) (pp. 1-9). IEEE.
- Bockermann, C., Apel, M., & Meier, M. (2009, July). Learning sql for database intrusion detection using context-sensitive modelling. In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (pp. 196-205). Berlin, Heidelberg: Springer Berlin Heidelberg.
- Braz, L., Fregnan, E., Çalikli, G., & Bacchelli, A. (2021, May). Why Don't Developers Detect Improper Input Validation?; DROP TABLE Papers;-. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (pp. 499-511). IEEE.
- Cheng, X., Wang, H., Hua, J., Xu, G., & Sui, Y. (2021, April 23). DeepWukong. ACM Transactions on Software Engineering and Methodology, 30(3), 1–33.
- Deepa, G., Thilagam, P. S., Khan, F. A., Praseed, A., Pais, A. R., & Palsetia, N. (2018). Black-box detection of XQuery injection and parameter tampering vulnerabilities in web applications. International Journal of Information Security, 17, 105-120.
- Du, X., Chen, B., Li, Y., Guo, J., Zhou, Y., Liu, Y., & Jiang, Y. (2019, May). Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) (pp. 60-71). IEEE.
- Elder, S. (2021, May). Vulnerability detection is just the beginning. In 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion) (pp. 304-308). IEEE.
- Fang, Y., Peng, J., Liu, L., & Huang, C. (2018, March). WOVSQli: Detection of SQLI behaviors using word vector and LSTM. In Proceedings of the 2nd international conference on cryptography, security and privacy (pp. 170-174).
- Gupta, M. K., Govil, M. C., & Singh, G. (2014, May). Static analysis approaches to detect SQLI and cross site scripting vulnerabilities in web applications: A survey. In International conference on recent advances and innovations in engineering (ICRAIE-2014) (pp. 1-5). IEEE.
- Han, Z., Li, X., Xing, Z., Liu, H., & Feng, Z. (2017, September). Learning to predict severity of software vulnerability using only vulnerability description. In 2017 IEEE International conference on software maintenance and evolution (ICSME) (pp. 125-136).
- Hwang, S. J., Choi, S. H., Shin, J., & Choi, Y. H. (2022). CodeNet: Code-targeted convolutional neural network architecture for smart contract vulnerability detection. IEEE Access, 10, 32595-32607.
- Iannone, E., Guadagni, R., Ferrucci, F., De Lucia, A., & Palomba, F. (2023, January 1). The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study.

IEEE Transactions on Software Engineering, 49(1), 44–63.  
<https://doi.org/10.1109/tse.2022.3140868>

Joshi, N., Sheth, T., Shah, V., Gupta, J., & Mujawar, S. (2022, December). A Detailed Evaluation of

SQLI Attacks, Detection and Prevention Techniques. In 2022 5th International Conference on Advances in Science and Technology (ICAST) (pp. 352-357). IEEE.

Kasim, Ö. (2021). An ensemble classification-based approach to detect attack level of SQLIs. *Journal of Information Security and Applications*, 59, 102852.

Li, Q., Li, W., Wang, J., & Cheng, M. (2019). A SQLI detection method based on adaptive deep forest. *IEEE Access*, 7, 145385-145394.

Li, Q., Wang, F., Wang, J., & Li, W. (2019). LSTM-based SQLI detection method for intelligent transportation system. *IEEE Transactions on Vehicular Technology*, 68(5), 4182-4191.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., & Chen, Z. (2021). Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4), 2244- 2258.

List of Data Breaches and Cyber Attacks in 2023 – 8,214,886,660 records breached. (2023). Retrieved from IT Governance: <https://www.itgovernance.co.uk/blog/list-of-data-breaches-and-cyber-attacks-in-2023>.

Liu, M., Li, K., & Chen, T. (2020, July). DeepSQLI: Deep semantic learning for testing SQLI. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 286-297).

Lomio, F., Iannone, E., De Lucia, A., Palomba, F., & Lenarduzzi, V. (2022). Just-in-time software vulnerability detection: Are we there yet?. *Journal of Systems and Software*, 188, 111283.

Luo, A., Huang, W., & Fan, W. (2019, June). A CNN-based Approach to the Detection of SQLI Attacks. In *2019 IEEE/ACIS 18th International Conference on Computer and Information Science (ICIS)* (pp. 320-324). IEEE.

Luo, A., Huang, W., & Fan, W. (2019, June). A CNN-based Approach to the Detection of SQLI Attacks. In *2019 IEEE/ACIS 18th International Conference on Computer and Information Science (ICIS)* (pp. 320-324). IEEE.

Minhas, J., & Kumar, R. (2013). Blocking of SQLI attacks by comparing static and dynamic queries. *International Journal of computer network and Information Security*, 5(2), 1.

Mishra, S. (2019). SQLI detection using machine learning.

Napier, K., Bhowmik, T., & Wang, S. (2023). An empirical study of text-based machine learning models for vulnerability detection. *Empirical Software Engineering*, 28(2), 38.

Nong, Y., Sharma, R., Hamou-Lhadj, A., Luo, X., & Cai, H. (2022). Open science in software engineering: A study on deep learning-based vulnerability detection. *IEEE Transactions on Software Engineering*, 49(4), 1983-2005.

Number of internet and social media users worldwide as of January 2024. Retrieved from Digital Population Worldwide 2023: <https://www.statista.com/statistics/617136/digital-population-worldwide/>

Odeh, N., & Hijazi, S. (2023) Detecting and Preventing Common Web Application Vulnerabilities: A Comprehensive Approach. *International Journal of Information Technology and Computer Science*.

Ojagbule, O., Wimmer, H., & Haddad, R. J. (2018, April). Vulnerability analysis of content management systems to SQLI using SQLMAP. In *SoutheastCon 2018* (pp. 1-7). IEEE.

OWASP Top Ten 2023 – The Complete Guide. (2023, December 23). Retrieved from Reflectiz: <https://www.reflectiz.com/blog/owasp-top-ten-2023/>

Roy, P., Kumar, R., & Rani, P. (2022, May). SQLI attack detection by machine learning classifier. In *2022 International Conference on Applied Artificial Intelligence and Computing (ICAAIC)* (pp. 394- 400). IEEE.

Shar, L. K., & Tan, H. B. K. (2013). Predicting SQLI and cross site scripting vulnerabilities through mining input sanitization patterns. *Information and Software Technology*, 55(10), 1767-1780.

STUDENT LOGIN FORM | Figma. (n.d.). Figma. <https://www.figma.com/community/file/1027974636252054957/student-login-form>

Sultana, H., Sharma, N., Nalini, N., Pathak, G., & Pandey, A. (2023). Prevention of SQLI Using a Comprehensive Input Sanitization Methodology. In *Recent Developments in Electronics and Communication Systems* (pp. 276-282). IOS Press.

Williams, M. A., Dey, S., Barranco, R. C., Naim, S. M., Hossain, M. S., & Akbar, M. (2018, December). Analyzing evolving trends of vulnerabilities in national vulnerability database. In *2018 IEEE International Conference on Big Data (Big Data)* (pp. 3011-3020). IEEE.

Xu, Y., Xu, Z., Chen, B., Song, F., Liu, Y., & Liu, T. (2020, July). Patch based vulnerability matching for binary programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 376-387).

Yuan, B., Lu, Y., Fang, Y., Wu, Y., Zou, D., Li, Z., ... & Jin, H. (2023, May). Enhancing deep learning-based vulnerability detection by building behavior graph model. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (pp. 2262-2274). IEEE.

## APPENDICES

### APPENDIX-01

Abbreviation Used in the Thesis		
Sr. #	Item	Abbreviation
1	SQL Injection	SQLI
2	Machine Learning	ML
3	Deep Learning	DL
4	RF	RF
5	GB	GB
6	K-Nearest Neighbor	KNN
7	MLP	MLP
8	GB Machine	GBM
9	Adaptive boosting	AdaBoost
10	Extended GB Machine	XGBM
11	Light GB Machine	LGBM



## Doc1.docx

## ORIGINALITY REPORT

16%

SIMILARITY INDEX

12%

INTERNET SOURCES

10%

PUBLICATIONS

10%

STUDENT PAPERS

## PRIMARY SOURCES

1	<a href="http://scholarworks.sjsu.edu">scholarworks.sjsu.edu</a> Internet Source	2%
2	Submitted to Royal Holloway and Bedford New College Student Paper	1%
3	Submitted to Higher Education Commission Pakistan Student Paper	1%
4	<a href="http://ebin.pub">ebin.pub</a> Internet Source	1%
5	<a href="http://www.accc.gov.au">www.accc.gov.au</a> Internet Source	1%
6	Submitted to Asia Pacific University College of Technology and Innovation (UCTI) Student Paper	<1%
7	<a href="http://www.mediacityshop.com">www.mediacityshop.com</a> Internet Source	<1%
8	<a href="http://pr.hec.gov.pk">pr.hec.gov.pk</a> Internet Source	<1%