University of Ibadan
Journal of Science and Logics in ICT Research

Integrating Zero-Trust Architecture with Deep Learning Algorithm to Prevent Structured Query Language Injection Attack in Cloud Database

[1]✉ Timadi M. E., [2]Obasi E.C.M.
[1]*Government House, Creek Haven, Yenagoa, Bayelsa*
[2]*Department of Computer Science and Informatics, Federal University, Otuoke, Bayelsa State.*
timadimatiel@gmail.com, obasiec@fuotuoke.edu.ng.

[2] https://orcid.org/0009-0001-1513-9887

Abstract

The increasing reliance on cloud databases has made them a prime target for cyber attacks, with Structured Query Language (SQL) injection being a particularly devastating threat. SQL injection attacks pose significant threats to database security, compromising sensitive information. Deep learning algorithms have emerged as effective solutions to detect and prevent SQL injection attacks. This study proposes a novel approach to detecting SQL injection attack by integrating deep learning-based detection with zero-trust architectute. The proposed system utilizes a Feed-Forward Neural Network (FNN)to analyze database queries and detect potential SQL injection attacks. The FNN model is trained on a dataset of labelled queries, allowing it to learn patterns and anomalies indictive of SQL injection attacks. The output of the FNN model is then integrated with zero-trust architecture, which enforces strict access controls and authentication mechanisms based on the results generated by the FNN model. The model exhibits a precision score approximating 100% accuracy in the classification of queries deemed normal, while achieving a 94% rate of correct classification for queries indicative of SQL injection attacks. By leveraging advanced machine learning techniques, our approach aims to identify and block malicious queries in real-time, ensuring the integrity and security of cloud-based data. Through a comprehensive evaluation, we demonstrate the effectiveness of our deep learning-based solution with zero-trust architecture in detecting SQL injection attacks with high accuracy and low false positives.

Keywords: *cyber attacks, SQL Injection Attack, zero-trust architecture, database protection, ML algorithm.*

1. Introduction

The rapid integration of cloud computing has fundamentally transformed the methodologies by which organizations store, organize, and retrieve data. In particular, cloud databases have emerged as a crucial element of contemporary information technology infrastructure, providing attributes such as scalability, adaptability, and economic efficiency. Implementing a robust access control to protect sensitive data in a database is crucial, as it directly impacts the integrity and confidentiality of query results. Obasi *et. al.* [11] improved the administration of an organization by adopting an easy query method using GraphQL and a Random Forest Model. Nevertheless, the heightened dependence on cloud databases has concurrently introduced novel security vulnerabilities, with Structured Query Language (SQL) injection attacks representing a paramount concern. SQL injection attacks are characterized by the use of maliciously formulated queries that can jeopardize the integrity of databases, extract confidential information, or even seize control of the entire system. Conventional security protocols, including firewalls and intrusion detection systems, frequently prove inadequate against the sophisticated nature of SQL injection attacks. Furthermore, the intricate nature and substantial volume of data within cloud databases render it particularly challenging to detect and mitigate these threats in real-time. Malicious actors can exploit web applications by injecting SQL commands or transmitting special characters via user input to target the database tier, thereby gaining unauthorized access to critical assets Chen [5].

The lack of adequate validation in certain web applications, often attributable to programmer

Timadi M. E. and Obasi E. C. M. (2025). Integrating Zero-Trust Architecture with Deep Learning Algorithm to Prevent Structured Query Language Injection Attack in Cloud Database, . *University of Ibadan Journal of Science and Logics in ICT Research (UIJSLICTR),* Vol. 13 No. 1, pp. 52 – 62

oversight, allows attackers to circumvent authentication protocols and access databases, which enables them to extract or alter data without the requisite authorization Zhang [17]. Consequently, there exists an urgent imperative for innovative methodologies capable of effectively safeguarding cloud databases from SQL injection attacks. Singh and Kumar [15] examined the security challenges associated with cloud databases and underscored the potential of deep learning algorithms in countering SQL injection attacks.

Recent developments in deep learning and machine learning have exhibited significant potential in the identification and prevention of cyber threats. Hilmi & Adnan [7] conducted a review on the Detection of SQL Injection Attacks utilizing Supervised Machine Learning Algorithms. Their investigation elucidates that machine learning possesses substantial capabilities in the processes of identifying and detecting SQL injection attacks. Patel & Bhattacharya. [14] executed a deep learning-oriented SQL injection prevention framework within a cloud-based e-commerce platform. Through the utilization of these advanced technologies, it is feasible to construct intelligent systems that can scrutinize queries, discern patterns, and detect anomalies in real-time, thereby averting SQL injection attacks. This research endeavors to explore the implementation of deep learning algorithms for the protection of cloud databases against SQL injection attacks, with the objective of formulating a robust and efficacious security solution.

1.1 Types of SQL Injection Attack
There are different types of SQL injection attacks can be used to extract sensitive data, modify database information, or even take control of the database server. They demonstrate the various techniques attackers use to exploit vulnerabilities in web applications and databases

1.1.1. Classic SQL Injection:
Injecting malicious SQL code into user input fields to manipulate database queries.

Example: SELECT * FROM users WHERE username = '$username'

1.1.2. Error-Based SQL Injection.
Exploiting error messages to extract sensitive data or database information.

Example: SELECT * FROM users WHERE username = '$username' OR 1=1

1.1.3. Blind SQL Injection.
Injecting malicious SQL code without receiving error messages or direct output.

Example: SELECT * FROM users WHERE username = '$username' AND IF(1=1,SLEEP(5),0)

1.1.4. Time-Based SQL Injection.
Injecting malicious SQL code that takes advantage of time delays to extract data.

Example: SELECT * FROM users WHERE username = '$username' AND SLEEP(5)

1.1.5. Boolean-Based SQL Injection.
Injecting malicious SQL code that uses Boolean logic to extract data.

Example: SELECT * FROM users WHERE username = '$username' AND 1=1

1.1.6. Union-Based SQL Injection.
Injecting malicious SQL code that uses UNION operators to combine queries.

Example: SELECT * FROM users WHERE username = '$username' UNION SELECT * FROM sensitive_data.

1.1.7. Stacked Queries SQL Injection.
Injecting malicious SQL code that executes multiple queries.

Example: SELECT * FROM users WHERE username = '$username'; DROP TABLE users;

1.1.8. Out-of-Band SQL Injection.
The act of deploying malicious SQL code that leverages external conduits for the purpose of data retrieval.

Example: SELECT * FROM users WHERE username = '$username' AND EXTRACTVALUE(xmltype('<root><data>'||username||'</data></root>'),'/root/data')

1.1.9. SQL Injection using Stored Procedures.

Injecting malicious SQL code into stored procedures.

Example: EXEC stored_procedure '$username'

1.1.10. Second-Order SQL Injection.

Injecting malicious SQL code that is executed later, often through stored data.

Example: INSERT INTO users (username, password) VALUES ('$username', '$password')

1.1.11. SQL Injection using XML.

Injecting malicious SQL code using XML data.

Example: SELECT * FROM users WHERE username = '$username' AND XMLType('<root><data>'||username||'</data></root>').extract('//data/text()')

1.1.12. SQL Injection using JSON.

Injecting malicious SQL code using JSON data.

Example: SELECT * FROM users WHERE username = '$username' AND JSON_EXTRACT(json_data, '$.username')

2. Related Works

The application of deep learning algorithms to secure against SQL injection (SQLi) attacks in cloud databases has gained significant attention, attributable to the escalating sophistication of these threats. Numerous investigations underscore the efficacy of neural networks and deep belief networks in identifying and alleviating the impact of such attacks.

Marina *et al.* [9] conducted a study entitled "Machine Learning Blunts the Needle of Advanced SQL Injections." The authors perform a comparative analysis of various methodologies, encompassing conventional Rule-based Intrusion Detection Systems (IDS) alongside advanced machine learning paradigms such as Support Vector Machines, Multilayer Perceptron, and Deep Learning models (including Long Short-Term Memory and Gated Recurrent Units). This comparative analysis is essential for evaluating the efficacy of machine learning in the detection of SQL injection attempts. The results presented in the study demonstrate that machine learning methodologies markedly enhance the identification of malicious patterns within HTTP query strings when juxtaposed with traditional techniques.

Obasi and Nlerum [12] worked on Intrusion Detection System for Structured Query Language Injection Attack in E-Commerce Database. Their system introduces a filter layer specifically designed to verify user inputs and mitigate known SQL injection threats, thereby enhancing the security of e-commerce platforms.

Ayush *et al.* [3] investigated a "Deep Learning Approach for Detection of SQL Injection Attacks Using Convolutional Neural Networks." The authors scrutinized the performance of an array of machine learning algorithms, which included Naive Bayes, Decision Trees, Support Vector Machines, and K-nearest neighbors. This comparative evaluation serves to establish a reference point for assessing the efficacy of their proposed methodology. The authors executed experiments to evaluate the performance of Convolutional Neural Networks (CNNs) in relation to the aforementioned algorithms, employing metrics such as accuracy, precision, recall, and the area under the Receiver Operating Characteristic (ROC) curve.

Stephan [16] examined "SQL Injection and Its Detection Using Machine Learning Algorithms and BERT." The manuscript advocates for the employment of machine learning strategies to augment the detection capabilities for SQL Injection attacks. The authors contend that the intricate nature of SQL queries, coupled with the imperative for swift detection, renders machine learning an appropriate solution. This observation aligns with a contemporary trend in cybersecurity research, wherein automated systems are progressively utilized to recognize threats. The objective of the paper is to identify a framework that achieves both accuracy and fine-tuning to yield optimal results while evaluating each algorithm against various performance metrics to discern the most effective one. BERT demonstrated superior performance, achieving a validation accuracy of 99.98%.

Abdalla *et al.* [1] engaged in research titled "An Efficient Model to Detect and Prevent SQL Injection Attack." They propose a model designed to detect and prevent SQL injection attacks, which employs runtime validation to ascertain the occurrence of such threats. Their proposed model is characterized by its

adaptability to any existing system, necessitating no alterations to the client or server, nor requiring access to the web application source code. Additionally, the independence of modifications is achieved through the integration of supplementary middleware situated between the client and server. Consequently, all verification processes are executed on this middleware, which functions as a proxy capable of sanitizing inputs for the detection and prevention of SQLIA. Notably, the accuracy of their proposed model reaches 86.6% in detecting and preventing SQLIA.

Obasi and Nlerum [13] developed a model for the Detection and Prevention of Backdoor Attacks using CNN with Federated Learning. The model was trained on a dataset that comprises of 9 classes of MNIST images, of which 8 classes of the dataset were of different classes of backdoor attacks and the class is of non-backdoor attack. The model achieved an accuracy of 99.99% for training and 99.98 for validation.

Hao *et al.* [6] conducted a study on the implementation and research of Deep Learning-Based Detection Technology for SQL Injection. Their research introduces a pioneering SQL injection attack detection strategy that leverages the capabilities of an enhanced TextCNN and Long Short-Term Memory (LSTM) networks, thereby significantly improving the recognition rate of SQL injection attacks while concurrently minimizing both false positive and false negative rates.

Maha *et al.* [8] investigated a Deep Learning Architecture for the Detection of SQL Injection Attacks Utilizing a Recurrent Neural Network Autoencoder Model. Their research proposes a novel architecture aimed at identifying SQL injection attacks through the application of a recurrent neural network autoencoder, exhibiting its efficacy in detecting SQL injection attacks with a superior level of accuracy relative to the alternative models analyzed in the research.

Majid [10] advanced the field by proposing deep learning methodologies for SQL injection detection, specifically assessing Artificial Neural Networks (ANNs), Convolutional Neural Networks (CNNs), and Recurrent Neural Networks (RNNs). His investigation critically appraises the performance metrics of these three predominant neural network configurations for SQL injection attack detection, revealing that the Convolutional Neural Network consistently outperforms the others across nearly all evaluated metrics.

Arzu [4] explored a deep learning methodology grounded in multi-view consensus for the detection of SQL injections. This research implemented an innovative deep learning-based SQL injection detection framework termed "Bidirectional LSTM-CNN based on Multi-View Consensus" (MVC-BiCNN), which incorporates a preprocessing phase that generates multiple perspectives from SQL data by semantically encoding SQL statements into their respective SQL tags.

Ahmed *et al*. [2] conducted an investigation into a Multi-Phase Algorithmic Framework aimed at mitigating SQL Injection Attacks through the utilization of advanced Machine Learning and Deep Learning methodologies to bolster real-time Database security. In this scholarly article, a multi-phase algorithmic framework is delineated, which incorporates enhanced parameterized machine learning and deep learning techniques to fortify database security in real-time at the database tier. The findings illustrate that the proposed approach is capable of preventing SQL Injection; ii) categorizing the type of attack during the detection phase, thereby ensuring the safeguarding of the database.

3. Methodology

This study employs a quantitative research design, utilizing a deep learning approach to detect and prevent SQL injection attacks in cloud databases. A labeled dataset of SQL queries, including benign and malicious queries (SQL attacks and non-SQL attacks) was obtained from an online database, Kaggle.com. The architecture of the proposed system is shown in Figure 1.

Tokenization, normalization, and feature extraction were carried out after obtaining the dataset which includes SQL queries from public sources and cloud database logs. The data was pre-processed by tokenizing SQL queries, removing stop words, and converting to

numerical representation. Tokenization splits SQL queries into individual words or tokens. Normalization transforms tokens into a consistent format such as lowercasing and punctuation removal. A Feed Forward Neural Network (FFNN) algorithm was used to classify SQL queries as benign or malicious (SQL attacks and non-SQL attacks). 80% of the dataset was used for training while 20% was used for testing. The model achieved a training result of about 98% and a test result of about 98%.

The classification report serves as a comprehensive summary of the metrics including accuracy, precision, recall, and f-measure. Precision pertains to the accurate classification of the model as it relates to false positives, false negatives, true positives, and true negatives. The precision score of the model indicates an approximate 100% accuracy in the classification of normal queries and a 94% accuracy in the classification of queries indicative of SQL injection attacks. The model has successfully identified anomalies within database queries and has signaled potential SQL injection threats. Having detected anomalies, Zero-trust principles are enforced based on the output of the deep learning model. The output of the model which is passed through a zero-trust access model determines whether a user will be granted access to a cloud database or not. Zero-trust policy engine can enforce zero-trust principles, such as:

i. Access Control: Deny or grant access to the database.

ii. Multi-Factor Authentication: Require additional authentication factors, such as one-time passwords, secret key to decrypt database information.
iii. Session Termination: Terminate the user session if there is high likelihood of SQL injection attack.

Enforcing zero-trust principles based on the output of the deep learning model, reduces the risk of SQL injection attacks.

4. Results and Discussion

4.1. Phase 1: Exploratory Data Analysis

Exploratory data analysis was carried out on the dataset to visualize data distributions and anomalies, detect anomalies, outliners, and missing values. Figure 2 illustrates a heat map function implemented in Python, which serves to identify the presence of missing values within the dataset. Figure 3 indicates that all missing values have been entirely eliminated from the dataset. In order to facilitate the training process of the data, the datasets depicted in Figures 2 and 3 must undergo tokenization and subsequent conversion into an array format as demonstrated in Figure 4. This objective was accomplished utilizing the CountVectorizer (), in conjunction with the application of stop-words and the tokenizer ().

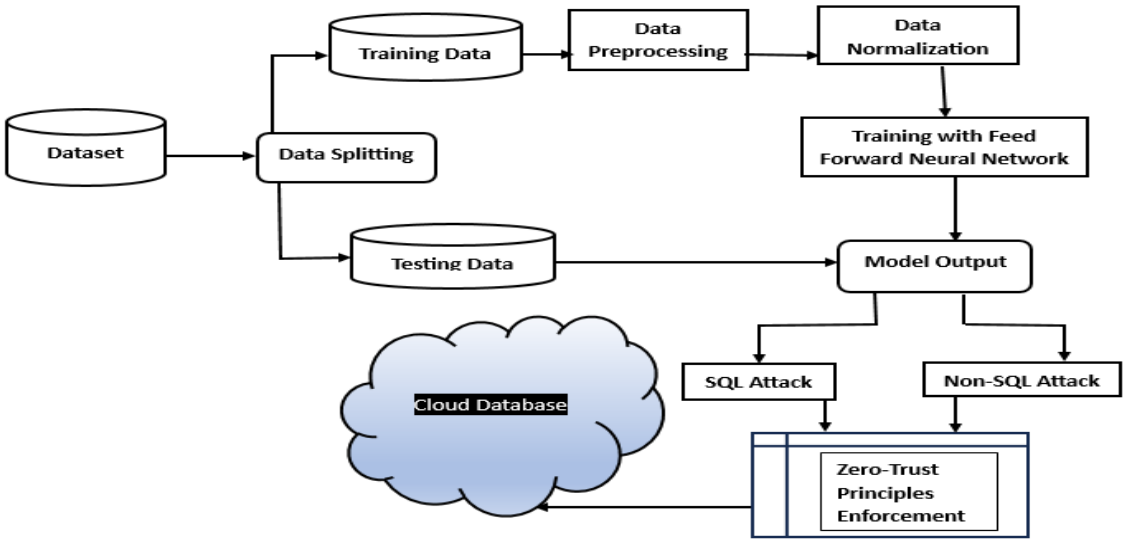

Figure 1: Architecture of the Proposed System.

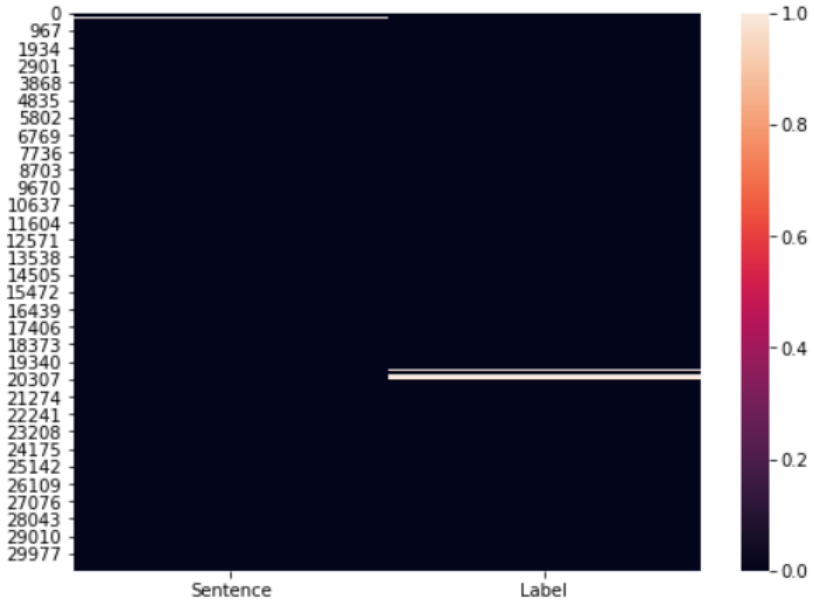

Figure 2: Result of dataset Heat map

Figure 3: Missing values has been removed from the dataset.

	0	1	2	3	4	5	6	7	8	9	...	4707	4708	4709	4710	4711	4712	4713	4714	4715	4716
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

Figure 4: Tokenized and converted data

4.2. Phase 2: Model Training

Upon conducting an exploratory analysis of the dataset, normalization and reshaping of the dataset were performed. The processed data was subsequently partitioned into two distinct subsets. The initial subset constitutes 80% of the dataset, while the subsequent subset comprises the remaining 20% of the dataset. In order to classify the SQL query as either malign or benign, a feedforward neural network algorithm was employed for training. The training procedure of the model is illustrated in Figure 5. Figure 6 presents the accuracy metrics achieved for both the training and validation phases. The training and validation accuracy metrics serve as indicators for evaluating the model's performance during the training phase and on an independent test dataset. The model attained a training accuracy of approximately 98% and a testing accuracy of about 98%. Figure 7 depicts the loss values incurred by the model for both the training and testing datasets. The model exhibited a loss value below 0.1 for both training and testing phases. Figure 8 shows the classification report of the model. Figure 9 shows the confusion matrix of the proposed system. Figure 10 shows normal query where injection is not detected. Figure 11 shows abnormal query where injection is detected. Figure 12 shows an access to a cloud database where zero trust principle is employed.

	Sentence	Label
0	" or pg_sleep (__TIME__) --	1
2	AND 1 = utl_inaddr.get_host_address (...	1
3	select * from users where id = '1' or @ @1 ...	1
4	select * from users where id = 1 or 1#" (...	1
5	select name from syscolumns where id = ...	1
6	select * from users where id = 1 +$+ or 1 =...	1
7	1; (load_file (char (47,101,116,99,47...	1
8	select * from users where id = '1' or ||/1 ...	1
9	select * from users where id = '1' or \.<\ ...	1
10	? or 1 = 1 --	1
11	) or ('a' = 'a	1
12	admin' or 1 = 1#	1
13	select * from users where id = 1 or " ( ]...	1
14	or 1 = 1 --	1
15	AND 1 = utl_inaddr.get_host_address (...	1

Figure 5: Training Process of the Model

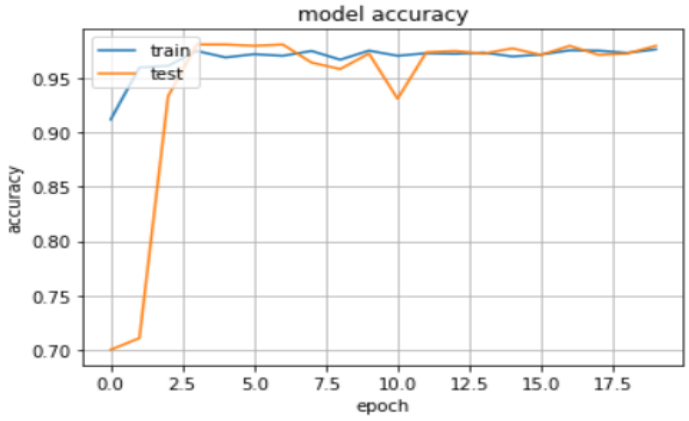

Figure 6: A graphical representation of Training Accuracy Vs Training Epochs.

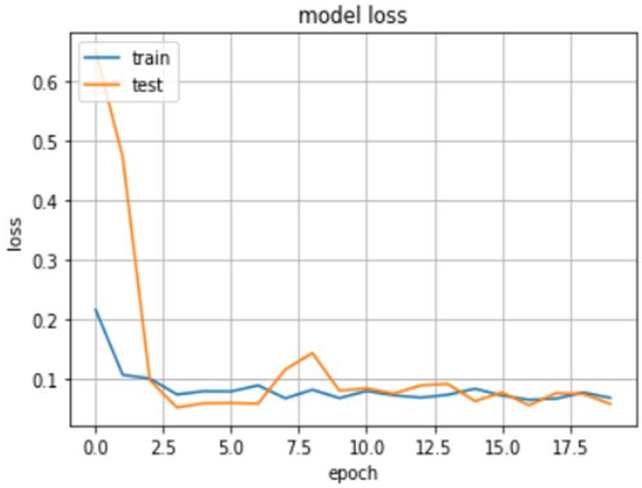

Figure 7: A graphical representation of Training Loss Values Vs Training Epochs.

```
print(classification_report(y_test, pred))
```

	precision	recall	f1-score	support
0	1.00	0.97	0.99	588
1	0.94	1.00	0.97	252
accuracy			0.98	840
macro avg	0.97	0.98	0.98	840
weighted avg	0.98	0.98	0.98	840

Figure 8: Classification report of Deep Learning.

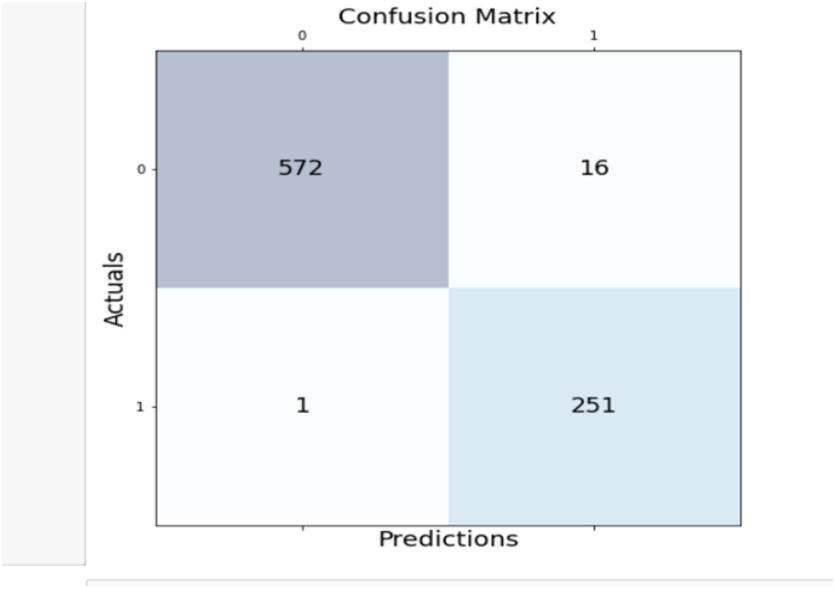

Figure 9: Confusion Matrix of the proposed Feed Forward Neural Network

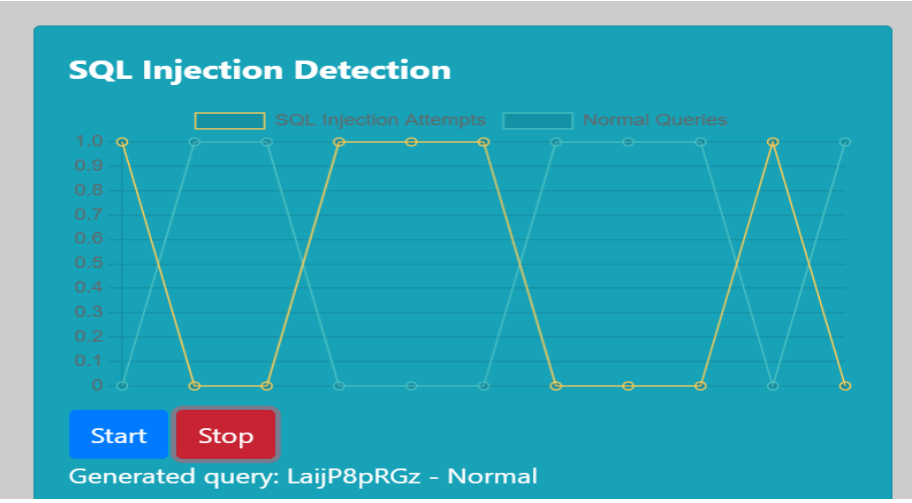

Figure 10: SQL Injection not detected

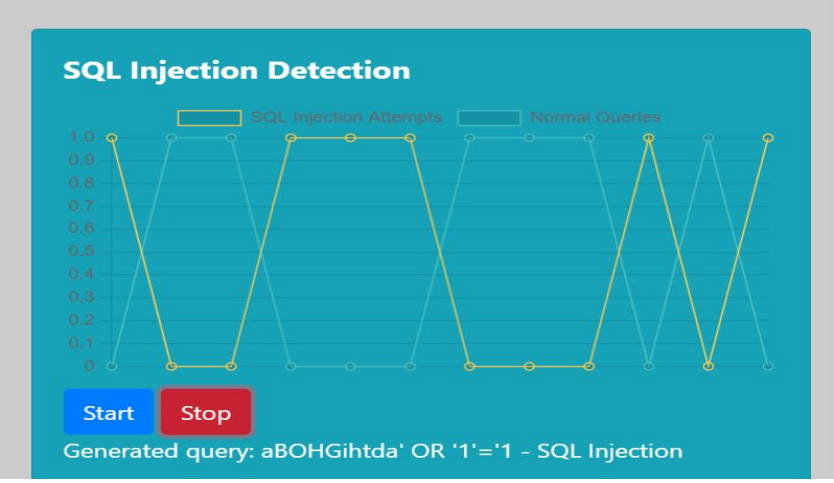

Figure 11: SQL Injection Detected

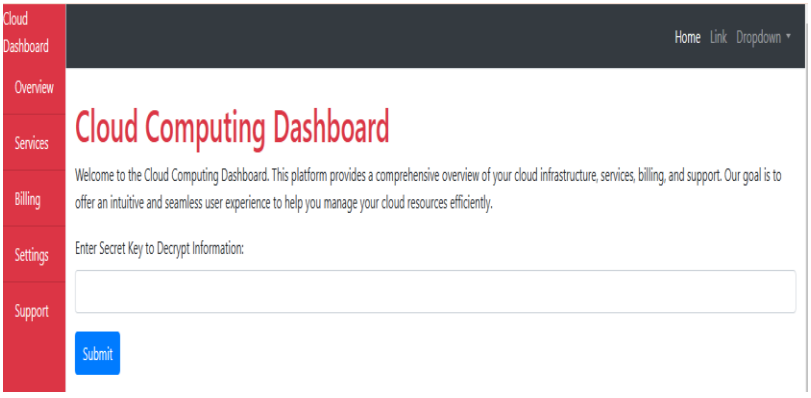

Figure 12: Access to a Cloud Database

4.3. Discussion of Results

From the research conducted, figure 2 shows the heat map function in python which is being used to check for missing values. The white lines in figure 2 shows that some rows in the label column are missing. The thick white line shows that there are some missing values in row 19307 and 20307.

In order to have a well trainable model, the data needs to be cleaned. That is to say that null or missing values, needs to be removed. Figure 3 shows that the missing values in the dataset has been removed completely. After this process, feature extraction was applied on the dataset to select the most important feature. Figure 3 shows that after feature extraction, the most notable features that are suitable for training the deep learning model are the query column and the label column. Before passing the data to the deep learning model, the query column needs to pass through tokenization process. This is to say that the query column needs to be tokenized and converted to arrays.

Figure 4 shows the tokenized and converted data. Tokenization divides text into words, phrases, or symbols, making it easier to process and data conversion transforms tokenized text into numerical representations that deep learning models can process. Figure 5 and 6 shows the accuracy obtained for both training and validation test. The training and validation accuracy are used in testing the performance of the model during training and on a test dataset. The model achieved a training result of about 98% and a test result of about 98%.

Figure 7 shows the losses of the model for both training and testing data. The model had a loss value below 0.1 for both training and testing. Figure 8 shows the classification report of the model. The classification report is a summation of accuracy, precision, recall and f- measure. Precision has to do with the correct classification of the model in terms of false positive, false negative, true positive and true negative. The precision score of the model is about 100% correct classification for queries that are normal and 94% correct classification for queries that are of SQL injection attack. The support shows the total number of classifications that was carried out by the model. Figure 9 shows the confusion matrix of the proposed system. Confusion matrix depicts the total number of correct prediction and the total number of false classifications.

The confusion matrix shows that out of 590 classifications on attacks that are of normal, the model predicted correctly for 572 and predicted falsely for 16 times. Then for attacks that are of SQL injection, the model correctly predicted 251 times and predicted falsely for just 1. This shows the performance of the model is in decent shape. Figure 10 shows that SQL injection is not detected. Figure 11 shows that the query contains SQL injection.

5. Conclusion

Due to the rapid growth of SQL injection attacks on web application, this research developed a deep learning model in detecting SQL injection attack. This paper presents a deep learning algorithm in detecting SQL Injection Attacks on web applications with high accuracy detection rate. The system detects advanced SQL injection (Second Order Attack, and Hybrid Attack). The implementation of this system was carried out beyond analysis and testing of model's performance using test data, but a real time implementation of SQL injection attacks was carried out by creating a web application using Python flask. The system achieved an accuracy rate of 97.65%.

To enhance the efficiency of the system, more SQL statements (both injected and non-injected statements) need to be considered for training and testing our model. The outcomes generated by the model are transmitted through a zero-trust engine to either authorize or restrict access to a cloud-based database. The amalgamation of deep learning-driven detection methodologies with zero-trust principles yields a formidable barrier against SQL injection threats. This introduces an innovative framework for fortifying the security architecture of cloud databases. Our investigation augments the comprehension of SQL injection threats and their detection mechanisms, thereby facilitating the formulation of more efficacious security protocols.

This inquiry may be further expanded through the utilization of hybrid deep learning algorithms. It can also be advanced by integrating the model into Android application environments. Our framework exhibits scalability in that any enhancements can be seamlessly integrated with minimal adjustments.

References

[1] Abdalla, Hadabi., Eltyeb, S., A.,Elsamani.,Ali, E., Abdallah., Rashad, Elhabob. (2022). An Efficient Model to Detect and Prevent SQL Injection Attack.
doi: 10.54388/jkues.v1i2.141.

[2] Ahmed, Abadulla, Ashlam., Atta, Badii., Frederic, T., Stahl. (2022). 2. Multi-Phase Algorithmic Framework to Prevent SQL

Injection Attacks using Improved Machine learning and Deep learning to Enhance Database security in Real-time. doi: 10.1109/SIN56466.2022.9970504.

[3] Ayush, Falor., Manav, Hirani., Henil, Vedant., Priyank, Mehta., Deepa, Krishnan. (2022). A Deep Learning Approach for Detection of SQL Injection Attacks Using Convolutional Neural Networks. 293-304. doi: 10.1007/978-981-16-6285-0_24.

[4] Arzu, Gorgulu, Kakisim. (2024). 1. A deep learning approach based on multi-view consensus for SQL injection detection. International Journal of Information Security. doi: 10.1007/s10207-023-00791-y.

[5] Chen, D.; Yan, Q.; Wu, C.; Zhao, J. SQL Injection Attack Detection and Prevention Techniques Using Deep Learning.(2021). J. Phys. Conf. Ser. 1757, 012055.

[6] Hao, Sun., Yuejin, Du., Qi, Li. (2023). 9. Deep Learning-Based Detection Technology for SQL Injection Research and Implementation. Applied Sciences, doi: 10.3390/app13169466.

[7] Hilmi S. A. & Adnan M.A. (2024). Detection of SQL Injection Attacks based on Supervised Machine Learning Algorithms: A Review. International Journal of Informatics, Information System and Computer Engineering.5(2). 152-165.

[8] Maha, Alghawazi., Daniyal, M., Alghazzawi., Suaad, Alarifi. (2023). 7. Deep Learning Architecture for Detecting SQL Injection Attacks Based on RNN Autoencoder Model. Mathematics. doi: 10.3390/math11153286.

[9] Marina, Volkova., Petr, Chmelar., Lukas, Sobotka. (2019). Machine Learning Blunts the Needle of Advanced SQL Injections. 25(1):23-30. doi:10.13164/MENDEL.2019.1.023.

[10] Majid, Alshammari. (2023). 6. Deep learning approaches to SQL injection detection: evaluating ANNs, CNNs, and RNNs. doi: 10.1117/12.3012620.

[11] Obasi, E. C. M., E., B., & Egbono, F. (2022). Query Processing of Distributed Databases using an Improved GraphQL Model and Random Forest Algorithm. International Journal of Scientific and Research Publications, 12(4), 454. https://doi.org/10.29322/ijsrp.12.04.2022.p12461.

[12] Obasi, E., & Nlerum, P. (2020). Intrusion Detection System for Structured Query Language Injection Attack in E-Commerce Database. International Journal of Scientific and Research Publications, 10(8), 446–453.https://doi.org/10.29322/IJSRP.10.08.2020.P10455.

[13] Obasi, E.C.M. & Nlerum, P.A. (2023). A Model for the Detection and Prevention of Backdoor Attacks using CNN with Federated Learning. University of Ibadan Journal of Science and Logics in ICT Research, 10(1), 9-21.

[14] Patel, A., & Bhattacharya, S. (2019). Deep learning-based SQL injection prevention. International Journal of Intelligent Information Systems, 8(2), 1-12.

[15] Singh, A., & Kumar, P. (2020). Cloud database security challenges. Journal of Cloud Computing, 9(1), 1-14.

[16] Stephan, Ladisch. (2023). SQL Injection and Its Detection Using Machine Learning Algorithms and BERT. Lecture Notes in Computer Science, 3-16. doi: 10.1007/978-3-031-28975-0_1.

[17] Zhang, W.; Li, Y.; Li, X.; Shao, M.; Mi, Y.; Zhang, H.; Zhi, G.(2022).Deep Neural Network-Based SQL Injection Detection Method. Secur. Commun. Netw.4836289