Received: 15 June 2020 | Accepted: 17 December 2020

DOI: 10.1049/sfw2.12018

IET Software

The Institution of Engineering and Technology WILEY

ORIGINAL RESEARCH PAPER

Long short-term memory on abstract syntax tree for SQL injection detection

Z. Zhuo[1] | T. Cai[1] | X. Zhang[2] | F. Lv[2]

[1]Data Security Department, Huawei Technologies, Chengdu, China

[2]Center of Statistical Research and School of Statistics, Southwestern University of Finance and Economics, Chengdu, China

Correspondence

Fengmao Lv, Center of Statistical Research and School of Statistics, Southwestern University of Finance and Economics, Chengdu, China.
Email: fengmaolv@126.com

Funding information

Fundamental Research Funds for the Central Universities, Grant/Award Number: JBK1806002

Abstract

SQL injection attack (SQLIA) is a code injection technique, used to attack data-driven applications by executing malicious SQL statements. Techniques like pattern matching, software testing and grammar analysis etc. are frequently used to prevent such attack. However, major bottlenecks still remain in detecting SQLIA with bypassing techniques, getting access to source code and requiring an additional manual operation to extract features. The authors propose a novel detection approach based on long short-term memory and abstract syntax tree, which could detect SQLIAs from the raw query strings and work under SQL detection bypassing scenario. Our deep learning technique explicitly uses both context and syntax information that previous methods failed to fully grasp. Experimental results clearly illustrate the superior performance of our method compared to other existing works when detecting with complete SQL raw queries.

1 | INTRODUCTION

Many web and mobile applications today involve interacting with back-end databases. Users submit queries through the web or mobile interface and retrieve information from the database. However, such front-end interfaces are vulnerable to SQL injection attacks (SQLIAs) if user input is not validated and sanitized. A specially crafted query could result in an unexpected logic in the database, causing scenarios like data breaches, sensitive information leakage, or even shutting down the database etc. [1].

This attack is widespread and easily exploited, and nearly every database-driven application has to face this common issue. SQLIA was rated the number one attack in the top 10 critical web application vulnerabilities in 2010, 2013 and 2017 by the Open Web Application Security Project (OWASP) [2]. Obviously, SQLIA is no doubt a major security issue that requires our special attention. To overcome this potential threat, many approaches have been proposed, and we will reorganize these approaches and related literature in Section 2. Application-based approaches like Analysis for Monitoring and NEutralizing SQL Injection Attacks (AMNESIA) [3], CANDID [4] and SQLRand [5] require reengineering the SQL-related source code. However, rewriting the code requires lots of effort, which could also lead to other potential bugs. Besides, disclosure of the source code is hard due to the security requirement of the company. IDS/WAF-based techniques like regular expressions, scripting languages or extended Backus–Naur form [6–8] suffer from bypassing techniques listed in [9] and could not match unknown attacks.

More advanced techniques use lexical and syntax analysis, such as Libinjection [10] and SqlChop [11]. However, Libinjection has high false positives and false negatives, while SqlChop has a user-defined threshold which is hard to choose. Machine learning-based approaches [12–15] have their own limitations, such as the inability to detect attacks using injection evasion techniques [16], over-fitting problems [17] and need manual screening to extract features. Moreover, these methods are mostly based on raw query string analysis and could not take advantage of the latest machine learning techniques as well as context and syntax information of available SQL strings.

The authors proposed a novel method for detecting SQLIAs using long short-term memory (LSTM) on abstract syntax tree (AST). We built an SQL query grammar analyser to help us overcome SQL injection bypassing techniques. We then used the word embedding technique in natural language processing to convert the names of lexical and grammar rules to embedding vectors. Unlike Fang et al. [17] which only apply

LSTM to the regular expression-matched SQL keywords, our LSTM is applied on the AST generated by an SQL grammar analyser; thus, our approach could use the SQL context and syntax information to help identify SQLIAs. To sum up, the contributions of this work are mainly threefold:

- We develop an SQL grammar analyser based on Antlr [18] parser generator, which enables us to overcome the SQL injection bypassing issue.
- By designing a deep LSTM neural network on ASTs, we propose an SQLIA detection method that could fully capture context and syntax information from raw SQL queries. Besides, we implement a prototype of the proposed approach.
- We conduct comparative experiments on different methods, and evaluation results show that our technique is effective and superior compared to current methods.

The rest of the study is organized as follows. In Section 2, we summarize existing SQL injection detection methods and compare them with our method. In Section 3, we briefly introduce the background knowledge of SQL injection and LSTM. In Section 4, we describe our proposed method in detail. How to implement the proposed approach is presented in Section 4.2. In Section 5, we show the dataset collection process and present evaluation results. Finally, we discuss our method in Section 6 and draw the conclusion in Section 7.

2 | RELATED WORKS

To combat the SQLIAs, many methods have been proposed in the literature as can be seen from figure 1. In this section, to give a better understanding of the current literature, we present past works from different application levels.

Application level: At this level, defenders need privileges to access and modify application source code or binary executables, and measures taken could get rid of most SQLIAs from the origin. Shar et al. [1] and OWASP's SQL Injection Prevention Cheat Sheet [19] both provide useful programming guidelines to defeat SQLIAs. AMNESIA [3] runs static analysis and runtime monitoring of the application code to examine possible vulnerabilities. The major limitation of this technique is that its success depends on the accuracy of its static analysis for building query models [20]. CANDID [4] is another code analysis approach that could infer the structure of SQL queries that programmers want and evaluate them dynamically. Boyd and Keromytis [5] proposed SQLrand as a protection against SQL injection. They transformed all SQL keywords by pending random strings that might be used in the web application code and stored that mapping. Before any legal SQL statement is executed in the database, it could be translated to its original form using mappings. Any additional SQL provided by the user, such as 'OR 1 = 1', would not match the augmented SQL tokens and would throw an error.

SQLGuard [21] is another novel technique to eliminate SQL injection. Because SQL injection changes the query the programmer wants and, therefore, the tree structure, this method examines the parse tree structure before and after adding user input at runtime to deduce whether an injection occurs. Although the application level defense methods are effective, they still require the source code or the binary. In addition, the related changes in the software could significantly increase the workload.

IDS/WAF level: An intrusion detection system (IDS) and the Web Application Firewall (WAF) are features of the application gateway that provide protection to back-end servers from common exploits and vulnerabilities. SQLIA is one of the common exploits, and detection at this stage would examine network traffic such as HTTP requests. Basically, approaches that occur at this level would attempt to match predefined rules and signatures using regular expressions, scripting languages or extended Backus–Naur Form [6–8]. However, IDS/WAF-based approaches are easily bypassed by obfuscation, such as URL encoding, case changing and buffer overflow etc. [9]. In addition, rules require expert knowledge and manual efforts, which are labour intensive. To overcome these shortages, some WAF implementations recruit deep learning [22], lexical and syntax analysis techniques [10,11]. Libinjection [10], for instance, converts the SQL query to a token string and then matches it to the signature library. Based on the idea of lexical analysis technique used in Libinjection, SqlChop [11] further added a syntax analyser, and it used a user-defined threshold to detect SQLIAs. Similar to their idea, we also adopted the idea of lexical and syntax analysis.

Query-based level: Query-based approach aims to infer SQLIAs from the raw SQL queries submitted to the database. The difference between IDS/WAF level and this layer is that IDS/WAF could see the user requests but has no idea what is actually submitted to back-end databases. On the other hand, techniques from the IDS/WAF level could be applied to this level, such as regular expression matching. Liu et al. [23] came up with SQLProb. SQLProb defines a set of all possible queries generated by a web application. User input is separated by a proxy between the web application and the database. Based on the query parse tree and the user input, a non-leaf node contains not only the user input but also control nodes, the query may be malicious. SQLProb has a high detection rate, but the time complexity of determining user input is also high, and it is hard to get all query structures of a web application. We believe that their method could be further improved when cooperating with WAF/IDS layer, the high-level idea is illustrated in Section 6.

Machine learning-based approaches showed their strength in the above two layers of SQLIAs detection as well [12,3]. Skaruz and Seredynski [24] presented an RNN neural network to detect SQLIAs, and they showed that the Jordan network is much better than the Elman network in the classification task. Based on the work of Mou et al. [14], Yu et al. [16] proposed a tree-based convolutional network on SQL parse tree to detect SQLIAs. They showed a high accuracy of 94.7%, and their approach functioned properly when testing with invisible attacks and attacks using evasion techniques. They implemented their approach based on PostGresSQL database query logs. Unlike us, they relied on the database itself to parse the query which could pose a high overhead for the database. Fang et al.

[17] used SQL keyword tokens and LSTM to detect the SQLIAs. However, their method used regular expressions to convert the SQL keywords into word tokens, which could be bypassed easily. McWhirter et al. [15] came up with a gap-weighted string subsequence kernel to extract features from the raw query sequence and fed it in an SVM classifier. However, the time complexity of their method is high. Sconzo [25] presented a random forest classifier which used only four features, namely the query length, entropy, legitimacy and malicious G-test value. The presented results are promising, but the G statistical test value depends on the specific dataset, which is not suitable for practical situations.

3 | PRELIMINARIES

3.1 | SQL injection

SQL injection occurs when an attacker changes the intended logic of an SQL query by inserting a new SQL query language into the original query. The formal definition of SQLIA is given in [26]. As summarized by Halfond et al. [20], mainly six types of SQLIAs exist known to date, each has its own attack goals or intents. For simplicity reason, we do not list the example of each SQLIA (refer to [20] for deeper understanding).

Tautologies: This kind of SQLIA generally injects codes in conditional statements (WHERE statement) such that the condition would be always true, such as 'or 1 = 1'.

Illegal/logically incorrect queries: It injects SQL code which would cause syntax, type and logical errors into the database; corresponding returned error message allows attackers to gain information about the type and structure of the database.

Union queries: In union-query attacks, attackers could carefully craft and insert a union select query to the original query. Because attackers could control the injected query, they can choose what information to retrieve from the database.

Piggy-backed queries: In this attack type, attackers do not modify the intent of the original query, instead, they attend to new distinct queries that concatenate after the original query, as several queries together summited to the database.

Stored procedures: Many database management systems ship with stored procedures, and customers could also add their own procedures. It is possible that attackers could use SQLIAs to call those stored procedures to run arbitrary codes, such as shutdown the database.

Inference: This kind of SQLIA is often used when no usable feedback is returned from the database for attackers to determine the vulnerable parameters (injection points). To do this, the attackers change the original query into a true/false question about a data value in the database and observe how the web application behaves after modifications. Two types of inference-based methods are used widely, one is called blind injections and the other is timing-based approach. Basically, blind injections get inference from the web page differences, and the timing-based methods mainly observe the response time delay from the database.

3.2 | Recurrent neural networks

Since the first introduction of recurrent neural network (RNN), it has been widely used for other applications such as image captioning, language translation and music generation. Unlike the traditional feed-forward network, RNN could perform the present task using recent information. For example, given a sentence of words, RNN could predict the current word using previous words in the same sentence. However, it is hard for RNN to preserve the long-term dependency of the context [27]. In SQLIA scenario, attackers could insert a union select query followed by a long normal look-like sequence. RNN might forget the union select keywords in the end. Compared to RNN, LSTM [28] is a particular type of RNN; it has contextual state cells that could modulate the output not only from the long-term historical context of inputs but also from the very last input.

The main structure of LSTM is shown in Figure 2, where C_{t-1} and C_t are the old and current cell state, respectively; h_{t-1} and h_t are the old and current hidden state, respectively. The key reason why LSTM could remember long-term dependency and discard non-relevant information is twofold: firstly, the forget gate controls what information to discard and secondly the input gate chooses what new information to store in the new cell state, illustrated as follows:

$$C_t = f_t^* C_{t-1} + i_t * C_t^*, \tag{1}$$

where $f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$, $i_t = \sigma(W_i * [h_{t-1}, x_t] + b_i)$ and $C_t^* = tanh(W_c * [h_{t-1}, x_t] + b_c)$.

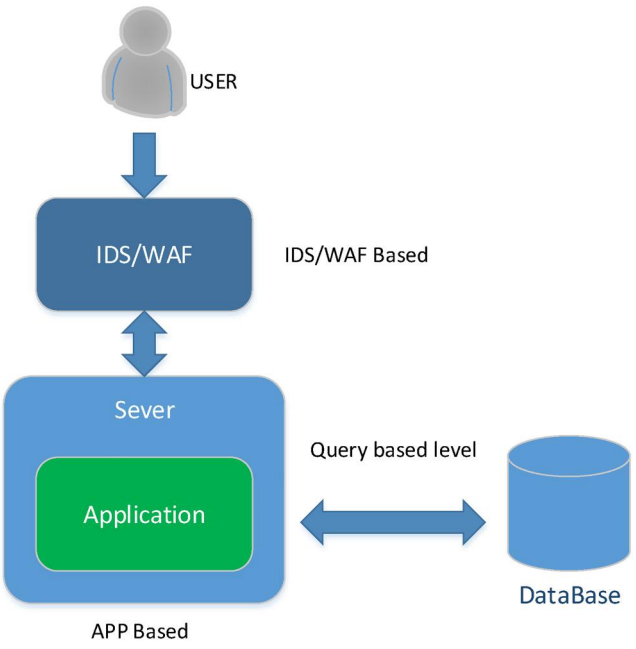

FIGURE 1 SQL defense in different levels

4 | METHODOLOGY

4.1 | Overview

We assume that our approach can only see the raw queries between the front application and back-end databases, as shown in Figure 3. Obviously, although some SQLIAs could use bypassing techniques to get through the IDS/WAF defense layer, the real SQL queries submitted to the database could be seen by our approach. However, this advantage also brings us challenges. Since the only information, we could get is the raw SQL queries and nothing more. It is really hard to detect SQLIAs using limited resources.

As we could see from Figure 3, our approach captures SQL queries submitted to the databases from network traffic. After we filter out SQL queries, to get more information from these query sequences, we construct an SQL grammar analyser to convert them into ASTs. The nodes in AST are predefined grammar rules and lexical symbols. We utilize the depth-first search (DFS) method to traverse the AST and convert the tree into a symbol sequence. Using DFS, we could therefore preserve the context information of the AST tree. Then, we use the word embedding technique to learn the word vector for each symbol presented in the AST. Word embedding techniques could transform each symbol in the tree to a vector representation, such that symbols with similar syntax information would have similar word embedding vector.

After we get the word embedding matrix for each symbol, we feed them into the designed LSTM network. And finally, the output is given by the LSTM network. The overall procedure of our method is depicted in Figure 5. The intuitive reason for using LSTM in our approach could be illustrated using an example in Figure 4. The red dot line area in the picture could reveal the sequence syntax pattern for tautologies SQLIA. Also, LSTM could choose to ignore the historical pattern in the AST that does not contribute to tautologies SQLIA. If an attacker performs an SQLIA, we could therefore judge from the important sequence syntax pattern learned by the LSTM.

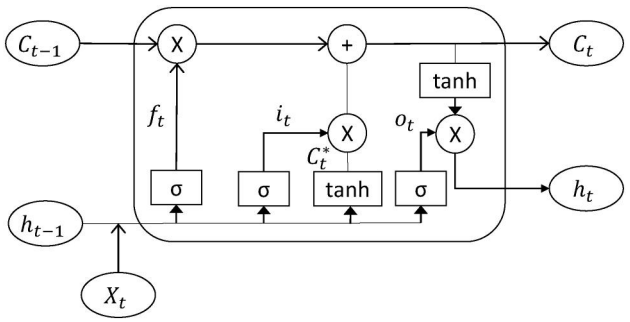

FIGURE 2 LSTM network structure. The *circle* in the figure means pairwise multiply ("X") or add ("+") operation

4.2 | Approach

SQL grammar parser: We implemented our prototype as a subsystem for Huawei's database security service (DBSS) [29]. DBSS is a smart protection service for cloud databases. This service can provide functions such as sensitive data discovery, data masking, database auditing and injection prevention. Our prototype passively monitored the SQL queries as shown in Figure 3. We implemented an SQL grammar parser based on Antlr4; currently, we only realized an SQL grammar parser for MySQL. We believe other databases could also use our proposed method. We used a modified version of [30] as our MySQL grammar for parsing SQL queries into AST. Because the original files provided by [30] have some grammar issues unsolved.

We try to illustrate our technique using an example. Suppose the given query is:

SELECT username FROM accounts WHERE username='' or 1=1 #' AND password=''.

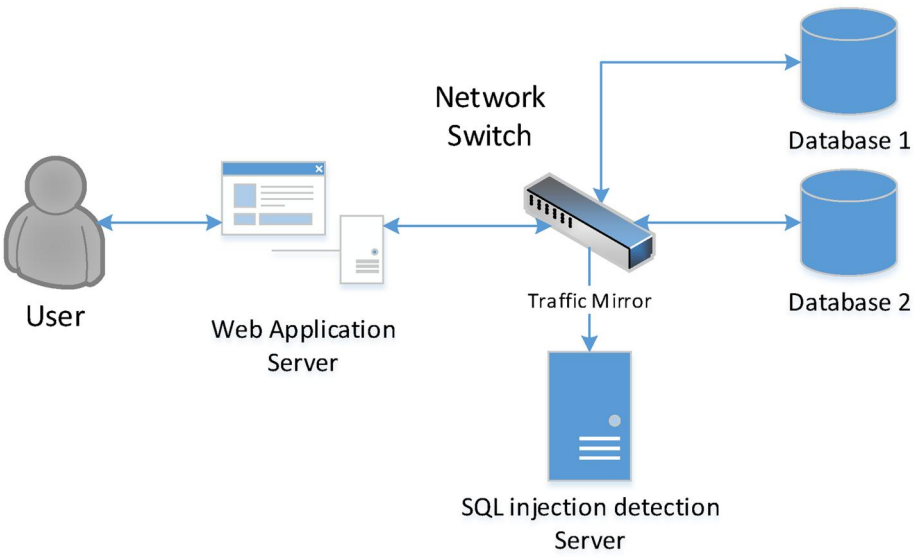

FIGURE 3 The system architecture overview

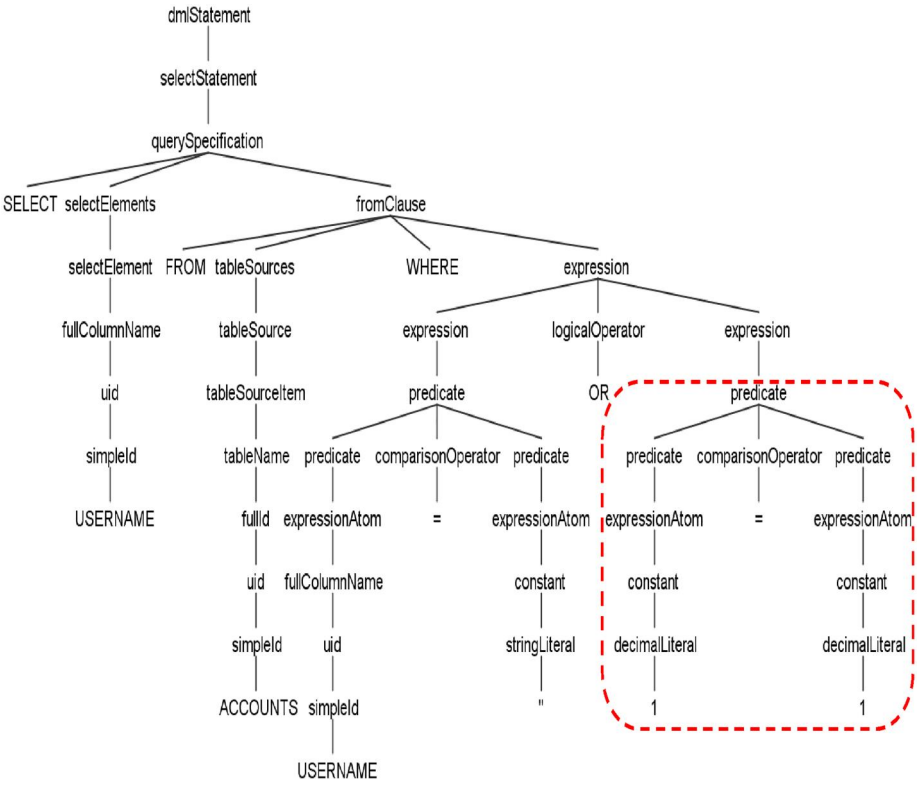

FIGURE 4 The abstract syntax tree for SELECT username FROM accounts WHERE username = '' or 1 = 1 #' AND password = '', notice that the statement after the comment is filtered by our grammar parser

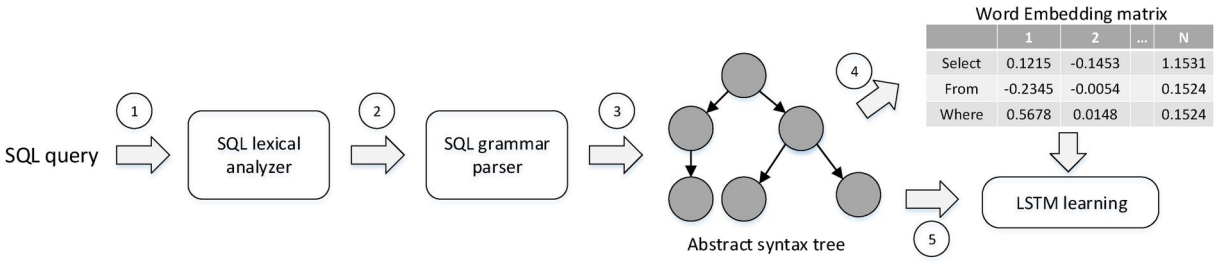

FIGURE 5 The overall procedure of the proposed approach

As shown in Figure 4, this is a typical SQLIA, whose '#' symbol comments out the query behind it. As we can see from Figure 4, the capital words are terminal nodes recognized by the lexer. The non-capital words are syntax rules' names recognized by the parser. Notice that the comment does not show in the AST, this is because the lexer in Antlr put all the non-related symbols in a hidden-channel; in other words, the lexer filters out non-related symbols, such as comment. That is why our proposed method could still work for SQLIAs with bypassing techniques.

Word embedding: Given the AST, we then used the DFS algorithm to traverse the tree, thus converting the AST to a symbol sequence. Using DFS, we could preserve the context information of the syntax tree. For AST shown in Figure 4, this would be ['dmlStatement', 'selectStatement', 'querySpecification', 'SELECT', …, 'decimalLiteral', 'DECIMAL_LITERAL'].

Suppose an SQL query set $Q = \{q_1, q_2, \ldots, q_n\}$ has n SQL queries, and it contains K unique symbols. We utilized a continuous bag of words (CBOW) as the word embedding method to learn the word embedding vector for each unique symbol. The reason why we chose CBOW is that we think the symbol in the sequence could be predicted with probabilities using other symbols around it. We also evaluated the default word embedding method implemented by Keras, and the results are shown in Section 5. As shown in Figure 5 step 4, each symbol in the AST is transformed to word vector, by stacking them together we have the word embedding matrix.

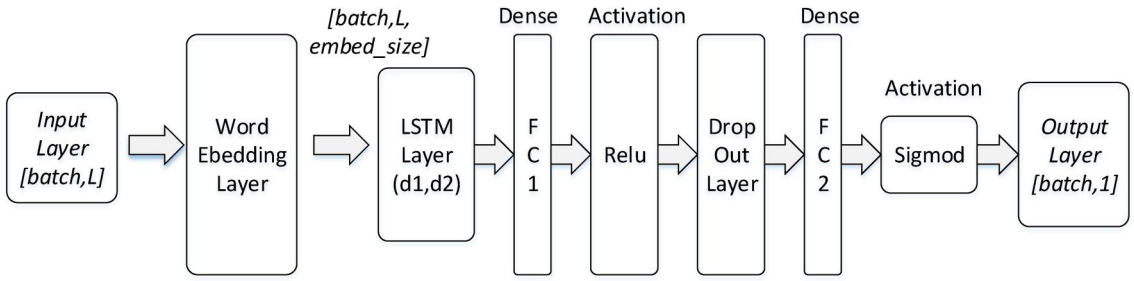

FIGURE 6 The designed LSTM network for SQLIAs detection

TABLE 1 LSTM network hyperparameters used for SQLIAs detection

Layer	Details
Input layer	Length: L
Embedding layer	Size: 1024, Method:default
LSTM layer	Units: 256, d1 = 0.2, d2 = 0.2
Fully connected (FC1)	Units: 100, Activation: Relu
Dropout layer	Drop rate: 0.5
Fully connected (FC2)	Units: 1, activation: Sigmod

Building LSTM network: We used Keras [31] to construct the LSTM network. The designed LSTM network for SQLIAs detection is depicted in Figure 6. The detailed hyperparameters for our designed model are listed in Table 1. We assume that the maximum length of the symbol sequence is L in our training set. For batch training convenience, for each SQL query, we padded it to the length L and fed it into the LSTM network. Note that we added a dropout layer in the network, and we also trained our LSTM network with two other dropout hyperparameters, namely d1 for LSTM input gate dropout and d2 for recurrent connection dropout. These setups would help us to prevent over-fitting problems.

5 | EXPERIMENTS

5.1 | Dataset collection and sanitation

To evaluate the performance of our proposed method, we manually collected normal SQL query strings and SQLIA queries from different sources. Firstly, we took the open-source SQL testing queries from GitHub, such as [30] and we also used none-sensitive SQL testing samples from the HexaTier company. Secondly, for malicious SQL queries, we built two well-known vulnerable web testing applications: DVWA [32] and Mutillidae [33]. Both DVWA and Mutillidae are open-sourced PHP/MySQL web applications for research purpose. To collect the raw string for SQLIA, we used the setup shown in Figure 7. We also wrote an SQLIA queries generation application in python, which randomly concatenates the normal query with SQL injection payloads collected from SQLmap and online sources. The generated SQL queries which passed the grammar validation test were used for our study. Since our approach can see only the raw SQL strings, SQL injection strings from HTTP requests cannot be directly used. To collect the real SQLs executed by the database, we logged all the successful queries executed by the database, as depicted in Figure 7.

We took all five SQL injection techniques provided by SQLmap to scan injection points. In total, we have collected 33,029 queries; however, we later found that many queries could not be used. Because many SQLmap generated queries are used to find out where the injection point is and to determine whether the injection point is a false positive. For instance, SQLmap generated a query:

SELECT first_name, last_name FROM users
WHERE user_id = '-4071) OR 5920=5920\#'.

In this example, the comment is inside the single quote; thus, the raw sequence after 'id=' is, however, be treated as the symbol: 'STRING_LITERAL' in our grammar parser, which is the same as what ended in the normal query. In this case, we treat samples like this as normal queries and filter out such SQLmap-generated queries.

We built two datasets as listed in Table 2, where SQL-A is a subset of SQL-B. We first ran our model on a smaller dataset (SQL-A) to get a first impression of our proposed method's performance and to optimize the model's hyperparameters. To further demonstrate the validness of our method, we built SQL-B on top of SQL-A. After filtering and removing duplications, the SQL-A dataset totally contains 5258 queries (500 generated queries[1]), together we had 2620 legit queries and 2638 malicious queries. As for SQL-B, it contains 20,334 unique queries (5000 generated queries), including 10,060 normal queries and 10,273 malicious queries. Note that the generated queries are obtained by concatenating real SQLIA samples collected from github.

5.2 | Experimental steps

1) *LSTM network training*: The dataset for SQLIA is hard to collect, and to evaluate more data, we randomly shuffled the dataset and split all samples into training (50%) and testing (50%). Thus, the training and testing size for SQL-A is 2629 and 10,167 for dataset SQL-B. Throughout our SQLIA detection evaluation process, each time we

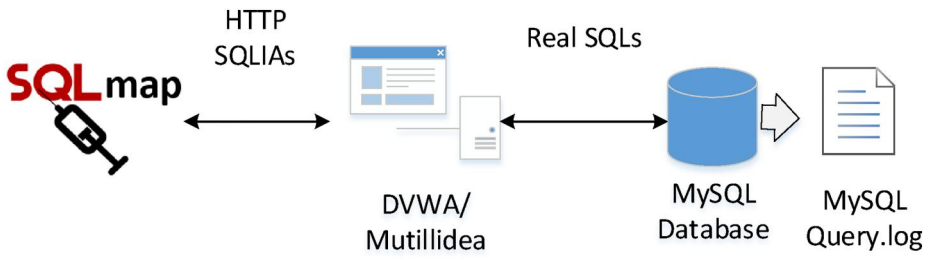

FIGURE 7 The setup for DVWA/Mutillidae SQL injection string collection

TABLE 2 Datasets used in our experiments

Name	Description	SQLmap	Gen
SQL-A	Performance first impression	4758	500
SQL-B	More data evaluation	15,334	5000

randomly shuffled and split the dataset based on different random states, and unless otherwise specified, we employed the same training and testing samples when comparing our method with other methods.

2) *Comparison study*: For a better understanding of our method and its performance, we choose three representative methods listed below for comparison study:
   - Machine learning-based method: We used an open-source random forest classifier proposed by Sconzo [25] as an example.
   - Pattern matching-based method: we took a python version of Libinjection [10], based on Libinjection version 3.1 released in May 2017. A lot of WAFs use this library instead of regular expressions because of the performance.
   - Syntax analysis method: We took an alpha version (0.6.1) of SqlChop [11] for our study. SqlChop is a novel SQL injection detection engine built on top of SQL tokenizing and syntax analysis.
3) *Define test metrics*: To compare the experimental results, we define the comparing metrics as follows. The overall accuracy (ACC) determines the general quality of the detection result. It is calculated as the ratio of the number of correctly identified queries to the total number of the query strings in the same testing set. True positive (TP) is the number that an SQL injection query is correctly identified as SQLIA. False negative (FN) is the number that an SQL injection query is incorrectly classified as a legit query. False positive (FP) is the number that a legit query is incorrectly classified as a malicious query. True negative (TN) means that an SQL injection query is correctly classified as a malicious query. True-positive rate (TPR): the probability that an SQL injection query is classified as malicious. False-positive rate (FPR): the probability that a normal query is incorrectly classified as an SQL injection query. False-negative rate (FNR): the probability that an SQL injection query is incorrectly classified as a normal query. TPR, FPR and FNR are formally defined as follows:

TABLE 3 Word embedding methods comparison results on dataset SQL-A

Method	TN	FP	FN	TP	TPR (%)	FPR (%)
Default	1314	17	3	1295	**99.77**	**1.28**
CBOW	1299	32	15	1283	98.84	2.40

TABLE 4 Evaluation results on dataset SQL-A

Method	TN	FP	FN	TP	TPR (%)	FPR (%)
LSTM	1314	17	3	1295	**99.77**	**1.28**
RF	1262	69	131	1167	89.91	5.18
LibI	1066	265	455	843	64.95	19.91
SC	466	865	27	1271	97.92	64.99

$$\begin{aligned} TPR &= TP/(TP+FN), \\ FPR &= FP/(FP+TN), \\ FNR &= FN/(TP+FN). \end{aligned} \tag{2}$$

5.3 | Results

LSTM network optimization: To understand which hyperparameters could contribute to our classification results. We conducted the following studies:

- How the word embedding vector size influences the classification result?
- How the word embedding method influences the classification result?
- How the LSTM network unit size and LSTM network dropout influence the classification result?

To determine the hyperparameters for our LSTM network, we tried various values to optimize the model performance. We used an 8-core (E5-2690 v4) and 16-GB memory virtual machine to run our evaluation. During the evaluation process, we fixed the test and training sets and other parameters (only change one parameter during each run). For word embedding size, we tried {64, 128, 256, 512, 1024} and 1024 gave us the best results. For LSTM network unit size, we tried {64, 128, 256, 512, 1024} and 256 gave us

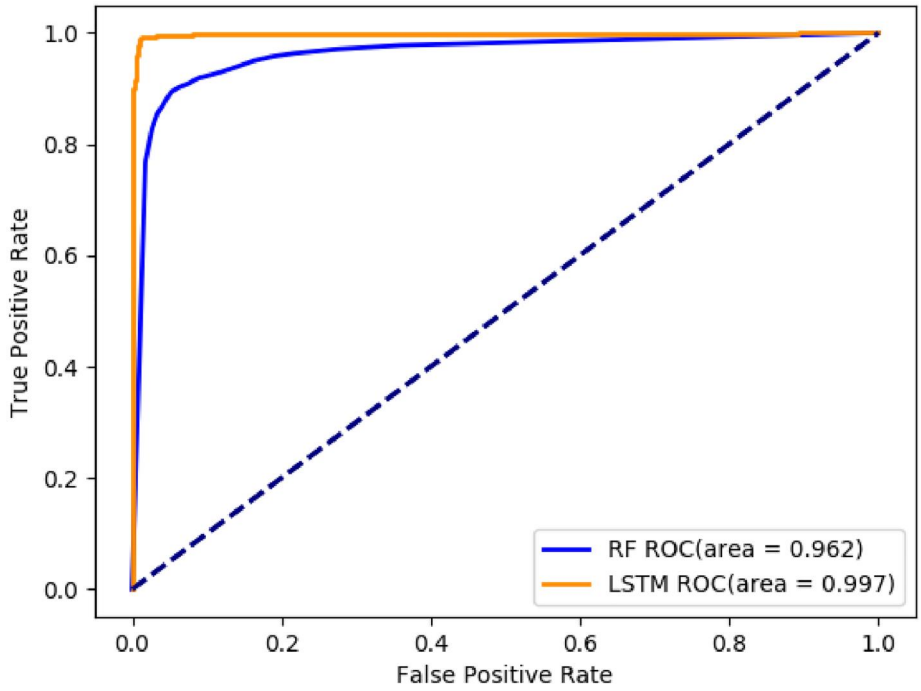

FIGURE 8 Random forest (RF) classifier ROC curve and LSTM network ROC curve, evaluated on SQL-A

TABLE 5 Evaluation results on dataset SQL-B

Method	TN	FP	FN	TP	TPR (%)	FPR (%)
LSTM(1)	4983	16	2	5166	**99.96**	0.32
RF(1)	4714	285	288	4880	94.43	5.70
LibI(1)	3992	1007	1771	3397	65.73	20.14
SC(1)	1784	3215	82	5086	98.41	64.31
GRU (1)	4985	14	42	5126	99.19	**0.28**
LSTM(2)	5130	13	2	5022	**99.88**	**0.77**
RF(2)	4853	290	278	4746	94.47	5.64
LibI(2)	4074	1069	1685	3339	66.46	20.79
SC(2)	1911	3232	89	4935	98.23	62.84

TABLE 6 SQLchop evaluation results using different thresholds

Threshold	TN	FP	FN	TP	TPR (%)	FPR (%)
1.5	1330	3669	31	5137	99.40	73.39
2.5	3241	1758	161	5007	96.88	35.17
3.0	4209	790	192	4976	96.28	15.80
3.5	4737	262	1036	4132	79.95	5.24

the best results. For dropout rates d1 and d2, we both tried {0.2, 0.4, 0.6} and our best results occurred with d1 = 0.2 and d2 = 0.2. For the word embedding method, we found that the default embedding layer provided by Keras performs slightly better than CBOW, as listed in Table 3. We show the values of these optimized parameters and other relevant settings of the network in Table 1.

SQLIA detection: We used the optimized hyper-parameters shown in Table 1 through the rest of the evaluation process. The evaluation results on dataset SQL-A are listed in Table 4. The vocabulary size for SQL-A and SQL-B is 632 and the maximum length L for SQL-A is 1261, and for SQL-B is 1737, respectively. As can be seen from Table 1, our method performs better than other methods (ACC = 99.23%), with a higher TPR, a lower FRP and the FNR is 0.23%. In another run, the ROC curve for our method and RF is shown in Figure 8; it also demonstrates that our method performs better than the RF classifier. Note that for the comparison test, we do not modify the default configuration of Libinjection (LibI) and SQLchop (SC). The default threshold for SQLchop to report an SQLIA is hard-coded to 2.1 in its alpha version.

To fully discover the deep learning's potential and to see how our method performance change when providing more data, we constructed the SQL-B dataset based on the SQL-A. We ran five different random tests based on different random states. For space limitation, we only show the first two test results, which are denoted as (1) and (2) in Table 5.

Since each run the split is random, so the total number of the TN and the TP is different. The results demonstrate that our LSTM method performs better than other methods. Also, we found that LibInjection has a lower TPR than SQLchop, but SC has a higher FPR in default settings. This is because the default threshold score is not suitable in our scenario. As can be seen from Table 6, we set different thresholds to test SQLchop using the same testing data as SC(1) in Table 5. The results indicate that after increasing the threshold to around 3, SC achieves a better performance than LibI(1). However, it is evident that both SQLchop and Libinjection have a high FPR

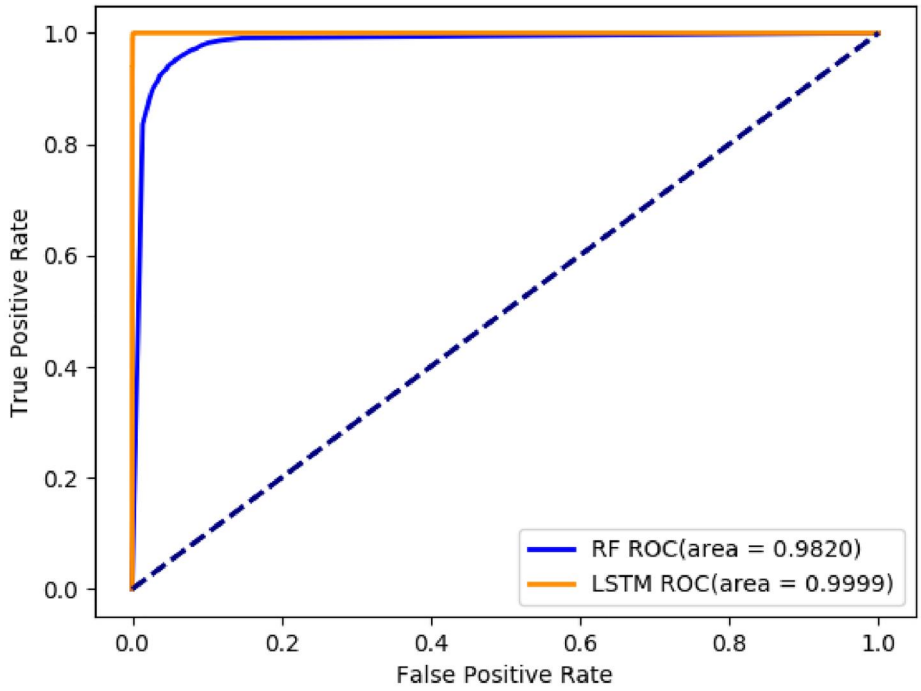

FIGURE 9 Random forest (RF) classifier ROC curve and LSTM network ROC curve, evaluated on SQL-B

in SQLIA detection when supplied with the full SQL query as input. The ROC curve shown in Figure 9 also testifies that our method is more accurate than the traditional random forest SQLIA classifier. The reason why our method could perform better than the SQLchop and Libinjection is that, they are both designed for SQLIA payloads detection, while in our case, we used full SQL queries. In other words, SQLchop is optimized for WAF level SQLIA detection, not suitable in our case. And the threshold for SQLchop is hard to determine. In addition, regular expression matching used by Libinjection could result in high false negatives. For the RF classifier, it only used four manually extracted features which are not able to fully capture the essence of SQLIAs.

We also tested on another kind of RNN, such as gated recurrent unit (GRU). GRU is similar to LSIM, but it has fewer parameters to train; therefore, it has higher computational efficiency. Our designed LSTM network took about half an hour to run one epoch; however, GRU took 20 minutes to finish one epoch. Hence, GRU can be usually treated as a simpler implementation of LSTM. As shown in Table 5, GRU can obtain similar performance to LSTM.

6 | DISCUSSION

Here, we focused on detecting the SQLIA from the raw SQL query string. We proposed a deep learning-based approach (LSTM network) to address this problem. We compared our method with some WAF level approaches: SQLchop and Libinjection, and those methods could work when provided with a partial SQL query, that is, the payload part of SQLIA. However, our method relies on the full query. We believe if the WAF layer could provide us with the user input for one specific query, we could utilize the AST to identify the SQLIA similar to Liu's idea [23]. If so, the key issue is to correlate the user input (http requests) with SQL queries generated by web applications. In addition, we only tested our proposed approach on MySQL; however, our idea could be easily expanded to other databases as well. Also, to the best of our knowledge, to further understand and interpret LSTM results internally is a common unsolved problem in deep learning.

However, our approach can obtain a relatively unacceptable FPR in some cases, for example, the FPR value is over one per cent for SQL-A. This can make our approach unusable in some sorts of realistic contexts. Our future works will focus on how to further reduce the FPR value while keeping TPR unaffected.

7 | CONCLUSION

Here, we present an LSTM network to identify SQLIA from the raw SQL query string. Unlike current approaches, our method does not extract features manually and does not need access to web application source code. Our technique took both context and syntax information from SQL into consideration; thus, it could work in situations like SQLIA with bypassing techniques. Furthermore, our technique is trained with the deep learning method; it does not need a pattern library. Therefore, it could also detect previous unseen SQLIAs. We evaluated our method on two synthetic datasets. Experimental results suggest that our method is more accurate than the existing approaches (random forest classifier, Libinjection and SQLchop) in full string SQLIA detection. Finally, our

method is easy to deploy and can protect the back-end database without any modifications to front-end web software or architecture.

ACKNOWLEDGEMENTS

This work is supported by the Fundamental Research Funds for the Central Universities of China (No. JBK1806002). The authors would like to thank the anonymous reviewers for their careful reading of this article and for their helpful and constructive comments.

ORCID

F. Lv https://orcid.org/0000-0003-1640-0992

REFERENCES

1. Shar, L.K., Tan, H.B.K.: Defeating SQL injection. Computer. 46(3), 69–77 (2013)
2. Owasp top ten project. (2018) https://owasp.org/www-_project-_top-_ten/
3. Halfond, W.G., Orso, A.: AMNESIA: analysis and monitoring for neutralizing SQL -injection attacks. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 174–183.ACM (2005)
4. Bandhakavi, S., et al.: Candid: preventing SQL injection attacks using dynamic candidate evaluations. In: Proceedings of the 14th ACM conference on computer and communications security, pp. 12–24, Virgina ACM (2007)
5. Boyd, S.W., Keromytis, A.D.: SQLrand: Preventing SQL injection attacks. In: International conference on applied cryptography and network security, pp. 292–302.Springer, Berlin (2004)
6. Modsecurity: Open source web application firewall. (2018). https://www.modsecurity.org/
7. Kemalis, K., Tzouramanis, T.: SQL-IDS: a specification-based approach for SQL-injection detection. In: Proceedings of the 2008 ACM symposium on applied computing, pp. 2153–2158.(2008)
8. Alnabulsi, H., Islam, M.R., Mamun, Q.: Detecting SQL injection attacks using snort ids. In: IEEE Asia-Pacific World Congress on computer science and engineering (APWC on CSE 2014), pp. 1–7.(2014)
9. OWASP: SQL injection bypassing WAF. (2018). https://www.owasp.org/index.php/SQL_Injection_Bypassing_WAF
10. Galbreath, N.: Libinjection, a SQL injection detection engine. (2018). SQL/SQLItokenizerparseranalyzer
11. Chaitin: A SQL injection detection engine. (2018). https://sqlchop.chaitin.cn/
12. Rawat, R., Shrivastav, S.K.: SQL injection attack detection using SVM. Int. J. Comput. Appl. 42(13), 1–4 (2012)
13. Moosa, A.: Artificial neural network based web application firewall for SQL injection, World Acad. Sci. Eng. Technol. 40, pp. 12–21, Paris (2010)
14. Mou, L., et al.: Convolutional neural networks over tree structures for programming language processing. AAAI. 2. p. 4 (2016)
15. McWhirter, P.R., et al.: SQL injection attack classification through the feature extraction of SQL query strings using a gap-weighted string subsequence kernel. J. Inf. Security Appl. 40, 199–216 (2018)
16. Yu, P., Wang, S.: Deep tree: sql injection detection by the power of deep learning. (2018). https://github.com/Aetf/tensorflow-tbcnn/blob/master/misc/deeptree.pdf.
17. Fang, Y., et al.: WOVSQLI: Detection of SQL injection behaviors using word vector and LSTM. In: Proceedings of the 2nd international conference on cryptography, security and privacy, pp. 170–174. ACM (2018)'
18. Parr, T., Fisher, K.: Ll (*): The foundation of the ANTLR parser generator. ACM Sigplan Notices. 46(6), 425–436 (2011)
19. SQL injection prevention cheat sheet. (2018). https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
20. Halfond, W.G., et al.: A classification of SQL-injection attacks and countermeasures. Proc. IEEE Intl. Symp. Secure Software Eng., vol. 1, pp. 13–15.(2006)
21. Buehrer, G., Weide, B.W., Sivilotti, P.A.: Using parse tree validation to prevent SQL injection attacks. In: Proceedings of the 5th international workshop on software engineering and middleware, pp. 106–113. ACM (2005)
22. Sergiodfdez: WAF-brain: The clever and efficient firewall for the web. (2018). https://github.com/BBVA/waf-_brain
23. Liu, A., Yuan, Y., Wijesekera, D.: StavrouA. Sqlprob: a proxy-based architecture towards preventing SQL injection attacks. In: Proceedings of the 2009 ACM symposium on applied computing, pp. 2054–2061. (2009)
24. Skaruz, J., Seredynski, F.: Recurrent neural networks towards detection of SQL attacks. In: IEEE international parallel and distributed processing symposium 2007. IPDPS 2007, pp. 1–8.(2007)
25. Sconzo, M.: SQL injection exercise (2014) https://nbviewer.jupyter.org/github/ClickSecurity/data_hacking/blob/master/sql_injection/sql_injection.ipynb
26. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications ACM SIGPLAN Notices, 41, pp. 372–382.ACM (2006)
27. Bengio, Y., Simard, P., Frasconi, P.: Learning long-term dependencies with gradient descent is difficult. IEEE Trans. Neural Netw. 5(2), 157–166 (1994)
28. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. 9(8), 1735–1780 (1997)
29. Cooperation, H.: Database security service (2019) https://www.huaweicloud.com/en-_us/product/dbss.html
30. Technologies, P.: Antlrv4 mysql grammar. (2019) https://github.com/antlr/grammars-_v4/tree/master/mysql
31. Chollet, F., et al.: Keras. (2015) https://keras.io
32. Team, D: Damn vulnerable web application (dvwa) (2019) http://www.dvwa.co.uk/
33. team, O: OWASP Mutillidae II (2019) https://github.com/webpwnized/mutillidae

How to cite this article: Zhuo Z, Cai T, Zhang X, Lv F. Long short-term memory on abstract syntax tree for SQL injection detection. *IET Soft*. 2021;15:188–197. https://doi.org/10.1049/sfw2.12018