Deep Tree: SQL Injection Detection by the Power of Deep Learning

Peifeng Yu
University of Michigan

Shiyu Wang
University of Michigan

Abstract

SQL injection attack is one of the major attacks to Web applications nowadays. In order to defend against such attack, lots of prevention and detection approaches have been proposed. In this paper, we propose Deep Tree, a new deep learning based approach to distinguish SQL injection attack from benign queries. It uses deep neural network as a classifier and leverages the structural information gained from the parse trees of SQL statements. By combining deep learning and the parse tree based validation techniques, Deep Tree succeeds in detecting SQL injection attacks with high accuracy (94.7%) even with unseen attacks and attacks using evasion techniques. We evaluate the performance and effectiveness of Deep Tree over a new SQL injection dataset we generated. Experimental results indicate that Deep Tree is practical because of its high accuracy and moderate latency.

1 Introduction

SQL injection is an attack performed by injecting malicious SQL statements into the input fields of the web applications, leading to some unexpected commands to be executed. A web application that does not have appropriate input filtering mechanism can be easily attacked by SQL injection. By SQL injection, attackers are able to get access to the sensitive data in the databases, modify the databases, and become the databases' administrators, etc.

Due to its severity and popularity, SQL injection attack is one of the major threats to web applications currently. In 2013, SQL injection attack ranked first in the OWASP top ten project. There is no doubt that it is one of the significant security problems really requiring lots of attentions.

There have been multiple prevention and detection approaches proposed so far in order to defend against SQL injection attacks. Parameterized queries is one of the most well known *prevention* techniques used to defend against SQL injection, but it requires correct implementation by application developers. Keyword based filtering technique, parse tree validation and machine learning technique are three main approaches for SQL injection *detection*. However, as noted in section 2, they all have their own limitations, whether are not suitable for the dynamic nature of the attacks or not strong enough to detect attacks using injection evasion techniques introduced in the following section.

As a result, by combining parse tree based technique with the state-of-art machine learning technique, we propose Deep Tree, a new SQL injection detection approach which takes advantages of both machine learning and parse tree based techniques, while also solves the potential problems of them.

1.1 Injection Evasion Techniques

The attackers have developed various evasion techniques to bypass injection detection and filtering. This often involves changing the signature of query string by inserting unnecessary functions, or hiding keywords through comments. The following lists a few common evasion techniques for reference.

- C-like comment
  `UNI/* anything */ON SE /* again */ LECT`
- String concatenation
  `EXEC("IN" + "SERT" + "IN" + "TO...")`
- Using `char`, `hex`, etc. functions to build the query string
  ```
  unhex(hex(Concat(Column_Name,0x3e,
                   Table_schema,
                   0x3e,table_Name)))
  ```
- Using HTTP Parameter Fragmentation (HPF)
  `/?a=1+union/*&b=*/select+1,2`

Goal	Keyword / Regex	Manual Feature Extraction (Parse Tree)	Raw String Based Traditional Machine Learning
High accuracy	No	Yes	Yes (If no evasion)
Immune to evasion techniques	No	Yes	No
Defend against new attacks	No	No	Yes

Table 1: Existing work and common goals

2 Related Work

2.1 Web Application Level Filtering

Currently, most websites use parametrized SQL queries [4] to escape potential SQL statements in queries so that they can protect themselves from injection attacks. Although parameterized queries can prevent most of the SQL injection attacks, it cannot guarantee 100% security. First, it relies on programmers' efforts, which means that implementation vulnerabilities can exist. Furthermore, parameterized queries is not able to defend against all of the SQL injections when dynamic SQL such as triggers and user-defined functions are needed, because when a top-level function calls another function, it is possible that somewhere in the subroutines is vulnerable to the SQL injection attack, as it is impractical to ensure that all functions in the subroutines use bound parameters.

Keyword-based filtering technique is another popular application level defense against SQL injection attacks. This technique applies filtering mechanism to make sure the given inputs do not have any SQL keywords such as `SELECT`, `--`, etc. However, this technique can be easily bypassed using evasion techniques.

2.2 Database Side Detection

Instead of relying on web application level filtering to prevent SQL injection attacks, passive detections on database side are preferred. There are lots of prior work on SQL injection attack and several detection methods have been proposed.

Current work includes detecting SQL injection using machine learning techniques. For example, Moosa suggests to protect web applications against SQL injection attack using Artificial Neural Network [6]. Singh, et al. propose a clustering technique [11] to detect SQL injection. Support Vector Machine can also be used to detect such attack [10].

However, these methods are mostly based on raw input string analysis, and fail to utilize the state-of-art machine learning techniques. There are also other drawback of these methods, including inefficient training stages due to the lack of understanding of existing SQL syntax, and sensitivity to insignificant changes (e.g. different variable name, values, statement structures).

On the other hand, being a step further than analyzing the raw input string, there are approaches detecting SQL injection attack based on features of the parse tree of the input. Buehrer et al. suggest to compare the parse tree of the SQL statement before and after including user input in order to detect SQL injection [3]. However, such approach is not suitable due to the dynamic nature of attacks. Moreover, the pattern database can only be updated after the attack has happened.

Table 1 summarizes major solutions defend against SQL injection, as well as three desired goals/features that should be included for a successful defend mechanism.

3 Design

3.1 Overview

Motivated by the accuracy provided by the parse tree, and the quick advance in machine learning algorithms, we propose Deep Tree, a new deep learning based SQL injection detection approach combining the state-of-art machine learning technology with the parse tree method. On the input side, a SQL parser is used. After all, why reinvent the wheel to extract information from an SQL statements when there are already pieces of software that can perfectly understand SQL language. The generated parse tree contains all the syntax and semantic information and is the canonical form of the statement. Therefore this will by nature be immune to any evasion techniques because the attacker has to submit a syntactically valid statement anyway. On the other end, we choose deep neural network as our classifier, from which generalizability can be gained. This is desirable for detecting new attacks, given their dynamic nature.

Figure 1 shows a high-level diagram of our proposed system. It sits between the web application and the database backend, accepting SQL statements generated from the application and originally directly issued to the database. After passing through the parser and classifier,

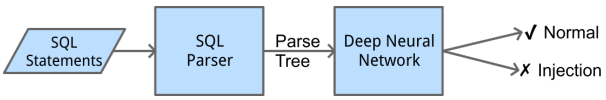

Figure 1: Architecture of Deep Tree

the system decides whether it is a normal query or a malicious injection attack. The classifier actually acts as a filter of queries before they are executed in the database system, and will deny or mark suspicious queries for further inspections.

Note that our goal is to distinguish injected statements from benign ones, not to discover hidden vulnerabilities in existing web applications. We believe that by acting as a middleware and providing a centralized place for SQL injection, the responsibility are taken away from individual web developers who usually do not have enough time/resource to comprehensively examine their application for vulnerabilities.

3.2 Network Architecture

The neural network structure plays an important role in the performance of the final classifier. Since our goal is to capture as much information as possible for the network, we want to encode the tree structure information into the input. Mou et al. [8] proposed a tree-based convolutional layer that can extract tree structure information using a sliding window on subtrees. However, their original work focused on classification of C programs. We build our system based on the similar idea, but adapted it to SQL language specific parse trees.

The network pipeline is shown in Figure 2. Beginning from a parse tree, the network first convert the tree into a vector representation that is suitable for later computation. Then the tree-based convolution is applied to extract local features, resulting in a new set of feature vectors with the exactly same tree structure of the input. However, the tree structures vary among different SQL inputs. Therefore, in order to be able to connect to the fully-connected softmax classifier, the tree is passed through a dynamic pooling layer to be reduced into fixed vector shape. The final step is one layer of fully-connected neurons with softmax output acting as a classifier.

3.2.1 Parse Node Embedding Pre-training

Using tree structure as input to neural network is not unique in language processing. The processing system traditionally treats tree nodes as discrete atomic symbols. This is quite different from image and audio processing systems, which work with rich high-dimensional inputs encoded as vectors of raw pixel-intensities or power spectral density coefficients, and all the information required to successfully perform tasks like recognition is encoded in the data (because humans can perform these task from the raw data). The discrete atomic symbols are arbitrarily encoded as `Id537` or `Id123`, and provide no useful information to the system regarding the relationships that may exist between the individual symbols. To overcome some of these obstacles, we can represent (embed) words in a continuous vector space where semantically similar words are mapped to nearby points, in other words, are embedded nearby each other.

We adopted the recursive autoencoders [12] proposed by Socher et al., combined with the continuous binary technique [7] to enable training on arbitrary structured parse tree inputs. At the high level, we use child nodes encode parent node, and parent node reconstructs child nodes, and minimize the difference between these two.

Formally, we denote the vector representation of node x as $\text{vec}(x) \in \mathbb{R}^{N_f}$, where N_f is the dimension of the vector, aka the dimension of features. Stacking the vector for all node types together as rows makes the embedding matrix $\boldsymbol{W}_e \in \mathbb{R}^{N_w \times N_f}$, where N_w is the total number of node types.

For each non-leaf node p in the parse tree, and its direct children $c_1, c_2, \ldots, c_{n_p}$, we want to find encoding weights $\boldsymbol{W}_i$ and embedding matrix $\boldsymbol{W}_e$ such that when using the vector representation of the children to encode that of the parent, the difference is minimized.

$$\text{vec}(p) \approx \tanh\left(\sum_{i=1}^{n_p} l_i \boldsymbol{W}_i \text{vec}(c_i) + \boldsymbol{b}\right) \tag{1}$$

where $\boldsymbol{W}_i \in \mathbb{R}^{N_f \times N_f}$ is the encoding weight for child node i, and $\boldsymbol{b}$ is a common bias term. l_i represents a coefficient that weights all children nodes by the number of leaves under c_i:

$$l_i = \frac{\#\text{leaves under } c_i}{\#\text{leaves under } p} \tag{2}$$

Note that unlike typical NLP tasks, our parse tree is mostly unlikely to be a binary tree, and each node can have arbitrary number of children. Thus the number of $\boldsymbol{W}_i$ is different for each tree node. This mean they can not be shared among inputs, even worse, can not be computed recursively in the same tree. This problem can be overcome by introducing two shared encoding weight matrices $\boldsymbol{W}_l$ and $\boldsymbol{W}_r$, and compute $\boldsymbol{W}_i$ as a linear combination of both.

$$\boldsymbol{W}_i = \begin{cases} \frac{n_p - n_i}{n_p - 1}\boldsymbol{W}_l + \frac{n_i - 1}{n_p - 1}\boldsymbol{W}_r & (n_p \geq 2) \\ \frac{1}{2}\boldsymbol{W}_l + \frac{1}{2}\boldsymbol{W}_r & (n_p = 1) \end{cases} \tag{3}$$

where n_p is the number of children of node p, and n_i is the number of children of c_i, which itself is a child of node p. Intuitively, this means, for a node closer to the

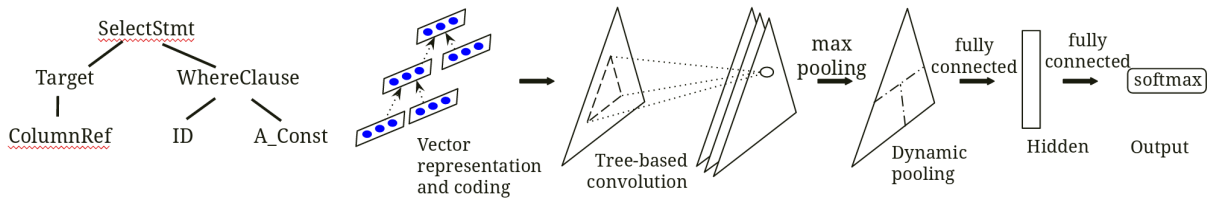

Figure 2: Network Pipeline [8]

left side, its encoding weight will be more influenced by $\boldsymbol{W}_l$, less influenced by $\boldsymbol{W}_r$ and vise versa.

Having defined all necessary terms, this whole embedding matrix pre-training can be formulated as

$$\begin{aligned} \text{minimize} \quad & d = \left\| \text{vec}(p) - \tanh\left(\sum_{i=1}^{n_p} l_i \boldsymbol{W}_i \text{vec}(c_i) + \boldsymbol{b} \right) \right\|_2^2 \\ \text{s.t.} \quad & \text{vec}(x) = \boldsymbol{W}_e[x,:] \\ & \boldsymbol{W}_i = \begin{cases} \frac{n_p - n_i}{n_p - 1} \boldsymbol{W}_l + \frac{n_i - 1}{n_p - 1} \boldsymbol{W}_r & (n_p \geq 2) \\ \frac{1}{2} \boldsymbol{W}_l + \frac{1}{2} \boldsymbol{W}_r & (n_p = 1) \end{cases} \\ & l_i = \frac{\#\text{leaves under } c_i}{\#\text{leaves under } p} \end{aligned} \tag{4}$$

3.2.2 Coding Layer

To further capture structural information during training, the pre-trained embedding matrix $\boldsymbol{W}_e$ is not directly used. Rather, it is again combined using the encoder technique similar in the pre-training stage.

$$\text{code}(p) = \tanh\left(\sum_{i=1}^{n_p} l_i \boldsymbol{W}_{code,i} \cdot \text{vec}(c_i) + \boldsymbol{b}_{code} \right) \tag{5}$$

$$\text{fea}(p) = \begin{cases} \boldsymbol{W}_{comb1} \text{vec}(p) + \boldsymbol{W}_{comb2} \text{code}(p) & \text{non-leaf} \\ \text{vec}(p) & \text{leaf} \end{cases} \tag{6}$$

where $\boldsymbol{W}_{comb1}$ and $\boldsymbol{W}_{comb2}$ are the combination matrices, and $\boldsymbol{W}_{code,i}, \boldsymbol{b}_{code}$ are new encoding weights and bias.

3.2.3 Tree-Based Convolution

Having built up a vector representation for each node in the parse tree, the tree-based convolutional layer [8] is then applied next to extract local features from the input. Similar to convolutional layers in image processing, a tree-based convolutional layer applies a set of fixed-depth feature detectors sliding over the entire tree, whose nodes are now feature vectors gained using $\text{fea}(\cdot)$ defined in previous layer.

The feature detector operates on a subtree of given depth, and computes one real number value from the input. Thus applying a set of feature detectors on a subtree generates a vector, as shown in Figure 3. Therefore, after compute all subtrees rooted at each position in the original tree, we get another vector tree of exactly the same tree structure, but the vector dimension in each node is changed from N_f to N_c, where N_c is the number of feature detectors.

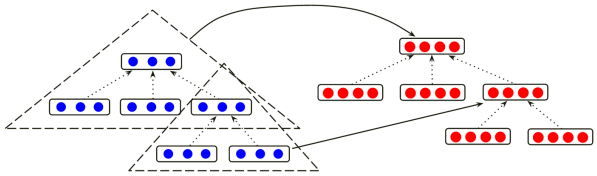

Figure 3: Feature detectors operates on subtrees and generates one vector at each position [8]

Formally, the set feature detectors can be collectively viewed as a function defined on a fixed-depth subtree, for all nodes in this subtree, whose vectors are $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$, the output is

$$\boldsymbol{y} = \tanh\left(\sum_{i=1}^{n} \boldsymbol{W}_{conv,i} \cdot \boldsymbol{x}_i + \boldsymbol{b}_{conv} \right) \tag{7}$$

where $\boldsymbol{y} \in \mathbb{R}^{N_c}$ is the output, $\boldsymbol{b}_{conv} \in \mathbb{R}^{N_c}$ is the bias, and $\boldsymbol{W}_{conv,i}$ is the detector weights.

Again, we are facing the same issue as in embedding pre-training and coding: the tree structure is arbitrary and the number of children for a node can not be determined in advance. Applying the continuous binary technique again, with three weight matrices this time, we can compute $\boldsymbol{W}_{conv,i}$ based on the position of the node in the subtree.

$$\begin{aligned} \boldsymbol{W}_{conv,i} &= \eta_i^t \boldsymbol{W}_{conv}^t + \eta_i^l \boldsymbol{W}_{conv}^l + \eta_t^r \boldsymbol{W}_{conv}^r \\ \eta_i^t &= \frac{d_{max} - d_i}{d_{max}} \\ \eta_i^l &= \begin{cases} \frac{i-1}{n-1}(1 - \eta_i^t) & \text{non-leaf} \\ 0.5(1 - \eta_i^t) & \text{leaf} \end{cases} \\ \eta_i^r &= (1 - \eta_i^t)(1 - \eta_i^l) \end{aligned} \tag{8}$$

where $\boldsymbol{W}^t_{conv}, \boldsymbol{W}^l_{conv}, \boldsymbol{W}^r_{conv} \in \mathbb{R}^{N_c \times N_c}$ are three base weight matrices used to compute all the weights, n is the number of siblings of the node, i is the index of the node. Note that this equation is slightly adjusted from the original version proposed [8], as we believe our version makes more sense conceptually, and also yields better result in our experiments. The equation can be intuitively interpreted similarly as a color triangle (Figure 4) which combines colors at three corners, whereas we combines matrices at three corners.

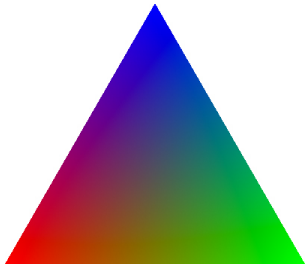

Figure 4: Color triangle: the color of each point inside the triangle is a combination of the color of three vertices [8]

We note that this combination is merely used to reduce the weights needed for training to a fixed number and shape, and we believe any sensible way of doing this should yield reasonable result. Yet, we didn't try other different ways as the current one performs well.

3.2.4 Fully Connected Softmax Classifier

The output of tree-based convolutional layer is still a tree structured set of vectors, which can not be directly fed into fully connected layer. Therefore, we applied dynamic pooling [12] to reduce the tree into a fixed shape vector. Specifically, max pooling is applied to each dimension of the vector in the input tree, resulting a single vector $\boldsymbol{x} \in \mathbb{R}^{N_c}$ per input tree (sample).

The final fully connected softmax classifier is a commonly used structure in all neural network designs for classification. It consists one or more layer of fully connected neurons, and lastly a softmax function applied to the output to make it sensible statistically. In our case, we used one layer of 1024 neurons, followed by 2 neurons (we have two classes as output, normal and malicious) regularized by softmax.

4 Implementation

4.1 Dataset Collection

We have not found any established public dataset available for SQL injection attacks, so we have to compile our own SQL injection dataset.

Positive Samples: Our positive samples are collected from multiple popular projects on GitHub and from our previous projects. We use them as samples of valid statements passed to the database, not as samples of vulnerable statements, so there is no need to discover hidden vulnerabilities in existing web applications, which is out of our scope.

We decide to collect most of the samples from real-world applications and SQL tutorials, so these positive samples contain multiple types of SQL statements such as `SELECT`, `UPDATE`, `DELETE`, `ALTER`, etc. Several complex SQL statements structures are also covered, examples including nested queries or `GROUP BY`. Hence the diversity and complexity of the positive samples can be guaranteed.

Negative Samples: 25% of our negative samples are generated by SQL injection tool named SQLMap. Furthermore, we also investigated papers about SQL injection and collected SQL injection signatures from [6, 1, 2]. We then wrote scripts to generate more injection statements by varying the substructures of the collected SQL injection signatures.

We paid special attention when generating our negative samples to include complex SQL statements with, e.g. nested queries, `HAVING` and `GROUP BY` clauses. As a statistic, our negative samples cover 5 common types of SQL injection techniques, which are tautologies, union queries, piggyback queries, malformed queries and inference queries. Figure 5 shows the distribution of different attack techniques included in our dataset.

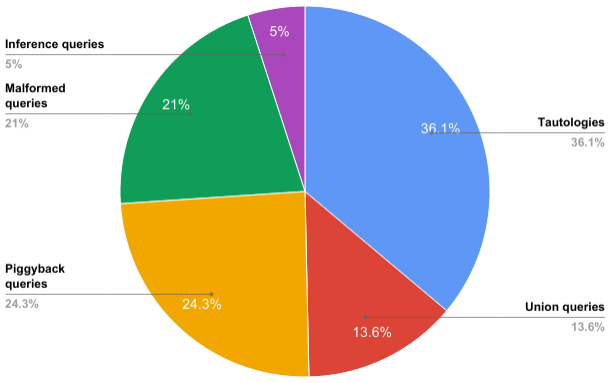

Figure 5: Attack techniques included in our dataset

In total, 4161 samples were collected and divided into three splits, among which 2496 samples used for training, 832 samples for validation, and 833 samples for testing.

4.2 SQL Parser

We used `libpg_query` to obtain parse trees of SQL statements. `libpg_query` shares the same source code with an SQL server commonly used in production, named PostgreSQL, and can export the internal parse trees as JSON objects. Although our approach currently can only handle SQL statements which are valid according to PostgreSQL syntax due to our usage of `libpg_query`, the methodology is not specific to PostgreSQL and can be generalized to other databases as well, as long as we can obtain parse trees of given SQL queries. The advantage of using a full fledged SQL parser that is actually used by PostgreSQL is that, we can handle SQL statements of arbitrary complexity as long as they are of valid syntax.

In order to be more agnostic to the output of `libpg_query`, which also includes useless information such as the location information, table names, we further process the resulting parse trees serialized as JSON objects. Since our Deep Tree uses the structures and node types of parse trees as features to train the neural network, the processed parse tree contains only these two types of information, well strips others.

Figure 6 is an example of the parse tree generated from a SQL statement.

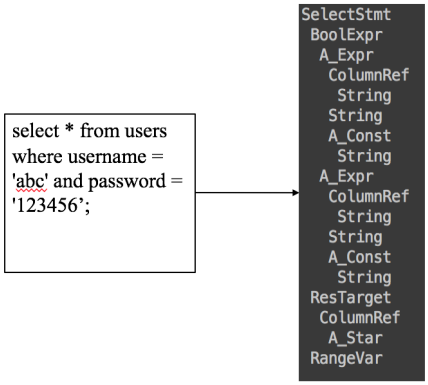

Figure 6: Example of parse tree

4.3 Training

The training was done on IBM Power8 CPU (80 cores @ 3.9 GHz) with 128 GB Memory, and accelerated with NVidia Tesla P100 GPU with 16GB Memory.

Following are the hyper parameter values we used during evaluation

- Feature dimension $N_f = 100$
- Convolutional feature detector $N_c = 50$
- Subtree depth for feature detector: 2
- Fully connected layer size: 1024
- Learning rate: 0.002 with 0.65 decay every 200 steps (Figure 7)
- Weight decay: 0.002

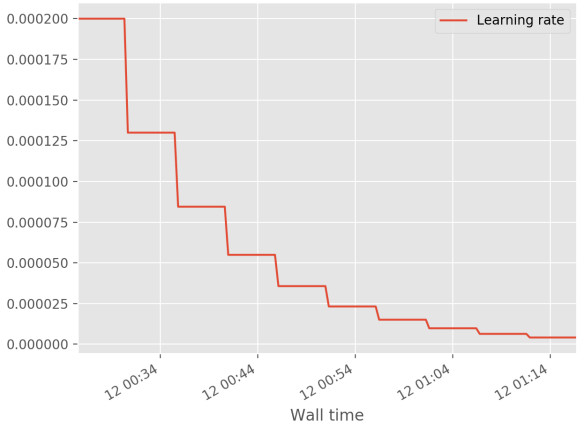

Figure 7: Exponential decay learning rate

We obtained convergent result after 2000 steps, which takes 51 minutes in total, as shown in Figure 8.

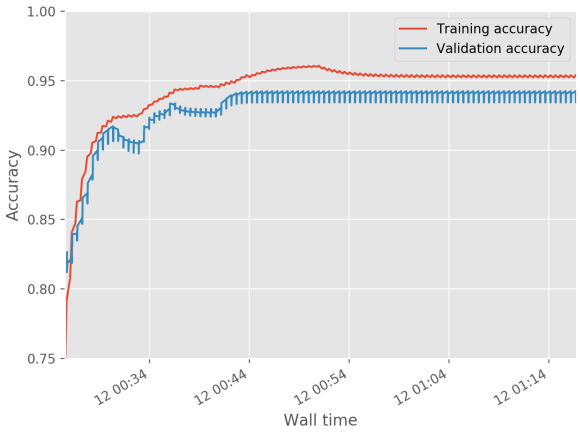

Figure 8: Accuracy while training

5 Evaluation

We first assess the learned vector representation by visualization, then the classification performance is analyzed on the testing dataset split.

5.1 Quality of Embedding Matrix

All samples in our dataset are used to pre-train the embedding matrix. Figure 9 gives the parse tree node type distribution. As demonstrated, `String`, `A_Const` and `ColumnRef` nodes make up large portion of the dataset.

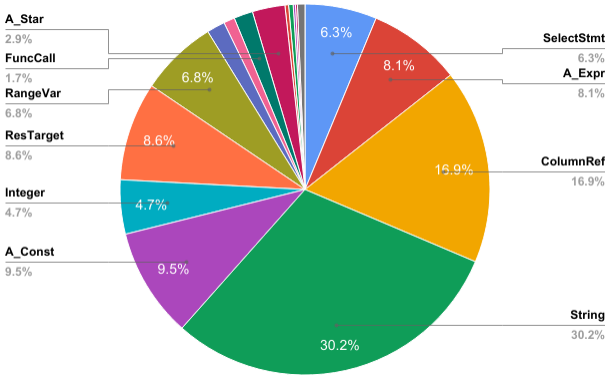

Figure 9: Node type distribution

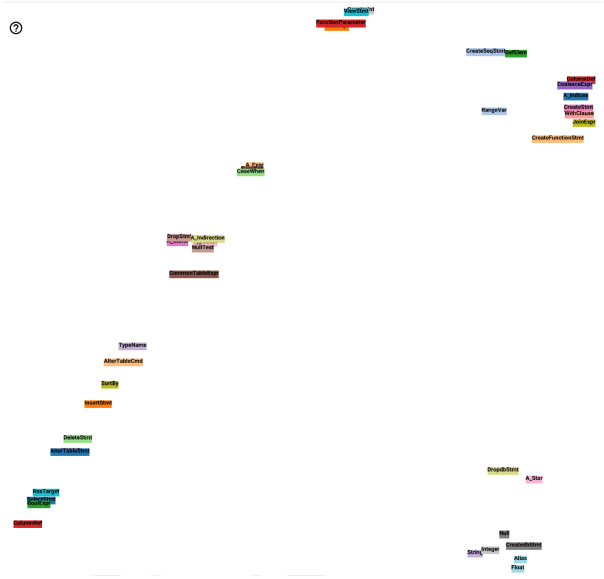

Figure 10: Visualization of embedding matrix

The quality of embedding can be assessed by evaluate how semantically similar node types are mapped to nearby points in the feature space. We apply T-SNE [5] visualization algorithm to the learned vectors (perplexity = 0.1, learning rate = 0.1) to better understand the relationship among feature vectors. The result [1] is shown in Figure 10. It can be seen that constant nodes (`Integer`, `Float`, `String`, etc.) are clustered together, while nodes share similar syntactic positions, e.g. `ColumnRef`, `BoolExpr`, `ResTarget` are near to each other. Therefore, we believe the embedding matrix pre-training is efficient and beneficial to our classification task.

5.2 Classification Performance

As stated in section 4.3, we achieved convergence after 2000 steps of training. Table 2 gives the achieved accuracy on all three dataset splits.

Dataset split	Accuracy
Training	96.9%
Validation	95.3%
Testing	94.7%

Table 2: Achieved accuracy

As a comparison, Table 3 summarizes the latency and accuracy achieved by various SQL injection detection approach we reviewed in related works. The first two methods do not use machine learning algorithms and achieved 100% accuracy on their dataset. Among the last three machine learning based algorithms, while the combination of SVM and parse tree achieves best accuracy, it takes almost a second to compute the result. Given the high volume of the Internet web application requests and tight latency requirements, this methods can hardly be used as a real time filter, which makes it less useful. The ANN solution makes lightening fast classification, but since its structure is quite simple, the accuracy is jeopardized. Our approach strikes a good balance, with comparable accuracy **94.7%** while maintains fast detection speed, with average runtime **4.31 ms**. We believe this is desirable for integration into production systems as real time filter even with tight latency requirements.

Method	Latency	Accuracy
Manual feature extraction + K-Mean Cluster [11]	N/A	100%
Parse tree + Rule [3]	18 ms - 127 ms	100%
Raw string + ANN [6]	0.45 ms	83.7%
Parse tree + SVM [9]	837 ms	95.7%
Parse tree + DNN (Ours)	4.31 ms	94.7%

Table 3: Accuracy and latency of SQL injection detection

6 Conclusion

We develop Deep Tree, a new deep learning based approach, combining the state-of-art machine learning technology and the parse tree method to detect SQL in-

jection. We evaluated it on our newly compiled SQL injection dataset and achieved 94.7% accuracy with 4.31ms/sample latency, indicating that our approach can detect SQL injection with high accuracy and accepted latency. At the same time, our approach is immune to evasion techniques because it utilizes the information from parse trees of SQL statements and the neural network can also detect new attacks. Our code for the implementation of Tree-based Convolutional Neural Network is open-sourced, and can be found at `https://github.com/Aetf/tensorflow-tbcnn` now.

We note that the accuracy and latency numbers we used in Table 3 are obtained directly from corresponding papers. Since there's no common dataset available, this makes the comparison less fair. In the future, we plan to re-implement these existing approaches, and measure the accuracy and latency on our own dataset to obtain the most accurate data.

We also believe that our approach has much potential for improvement. In addition to leveraging syntactic features, we would like to capture semantical features of parse trees during embedding process as well. Other future work includes collecting data from real attacks to establish the dataset by using some honey pot systems, and deploying Deep Tree to real world system for further performance measurement.

7 Contribution Break-down

- **Shiyu Wang**: Idea, data collection, parsing, documentation
- **Peifeng Yu**: Network implementation and training

References

[1] ANLEY, C. Advanced sql injection in sql server applications. *Next Generation Security Software* (2002).

[2] ANLEY, C. Hackproofing mysql. *Next Generation Security Software* (2004).

[3] BUEHRER, G., WEIDE, B. W., AND SIVILOTTI, P. A. Using parse tree validation to prevent sql injection attacks. In *Proceedings of the 5th international workshop on Software engineering and middleware* (2005), ACM, pp. 106–113.

[4] LWIN KHIN SHAR, H. B. K. T. Defeating sql injection. *Computer 46*, 3 (2013), 69–77.

[5] MAATEN, L. V. D., AND HINTON, G. Visualizing data using t-sne. *Journal of Machine Learning Research 9*, Nov (2008), 2579–2605.

[6] MOOSA, A. Artificial neural network based web application firewall for sql injection. *World Academy of Science, Engineering & Technology 64*, 4 (2010), 12–21.

[7] MOU, L., LI, G., LIU, Y., PENG, H., JIN, Z., XU, Y., AND ZHANG, L. Building program vector representations for deep learning. *arXiv preprint arXiv:1409.3358* (2014).

[8] MOU, L., LI, G., ZHANG, L., WANG, T., AND JIN, Z. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of 30th AAAI Conference on Artificial Intelligence (to appear)* (2016).

[9] PRIYAA, B. D., AND DEVI, M. I. Fragmented query parse tree based sql injection detection system for web applications. In *Computing Technologies and Intelligent Data Engineering (ICCTIDE), International Conference on* (2016), IEEE, pp. 1–5.

[10] RAWAT, R., AND RAGHUWANSHI, S. Sql injection attack detection using svm. *International Journal of Computer Applications 42*, 13 (2012), 1–4.

[11] SINGH, G., KANT, D., GANGWAR, U., AND SINGH, A. P. Sql injection detection and correction using machine learning techniques. In *Emerging ICT for Bridging the Future-Proceedings of the 49th Annual Convention of the Computer Society of India (CSI) Volume 1* (2015), Springer, pp. 435–442.

[12] SOCHER, R., HUANG, E. H., PENNINGTON, J., NG, A. Y., AND MANNING, C. D. Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In *NIPS* (2011), vol. 24, pp. 801–809.

Notes

[1] Due to space constrains, we only included a scaled down version of the result, which is somewhat hard to read. You can find the full version at `https://github.com/Aetf/tensorflow-tbcnn/blob/master/misc/embedding_vis.png`