SQLStateGuard: Statement-Level SQL Injection Defense Based on Learning-Driven Middleware

Xin Liu
School of Information Science & Engineering, Lanzhou University
Lanzhou, Gansu, China
bird@lzu.edu.cn

Yuanyuan Huang*
Tianyi Wang*
huangyy19@lzu.edu.cn
wangty2020@lzu.edu.cn
School of Information Science & Engineering, Lanzhou University
Lanzhou, Gansu, China

Song Li†
The State Key Laboratory of Blockchain and Data Security, Zhejiang University
Hangzhou, Zhejiang, China
songl@zju.edu.cn

Weina Niu
School of Computer Science and Engineering, University of Electronic Science and Technology of China
Chengdu, Sichuan, China
vinusniu@uestc.edu.cn

Jun Shen
School of Computing and Information Technology, University of Wollongong
Wollongong, New South Wales
Australia
jshen@uow.edu.au

Qingguo Zhou†
School of Information Science & Engineering, Lanzhou University
Lanzhou, Gansu, China
zhouqg@lzu.edu.cn

Xiaokang Zhou
Faculty of Business Data Science, Kansai University
Suita, Osaka, Japan
zhou@kansai-u.ac.jp

ABSTRACT

SQL injection is a significant and persistent threat to web services. Most existing protections against SQL injections rely on traffic-level anomaly detection, which often results in high false-positive rates and can be easily bypassed by attackers. This paper introduces SQLStateGuard, the world's first middleware-driven statement-level SQL injection defense approach, to address these issues. The SQLStateGuard uses a custom SQL middleware based on the idea of Runtime Application Self-Protection to capture raw SQL statements. These statements are then analyzed by SQLSG-Net, a database-oriented detection network based on gated linear units. If SQLSG-Net detects malicious SQL statements, the SQL middleware will block them. Experiments show that the detection accuracy of SQLStateGuard exceeds 99%, outperforming existing approaches, and it can identify the type of a specific SQL injection. Additionally, SQLStateGuard has no fingerprint and does not respond to SQL syntax errors, making it more challenging for attackers to gather information. This paper also presents a novel dataset generation process for SQLStateGuard and shares two statement-level SQL injection datasets with the research community, including over 145,000 malicious SQL statements categorized by the type of SQL injection.

*Both authors contributed equally to this research.
†Corresponding authors.

SoCC '24, November 20–22, 2024, Redmond, WA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1286-9/24/11.
https://doi.org/10.1145/3698038.3698569

CCS CONCEPTS

• **Security and privacy** → **Firewalls**; *Database activity monitoring*.

KEYWORDS

SQL Injection, Attack Detection, Deep Learning, Web Security, Data Security

ACM Reference Format:
Xin Liu, Yuanyuan Huang, Tianyi Wang, Song Li, Weina Niu, Jun Shen, Qingguo Zhou, and Xiaokang Zhou. 2024. SQLStateGuard: Statement-Level SQL Injection Defense Based on Learning-Driven Middleware. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3698038.3698569

1 INTRODUCTION

SQL injection is a technique attackers use to inject malicious payloads into SQL statements executed by a database server. Attackers can use it to steal confidential information or compromise the integrity of the database [45]. According to a recent research [4], despite being well-known for over two decades, SQL injection remains among the top attack techniques due to their low cost, high impact, and flexibility.

Traditional SQL injection detection approaches [6, 7, 29, 40] generally detect and filter SQL injections through rule matching. The effectiveness of these approaches highly depends on the design of predefined rules. If a user's request matches a rule, it will be reported as malicious. Therefore, these rules are very fragile in the face of new threats. Besides, they also suffer from sensitivity challenges - lax matching rules can significantly increase false positives. In contrast, stringent matching rules can substantially lead to lower detection rates.

With advancements in machine learning, researchers [18, 24, 39] started to use learning-based approaches to detect SQL injections. Most of them have focused on detection at the traffic level. However, the complexity and diversity of traffic can pose challenges for machine learning models to detect SQL injections accurately. [12] The difference between normal traffic and malicious traffic can be very slight. On the other hand, most fuzzing payloads do not cause a real hazard, while traffic-level detection will generate numerous meaningless alerts because of them, severely increasing the burden on security engineers.

In this paper, we present SQLStateGuard, a deep learning-based solution to defend against SQL injections. SQLStateGuard is based on middleware and operates at the statement level, enhancing detection accuracy and reducing false alarms. It consists of two core components: the SQL middleware and SQLSG-Net. The SQL middleware, built on the Runtime Application Self-Protection (RASP) [2] idea, is responsible for extracting SQL statements and detecting and blocking SQL injections. SQLSG-Net, on the other hand, is a statement-level SQL injection detection network based on Natural Language Processing (NLP). It uses database-oriented models to detect SQL injections and identify their types accurately. We summarize our contributions as follows.

- This paper introduces the idea of RASP into the middleware, thereby enabling security analysis and blocking SQL injection at the statement level. Compared to existing approaches, SQLStateGuard captures the statements about to be executed, removing most of the unnecessary information, resulting in non-intrusive and accurate blocking of potentially harmful SQL injections.
- A new SQL statement-level injection detection network, SQLSG-Net, is proposed to incorporate NLP techniques. SQLSG-Net conducts SQL semantic parsing and then utilizes database-oriented models based on gated linear units mechanism to perform high-precision statement-level difference mining between SQL statements. This results in more accurate and faster detection, effectively reducing the number of false positive reports generated by meaningless SQL injection vectors.
- A prototype of SQLStateGuard has been implemented and experimentally validated using datasets generated through a novel dataset generation process. The results demonstrate that SQLStateGuard offers significant advantages over existing approaches. With a binary classification accuracy of over 99.9%, SQLStateGuard effectively detects SQL injections with a meager false positive rate and can accurately identify the different types of SQL injections.
- This paper shares two statement-level SQL injection datasets [1] with the research community, including more than 145,000 malicious SQL statements categorized by the type of SQL injection.

The rest of this paper is organized as follows: Section 2 presents the existing approaches of SQL injection protection. Section 3 presents the design of SQLStateGuard. Section 4 uses multiple datasets to evaluate SQLStateGuard. Section 5 discusses the weaknesses and future improvements of SQLStateGuard. Section 6 concludes this paper.

2 RELATED WORK

Existing SQL injection protection approaches mainly rely on rules. [3] Depending on the object of analysis, existing approaches can be classified into code-level protection and traffic-level protection.

2.1 Traditional SQL Injection Protection

In terms of code-level protection, existing research is usually based on static approaches such as taint propagation analysis [36] to find SQL injection vulnerabilities at the code level, addressing the root cause of SQL injection problems.

[1]https://github.com/dstsmallbird/SQLStateGuard_Dataset

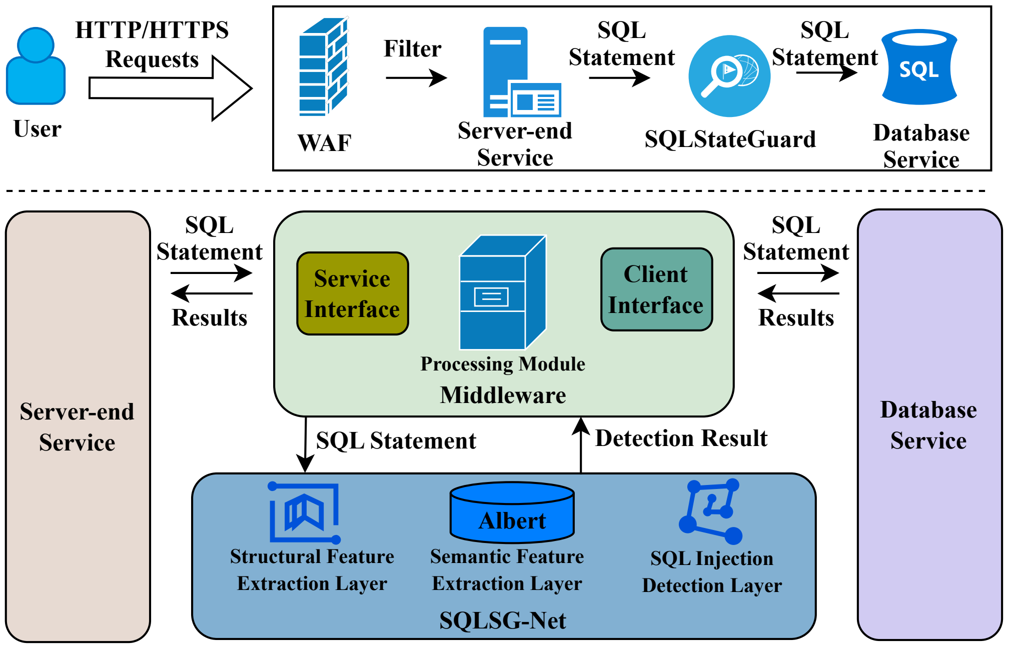

Figure 1: Workflow and Architecture of SQLStateGuard

RIPS [11], a static code analysis tool developed for PHP applications, is one of the most representative approaches of this kind. Similarly, Mei et al. [19] proposed a grey-box approach to detect SQL injections in Java runtime. Zhang et al. [46] tried to use Moving Target Defense (MTD) to reduce the attack surface of SQL injections.

Most traffic-level protection is based on syntax analysis of user requests [30], and it is very rare to conduct detection based entirely on statistical characteristics of network traffic, such as request frequency, request packet size, and the number of connections. The only existing work is the approach proposed by Yufei et al. [47] for detecting automated SQL injection tests. Most of the approaches based on syntactic analysis of user requests are Intrusion Detection System (IDS)-related studies, which directly detect the content of user requests (e.g., HTTP messages) based on the completed protocol parsing, and WAF is the representative. [8] Traditional IDS can also detect SQL injection attacks, such as Zolotukhin et al. [48], who model normal user behavior using anomaly detection algorithms and clustering algorithms and then identify malicious requests by the deviation values of user behavior. There are also some ideas to protect against SQL injection attacks by exploiting the information gap between attack and defense, e.g., SQLrand [7] uses inserting random numbers into SQL keywords and modifying SQL interpreters to counter injections.

2.2 Learning-based SQL Protection Detection

In code-level protection, researchers have tried to let AI understand the code and find potential security risks, e.g., VulHunter [14] based on deep learning and bytecode to detect SQL injection vulnerabilities in PHP applications. Traffic-level protection is currently the most popular research topic in learning-based SQL injection protection. Similar to traditional approaches, although there are approaches [44] based entirely on statistical features of traffic, user requests analysis based on packet inspection and protocol analysis using deep learning techniques is still predominant. Liu et al. [25] proposed OwlEye, a hybrid attack detection sensor based on Hidden Markov Model (HMM) designed to defend against web-layer code injection attacks, achieving a high detection rate with an acceptable false positive rate through its bidirectional scoring architecture that leverages both benign and malicious traffic in model training. Peng et al. [35] use

Multilayer Perceptron (MLP) and Long Short-Term Memory (LSTM) to detect SQL injections based on packet inspection. Liu et al. [41] proposed the Multi-class S-TCN model, which enhances the detection speed and accuracy of SQL injection attacks by leveraging time convolutional networks, significantly improving real-time traffic analysis in IoT scenarios. Crespo-Martínez et al.[10] use a Logistic Regression-based model to detect SQL injections. Abaimmov et al. [1] proposed CODDLE system, which uses Convolutional Neural Network (CNN) to detect injections. Lu et al. [27] construct a model named synBERT to dectect SQL injections. In recent years, there have been some researchers in statement level detection, but most of their detection rules are relatively simple. William G. J. et al. [15] combined static analysis with dynamic monitoring to develop a tool named AMNESIA. This tool constructs SQL query models through static analysis and subsequently monitors queries at runtime to detect potential SQL injection attacks. Konstantinos et al. [20] detect SQL injection attacks by intercepting SQL statements and determining whether the statements conform to predefined specifications.

One of the most important features of SQL injections is the use of special symbols to break through the original statement structure, which has been discussed in depth by existing approaches [26, 42, 43]. However, since user-generated content is highly variable, it is difficult for datasets to cover all normal user input, making normal user requests (especially those including special symbols such as single quotes) easily mistaken for malicious requests. Since the learning-based approaches rely on generalization, it is difficult to adjust the rules as traditional approaches do when such problems arise, which reduces the feasibility of the learning-based approach in the real world.

3 DESIGN OF SQLSTATEGUARD

3.1 Overview

SQL injection detection is a strong-demand old topic without Silver Bullets. Successful SQL injections are not specific at the statement level, so even experienced security experts cannot tell if a statement is an SQL injection without understanding the code implementation, which is the biggest challenge in statement-level defense. We choose to sacrifice flexibility for accuracy. SQLStateGuard supports every SQL variant or dialect, but each database in the database server that needs to be protected needs its own dataset and a corresponding detection model trained for it.

The part above the dotted line in Figure 1 shows an application scenario where the user sends a request to the application interface, and then the back-end program does the actual processing and interacts with the database. As seen from the figure, SQLStateGuard works between the back-end program and the database service, and all SQL statements are passed through it for security checks. SQLStateGuard has two working modes: Blocking Mode and Bypass Mode. In Blocking Mode, SQLStateGuard will check the security of SQL statements before deciding whether to forward them to database services, which means this mode can block SQL injections in real-time. Bypass Mode is suitable for services with high-frequency requests but low data sensitivity. In Bypass Mode, SQLStateGuard will forward SQL statements to the database service and make security checks simultaneously, and the two processes are independent of each other. SQLStateGuard in Bypass Mode will alert security operators when it finds a malicious SQL statement rather than blocking SQL injections in real-time.

The architecture of the detection system is shown below the dotted line in Figure 1, and consists of two significant components: SQL Middleware and SQLSG-Net. SQL middleware is mainly used to intercept the SQL statements between the back-end application and the database and block the malicious ones, and its mode of operation is similar to that of a proxy server. SQLSG-Net is a statement-level SQL injection detection network based on gated linear units[17], and its main task is to perform security analysis of SQL statements.

3.2 SQL Middleware

SQL Middleware is the key component used to extract SQL statements and complete SQL injection detection and blocking, and it has three modules: service interface, client interface, and processing module. The service interface is used to establish connections with the back-end program. For the back-end program, the service interface behaves as the interface of the database; the client interface is used to establish connections with the database. For the database, the client interface behaves as the interface of the back-end program. The processing module is used to complete the logic and data forwarding of the SQL middleware.

When SQLStateGuard in Bypass Mode receives the SQL statement request from the back-end program, the processing module will provide the SQL statement from the service interface to the detection network SQLSG-Net for security analysis while using the client interface to forward the SQL statement to the database and send the return of the database back to the back-end program through the service interface. Then, according to the results of the analysis of SQLSG-Net, it will decide whether to send alarm messages to security personnel. When SQLStateGuard in Blocking Mode receives an SQL statement from the back-end program, the processing module will provide the SQL statement from the service interface to SQLSG-Net for security analysis. If SQLSG-Net labels the SQL statement as normal, the processing module forwards this statement to the database and provides the result returned by the database service to the back-end program.

In the real world, attackers usually perform fuzzy tests against WAFs and other protection systems to discover flaws in their rules before performing an effective SQL injection, mainly by inserting special characters. SQL Middleware only blocks SQL statements that are labeled malicious by SQLSG-Net. If the attacker's test vectors do not constitute SQL injections, SQL Middleware will not block them, allowing the attacker to obtain normal returns. It makes it much more difficult for the attacker to find the vulnerability in the interception rules through fuzzy tests as in the case of traditional WAFs, thus increasing the attacker's attack cost.

3.3 SQLSG-Net

The structure of SQLSG-Net is shown in Figure 2. SQLSG-Net extracts both the syntactic structure information from SQL keywords and the semantic information from the whole statement to distinguish malicious SQL statements from normal SQL statements. The SQLSG-Net consists of three layers: Structural Feature Extraction Layer, Semantic Feature Extraction Layer, and SQL Injection Detection Layer.

3.3.1 *Structural Feature Extraction Layer.*

For SQL injection, the core technique is to break the structure of the SQL statement originally designed and expected by the developers, and the difference in structure is one of the main differences between normal SQL statements and malicious SQL statements.

Table 1: Generalization Rules

Target	Generalization Results
Column Name	TK_IDTF
Table Name	TK_IDTF
Numerical Value	0
String	TK_STR
Function	TK_F
System/User Variable	TK_VAR
Alias	TK_IDTF
Comment	TK_C

The input of this layer is original text, and the output is processed text. This layer generalizes the components of the SQL statement, eliminates redundant information, and preserves the structural features of the SQL statement, thus laying the foundation for the subsequent semantic analysis. This layer starts with the construction of an Abstract Syntax Tree (AST) for the SQL statement, which is implemented using the Go-based SQL syntax parser provided by PingCAP[31], and Figure 3 shows an SQL statement and the AST corresponding to it. By traversing the AST, this layer extracts the SQL statement components, such as table names and column names, that are irrelevant to SQL injections and generalizes them to fixed values. Specifically, the generalization targets and results are shown in Table 1. This layer does not generalize SQL keywords such as SELECT, INSERT, BEGIN, SUM, and SQL built-in functions. Unlike other strings, these keywords are essential components of statement structure and directly reflect the function of the statement. Through this layer, the changes caused by user-side input in the SQL statement are eliminated considerably, while the structural features of the SQL statement are preserved.

As mentioned earlier, an attacker usually performs a series of fuzz tests before performing an effective SQL injection. In addition to detecting flaws in the protection system, the attacker may use these fuzz tests to cause syntactic errors in the SQL statements spliced by the back-end program to determine the specific injection techniques. Since this layer parses the syntax of SQL statements, SQLStateGuard can capture the appearance of SQL syntax errors and help security personnel find the attacker and locate the injection point before an effective attack occurs, thus alleviating the offensive and defensive asymmetry of SQL injection attacks and establishing the information advantage for the blue side. Since normal SQL statements do not have syntax errors in this layer, an abnormal statement will be determined if an SQL statement has a syntax error.

3.3.2 *Semantic Feature Extraction Layer.*

The input of this layer is text, and the output is a sentence vector. The SQL statements are treated as a natural language to extract their contextual relationships and semantic features in this layer. Then, the generalized SQL statements output from the Structural Feature Extraction Layer are vectorized to generate numerical sentence vector representations for subsequent SQL injection detection. After obtaining the generalized SQL statements from the Structural Feature Extraction Layer, the semantic feature extraction layer performs segmentation and encoding to get the subscript sequence of the SQL statements, which is illustrated in Figure 4. The "subscript sequence" refers to the sequence number of a particular word in a pre-set vocabulary. In this paper, we use AlbertTokenizer provided by Transformer [37] open-source project to do this part, which is based on SentencePiece [22].

After obtaining the subscript sequence, this layer uses the Albert [23] model to generate a sentence vector. Bert [21] is a widely used pre-training model in NLP, which is based on Transformer and performs well on a range of semantic understanding tasks. Albert is a variant of the Bert, which reduces the number of parameters by weight sharing and matrix decomposition to reduce parameter data, thus decreasing spatial complexity. This layer takes the encoded subscript

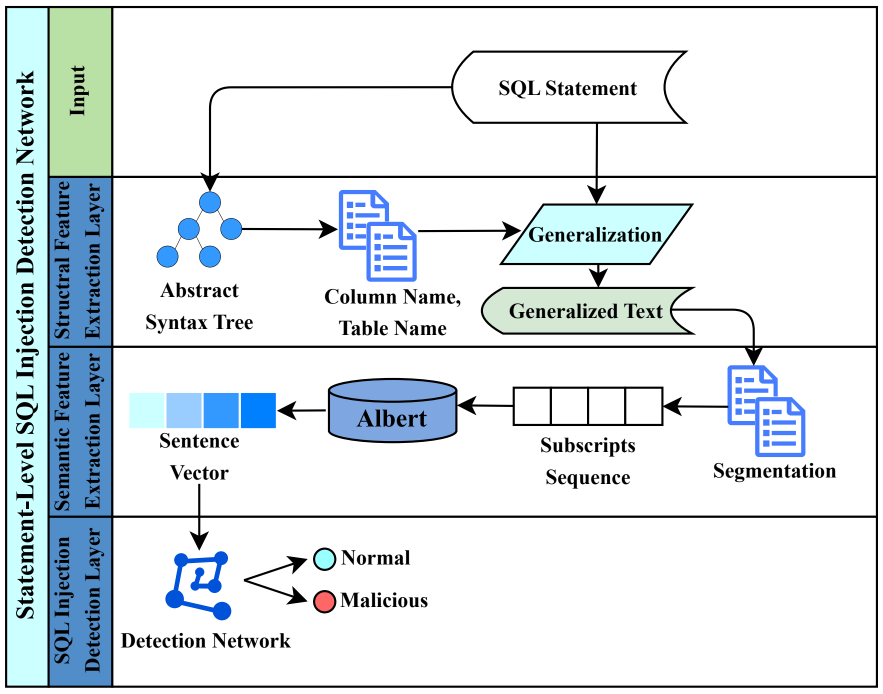

Figure 2: SQLSG-Net Structure

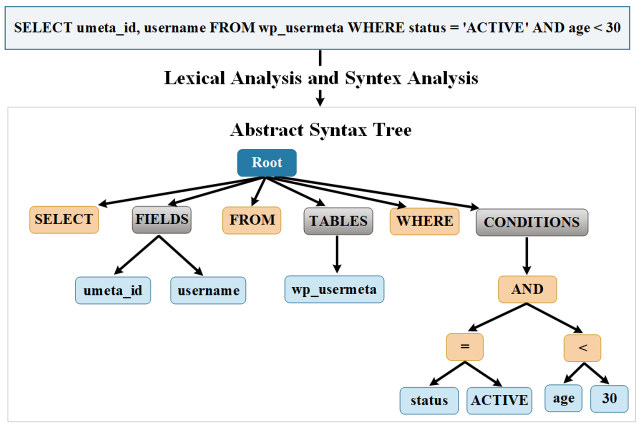
SELECT umeta_id, username FROM wp_usermeta WHERE status = 'ACTIVE' AND age < 30
Lexical Analysis and Syntex Analysis
Abstract Syntax Tree
Root
SELECT
FIELDS
FROM
TABLES
WHERE
CONDITIONS
umeta_id
username
wp_usermeta
AND
=
<
status
ACTIVE
age
30

Figure 3: AST Construction

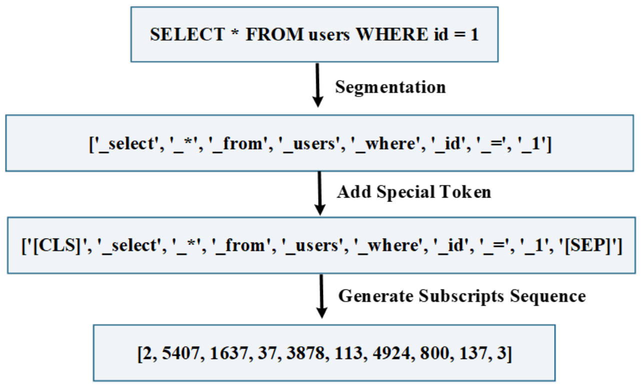
SELECT * FROM users WHERE id = 1
Segmentation
['_select', '_*', '_from', '_users', '_where', '_id', '_=', '_1']
Add Special Token
['[CLS]', '_select', '_*', '_from', '_users', '_where', '_id', '_=', '_1', '[SEP]']
Generate Subscripts Sequence
[2, 5407, 1637, 37, 3878, 113, 4924, 800, 137, 3]

Figure 4: Segmentation and Encoding Example

sequence as the input of the Albert model and obtains its sentence vector output. Albert provides four different sizes of pre-trained models, and this layer uses the most miniature scale model, Albert-Base, which has only 12 million parameters and 768 hidden units.

3.3.3 SQL Injection Detection Layer. This layer is based on Gated Linear Unit (GLU). Its input is the sentence vector from the Semantic Feature Extraction Layer, and its output is the multi-classified SQL injection detection result. Its structure is shown in Figure 5. Introducing GLU for this layer helps the network focus on key information and structures and thus achieve fine-grained SQL injection detection. **Please note that the models in this layer are database-oriented,**

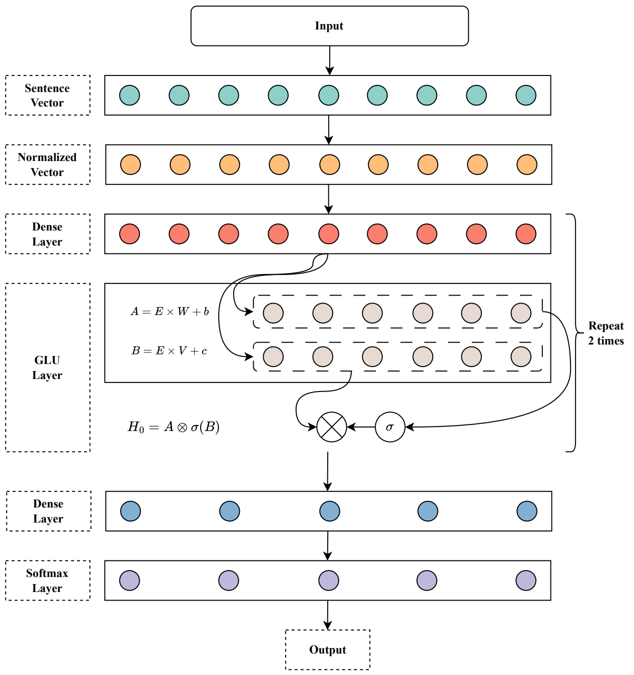

Figure 5: SQL Injection Detection Network Based on GLU

meaning multiple detection models are trained for different databases. Next, we introduce the computation process of GLU used in this layer. We denote the sentence vector input to this layer as X:

$$X = \{X_1, X_2, ..., X_n\} \tag{1}$$

Then, normalize the input sentence vector:

$$N = \frac{x - E(x)}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta\,,\ for\ x\ in\ X \tag{2}$$

Next, use the Dense layer to process the normalized data.

$$D = W_0 N + b \tag{3}$$

Then, use the GLU layer to process the output of the Dense layer.

$$A = E \times W + b \tag{4}$$
$$B = E \times V + c \tag{5}$$

$$H_0 = A \otimes \sigma(B) \tag{6}$$

In this layer, the sentence vector of the input SQL statement is processed by the GLU-based network, and Dense and Softmax obtain the final classification probability output:

$$Output = Activation((Input \cdot Weight) + Bias) \tag{7}$$

In Eq. 7, $Input$ is the input vector of Dense, $Weight$ is the weight matrix, $Bias$ is the bias matrix, and $Activation$ is the activation function. The main work of Dense is to map the GLU Layer output feature vector to the output space by a nonlinear transformation based on the fully connected structure:

$$Z = W_{m \times n} \cdot X_{n \times d} + B_{m \times d} \tag{8}$$

In Eq. 8, m is the number of neurons of Dense, d is the dimensionality of the input vector, and n is the number of samples. The W_{m*n} is the weight matrix, X_{n*d} is the matrix composed of samples, and B_{m*d} is the bias matrix. Finally, Softmax maps the output of Dense to a probability distribution:

$$Softmax(z_i) = \frac{e^{z_i}}{\sum_{i=1}^{m} e^{z_i}} \tag{9}$$

Where z_i is the i-th input value, the probability of the sample belonging to each category is obtained by Softmax, and the category with the highest probability is used as the result of SQL injection detection.

3.4 Dataset Generation Process for SQLStateGuard

Even experienced security experts cannot tell if a statement is an SQL injection without understanding the code implementation. It directly leads to the lack of statement-level SQL injection datasets, making statement-level SQL injection detection challenging to implement. This part shows how to generate statement-level datasets for the databases that need to be protected by SQLStateGuard.

SQLStateGuard requires both normal and malicious samples for training. To generate these samples, the dataset generation process proposed in this paper can be briefly divided into three steps: injection payload collection, template statement collection, and statement splicing.

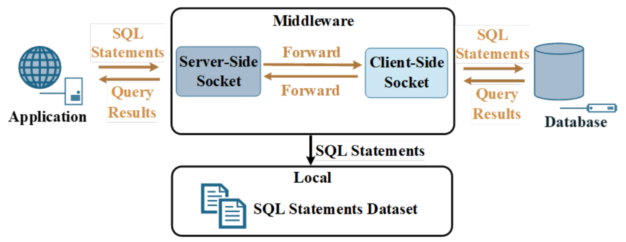

Figure 6: Template Statement Collection

We first need to complete the injection payload collection. Since there are many payload-level SQL injection detection studies and the related datasets are rich, this paper collects payloads from existing public datasets such as Data-Hacking [34] and LibInjection [13]. In addition, we also collect the payloads used by automated testing tools such as SqlMap [5]. Then, we conduct template statement collection, a step that is also done based on middleware. As shown in Figure 6, we collect the SQL statements generated by performing functional tests after setting up middleware between the application and the database. These statements are both *normal samples* in the dataset and template statements used to construct malicious samples.

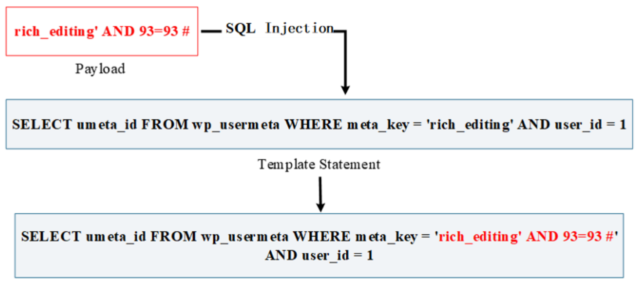
rich_editing' AND 93=93 #
Payload
SQL Injection
SELECT umeta_id FROM wp_usermeta WHERE meta_key = 'rich_editing' AND user_id = 1
Template Statement
SELECT umeta_id FROM wp_usermeta WHERE meta_key = 'rich_editing' AND 93=93 #' AND user_id = 1

Figure 7: Payload-to-Template Splicing

Finally, we splice the payloads into the template statements and label the output statements as malicious samples. Figure 7 is an example of this process. In principle, SQLStateGuard accomplishes injection detection by determining whether the statement to be tested is "abnormal" or not. Therefore, this dataset generation process is the basis of SQLStateGuard.

4 EXPERIMENTS

This paper uses different datasets and the following questions to evaluate SQLStateGuard:

- **RQ1:** Can SQLStateGuard effectively detect SQL injections against database services?
- **RQ2:** Is SQLSG-Net based on GLU network better than other implementations?
- **RQ3:** Is the detection cost of SQLStateGuard acceptable in real-world environments?
- **RQ4:** Is SQLStateGuard better than existing SQL injection detection approaches?

4.1 Datasets

This paper uses two statement-level SQL injection datasets generated by the process in Section 3.4 and two HTTP request datasets constructed based on them. The statement-level SQL dataset is used to train and evaluate the SQLSG-Net in SQLStateGuard, while the HTTP request dataset is used to compare SQLStateGuard and other traffic-based approaches in a side-by-side manner.

Table 2: Details of Statement-Level SQL Injection Datasets

Dataset	Dataset ID	Type	Count
Self-Built	I	Malicious	74,555
		Normal	75,000
WordPress	II	Malicious	71,757
		Normal	79,316

Table 3: Details of Malicious Statement-Level SQL Injection Datasets

Detail Type	Dataset I	Dataset II
Time-based	1,588	1,392
Boolean-based	2,196	2,012
Error-based	3,491	3,206
Tautology-based	9,135	8,359
Union-based	58,145	56,788

The details of the statement-level SQL injection datasets used in this paper are shown in Table 2 and Table 3, which come from a self-built Web application (Dataset I) and a WordPress [38] instance (Dataset II), respectively. The self-built Web application is an Application Programming Interface (API) gateway based on Sanic Framework [33]. WordPress is the most widely used open-source content management system in the world. The total count of SQL statements in the two datasets exceeds 300,000.

Table 4: Details of HTTP Request Dataset

Dataset	Dataset ID	Main Label	Count
Self-Built	III	Malicious	22,334
		Normal	22,534
WordPress	IV	Malicious	13,421
		Normal	13,335

The HTTP request datasets based on the statement-level SQL injection datasets are similarly divided into Dataset III (self-built Web application) and Dataset IV (WordPress). The details of these two datasets are shown in Table 4. Since the application processing of an HTTP request may generate multiple SQL statements, the number of samples in the HTTP request dataset and the statement-level SQL injection dataset is different.

Please note that since some abnormal samples are injected based on syntax errors, they are blocked at the structural feature extraction layer. So, for RQ 1, 2, and 3, the dataset is all SQL statements that pass through the structural feature extraction layer. The relationship between the original dataset and the dataset that passes through the structural feature extraction layer is shown in Table 5.

4.2 Implementation, Setup, and Metrics

In terms of the hardware environment, we use a GPU server (NVIDIA RTX A6000) to complete the training and evaluation of SQLStateGuard. In terms of software implementation, this paper implements a prototype system of SQLStateGuard based on Python [32] 3.8.5, and other approaches involved in the experiments come from their official implementations or the implementations provided by the authors.

We use *Accuracy*, *Recall*, *Precision*, *F1-Score*, and *False Positive Rate* ($FPR = \frac{FP}{TN+FP}$) to evaluate SQLStateGuard. Considering the unbalanced datasets used in evaluating SQLStateGuard, we will use weighted metrics to complete the multi-classification evaluation of SQLStateGuard. In this paper, the weight of each classification is defined as the proportion of samples in the total sample and is denoted as $W_i = \frac{\sum_i C_{i,j}}{\sum_i \sum_j C_{i,j}}$. Then, we calculate the metrics in a multi-classification scenario by weighting the average for a more fine-grained evaluation of SQLStateGuard. We have:

$$Accuracy_{avg} = \sum_i W_i * Accuracy_i \tag{10}$$

$$Precision_{avg} = \sum_i W_i * Precision_i \tag{11}$$

$$Recall_{avg} = \sum_i W_i * Recall_i \tag{12}$$

$$F1 - Score_{avg} = \sum_i W_i * F1 - Score_i \tag{13}$$

4.3 Training and Detection Performance (RQ1)

To answer RQ1, two statement-level SQL injection datasets (Dataset I and Dataset II) are used to train SQLSG-Net. In this experiment, the SQL statements in the dataset are firstly converted into sentence vectors by the Structural Feature Extraction Layer and the Semantic Feature Extraction Layer in SQLSG-Net while preserving the labels. Then, these sentence vectors and their labels are used to train the SQL injection detection models based on GLU.

In this experiment, we divide the training set, test set, and validation set according to the ratio of 6:2:2. First, we use Dataset I for the training evaluation of SQLSG-Net, derived from a self-built Web application to evaluate the SQL injection detection performance for APIs.

After the training based on Dataset I is completed, the performance of SQL injection detection is evaluated based on binary classification and multi-classification criteria using the test set divided from Dataset I, as shown in Table 6. The binary classification evaluation only distinguishes between *normal* and *malicious*, which mainly shows the performance of SQLSG-Net to detect SQL injections. The multi-classification evaluation is more fine-grained and evaluates SQLStateGuard's ability to distinguish different SQL injection types, and better differentiation means that SQLStateGuard can provide more information to security operators and help them locate and analyze security threats.

Then, we use Dataset II to perform training evaluation on SQLSG-Net. Compared with Dataset I, the content-driven Dataset II is more complex and has a greater variety of SQL statement structures. Here are two injected statements of the same type, and we can see that the structures of the statements in Dataset II are more complex than those in Dataset I:

Listing 1: Examples of Malicious Statements in Dataset I and Dataset II

```
A Time-based Injection in Dataset I
select user_id, password from info.users where
    username='admin'; select sleep(5);-- '

A Time-based Injection in Dataset II
SELECT SQL_CALC_FOUND_ROWS wp_posts.ID FROM
    wp_posts WHERE 1=1 AND wp_posts.post_type = '
    wp_block' AND ((wp_posts.post_status = 'publish'));
    SELECT SLEEP(5)-- ')) ORDER BY wp_posts.
    post_date DESC LIMIT 0, 100
```

After the training based on Dataset II is completed, we use the test set divided from Dataset II to evaluate SQLStateGuard, and the results are shown in Table 7. We can see that SQLStateGuard also works well on Dataset II, but the complexity and diversity slightly degrade the detection performance.

In summary, this experiment gives a pretty satisfactory answer to RQ1. SQLStateGuard has excellent SQL injection detection capability and can accurately identify the type of SQL injections, thus helping security personnel locate and analyze security threats.

Case Study I: The results show that the overall performance of the multi-classification evaluation is reduced compared to the binary-classification evaluation. After analyzing each category in the datasets, we found that some malicious samples suffered from multiple types of SQL injections, such as the payload "*admin' AND SLEEP(5) AND 'fhez'='fhez*", which combines both time-blind and tautology-based SQL injections. However, these samples are only classified into a single category in the datasets, so the model may predict them as inconsistent with the tagged labels.

Case Study II: The following two SQL statements illustrate examples of those captured and missed by our system.

Table 5: The Relationship between the Original Dataset and the Dataset that Passes through the Structural Feature Extraction Layer

	Original		Structural Feature Extraction	
	Normal	Malicious	Normal	Malicious
Dataset I	75,000	74,555	75,000	74,555
Dataset II	79,316	71,757	79,316	64,541

Table 6: Detection Evaluation on Test Set of Dataset I

Type	Accuracy	Recall	Precision	F1-Score	FPR
Binary	100.00%	100.00%	100.00%	100.00%	0.00%
Multiple	99.05%	98.58%	97.43%	97.98%	N/A

* Multi-classification metrics are weighted averages

Table 7: Detection Evaluation on Test Set of Dataset II

Type	Accuracy	Recall	Precision	F1-Score	FPR
Binary	99.97%	99.95%	99.98%	99.97%	>0.00%
Multiple	98.93%	95.06%	96.06%	95.55%	N/A

* Multi-classification metrics are weighted averages

We believe the first statement, which was not captured, contains a large number of query units, which likely obscured its malicious characteristics. In contrast, the second statement, with fewer query units, was easily detected by our system. However, cases like the first statement, where the system fails to capture it, are extremely rare (only 6 in datasets I and II), and thus we believe they have minimal impact on our system's overall performance.

Listing 2: Examples of Malicious Statements in Dataset I and Dataset II

```
An Malicious SQL System didn't Caught
SELECT COUNT(*) FROM wp_term_relationships,
    wp_posts WHERE wp_posts.ID =
    wp_term_relationships.object_id AND post_status IN
    ('publish') AND post_type IN ('post ') AND
    term_taxonomy_id = 1 and 1=0 union select 1,
    concat_ws(0x3a, version () , user () , database () )
      ,3,4,5,6,7,8,9,10,11,12--

An Malicious SQL System Caught
SELECT meta_id FROM wp_postmeta WHERE meta_key
    = '_edit_lock' AND post_id = 1 and 1=0 union select
    1,2,3, concat_ws(0x3a, version () , user () , database () )
    ,5--
```

4.4 Implementation Comparison of SQLSG-Net (RQ2)

This experiment focuses on verifying whether SQLSG-Net's implementation based on GLU has a comparative advantage over other solutions. In this experiment, we use detection models from other learning-based techniques to replace the SQL Injection Detection Layer in SQLSG-Net and then compare the detection effectiveness of the different implementations. There are two other learning-based models involved in this experiment: SVM [9] and LSTM [16].

The results of this experiment are shown in Table 8. It is obvious from the table that SVM has poor performance, especially its false positive rate, which makes SQL injection blocking impractical. The overall performance of the models on Dataset II is slightly worse than that of Dataset I, especially when performing the multi-classification evaluation, mainly because WordPress generates a wider variety of SQL statement types than self-built web applications.

Besides, we can see the performance of LSTM is pretty good, though still inferior to the GLU network used by SQLSG-Net. It is an excellent example of the effectiveness of SQLStateGuard's architecture and feature engineering design. Finally, SQLSG-Net has the best performance in all evaluation metrics, demonstrating the comparative advantage of its implementation over other solutions.

Table 8: Detection Layer Evaluation on Dataset I and Dataset II

Implementation	Dataset	Classification Type	Accuracy	Recall	Precision	F1-Score	FPR
SVM	I	Binary	99.92%	99.84%	100.00%	99.92%	0.00%
		Multiple	97.26%	77.22%	88.94%	81.16%	N/A
	II	Binary	97.81%	99.68%	95.87%	97.74%	3.89%
		Multiple	96.27%	65.71%	86.43%	67.74%	N/A
LSTM	I	Binary	99.99%	100.00%	99.97%	99.99%	>0.00%
		Multiple	98.37%	89.60%	91.76%	90.61%	N/A
	II	Binary	99.95%	99.81%	99.92%	99.94%	>0.00%
		Multiple	97.77%	80.53%	89.51%	83.61%	N/A
SQLSG-Net	I	Binary	100.00%	100.00%	100.00%	100.00%	0.00%
		Multiple	99.05%	98.58%	97.43%	97.98%	N/A
	II	Binary	99.97%	99.95%	99.98%	99.97%	>0.00%
		Multiple	98.93%	95.06%	96.06%	95.55%	N/A

* Multi-classification metrics are weighted averages

4.5 Cost Evaluation (RQ3)

This experiment is designed to answer RQ3. The presence of additional security detection will inevitably lead to increased latency, and whether these additional latencies will seriously interfere with the use of database services by back-end programs is one of the critical questions of whether SQLStateGuard can be deployed in the real world. In this experiment, we combine Dataset I and Dataset II into one and evaluate the detection time overhead of SQLStateGuard, and the results are shown in Table 9.

The feature extraction layers of SQLSG-Net mainly cause the overhead of SQLStateGuard. The additional time overhead of 10.499*ms* for a single SQL statement can satisfy the need for SQL injection blocking in most application scenarios. However, if a single application interface call contains many SQL statements, SQLStateGuard may slow down the result return speed. Therefore, this paper designs a Bypass Mode to provide SQL injection detection for latency-sensitive scenarios.

4.6 System-Level Comparative Evaluation (RQ4)

To answer RQ4, two existing approaches, Luo et al. [28] and Yong et al. [43], are selected for comparison with SQLStateGuard in this experiment. The first approach is based on log analysis and CNN. The second is a malicious traffic detection approach based on Hidden Markov Model (HMM).

Since both of them analyze samples at the HTTP traffic level, and the analysis results are only *normal* and *malicious*, we use two HTTP request datasets (Dataset III and Dataset IV) to complete this experiment and perform a system-level evaluation based on binary-classification metrics.

In this experiment, we first parse the HTTP requests in Dataset III and Dataset IV into the log formats required by these two approaches and respectively evaluate their performance. The SQLStateGuard proposed in this paper works at the statement level, and usually an HTTP request contains more than one SQL statement, which means that for a malicious request, as long as any of its corresponding statements is determined to be malicious, the HTTP request is malicious, which greatly increases the success rate of SQLStateGuard's judgment. If the back-end application sends an SQL statement to the database service while processing an HTTP request, SQLStateGuard will record and analyze this statement. The results of this experiment are shown in Table 10.

The results show that the false positive is the most critical weakness of existing approaches. By analyzing their false positive samples, we found that Luo et al. labeled unfamiliar words (e.g., usernames not in the corpus) malicious due to the limited generalization ability. At the same time, both Luo et al. and Yong et al. have difficulty coping with the complex traffic variation. It is prone to false positives when detecting samples containing special characters and long strings of numbers and letters.

Overall, compared with existing approaches, SQLStateGuard has significant advantages in all metrics of SQL injection detection, including Dataset IV with relatively complex samples, which answers RQ4 very well.

Table 9: Average Time Overhead of Single-Statement Detection

Working Mode	SQL Middleware	SQLSG-Net			Total
		Structural	Semantic	Detection	
Bypass	0.31ms	0	0	0	0.31ms
Blocking	0.31ms	0.37ms	9.97ms	0.16ms	10.50ms

Table 10: Comparison Evaluation Results

	Luo et al.		Yong et al.		Ours	
Dataset	III	IV	III	IV	III	IV
Accuracy	88.68%	86.43%	98.46%	93.35%	100.00%	100.00%
Recall	88.68%	86.43%	98.46%	93.35%	100.00%	100.00%
Precision	89.15%	87.25%	98.51%	93.59%	99.99%	100.00%
F1-Score	88.65%	86.35%	98.46%	93.34%	100.00%	100.00%
FPR	16.70%	21.10%	0.00%	10.41%	>0.00%	0.00%

5 DISCUSSION

5.1 Weakness of SQLStateGuard

Experiments show that the SQLStateGuard proposed in this paper is significantly better at detecting SQL injections than existing approaches, making it seem to have a severe overfitting risk - but this is not the case. SQLStateGuard is not the end of the story, and its impressive detection performance is mainly due to two factors:

First, SQLStateGuard needs to train different detection models for different databases. This design significantly reduces the variation of normal statements, thus making malicious SQL statements more prominent. On the other hand, it also increases the deployment costs, making SQLStateGuard very challenging in protecting the applications designed on new agile development models such as DevOps. Therefore, SQLStateGuard only suits the services provided by traditional organizations (e.g., government), which are always data-rich and security-critical but have very little code change after deployment.

Second, SQLStateGuard simultaneously uses different dimensions of SQL statement information for feature extraction. Combined with a GLU-based network, tiny differences between the normal and the injected statements can be effectively found. However, this design also increases the time overhead, which affects the real-time performance of SQLStateGuard.

In the current context of the popularity of cloud computing, there may be so many requests for access to a particular service that each request needs to be responded to with very low latency. For this situation, we designed bypass mode for performance sensitive situation, in this mode our purpose is not to block the attack, but for quick warning, the latency of 0.3ms in this mode is acceptable in most of the situations. However, if the service is still very performance sensitive or there are huge amount of requests, we can deploy our solution using, for example, port mirroring, which can eliminate the latency.

5.2 Future Improvements

In the future, we will mainly improve SQLStateGuard in two aspects. First, we will reduce the number of models as much as possible so that each model can serve more databases, thus reducing model training and deployment costs. Second, we will improve the feature engineering process of SQLStateGuard and shorten the feature extraction time so that SQLStateGuard can run in Blocking Mode more often and achieve better SQL injection protection.

In addition, we will also try to improve the adaptability of SQLStateGuard to code changes by introducing a novel scoring mechanism to reduce the number of retraining while ensuring low FPR, thus meeting the needs of emerging software engineering models like DevOps.

6 CONCLUSION

In this paper, we propose SQLStateGuard, a novel statement-level SQL injection detection approach. SQLStateGuard is based on the idea of RASP, incorporates middleware into the database service process, and implements statement-level SQL injection detection based on GLU. It is proved that SQLStateGuard can accurately detect SQL injections with a very low FPR compared to existing approaches and even identify the tactical types of SQL injections quite accurately.

SQLStateGuard's SQL injection detection capabilities are overwhelmingly superior to existing approaches.

ACKNOWLEDGMENTS

This work is supported by HY-Project under No.4E49EFF3, the Gansu Province Key Research and Development Plan - Industrial Project under Grant No. 22YF7GA004, and Supercomputing Center of Lanzhou University.

REFERENCES

[1] Stanislav Abaimov and Giuseppe Bianchi. 2019. CODDLE: Code-Injection Detection With Deep Learning. *IEEE Access* 7 (2019), 128617–128627. https://doi.org/10.1109/ACCESS.2019.2939870

[2] Edward Amoroso. 2018. Recent progress in software security. *IEEE Software* 35, 2 (2018), 11–13.

[3] Nuno Antunes and Marco Vieira. 2010. Benchmarking vulnerability detection tools for web services. In *2010 IEEE International Conference on Web Services*. IEEE, 203–210.

[4] Matthew Bach-Nutman. 2020. Understanding the top 10 owasp vulnerabilities. *arXiv preprint arXiv:2012.09960* (2020).

[5] Miroslav Stampar Bernardo Damele Assumpcao Guimaraes. 2016. SQLMAP. https://sqlmap.org/

[6] Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. 2010. CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks. *Acm Transactions on Information & System Security* 13, 2 (2010), 398–404.

[7] Stephen W Boyd and Angelos D Keromytis. 2004. SQLrand: Preventing SQL injection attacks. In *International conference on applied cryptography and network security*. Springer, 292–302.

[8] Xiaoyi Chen, Qingni Shen, Peng Cheng, Yongqiang Xiong, and Zhonghai Wu. 2022. RuleCache: Accelerating Web Application Firewalls by On-line Learning Traffic Patterns. In *2022 IEEE International Conference on Web Services (ICWS)*. IEEE, 229–239.

[9] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine learning* 20, 3 (1995), 273–297.

[10] Ignacio Samuel Crespo-Martínez, Adrián Campazas-Vega, Ángel Manuel Guerrero-Higueras, Virginia Riego-DelCastillo, Claudia Álvarez Aparicio, and Camino Fernández-Llamas. 2023. SQL Injection Attack Detection in Network Flow Data. *Computers & Security* 127 (2023), 103093. https://doi.org/10.1016/j.cose.2023.103093

[11] Johannes Dahse and Jörg Schwenk. 2010. RIPS-A static source code analyser for vulnerabilities in PHP scripts. In *Seminar Work (Seminer Çalismasi). Horst Görtz Institute Ruhr-University Bochum*. Citeseer.

[12] Jesús E Díaz-Verdejo, Rafael Estepa Alonso, Antonio Estepa Alonso, and German Madinabeitia. 2022. A critical review of the techniques used for anomaly detection of HTTP-based attacks: taxonomy, limitations and open challenges. *Computers & Security* (2022), 102997.

[13] Nick Galbreath. 2014. Libinjection. https://github.com/client9/libinjection

[14] Ning Guo, Xiaoyong Li, Hui Yin, and Yali Gao. 2019. Vulhunter: An automated vulnerability detection system based on deep learning and bytecode. In *International Conference on Information and Communications Security*. Springer, 199–218.

[15] William GJ Halfond and Alessandro Orso. 2005. AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 174–183.

[16] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-term Memory. *Neural computation* 9 (12 1997), 1735–80. https://doi.org/10.1162/neco.1997.9.8.1735

[17] Weizhe Hua, Zihang Dai, Hanxiao Liu, and Quoc Le. 2022. Transformer quality in linear time. In *International conference on machine learning*. PMLR, 9099–9117.

[18] Anamika Joshi and V Geetha. 2014. SQL Injection detection using machine learning. In *2014 international conference on control, instrumentation, communication and computational technologies (ICCICCT)*. IEEE, 1111–1115.

[19] Mei Junjin. 2009. An Approach for SQL Injection Vulnerability Detection. *2009 Sixth International Conference on Information Technology: New Generations* (2009), 1411–1414.

[20] Konstantinos Kemalis and Theodores Tzouramanis. 2008. SQL-IDS: a specification-based approach for SQL-injection detection. In *Proceedings of the 2008 ACM symposium on Applied computing*. 2153–2158.

[21] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, Vol. 1. Minneapolis, Minnesota, 2.

[22] Taku Kudo and John Richardson. 2018. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226* (2018).

[23] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942* (2019).

[24] Q. Li, W. Li, J. Wang, and M. Cheng. 2019. A SQL Injection Detection Method based on Adaptive Deep Forest. *IEEE Access* PP, 99 (2019), 1–1.

[25] Xin Liu, Qingchen Yu, Xiaokang Zhou, and Qingguo Zhou. 2018. OwlEye: An Advanced Detection System of Web Attacks Based on HMM. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*. IEEE, 200–207.

[26] Xin Liu, Wenqiang Zhang, Xiaokang Zhou, and Qingguo Zhou. 2021. MECGuard: GRU enhanced attack detection in Mobile Edge Computing environment. *Computer Communications* 172 (2021), 1–9.

[27] Dongzhe Lu, Jinlong Fei, and Long Liu. 2023. A Semantic Learning-Based SQL Injection Attack Detection Technology. *Electronics* 12, 6 (2023), 1344. Issue 6. https://doi.org/10.3390/electronics12061344

[28] Ao Luo, Wei Huang, and Wenqing Fan. 2019. A CNN-based Approach to the Detection of SQL Injection Attacks. In *2019 IEEE/ACIS 18th International Conference on Computer and Information Science (ICIS)*. IEEE, 320–324.

[29] J. Mei. 2009. An Approach for SQL Injection Vulnerability Detection. In *Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on*.

[30] Basem Ibrahim Mukhtar and Marianne A Azer. 2020. Evaluating the modsecurity web application firewall against sql injection attacks. In *2020 15th International Conference on Computer Engineering and Systems (ICCES)*. IEEE, 1–6.

[31] PingCAP. 2022. MySQL Parser. https://github.com/pingcap/parser

[32] Python.org. 2022. The official home of the Python Programming Language. https://python.org/.

[33] Sanic. 2022. Sanic Framework. https://sanic.dev/en/.

[34] SuperCowPowers. 2013. Data-Hacking. https://github.com/SuperCowPowers/data_hacking/tree/master/sql_injection/data

[35] Peng Tang, Weidong Qiu, Zheng Huang, Huijuan Lian, and Guozhen Liu. 2020. Detection of SQL injection based on artificial neural network. *Knowl. Based Syst.* 190 (2020), 105528.

[36] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. *ACM*

Sigplan Notices 44, 6 (2009), 87–97.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.

[38] WordPress. 2022. Blog Tool, Publishing Platform, and CMS. https://wordpress.org/.

[39] X. R. Wu and Ppk Chan. 2012. SQL injection attacks detection in adversarial environments by k-centers. In *International Conference on Machine Learning & Cybernetics*.

[40] Z. Xiao, Z. Zhou, W. Yang, and C. Deng. 2017. An approach for SQL injection detection based on behavior and response analysis. In *2017 IEEE 9th International Conference on Communication Software and Networks (ICCSN)*.

[41] Liu Xin, Liu Ziang, Zhang Yingli, Zhang Wenqiang, Lv Dong, and Zhou Qingguo. 2022. TCN enhanced novel malicious traffic detection for IoT devices. *Connection Science* 34, 1 (2022), 1322–1341.

[42] Wenchuan Yang, Wen Zuo, and Baojiang Cui. 2019. Detecting Malicious URLs via a Keyword-Based Convolutional Gated-Recurrent-Unit Neural Network. *IEEE Access* 7 (2019), 29891–29900. https://doi.org/10.1109/ACCESS.2019.2895751

[43] Binbin Yong, Xin Liu, Qingchen Yu, Liang Huang, and Qingguo Zhou. 2019. Malicious Web traffic detection for Internet of Things environments. *Computers & Electrical Engineering* 77 (2019), 260–272.

[44] Huafeng Zhang, Bo Zhao, Hui Yuan, Jinxiong Zhao, Xiaobin Yan, and Fangjun Li. 2019. SQL injection detection based on deep belief network. In *Proceedings of the 3rd International Conference on Computer Science and Application Engineering*. 1–6.

[45] Su Zhang and Ying Zhang. 2022. Privacy Leakage Vulnerability Detection for Privacy-Preserving Computation Services. In *2022 IEEE International Conference on Web Services (ICWS)*. 219–228. https://doi.org/10.1109/ICWS55610.2022.00043

[46] Yaqin Zhang, Duohe Ma, Xiaoyan Sun, Kai Chen, and Feng Liu. 2020. WGT: Thwarting Web Attacks Through Web Gene Tree-based Moving Target Defense. In *2020 IEEE International Conference on Web Services (ICWS)*. 364–371. https://doi.org/10.1109/ICWS49710.2020.00054

[47] F ZHAOY, G XIONG, et al. 2016. SQL injection behavior detection method for network environment [J]. *Journal of Communications* 37, 2 (2016), 88–97.

[48] M. Zolotukhin, T. Hämäläinen, T. Kokkonen, and J. Siltanen. 2014. Analysis of HTTP Requests for Anomaly Detection of Web Attacks. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*. 406–411. https://doi.org/10.1109/DASC.2014.79