future internet

Article

SSQLi: A Black-Box Adversarial Attack Method for SQL Injection Based on Reinforcement Learning

Yuting Guan , Junjiang He *, Tao Li, Hui Zhao and Baoqiang Ma

School of cyber Science and Engineering, Sichuan University, Chengdu 610065, China
* Correspondence: hejunjiang@scu.edu.cn

Abstract: SQL injection is a highly detrimental web attack technique that can result in significant data leakage and compromise system integrity. To counteract the harm caused by such attacks, researchers have devoted much attention to the examination of SQL injection detection techniques, which have progressed from traditional signature-based detection methods to machine- and deep-learning-based detection models. These detection techniques have demonstrated promising results on existing datasets; however, most studies have overlooked the impact of adversarial attacks, particularly black-box adversarial attacks, on detection methods. This study addressed the shortcomings of current SQL injection detection techniques and proposed a reinforcement-learning-based black-box adversarial attack method. The proposal included an innovative vector transformation approach for the original SQL injection payload, a comprehensive attack-rule matrix, and a reinforcement-learning-based method for the adaptive generation of adversarial examples. Our approach was evaluated on existing web application firewalls (WAF) and detection models based on machine- and deep-learning methods, and the generated adversarial examples successfully bypassed the detection method at a rate of up to 97.39%. Furthermore, there was a substantial decrease in the detection accuracy of the model after multiple attacks had been carried out on the detection model via the adversarial examples.

Keywords: SQL injection; adversarial attack; black-box attack; reinforcement learning; attack-rule matrix; adversarial example

check for updates

Citation: Guan, Y.; He, J.; Li, T.; Zhao, H.; Ma, B. SSQLi: A Black-Box Adversarial Attack Method for SQL Injection Based on Reinforcement Learning. *Future Internet* **2023**, *15*, 133. https://doi.org/10.3390/fi15040133

Academic Editors: Mazdak Zamani, Rohit Tanwar, Behrang Samadi and Touraj Khodadadi

Received: 7 March 2023
Revised: 24 March 2023
Accepted: 28 March 2023
Published: 30 March 2023

1. Introduction

The proliferation of web applications has led to an increase in the frequency and the severity of SQL injection attacks that pose a significant threat to data security. As one of the top 10 cybersecurity risks according to the Open Web Application Security Project (OWASP) [1], SQL injection attacks have been responsible for the discovery of over 600 zero-day vulnerabilities in 2021 alone, according to CVE Details [2]. By exploiting such attacks, hackers can easily manipulate sensitive information on a website and damage network infrastructure. Therefore, the detection of SQL injection attacks is crucial for the protection of both web and data security. Traditional SQL injection detectors, such as web application firewalls (WAFs), rely on a set of detection rules based on semantic analysis and regex match. However, the flexible nature of SQL syntax has made it challenging for the WAF rule library to include all possible SQL injection attack scenarios, resulting in a lower detection efficiency. In recent years, researchers have applied machine-learning and deep-learning approaches for SQL injection detection. Generally, these approaches have involved the translation of queries into word vectors and the extraction of consecutive characters as features to identify malicious requests [3–5]; although these studies have achieved remarkable results, vulnerability to adversarial attacks continues to be a significant challenge.

The adversarial attack can generate adversarial examples by fine-tuning the original example so that the detection model misinterprets it [6]. Cross-Site Scripting (XSS) attacks and SQL injection attacks are both types of text-based injection attack methods. XSS attacks can exploit the victim's browser to execute malicious code, which can cause significant damage by stealing or modifying sensitive information. Therefore, in recent years, the detection of XSS attacks has received considerable attention in research on web application security. Adversarial attacks on XSS detection models have received considerable attention in recent years, with researchers proposing various methods, such as GAN-based [7] and reinforcement-learning-based [8–10] adversarial attack methods, significantly diminishing the ability of XSS detection models to accurately identify such attacks. In contrast, the research on the adversarial attacks against the SQL injection detection models is only in the early stages, with a limited number of studies conducted in this area. Liu et al. proposed the use of a transformer model to learn the semantic structure of SQL statements and translate the original SQL injection payloads into different forms of attack statements with the same intent [11]. Valena et al. constructed a mutation rule tree for SQL injection payloads, where the system randomly selected mutation rules in the tree to mutate the payload samples, bypassing the detector [12]. However, these studies have had limitations as they have not included black-box attack methods that better resemble a real-world scenario; in addition, they have relied on a single attack strategy and have been unable to stack. Therefore, researchers have proposed reinforcement-learning-based frameworks for black-box WAF bypass and validated their schemes with SQL injection attacks [13–16]. However, these studies only tested the effectiveness of their frameworks on a limited number of WAFs. Moreover, they did not fully investigate the characteristics of SQL injection attacks and lacked effective state-vector-representation algorithms for SQL statements.

To study the potential influence of adversarial examples on SQL injection detection, we proposed SSQLi, a novel adversarial example generator for SQL injection that is based on reinforcement learning. We named our approach SSQLi because it is an adversarial attack method against SQL injection detectors based on the soft actor–critic (SAC) algorithm in reinforcement learning. We proposed a comprehensive attack strategy matrix by mining the vulnerabilities of existing SQL injection detection models. We then employed a reinforcement-learning framework, in which the results returned by the detector were treated as rewards from the environment to the agent and trained the agent to select attack strategies based on the detection results, without knowing the specific parameters of the detector. Additionally, the inclusion of entropy in the training process promoted a more diverse selection of attack strategies by the agent. Finally, we tested SSQLi on 10 different types of detectors and provided suggestions for detector improvements based on the experimental results. In summary, the main contributions of this paper were the following:

- State-vector-representation algorithm: We investigated the features of SQL query statements and proposed an approach for converting SQL query statements into vectors that could capture the essential characteristics of the original statement and provide more comprehensive feature extraction.
- Attack strategy matrix: We examined the vulnerabilities of existing detectors and proposed an attack strategy matrix containing a total of 14 attack strategies based on their vulnerabilities. The 14 attack strategies were developed based on the reports from the open-source security intelligence team and information provided in the official MySQL manual. These strategies include three major categories: syntax transformation strategy, syntax analysis interference strategy, and semantic analysis interference strategy. The goal of these attack strategies was to conceal the attack characteristics of the SQL injection payload, and all the attack strategies in the matrix could be used in tandem, regardless of overlap, without changing the attack target of the original payload.
- SQL injection adversarial attack method: We proposed an adversarial attack method based on the soft actor–critic (SAC) algorithm, in which the agent adapted the attack strategy to generate adversarial examples based on the feedback from the detection model.
- We constructed eight machine- and deep-learning-based SQL injection detection models, according to mainstream detection frameworks. We tested our attack methods on these detection models and two open-source WAFs. SSQLi resulted in a significant

reduction in the detection rate for all these detectors, with three of them showing a detection rate reduction of approximately 90%.

2. Related Work

Recently, much research has focused on adversarial learning against various fields. Guo et al. proposed SimBA based on a black-box framework to deceive image recognition applications in real-world scenarios [17]. Goswami et al. designed a framework of tools for natural language adversarial attacks [18]. Morris et al. evaluated adversarial attacks against DNNs on facial recognition files [19]. Zhang et al. proposed a poisoning attack in recommender systems that generates plausible fake profiles using generative adversarial networks (GAN). They used the adversarial approach of generative adversarial networks (GAN) to generate reasonable fake profiles that resemble normal profiles, and achieved better results than previous state-of-the-art attacks attacking performance [20].

In the web security field, plenty of research has been focused on detection using machine-learning and deep-learning frameworks. Zhou et al. utilized ensemble-learning based on Bayesian networks to detect XSS [21]. Kar et al. proposed a graph feature framework to detect SQL injection in SVMs [4]. Kasim et al. utilized ensemble learning and a set of regex rules as features to identify SQL injection [3]. Li et al. showed that an LSTM model could perform well in SQL injection detection [5].

With so many classifier models proposed in the literature, their reliability has become a growing concern.Recent research has focused on XSS attacks. Zhang et al. utilized a GAN-based approach for MCTS-T to explore adversarial attacks against XSS detectors [7]. Fang et al. then proposed a framework based on the DDQN networks for further exploration [8]. Wang et al. proposed more feasible strategies and utilized soft Q-learning to generate fuzz examples that significantly improved the escape rates of previous works [9].

Unfortunately, little research has focused on adversarial attacks on SQL injection. Liu et al. utilized a transformer model to translate an SQL injection payload into quality syntax to analyze the vulnerabilities in websites [11]. Since their framework was based on interactive application security testing, their system did not generate adversarial examples against SQLi detectors. The A-MoLE approach attempted to explore adversarial attacks against SQL with mutational fuzzing [12]. However, since the mutation search strategies were based on a simple mutation tree, they could not achieve optimized multi-fold strategies to escape detection. In addition, their mutation strategies were designed for common mutations against weak WAFs. For example, modern detectors are insensitive to URL-encoding and changes in string cases. Wang and Han proposed an evasive WAF based on reinforcement learning, and they formulated the WAF evasion problem into a reinforcement-learning problem to test their framework on XSS and SQL-injection payloads [14]. However, they did not include grammar-aware state representation or mutations for SQL injection. Amouei et al. presented an automated black-box testing strategy to discover injection vulnerabilities in WAFs [15]. They clustered string-based code-injection payloads, such as XSS and SQL injection, and utilized a reinforcement-learning technique combined with a novel adaptive search algorithm to search the clusters and discover bypassing attack patterns. Their proposal could discover vulnerabilities in WAFs efficiently; however, they could only select existing attack payloads and could not create new SQL injection payloads. Hemmati and Hadavi proposed a framework to bypass WAFs using deep reinforcement learning and designed specific action spaces for SQL injection attacks [13,16]. However, they only tested their framework on ModSecurity-CRS, Naxsi, and WAF-Brain, without testing the deep-learning-based detection model proposed by the academy.

Based on the aforementioned research and current challenges in the field, it is urgent to establish a feasible adversarial attack framework to evaluate the potential risk of SQL injection vulnerabilities.

3. Proposed Method

Therefore, due to the lack of black-box attacks and incomplete attack rules concerning adversarial SQL-injection attacks, we proposed a novel method, SSQLi. As Figure 1 shows, our SSQLi method consisted of three parts: a state-vector-representation algorithm, an attack-strategy matrix, and an adversarial example generation method based on the SAC algorithm. The original payload was an SQL injection code that contained the attack target and could be identified as a malicious example by detection models. The state-vector-representation algorithm performed the feature extraction of the original payload, which transformed the code into a vector containing syntactic and semantic information. We then designed a matrix containing three types of stackable variation strategies for attacking detectors, as actions for selection by the agent in reinforcement learning. Finally, without knowing the specific parameters of the detection model, the agent could adjust the payload variation direction continuously according to the returned results of the detection model and record each variation strategy for the experience pool until the final adversarial example could be generated. In the following sections, we provide details concerning the three parts of the framework.

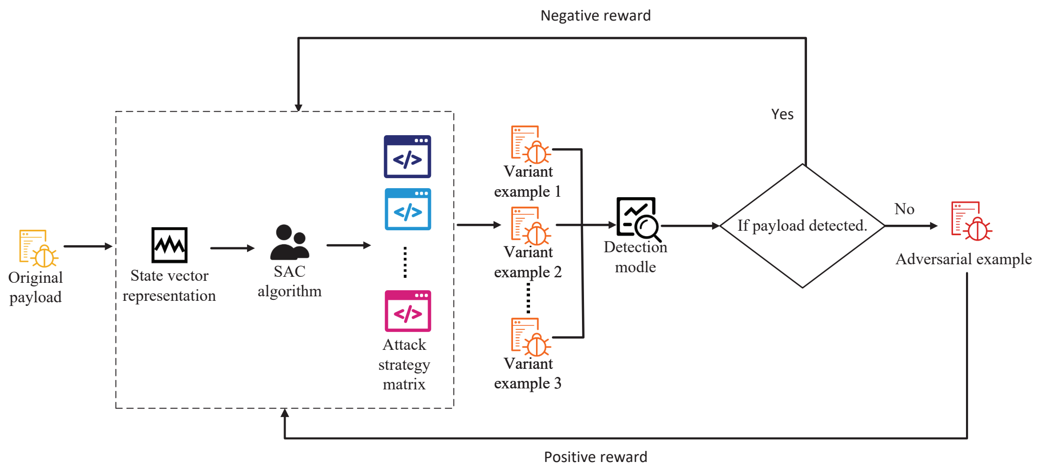


Figure 1. Framework of SSQLi.

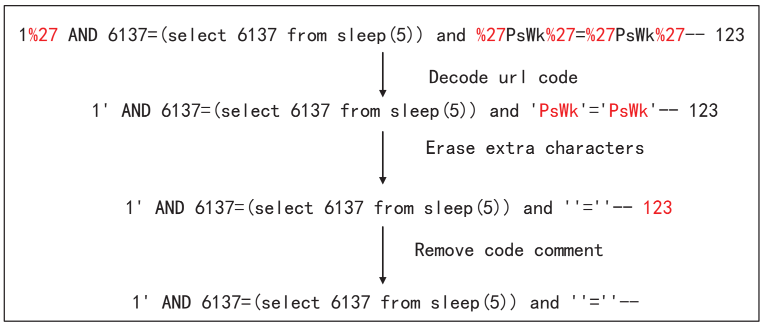
3.1. State-Vector-Representation Algorithm

The state-vector-representation algorithm should be regarded as the data preprocessing method in this paper that could map the payload in text form into one-hot vectors for subsequent processing. The transformed vectors included a static vector, representing the attack intention and the mode of the original payload, and a dynamic vector, representing the sample variation mode. Next, we introduce the generation methods for the static and dynamic vectors.

3.1.1. Static State Vector

Static vectors were used to represent the attack features of the original payloads while preserving the semantic information and attack intention. For the subsequent feature extraction and transformation, we first needed to process the data of the original payloads, and the processing steps are shown in Figure 2.

First, to facilitate the analysis of the structure and the text of the payload by the state-representation algorithm, we decoded the URL-encoding of the payload. Next, we removed any extra strings and comment information that could be re-integrated later for training. After preprocessing the original payload, we extracted the attack features from the payload and generated a static-state vector based on the attack features.



```
1%27 AND 6137=(select 6137 from sleep(5)) and %27PsWk%27=%27PsWk%27-- 123
1' AND 6137=(select 6137 from sleep(5)) and 'PsWk'='PsWk'-- 123
1' AND 6137=(select 6137 from sleep(5)) and ''=''-- 123
1' AND 6137=(select 6137 from sleep(5)) and ''=''--
```

Figure 2. An example of data preprocessing.

Note that a payload could satisfy more than one rule. For example, the payload in Table 1 was a union-based payload and also a time-based payload, according to our rules. The dimensionality of the static vector V_{static} we ultimately generated would be equal to the number of the types of attack targets, which was a total of 9 dimensions. Variable R_i was the i-th element of the static vector V_{static}, and its value was determined by whether the original payload had the i-th type of attack intent. This process was represented by Equation (1).

$$R_i\left(\text{payload}_{\text{ origin}}\right) = \begin{cases} 1, & \text{if payload satisfied rules i} \\ 0, & \text{else} \end{cases} \tag{1}$$

The static representation of that payload could then be represented by Equation (2). This vector would never change during the mutation process because it represented the original intention of the payload.

$$V_{\text{static}} = [R_i\,(\text{payload}\,)], i = 1, 2, \ldots\ldots, 9 \tag{2}$$

Table 1. Category of SQL injection.

Index	Type	Description	Example
1	Tautologies	If payload is a complete SQL statement.	?id=1 or 1=1
2	Union-based	If union query in payload.	?id=1 union select 1, updatexml(0x7e, user(), 0x7e), 3, 4, 5
3	Error-based	If common error functions in payload.	?id=updatexml(0x7e, user(), 0x7e)
4	Boolean-based	If conditions statement in payload.	?id=1 and 1=1
5	Stack-based	If payload is multi-query statement.	?id=1; select 1, version(), 3, 4, 5
6	Time-based	If time delay functions in SQL statement.	?id=if(1=1, sleep(5), 1)
7	Contain functions	If payload contains mysql function.	?id=1 and 'root' = user()
8	Contain keywords	If payload contains keywords.	?id=1 and 1=1
9	Contain subqueries	If payload contains subqueries.	?id=1 and 1=(select 1)

3.1.2. Dynamic State Vector

In contrast to static-state vectors, dynamic-state vectors represented the variation strategy used by the payload, and so it would change throughout the mutation process. Figure 3 shows the mutation process of a payload, where the red text denotes the text changes during mutation. In this subsection, we used strategies 11 and 13 as examples to demonstrate the variations in dynamic-state vectors.Other strategies and their justifications are elaborated further in Section 3.2.

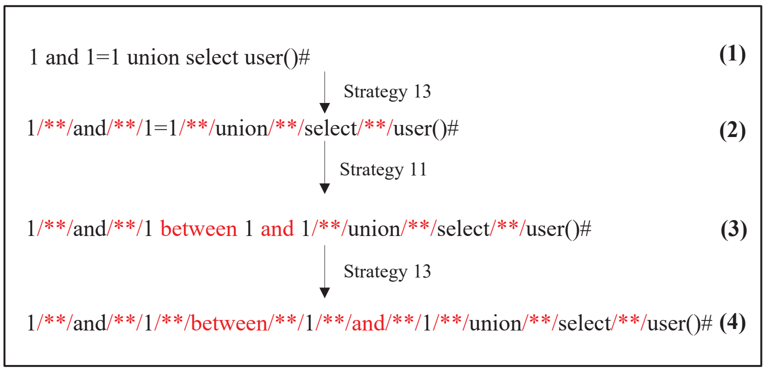
Strategy 13 was used to replace all spaces in the payload with comment characters, and strategy 11 was used to replace the equation with a "between end" statement. Since no variation strategy was used for the original sample, both $V_{\text{dynamic11}}$ and $V_{\text{dynamic13}}$ are zero. After implementing strategy 13, $V_{\text{dynamic 13}} = 1$. However, after implementing strategy 11, there were additional spaces added into the payload; therefore, $V_{\text{dynamic 13}} = 0$ and

$V_{\text{dynamic } 11} = 1$. We considered the characteristics of strategy i as dependence i, and then the dynamic-state vector could be defined by Equations (3) and (4).

$$D_i(\text{ payload }) = \begin{cases} 1, & \text{if payload statisfied dependence } i \\ 0, & \text{else} \end{cases} \tag{3}$$

$$V_{dynamic} = [D_i(\text{ payload })], i = 1, 2, \ldots\ldots, n \tag{4}$$

After obtaining the static-state and dynamic-state vectors, we spliced the two vectors together to generate a state vector with complete information. We could easily reveal the attacking intent of the original payload and the current variation strategy of the vector, which added in the training of the subsequent reinforcement learning.



```
1 and 1=1 union select user()#                                          (1)
1/**/and/**/1=1/**/union/**/select/**/user()#                           (2)
1/**/and/**/1 between 1 and 1/**/union/**/select/**/user()#             (3)
1/**/and/**/1/**/between/**/1/**/and/**/1/**/union/**/select/**/user()# (4)
```

Figure 3. Dynamic variation process of the payload during mutation.

3.2. *Strategy Matrix*

In this chapter, we present all the detector attack strategies that we used to generate the adversarial examples. In this study, based on a survey of the existing common methods to bypass SQL injection attacks [12], three major types of attack strategies were defined according to MySQL syntax and the limitations of existing detection models. Although attackers often use encoding methods for bypassing detection [22], after being encoded multiple times by reinforcement-learning algorithms, the attack example lost the original attack intent, which could not be easily restored.In order to ensure the availability of the adversarial examples in a real environment, the encoding class strategy was not used in this work. Instead, we developed our three types of attack strategies: 11 SQL syntactic transformation strategies, 2 syntactic analysis interference strategies, and the semantic analysis interference strategy.

3.2.1. Syntactic Transformation Strategy

Based on the common bypass techniques summarized by the open-source security intelligence team [23] and the preset functions provided by the official MySQL manual [24], this study developed the following 11 syntax transformation strategies:

1. Delay function replacement: The benchmark function in MySQL was used to test the performance of SQL statements and cause a delay when executing complex functions multiple times, so that the benchmark function could replace the sleep function, which is commonly used in SQL injection attacks.
2. Error function replacement: When strings and numbers were executed as parameters of the bin_to_uuid function, the error message of MySQL database printed the value of incoming parameters, so the bin_to_uuid function could replace common error-reporting functions, such as the “updateXML” in the SQL-injection-attack samples.
3. Comment confusion: To reduce the rate of false alarms, some WAFs ignore the content of the annotations; therefore, this method could hide the attack characteristics in these WAFs.

4. String-to-hex: Some detectors trigger alerts for SQL query statements accessing sensitive directories, in which case, converting strings to hexadecimals could evade the detection of sensitive directories.
5. Equation conversion: To reduce the rate of false alarms, most WAFs only trigger alerts if the product has only keywords and equations presented together, so this approach could hide the attack signature.
6. Table-name confusion: WAFs often use sensitive words, such as table names, to detect SQL injection attacks; therefore, this method could hide this attack feature.
7. Replace spaces with comments: Some detectors are sensitive to space characters, so replacing them could reduce the likelihood of being detected.
8. Redundant keyword omission: There are some keywords, including "as" in MySQL, that can be ignored when used but are easily detected as attacks, so we needed to eliminate these redundant keywords to avoid attacks.
9. Keyword replacement: We could use special characters as substitutes for sensitive keywords: For example, we could replace "and" with "&&".
10. Special table-name substitution: In addition to table name interference, some built-in MySQL tables have functional equivalences, such as some tables in the system library that could replace information in the information_schema library and could hide the attack characteristics of common tables.
11. ODBC-syntax interference: ODBC syntax is used to solve data-sharing problems between heterogeneous databases. This syntax places curly brackets around the execution of the shared content and then adds a string identifier that MySQL ignores when parsing and executing the shared content directly.

We show examples of the above 11 transformation strategies in Table 2, where the text marked in red denotes the replaced content.

Table 2. The example of syntactic transformation strategy.

Index	Rule Name	Example
1	Delay function replacement	sleep(5) → benchmark(500000, MD5(1))
2	Error function replacement	Updatexml(0x7e, user(), 0x7e) → bin_to_uuid(user())
3	Comment confusion	1 and 1=1 → 1ˆ'a/*a' and 1='a*/a'ˆ1
4	String-to-hex	Select 'abc' → select 0x616263
5	Equation conversion	1 and 1=1 → 1 and 1 between 1 and 1
6	Table-name confusion	Select user from mysql.user → select user from mysql/**/./**/user
7	Replace spaces with comments	Select 1,2,3 → select/**/1,2,3
8	Redundant keywords omission	Select table_name as t1 → select table_name t1
9	Keyword replacement	1 and 1=1 → 1 && 1=1
10	Special table-name substitution	select table_name from information_schema.tables → select table_name from sys.sechema_table_statistics_with_buffer
11	ODBC syntax interference	sleep(5) → {anyword sleep(5)}

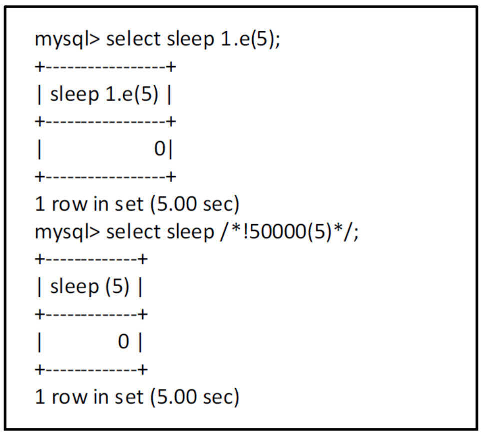
These interference strategies could attack some feature-based matching methods and affect syntax-analysis-based detectors, to some extent.

3.2.2. Syntactic Analysis Interference Strategy

Existing syntax-analysis-based detectors use the SQL syntax-parsing function provided by the libinjection library to preprocess the data. However, there are differences between the underlying parsing logic of the libinjection library and the way the actual database product parses the data. We exploited the detector vulnerability caused by these differences and then proposed the following two attack strategies:

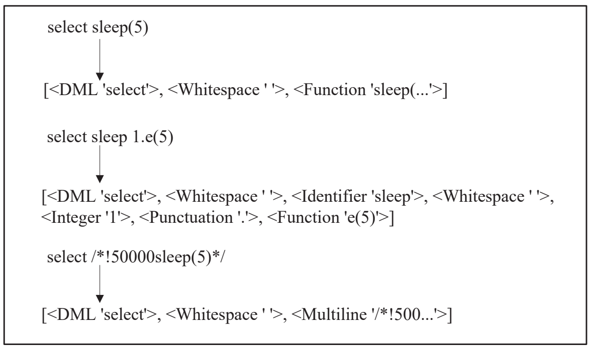
1. Scientific notation interference: When MySQL parsed a function, if the function's arguments had used scientific notation, MySQL ignored the scientific notation characters and placed the arguments directly into the function for execution. For example, when MySQL executed $sleep\ 1.e(5)$, it was equivalent to executing $sleep(5)$. Since this feature does not exist in the libinjection library, this interference would result in a difference in the parsing outcomes and introduce vulnerability.
2. Embedded execution of interference strategy: MySQL exists in a large number of distributions. To satisfy the requirements for backwards compatibility, MySQL can execute version-specific syntax based on the version content marked in the comments. For example, when executing $/*!50000\,\text{sleep}(5) * /$, MySQL executed $sleep(5)$ after adjusting the execution environment for MySQL5. However, the libinjection library would simply treat those characters as a comment and ignore them.

The results of MySQL executions A and B are shown in Figure 4, and we observed that although they were not identical, they had produced the same results. Furthermore, as shown in Figure 5, three lines of code with the same execution result had been parsed by the libinjection library as significantly different syntactic units. The semantic interference strategies could interfere with semantic-analysis-based detection methods and cause changes in the structure and content of the code without affecting the execution results.



```
mysql> select sleep 1.e(5);
+----------------+
| sleep 1.e(5) |
+----------------+
|             0|
+----------------+
1 row in set (5.00 sec)
mysql> select sleep /*!50000(5)*/;
+------------+
| sleep (5) |
+------------+
|         0 |
+------------+
1 row in set (5.00 sec)
```

Figure 4. MySQL execution results for two coding strategies.



```
select sleep(5)
[<DML 'select'>, <Whitespace ' '>, <Function 'sleep(...'>]

select sleep 1.e(5)
[<DML 'select'>, <Whitespace ' '>, <Identifier 'sleep'>, <Whitespace ' '>,
<Integer '1'>, <Punctuation '.'>, <Function 'e(5)'>]

select /*!50000sleep(5)*/
[<DML 'select'>, <Whitespace ' '>, <Multiline '/*!500...'>]
```

Figure 5. Execution results of the libinjection library for two coding strategies.

3.2.3. Semantic-Analysis-Interference Strategy

The semantic-analysis-interference strategy was used primarily for natural language-processing-based methods for SQL injection detection. Such detectors typically serialize attack examples, map the text to word vectors via word-embedding models, and then use downstream deep-learning algorithms for prediction. Relevant learning algorithms, such as CNNs, can extract key features from the text, or others, such as LSTM, can analyze the contextual structure to obtain the text state. As shown in Figure 6, the general detection model framework consisted of three parts: preprocessing, feature extraction, and a classifier model. Therefore, in order to bypass the natural language-processing-based detector, our concept involved adding normal features and obscure attack features to the example.

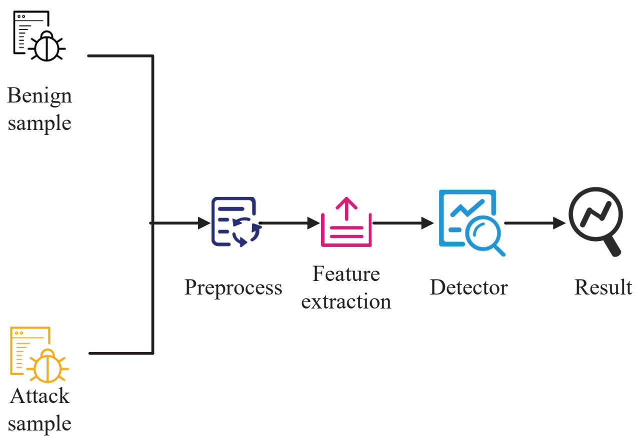


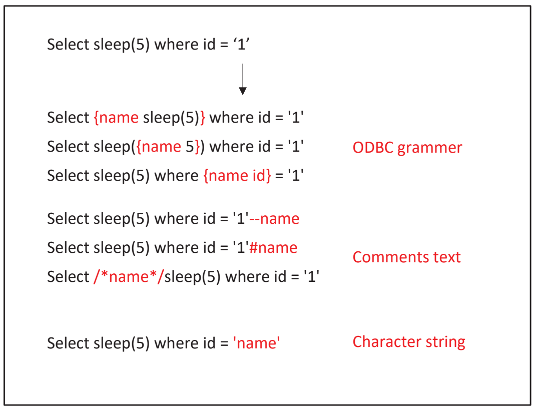
Figure 6. The framework of the detection method.

The previously described syntactic transformation strategy was able to obfuscate attack features, and the task of adding normal features was solved by the following experiments. In this study, by web-crawling Amazon Alexa's top 1000 most-visited websites, the parameter values of 210,000 APIs and 190,000 normal samples were acquired cumulatively through a three-layer crawling strategy. After the statistical analysis of the data, the 25 most-frequent words related to parameter values were selected as interference words for the semantic analysis in this paper. Table 3 shows the top five samples and their frequency rates.

Table 3. Top five high-frequency words in the normal sample.

Number	Parameter Values	Frequency of Occurrence
1	name	3046
2	duration	2967
3	subscribe	2648
4	startTime	2116
5	resource	2104

As shown in Figure 7, without changing the functionality of the original payload and breaking the SQL syntax, we chose three points of insertion: the ODBC syntax identification, the comment content, and the string content.



```
Select sleep(5) where id = '1'

Select {name sleep(5)} where id = '1'
Select sleep({name 5}) where id = '1'        ODBC grammer
Select sleep(5) where {name id} = '1'

Select sleep(5) where id = '1'--name
Select sleep(5) where id = '1'#name           Comments text
Select /*name*/sleep(5) where id = '1'

Select sleep(5) where id = 'name'             Character string
```

Figure 7. The points of insertion in normal payloads.

Together with the methods introduced in the first two sections of this paper, we designed a complete set of adversarial attack matrices for detection models based on feature-matching, syntactic analysis, and semantic analysis.

3.3. SAC Agent

When launching an attack on a detection model in a real-world scenario, the exact parameters of the model's settings are typically unknown. Therefore, in order to simulate a real environment, we designed a black-box attack model that could adaptively adjust the choice of attack strategy based only on the outcomes of the detector. We used reinforcement learning in our design, wherein an agent selected the attack strategy. We considered the attack strategy matrix as the action space A and the results returned by the detector as the environment feedback, with the goal of allowing the agent to select the best attack strategies. The process by which the agent identified the optimal strategy in order to maximize the reward provided by the environment was expressed as Equation (5), where $\mathbb{E}_{(s_t,\mathrm{a}_t)\sim\rho_\pi}[r(\mathrm{~s}_t,\mathrm{a}_t)]$ denotes the reward expectation based on strategy π. To avoid the effect of the localized optima on the final strategy selection [25], we used the stochastic-strategy-gradient algorithm.

$$\pi^* = \arg\max_\pi \sum_t \mathbb{E}_{(s_t,\mathrm{a}_t)\sim\rho_\pi}[r(\mathrm{~s}_t,\mathrm{a}_t)] \tag{5}$$

Ziebart et al. [26] introduced the concept of entropy to optimize strategic exploration and prevent a model from becoming enmeshed in localized optimal solutions, and this approach also encouraged the agent to explore more diverse strategies during the training phase. The attack strategy matrix proposed in this paper enabled the overlapping of multiple strategies in order to promote strategic diversification. We added entropy via $\alpha\mathcal{H}(\pi(\cdot \mid \mathrm{s}_t))$ to Equation (5) to create Equation (6), where b indicates the weight of the entropy value, where a larger value of α indicates more focus on a multi-modal strategic selection, and a smaller value indicates more focus on the maximum cumulative feedback expectation.

$$\pi^* = \arg\max_\pi \sum_t \mathbb{E}_{(s_t,\mathrm{a}_t)\sim\rho_\pi}[r(\mathrm{~s}_t,\mathrm{a}_t) + \alpha\mathcal{H}(\pi(\cdot \mid \mathrm{s}_t))] \tag{6}$$

The state-value function was represented by Equation (7), and the soft Q-function was represented by Equation (8), as follows:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \tag{7}$$

$$q_\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] \tag{8}$$

After the introduction of the entropy value, the reward function could be expressed as follows:

$$r_{\text{soft}}(s_t,a_t) = r(s_t,a_t) + \gamma\alpha\mathbb{E}_{s_{t+1}\sim\rho}H(\pi(\cdot \mid s_{t+1})) \tag{9}$$

In order to improve the efficiency of the sample utilization and ensure the rapid convergence of the model, this study used the SAC algorithm to train the actor network to implement strategy selection and the critic network to evaluate the value of actions [27]. Since the original SAC algorithm had been applied to continuous data, in the subsequent training, we referred to [28] for the implementation of the SAC algorithm. The training process of SSQLi is shown in Algorithm 1.

Algorithm 1 SSQLi based on SAC.

```
Input: Detector: SQL injection detector; Payloads: the set of original payloads; SAC: SAC
    agent; state_function: state-vector-representation algorithm attack_matrix; MaxStep;
    epoch; bach_size;
Output: SAC model; adversarial_examples;
 1: for each epoch do
 2:     for each payload in payloads do
 3:         while end_control == 0 do
 4:             state = state_function(payload)
 5:             action = SAC.pick_action(state)
 6:             attack_function = attack_matrix(action)
 7:             new_payload = attack_function(payload)
 8:             reward = Detector.test(new_payload)
 9:             n_state = state_function(new_paylaod)
10:             experiences.push([state,action,n_state,reward])
11:             if reward>0 then
12:                 adversarial_examples.push(new_payload)
13:                 end_control = 1
14:             end if
15:             if step_number>MaxStep then
16:                 end_control = 1
17:             end if
18:         end while
19:         history = experience.sample(batch_size)
20:         SAC.learn(history)
21:     end for
22: end for
23: return SAC model; adversarial_examples;
```

4. Experiment

4.1. Experimental Setup

Dataset: In this study, we collected 34,395 SQL-injection-attack examples from open-source attack intelligence websites, GitHub, and a specific dataset for SQL injection detection research [29]. Next, we crawled a large number of SQL query statements based on Amazon Alexa’s top-1000 websites, of which 45,991 were randomly selected as a normal example in the dataset. We used 34,395 items from each of the attack and normal examples for training; the remaining 11,596 samples were used as the testing set.

Devices and environment: We built our models in an ArchLinux 5.15.13 environment. In addition, the deep-learning-based detector was coded in Tensorflow-GPU 2.7.0 and the machine-learning-based detector was coded in the SKlearn library, whereas JIT techniques from the Numba library were used during the feature-vector-generation phase.The configuration of the server for the experiment was the following: an i912900KF CPU and an NVIDIA GeForce RTX 3090Ti 24G GPU.

Comparison experiment: In order to prove the advancement and the effectiveness of the method proposed in this paper, two models were selected for comparison: WAF-A-Mole [12] and the DDQN [30] algorithm-based adversarial attack model. Our adversarial attack method was tested on four types of detectors: WAFs, deep-learning-based detectors, machine-learning-based detectors, and ensemble-learning-based detectors.

Attack method parameters: We tested the performance of three attack methods in our experiments: SSQLi based on the SAC algorithm, as proposed in this paper; SSQLi based on the DDQN algorithm; and WAF-A-Mole as proposed in [12]. The following describes the parameter settings for these attack methods:

1. SAC: Our strategy selector π contained 64×64 hidden layer and used a softmax function as the activation function of the output layer. The critic model contained a

64×64 hidden layer and used a softmax function as the activation function of the output layer. Furthermore, the size of experience buffer was 100,000. The optimal maximum step-size and the number of training iterations would be selected in the parameter setting experiment.

2. DDQN: This model contained a 30×15 hidden layer, a maximum iteration step-size set at 100, epochs set at 10, and a batch-size set at 256.
3. WAF-A-Mole: WAF-A-Mole was the most recent adversarial attack model for SQL injection in the existing research to date. The model built a mutation strategy tree and would return to a previous node to choose another strategy when the current mutation method could not successfully bypass the detection model. To be consistent with the model setup in this paper, the maximum iterations of WAF-A-Mole was set at 100.

Detector parameters: In our experiments, we selected two mainstream WAFs and built eight detection models based on deep learning and machine learning. The two WAFs were as follows:

1. SafeDog [31]: SafeDog is a mature WAF system based on rule-matching. The version used in this experiment was updated on 10 February 2022, and the detection rules and levels of this system were set to the default mode.
2. ModSecurity [32]: ModSecurity is an open-source WAF system developed by the ModSecurity team. The system uses both a rule-matching method and a syntax-analysis method to prevent SQL injection. In this experiment, all parameters of the system were set to default.

The overall framework of the detection model is shown in Figure 6, and the relevant settings for the deep-learning- and machine-learning-based detection models are shown in Table 4. Furthermore, to build the deep-learning-based detector, we referenced [5,29,33]; to build the machine-learning-based detector, we referenced [4,34]; and to build the ensemble-learning-based detector, we referenced [3]. For feature extraction, we used the method proposed in [29,33]. In [29], the authors used the Word2Vec algorithm to map the payload text to word vectors and then stitch them together into sentence vectors. The authors of [33] used semantic analysis to serialize the SQL samples into tokens, used the three-grams algorithm to extract feature words for the sequences, and then calculated the positive-likelihood-ratio (PLR) and negative-likelihood-ratio (NLR) for each sample to finally transform the tokens into word vectors.

Table 4. The parameters of the detection model.

Detection Model	Feature Extraction	Batch Size	Epoch
MLP	Word2Vec	128	50
CNN	Word2Vec	128	50
LSTM	Word2Vec	128	50
SVM	3-grams and TF-IDF	-	-
AdaBoost	3-grams and TF-IDF	-	-
RandomForest	3-grams and TF-IDF	-	-
XGBoost	3-grams and TF-IDF	-	-
GradientBoost	3-grams and TF-IDF	-	-

Evaluation standard: In order to verify the effectiveness of SSQLi, we first had to examine the performance of the detection models in the absence of an adversarial attack. Four metrics were chosen to evaluate the performance of the original detection model: *Accuracy, Precision*, *Recall*, and the *F1-score*, as shown in Equations (10)–(13).

$$Accuracy = \frac{TP + FN}{TP + FP + TN + FN} \tag{10}$$

$$Precision = \frac{TP}{TP + FP} \tag{11}$$

$$Recall = \frac{TP}{TP + FN} \tag{12}$$

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} = \frac{2 \cdot TP}{2 \cdot TP + FP + FN} \tag{13}$$

After verifying the detection model performance, we used the detection and escape rates to evaluate the attack model's performance:

$$DR = \frac{TP}{TP + FN} \tag{14}$$

$$ER = \frac{FN}{TP + FN} \tag{15}$$

where TP is the number of SQL injection payloads correctly classified as attacks, TN is the number of benign SQL queries correctly classified as benign, FN is the number of of SQL injection payloads classified as benign, and FP is the number of benign SQL queries classified as attacks. Furthermore, this study aimed to verify that as the adversarial attack model performed additional mutation rounds, the detection model would have a lower detection rate for the samples. For this purpose, we set the average detection rate for n examples as follows: The m_i examples were detected in round i of mutations, and then the average detection rate (DR) could be expressed as Equation (16).

$$\overline{DR_i} = \frac{m_i}{n} \tag{16}$$

4.2. Performance Comparison

In this study, two experiments were conducted. First, the parameter settings for the adversarial attack model were established to explore the effects of different parameters and strategies on the effectiveness of adversarial attack. Second, the detection model for the adversarial attack model attack was evaluated to verify that the proposed method could outperform existing methods effectively and be integrated into different types of detectors.

4.2.1. Parameter Setting Experiment

To establish the parameter settings, the first experiment aimed to determine the optimal maximum step-size and the ideal number of iterations for the SSQLi approach. In this experiment, we used ModSecurity as the benchmark for validating detection efficiency. The experimental results for the maximum step selection are shown in Table 5. We observed that when the maximum step size reached 100 in the early stages of training, the bypass rate did not increase insignificantly. However, as the training time would increase linearly, we chose 100 as the maximum step size for the subsequent experiments.

Table 5. Experimental results of parameter selection for maximum step size.

	50	**75**	**100**	**125**	**150**
ER	50.69%	56.96%	62.94%	62.54%	63.87%
Training time	2540 s	4038 s	5714 s	7161 s	8766 s

Changing the epoch parameters demonstrated only a slight influence on the escape rate. Therefore, as shown in Table 6, we chose a parameter value at which the enhancement effect was the most obvious, 50. Therefore, we had selected all necessary parameters to analyze SSQLi, so the next step was to experiment with the attack detector.

Table 6. Experimental results of parameter selection of epochs.

	30	**40**	**50**	**60**	**70**
ER	59.61%	60.50%	62.94%	63.60%	62.68%
Training time	3627 s	4818 s	5714 s	6489 s	7808 s

4.2.2. Experiment of Attack Effect

The purpose of this experiment was to verify the effect of SSQLi on the detector attack. Since WAFs did not require training, in contrast to other other learning-based detection models, we first tested the effect of the two WAFs alone, as shown in Table 7. We observed that after being attacked by the adversarial sample, the detection rate of both WAFs decreased dramatically.

Table 7. The performances of WAF applications.

	Original Payload	**Adversarial Example**
	ER	**ER**
SafeDog	15.27%	94.96%
Modesecurity	0.10%	62.94%

Next, we tested the detectors on deep-learning, machine-learning, and integrated-learning models. First, however, we had to train each detection model to improve their performance, as shown in Table 8.

Table 8. The performance of original detection model.

Detection Model	**Accuracy**	**Precision**	**Recall**	**F1 Score**
LSTM	99.97%	99.99%	99.96%	99.97%
CNN	99.96%	99.33%	99.96%	99.64%
MLP	99.93%	99.91%	99.96%	99.93%
SVM	99.94%	100%	99.88%	99.94%
AdaBoost	99.95%	99.96%	99.94%	99.95%
RandomForest	99.97%	100%	99.93%	99.97%
XGBoost	99.96%	99.98%	99.93%	99.96%
GradientBoost	99.95%	99.97%	99.93%	99.96%

Then, we attacked the detector with the SAC-algorithm-based SSQLi, proposed in this paper, an SSQLi using another reinforcement-learning algorithm DDQN, and the WAF-A-Mole method. The experimental results are shown in Table 9. Moreover, we have documented the overall time consumption of the SSQLi attack detector in Table 10. When not being attacked, all detectors, except for SafeDog, had detection rates over 99%, and the vast majority of the attack samples had been detected. After attacking with the SAC-algorithm-based SSQLi, the detection rate of each detector decreased significantly, and the detection rates of the SVM and LSTM models were below 5%. During the training process, we recorded the changes in the average detection rates of the three attack methods, which are shown in Figure 8. Among all the detector models, the ensemble-learning-based detector had the best robustness with a consistent detection rate of approximately 60% even after an attack. As compared to the SAC-based SSQLi, the DDQN-based SSQLi only had a significant effect on the SafeDog, LSTM, and SVM models, but the degree of effect was relatively low. WAF-A-Mole had only a subtle effect on the detection model.

Table 9. Comparison of detection performance of three adversarial attack methods on the detector.

	Original Performance	SSQLi-SAC	SSQLi-DDQN	WAF-A-MoLE
	ER	ER	ER	ER
SafeDog	15.27%	94.96%	93.28%	35.38%
ModSecurity	0.10%	62.94%	0.83%	5.08%
LSTM	0.04%	97.39%	27.44%	1.85%
CNN	0.04%	56.53%	4.07%	1.96%
MLP	0.04%	46.40%	0.04%	1.96%
SVM	0.12%	96.37%	84.20%	2.64%
AdaBoost	0.06%	44.03%	0.37%	2.33%
RandomForest	0.07%	31.98%	0.17%	0.96%
XGBoost	0.07%	34.36%	2.59%	2.09%
GradientBoost	0.07%	37.20%	0.07%	2.66%

Table 10. Time consumption of SSQLi.

	SafeDog	ModSecurity	LSTM	CNN	MLP	SVM	AdaBoost	RandomForest	XGBoost	GradientBoost
Training Time	2149.74 s	3536.26 s	29,970.53 s	34,843.29 s	12,522.81 s	2834.67 s	47,894.34 s	43,474.43 s	43,463.45 s	39,957.42 s

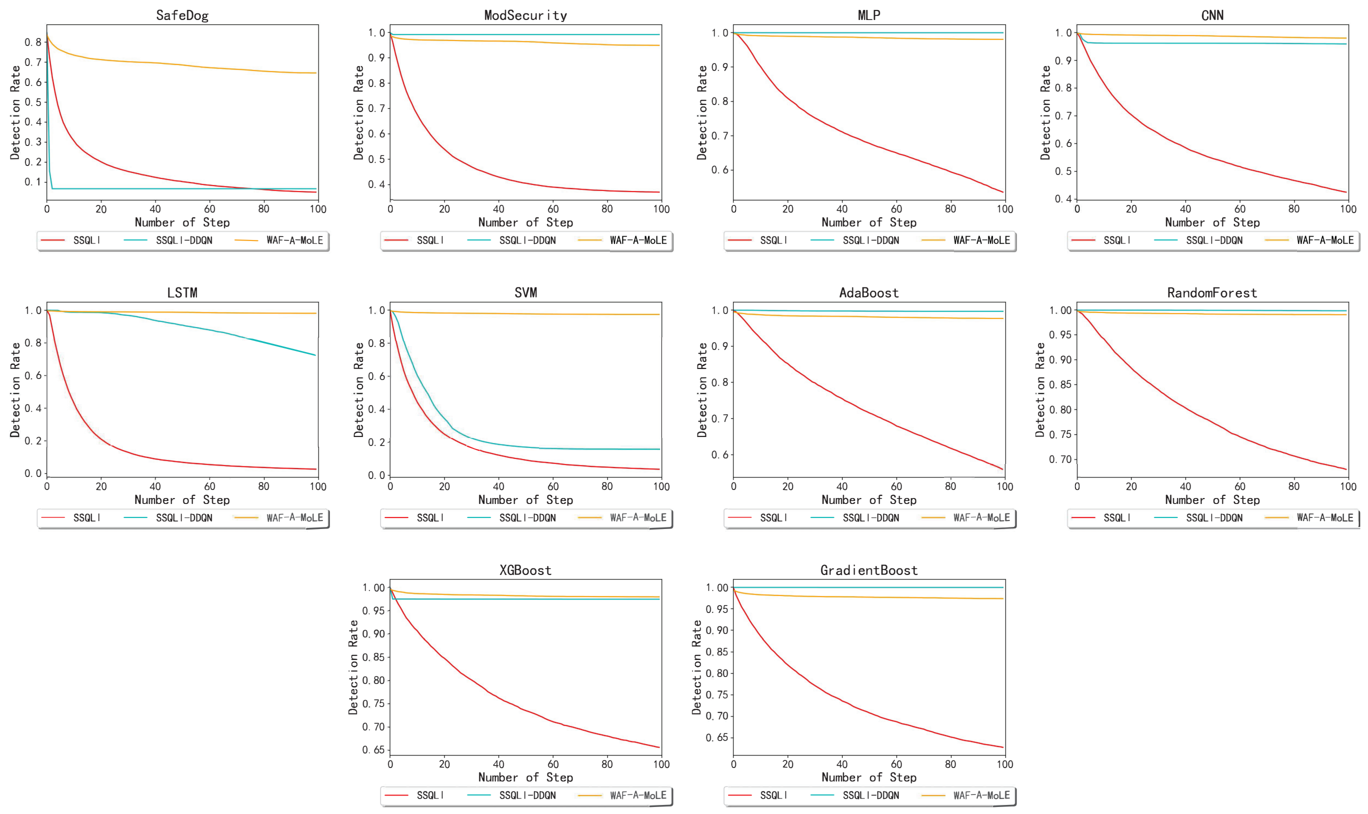
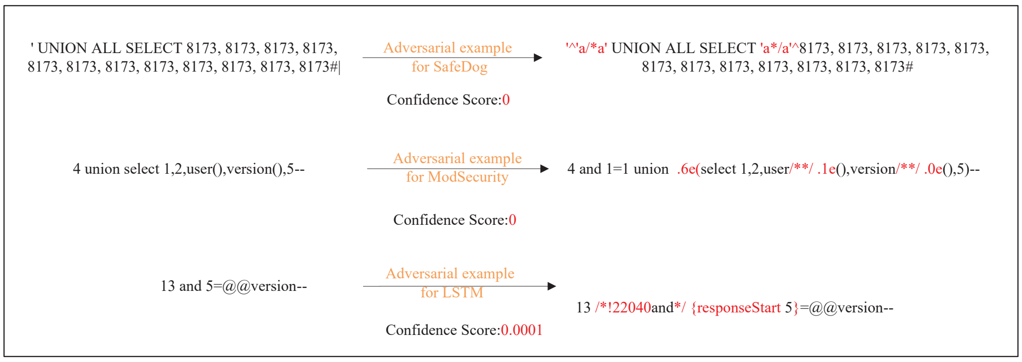


Figure 8. Mean detection rate variation curve of each detector against adversarial attacks.

Moreover, as a black-box-attack method, SSQLi automatically adopted different attack strategies for different detection methods without knowing the specific parameters of the detector. As shown in Figure 9, we were able to demonstrate the variation in the adversarial attack models by using SafeDog, ModSecurity, and LSTM as examples.


Adversarial example for SafeDog
Confidence Score:0
Adversarial example for ModSecurity
Confidence Score:0
Adversarial example for LSTM
Confidence Score:0.0001

```
' UNION ALL SELECT 8173, 8173, 8173, 8173,
8173, 8173, 8173, 8173, 8173, 8173, 8173, 8173#|
```

```
'^'a/*a' UNION ALL SELECT 'a*/a'^8173, 8173, 8173, 8173, 8173,
8173, 8173, 8173, 8173, 8173, 8173, 8173#
```

```
4 union select 1,2,user(),version(),5--
```

```
4 and 1=1 union  .6e(select 1,2,user/**/ .1e(),version/**/ .0e(),5)--
```

```
13 and 5=@@version--
```

```
13 /*!22040and*/ {responseStart 5}=@@version--
```

Figure 9. Examples of an adversarial attack models.

4.3. Discussion

After analyzing 10 detectors, we were able to demonstrate that SSQLi had a significant impact on various types of detectors. Of all the detectors, SafeDog, which was based on rule-matching, was the most severely affected because the SQL syntax is very flexible, making it difficult for the rule library to consider all attack features in order to reduce the rate of false alarms. The ensemble-learning-based detectors were more robust and better protected against black-box adversarial attacks because they used multiple weak-learners to jointly vote for classification.

As compared to other adversarial attack methods, the SAC-algorithm-based SSQLi was significantly more advanced. For example, as compared to the WAF-A-Mole, our model had a more flexible attack-strategy matrix that supported multiple rounds of strategy stacking. When using the same attack-strategy matrix, the SAC algorithm, which combined the advantages of critic and actor, was able to select the most appropriate attack strategy more rapidly and effectively than the DDQN algorithm, which used only a critic approach.

Despite our successful outcomes, there is more to be explored in future experiments. First, the experiments in this study focused on verifying whether our adversarial attack approach could effectively impact the detector from the attacker's perspective. However, viewing and counting the attack characteristics of the adversarial examples that were finally identified as normal samples would be more beneficial for a study focused only on detectors. Studying the attack strategies used to generate adversarial examples that are classified as normal by a detector can reveal weaknesses of the detector, which can help us understand why these weaknesses exist and further research them. Therefore, research on adversarial attacks is also beneficial for the development of attack detection methods.

Secondly, the attack strategy matrix in this study did not include coding strategies, yet coding strategies are the most common methods used by attackers on detectors in real-world scenarios. If the encoding strategy had been used, it may have had more of an impact on the detector, but this was outside of the scope of this research, as we would also have had to ensure that the attack intent of the payload would not change after using an encoding strategy. We recommend that a decoding layer be added detectors in order to mitigate this issue.

5. Conclusions

In this study, we examined the characteristics of SQL injection and proposed an adversarial attack method for SQL injection detectors based on reinforcement learning. We built eight machine- and deep-learning-based detection models according to the methods proposed in previous literature on SQL injection detection, and then we verified the effectiveness of our adversarial attack method on our detection models, as well as two mainstream WAFs. We studied the prevention of SQL injection from the attacker's perspective and showed that an adversarial attack method, such as SSQLi, could induce significant decreases in the detection rates of mainstream detectors, and we provided suggestions for improving the detection of SQL injection and the robustness of detection models. A limitation in this study was that our experiments and results are relevant only for relational databases, such as MySQL. Further research into the application of these models on other types of databases could be a fruitful direction for future work. Furthermore, the training time of our proposed method is relatively long. In future research, efforts can be made to explore approaches for reducing the training time.

Author Contributions: Conceptualization, Y.G.; Methodology, J.H.; Software, B.M.; Data curation, H.Z.; Writing—original draft, Y.G.; Writing—review & editing, Y.G., J.H. and T.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported, in part, by the National Key Research and Development Program of China (No. 2020YF B1805400); in part, by the National Natural Science Foundation of China (No. U19A2068, No. 62032002, and No. 6210 1358); in part, by the China Postdoctoral Science Foundation (No. 2020M683345); and in part, by the Fundamental Research Funds for the Central Universities (Grant No. SCU2021D052).

Data Availability Statement: Data presented in this study are available by request from the corresponding author. Due to ownership concerns, the data cannot be made publicly available at this time.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. OWASP. OWASP Top Ten. Available online: https://owasp.org/ (accessed on 4 June 2022).
2. Clay Keller. Vulnerability Dstribution of CVE Security Vulnerabilities by Types. Available online: https://www.cvedetails.com/vulnerabilities-by-types.php (accessed on 20 March 2021).
3. Kasim, Ö. An ensemble classification-based approach to detect attack level of SQL injections . *J. Inf. Secur. Appl.* **2021**, *59*, 102852. [CrossRef]
4. Kar, D.; Panigrahi, S.; Sundararajan, S. SQLiGoT: Detecting SQL injection attacks using graph of tokens and SVM . *Comput. Secur.* **2016**, *60*, 206–225. [CrossRef]
5. Li, Q.; Wang, F.; Wang, J.; Li, W. LSTM-based SQL injection detection method for intelligent transportation system . *IEEE Trans. Veh. Technol.* **2019**, *68*, 4182–4191. [CrossRef]
6. Qiu, S.; Liu, Q.; Zhou, S.; Wu, C. Review of artificial intelligence adversarial attack and defense technologies. *Appl. Sci.* **2019**, *9*, 909. [CrossRef]
7. Zhang, X.; Zhou, Y.; Pei, S.; Zhuge, J.; Chen, J. Adversarial examples detection for XSS attacks based on generative adversarial networks. *IEEE Access* **2020**, *8*, 10989–10996. [CrossRef]
8. Fang, Y.; Huang, C.; Xu, Y.; Li, Y. RLXSS: Optimizing XSS detection model to defend against adversarial attacks based on reinforcement learning. *Future Int.* **2019**, *11*, 177. [CrossRef]
9. Wang, Q.; Yang, H.; Wu, G.; Choo, K.K.R.; Zhang, Z.; Miao, G.; Ren, Y. Black-box adversarial attacks on XSS attack detection model. *Comput. Secur.* **2022**, *113*, 102554. [CrossRef]
10. Chen, L.; Tang, C.; He, J.; Zhao, H.; Lan, X.; Li, T. XSS Adversarial Example Attacks Based on Deep Reinforcement Learning. *Comput. Secur.* **2022**, *120*, 102831. [CrossRef]
11. Liu, M.; Li, K.; Chen, T. DeepSQLi: Deep semantic learning for testing SQL injection. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, 18–22 July 2020; pp. 286–297.
12. Demetrio, L.; Valenza, A.; Costa, G.; Lagorio, G. Waf-a-mole: Evading web application firewalls through adversarial machine learning. In Proceedings of the 35th Annual ACM Symposium on Applied Computing, Brno, Czech Republic, 30 March–3 April 2020; pp. 1745–1752.
13. Hemmati, M.; Hadavi, M.A. Bypassing Web Application Firewalls Using Deep Reinforcement Learning. *ISeCure* **2022**, *14*, 131–145.
14. Wang, X.; Han, H. Evading Web Application Firewalls with Reinforcement Learning. Available online: https://openreview.net/forum?id=m5AntlhJ7Z5 (accessed on 4 June 2022).
15. Amouei, M.; Rezvani, M.; Fateh, M. RAT: Reinforcement-Learning-Driven and Adaptive Testing for Vulnerability Discovery in Web Application Firewalls. *IEEE Trans. Dependable Secur. Comput.* **2021**, *19*, 3371–3386. [CrossRef]
16. Hemmati, M.; Hadavi, M.A. Using Deep Reinforcement Learning to Evade Web Application Firewalls. In Proceedings of the 2021 18th International ISC Conference on Information Security and Cryptology (ISCISC), Isfahan, Iran, 1–2 September 2021; pp. 35–41.
17. Guo, C.; Gardner, J.; You, Y.; Wilson, A.G.; Weinberger, K. Simple black-box adversarial attacks. In Proceedings of the International Conference on Machine Learning, Long Beach, CA, USA, 18–23 January 2019 ; pp. 2484–2493.

18. Goswami, G.; Ratha, N.; Agarwal, A.; Singh, R.; Vatsa, M. Unravelling robustness of deep learning based face recognition against adversarial attacks. In Proceedings of the AAAI Conference on Artificial Intelligence, New Orleans, LA, USA, 2–7 February 2018; Volume 32.
19. Morris, J.X.; Lifland, E.; Yoo, J.Y.; Qi, Y. Textattack: A framework for adversarial attacks in natural language processing. *arXiv* **2020**, arXiv:2005.05909.
20. Zhang, X.; Chen, J.; Zhang, R.; Wang, C.; Liu, L. Attacking Recommender Systems With Plausible Profile. *IEEE Trans. Inf. Forensics Secur.* **2021**, *16*, 4788–4800. [CrossRef]
21. Zhou, Y.; Wang, P. An ensemble learning approach for XSS attack detection with domain knowledge and threat intelligence. *Comput. Secur.* **2019**, *82*, 261–269. [CrossRef]
22. Kumar, P.; Pateriya, R. A survey on SQL injection attacks, detection and prevention techniques. In Proceedings of the 2012 Third International Conference on Computing, Communication and Networking Technologies (ICCCNT'12), Coimbatore, India, 26–28 July 2012; pp. 1–5. [CrossRef]
23. OWASP. The SQL injection Knowledge Base. Available online: https://www.websec.ca/kb/sql_injection (accessed on 22 March 2022).
24. ORACLE. MySQL 8.0 Reference Manual. Available online: https://dev.mysql.com/doc/refman/8.0/en/ (accessed on 23 March 2022).
25. Haarnoja, T. *Acquiring Diverse Robot Skills via Maximum Entropy Deep Reinforcement Learning*; University of California: Berkeley, CA, USA, 2018.
26. Ziebart, B.D.; Maas, A.L.; Bagnell, J.A.; Dey, A.K. Maximum entropy inverse reinforcement learning. In Proceedings of the Aaai, Chicago, IL, USA, 13–17 July 2008; pp.1433–1438.
27. Haarnoja, T.; Zhou, A.; Abbeel, P.; Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Proceedings of the International Conference on Machine Learning, Stockholm, Sweden, 10–15 July 2018; pp. 1861–1870.
28. Christodoulou, P. Soft actor-critic for discrete action settings. *arXiv* **2019**, arXiv:1910.07207.
29. Luo, A.; Huang, W.; Fan, W. A CNN-based Approach to the Detection of SQL Injection Attacks . In Proceedings of the 2019 IEEE/ACIS 18th International Conference on Computer and Information Science (ICIS), Beijing, China, 17–19 June 2019; pp. 320–324.
30. Hasselt, H.V.; Guez, A.; Silver, D. Deep Reinforcement Learning with Double Q-learning. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, Phoenix, AZ, USA, 12–17 February 2016.
31. Safedog. Safedog WEB Firewall Application. Available online: http://free.safedog.cn/ (accessed on 2 December 2021).
32. Martinhsv. SpiderLabs ModSecurity repository in github. Available online: https://github.com/SpiderLabs/ModSecurity (accessed on 29 January 2021).
33. Fang, Y.; Peng, J.; Liu, L.; Huang, C. WOVSQLI: Detection of SQL Injection Behaviors Using Word Vector and LSTM. In Proceedings of the the 2nd International Conference, Guiyang, China, 16–19 March 2018.
34. Mcwhirter, P.R.; Kifayat, K.; Qi, S.; Askwith, B. SQL Injection Attack classification through the feature extraction of SQL query strings using a Gap-Weighted String Subsequence Kernel. *J. Inf. Secur. Appl.* **2018**, *40*, 199–216. [CrossRef]