# SQIRL: Grey-Box Detection of SQL Injection Vulnerabilities Using Reinforcement Learning

Salim Al Wahaibi, Myles Foley, and Sergio Maffeis, *Imperial College London*

https://www.usenix.org/conference/usenixsecurity23/presentation/al-wahaibi

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

# SQIRL: Grey-Box Detection of SQL Injection Vulnerabilities Using Reinforcement Learning

Salim Al Wahaibi
*Department of Computing*
*Imperial College London*
s.al-wahaibi21@imperial.ac.uk

Myles Foley
*Department of Computing*
*Imperial College London*
m.foley20@imperial.ac.uk

Sergio Maffeis
*Department of Computing*
*Imperial College London*
sergio.maffeis@imperial.ac.uk

## Abstract

Web security scanners are used to discover SQL injection vulnerabilities in deployed web applications. Scanners tend to use static rules to cover the most common injection cases, missing diversity in their payloads, leading to a high volume of requests and false negatives. Moreover, scanners often rely on the presence of error messages or other significant feedback on the target web pages, as a result of additional insecure programming practices by web developers.

In this paper we develop SQIRL, a novel approach to detecting SQL injection vulnerabilities based on deep reinforcement learning, using multiple worker agents and grey-box feedback. Each worker intelligently fuzzes the input fields discovered by an automated crawling component. This approach generates a more varied set of payloads than existing scanners, leading to the discovery of more vulnerabilities. Moreover, SQIRL attempts fewer payloads, because they are generated in a targeted fashion.

SQIRL finds all vulnerabilities in our microbenchmark for SQL injection, with substantially fewer requests than most of the state-of-the-art scanners compared with. It also significantly outperforms other scanners on a set of 14 production grade web applications, discovering 33 vulnerabilities, with zero false positives. We have responsibly disclosed 22 novel vulnerabilities found by SQIRL, grouped in 6 CVEs.

## 1 Introduction

The prevalent use of SQL databases as part of web applications constitutes a significant attack surface for malicious actors. Despite a substantial body of research on SQL injection (SQLi) [4–6, 8, 12, 13, 17–19, 21, 24, 28, 34, 55, 58], recent surveys such as [52] and the OWASP Top Ten [35] still find SQLi to be one of the most common attacks against web applications. Sanitisation methods, such as those recommended by OWASP [36], and best-practice defenses, such as parameterised queries and stored procedures, are often ignored or incorrectly applied by developers [7, 45]. Such insecure practices and misconfigurations can lead to SQLi vulnerabilities which are hard to find, as they are triggered only by complex payloads [57].

Vulnerability scanners are commonly used to find vulnerabilities before they are placed in production. However, they sometimes rely upon incorrect handling of errors and feedback from SQL queries on a web page, leading to reduced effectiveness when these are not present. Scanners such as Wapiti [2], use a small set of payloads from a payload list, with minimal mutation or obfuscation techniques. Others, such as Sqlmap [1], use simple rule-based approaches to cover the most common SQLi payloads. By using payloads which lack diversity, and due to their inability to tailor payloads to specific web applications, scanners miss legitimate vulnerabilities and use a high number of requests in doing so [29, 32, 57].

To overcome such limitations, we develop a novel approach to the automatic discovery of SQLi vulnerabilities which leverages Reinforcement Learning (RL) to generate payloads targeted to each injection point, specific context, and defenses.

RL has seen little adoption in web security [10, 16, 23, 25] due to the difficulty of formulating security challenges as RL games. We create an RL environment which separates the complexities of crawling and interacting with a web application and potentially multiple databases from the task of discovering new payloads. We do not require access to the application source code, but leverage access to the database logs. This grey-box approach makes it possible to avoid false positives and to effectively identify the application inputs that occur in SQL queries. It encourages best programming practices, and is a useful tool for defenders (which can access logs) and it does not enable uninvited attacks.

Link [23] was the first deep RL model used to find injections, specifically XSS, in web applications in a black box fashion. In comparison to practical tools such as BurpSuite, Link managed to reduce the number of requests per vulnerability found, and find a comparable number of vulnerabilities in production web applications.

Off-the-shelf deep RL models, such as those used by Link, suffer from a number of limitations which we address by designing a more sophisticated RL architecture. First, to overcome difficulty in using the variable length and complexity of SQL statements and SQLi payloads, we introduce autoencoders to provide a meaningful latent representation. Second, we use a diverse action space that requires no bootstrapping by existing payloads. Third, we leverage multiple worker agents to improve the stability of the learning process, and runtime performance. We also explore the possibility of training the agents in a federated learning fashion, providing each with different payloads.

We demonstrate our approach is effective and practical by comparing it against a number of established web application scanners, both on a micro-benchmark, and on production grade web applications. We then compare the advantages of our design choices in a model ablation study. Our approach finds more vulnerabilities than other scanners, and does so using fewer requests. In particular, it discovers 22 new SQLi vulnerabilities, which we have responsibly disclosed.

*Summary of contributions:*

- We implement an RL environment for agents to fuzz web applications for SQLi that identifies inputs via a state-of-the-art crawler we extend. The environment implements actions for different database management systems (DBMS), so agents are target-agnostic.
- We design and implement a novel deep RL approach to fuzzing called SQIRL. A single RL agent completes three tasks of SQLi: syntax fixing, context escape, and sanitisation bypass. Using worker agents in a federated style this knowledge is then shared among the agents. We explain the significance of our model design choices with an ablation study, and make our implementation publicly available at https://github.com/ICL-ml4csec/SQIRL.
- We compare SQIRL against 5 state-of-the-art scanners on a novel benchmark and 14 web applications. Our tool identifies the most SQLi vulnerabilities (63) of any scanner, and has currently led to 22 new vulnerabilities summarised in 6 CVEs.
- We analyse the payloads of the web scanners we use in our evaluation to determine their payload diversity. SQIRL achieves the greatest payload diversity and uses the most features in doing so.

## 2 Background

### 2.1 Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning that has an agent learn an optimal policy ($\pi$) of actions ($a$) in an environment. The agent does so by maximising the expected reward ($r$) over a sequence of timesteps referred to as an episode. After a timestep the agent receives a state ($s$) which it uses to choose the next action [49]. The reward, $r_t$, received from taking action $a_t$, is used to determine the performance of the action, leading to the expected episodic reward after the terminal step $T$ giving: $R = \sum_{t=0}^{T} \gamma r_t$, where $\gamma$ is the discount factor.

To encourage the agent to explore its environment, an $\varepsilon$-greedy decay exploration takes random actions with probability $\varepsilon$, otherwise taking the policy action. $\varepsilon$ is initially set to a high value and gradually decays to a minimum, placing emphasis on the policy.

One way to compute optimal actions is Q-learning, as described by the Bellman equation: $q_\pi(s,a) = \mathbb{E}_\pi[r_{t+1} + \gamma \max_{a_{t+1}} q_\pi(s_{t+1}, a_{t+1})]$ which, given the previous action and state, computes Q-values corresponding to each action [31]. However, Q-learning suffers from the "curse of dimensionality" as state-action spaces become large. Instead, deep learning can be used to approximate the Q-values in a *Deep Q-Network*: a Q-Network ($Q$) computes Q-values using the state in a semi-supervised fashion; a Target Q-Network ($\hat{Q}$) is used to compute the loss of the agent; $\hat{Q}$ is periodically updated to $Q$, allowing the networks to converge to an optimal policy [53].

### 2.2 Federated Learning

Federated Learning (FL) is a decentralised form of machine learning. A number of *client* models are trained independently for a period, sending the parameters they have learned to a central *server* model, which aggregates them (typically by averaging) and then broadcasts the updated weights back to the clients. FL has been recently proposed for use in RL models [41]. Federated RL comes in two different flavors: *horizontal*, where clients interact with different environments, and *vertical*, where clients interact with multiple instances of the same environment. Vertical RL is often referred to as using 'rollout workers' or simply 'workers' in RL settings [30].

## 3 Motivation and Challenges

An injection vulnerability aims to trigger unintended functionality on a web application, typically violating a (possibly implicit) security policy. The particular functionality can vary from remote code execution to data exfiltration. In this paper, we focus on SQLi, which injects a payload, typically into the client side of a web application, aiming to cause the execution of an unintended SQL query on the web application database on the server. The simplest form of SQLi produces some feedback on the web page that triggered it, where a query result is returned. A harder to find, but more powerful variant, called *blind* SQLi, instead works even if there is no query feedback on the web page, for example when a web application server only issues generic responses to the client, such as '*Thanks for submitting your survey*'. In our research, we address both forms of injection.

There are two mains steps for finding SQLi vulnerabilities.

Table 1: **A**: SQLi payload (highlighted in pink) escaping its SQL context to cause the database to pause for 1 second.
**B-D**: SQL queries showing 13 semantically different positions where user input can occur.
**E**: current WordPress example of dangerous parameterised query pattern.

```
A: SELECT * FROM tab WHERE val = '' AND SLEEP(1) -- '
B: SELECT input_1 FROM input_2 WHERE input_3 = 'input_4' GROUP BY 'input_5'
C: UPDATE input_6 SET ("input_7", "input_8")
D: INSERT INTO input_9 (input_10, input_11) SET ('input_12', 'input_13')
E: $query = $wpdb->prepare( "SELECT $id_column FROM $table WHERE meta_key = %s", $meta_key );
```

1. Identify the input locations that may be vulnerable to SQLi. These include URLs, input tags, and dynamic link elements associated with a web page.

2. Once the input locations are found, relevant payloads can be crafted. Payloads must meet 4 criteria to trigger SQLi: 1) produce syntactically valid SQL, to avoid errors in execution, 2) escape the intended SQL context, 3) bypass any blacklist-based filtering applied to the payload, 4) change the intended behaviour of the SQL statement. For example, assuming no filtering, the payload shown in red in Table 1 escapes the ' and executes a `SLEEP` statement.

Table 1 shows three SQL statements (B-D) that could have 13 different input locations, and 10 different contexts (as inputs in parentheses share the same context). For each context, a different escape pattern is required to trigger a vulnerability.

While methods to prevent SQLi are well-known, they are often ignored or misused [7, 45], and SQLi was still the most common web injection attack in 2020 (68.21%), accounting for over 4 billion attacks [3]. The safest way to prevent SQLi is to use parameterised queries to render user inputs benign. This practice has developers define the types of SQL query parameters before binding them to a query template, enforcing type preservation. Although most modern web applications use parameterised queries, these are sometimes used following insecure patterns. For example, current WordPress code [56] includes line E from Table 1, which protects its last parameter by coercing `$meta_key` to a string, but could be vulnerable to SQLi if variable `$id_column` was previously injected with the payload `* FROM accounts --`.

Some modern web applications, and many legacy ones (which constitute the bulk of existing websites), attempt to prevent SQLi by filtering user input. Developers can take many different approaches including basic string manipulation, regex matching, and rule-based approaches. However, these implementations are dependent on the skill and knowledge of the developer. Thus some instances of these inputs may be vulnerable to SQLi, but require specific, tailored payloads to be found. Existing vulnerability scanners consider only the most common cases of SQLi, failing to capture the complex cases that occur due to different developer knowledge or implementation. Moreover, existing approaches lead to a high volume of requests and a corresponding substantial execution time, which is undesirable.

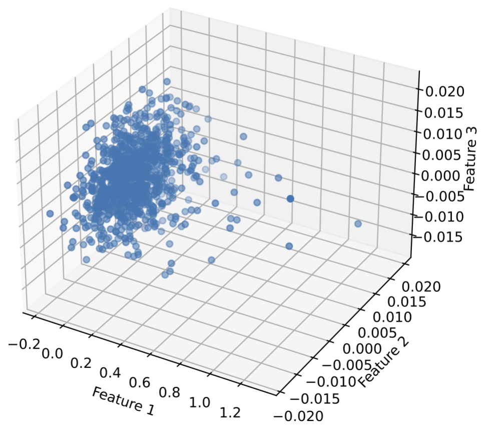

Figure 1: Principal Component Analysis of SQLi payloads.

### 3.1 SQLi Payload Distribution

Recent work [57] has shown the increasing complexity of payloads that are required to detect SQLi, recognising several characteristics of existing SQLi payloads. One is the sparsity of existing payloads, specifically that in response to input validation and sanitisation, payloads that are able to bypass them become increasingly complex. Another characteristic is the uneven distribution of existing payloads that leads to clusters, where vulnerabilities occur from variations of a payload.

To illustrate this we provide a Principal Component Analysis (PCA) dimensionality reduction of 1028 unique payloads gathered from open-source injection payload lists [38, 50, 51] and scanners (e.g. Wapiti). This is achieved by tokenising the payloads via Byte Pair Encoding [44], extracting 128 feature embeddings of each payload using Doc2Vec [22], and performing PCA to reduce the embeddings from 128 dimensions down to 3.

The three-dimensional PCA in Figure 1 demonstrates the uneven distribution of payloads present in the latent payload space, seen by the dense cluster and points that spread around it. Importantly, it further confirms that commonly used SQLi payloads, including those in scanners, share similar features in a dense distribution.

It is unlikely that all legitimate payloads are captured in this distribution, as there is a large space left unexplored. We hypothesise that in this space there are alternative payloads that lead to SQLi. Note however, that not all points in the latent space of Figure 1 correspond to legitimate SQLi payloads, as they must also conform to the syntax requirements of SQL.

## 3.2 Existing Scanner Methods

While research has been conducted to detect SQLi using traditional fuzzing paradigms, popular static analysis methods such as constraint solvers are unable to solve the different type of constraints precisely, leaving behind a high number of false negatives and false positives [29, 32].

A common defense technique is to use web vulnerability scanners to identify SQLi early, before attackers do, by issuing a curated set of requests to the web application, and observing its responses. However, current scanners still miss legitimate SQLi vulnerabilities. Scanners such as Wapiti, and Arachni [27] use a small number of simple payloads, and do not mutate them, offering no dynamic payload generation methods to bypass sanitisation. While Sqlmap [1] is seen as a "gold standard" for fuzzing for SQLi, it doesn't focus on generating diverse payloads, instead focusing on breadth of functionality, offering a number of different configurations such as type cast, false positive, and length constraint detection. Whilst payloads in Sqlmap are generated by a rule-based system or automaton, Sqlmap is not able to tailor generation to specific targets, limiting payload diversity. Furthermore, it only considers a small set of anti-sanitisation methods (e.g. for `addslashes` of PHP).

## 3.3 Challenges of Using RL for SQLi

Given the sparsity and uneven distribution of SQLi payloads, it is difficult to develop automata or heuristics to adequately cover their diversity. RL is able to search over large state spaces to find an optimal solution by balancing exploration and exploitation, and it is a natural candidate for this task. An RL-based solution could tailor its strategy to individual test cases, thus efficiently generating diverse payloads. However, there are several challenges that must be overcome.

### 3.3.1 Interaction with the Web Application

In game playing settings, it is clear how an agent plays a game. For SQLi, it is not so clear how an agent should interact with a web application. In prior work [13] this challenge was not fully explored, only using RL in contained environments and not on real web applications. This is due to the difficulty in designing a general *environment* where an agent is able to interact with different web applications and DBMS.

### 3.3.2 Maximising Information Usage

An RL agent needs to have an understanding of the structure and syntax of the current payload so that it knows which mutations, or actions, to take. How the agent observes this payload will dictate how the agent performs. The question is how to provide this information in a meaningful and concise manner. In prior work [13, 23], this has been captured by a set of manually defined features about the payloads themselves. While domain knowledge is often useful for RL, each set of bespoke features will impact the agent differently, and may not contain enough meaningful information. For example, traditional features would struggle to relate the presence of a SQL keyword with the position where it occurs. Hence, we face a challenge of how to maximise the usage of information about the payload we are generating and the whole SQL query we are trying to exploit to improve the learning process.

### 3.3.3 Generating New and Diverse Payloads

RL in general has difficulty in adapting to new actions, so most approaches adopt fixed action spaces. This is even more so in deep RL where input states, and output Q-values are vectors of fixed length.

The agents in [13, 23] use a fixed action space. With a fixed action space, useful payload-generation tasks such as matching the number of open parentheses in an expression can only be done by relying on nondeterminism, which has a high performance cost. This is analogous to randomly sampling the correct string from the language generated by an automaton or grammar.

Using a dynamic action space, an agent can modify a payload at specific meaningful points. For example, consider the query snippet `WHERE (x=1 AND y='$input') LIMIT 1`. A typical first attempt at exploiting the query would be the payload `' OR 1=1 --`, which fails because of unmatched parenthesis. An agent with a dynamic action space could learn to insert `)` directly in the right place to prevent the error and obtain results from the query, using payload `') OR 1=1 --`.

## 4 SQIRL

We developed a novel approach to fuzz for SQL injection vulnerabilities leveraging RL worker agents to search over the potentially infinite space of injection payloads, strategically tailoring payloads to different injection points. We show the full architecture of SQIRL in Figure 2. Below, we describe the individual components and the architecture design.

## 4.1 An RL Environment for SQLi

The environment abstracts the web application to the RL agents as a component that receives an action and returns an observation. It encompasses all mechanisms that allow for payloads to be submitted to the web application. It also includes crawling the web application and monitoring the SQL database to provide the feedback mechanism that allows the agent to learn how to tailor SQLi payloads.

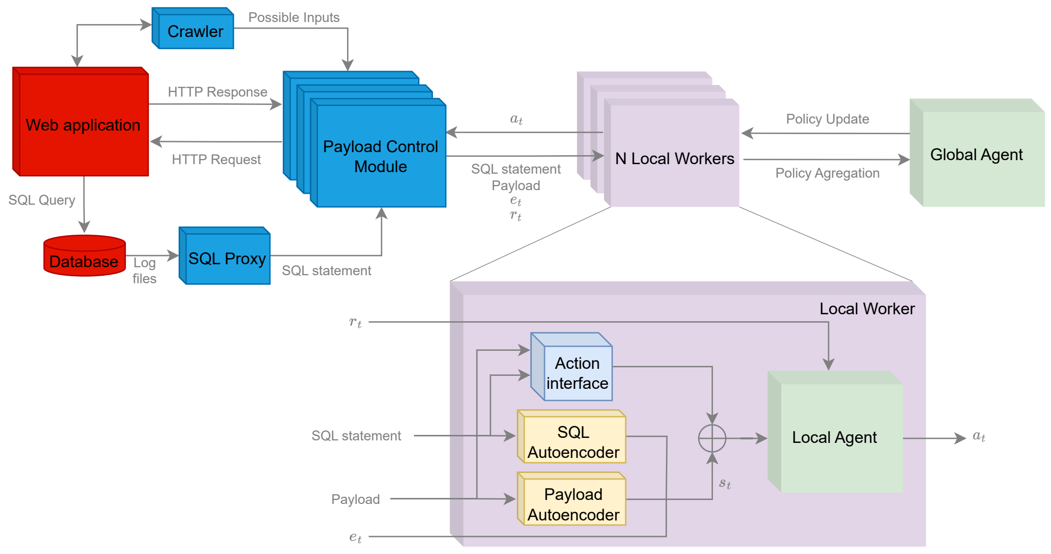

Figure 2: SQIRL architecture. RL agents (green) compute the mutation of the current payload. The *N* federated local workers (purple) train different agents to gather experiences for the global agent. Environment elements that interact with the web application (blue) then find input locations, submit payloads in requests, and monitor the database for successful injections.

We reformulate the problem of creating valid exploits into three separate RL games. *Context Escape* consists of escaping the intended context of the input in the SQL statement. This allows the payload to execute as SQL as discussed in Section 3, thereby reducing false negatives. This is challenging as payloads can be injected in a variety of different contexts in SQL. By doing this, agents implicitly learn and correct the SQL syntax. Examples of SQL contexts are shown in Examples B - D in Table 1. *Behaviour Change* consists of changing the behaviour of the SQL statement to cause unintended functionality. In SQLi this can be either a change in the output of the query, or a manipulation of the database or its data. To prevent any damage on the target, SQIRL demonstrates SQLi with a `SLEEP` statement as proof-of-concept. *Sanitisation Escape* is triggered when trying to solve the first two games. When a web application changes the payload and prevents it from being effective, SQIRL must attempt to escape sanitisation by replacing the affected statement with a semantic equivalent one, an approach that is limited in existing scanners (see Section 3.2).

### 4.1.1 Crawler

Given a starting URL, the Crawler is able to recursively look for anchor tags and URLs within the same domain, which are then queued for exploration. On each web page, the Crawler finds two types of possible input: forms and dynamic URLs. In each input, it injects a unique token that is searched for by the SQL Proxy component, to identify unique input-output combinations that correspond to interaction with the web application database.

Note that an exhaustive search of the input injection space is beyond the scope of this work. Instead, we extended the state-of-the-art Black Widow crawler [14]. In particular we capture dynamic AJAX requests to increase the attack surface analysed by SQIRL. We also integrate the crawler with the new SQL Proxy component, to detect the unique tokens injected in each input without relying on information leakage on a web page, extending our coverage to blind injections.

### 4.1.2 SQL Proxy

The SQL Proxy acts between the Payload Control Module and the database. It retrieves an SQL statement from the database log files which contains the unique token present in the current payload, or reports if an error occurred. The SQL Proxy is used for the crawling and fuzzing phase of SQIRL. To reduce redundancy in the fuzzing effort, the SQL Proxy removes duplicate statements, and if multiple SQL statements exist containing the unique token, it selects the one most similar to the original SQL statement triggered during crawling.

### 4.1.3 State

A *state* at time $t$ is made of three different components: the payload injected at $t-1$; the SQL statement that has been extracted from the SQL Proxy at time $t-1$ (in the case of an error and no new SQL statement, we use the one captured at $t-2$); and a bit $e$ to flag if there was a SQL error at the previous time step. To reduce the space of possible states, and encourage generalisation, the payload and SQL statements are converted into a generic form: string and integer values

**Algorithm 1:** The transition function used to compute the reward, termination condition, and next game.

```
Function transition($e_t$, payload, step, max_step):
    done = False
    step++
    if $e_t$ == 0:
        if behaviourChanged(payload):
            r = 0
            done = True
        elif sanitised(payload):
            game = sanitisation_escape
            r = -1
        elif escapedContext(payload):
            game = behaviour_change
            r = -1
        else:
            game = context_escape
            r = -1
    else:
        game = context_escape
        r = -1
    if step == max_step:
        done = True
    return r, done, game
```

are replaced by the generic tokens `STR` and `INT`, respectively.

### 4.1.4 Extrinsic Reward

RL agents need feedback on how their actions perform. Thus, we define a reward function to motivate the discovery of SQLi vulnerabilities. Our handcrafted, or *extrinsic* reward is designed to have the agent progress through the three games to reach a legitimate vulnerability. We base this on the standard sparse reward function used in RL, where agents are only rewarded for completing the final objective; however, we use a negative reward to incentivise the agent to find legitimate payloads as quickly as possible.

The transition function used to compute the extrinsic reward is shown in Algorithm 1. If no error occurs, we detect if the payload changed the SQL behaviour, triggering the successful termination condition and rewarding 0. For instance, if a `SLEEP` keyword is injected and is unsanitised outside of the intended context it would be a legitimate SQLi payload. Otherwise this provides a reward of -1 if the payload: 1) fails to escape the context, 2) is sanitised, 3) doesn't change the behaviour of the executed SQL, or 4) results in a SQL syntax error. The `behaviourChanged`, `sanitised` and `escapedContext` conditions in the transition function are implemented as syntactic checks and regular expressions.

### 4.1.5 Payload Control Module

The local agents interact with the web application via the Payload Control Module (PCM). First, the PCM translates an action into a concrete payload, and injects it into the web application. Then, it computes the resulting state and the reward, and feeds them back to the relevant agent.

To generate concrete payloads, the PCM uses a vocabulary of tokens dependent on the current objective of SQIRL. Tokens can be found in Table 2. Tokens are curated using existing payloads found in the literature [6, 58] and a popular GitHub SQL payload list with 809 unique SQLi payloads, 3,400 stars and 900 forks at time of writing [51]. We split actions into three categories corresponding to each game, to reduce the number of actions an agent can take:

- *Context Escape:* Add or remove *basic tokens* such as `'`, `#`, `1=1`, `SELECT` from the payload.
- *Behaviour Change:* Add or remove *behaviour changing* tokens such as `AND SLEEP(0)`, `WHERE SLEEP(0)`, `OR SLEEP(0)` from the payload.
- *Sanitisation Escape:* Alter existing payload tokens, for example by keyword capitalisation (`SeLect`), char token (`CHAR(20)`), and commenting out white space.

Each of the Context Escape and Behaviour Change tokens can be inserted at any point in the payload. Each of the tokens can also be removed at any point in the payload. This allows SQIRL to search the entire payload space for valid payloads: as tokens can be inserted at any point, there is an infinite number of payloads that SQIRL can create. In practice, SQIRL will learn to search this space for patterns in the payloads, and tailor payloads to each injection point. The tokens present in Table 2 are also used to represent the payload and SQL statement for input into the autoencoders of SQIRL. Greyed tokens are used only for representation, and not for actions. One advantage of our token structure is that the concrete actions on the payload can be changed depending on the specific SQL syntax, allowing for even greater coverage between the different SQL dialects (e.g. PostgreSQL and MySQL) without having to retrain SQIRL.

After forming the concrete payload, the PCM sends it to the relevant input of the web application and receives from the SQL Proxy the new SQL statement executed. The PCM then uses the transition function in Algorithm 1 (described in Section 4.1.4) to determine the next game, the termination condition, and the extrinsic reward for the payload.

To generate a payload, SQIRL takes a series of actions or steps in an episode to mutate the payload until a vulnerability has been found, or the `max_steps` termination condition is reached. When this occurs, the PCM resets the payload to the empty token (' '), generates a new state, and sets the game to Context Escape.

## 4.2 Local Worker

The local worker is the core of SQIRL, containing the logic to handle actions, state and perform the main learning tasks.

Table 2: Token space used for SQL payload and statements. SQIRL actions are shown in white, grey for SQL representation.

| Action Type | Token/Action | Example | Description |
|---|---|---|---|
| Basic Tokens (addition, removal) | Comma | , | |
| | Comment | #,–, convert # to – | Comments at any location |
| | Comment tokens | `# TOKEN_TO_COMMENT` | Comment out specific tokens |
| | Conditional | `1=1`, `"wdd" = "wq"` | A conditional statement |
| | Identifier (MD5 Hash) | 351bf115b49fdc38b92e00482000e9cd | Identifier based on step counter and random salt |
| | Keyword | `SELECT`, `OR` | SQL Keywords |
| | Number | `1.0`,`2` | Number tokens as Float or Int |
| | Operator | >, =>, /,* | Logical Operands |
| | Paranthesis | `),(` | |
| | Quotes | `',",`` ` `` | |
| | Statement | `SELECT * FROM Users WHERE SLEEP(0)` | Complex SQL statements |
| | String | `"ed"` | Abstract strings present in the SQL statement to a simpler form |
| | Whitespace | `( )` | A whitespace token |
| | Hex | 0x12A | Hex characters |
| Behaviour Changing Tokens (addition, removal) | `AND` | `AND SLEEP(0)` | SQL AND statement |
| | `OR` | `OR SLEEP(0)` | SQL OR statement |
| | `IF` | `IF (1=1) THEN SLEEP(0) ELSE SLEEP(10)` | SQL IF statement |
| | `UNION` | `UNION SELECT * FROM Users WHERE SLEEP(0)` | SQL UNION statement |
| | `WHERE` | `WHERE SLEEP(0)` | SQL WHERE statement |
| | `SLEEP` | `SLEEP(0)` | SQL SLEEP statement |
| | Keyword random capitalistion | `SelECt` | Randomly capitalise letters in all selected Keyword |
| | Whitespace obfuscation | ( ) becomes (**\) | Convert all whitespace elements in the payload to comments |
| | AND obfuscaion | `AND` becomes `&` | convert all AND tokens to & |
| Sanitisation Escape (token operation) | Char token | `CHAR(TOKEN)` | Convert the selected token to its numeric ASCII representation and include it in a CHAR operation |
| | Concat token | `CONCAT('TOKEN_PART_1', 'TOKEN_PART_2')` | Split the selected token in two parts then include in a CONCAT operation |

#### 4.2.1 Representation Learning

In Section 3.3.2, we argued in favour of representing SQL statements and payloads in a way that captures the interdependencies between tokens. Since SQL statements and payloads have no fixed size, we use Gated Recurrent Unit (GRU) autoencoders, which are designed to represent variable length sequences and the semantic relationship between their elements. Tokenising and one-hot-encoding techniques also capture dependencies, but rely on fixed length inputs. Clipping payloads and SQL statements to a fixed length introduces noise or discards information valuable to the action-selection process, making such techniques unsuitable.

We use separate autoencoders for representing SQL statements and payloads, as we expect them to learn different latent representations for the same tokens (for example, '`'`' will have more significance in the latter, and will not appear there in matched pairs).

Starting from the generic representation of the payload and SQL statement, we convert them into sequences of tokens from Table 2. Each autoencoder takes as input a token at a time. The first layer produces as output an 800-float vector for the SQL statement, and of 280-float vector for the payload. These are then fed into the respective GRUs together with the output of the GRUs on the previous tokens of each sequence. The output of each GRU is a 1024-float vector. Using a reconstruction loss, we train the autoencoders in a self-supervised way to reconstruct their inputs (the SQL statements, or the payloads). To train these models we generate a dataset of 100,000 samples for each autoencoder. Each payload is generated by taking 30 random actions. MySQL statements are generated using the grammar in Table 12. Through the process of training, the autoencoders learn to recreate the input data. When running SQIRL, the autoencoder outputs and the state error bit $e_t$ are concatenated, yielding a state of 2049 features.

The use of autoencoders provides SQIRL with an automated feature set that represents the payload in the latent space, avoiding the pitfalls that RL agents can fall into from manual features, such as difficulty learning, learning an incorrect policy for the task, or exploiting unintended mechanics.

The features are then extracted from the autoencoders and input into a DQN. This contributes an off-policy RL algorithm, more data efficient than alternative RL algorithms such as A2C [30] or PPO [43], which discard data after each update, requiring longer training. Tabular Q-learning was considered, but it is infeasible to handle the large number of states resulting from our formulation of the SQLi game.

#### 4.2.2 Action Ranking

In SQIRL, the number of possible actions increases with the payload length, as tokens can be added or removed from any location within the payload. For example, at time $t = 0$ there is only the possibility to add a token at one location, hence

27 possible actions. At $t = 1$, assuming the payload is not sanitised, there is the possibility to add tokens before or after the previous token, or remove the previous token, hence there are 55 possible actions.

Traditional RL computes a Q-value for each action as the output of the DQN, selecting the action with the highest value. Due to the dynamic action space of SQIRL, we are unable to use this same mechanism. Instead we alter the DQN architecture to compute a Q-value one action at a time. We do this by taking the available actions at the current timestep and converting these into a set of action representations, via the Action Interface in Figure 2. These representations consist of a 4-tuple that includes: the action number, indices of where the action would mutate the payload (e.g. placing # at index 4, or keyword capitalisation from index 4-9), and action class type (e.g. Comment or Operator).

We are thus able to compute Q-values for a variable number of actions and select the action with the highest associated Q-value as the mutation to perform on the payload. This allows SQIRL to take dynamic actions to generate payloads, going beyond the state-of-the-art practice of using a pre-existing formula, a grammar, or a set of payloads.

#### 4.2.3 Intrinsic Reward

Besides the extrinsic reward of Section 4.1.4, we use Random Network Distillation (RND) [9] to generate an *intrinsic* reward, which encourages the local agents to take different actions and find new states. The ability of agents to be 'curious' in finding new states has a distinct advantage: it results in agents that are able to generalise better when facing new tasks (such as exploiting different SQLi endpoints).

The RND is implemented by associating to each local agent a Target and a Predictor neural networks. A local agent feeds a state $s_t$ to both, and over time the Predictor learns to predict the output of the Target. The intrinsic reward is computed as the loss between the outputs of the two networks, and is bound between 0 and 0.7 (see Table 9).

### 4.3 Global Agent

Rather than training a single RL agent, we use worker agents in a vertical federation to accelerate learning, and provide a more stable learning curve. The intuition is that we want to harness different experiences that agents gather from their independent interactions with the same target. To do so, we create a Global Agent containing a global policy, and several local workers with their local policies. Each of the workers is able to interact with its own instance of the PCM, keeping the objective dynamics separate as the agents execute separate payloads in parallel. This knowledge is then aggregated every 20 episodes by averaging the neural network weights of the workers into the global policy which in turn updates the workers individual weights.

### 4.4 Training

As a source of examples to train SQIRL, and as a basis for comparison with other scanners, we created a novel *SQLi MicroBenchmark* (SMB). The 30 vulnerable samples cover a didactic and realistic range of different SQLi contexts and sanitisations, and are drawn from CVEs and established sources [11,37,40]. For each vulnerable sample, we add to the benchmark also a non-vulnerable version of the same injection point, where we parameterise the underlying query and sanitise each parameter. We implement the SMB as an easy-to-crawl PHP web application which reflects the user input in the response web page, so that a crawler can find the test input it submitted.

We used the 20 easiest vulnerable samples to train SQIRL, and we left the harder 10 as a test set to demonstrate its generalisation ability to novel examples not seen during training. During testing, we include also the 30 non-vulnerable samples to estimate false positives. The training set contains multiple sanitisations and a variety of different contexts, and SQIRL can learn the value of all actions in its action space. See Table 10 for the SQLi statements and sanitisations.

We conduct training and experiments on a 64GB Intel i7 CPU. For each sample, we train until SQIRL is able to produce 14 valid payloads in the last 20 episodes, or until the maximum of 200 episodes is reached. Each episode consists of up to 30 steps (each corresponding to a fuzzing request). Random actions are taken according to an $\varepsilon$-decay policy to ensure exploration of the payload space. This is to balance the policy of an agent to produce valid payloads, and the stochasticity of $\varepsilon$ taking random actions, in line with [16]. We use 4 worker agents for training. A grid-search was conducted on the individual components of the workers to select hyperparameter values, details of which can be found in Table 9.

## 5 Evaluation

Our evaluation compares SQIRL to 5 state-of-the-art scanners on the SMB of Section 4.4 and on 14 production grade web applications. The primary performance measure is the ability to find SQLi vulnerabilities (true positives). The high volume of traffic generated by automated web fuzzing tools is considered one of their main shortcomings, alerting defenders that an attack is taking place. Moreover, if a scanner needs fewer requests than another to exploit the same injection point, it can be considered more 'intelligent' [10]. Hence, our secondary performance measure is the number of fuzzing requests sent (excluding crawling requests). Finally, we also report average run time, although broadly speaking the total time, running in minutes at most, is not a significant concern for vulnerability scanning.

Table 3: Comparison of SQIRL against state of the art scanners on the SMB. Dark solid bars (red) emphasise poor performance, light bars (green) emphasise good performance. *Spurious Positives* are TPs deemed invalid after manual analysis.

| Tool | | Feedback | Exception | Avg Requests per Vuln Input | Avg Requests per Non-Vuln Input | Average Run Time (s) | Spurious Positives | FP | TN | FN | TP |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ZAP | built-in | ✓ | - | 86.9 | 98.7 | 1.2 | 10 | 21 | 9 | 7 | 23 |
| | advanced | ✓ | - | 7760.1 | 8208.4 | 61.1 | 0 | 0 | 30 | 9 | 21 |
| | built-in | ✗ | - | 99.8 | 100.6 | 1.4 | 22 | 22 | 8 | 8 | 22 |
| | advanced | ✗ | - | 8031.9 | 8208.4 | 66.2 | 0 | 0 | 30 | 9 | 21 |
| Sqlmap | | ✓ | - | 2234.5 | 4524.5 | 148 | 0 | 0 | 30 | 13 | 17 |
| | | ✗ | - | 2212.5 | 4533.2 | 145 | 0 | 0 | 30 | 13 | 17 |
| BurpSuite | | ✓ | - | 220.5 | 234.3 | 47.1 | 0 | 0 | 30 | 8 | 22 |
| | | ✗ | - | 279.5 | 279.0 | 46.9 | 0 | 0 | 30 | 13 | 17 |
| Arachni | | - | ✓ | 117.7 | 121.5 | 17.8 | 0 | 0 | 30 | 0 | 30 |
| | | - | ✗ | 121.3 | 121.6 | 17.2 | 0 | 0 | 30 | 30 | 0 |
| Wapiti | | - | ✓ | 23.8 | 35.0 | 1.0 | 0 | 0 | 30 | 9 | 21 |
| | | - | ✗ | 35.1 | 34.6 | 1.9 | 0 | 0 | 30 | 30 | 0 |
| RAND-SQIRL | | - | - | 92.9 | 300.0 | 5.7 | 0 | 0 | 30 | 8 | 22 |
| SQIRL | | - | - | 24.8 | 300.0 | 65.5 | 0 | 0 | 30 | 0 | 30 |

## 5.1 Experimental Setup

We compare SQIRL against a number of state-of-the-art scanners. We select Sqlmap v1.6, an open source SQLi-specific scanner with 24.6k stars on GitHub, also included in the Kali Linux distribution popular with penetration testers. We further select 4 general security scanners with SQLi functionality: OWASP ZAP v2.11.1, BurpSuite Pro v2022.6.1, Arachni v1.6.1.3, and Wapiti 3.1.2. Whilst we would have liked to compare with existing academic scanners for black-box detection of SQLi, the implementations of these are not available [4, 28, 42, 55]. DeepSQLi [26] references an available implementation, but the cited repository appears to be incomplete and not usable. We run each of the scanners for a maximum of 3 hours to ensure consistency in the scanning. We report scanner configurations in Table 7.

During testing, we modify SQIRL by halving the learning rate, and reducing the rate to take random actions to 0.1, using the same decay. This places greater emphasis on the learnt policy but still allows for exploration of new states. We allow SQIRL a maximum of 10 episodes per parameter fuzzed, relying on its ability to discover payloads quickly, and limiting unnecessary attempts for non-vulnerable inputs.

As an additional fuzzing baseline, we force SQIRL to *only* take random actions to generate payloads, ignoring policy and decay. We call this variant RAND-SQIRL.

## 5.2 Comparison on SMB

In this experiment we investigate the performance of SQIRL and the state-of-the-art scanners of Section 5.1 on the SMB of Section 4.4, consisting of 30 different examples of SQLi (*positives*), which use a variety of different contexts and sanitisations, and their 30 fixed variants (*negatives*).

Sqlmap, ZAP, and BurpSuite can benefit from receiving the SQL query result in the web page as feedback. Arachni and Wapiti instead can benefit from seeing on the web page the exception trace caused by malformed queries. We test each of these scanners twice: in the default configuration of the benchmark (reflect input only), and with the additional beneficial output (query results or exception trace, as needed). ZAP can use its built-in SQLi plugin, or use an advanced plugin to improve performance: we test it in both configurations. Table 3 shows the aggregate results of this experiment. The key performance metric is the number of vulnerabilities found (TP). SQIRL and Arachni are the only scanners able to find all the 30 vulnerabilities. BurpSuite, RAND-SQIRL (22), ZAP-advanced and Wapiti (21) find two-thirds of the vulnerabilities, and Sqlmap (17) finds half.

Wapiti and Arachni (0) are ineffective when the exception trace is not shown on the page, indicating that their performance on production web applications may suffer. Sqlmap does not appear to leverage the feedback, as it finds 17 injections irrespective of feedback. RAND-SQIRL and SQIRL need neither feedback nor exceptions, as they leverage the SQL Proxy information.

ZAP-built-in suffers from a high percentage of FPs (two thirds). Moreover, upon manual inspection, it turns out that most of its TPs (10 with feedback, 22 without) are *spurious*. This is due to tests which, on certain inputs, fail to verify the presence of SQLi, yet lead ZAP to report SQLi. A concrete example is a test that compares the page output length after injecting two payloads, expecting that the first would cause the query to return some results (`ZAP' OR '1'='1' --`) and that the second would not (`ZAP' AND '1'='2' --`). The injection fails (say single quotes are sanitised), but ZAP thinks it succeeds, because it mistakenly detect a difference in page size due to the reflected (ineffective) payloads.

The average number of requests per vulnerable input measures how 'intelligent' a scanner is, in terms of using effective payloads instead of brute-force. Wapiti and SQIRL, with less than 25 requests, are the most effective. Wapiti leverages a curated set of 35 payloads, and SQIRL benefits from the use of RL (RAND-SQIRL takes 4 times as many requests). Thanks to its small set of payloads, Wapiti is also the scanner that uses fewer requests to detect a TN, as the other tools give

Table 4: Comparison of SQIRL and state of the art tools, on production grade web applications.

| Tool | Average Requests per Vuln Input | Average Time (s) | WordPress & Plugins | | | B2evolution | | | Sourcecodester e-learning | | | Sparks Hotel Management | | | Total | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | FP | FN | TP | FP | FN | TP | FP | FN | TP | FP | FN | TP | FP | FN | TP |
| ZAP (bulit in and advanced) | 4414.0 | 452.5 | 5 | 4 | 4 | 0 | 1 | 0 | 0 | 2 | 4 | 0 | 1 | 17 | 5 | 8 | 25 |
| Sqlmap | 2280.5 | 778.2 | 0 | 1 | 7 | 0 | 0 | 1 | 0 | 4 | 2 | 0 | 6 | 12 | 0 | 11 | 22 |
| BurpSuite | 211.0 | 276.6 | 0 | 3 | 5 | 0 | 0 | 1 | 0 | 1 | 5 | 0 | 9 | 9 | 0 | 13 | 20 |
| Arachni | 720.0 | 446.8 | 0 | 6 | 2 | 0 | 1 | 0 | 0 | 1 | 5 | 0 | 5 | 13 | 0 | 13 | 20 |
| Wapiti | 30.0 | 63.1 | 0 | 7 | 1 | 0 | 0 | 1 | 0 | 6 | 0 | 0 | 14 | 4 | 0 | 27 | 6 |
| RAND-SQIRL | 475.0 | 133.1 | 0 | 0 | 8 | 0 | 0 | 1 | 0 | 0 | 6 | 0 | 1 | 17 | 0 | 1 | 32 |
| SQIRL | 111.1 | 159.2 | 0 | 0 | 8 | 0 | 0 | 1 | 0 | 0 | 6 | 0 | 0 | 18 | 0 | 0 | 33 |

up finding a vulnerability after a higher number of attempts. Sqlmap's rule-based approach shows some effectiveness in payload generation, in that it takes half as many attempts in exploiting a vulnerable input than in exhausting a non-vulnerable one. ZAP-advanced instead, with over 8k requests, seems to rely mostly on brute force.

In terms of time elapsed, the average time spent on an input varies between 1 second and 2.5 minutes. Simpler approaches, in terms of number of payloads (Wapiti), or design (RAND-SQIRL) are faster than high volume (Sqlmap, ZAP-advanced) or intelligent scanners (SQIRL).

## 5.3 Comparison on Production Web Apps

In this experiment, we compare the ability of all scanners to find vulnerabilities in production grade web applications. We select at random 10 open source web applications from a popular repository [33], and 4 web applications with SQLi CVEs reported in 2021 with CVSS 3.1 scores more than 7.1. One of these is WordPress, for which we select 7 popular plugins with more than 2k downloads. The list of selected applications and plugins is in Table 8.

In order to avoid any bias due to the use of different crawlers, we test all of the scanners on each input (URL and parameter) found by SQIRL, and we also let the scanners crawl the entire web applications on their own, to verify that SQIRL's crawler did not miss any additional vulnerabilities. We also provide each scanner the session cookie, to ensure they are logged in. For ZAP, we enable both the built-in and advanced plugins.

Table 4 reports the results of the experiment on the 4 applications for which vulnerabilities were found. SQIRL is the only scanner able to identify all of the 33 vulnerabilities. Arachni, which found all vulnerabilities in the SMB, found only 20 in production. A similar drop in performance is observed in Wapiti, detecting only 6. This was expected as both scanners performed poorly on the SMB without exception feedback, and production applications should not expose exception traces. For example, Arachni fails to detect the vulnerability CVE-2022-2489 in Sourcecodester e-learning. The vulnerability is on a web page with a search bar, which internally runs the following vulnerable query: `SELECT * FROM posts WHERE body LIKE '%USER_INPUT%' AND courseCode='class101_a' ORDER BY id DESC`. The search results are shown on the page itself, and several scanners are able to find this vulnerability successfully. For example, BurpSuite uses the payload `41896576' or 1334=1334--`. Arachni instead attempts payloads of the form `%TEXT"'`--` which intentionally trigger an exception (due to the ` between the ' and the comment). Since Sourcecodester does not reveal the exception on the page, returning instead a default empty search result, Arachni fails to detect success, and reports an FN.

SQIRL has the lowest average number of requests (111) of any effective scanner. ZAP, which is the third-best scanner with 25 TPs, requires 44 times more requests than SQIRL, takes 3 times longer, and is the only scanner to produce false positives.

RAND-SQIRL finds 32 TPs, missing only one, but takes more than 4 times as many requests as SQIRL, for a comparable execution time. This shows that although our dynamic action space is already effective in generating diverse payloads, the additional use of RL in our approach substantially reduces both the number of requests required, and the variance in the payload discovery process: SQIRL has perfect performance in both experiments, whereas RAND-SQIRL had middling performance on the SMB. The ability of SQIRL to identify in total 63/63 vulnerabilities with a low number of requests, after being trained on 20 of them, shows that it has effectively learned, and was able to generalise to new data.

Remarkably, 22 of the 33 vulnerabilities discovered by SQIRL are new zero-days in apps/versions which had already been reviewed for vulnerabilities (the remaining 11 vulnerabilities), demonstrating that the present approach is practically useful and effective. We have responsibly disclosed the new vulnerabilities in WordPress (CVE-2022-2717, CVE-2022-2718), Sparks (CVE-2022-38918, CVE-2022-38919, CVE-2022-38920), Sourcecodester (CVE-2022-38921), and B2evolution.

## 5.4 Distributed Learning Capability

SQIRL uses multiple worker agents which fuzz the same SQLi and reconcile their learning via the global agent every 20 episodes. This is the simplest case of federated learning. We investigate if SQIRL could benefit from applying the workers

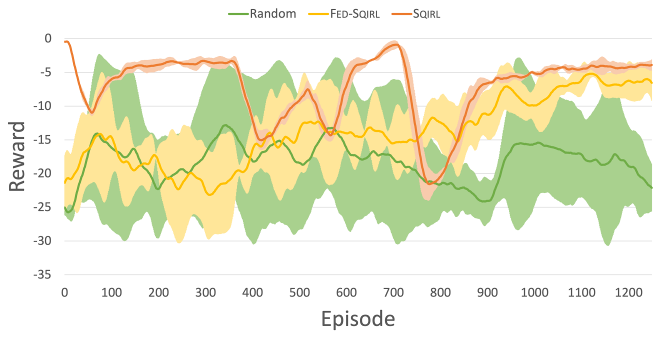

Figure 3: Four federated workers of SQIRL fuzzing random targets of the SMB.

to different tasks (input targets, or even web applications), sharing their experiences, in a potentially privacy-preserving way, as in traditional FL.

We therefore use federated RL using 4 local workers to interact with different, randomly chosen SMB vulnerabilities. We call this variant FED-SQIRL. We assess its learning performance by measuring the reward the workers receive. We compare with two baselines: a version of FED-SQIRL that can take only random actions, and SQIRL with 4 workers using the same budget of episodes (hence all workers train on the same vulnerabilities).

We show the experimental results in Figure 3. The random baseline shows significant variance, as expected, as each worker targets a training sample of different complexity. SQIRL shows minimal variance, as it benefits from each worker targeting the same training sample. The irregularity in the learning curve, corresponds to the switch to different training samples, where learning 'restarts' and eventually stabilises. FED-SQIRL sows a significant variance in the workers moving averages. This is due to the different SQLi that each worker is exploiting, requiring the generation of different payloads. Yet, the incremental increase in the moving average, and the gradual reduction of the variance around it, demonstrate that learning is still happening, as the agents improve the quality of the exploits.

The significance of FED-SQIRL is that, with a small loss of performance from SQIRL, we can distribute the learning across local agents which could be deployed across different organisations, targeting proprietary web applications, yet mutually improving their SQLi detection capabilities. We leave the investigation of a practical deployment of distributed learning to future work.

### 5.5 SQIRL Ablation Study

To measure the impact of the various design choices behind SQIRL, we present an ablation study of the model. We start from RAND-SQIRL, as described above, and gradually introduce and compare the use of RL, its enhancement with autoencoders (AEs) and RND, and the use of the multi-worker architecture (SQIRL) and distributed learning (FED-SQIRL).

Table 5: Ablations of SQIRL on the SMB

| Agent | Avg Cumul Reward | SQLi Found | Avg Time (s) per Vuln Input | Avg Requests per Vuln Input |
|---|---|---|---|---|
| RAND-SQIRL | 585.6 | 22 | 5.7 | 91.9 |
| DQN, 1-hot encoded | 682.4 | 25 | 191.7 | 65.8 |
| DQN, AEs | 765.3 | 26 | 26.8 | 50.1 |
| DQN, AEs, RND | 799.1 | 30 | 31.1 | 41.8 |
| SQIRL: Multi-Worker DQN, AEs, RND | 984.3 | 30 | 27.9 | 24.0 |
| FED-SQIRL: Distributed DQN, AEs, RND | 917.7 | 29 | 29.7 | 33.0 |

We train each model on the SMB (Section 4.4) and challenge each to find its 30 vulnerabilities. The results can be found in Table 5. We include the average cumulative reward, normalised so that it is analogous to common RL practice, where greater positive reward implies better performance. At time $T$, the reward is computed as follows:

$$r_{sum} = \sum_{t=0}^{T} min(40 + (r_t), 30) \tag{1}$$

where $r_t$ is the average reward of the the workers in the episode $t$.

We observe that each improvement from RAND-SQIRL to SQIRL increases the number of vulnerabilities detected and decreases the number of requests necessary to do so, justifying its introduction. RND improves the generalisation of the model, leading to an increase in reward and reduction in number of requests. The AEs drastically reduce the average time. The multi-worker architecture almost halves the number of requests used. Finally, the introduction of distributed learning introduces only a small performance cost, to the potential benefit of a broader application scenario.

## 6 SQLi Payload Analysis

The payloads generated by scanners reveal more about their general capabilities. In Table 6, we display the different features that each scanner generated, including context escape (comments, quote and parentheses) and sanitisation bypass (concatenation of parameters, capitalising key features, whitespace escaping, `AND` escaping). These are not exhaustive, as they come from the payloads observed during the experiments in Section 5. We provide SQIRL payloads in Listing 1, and payloads from all scanners in Table 11.

ZAP-built-in is only able to insert comments to clean up the SQL after the payload, even when there is feedback on the page, generating two payload types. With the advanced plugin, it escapes both single quotes and parentheses. Relying only on these features causes ZAP to miss the most complex cases in the SMB and the real world web applications. ZAP

Listing 1: Generic representation of payloads generated by SQIRL.

```
❶ BASE64 AND SLEEP (0.0)
❷ BASE64" And SLEEP (0.0)'  AND SLEEP (0.0)#
❸ BASE64) anD SLEEP (0.0))#
❹ BASE64' aND slEEp (0.0)#
❺ BASE64'/**/&/**//**/SLEEP/**/(0.0))#
❻ BASE64/**/AnD/**//**/SLEEP/**/(0.0)" /**/AND/**//**/SLEEP/*
    */(0.0)#
❼ BASE64 and sleep(0.0)
❽ BASE64' & SLEEP (0.0)#
❾ BASE64' AND SLEEP (0.0)-- --
❿ BASE64'#
```

tends to use payloads that are similar in structure, with little variance. It is the only scanner to have payload structure overlap with another scanner, Sqlmap, with two in common. Sqlmap is the only scanner to use the `CONCAT` function to avoid sanitisations. BurpSuite displays consistency in the features it uses to generate payloads, irrespective of feedback. It creates simple quote, `select`, and `or` based payloads, making the most use of feedback compared to other scanners, producing 6 additional payloads. Arachni uses both comments and single quotes in SMB to identify all SQLi when there is an exception present in the page. It uses two simple payloads (`'"`--`, and `INT!%TEXT\"'`--`) to escape context and use a comment to clear up the remaining SQL. Wapiti, like Arachni, uses simple payloads that attempt to escape multiple contexts without using SQL keywords (`'"()`). This lets it identify 13 SQLi when the exception trace is present in the SMB. For the web applications, Wapiti only uses one alternative payload structure: `AND INT=INT AND INT=INT`. This limited variety in payloads leads to the poor performance observed in Table 4.

As intended, SQIRL generates a greater variety of payload structures than other scanners (Listing 1). SQIRL uses capitalisation obfuscation in payloads ❹, ❺ to solve 12 and 13 in SMB (where other scanners require feedback or exceptions to identify this). We see SQIRL's ability to generalise as it reuses ❹ to identify CVE-2021-24786. SQIRL is also the only scanner that makes use of whitespace obfuscation to exploit 14, and that replaces `AND` with `&` (in ❽) to bypass sanitisation of 15. We can see that SQIRL favors payloads of the form similar to ❶, mutating this to include quote marks that escape the context, comments (`--`, `#`) to prevent issues with trailing SQL, in addition to the obfuscation techniques. This shows that SQIRL has leveraged reinforcement learning to develop a minimal payload and then mutate this to tailor it to each input case, resulting in the low number of requests seen in Section 5.

Using the same technique as in Section 3.1, we generate latent representations and preform a PCA of existing payloads and payloads generated by SQIRL. Figure 4 shows that some payloads from SQIRL overlap with existing payloads (faint orange points are visible towards the right-center of the distribution) yet a significant fraction occupies new space, showing

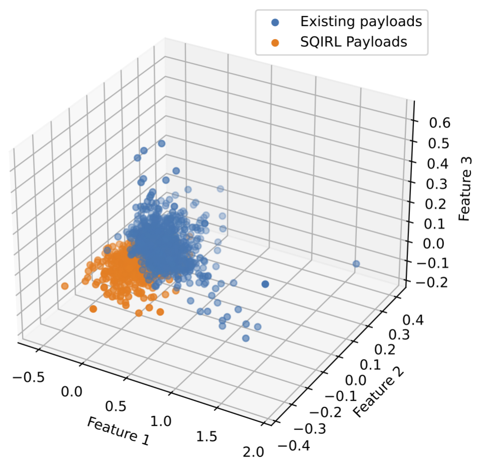

Figure 4: PCA of existing payloads and those from SQIRL.

Table 6: Features that different scanners can make as observed in the payloads seen. **F** denotes use of feedback in a web page, **E** exceptions.

| Tool | **F** | **E** | Comment | Payload Types | Single Quote Escape | Parentheses Escape | Concat | Caps Escape | Whitespace Escape | `AND` Escape |
|---|---|---|---|---|---|---|---|---|---|---|
| ZAP (built-in) | ✓ | – | ✓ | 2 | | | | | | |
| ZAP (advanced) | ✓ | – | ✓ | 5 | ✓ | ✓ | | | | |
| ZAP (built-in) | ✗ | – | | 0 | | | | | | |
| ZAP (advanced) | ✗ | – | | 4 | ✓ | ✓ | | | | |
| Sqlmap | ✓ | – | ✓ | 5 | ✓ | ✓ | ✓ | | | |
| | ✗ | – | | 3 | ✓ | ✓ | | | | |
| BurpSuite | ✓ | – | ✓ | 8 | ✓ | | | ✓ | | |
| | ✗ | – | ✓ | 2 | ✓ | | | ✓ | | |
| Arachni | – | ✓ | ✓ | 5 | ✓ | | | | | |
| | – | ✗ | ✓ | 0 | | | | | | |
| Wapiti | – | ✓ | | 2 | ✓ | ✓ | | | | |
| | – | ✗ | | 0 | | | | | | |
| SQIRL | – | – | ✓ | 10 | ✓ | ✓ | | ✓ | ✓ | ✓ |

that SQIRL has learnt to use the action space to produce out-of-distribution payloads compared to existing payloads. Note that only a fraction of the latent space is occupied, confirming the intuition that the space of valid SQLi payloads is smaller than all possible payloads.

## 7 Discussion and Limitations

*Grey-Box Approach.* SQIRL makes use of information from the SQL database, but does not have access to the web application source code, making it a grey-box tool. Thus, we compare it against scanners that require similar levels of access to web applications, and white-box approaches are out of scope. We argue that that the grey-box approach is well-suited for developers and security practitioners, for two reasons. First, vulnerabilities can be found without relying on the insecure practice of leaking error information on the web pages themselves (SQL statement results or exceptions). Second, the approach is effective for pentesters, whom can be allowed to obtain access to the database logs, whereas it is not useful for malicious actors targeting deployed web applications.

*Training vs Testing.* Note that 20 of the SMB positives were already used to train SQIRL: their reuse for testing is standard RL practice, as success during training is already evidence of the RL agent success in its task. The agent does not receive feedback from the labels of the samples, but only from changes to the observable state. The case is different for supervised learning, where training feedback 'leaks' information about the sample labels, which should therefore not be used for testing. While training on a vast number of SQLi could increase the ability of SQIRL to generate even more diverse payloads, we found it valuable to demonstrate that even a small training set of 20 samples was enough to find all the vulnerabilities in the experiments of Section 5, achieving a better performance than established scanners.

*Crawling.* As explained in Section 5, we took steps to remove crawling-induced bias from our experiments. The crawler used in SQIRL was developed as a tool for the payload generation by extending a state-of-the-art crawler, and we do not regard it as a key contribution. Our crawler is unable to utilise CSRF tokens and unique tokens used in forms and dynamic links unless the web page is accessed again. This was investigated, but would double the number of requests sent to the web application. A deeper investigation of crawling is left for future work.

## 8 Related work

*Payload Generation for Web Fuzzing.* Commix [46] is able to develop payloads for command injection using a side channel attack. ML using recurrent neural networks has been used to fuzz web browsers by generating malformed HTML [42]. XML and SQL attacks have also been considered in [5,18,19] by a search strategy to find abnormal responses.

Simpler RL and deep RL methods for XSS payload generation are presented in [10,15,16,23]. While such methods show improvement from using off-the-shelf RL techniques, our ablation study in Section 5.5 suggests that the techniques we considered such as using autoencoders or multiple workers could further improve their payload generation abilities.

*Fuzzing for SQL Injection.* One of the first papers [6] to separate mutations into different classes used rule-based mutation. While SQIRL uses a similar concept to separate actions for our agents, using RL means it can learn the grammar, yet also violate it, to develop SQLi. Lei *et al.* [24] identify inputs in web applications that are 'most likely' to lead to a vulnerability. This work focuses mainly on crawling web applications, using predefined payloads, and heuristic mutations. Zhao *et al.* [58] use combinatorial testing to select mutations to apply to a SQL statement. This worked better against sanitisations, but using predefined SQL statements limits the scanner when finding edge cases, as opposed to SQIRL.

DeepSQLi [26] uses a neural language model to develop SQLi payloads. Starting from a user input, DeepSQLi iteratively generates a new payload using a transformer, guided by a beam search to improve diversity, until it manages to exploit a vulnerability. Similarly to our approach, they use a SQL proxy to provide feedback on the payload success. Comparing against Sqlmap, DeepSQLi achieves a maximum improvement of 29.5% in terms of vulnerabilities found. In contrast, SQIRL achieves a 70.2% improvement on Sqlmap in our experiments. This comparison is only indicative, as the target web applications are different.

Aliero *et al.* [4] present an algorithmic approach, using an object-oriented model. This is however limited in scope, requiring error responses and regex for login conditions. Luo [28] uses genetic algorithms to mutate predefined payloads. A neural network is used to adjust the strategy of the low-level mutations. RL has also been used to modify SQLi payloads in order to bypass web application firewalls [55]. Del Verme *et al.* [12] discuss the difficulties of exploring the diverse payload space and the importance of mutations in SQLi. Erdodi *et al.* [13] then develop two simple RL agents to generate payloads using tabular Q-learning and a DQN. The models are demonstrated on limited cases and are not extended to handle the variety of SQLi of interest for practical use in web applications.

*Defensive coding against SQLi.* SQLi can also be prevented via defensive coding methods. Su and Wassermann [47] track user inputted queries, at runtime, to block those where substrings change the syntactic nature of the query. Valeur *et al.* [54] and Halfond *et al.* [17] develop statistical models of queries, using anomaly detection to prevent queries that do not conform to normal behaviour. Context-Sensitive String Evaluation (CSSE) [39] is application agnostic, using only metadata about queries to determine untrusted user inputs to enforce strict channel separation. Sun and Beznosov [48] present another application agnostic method, which intercepts queries to enforce conformity of input type and intention of the SQL. Kasim [20] presents ensemble methods, including tree search and regex-based feature extraction, to classify SQL queries as benign or malicious.

A widespread adoption of the defensive techniques surveyed here, and of parameterised queries, would drastically reduce the occurrence of SQLi. Yet these solutions have been around at least since 2005, and SQLi still plagues web applications, justifying further research on detection techniques.

## 9 Conclusions

To increase the variety of payloads for SQLi fuzzing, we leverage grey-box fuzzing and present what we believe to be the first fully automated SQLi web scanner that uses RL. Our tool, called SQIRL, uses a single agent to learn how to fix SQL syntax, escape contexts, and bypass sanitisation. SQIRL uses worker agents to improve performance, reducing the number of HTTP requests needed compared to a single agent, and is able to generalise to new, unseen inputs to tailor SQLi payloads. Furthermore, SQIRL was able to find more SQLi vulnerabilities in our SMB and 14 production grade web applications, achieving this in fewer requests than any of five the state-of-the-art scanners we compare against. Vulnerabilities identified by SQIRL have currently led to 6 CVEs. Payload analysis also demonstrates that SQIRL uses more features, to generate more payload types that are out of distribution for existing payload lists used in scanners. Our results demonstrate the use of RL techniques in web applications security, leaving a rich space for future work.

**Acknowledgements**

Research partially funded by EPSRC grant EP/T51780X/1.

## References

[1] sqlmap: automatic SQL injection and database takeover tool, 2023.

[2] Wapiti : a Free and Open-Source web-application vulnerability scanner in Python for Windows, Linux, BSD, OSX, 2023.

[3] AKAMAI. Phishing for Finance. *State of the Internet 7* (2021).

[4] ALIERO, M. S., GHANI, I., QURESHI, K. N., AND ROHANI, M. F. An algorithm for detecting SQL injection vulnerability using black-box testing. *Journal of Ambient Intelligence and Humanized Computing 11* (2020).

[5] ALSMADI, I., ALEROUD, A., AND SAIFAN, A. A. Fault-based testing for discovering SQL injection vulnerabilities in web applications. *International Journal of Information and Computer Security 16* (2021).

[6] APPELT, D., NGUYEN, C., BRIAND, L., AND ALSHAHWAN, N. Automated testing for sql injection vulnerabilities: An input mutation approach. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (2014).

[7] BALZAROTTI, D., COVA, M., FELMETSGER, V., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)* (2008).

[8] BOYD, S., AND KEROMYTIS, A. SQLrand: Preventing SQL Injection Attacks. In *Applied Cryptography and Network Security* (2004).

[9] BURDA, Y., EDWARDS, H., STORKEY, A., AND KLIMOV, O. Exploration by Random Network Distillation. *arXiv:1810.12894 [cs, stat]* (2018).

[10] CATURANO, F., PERRONE, G., AND ROMANO, S. P. Discovering reflected cross-site scripting vulnerabilities using a multiobjective reinforcement learning environment. *Computers & Security 103* (2021).

[11] CLARKE, J. *SQL Injection Attacks and Defense (Second Edition)*, second edition ed. 2012.

[12] DEL VERME, M., SOMMERVOLL, A., ERDODI, L., TOTARO, S., AND ZENNARO, F. SQL Injections and Reinforcement Learning: An Empirical Evaluation of the Role of Action Structure. In *Secure IT Systems* (2021).

[13] ERDODI, L., SOMMERVOLL, A. A., AND ZENNARO, F. M. Simulating SQL Injection Vulnerability Exploitation Using Q-Learning Reinforcement Learning Agents. *arXiv:2101.03118 [cs]* (2021).

[14] ERIKSSON, B., PELLEGRINO, G., AND SABELFELD, A. Black Widow: Blackbox Data-driven Web Scanning. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)* (2021).

[15] FANG, Y., HUANG, C., XU, Y., AND LI, Y. RLXSS: Optimizing XSS Detection Model to Defend Against Adversarial Attacks Based on Reinforcement Learning. *Future Internet 11* (2019).

[16] FOLEY, M., AND MAFFEIS, S. HAXSS: Hierarchical Reinforcement Learning for XSS Payload Generation. In *2022 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)* (2022).

[17] HALFOND, W. G. J., AND ORSO, A. AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (2005).

[18] JAN, S., NGUYEN, C. D., ARCURI, A., AND BRIAND, L. A Search-Based Testing Approach for XML Injection Vulnerabilities in Web Applications. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)* (2017).

[19] JAN, S., PANICHELLA, A., ARCURI, A., AND BRIAND, L. Automatic Generation of Tests to Exploit XML Injection Vulnerabilities in Web Applications. *IEEE Transactions on Software Engineering 45* (2019).

[20] KASIM, O. An ensemble classification-based approach to detect attack level of SQL injections. *Journal of Information Security and Applications 59* (2021).

[21] KIEYZUN, A., GUO, P., JAYARAMAN, K., AND ERNST, M. Automatic creation of sql injection and cross-site scripting attacks. In *2009 IEEE 31st International Conference on Software Engineering* (2009).

[22] LE, Q., AND MIKOLOV, T. Distributed Representations of Sentences and Documents. In *Proceedings of the 31st International Conference on Machine Learning* (2014).

[23] LEE, S., WI, S., AND SON, S. Link: Black-Box Detection of Cross-Site Scripting Vulnerabilities Using Reinforcement Learning. In *Proceedings of the ACM Web Conference 2022* (2022).

[24] LEI, X., QU, J., YAO, G., CHEN, J., AND SHEN, X. Design and Implementation of an Automatic Scanning Tool of SQL Injection Vulnerability Based on Web Crawler. In *Security with Intelligent Computing and Big-data Services* (2020).

[25] LIU, E. Z., GUU, K., PASUPAT, P., SHI, T., AND LIANG, P. Reinforcement Learning on Web Interfaces Using Workflow-Guided Exploration. *arXiv:1802.08802 [cs]* (2018).

[26] LIU, M., LI, K., AND CHEN, T. DeepSQLi: Deep Semantic Learning for Testing SQL Injection, 2020.

[27] LLC, S. Arachni - web application security scanner framework, 2014.

[28] LUO, Y. SQLi-Fuzzer: A SQL Injection Vulnerability Discovery Framework Based on Machine Learning. In *2021 IEEE 21st International Conference on Communication Technology (ICCT)* (2021).

[29] MARASHDEH, Z., SUWAIS, K., AND ALIA, M. A Survey on SQL Injection Attack: Detection and Challenges. In *2021 International Conference on Information Technology (ICIT)* (2021).

[30] MNIH, V., BADIA, A. P., MIRZA, M., GRAVES, A., LILLICRAP, T., HARLEY, T., SILVER, D., AND KAVUKCUOGLU, K. Asynchronous Methods for Deep Reinforcement Learning. In *International Conference on Machine Learning* (2016).

[31] MNIH, V., KAVUKCUOGLU, K., SILVER, D., GRAVES, A., ANTONOGLOU, I., WIERSTRA, D., AND RIEDMILLER, M. Playing Atari with Deep Reinforcement Learning. *arXiv:1312.5602 [cs]* (2013).

[32] NAGY, C., AND CLEVE, A. A Static Code Smell Detector for SQL Queries Embedded in Java Code. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2017).

[33] NODISCC. Awesome-Selfhosted, 2022.

[34] NTAGWABIRA, L., AND KANG, S. Use of query tokenization to detect and prevent sql injection attacks. In *2010 3rd International Conference on Computer Science and Information Technology* (2010), vol. 2, pp. 438–440.

[35] OWASP. OWASP Top 10 Web Application Security Risks, 2021.

[36] OWASP. OWASP sql injection prevention cheat sheet, 2022.

[37] OWASP. SQL Injection | OWASP Foundation, 2022.

[38] OWASP. Testing for sql injection, 2022.

[39] PIETRASZEK, T., AND BERGHE, C. V. Defending Against Injection Attacks Through Context-Sensitive String Evaluation. In *Recent Advances in Intrusion Detection* (2006).

[40] PORTSWIGGER. What is SQL Injection? Tutorial & Examples | Web Security Academy, 2022.

[41] QI, J., ZHOU, Q., LEI, L., AND ZHENG, K. Federated Reinforcement Learning: Techniques, Applications, and Open Challenges, 2021.

[42] SABLOTNY, M., JENSEN, B. S., AND JOHNSON, C. W. Recurrent Neural Networks for Fuzz Testing Web Browsers. In *Information Security and Cryptology* (2019).

[43] SCHULMAN, J., WOLSKI, F., DHARIWAL, P., RADFORD, A., AND KLIMOV, O. Proximal Policy Optimization Algorithms. *arXiv:1707.06347 [cs]* (2017).

[44] SENNRICH, R., HADDOW, B., AND BIRCH, A. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (2016).

[45] SHAR, L. K., AND TAN, H. B. K. Defeating SQL Injection. *Computer 46* (2013).

[46] STASINOPOULOS, A., NTANTOGIAN, C., AND XENAKIS, C. Commix: automating evaluation and exploitation of command injection vulnerabilities in Web applications. *International Journal of Information Security 18* (2019).

[47] SU, Z., AND WASSERMANN, G. The essence of command injection attacks in web applications. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2006).

[48] SUN, S.-T., AND BEZNOSOV, K. Retrofitting Existing Web Applications with Effective Dynamic Protection Against SQL Injection Attacks. *International Journal of Secure Software Engineering (IJSSE) 1* (2010).

[49] SUTTON, R. S., AND BARTO, A. G. *Reinforcement learning: an introduction*, second edition ed. Adaptive computation and machine learning series. 2018.

[50] SWISSKYREPO. Payloads all the things, 2022.

[51] TASDELEN, I. payloadbox/sql-injection-payload-list, 2022.

[52] TOUSEEF, P., ALAM, K., JAMIL, A., TAUSEEF, H., AJMAL, S., ASIF, R., REHMAN, B., AND MUSTAFA, S. Analysis of Automated Web Application Security Vulnerabilities Testing. In *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems* (2019).

[53] V. MNIH ET AL. Human-level control through deep reinforcement learning. *Nature 518* (2015).

[54] VALEUR, F., MUTZ, D., AND VIGNA, G. A Learning-Based Approach to the Detection of SQL Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (2005).

[55] WANG, X., AND HU, H. Evading Web Application Firewalls with Reinforcement Learning.

[56] WORDPRESS. Source code of `WordPress/wp-includes/meta.php`, 2023.

[57] ZHANG, L., ZHANG, D., WANG, C., ZHAO, J., AND ZHANG, Z. ART4SQLi: The ART of SQL Injection Vulnerability Discovery. *IEEE Transactions on Reliability 68* (2019).

[58] ZHAO, J., DONG, T., CHENG, Y., AND WANG, Y. CMM: A Combination-Based Mutation Method for SQL Injection. In *Structured Object-Oriented Formal Language and Method* (2020).

## A Appendix

Table 7: Settings used for scanners when conducting experiments. Note these are for crawling and vulnerability identification. In Section 5.3 we also test without crawling, pointing the scanners to each specific vulnerability found by SQIRL.

*Arachni:*
```
./bin/arachni [url] --check=sqli
--browser-cluster-pool-size=2
--http-cookie-string='Path=[cookie_file]'
```

*BurpSuite:* We used the following settings:

1. Default crawling settings
2. Set the authentication using active cookie
3. Audit using all SQL injection options

*Sqlmap:*
```
sqlmap -u [url] --cookie=[cookie] --crawl 10
--batch --forms
```

*Wapiti:*
```
./bin/wapiti -u [url] -c [cookie_file] -m sql
```

*ZAP:* We used the following settings:

1. Traditional spider
2. AJAX spider with headless chrome
3. Context with the relevant authentication
4. Active scans using SQL injection option, using the built-in/advanced plugin

Table 8: List of Production Grade Web Applications

1. WordPress core v6.0 and plugins (Download Monitor WordPress V 4.4.4, WP User Frontend 3.5.25, Sliced Invoices 3.8.2, Plugin Photo Gallery 1.5.34, Supsystic Ultimate Maps 1.1.12, WP Statistics 13.0.7, JoomSport)
2. B2evolution v7.2.3-stable
3. BBpress v2.6.9
4. Big tree CMS v4.4.16
5. Drupal v9.3.18
6. Joomla v4.2.0
7. Admidio v4.0
8. Gila CMS
9. Media wiki v1.38.2
10. Pbboard v3.0.3
11. Impresscms v1.4.4
12. WackoWiki v6.0.31
13. Sourcecodester E-learning System v1.0,
14. Sparks Hotel Management System v1.0

Table 9: Hyperparameters used to train the components of SQIRL. We performed a grid search on these values to optimise each model separately.

| Model | Value | Min | Max | Step | Range | Selected |
|---|---|---|---|---|---|---|
| DQN | ε decay | 0.999 | 0.9999 | 0.0001 | - | 0.9999 |
| | ε minimum | 0.1 | 0.5 | 0.1 | - | 0.2 |
| | ε start | 0.9 | 0.5 | 0.1 | - | 0.7 |
| | Intrinisc reward limit | - | - | - | [0.1,0.3,0.7, 0.9] | 0.7 |
| | Batch size | 124 | 2,048 | $\times 2$ | - | 512 |
| | learning rate | $10^{-4}$ | $60^{-4}$ | $10^{-4}$ | - | $50^{-4}$ |
| | Q-policy update | - | - | - | [10, 50, 100, 200, 400] | 200 |
| | Network Archecture | - | - | - | (1024-512), (2048-1024-512), (4096-2048-1024-512) | (2048-1024-512) |
| Autoencoder | learning rate | $10^{-4}$ | $90^{-4}$ | $10^{-4}$ | - | $10^{-4}$ |
| | Fixed feature size | - | - | - | [10,000,1024,5012] | 1,024 |
| | Batch size | 124 | 2048 | $\times 2$ | - | 512 |
| Federated RL agent | Client Aggregation Rate | - | - | - | [10,20,40,80] | 20 |

Table 10: SQL statements and sanitisations used in the SMB where tasks increase in difficulty. Tasks 1-20 are used for training SQIRL. Tasks 1-30 constitute the positive class for the SMB benchmark.

| Task | SQL Statement | Sanitisation |
|---|---|---|
| 1 | SELECT * FROM users WHERE name=INPUT | - |
| 2 | SELECT * FROM users WHERE name='INPUT' | - |
| 3 | UPDATE users SET pass ='pss' WHERE (name='INPUT') | - |
| 4 | INSERT INTO `users` (`name`,`pass`) VALUES ('INPUT','INPUT') | - |
| 5 | SELECT * FROM users WHERE name='INPUT' LIMIT 1 | - |
| 6 | SELECT * FROM users WHERE name='INPUT' group by `user` | - |
| 7 | SELECT * FROM users WHERE name="INPUT" | - |
| 8 | SELECT count(name) FROM users group by `INPUT` | - |
| 9 | SELECT count(name) FROM users group by INPUT | - |
| 10 | SELECT count(name) FROM users group by ('INPUT') | - |
| 11 | SELECT * FROM users WHERE name=(INPUT) | - |
| 12 | SELECT * FROM users WHERE name=`INPUT` group by ID | - |
| 13 | SELECT MIN(name) from users GROUP BY id HAVING id=INPUT | - |
| 14 | SELECT * FROM users WHERE id = INPUT | MySQLi real_escape_string |
| 15 | SELECT * FROM users WHERE id = INPUT | Filter capitalised and lowercase SQL keywords |
| 16 | SELECT * FROM users WHERE id = ('INPUT') | Filter capitalised and lowercase SQL keywords |
| 17 | SELECT * FROM users WHERE id = ('INPUT') | Filter out spaces |
| 18 | SELECT * FROM users WHERE id = ('INPUT') | Escape all AND keywords |
| 19 | SELECT * FROM users WHERE id LIKE 'INPUT' | Filter capitalised and lowercase SQL keywords |
| 20 | SELECT * FROM users WHERE name='INPUT1' OR name='INPUT2' OR name='INPUT3' LIMIT 0, 1 | MySQLi real_escape_string on input1 and input3 |
| 21 | UPDATE users SET name='name WHERE id = INPUT | CVE-2020-8637 |
| 22 | UPDATE users SET name='INPUT1' WHERE id = INPUT2 | CVE-2020-8638 |
| 23 | SELECT * FROM users WHERE name LIKE 'INPUT' | CVE-2023-30605 |
| 24 | SELECT * FROM users WHERE name=INPUT | CVE-2023-24812 |
| 25 | SELECT * FROM users WHERE (id=INPUT1 AND name='INPUT2') | CVE-2020-8841 |
| 26 | SELECT * FROM users WHERE id='INPUT' LIMIT 0, 1 | Remove all inputs containing AND |
| 27 | SELECT MIN(name) from users GROUP BY id HAVING id=('INPUT') | Filter capitalised and lowercase SQL keywords |
| 28 | INSERT INTO `users` (`name`,`pass`) VALUES ('INPUT','INPUT') | Filter out spaces |
| 29 | SELECT * FROM users WHERE id LIKE "%INPUT%" LIMIT 0, 1 | Escape all AND keywords |
| 30 | SELECT * FROM users WHERE id LIKE (("%INPUT%")) LIMIT 0, 1 | Escape all AND keywords |

Table 11: Payload types and the scanners that can create them. We replace integers, strings, and base64 with `INT`, `TEXT`, and `BASE64`; otherwise leaving the payloads the same. These payloads are gathered from the experiments in Section 5. **F** denotes use of feedback, **NF** denotes no feedback, and **E** denotes presence of exceptions.

| | ZAP | | | Sqlmap | | BurpSuite | | Arachni | Wapiti | SQIRL |
|---|---|---|---|---|---|---|---|---|---|---|
| Payload Type | Built-in **F** | Adv. **F** | Adv. **NF** | **F** | **NF** | **F** | **NF** | **E** | **E** | |
| TEXT' OR '1'='1' -- | ✓ | | | | | | | | | |
| TEXT%" -- | ✓ | | | | | | | | | |
| TEXT') UNION ALL SELECT CONCAT(HEX,HEX,HEX),NULL,NULL# | | ✓ | | | | | | | | |
| (SELECT * FROM (SELECT(SLEEP(5)))TEXT) | | ✓ | ✓ | | | | | | | |
| TEXT') AND (SELECT * FROM (SELECT(SLEEP(5)))TEXT) AND ('TEXT'='TEXT | | ✓ | ✓ | | | | | | | |
| TEXT') TEXT (SELECT (CASE WHEN (INT=INT) THEN HEX ELSE 0x28 END)) AND ('TEXT'='TEXT') | | ✓ | | ✓ | ✓ | | | | | |
| TEXT' RLIKE (SELECT * FROM (SELECT(SLEEP(5)))TEXT) AND 'TEXT'='TEXT | | ✓ | ✓ | ✓ | ✓ | | | | | |
| (SELECT * FROM (SELECT(SLEEP(5)))INT) | | | ✓ | | | | | | | |
| TEXT AND (SELECT * FROM (SELECT(SLEEP(5)))TEXT) | | | | | ✓ | | | | | |
| TEXT' UNION ALL SELECT NULL,CONCAT(HEX,HEX,HEX),NULL-- | | | | ✓ | | | | | | |
| id=INT') UNION ALL SELECT CONCAT(CONCAT('TEXT','TEXT'),'TEXT'),NULL,NULL- TEXT | | | | ✓ | | | | | | |
| (SELECT (CASE WHEN (INT=INT) THEN INT ELSE (SELECT INT UNION SELECT INT) END | | | | ✓ | | | | | | |
| '+(select*from(select(sleep(20)))TEXT)+' | | | | | | ✓ | ✓ | | | |
| (select*from(select(sleep(20)))TEXT) | | | | | | ✓ | ✓ | | | |
| \' | | | | | | ✓ | | | | |
| INT or INT=INT | | | | | | ✓ | | | | |
| and (select*from(select(sleep(20)))TEXT)-- | | | | | | ✓ | | | | |
| INT' or INT=INT-- and INT' or INT=INT-- | | | | | | ✓ | | | | |
| INT' or 'INT'='INT and INT' or 'INT'='INT | | | | | | ✓ | | | | |
| ' and INT=INT-- and ' and INT=INT-- | | | | | | ✓ | | | | |
| TEXT\"'`-- | | | | | | | | ✓ | | |
| INT!%TEXT\"'`-- | | | | | | | | ✓ | | |
| INT'=sleep(16)=' | | | | | | | | ✓ | | |
| INT' and sleep(16)=' | | | | | | | | ✓ | | |
| 0 or sleep(16) # | | | | | | | | ✓ | | |
| INT¿'"( | | | | | | | | | ✓ | |
| AND INT=INT AND INT=INT | | | | | | | | | ✓ | |
| BASE64 AND SLEEP (0.0) | | | | | | | | | | ✓ |
| BASE64" And SLEEP (0.0)' AND SLEEP (0.0)# | | | | | | | | | | ✓ |
| BASE64)' anD SLEEP (0.0))# | | | | | | | | | | ✓ |
| BASE64' aND slEEp (0.0)# | | | | | | | | | | ✓ |
| BASE64'/**/&/**//**/SLEEP/**/(0.0))# | | | | | | | | | | ✓ |
| BASE64/**/AnD/**//**/SLEEP/**/(0.0)"/**/AND/**//**/SLEEP/**/(0.0)# | | | | | | | | | | ✓ |
| BASE64 and sleep(0.0) | | | | | | | | | | ✓ |
| BASE64' & SLEEP (0.0)# | | | | | | | | | | ✓ |
| BASE64' AND SLEEP (0.0)-- -- | | | | | | | | | | ✓ |
| BASE64'# | | | | | | | | | | ✓ |

Table 12: Context Free Grammar used to generate MySQL statements. Blue indicates terminals.

```
root: (select_stmt | update_stmt)
select_stmt: ('SELECT', selectable, (where_cond | having_cond), (order_by | join_stmt), (connector | union_stmt), limit_stmt)
union_stmt: ('UNION', select_stmt | '')
update_stmt: ('UPDATE', selectable, 'SET', where_cond, join_stmt)
where_cond: ('WHERE', var, bool_op, var)
having_cond: ('HAVING', var, bool_op, var)
order_by: ('ORDER BY', var)
join_stmt: ('JOIN', join_op, var)
join_op: ('CROSS JOIN' | 'FULL OUTER JOIN' | 'HASH JOIN' | 'INNER JOIN' | 'LEFT JOIN' | 'LEFT OUTER JOIN' | 'OUTER JOIN' | 'RIGHT
      JOIN' | 'RIGHT OUTER JOIN')
connector: ((' AND',  (select_stmt | update_stmt))| (' OR', (select_stmt | update_stmt))| ('
      XOR', (select_stmt | updata_stmt)) | '')
selectable: '*' | 'COUNT(', var, ')' | 'ABS(', var, ')' | 'AVG(', var, ')' | 'FLOOR(', var ')' | 'MAX(', var, ')' |  'BIN(',
      var, ')' | 'MIN(', var, ')' | 'STD(', var, ')' | 'STDDEC(', var, ')' )
bool_op: ( '==' |'!=)' |  '<)' | '>)' | '=<)' | '>=)' | 'IN' | 'BETWEEN')
limit_stmt: ('LIMIT', INT | '')
var: (INT |  STRING)
```