RAT: Reinforcement-Learning-Driven and Adaptive Testing for Vulnerability Discovery in Web Application Firewalls

Mohammadhossein Amouei, Mohsen Rezvani, Mansoor Fateh

Abstract—Due to the increasing sophistication of web attacks, Web Application Firewalls (WAFs) have to be tested and updated regularly to resist the relentless flow of web attacks. In practice, using a brute-force attack to discover vulnerabilities is infeasible due to the wide variety of attack patterns. Thus, various black-box testing techniques have been proposed in the literature. However, these techniques suffer from low efficiency. This paper presents Reinforcement-Learning-Driven and Adaptive Testing (*RAT*), an automated black-box testing strategy to discover injection vulnerabilities in WAFs. In particular, we focus on SQL injection and Cross-site Scripting, which have been among the top ten vulnerabilities over the past decade. More specifically, *RAT* clusters similar attack samples together. It then utilizes a reinforcement learning technique combined with a novel adaptive search algorithm to discover almost all bypassing attack patterns efficiently. We compare *RAT* with three state-of-the-art methods considering their objectives. The experiments show that *RAT* performs 33.53% and 63.16% on average better than its counterparts in discovering the most possible bypassing payloads and reducing the number of attempts before finding the first bypassing payload when testing well-configured WAFs, respectively.

Index Terms—Security testing, injection attack, adaptive testing, web application firewall (WAF), test case clustering.

✦

1 INTRODUCTION

IN recent decades, most traditional brick and mortar businesses have transformed into online ones, such as online shopping, e-banking, social media, etc. Thus, an enormous amount of private data of individuals and organizations is stored in web applications databases, making them tempting targets for attackers. A recent report reveals that web applications may experience up to 26 attacks per minute [1]. Moreover, according to Symantec's security report, 76% of websites are vulnerable to several attacks [2].

A proper way to provide the security is to use Web Application Firewalls (WAFs) which analyze HTTP(S) traffic to prevent malicious requests from reaching the web applications. The Open Web Application Security Project (OWASP[1]) defines WAF as "a security solution on the web application level which - from a technical point of view - does not depend on the application itself." [3]. To put it in perspective, WAFs intercept bi-directional HTTP(S) traffic, analyze it, and decide whether it is malicious or benign. Common rule-based WAFs use a set of rules to make a decision. For instance, WAFs utilize regular expressions to detect SQL injection (SQLi) attacks. Moreover, recently, extensive research has been done on intelligent Machine-Learning-Based (ML-Based) WAFs to distinguish between malicious and benign traffic with machine learning algorithms [4–6].

A recent study confirmed that web attacks had grown in size and sophistication [7]; consequently, it is crucial to regularly test and maintain WAFs to keep them secure and efficient. In Fig. 1, the procedure for testing a WAF is illustrated. As web attacks become more sophisticated, traditional WAF rules become more complex, and ML-Based WAFs tend to learn novel attacks. At the same time, manual testing and maintenance become more arduous tasks. Testing WAFs is also extremely expensive, especially in

Fig. 1: Testing procedure for a WAF to discover vulnerabilities.

time, due to the massive variety of attacks. Hence, optimal automated testing is essential for WAFs to protect web applications and services efficiently.

One of the common and destructive categories of attacks is injection. Injections are attacks in which the attacker injects a malicious input to a web application. Then, the application interprets this input as a part of a command or a query, which can result in severe damages. In this paper, we focus our tests on two common injection attacks: SQLi and Cross-site Scripting (XSS). These attacks are reported as top 10 vulnerabilities [8] and have attracted a lot of attention[1, 7, 9–19].

There are various types of security testing proposed in the literature, such as white-box testing, model-based testing, and black-box testing. However, these techniques suffer from limitations that can affect their practical applicability. White-box testing needs access to the applications' source code, which might not be possible in testing industrial applications. Moreover, each white-box testing tool supports only some specific programming languages; thus, they can only test applications developed with those programming

- M. Amouei, M. Rezvani and M. Fateh are with the Faculty of Computer Engineering, Shahrood University of Technology, Iran.
  E-mail: {mhamooei,mrezvani,mansoor_fateh}@shahroodut.ac.ir

[1]https://owasp.org

languages. Model-based testing techniques require a model representing the security policies, which is difficult to create and is often unavailable. The black-box testing, nevertheless, does not have the mentioned limitations of the two other techniques [18]. However, despite the remarkable effort that has been devoted to black-box testing, studies indicate that black-box testing is inefficient, and a large number of vulnerabilities remains undiscovered [20, 21].

In recent years, with the power of artificial intelligence, novel black-box testing techniques have shown a significant improvement in efficiency and effectiveness [16–18]. These techniques utilize artificial intelligence methods, such as evolutionary algorithms [12–14] or machine learning techniques [15, 17, 22] to improve the black-box testing performance. However, these techniques still consume many requests, and yet many vulnerabilities remain undetected. Therefore, black-box testing requires further improvement.

This research aims to design a practical automated black-box security testing approach to uncover WAFs' vulnerabilities efficiently. Thus, in this paper, we propose a method called *RAT*, which uses machine learning algorithms to provide better effectiveness and efficiency to the black-box testing. *RAT* first tokenizes attack payloads using n-gram. It then clusters similar attack payloads and uses a reinforcement learning technique called decayed ϵ-greedy policy combined with a novel adaptive search technique to find its way through the testing jungle.

Furthermore, we compare *RAT* with three state-of-the-art techniques, including *Ml-Driven E* [18], *ART4SQLi* [7] and *XSSART* [19]. These tests are designed considering our counterparts' objectives. More specifically, *Ml-Driven E* tend to discover the highest possible number of SQLi vulnerabilities, and *ART4SQLi* and *XSSART* aim to find the very first bypassing payload with the lowest number of requests. Thus, to compare *RAT* with *Ml-Driven E* we measure the total bypassing payloads within a limited number of requests for both techniques, and for the comparison between *RAT* and *ART4SQLi* and *XSSART*, we compare the number of blocked payloads before finding the first bypassing payload for all three techniques. We use SQLi dataset to compare *RAT* with *Ml-Driven E* and *ART4SQLi*, and XSS dataset to compare *RAT* with *XSSART*. We also compare *RAT* with a simple random testing technique (we name it *Random Fuzzer*) as a basic method. Our comparative experiments show that *RAT* can discover an average of 33.53% more bypassing attack than *Ml-Driven E* within a limited number of requests. Moreover, according to our experiments, our adaptive search algorithm is an average of 61.43% faster than *ART4SQLi* when facing well-configured WAFs. However, *ART4SQLi* could discover the first bypassing payload about 38.70% faster than *RAT* in testing a WAF with massive vulnerabilities. Moreover, results show that *RAT* is an average of 64.90% faster than *XSSART* in finding the first bypassing payload.

The main contributions of the paper summarized as:

1) Since string-based injection attack payloads are sequences of specific string tokens, we employ n-gram, known for simplicity and scalability [23], as a feature extraction method. In our experiments, we observed that n-gram could model sophisticated attack patterns while extracting significantly fewer features than the *ML-Driven E*, resulting in a better efficiency than *ML-Driven E*.
2) We evaluate the effects of the clustering and propose a method to cluster the similar attack payloads.
3) We use the ϵ-greedy algorithm and a novel adaptive search technique to enhance the efficiency of our black-box testing approach.

The remaining sections are organized as follows. Section 2 describes SQLi and XSS attacks as well as previous

```java
public void doPost(HttpServletRequest
    request, HttpServletResponse response)
throws ServletException, IOException {

  String user_name =
      request.getParameter("username");
  String user_pass =
      request.getParameter("password");

  String sql_statement = "SELECT * FROM
      users WHERE username = '"
  + user_name
  + "' AND password = '"
  + user_pass + "' ";

  result = Database.execute(sql_statement)
}
```

Fig. 2: Example of an unsafe SQL statement formation in Java.

research. Section 3 details the proposed approach. In Section 4, we explain research questions as well as the experimental environment. Section 5 discusses the experiments and evaluation results. Section 6 concludes this paper.

2 BACKGROUND AND RELATED WORK

This section aims to provide a brief description of the code injection attacks, such as SQL Injection and Cross-Site Scripting. In the final subsection, we narrate the research story of black-box testing for these two web attacks.

2.1 SQL Injection

SQL databases, also known as relational, are the most popular ones among developers [24]. Web-based applications that use SQL databases communicate with database engines by statements that are defined in a language, named SQL. These statements are usually formed dynamically by concatenating various substrings. Each substring is provided either by a user or the application itself. Once statement formation has done, the database engine executes it.

Fig. 2 shows an instance of an unsafe SQL statement formation. In Fig. 2, the variables user_name and user_pass are provided by the user through HTTP request (line 4-5). Then, they are concatenated with the SQL statement without any sanitization and stored in the variable sql_statement (line 7-10). Finally, the sql_statement passed to the function execute of the class Database to be executed by the database server (line 12). In case of malicious inputs, database server executes the infected statement and performs the attacker's desired action.

SQLi is an injection attack in which the attacker targets applications with unsafe SQL statement formation, and passes malicious input parameters to the victim application and misleads the application to perform an unauthorized action (e.g., accessing confidential data without granting required permission). For instance, in Fig. 2, if instead of passing a real username to the variable user_name, an attacker fills the username parameter with ` OR 1 = 1 # and leaves the password empty, then the value of the sql_statement would be SELECT * FROM users WHERE username = `` OR 1 = 1 # AND password = `` which is equal to SELECT * FROM users WHERE username = `` OR 1 = 1. In SQL, the symbol # is an inline comment operator, and 1 = 1 is an always True condition, thus if the database executes the resulting statement, it returns all rows of the table users.

SQLi attacks are a high-risk security threat to organizations. By a successful SQLi exploit, confidential data of organizations or individuals can be modified or stolen. Thus, protecting applications against SQLi attacks is vital.

Fig. 3: Example of an XSS attack attempt, in which the attacker steals the victim's cookie by submitting a malicious post to a vulnerable social media.

```html
1  <!DOCTYPE html>
2  <html>
3  <h1> Latest Posts </h1>
4  ...
5  <script>
6    $(document).ready(function() {
7        var cookie = document.cookie;
8        $.post("http://test.com/cookie",
9        {cookie: cookie},
10       function(result) {});
11   });
12 </script>
13 ...
14 </html>
```

Fig. 4: Example of a malicious HTML containing JavaScript to steal user's cookie.

2.2 Cross-Site Scripting (XSS)

XSS is a code injection attack in which the attacker injects malicious scripts into legitimate websites to execute them in the web browser of end-users. Any web application that generates output using input from users can be vulnerable to such an attack. XSS occurs when a victim visits an infected web application that carries the malicious script to the browser. Within the browser, the script can access user's sensitive information (e.g., stealing costumer's payment info), cookies and, any other data that related to the web application and retained by the user's browser.

Fig. 3 shows an instance of an XSS attack attempt in which the attacker tries to steal the user's cookie of a vulnerable social media. The details are as follows:

1) The attacker submits a malicious post containing JavaScript code to the application, which inserts it into the application's database without sanitization.
2) The victim user sends a GET request to the application to get the latest posts. Afterward, the application retrieves the posts, including the malicious one, from its database and locates them into the HTML page as the response to the client.
3) The application responds to the user with a page containing the malicious script (Figure 4).
4) Once the user's browser receives the page, render it and executes its scripts, and as a result, the JavaScript code in Figure 4 (line 5-12) sends the user's cookie to the attacker's server.

2.3 Related Work

Over the past decade, both SQLi and XSS attacks have been attractive topics for researchers, and remarkable research efforts have been made toward vulnerability detection methods, particularly automated black-box testing [25].

Various combinatorial testing methods have been proposed in the literature for both XSS and SQLi attacks [1, 9–11]. These methods parametrize attack patterns in the form of BNF grammar rules. Then they test combinations of parameters by covering t-way interactions in which t is the number of parameters, and a higher value for t can produce more complex patterns. However, a large t consumes a vast number of HTTP requests.

Knowing the capability of Artificial Intelligence (AI), researchers tend to provide better-optimized solutions using AI for various problems, including security assessment. For example, Thomé et al. [12] proposed a fitness function to measure how close an SQLi literal is from generating a bypassing payload. Avancini and Ceccato [13] used the Genetic Algorithm (GA) to find vulnerable inputs in webpage. Moreover, Duchene et al. [14] applied GA to generate XSS payloads in fuzz testing. There are also adversarial and learning-based methods which we describe in the following.

2.3.1 Adversarial

Elderman et al. [22] simulated an adversarial cybersecurity game in which an attacker and a defender are two adversarial agents that use reinforcement learning techniques to win the game. Demetrio et al. [15] proposed *WAF-A-MoLE*, an adversarial method for mutating attack strings to bypass ML-Based WAFs. Mostly, adversarial methods aim to bypass ML-Based WAFs, whereas our method targets signature-based WAFs. Nevertheless, our approach can be combined with adversarial techniques to achieve higher performance. For instance, it can be integrated into *WAF-A-MoLE* to function as a guide to increase its efficiency.

2.3.2 Learning-based

Tripp et al. [16] suggested *XSS Analyzer*, a web security testing approach with learning capability. The authors proposed a method that learns from previous attempts to select the next payload with a higher probability of exposing a vulnerability in the System Under Test (SUT). In particular, *XSS Analyzer* generates payloads for a grammar, devised based on a comprehensive dataset of XSS payloads, and learns which tokens are preventing an attack from evading the security.

Inspired by *XSS Analyzer*, Appelt et al. [17, 18] proposed *ML-Driven B*, *ML-Driven D* and later on, *ML-Driven E*, learning approaches to SQLi vulnerability detection. The same as *XSS Analyzer*, *ML-Drivens* benefit from the idea of learning literals that prevent an attack from bypassing the security from previous attempts. The main difference is within the learning process. Whereas *XSS Analyzer* only learns individual literals, *ML-Drivens* learn the combinations of literals. More specifically, *ML-Drivens* split each attack's derivation tree into subtrees and then measures the likelihood of bypassing attacks using a decision tree. The authors prove that *ML-Drivens* can learn more complex attack patterns and outperform state-of-the-art tools, thus, suggesting that learning combinations of literals can effectively guide the testing procedure.

Usually, bypassing payloads are rare in a comprehensive payload collection which makes it challenging to find effective payloads with a reasonable number of tests. Zhang et al. [7] proposed *ART4SQLi*, an adaptive random testing approach for SQLi vulnerability and Lv et al. [19] proposed *XSSART* an adaptive random testing approach for XSS vulnerability detection. Both *ART4SQLi* and *XSSART* use similarity metrics to expedite the process of finding an effective attack payload, and demonstrate that payload spaces are sparse, and bypassing payloads tend to cluster together. Thus, in this research, we consider clustering similar payloads and evaluate the effects of clustering on efficiency.

Although *ML-Drivens* can discover a large number of bypassing payloads, they require a large number of observations to learn effective patterns. *ML-Drivens* require

Fig. 5: Overview of the first phase of the proposed approach, which shows the preparation of the test oracle.

an initial collection of both passed and blocked attacks to train a decision tree; thus, before training the decision tree, they perform a random search which, facing a well-configured WAF, consumes a large number of requests. In contrast to *ML-Drivens*, our adaptive search technique only uses blocked attacks to uncover bypassing ones in the very beginning of the test. It then uses the discovered bypassing attacks to improve its performance.

Moreover, *ML-Drivens* break payload derivation trees into sub-trees to use them as features, creating a vast feature space that exponentially expands if we try to test longer attack payloads (e.g., XSS). Training a decision tree with this massive number of features is non-practical; thus, the authors suggested selecting a small random subset of the feature space, resulting in low efficiency of their approach. In comparison, RAT showed a better performance in our experiments, since first of all, n-gram extracts significantly fewer features than *ML-Drivens* feature extraction algorithm. Secondly, our clustering and feature reduction technique can effectively reduce the number of features, resulting in better effectiveness and efficiency.

Last but not least, the decision tree algorithm suffers from the local minima problem, and it is unstable as small changes in training data remarkably affect classification results [26]. Thus, the authors in [18] discussed using an ensemble model such as a random forest algorithm may address these problems. However, it adds high computational overhead as multiple decision trees have to be trained during each update, and their experiments showed that its improvement is not significant enough. Similar to *ML-Drivens*, in our work, the ϵ-greedy policy suffers from local optima, and it is unstable because the cluster selection order can significantly change the final results. Nevertheless, we try to avoid local optima by controlling exploration and exploitation rate using decayed ϵ-greedy policy, which does not add any computational burden. More specifically, this policy starts with a big ϵ to explore and find clusters that include bypassing attacks. Since our adaptive search algorithm can quickly find bypassing payloads inside clusters, decayed ϵ-greedy can quickly filter the clusters with non-bypassing payloads. Then, the ϵ decreases, so the *RAT* focuses on the clusters with bypassing attacks. These quick exploration and exploitation significantly improve *RAT*'s stability and efficiency. Our experiments show that the *RAT* is considerably more stable and efficient than *ML-Driven E.*

3 APPROACH

In this section, we first describe an overview of the proposed approach. We then explain the details of each module in next subsections.

Fig. 6: Decomposition of a sample attack payload to its constitutive tokens.

3.1 Framework Overview

The code injection attack payload is formed by concatenating miscellaneous string fragments. Considering these fragments are the test parameters, each fragment is responsible for either failure or success of an attack payload in circumventing the firewall. In black-box testing, we do not have any information about the source code of the application under the test to distinguish effective from ineffective fragments. Nevertheless, it is possible to find effective fragments based on feedback obtaining from the application during the test process. However, due to the large variety of fragments in a rich dataset, a testing tool needs too many observations to gather enough information. Therefore we consider clustering similar payloads together; thus, we can reduce the variety of fragments to distinguish between them rapidly. Moreover, since bypassing payloads tend to cluster together, devoting search effort to effective clusters significantly improves the performance.

Fig. 5 illustrates an overview of *RAT*. It shows the process of preparing attack samples for the testing phase. Here, `Dataset` is the collection of attack payloads (see section 4.3.2 for more details about the datasets). At the very first step, n-gram `Tokenizer` (Section 3.2.1) tokenizes the attack samples of the dataset. Then, the `Word Embedding` (Section 3.2.2) module maps each token to the vector of real numbers, and `Hierarchical Clustering` (Section 3.2.3) clusters tokens using these vectors. `Binary Encoder` (Section 3.2.4) then forms a binary feature vector using clusters that are obtained from `Hierarchical Clustering` module, and passes these vectors to `Deep Embedding Module (DEN)` (Section 3.2.5) to cluster attack samples. Finally, in the feature extraction phase, `Token Selector` (Section 3.3.1) selects only effective tokens for each cluster, and then, `IDF Calculator` (Section 3.3.2) forms the main feature vector for each attack payload. Clusters, attack samples, and their feature vectors are then passed to the `Test Oracle` (Section 3.4) to be used in our testing algorithm.

3.2 Clustering Payloads

At the very first stage of our approach, we decompose attack payloads into string fragments and then cluster them together based on their common fragments. In other words, we split the dataset into smaller datasets; thus, later, we can search in each mini dataset independently. The following sections detail the process, and our experiments show that clustering significantly reduces the number of features. This feature reduction is important as a high number of features require many observations to learn patterns; thus, removing irrelevant features improves accuracy and efficiency - reducing features decreases the computational complexity and working with small feature vectors requires fewer resources (e.g. disk storage or memory) than big feature vectors [27].

3.2.1 n-gram Tokenizer

We can decompose every payload of a code injection attack into smaller pieces called tokens. Each token can be a single character or a sequence of characters. As an example, in SQLi, constitutive tokens of the sample payload showed in Fig. 6. Simply, each token is responsible for either failure or

TABLE 1: Example of payload decomposition for the payload `0)␣or␣not␣0>(␣!␣␣0)--` using n-gram.

Unigram ($n = 1$)	Bigram ($n = 2$)	Trigram ($n = 3$)
`0` , `)` , `␣` , ...	`0)` , `)␣` , `␣or` , ...	`0)␣` , `)␣or` , `␣or␣` , ...

Fig. 7: Architectures of Word2Vec models: Continuous Bag of Words (CBOW) and Skip-gram.

Fig. 8: Example of learning numerical vector representation for n-grams of an SQLi attack payload using the skip-gram model.

success of an attack payload; however, not always a single token but also a combination of tokens can lead a payload to success. Therefore, we extract combinations of tokens instead of single tokens using a well-known method called n-gram. In this paper, we refer to n-gram extracted tokens as fragments.

N-gram is a contiguous sequence of N tokens of a string. We use n-gram to decompose attack payloads into smaller fragments, which help us consider more complex patterns in our search strategy. Table 1 shows an example of payload decomposition for the payload `0)␣or␣not␣0>(␣!␣␣0)--` with three different values for N. As shown in Table 1, with the bigger N, we can extract more complex patterns; however, as we increase the size of N, the variety of fragments grows. In our empirical study, we investigate the effect of N on search results for each dataset.

3.2.2 Word Embedding
In code injection attacks different string fragments can be used interchangeably (i.e., in SQLi, `1=1` is an alternative

TABLE 2: Example of a binary representation for a payload p, which shows whether the corresponding payload contains any fragment from the cluster C_i where i is the cluster's number.

payload	C_1	C_2	C_3	C_4	C_5	C_6
p	1	0	1	1	0	0

for `"a"="a"`). In order to cluster similar attack payloads, we can consider alternative fragments as a single feature. For this purpose, we use word2vec [28] to learn a vector representation for each string fragment; thus, later, we can measure the similarity between fragments.

Continuous Bag of Words (CBOW) and skip-gram are two architectures of Word2Vec [28]. These two simple neural network models are illustrated in Fig. 7. In both models, the middle layer is where the numerical vector is learnt for each unique word. CBOW learns this vector representation by using the surrounding words to predict the word in the middle. Opposed to CBOW, skip-gram tries to predict the surrounding words given the middle word. In our research, we observed that skip-gram could produce better word representation than CBOW. The problem with CBOW might be that since the target words are the output of the neural network, rare words have to compete with their alternatives that are repeated frequently. Therefore, rare words receive low attention from the model if the dataset is unbalanced. On the other hand, in skip-gram, the target words are the input of the neural network. Thus, rare words will not compete with frequent ones, and the model fairly learns the vector representation for every word. In this research, due to the rarity of fragments in our massive datasets, we use the skip-gram model.

In this research, each dataset is used to train a separate skip-gram model from scratch. Fig. 8 depicts an example of transforming SQLi n-grams into numerical vectors. First, payloads are tokenized using n-gram. Then, a window with a specific size (see Section 4.4.2) moves on the new strings to select sub-strings to train the skip-gram. Finally, skip-gram learns a vector representation for each n-gram token. In this specific example, the token `||` means `OR`, and the token $\sim$ means whitespace; thus, fragments `||`$\sim$ and `OR␣` are semantically the same, and they can be used interchangeably. Therefore, after completing the skip-gram training, these two tokens are expected to have similar vectors.

3.2.3 Hierarchical Clustering
After vectorizing the fragments, we use hierarchical clustering with cosine distance to cluster similar fragments.

Suppose vectors v_1 and v_2 represent fragments f_1 and f_2 respectively, the following equation is the distance calculation between f_1 and f_2.

$$distance(f_1, f_2) = 1 - \frac{v_1.v_2}{\|v_1\|_{l_2} \times \|v_2\|_{l_2}} \quad (1)$$

The output of Eq. (1) is a number in the range $[0, 1]$. For two identical vectors, the calculated distance is 0, and a calculated distance as 1 indicates that two vectors are orthogonal. Table 2 shows the binary vector for p.

3.2.4 Binary Encoder
Given a set of payload collection with clustered fragments, we transform each payload into a binary vector as an input to our clustering algorithm. Each item in our vector represents a cluster that indicates whether a corresponding payload contains any fragment belongs to that cluster. For instance, assume a payload p contains a set of unique fragments $F = \{f_1, f_2, f_3, f_4, f_5\}$ and the set of obtained clusters from the previous step is $C = \{C_1, C_2, C_3, C_4, C_5, C_6\}$ in which $\{f_1, f_3\} \subseteq C_1$, $\{f_2\} \subseteq C_3$ and $\{f_4, f_5\} \subseteq C_4$. In Table 2, the corresponding values for C_2, C_5, and C_6 are set

Fig. 9: Architecture of Deep Embedding Network, we used to cluster our dataset. We feed the k-means algorithm with the output of the encoder.

as 0 indicates that p does not contain any fragment belongs to these clusters.

3.2.5   Deep Embedding Network (DEN)

As the last step of this section, we use the output of Binary Encoder to cluster attack payloads. The output of the Binary Encoder is a complex binary vector in which the items are respective to each other. Although numerous similarity measures have been proposed for binary features [29], we found that clustering our high-dimensional raw data using these similarity measures results in poor performance. Thus, we considered using feature transformation methods to map our raw data to a much distinguishable feature space.

Mainly, data transformation methods include linear transformation such as Principal component analysis (PCA) [30] and non-linear transformation such as kernel methods [31]. In recent years, the development of deep learning has facilitated the non-linear transformation of raw features into more clustering-friendly representation [32]. Consequently, abundant deep learning-based clustering methods have been proposed in the literature [33–36].

In this research, we use DEN [36], which first, utilizes a deep autoencoder to learn lower-dimensional representation from the input data. Then, it uses k-means to cluster learned features. In the following paragraphs, we detail the process.

Autoencoder is a kind of unsupervised neural network consists of two parts: encoder function $h = f(\hat{x})$ and decoder function $r = g(h)$. The encoder maps raw data into a latent representation, and then the decoder reconstructs the raw input data from latent features (see Fig. 9). Simply, an autoencoder learns latent representation by minimizing reconstruction loss function $L(\hat{x}, g(f(\hat{x})))$. Considering that the input is binary, we use binary cross-entropy as the loss function. Given an input vector $\hat{x} = [x_1, x_2, \ldots, x_n]$ and corresponding output vector $\hat{y} = [y_1, y_2, \ldots, y_n]$, the loss calculation is

$$L(\hat{x}, g(f(\hat{x}))) = -\frac{1}{n} \sum_{i=1}^{n} x_i \times \ln y_i + (1 - x_i) \times \ln(1 - y_i) \tag{2}$$

In this paper, we use an autoencoder with six dense hidden layers to map features into three-dimensional feature space (see Section 4.4.3 for more details). After training the autoencoder, we detach the decoder part and use the encoder to transform our data to a new clustering-friendly representation. Then, we cluster new features using the k-means algorithm for simplicity (see Fig. 9). It is to be noted that the impact of the clustering algorithm can be studied in further researches.

3.3   Feature Extraction

At this stage, we pick only principal fragments from the fragment collection, which we have earlier created in Section 3.2.1. Then we calculate Inverse-Document-Frequency (IDF) for each fragment to build the final feature vector. We repeat this stage for each cluster individually.

3.3.1   Token Selector

Since each cluster is a subset of the original dataset, a cluster's fragment set is a subset of the main fragment collection. Therefore, as the first step, we remove unused fragments, and then we keep the remaining. For instance, let us assume that there are four fragments extracted from the whole dataset. However, payloads of a cluster are consisting of only two of them. Thus, since the adaptive searching in each cluster is done independently, we do not need those two extra fragments in that cluster. We, therefore, do not consider those unused fragments in building feature vectors in that cluster.

In order to improve our algorithm's performance, within each cluster, we remove non-informative fragments such as highly repetitive and exceedingly rare fragments from our fragment set. For this purpose, first, we calculate entropy for each fragment, and then we remove fragments with an entropy value lower than a threshold. The entropy calculation is given below

$$E(f_i) = -(p(f_i) \log_2 p(f_i) + (1 - p(f_i)) \log_2(1 - p(f_i)))$$
$$= -((\frac{k_i}{N} \times \log_2(\frac{k_i}{N})) + (\frac{N - k_i}{N} \times \log_2(\frac{N - k_i}{N}))) \tag{3}$$

where i is the fragment's number, E is the entropy, $p(f_i)$ is the probability of a fragment f_i being present in a randomly selected payload, k_i is the number of payloads containing the fragment f_i, and N is the size of corresponding cluster.

3.3.2   IDF Calculator

Among the remaining fragments, rare ones are more valuable; thus, we calculate a weight for each unique fragment using inverse document frequency [37]. Then, we create a feature vector for each payload. If a payload contains a fragment, the value of the fragment in the feature vector will be its weight; otherwise, it will be zero. The IDF calculation is as follows:

$$w_f^i = \ln(\frac{N}{k_i}) \tag{4}$$

where i is the fragment's number, w is the weight for fragment f_i, N is the number of payloads, and k_i is the number of payloads containing the fragment f_i within the corresponding cluster.

3.4   Test Oracle

Since our approach is a type of black-box testing, the only information we can obtain from a protector WAF is whether a request is identified as malicious or benign. Therefore, we propose an adaptive search technique; we call here AdaptiveSearch that benefits from prior experiences to minimize failed attempts. The key insight underlying AdaptiveSearch is that if a fragment attends more previously blocked attacks than others, it is more likely to cause a failure. Thus, if a new test payload contains these fragments, it is more likely to be recognized by the WAF. More specifically, in AdaptiveSearch, we assume that the set of previously blocked attacks is a document, and new test candidates are the queries. To find a payload with the highest likelihood of bypassing the WAF, for each query, we calculate a term frequency-inverse document frequency (TF-IDF) score that indicates the relevance of the query to the document [38]. Finally, we pick the payload with the lowest score and then execute it against the WAF. If the chosen payload bypasses the WAF, in further tests, we do not consider its fragments in our score calculation; otherwise, we add the payload to the document.

The pseudo-code for AdaptiveSearch is given in Algorithm 1. Line 2 initializes the payload collection of the cluster. Line 3 defines an array to keep term frequency of

Algorithm 1 Adaptive Search algorithm

```
1:  procedure ADAPTIVESEARCH(cluster, rounds)
2:      P ← getPayloads(cluster)
3:      BV ← getBlockedVector(cluster)
4:      PV ← getBypassingVector(cluster)
5:      S ← {∅}
6:      SR ← +1                              ▷ Reward
7:      FR ← -0.5                            ▷ Punishment
8:      R ← 0                                ▷ Sum of Rewards
9:      for i = 1 to rounds do
10:         if is first time then
11:             test_candidate ← pickRandomPayload(P)
12:         else
13:             rt ← RankPayloads(P, BV, PV)
14:             testCandidate ← pickBestPayload(rt)
15:         end if
16:         P ← P − testCandidate
17:         result ← evaluate(testCandidate)
18:         if result is successful then
19:             S ← S ∪ {testCandidate}
20:             PV ← updateBypassingVector(testCandidate)
21:             R ← R + SR
22:         else
23:             BV ← updateBlockedVector(testCandidate)
24:             R ← R + FR
25:         end if
26:     end for
27:     saveClusterState(P, BV, PV)
28:     return S, R
29: end procedure
```

TABLE 3: Example of the payloads' term frequencies in cluster c.

p.id	Term Frequencies				Inverted Document Frequencies			
	f1	f2	f3	f4	f1	f2	f3	f4
1	2	0	1	0	0.28	0	0.69	0
2	1	1	0	0	0.28	0.28	0	0
3	0	1	2	3	0	0.28	0.69	1.38
4	2	2	0	0	0.28	0.28	0	0

Algorithm 2 Epsilon Greedy Policy

```
1:  procedure EPSILONGREEDY
2:      PC ← Payload Clusters
3:      S ← {∅}
4:      max_ε, ε ← Epsilon
5:      R ← Number of Searching Rounds per episode
6:      K ← Update Rate for Epsilon
7:      E ← Episodes
8:      AR ← Initial Average Reward of Each Cluster
9:      for i = 1 to E do
10:         c ← pickCluster(ε)
11:         bypassing, reward ← ADAPTIVESEARCH(c, R)
12:         AR ← updateAverageReward(c, reward, AR)
13:         S ← S ∪ bypassing
14:         ε ← updateEpsilon(max_ε, i, K)
15:     end for
16: end procedure
```

the previously blocked attacks' fragments. We also initialize a binary vector that keeps the state of each fragment (line 4). If a fragment attends a bypassing payload, its value will be zero; otherwise, it will be one. The value one for a fragment means that we count it in the score calculation.

At the very beginning of the search, we pick the first payload randomly (lines 10-12). In further searches, for each new attempt, we calculate the score of each payload, and then we pick the best payload (lines 13-14). The score calculation is given below

$$score(p) = \sum_{i=1}^{n} b_i \times v_i \tag{5}$$

where i is the fragment's number, b_i is the frequency of the fragment f_i in the blocked attacks, v is the feature vector for payload p, calculated in Section 3.3.2, and n is the size of v.

Each time we pick a payload, we remove it from payload collection; then, we execute it against the WAF (lines 16-17). If a chosen attack bypasses the WAF, we add it to the bypassing collection (line 19). Then in the bypassing vector, we set the value of the items that represent a fragment belonging to the successful payload to zero (line 20). Otherwise, we update the failure vector (line 23) using the following equation:

$$b_i = s_i \times (tf_i + b_i) \tag{6}$$

where i is the fragment's number, b_i is the frequency of the ith fragment in failure vector, tf_i is its frequency in the payload p, and s_i is the state value of fragment f_i in success vector.

To put it in more perspective, let us assume that there are four attack payloads in cluster c. Table 3 shows the frequencies of fragments in these payloads as well as their feature vectors. Assuming the first two payloads are tested and blocked, we sum their term frequency vectors to calculate the vector $\hat{b}$. As a result, the vector $\hat{b}$ is $\hat{b} = [3, 1, 1, 0]$. To select the next payload to test, we calculate the dot product of b and the remaining payloads' feature vectors. The resulting ranks are: $Rank(p_3) = 0.97$ and $Rank(p_4) = 1.12$. Thus, we select the third payload, which has the lower rank. If p_3 bypasses the WAF, we set the values of fragments f_2, f_3 and f_4 in $\hat{s}$ vector to zero as p_3 contains these fragments. The resulting $\hat{s}$ is $\hat{s} = [1, 0, 0, 0]$. Therefore, in the next update, frequencies of these fragments in $\hat{b}$ will be set to zero.

On lines 21 and 24, Algorithm 1 calculates a reward, which we explain in the following sections. Finally, it saves the current state of the cluster (line 27) and returns bypassing attacks and the reward (line 28).

In our observations, we realized that effective payloads are usually rare in payload collection; thus, only a few numbers of clusters contain bypassing payloads. Therefore, limiting searches to these clusters reduces the number of failed attempts significantly. For this purpose, we use the ϵ-greedy policy for cluster selection.

To use ϵ-greedy policy in our method, we define the required terms and parameters as follow:

1) Reward: A positive constant value for each successful attempt.
2) Punishment: A negative constant value for each unsuccessful attempt.
3) Action: An action is the R (line 5 of Algorithm 2) number of attempts that are made for a single cluster.
4) Action Reward: Sum of rewards and punishments per action.
5) Average Reward: Average of a cluster's action rewards.

In ϵ-greedy policy (Algorithm 2), first, we value the actions based on their average rewards. Then we either select a random action with the probability of ϵ or the best action with the probability of $1 - \epsilon$ (line 10).

At the very beginning of the search, we do not have any information about clusters; thus, we set initial ϵ equal to a large number (e.g., 0.9) to perform exploration. As we search more in clusters, we gain more knowledge. Therefore

after each episode, we decrease the value of the ϵ to perform more exploitation (line 14). The calculation for ϵ reduction is given below:

$$\epsilon = max\_\epsilon \times e^{-(k \times \tau)} \qquad (7)$$

where max_ϵ is the maximum value of the ϵ, k is a constant value (see Section 4.4.6), and τ is the number of played episodes.

4 EMPIRICAL STUDY

This section aims to evaluate our proposed method on its efficiency and effectiveness. In Section 4.1, we introduce the research questions. Section 4.2, briefly explains the process of testing the WAFs. In Section 4.3 we describe the experimental environment, including subject WAFs, datasets, and evaluation metrics. Section 4.4 demonstrates the algorithm parameters, and finally, Section 5 answers the research questions.

4.1 Research questions

In our empirical study, we aim to answer the following questions:

Q1: Does the choice of n-gram matter?

Q2: How does clustering affect the performance?

Q3: How does RAT compare with the state-of-the-art techniques?

Q4: Is the efficiency and effectiveness of RAT acceptable in practice?

Q1 and *Q2* assess the effect of clustering and n-gram on the performance of our approach. *Q3* compares our technique with state-of-the-art methods, and *Q4* investigates whether the efficiency and effectiveness of our approach are acceptable in practice. The following sections answer these questions.

4.2 Procedure

In our experiments, we only target the WAF itself, not the application behind it. Since WAFs are independent of the application under protection, for the HTTP key-value pairs, they validate values regardless of keys. Thus, we apply attacks through dummy keys as the HTTP query and the cookie. For instance, we consider the HTTP GET query key as `q`, and then we set an attack payload "`0%20or%201=1%23`"as the query string for `q`. As a result, the URL for the sample request is "`http://example.com/?q=0%20or%201=1%23`". For each test, we either target the GET query parameter or the cookie. In our experiments for SQLi, we test both HTTP GET query parameter and cookie; and for XSS, we only test the GET parameter. It is worth mentioning that our method is independent of the communication protocol; thus, it can be implemented using different request types such as POST, GET, and SOAP messages. In this research, since the subject WAFs' rules are the same for both GET and POST requests, we conduct our experiments using GET for simplicity.

It is to be noted that when WAF receives a request, it investigates the request and if it detects an attack, responds that the request is forbidden. Thus, we understand that the attempt is failed; otherwise, we mark the attack pattern as bypassing.

4.3 Experimental environment

RAT consist of two parts: Data Processor and Test Oracle. The Data Processor is a one time process and requires at least 32GB of RAM and a CUDA-enabled Graphic Card with the minimum compute capability 3.0 whereas Test Oracle requires the minimum 8GB of RAM with 2.10GHz duo core CPU. However, to speed up tests, all the experiments were conducted on a Server with two 2.10GHz Intel(R) Xenon(R) processors and 64GB RAM running Windows 10

TABLE 4: Number of samples in each dataset.

Dataset name	Number of Payloads
SQL Injection	2,417,720
Cross-site Scripting	1,798,062

TABLE 5: Percentage of bypassing payloads for each open-source WAF.

WAF	SQLi		XSS
	GET Parameter	Cookie	GET Parameter
ModSecurity	0.007%	0.079%	0.015%
NAXSI	0.005%	0.005%	0.004%

pro. The program codes were written in Python 3.6, and the source code is publicly available on GitHub[2]. Furthermore, the deep autoencoder implemented with Keras-GPU 2.2.4 running on the top of Tensorflow 1.9 and was executed on Google Colab[3]. The following sections provide details about Subject WAFs, Datasets and metrics.

4.3.1 Subject WAFs

In our case studies, we apply our tests on a custom-built WAF and two famous open-source WAFs: *ModSecurity*[4] and *Naxsi*.[5]

ModSecurity is a toolkit that provides real-time protection for web applications. It protects web applications against various types of attacks, such as SQLi, XSS, and denial of service. We deployed the *ModSecurity* with an Apache HTTP server on a local virtual machine.

Naxsi stands for "Nginx Anti XSS and SQL Injection,"which is a third-party module for Nginx web server that protects web applications against SQLi and XSS attacks. Similar to *ModSecurity*, we deployed *Naxsi* on a local virtual machine.

Custom-built WAF is the modified version of ModSecurity, customized to protect a real-world application's web services. This application provides various educational services, and the private data of thousands of students is stored in its database. Therefore, custom-built WAF is responsible for the privacy of students' data.

4.3.2 Datasets

In this research, we evaluate our technique on two different injection datasets (available on GitHub[6]). The first dataset is a collection of SQLi payloads which we generated using the finite BNF grammar proposed in [18]; thus, we can fairly compare *RAT* with *ML-Driven E*. The next dataset is a collection of XSS payloads, we generated using an opensource fuzzer tool, named dharma.[7] Table 4 shows the number of payloads in each dataset.

4.3.3 Effectiveness metrics

This research aims to increase the number of discovered bypassing attacks while reducing failed attempts. Thus, we consider bypassing payloads as positives and blocked payloads as negatives. As a result of this naming, successful attempts are True Positives (TPs), and unsuccessful ones are False Positives (FPs).

To evaluate and compare the effectiveness of *RAT* and *Ml-Driven E*, we measure TP over the limited number of requests. Since deploying *NAXI* and *ModSecurity* on the local

---

[2]https://github.com/mhamouei/rat
[3]https://colab.research.google.com
[4]https://modsecurity.org
[5]https://www.nbs-system.com
[6]https://github.com/mhamouei/rat_datasets
[7]https://github.com/MozillaSecurity/dharma

TABLE 6: Architecture of the AutoEncoder.

Attack Type	Input layer	Encoder			Bottleneck	Decoder			Output layer
		Hidden layers				Hidden layers			
SQLi	52	36	23	10	3	10	23	36	52
XSS	347	261	175	89	3	89	175	261	347

machines provided the feasibility of brute-force attacks, we applied brute-force attacks on these open-source WAFs by exhaustively testing all attack payloads of our collections on the WAFs. Then, we measured the number of positives for each request parameter. Table 5 shows the percentage of bypassing attack payloads existing in our collections for both WAFs. Knowing the total number of positives, we can calculate $\frac{TP}{positives}$ as True Positive Rate (TPR) or open-source WAFs. However, since communication with the custom-built WAF is through the Internet, we cannot calculate positives for the custom-built WAF. Thus, in our experiments, we only report the number of TPs for the custom-built WAF. Furthermore, to compare *RAT* with *ART4SQLi* and *XSSART*, we need to measure the number of FPs before finding the first TP. The lower FP shows that the approach is faster in discovering the first bypassing payload.

4.3.4 Efficiency metrics

To evaluate the efficiency of *RAT*, we measure the time spent per request (TSR). For this purpose, in each episode, we record the time used in each cluster, then we divide it by the total number of requests made from that cluster. Finally, we report the average TSR.

4.4 Parameter settings

For each step, there are various parameters to set. For the n-gram tokenizer, the only parameter to set is the size of n, which we discuss in detail in Section 5.1. The binary encoder does not have any parameters, and the remaining parameters are as follows:

4.4.1 Hierarchical Clustering

In this step, we need to set a threshold to cluster tokens with a dissimilarity of less than the threshold. Clustering more tokens together results in losing details, and clustering fewer tokens keeps more details, directly affect payload clusters quality. In this step, we do not want to either lose or keep too many details. In our experiments, we observed that a threshold between 0.2 to 0.5 could be a good choice. However, we obtained the best performance in discovering bypassing payloads by setting this threshold to 0.3, which means that if the cosine similarity between the tokens is more than 70%, they should cluster together.

4.4.2 Skip-gram

For the skip-gram, we used a Python library named Gensim[8] with the almost default parameter settings. We only set the window size considering the value of n in n-gram. For example, for $n = 2$, we set the window size to 5. It is because when $n = 2$, half of the middle token is repeated in the prior token and the other half is repeated in the posterior token. Thus, increasing the window size by two results in more meaningful training of the skip-gram as there is no intersection between these two tokens and the middle token (see Fig. 8).

4.4.3 AutoEncoder Architecture

In this research, we use the simplest possible AutoEncoder architecture to extract features for our purpose. Since the AutoEncoders' inputs are low-dimensional binary vectors, a deep AutoEncoder with dense layers can achieve high reconstruction accuracy. However, the choice of hyperparameters of deep learning models, such as the number of hidden

(a) SQLi (b) XSS

Fig. 10: Silhouette scores for different k in k-means in clustering SQLi and XSS datasets.

layers and their size, significantly affect accuracy. Since the way of tuning these parameters is still an open problem [39], we tuned AutoEncoder manually. In the tuning process, we observed that three hidden layers in each encoder and decoder parts are enough, and we could achieve slightly better accuracy by reducing and increasing the size of hidden layers uniformly. We tend to produce feature vectors with the lowest possible dimensionality for easier and better clustering. In our experiments, we attained a reasonable reconstruction accuracy (95.92% and 96.76% for XSS and SQLi datasets, respectively) by reducing the dimensionality to 3. The final architectures of the AutoEncoders are shown in Table 6.

4.4.4 Number of Clusters

The time complexity for selecting each test candidate is $O(n) \times O_s$, where n is the size of the cluster and O_s is the time complexity of the *Rank* function. Therefore, searching inside small clusters requires fewer computations than large clusters. Increasing the number of clusters reduces cluster size and, consequently, features; thus, our adaptive search algorithm would also require fewer searches inside smaller clusters to find bypassing payloads. However, having too many clusters poses two major problems:

1) Bypassing samples spread over more clusters, resulting in the rarity of bypassing payloads inside clusters. Thus, our search technique effectiveness drops.
2) ϵ-greedy algorithm requires more exploration to filter clusters, and due to the previous problem, it is harder to escape local optima.

In our observations, we realized that a number between 25 to 50 clusters is a good compromise for our datasets' size. To find the exact number of clusters, we used silhouette score [40], representing how good clusters are apart and distinguished from each other. The score range is between -1 to 1, and the higher score means that the clusters are better apart and distinguished. We clustered both datasets with all numbers between 25 and 50. We then measured the silhouette score for these clusters and picked the best clusters. Fig. 10 shows the Silhouette scores for the different number of clusters in clustering SQLi and XSS. In Fig. 10a, 25 clusters achieved the highest score, and in Fig. 10b, the highest score belongs to 46 clusters. Therefore, we divide the SQLi and XSS datasets into 25 and 46 clusters, respectively.

4.4.5 Entropy threshold

For the final step of feature reduction, we use entropy to remove improper features. Note that an inadequate thresh-

(a) ModSecurity

(b) NAXSI

(c) Custom-built WAF

Fig. 11: Average true positive in testing different WAFs for SQLi vulnerabilities with different values for n.

(a) ModSecurity

(b) NAXSI

(c) Custom-built WAF

Fig. 12: Average true positive in testing different WAFs for XSS vulnerabilities with different values for n.

old value causes the improper features to remain, and a high threshold value results in information loss. In our experiments, with trial-and-error, we found the threshold of $T = 0.05$ to be the optimal value.

4.4.6 ϵ-greedy parameters

For the ϵ-greedy algorithm, we set the reward to 1, and since we expect to experience failure more than success, we set the punishment to half of the reward value. We set the maximum ϵ to 0.9, and for the epsilon reduction (Eq. (7)), we set the k to 5e−3. The value of k controls the exploration and exploitation rate by adjusting the epsilon reduction speed. In other words, a high k means RAT spends more episodes exploring clusters than a low k. We calibrated k with trial-and-error.

5 RESULTS

In this section, we answer the research questions described in Section 4.1 by our experiments.

5.1 Q1: Does the choice of n-gram matter?

To answer Q1, first, we picked four random subsets with the size of 25000 payloads containing bypassing payloads from both datasets (two subsets for each dataset). We then performed AdaptiveSearch on them using unigram, bigram, and trigram (100 repetitions for each). Then, we calculated the average TP over 1000 requests.

Fig. 11 shows the results of applying AdaptiveSearch on the SQLi subsets, and Fig. 12 shows the results of the same experiment on the XSS subsets. Obtained results show that overall, unigram has poor performance as it can not models sophisticated patterns. We also observed that on the whole, the bigram was slightly more efficient than trigram. This is because although trigram extracts more sophisticated patterns than bigram, the number of distinct patterns extracted by bigram is fewer than trigram. Thus, bigram requires fewer observations than trigram. Moreover, the results show that bigram offers more robust results than others when testing different WAFs using different datasets. Therefore, in our further experiments, we used bigram for both SQLi and XSS datasets.

5.2 Q2: How does clustering affect the performance?

To answer Q2, we investigated the effects of clustering from two aspects. The first aspect is that decreasing the number of test parameters reduces the number of observations. Thus we measured the number of fragments before and after the clustering phase. As shown in Table 7, in the worst case, before applying the feature reduction ($t = 0$), the clustering phase reduced the number of test parameters by 6.48% and 57.26% for SQLi and XSS datasets, respectively. With the clustering and feature reduction with the entropy threshold of $t = 0.05$, in the worst case, we could reduce the number of parameters by 31.89% and 98.50% for SQLi and XSS datasets, respectively. We also measured the ratio $\frac{\text{number of samples}}{\text{number of test parameters}}$ for each cluster to check whether the ratio between samples and test parameters is acceptable. The worst ratio for each dataset is reported in the last column of Table 7.

The second aspect is that bypassing attacks tend to cluster together; thus, only a limited number of clusters contain bypassing samples. To prove our claim, we applied brute force attacks on both Naxsi (NX) and Modsecurity (MS). Targeted parameters for the SQLi attack are a random HTTP GET parameter and Cookie, and for the XSS attack, we targeted a random HTTP GET parameter. Then, we analyzed the distribution of bypassing attacks within the clusters for each tested parameter.

The obtained results are illustrated in Fig. 13. In this figure, each column represents the number of a cluster's bypassing payloads in testing the corresponding parameter and WAF. For instance, in Fig. 13b, the 31st cluster contains more than 100 bypassing payloads in testing the HTTP GET parameter protected by *ModSecurity*. The same cluster has about 40 bypassing payloads testing the same parameter protected by *NAXSI*.

According to Fig. 13, bypassing samples are distributed within a few clusters. Therefore, finding effective clusters using ϵ-greedy policy can significantly reduce unsuccessful attempts in uncovering bypassing payloads by discarding ineffective clusters.

5.3 Q3: How does RAT compare with the state-of-the-art techniques?

To answer Q3, first, we implemented *Ml-Driven E*[9] [18], *ART4SQLi*[10] [7] and *XSSART*[11] [19] based on the original papers, and then we compared *RAT* with *Ml-Driven E*,

---

[9]https://github.com/mhamouei/ml-driven
[10]https://github.com/mhamouei/art4sqli
[11]https://github.com/mhamouei/xssart

(a) SQLi

(b) XSS

Fig. 13: Distribution of bypassing payloads (TP) over clusters.

TABLE 7: Number of test parameters before and after clustering.

Attack Type	Before Clustering	After Clustering						Ratio
	Total Parameters	$T = 0$			$T = 0.05$			
		min	max	mean	min	max	mean	
SQLi	185	51	173	73	49	126	69	247
XSS	113408	826	48470	20714	103	1702	933	3.13

(a) ModSecurity

(b) NAXSI

Fig. 14: Average true positive rate in testing the open-source WAFs for SQLi vulnerabilities.

Fig. 15: Boxplots of the average true positive rate in testing the open-source WAFs for SQLi vulnerabilities.

ART4SQLi and a *Random Fuzzer* using the SQLi dataset, and *XSSART* using the XSS dataset. The comparison tests are designed concerning the objective of each approach. Moreover, to verify that *RAT* also works with other attacks, we compare *RAT* with a *Random Fuzzer* using the XSS dataset.

To compare *RAT* with *Ml-Driven E* and the *Random Fuzzer*, since the objective of *Ml-Driven E* is to discover the highest possible number of SQLi vulnerabilities, we measured TPR over 30,000 requests for each approach. We then assess the results to compare their effectiveness.

ART4SQLi and *XSSART* tend to find the very first bypassing payload with the lowest number of requests. Therefore, we measured FPs before finding the first bypassing payload to compare *RAT* with these techniques. For this purpose in each episode, we selected a random cluster and applied the *AdaptiveSearch* to the chosen cluster for one round.

We applied all techniques to open-source WAFs, and then we calculated the average TPR for all parameters (Section 4.2). Comparative tests between *RAT*, *Ml-Driven E* and the *Random Fuzzer* are repeated 30 times as these experiments are time-consuming, and increasing the number of tests was not feasible with our resources. On the other hand, comparative tests between *RAT*, *ART4SQLi* and *XSSART* are not time-consuming. Moreover, since *ART4SQLi* and *XSSART* are at most 27% more efficient than random testing, their tests require more repetition to report reliable results. Therefore, we repeated the comparative tests between *RAT*, *ART4SQLi* and *XSSART* 100 times.

To statistically compare different methods, we used Wilcoxon rank-sum test with the significance level of $\alpha = 0.05$. Since this test is non-parametric, it does not require the samples to be normally distributed. To perform the Wilcoxon test, we collected all test results for each repetition and then grouped them by the type of attack (e.g., SQLi or XSS). Thus, we report the Wilcoxon results for each type of attack individually.

Fig. 14 depicts the result of the comparison between *RAT*, *Ml-Driven E* and the *Random Fuzzer*, and Fig. 15 illustrates the same result in the form of boxplots to visualize statistical variation. As shown in Fig. 14, within 30,000 requests, the *RAT* could achieve an average of 95.37% and 100% TPR in testing *ModSecurity* and *NAXSI*, respectively. Moreover, in the worst case, It could find 83.53% of bypassing payloads (Fig. 15). In comparison, in the best case, *Ml-Driven E* could find 48.51% and 76.56% of bypassing payloads in testing *ModSecurity* and *NAXSI*, respectively. The observations reveal that *RAT* and *Ml-Driven E* can find bypassing attacks, whereas *Random Fuzzer* failed due to the rarity of bypassing attacks. Moreover, the results clearly show that the *RAT* has a lower false-positive rate and significantly outperforms counterparts.

Table 8 shows how each approach outperforms others after each $10,000$ requests. We measured TPR after each $10,000$ requests for each repetition to create this table and then used the Wilcoxon test to compare TPRs of different

TABLE 8: Result of the Wilcoxon test in testing SQLi vulnerabilities of ModSecurity and NAXSI.

Requests	Random Fuzzer	Ml-Driven E $(p < .001)$	RAT $(p < .001)$
10,000	—	Random Fuzzer	Ml-Driven E Random Fuzzer
20,000	—	Random Fuzzer	Ml-Driven E Random Fuzzer
30,000	—	Random Fuzzer	Ml-Driven E Random Fuzzer

TABLE 9: Number of false positives before finding the first SQLi bypassing payload.

Method	GET Parameter		Cookie	
	ModSecurity	NAXSI	ModSecurity	NAXSI
RAT	142.4	209.3	179.33	210.78
ART4SQLi	—	—	465.02	—
Random	—	—	—	—

(a) ModSecurity

(b) NAXSI

Fig. 16: Average true positive rate in testing the open-source WAFs for XSS vulnerabilities.

methods statistically. In Table 8, each cell represents the approaches outperformed by the approach matching the corresponding column when the $p - values < .001$.

Table 9 shows the result of the comparison between *RAT*, *ART4SQLi* and random technique. The observations show that *RAT* could find the first bypassing with the reasonable number of false positives, whereas *ART4SQLi* failed to find a bypassing payload within the limited number of attempts except for the cookie parameter of *ModSecurity* in which *RAT* was 61.43% faster than *ART4SQLi* on average. We also compared FPs in testing cookie parameter of *ModSecurity* using the Wilcoxon test. As a result, *RAT* could outperform *ART4SQLi* with the $p - values < .001$.

Furthermore, we performed the same experiment on *RAT* and *XSSART* using the XSS dataset. We observed that for ModSecurity, *RAT* could find the first bypassing payload after 581.45 unsuccessful attempts and failed in finding the first bypassing payload within the limited number of attempts in testing NAXSI. In comparison, *XSSART* failed in both situations as well as random techniques.

Finally, to evaluate the performance of *RAT* at testing a different attack type, we applied *RAT* to *ModSecurity* and *NAXSI* using the XSS dataset. Fig. 16 depicts the result of testing *ModSecurity* and *NAXSI* for XSS vulnerabilities using *RAT* and the Random Fuzzer, and Fig. 17 shows the statistical variation of the test results. According to Fig. 16, *RAT* could find 100% of bypassing payloads before reaching 10,000 requests, whereas the *Random Fuzzer* could only discover 1% of bypassing payloads. Moreover, we compared these two methods using Wilcoxon test. The result verified that *RAT* can clearly outperform *Random Fuzzer* when the $p - values < .001$.

Fig. 17: Boxplots of the average true positive rate in testing the open-source WAFs for XSS vulnerabilities.

(a) Line Chart

(b) Boxplots

Fig. 18: Average number of bypassing attacks in testing the custom-built WAF for SQLi vulnerabilities.

(a) Line Chart

(b) Boxplots

Fig. 19: Average number of bypassing attacks in testing the custom-built WAF for XSS vulnerabilities.

TABLE 10: The result of comparison between *RAT*, *ART4SQLi*, and *Random Fuzzer* in testing the custom-built WAF for SQLi vulnerabilities. In this table, the mean is the average FP before finding the first bypassing payload.

	Random	ART4SQLi	RAT
Mean	4.93	4.64	7.57
Std	5.96	4.63	7.21
Wilcoxon result	RAT $(p < 0.001)$	RAT $(p < 0.01)$	—

5.4 Q4: Is the efficiency and effectiveness of RAT acceptable in practice?

To answer the final question, we applied *RAT* and its counterparts to the custom-built WAF over the internet (30 repetitions for each test). To evaluate the effectiveness, we tested 30,000 payloads with each method and measured the number of uncovered bypassing payloads (TP). Since it is a real-world application, the brute-force attack was infeasible, and we could not measure the total number of bypassing attacks to calculate TPR. Therefore we only report TP.

Fig. 18a shows the result of applying the three methods to the custom-built WAF for SQLi discovery, and Fig. 18b shows the same result in the form of box-plots for better

TABLE 11: The result of comparison between *RAT*, *XSSART*, and *Random Fuzzer* in testing the custom-built WAF for XSS vulnerabilities. In this table, the mean is the average FP before finding the first bypassing payload.

	Random	XSSART	RAT ($p < 0.001$)
Mean	92.95	107.61	37.77
Std	92.16	116.63	15.65
Wilcoxon result	—	—	Random XSSART

statistical visualization. It shows that the *RAT* can uncover a significant number of bypassing payloads within a reasonable number of requests. On average, *RAT* sent 2.83 requests for each bypassing payload, whereas this number for *Ml-Driven E* is 4.06 and 13.56 for the Random method.

To evaluate the efficiency of *RAT*, we measured the average TSR for *RAT* and *Ml-Driven E*. The average TSR value for *RAT* was 0.74 seconds, and for *Ml-Driven E*, this value was 0.80 seconds. The TSR values for both methods were almost the same, and it is a reasonable value in practice.

We also conducted the same experiment for the XSS attack (see Fig. 19a and 19b). The results are as follows.

1) RAT sent 4.44 requests on average, whereas the Random technique sent 184.11 requests.
2) The average TSR value for *RAT* in this experiment was 0.39 seconds, which was lower than the TSR value of the SQLi test due to the fewer samples.

We conducted the Wilcoxon test to compare TPs of *RAT* with *Ml-Driven E* and *Random Fuzzer*. The results of testing SQLi were the same as in Table 9, and in testing XSS, *RAT* could outperform *Random Fuzzer* when the $p - values < .001$.

We also repeated our comparative experiments between *RAT*, *ART4SQLi*, and *XSSART* for the custom-built WAF to compare the performance of these approaches in practice. The results are shown in Tables 10 and 11. According to Table 10, Random strategy and *ART4SQLi* could discover the first bypassing payload with fewer attempts than *RAT*. Wilcoxon results also verify that Random strategy and *ART4SQLi* could outperform *RAT*. However, neither *ART4SQLi* nor Random strategy could outperform the other one. The possible reason for the obtained results can be the massive number of SQLi vulnerabilities of the custom-built WAF. Despite the results shown in Table 10, Table 11 shows that *RAT* could significantly perform better than *XSSART* and the Random strategy in discovering the first XSS bypassing payload.

In conclusion, our answer to the Q4 is that yes, *RAT* is efficient and effective in practice.

6 CONCLUSION

In this paper, we proposed the *RAT*, a search-based technique that combines a reinforcement learning algorithm with an innovative adaptive search method. *RAT* automatically extracts patterns from attack payloads. It then clusters similar payloads together and discovers the clusters which contain bypassing payloads using a reinforcement learning technique. Finally, *RAT* ranks test candidates and selects the payload with the highest probability of bypassing the WAF.

Empirical results suggested that considering the sequence of tokens rather than single literals and clustering payloads improves the performance of discovering effective payloads. Comparative experiments, moreover, showed that *RAT* significantly performs better than the state-of-the-art algorithms. Finally, the result of applying *RAT* to a real-world WAF demonstrated that *RAT* is efficient and effective in practice. However, *RAT* is highly dependent on the dataset, and the comprehensiveness of the dataset directly affects the *RAT*'s performance. Furthermore, *RAT* can only test rule-based WAFs, and it cannot be used alone to test Ml-based WAFs. Nevertheless, *RAT* is capable of being combined with generative adversarial techniques (e.g., *WAF-A-MoLE* [15]) to reduce its dependency on datasets and test Ml-based WAFs.

In our future studies, we will work on combining *RAT* with Generative Adversarial Networks to propose an adversarial test strategy as well as a solution for altering WAFs. We will also focus on building comprehensive datasets with the least number of samples, and we will investigate different clustering strategies to improve *RAT*'s efficiency.

7 ACKNOWLEDGEMENT

REFERENCES

[1] D. E. Simos, B. Garn, J. Zivanovic, and M. Leithner, "Practical combinatorial testing for xss detection using locally optimized attack models," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2019, pp. 122–130.

[2] K. Chandrasekar, G. Cleary, O. Cox, H. Lau, B. Nahorney, B. O. Gorman, D. O'Brien, S. Wallace, P. Wood, and C. Wueest, "Internet security threat report (ISTR)," Symantec, Tech. Rep. April, 2017.

[3] O. G. Chapter, "Owasp best practices: Use of web application firewalls.[whitepaper]," 2008.

[4] A. Tekerek and O. Bay, "Design and implementation of an artificial intelligence-based web application firewall model," *Neural Network World*, vol. 29, no. 4, pp. 189–206, 2019.

[5] A. M. Vartouni, M. Teshnehlab, and S. S. Kashi, "Leveraging deep neural networks for anomaly-based web application firewall," *IET Information Security*, vol. 13, no. 4, pp. 352–361, 2019.

[6] H. Mac, D. Truong, L. Nguyen, H. Nguyen, H. A. Tran, and D. Tran, "Detecting attacks on web applications using autoencoder," in *Proceedings of the Ninth International Symposium on Information and Communication Technology*. ACM, 2018, pp. 416–421.

[7] L. Zhang, D. Zhang, C. Wang, J. Zhao, and Z. Zhang, "Art4sqli: The art of sql injection vulnerability discovery," *IEEE Transactions on Reliability*, vol. 68, no. 4, pp. 1470–1489, 2019.

[8] D. Wichers and J. Williams, "Owasp top-10 2017," *OWASP Foundation*, 2017.

[9] J. Bozic, B. Garn, I. Kapsalis, D. Simos, S. Winkler, and F. Wotawa, "Attack pattern-based combinatorial testing with constraints for web security testing," in *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2015, pp. 207–212.

[10] D. E. Simos, K. Kleine, L. S. G. Ghandehari, B. Garn, and Y. Lei, "A combinatorial approach to analyzing cross-site scripting (xss) vulnerabilities in web application security testing," in *IFIP International Conference on Testing Software and Systems*. Springer, 2016, pp. 70–85.

[11] D. E. Simos, J. Zivanovic, and M. Leithner, "Automated combinatorial testing for detecting sql vulnerabilities in web applications," in *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*. IEEE, 2019, pp. 55–61.

[12] J. Thomé, A. Gorla, and A. Zeller, "Search-based security testing of web applications," in *Proceedings of the 7th International Workshop on Search-Based Software Testing*, 2014, pp. 5–14.

[13] A. Avancini and M. Ceccato, "Security testing of web applications: A search-based approach for cross-site

scripting vulnerabilities," in *2011 IEEE 11th international working conference on source code analysis and manipulation*. IEEE, 2011, pp. 85–94.

[14] F. Duchene, R. Groz, S. Rawat, and J.-L. Richier, "Xss vulnerability detection using model inference assisted evolutionary fuzzing," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 815–817.

[15] L. Demetrio, A. Valenza, G. Costa, and G. Lagorio, "Waf-a-mole: evading web application firewalls through adversarial machine learning," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, pp. 1745–1752.

[16] O. Tripp, O. Weisman, and L. Guy, "Finding your way in the testing jungle: a learning approach to web security testing," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 347–357.

[17] D. Appelt, C. D. Nguyen, and L. Briand, "Behind an application firewall, are we safe from sql injection attacks?" in *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 2015, pp. 1–10.

[18] D. Appelt, C. D. Nguyen, A. Panichella, and L. C. Briand, "A machine-learning-driven evolutionary approach for testing web application firewalls," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 733–757, 2018.

[19] C. Lv, L. Zhang, F. Zeng, and J. Zhang, "Adaptive random testing for xss vulnerability," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2019, pp. 63–69.

[20] G. McGraw, "Software security," *IEEE Security & Privacy*, vol. 2, no. 2, pp. 80–83, 2004.

[21] M. E. Khan, F. Khan *et al.*, "A comparative study of white box, black box and grey box testing techniques," *Int. J. Adv. Comput. Sci. Appl*, vol. 3, no. 6, 2012.

[22] R. Elderman, L. J. Pater, and A. S. Thie, "Adversarial reinforcement learning in a cyber security simulation," Ph.D. dissertation, Faculty of Science and Engineering, 2016.

[23] A. Mnih and G. E. Hinton, "A scalable hierarchical distributed language model," in *Advances in neural information processing systems*. Citeseer, 2009, pp. 1081–1088.

[24] S. Overflow, "Stack overflow annual developer survey," 2019.

[25] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner, "Security testing: A survey," in *Advances in Computers*. Elsevier, 2016, vol. 101, pp. 1–51.

[26] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, "Practical machine learning tools and techniques," *Morgan Kaufmann*, p. 578, 2005.

[27] K. Singh, H. M. Devi, A. K. Mahanta *et al.*, "Document representation techniques and their effect on the document clustering and classification: A review." *International Journal of Advanced Research in Computer Science*, vol. 8, no. 5, 2017.

[28] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[29] S.-S. Choi, S.-H. Cha, and C. C. Tappert, "A survey of binary similarity and distance measures," *Journal of Systemics, Cybernetics and Informatics*, vol. 8, no. 1, pp. 43–48, 2010.

[30] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[31] T. Hofmann, B. Schölkopf, and A. J. Smola, "Kernel methods in machine learning," *The annals of statistics*, pp. 1171–1220, 2008.

[32] R. Min, D. A. Stanley, Z. Yuan, A. Bonner, and Z. Zhang, "A deep non-linear feature mapping for large-margin knn classification," in *2009 Ninth IEEE International Conference on Data Mining*. IEEE, 2009, pp. 357–366.

[33] D. Chen, J. Lv, and Y. Zhang, "Unsupervised multi-manifold clustering by learning deep representation," in *Workshops at the thirty-first AAAI conference on artificial intelligence*, 2017.

[34] B. Yang, X. Fu, N. D. Sidiropoulos, and M. Hong, "Towards k-means-friendly spaces: Simultaneous deep learning and clustering," in *international conference on machine learning*, 2017, pp. 3861–3870.

[35] K. Ghasedi Dizaji, A. Herandi, C. Deng, W. Cai, and H. Huang, "Deep clustering via joint convolutional autoencoder embedding and relative entropy minimization," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 5736–5745.

[36] P. Huang, Y. Huang, W. Wang, and L. Wang, "Deep embedding network for clustering," in *2014 22nd International conference on pattern recognition*. IEEE, 2014, pp. 1532–1537.

[37] K. Papineni, "Why inverse document frequency?" in *Second Meeting of the North American Chapter of the Association for Computational Linguistics*, 2001.

[38] S. Jabri, A. Dahbi, T. Gadi, and A. Bassir, "Ranking of text documents using tf-idf weighting and association rules mining," in *2018 4th International Conference on Optimization and Applications (ICOA)*. IEEE, 2018, pp. 1–6.

[39] H. Shaziya and R. Zaheer, "Impact of hyperparameters on model development in deep learning," in *Proceedings of International Conference on Computational Intelligence and Data Engineering*. Springer, 2021, pp. 57–67.

[40] P. J. Rousseeuw, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis," *Journal of computational and applied mathematics*, vol. 20, pp. 53–65, 1987.

Mohammadhossein Amouei is an MSc student in the Faculty of Computer Engineering at the Shahrood University of Technology, Shahrood, Iran. His research interests include artificial intelligence and computer security.

Mohsen Rezvani received the PhD degree in computer science from the University of New South Wales, Australia. He is a faculty member in the Faculty of Computer Engineering, Shahrood University of Technology, Iran. His research focuses on computer security, privacy, trust and reputation systems.

Mansoor Fateh received the M.S. degree in Biomedical Engineering from Tarbiat Modares University, Tehran, Iran, and the PhD from Tarbiat Modares University, Tehran, Iran. He is a faculty member in the Faculty of Computer Engineering, Shahrood University of Technology, Iran. His research interests include machine learning and image processing.