Proceedings of Machine Learning Research vol 278:1–13, 2025 2025 2nd International Conference on Machine Learning and Intelligent Computing

WAFUzz:A Fuzz-based WAF protection function testing technology

Enhui Mao MAOENHUI@GS.ZZU.EDU.CN
School of Cyber Science and Engineering, Zhengzhou University Zhengzhou 450002, China

Danbo Li LDB09@GS.ZZU.EDU.CN
School of Cyber Science and Engineering, Zhengzhou University Zhengzhou 450002, China

Xuexiong Yan* YANXUEXIONG@SOHU.COM
The School of Cyberspace Security, Information Engineering University, Zhengzhou 450001, China
**Corresponding author*

Editors: Nianyin Zeng, Ram Bilas Pachori and Dongshu Wang

Abstract

Web Application Firewalls (WAFs) are designed to detect and intercept potentially malicious HTTP requests, thereby protecting web applications from various attacks. However, if the WAF's rule set and detection strategy are flawed, its protection function may fail under certain conditions, making it difficult to ensure comprehensive application security. Existing WAF protection testing methods either rely on fixed attack payload datasets, which may lead to inefficient testing due to dataset limitations, or use machine learning to pre-train adversarial WAF models, which are not suitable for testing WAF services deployed in the real world.

To address this issue, we propose a new WAF evaluation technique based on fuzz testing. This method uses context-free grammars to generate diverse attack payloads and combines Monte Carlo Tree Search (MCTS) to optimize mutation paths, thereby achieving systematic testing of WAF defense measures. Specifically, we predefine context-free grammars for SQL injection (SQLi) and cross-site scripting (XSS) based on expert knowledge to generate the initial input for fuzz testing and serve as seed payloads for subsequent mutations. Then, MCTS guides the mutation process by dynamically adjusting node weights to prioritize the exploration of promising paths, thereby improving test efficiency and effectiveness.

Experimental results show that our approach reduces the protection failure rate of SQLi and XSS to 48.80% and 37.80%, respectively, outperforming benchmark tools such as WAF-A-MOLE and SqlMap. In addition, the invalid payload rate is also reduced to 5.63% and 6.72% for SQLi and XSS, and the number of WAF queries is reduced by more than 22 times, demonstrating the excellent evaluation efficiency of our approach.

Keywords: Web application firewall (WAF); fuzzing; context free grammar; attack payload; monte carlo tree search

1. Introduction

With the continuous advancement of digital technology and the widespread adoption of the internet, web applications have become the backbone of modern services across diverse industries, including finance, healthcare, e-commerce, education, and government (Vallabhaneni et al., 2024). These applications often handle sensitive user data, business logic, and mission-critical operations, making them prime targets for malicious actors. In response, Web Application Firewall (WAF) have emerged as a key security layer to detect, filter, and block potentially harmful HTTP traffic, aiming to mitigate the risks posed by a wide range of web-based attacks.

Among the most prevalent and damaging attack types are SQL Injection (SQLI) (Kumar et al., 2024) and Cross-Site Scripting (XSS) (Alanda et al., 2024). SQLI enables attackers to manipulate backend database queries by injecting malicious input, potentially gaining unauthorized access to or altering sensitive data. A classic example includes manipulating a query via inputs such as id=1' OR 1=1–+, which tricks the database into returning unintended results. More sophisticated variants utilize obfuscation techniques to evade detection, such as hexadecimal encoding, inline comments, or rare syntax combinations (e.g.,id=1'||0×1 /*!44444=*/(SELECT 1)–+). XSS attacks, in contrast, inject malicious scripts into web pages viewed by other users, leading to session hijacking, cookie theft, or malicious redirection. These attacks exploit the trust a user places in a legitimate website and are particularly insidious due to their ability to affect users without direct server-side impact.

Due to the ease of use and widespread nature of Web application attack techniques, Web applications are facing more and more security threats, and the attack methods are diverse. According to the 2023 Cloudflare Application Security Report (Tremante et al., 2023), around 6% of daily HTTP traffic is classified as malicious, with WAF responsible for intercepting 41% of these threats. Despite these efforts, the effectiveness of WAF is heavily dependent on their ability to adapt to emerging threats. Rule-based detection relies on rules predefined by security experts. It is effective in intercepting malicious requests but less effective in intercepting new attacks. Machine learning-based detection automatically identifies potential threats by learning from a large amount of labeled data, but requires continuous updating of its rule model. However, no matter what type of WAF, if its rules or models fail to cover all malicious attacks, attackers may cause the WAF protection function to fail by confusing requests.

This paper proposes a Fuzz-based WAF protection function testing method that combines grammar-generated Fuzz with the Monte Carlo Tree Search (MCTS) strategy. Specifically, this paper introduces a method that leverages context-free grammar (CFG) to define syntactically valid and semantically coherent payloads for SQLI and XSS attacks. These grammars encode structural patterns of malicious inputs and enable automated construction of diverse initial payloads. To guide the mutation process and efficiently explore high-impact test cases, we integrate Monte Carlo Tree Search (MCTS), a heuristic search algorithm commonly used in game theory and decision-making problems. MCTS incrementally builds a search tree and uses exploration-exploitation trade-offs to prioritize promising paths, thereby reducing redundancy and improving testing depth.

Our contributions are summarized as follows:

(1) This paper improves a method of attack payload generation based on context-free grammar. By summarizing the attack modes of SQL injection (SQLI) and cross-site scripting (XSS), a context-free semantics (CFG) rule for payload generation is designed. Under the premise of ensuring semantic consistency, the initial attack payload is randomly selected and generated through the CFG rules in the grammar file, which improves the test effect of the test case.

(2) This paper proposes a guided mutation method for improving the Monte Carlo Tree Search algorithm. MCTS is introduced in the mutation process to dynamically optimize the mutation path. The generation of attack payloads is guided by the improved UCT formula, and potentially more effective mutation paths are explored first, avoiding the search redundancy problem caused by random mutation.

(3) We implement the proposed approach as a prototype tool named WAFuzz, capable of evaluating WAF protection effectiveness. Experimental evaluation on real-world WAF systems demonstrates that WAFuzz achieves significantly lower protection failure rates and invalid payload ratios

compared to existing tools. Furthermore, WAFuzz successfully identifies multiple evasion cases, revealing latent weaknesses in commercial and open-source WAF implementations.

2. Related Work

As Web Application Firewall (WAF) become increasingly critical in defending against malicious traffic, various WAF testing tools have been developed to assess their protective capabilities under diverse attack scenarios. Existing WAF testing techniques generally fall into two categories: (1) payload-driven search methods, which rely on curated attack payload collections combined with optimization strategies or machine learning; and (2) mutation-based methods, which aim to generate novel attack variants by transforming existing payloads. Both approaches have demonstrated utility, yet each faces inherent limitations.

(1) Payload-Based Testing and Search Optimization

Payload-based methods typically utilize publicly available datasets such as sqliv5 (nidnogg, 2023) and the XSS Cheat Sheet (Heyes, 2024) to construct attack test suites. These datasets, while comprehensive, may not fully reflect modern evasion techniques and are often integrated into WAF detection rules, diminishing their effectiveness.

Tripp et al. (2013) proposed XSS Analyzer, a pattern-learning-based black-box testing system that employs large-scale XSS payload testing to identify WAF filtering rules. By pruning payloads likely to be blocked by WAF, the system reduces ineffective attempts and narrows the search space. However, the vast payload pool—exceeding 500 million entries—and limited targeting effectiveness hinder its practicality.

To enhance adaptability, Amouei et al. (2021) introduced the Reinforcement-learning Automated Tester, which clusters attack payloads using n-gram feature extraction and applies reinforcement learning to prioritize testing based on similar payload patterns. While RAT improves efficiency over exhaustive search, it suffers from high computational overhead and heavy reliance on training data quality. Moreover, its greedy search strategy may lead to suboptimal results by over-exploring specific clusters and neglecting others.

(2) Mutation-Based Testing and Semantic-Aware Generation

Mutation-based approaches aim to increase payload diversity by applying syntactic or semantic transformations to existing attack strings.Demetrio et al. (2020) proposed WAF-A-MoLE, an adversarial machine learning framework that mutates initial payloads through operations such as synonym substitution, casing, whitespace injection, and comment insertion. It selects the most promising variants based on feedback from a pretrained WAF model. While effective in reducing detection confidence, WAF-A-MoLE struggles with real-world WAF compatibility due to discrepancies between training datasets and actual deployments. Additionally, its use of regular expressions for mutation limits its ability to preserve complex SQL semantics.

To maintain payload semantics during mutation, Appelt et al. (2018) proposed an ML-driven testing framework that integrates classical evolutionary algorithms with supervised learning. They defined three SQLI attack templates and used context-free grammar (CFG) to generate parse trees, which were then sliced into semantic components. These components were evaluated by a classifier to identify likely evasion combinations and refined via evolutionary mutation. While this approach improves test success rates, it is constrained by the need for high-quality seed files and considerable training time, limiting its applicability to dynamic real-world WAF testing.

To address efficiency and semantic consistency, Qu et al. (2022) developed AutoSpear, which leverages grammatical structure and semantic parsing to guide payload mutation. It transforms attack strings into tree representations and uses MCTS to efficiently explore evasive variants. Although AutoSpear shows promise in discovering SQLI weaknesses, it relies on pre-defined delimiters during parsing, reducing its generalizability to diverse payload forms. Furthermore, its focus is limited to SQLI vulnerabilities, leaving other threat types such as XSS largely untested.

3. Methodology

Figure 1 shows the WAFuzz system architecture. First, WAFuzz uses the grammar file to generate an initial test case as a seed payload. While generating the initial test case, it builds a corresponding grammar tree. Next, the seed payload is sent to the WAF under test to simulate the actual attack request. The response analysis module parses the WAF's response to the request (such as the interception status and HTTP return code) to determine whether the attack payload is intercepted. When the payload is intercepted by the WAF, the system uses MCTS to guide the mutation of the attack payload. On the basis of maintaining semantic consistency, a new mutated payload is generated through the mutation strategy. Before the attack payload is not intercepted, the mutation operation is repeated. When it is detected that the attack payload is not intercepted, the mutation ends and the program ends.

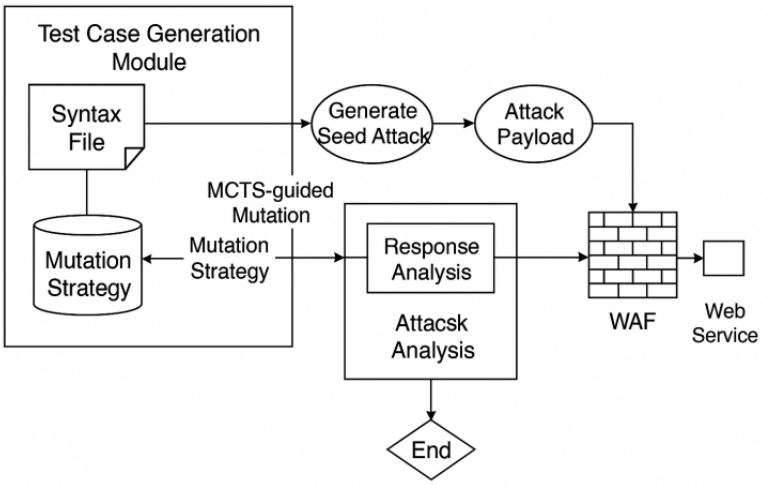

Figure 1: WAFuzz System Architecture.

3.1. CFG-based grammar file construction

To avoid problems similar to those encountered by WAF-A-MoLE (mutations may lead to semantic corruption due to the use of regular expressions), WAFuzz adopts context-free grammar (CFG) (Esparza et al., 2024) to build grammar files. In the Chomsky hierarchy (Someya et al., 2024), regular grammar is a lower-level representation that is only suitable for simple pattern matching, while context-free grammar is at a higher level and supports the complexity of programming lan-

guages and database languages. CFG is able to preserve the hierarchical structure of complex attack payloads, thereby maintaining their semantic integrity during testing.

During the grammar file construction phase, we first organize and analyze the collected test sets to summarize the attack payload characteristics of SQL injection (SQLI) and cross-site scripting (XSS). For example, in SQL injection attacks, payloads are classified according to the injection point type, which can be a number or a string. String-based injections can be further subdivided into single quote and double quote types. SQL injection attacks contain a variety of attack patterns, and examples of these patterns are listed in Table 1.

In order to formalize the components of the attack payload, we developed a symbolic representation of the elements of each SQL injection and XSS attack payload. Table 2 shows some examples of the formalization of the SQL injection component symbols. This symbolic representation is crucial for converting the attack payload into a formal context-free grammar.

Table 1: SQLI Mutation Strategy Table.

Attack Mode	Example Payload
Boolean Injection	' or '1'='1'#
Error-Based Injection	(numeric) and updatexml(0x7e,concat(0x7e,database()),0x7e) –
Union Query Injection	union select 1,database()–
Stacked Injection	"; drop table users–
Time-Based Injection	" and (sleep(5))–

Table 2: SQL Injection Component Symbol Formalization Example.

Component	Formal Definition	Component	Formal Definition
Numeric Injection	<numericContext>	Double Quote Injection	<dQuoteContext>
Or Keyword	<opOr>	Boolean Attack	<booleanAttack>
Single Quote Injection	<sQuoteContext>	Space Character	<wsp>

By formalizing the components of attack payloads, we can use CFG to represent these elements and define the overall structure of SQL injection and XSS payloads. The following Backus-Naur Form (BNF) rules define the structure of SQL injection types, starting with '<start>', which is the rule's starting symbol:

<start> ::= <numericContext> | <sQuoteContext> | <dQuoteContext>

WAFuzz is also extensible, as support for additional vulnerabilities can be easily integrated by adding corresponding CFG grammar files following the BNF format.More grammar files for testing additional vulnerability types will be incorporated in future work.

3.2. Syntax tree-based mutation method

In the process of generating attack payloads using grammar files, WAFuzz builds a grammar tree by recursively traversing the context-free grammar rules starting with the start symbol '<start>'. Each non-leaf node in the grammar tree represents a formal grammar rule, while each leaf node corresponds to a terminal string. The final attack payload is formed by connecting the leaf nodes from left to right.

WAFuzz uses a weighted random selection strategy to build the grammar tree. The weights of different rule selections are pre-defined based on empirical evaluations against common WAF rule sets, such as the Core Rule Set (CRS), to increase the probability of generating payloads. Figure 2 shows an example of a SQL injection grammar file. Starting from '<start>', productions such as <numericContext> can be selected. In this process, rules like '<terDigitIncludingZero> <wsp> <booleanAttack> <wsp>' are selected and each of their components is recursively expanded according to the grammar rules. When there are no other production rules available, the corresponding terminal string is assigned to the leaf node. Once the tree is fully constructed, the initial payload is assembled by connecting all terminal nodes in order. Following this process, an initial SQL injection payload can be generated, such as '0+or%0brand()!=0%0a'. After the initial payload is generated, WAFuzz applies mutations by modifying subtrees of the syntax tree while strictly adhering to the CFG rules. Consider a subtree rooted at the leaf node 'rand()!=0'. By analyzing its context in the syntax, valid semantic equivalents such as true can be identified. Mutations can be performed by inserting or replacing nodes - for example, replacing 'rand()!=0' with true, resulting in a new syntax tree. Connecting the leaf nodes of this updated tree results in the mutated payload: '0+or%0btrue%0a'.

```
'<start>':['<numericContext>','<sQuoteContext>'],

'<numericContext>':['<terDigitIncludingZero><Wsp><sqliAttack><cmt>',
                    '<terDigitIncludingZero><Wsp><booleanAttack><wsp>',...],
'<terDigitIncludingZero>':['<terDigitZero>'],
'<terDigitZero>':['0'],
'<booleanAttack>':['<orAttack>','<andAttack>'],
'<orAttack>':['<opOr><wsp><booleanTrueExpr>'],
'<booleanTrueExpr>':['<binaryTrue>','<unaryTrue>'],
'<unaryTrue>':['<trueAtom>','<opNot><wsp><flaseAtom>'...],
'<trueAtom>':['rand()!=0','true'...],
'<opOr>':['or','||'],
'<wsp>':['<blank>'],
'<blank>':['+','%0b','%09','%0a','%0c','%0d']
```

Figure 2: Grammar file example.

This mutation mechanism ensures that all syntactic and semantic constraints are preserved, avoiding the issues of semantic corruption commonly observed in regex-based or random mutation strategies. By leveraging the structure of the syntax tree and respecting contextual rules, each mutation operation produces payloads that maintain the intent and logic of the original, increasing the likelihood of bypassing WAF without introducing invalid input.

3.3. Guided mutation based on Monte Carlo tree search

This paper improves the traditional UCB formula used in Monte Carlo tree search(MCTS) (Świechowski et al., 2023) and optimizes each stage of MCTS to make it more suitable for WAF protection function testing scenarios. Finally, an improved UCT-Rand algorithm is proposed. The algorithm code is shown in Algorithm 1. Specifically, UCT-Rand uses the UCB formula shown in Formula 1 to select the best child node, and uses weighted random selection to explore the tree, and maintains a balance between utilization and utilization during the search process.

Algorithm 1 UCT-Rand Algorithm

Input: Syntax tree T
Initialization: $V \leftarrow \{T.\text{root}\}$
repeat
While V is not empty
    $v \leftarrow V.\text{pop}()$
    If v is not a leaf node then
        $c \leftarrow \text{weighted\_rand}(v.\text{children})$
    If c is not fully expanded then
        $p \leftarrow \text{mutate}(c)$
        $c.\text{add\_child}(p)$
        $V.\text{push}(c)$
    simulate(c)
    If bypass_successful(c) then
        exit
    else
        backpropagation(c)

$$\pi(v) := \underset{v' \in v.\text{ children}}{\text{weighted\_rand}} \left(Q\left(v, v'\right)\right) + \sqrt{\frac{2 \ln N(v)}{N\left(v, v'\right)}} \tag{1}$$

The guided mutation process in Monte Carlo Tree Search (MCTS) involves four stages: selection, which follows an optimal path (e.g., using UCB) to an unexpanded node; expansion, which adds new child nodes; simulation, which estimates node potential via random playouts; and backpropagation, which updates visit counts and potential values along the path based on simulation results.

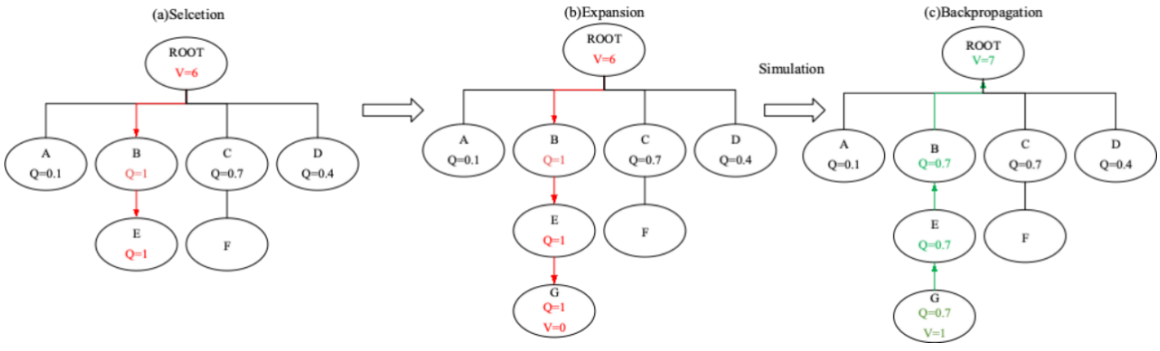

Figure 3: Payload Mutation Process.

Taking a state of the load '0+or%0btrue%0a' as an example, as shown in Figure 3(a), the Q value in each node represents its potential value, and V represents the number of visits. Assuming that the E node represents the load '+', in the selection stage, it is first randomly selected according to the weight in the child nodes of the ROOT node. B is selected because of its high weight due

to its number of visits of 0. Then, in the child nodes of B, E is selected again because of its number of visits of 0. In the expansion phase shown in Figure 3(b), because node E has not been fully expanded, a mutation strategy is randomly selected according to the grammar file to mutate it. Assume that the selected strategy is to replace '+' with '%0b' synonymously, and use '%0b' as the child node G of node E. The Q value and V value of node G are initialized. At this time, the load represented by the Monte Carlo tree is mutated to '0%0bor%0btrue%0a'. The simulation phase is to use the current load to test the WAF under test. According to the simulation results, whether the current load can make the WAF protection invalid; according to the simulation results, enter the back propagation link. If the simulation fails, the Monte Carlo is adjusted. As shown in Figure 3(c), the Q value of node G and its ancestor nodes is deducted, and the V value is increased by 1 visit number. In this way, the Monte Carlo tree can tend to other nodes with higher potential or unvisited nodes in the future selection phase, balancing exploration and utilization. At this point, WAFuzz will return to the selection phase and restart the four-phase cycle. The system will continue to explore new paths in the search tree, generate new mutation payloads and test them until a payload is generated that successfully disables WAF protection, completing the entire guided mutation process based on Monte Carlo tree search.

4. System Testing and Analysis

In order to evaluate WAFuzz effectively and objectively, several typical Web Application Firewall (WAF) were selected to test the implementation. A DVWA (Damn Vulnerable Web Application) (DigiNinja) environment was deployed behind the WAF for testing purposes. To assess the effectiveness of WAFuzz, this paper follows the process of fuzz testing, setting up comparison experiments for both the attack payload generation and mutation stages. These results are then compared and analyzed with the experimental results from other WAF testing systems, leading to the final evaluation conclusions.

4.1. Metrics and Parameters

The Protection Failure Rate (PFR) represents the proportion of malicious payloads that are not detected or blocked by the Web Application Firewall (WAF).

The Invalid Payload Rate (IPR) reflects the proportion of payloads in the test set that, although they successfully bypass the WAF, do not produce a valid attack or fail to achieve the desired effect.

Query Count (QC) represents the number of interactions with the WAF. Fewer queries indicate higher efficiency of the attack method.

4.2. Datasets and Related Tools

For the construction of the syntax file in this project, two public datasets were referenced: sqliv5 sqliv5 and Cross-site scripting (XSS) cheat sheet. To evaluate the quality of payload generation in WAFuzz, we use WAFuzz to randomly construct a dataset M containing 2,000 non-repetitive attack payloads. As tools like ML-Driven rely on machine learning techniques with models trained on specific datasets, they struggle to adapt to the testing scenarios created in this paper. Therefore, to compare with ML-Driven, we used the SQL injection syntax file from ML-Driven to generate a dataset M1 containing 2000 unique payloads. Since there is a lack of research testing WAF protection capabilities against XSS attacks, two open-source WAF testing tools—WAF-Bypass (digininja,

2025) and Gotestwaf (Wallarm, 2025) —were selected. The SQL injection and XSS test sets from these tools were combined to form datasets M2 and M3. It is worth noting that some tools mentioned in the related work were not included in the experimental comparison. Most of these tools are not open-source, making it difficult to reproduce their results, and they rely on pretrained machine learning models, which are not suitable for testing real-world WAF deployments. Therefore, they are not directly comparable in this study.

4.3. Experimental Environment

In the experiments, multiple WAF were tested, including ModSecurity, uuwaf, Safedog, Safeline, and Yunsuo, to evaluate their performance against various attack payloads. The detailed information about the WAF tested is provided in Table 3. All experiments were repeated 10 times, and the average results were reported.

Table 3: WAF Target Information.

WAF Under Test	Version Information
ModSecurity	V3.0.5 with CRS 3.2.3
ModSecurity	V3.0.5 with CRS 3.3.4
uuwaf	V2.6.0
Safedog	V4.0
Safeline	V15.2
Yunsuo	V3.1.20.15

4.4. Comparison Experiments and Result Analysis

(1)Attack Payload Generation Comparison Experiment

In the attack payload generation comparison experiment, to assess the effectiveness of payloads generated by WAFuzz, the attack payloads from datasets M, M1, M2, and M3 were sent to the six target WAF. The bypass success rates and invalid payload rates were analyzed, with the final results being the average of the six WAF tests. The results of the basic test case set PFR comparison experiment are shown in Table 4, and the IPR comparison experiment results are shown in Table 5.

From the data in Table 4, it is evident that WAFuzz significantly outperforms traditional test case sets in terms of Protection Failure Rate (PFR) for both SQL injection (SQLI) and Cross-Site Scripting (XSS). WAFuzz achieved a PFR of 48.80% for SQLI and 37.80% for XSS, far surpassing the performance of other test sets. This result demonstrates that WAFuzz, through grammar-based fuzz testing, can generate higher-quality test cases.

From Table 5, it can be observed that WAFuzz also excels in terms of Invalid Payload Rate (IPR). The IPR for SQLI and XSS in WAFuzz was 5.63% and 6.72%, respectively, which is significantly lower than the IPR of other test case sets such as M1 (11.44%), M2 (10.88%), and M3 (12.12%). The lower IPR indicates that fewer invalid payloads are generated by WAFuzz, and more payloads effectively bypass the WAF and achieve the intended attack effect, enhancing the validity of the test results.

(2)Attack Payload Mutation Comparison Experiment

To validate WAFuzz's performance in the mutation phase, we conducted comparison experiments with WAF-A-MOLE and SqlMap (Damele and Stampar). In the experiments, SqlMap used

Table 4: Basic Test Case Set PFR Comparison Experiment.

Basic Test Case Set	SQLI PFR	XSS PFR
M	48.80%	37.80%
M1	11.44%	-
M2	-	11.37%
M3	-	24.79%

Table 5: Basic Test Case Set IPR Comparison Experiment.

Basic Test Case Set	SQLI IPR	XSS IPR
M	5.63%	6.72%
M1	11.44%	-
M2	-	10.88%
M3	-	12.12%

its built-in payload obfuscation script to mutate the attack payload, randomly selecting a script until it successfully bypassed the WAF. Since both WAF-A-MOLE and SqlMap support only SQL injection attack bypasses, we also conducted experiments comparing WAFuzz using the traditional UCT strategy with WAFuzz using the UCT-Rand strategy. In this study, the primary performance metric is not merely execution time, but rather the number of interactions with the WAF. By guiding mutations through MCTS, WAFuzz reduces redundant requests and minimizes the risk of being blocked by real-world WAFs due to aggressive access behavior.

Table 6: Test Case QC Comparison Experiment.

Testing Tool	Query Count (QC)
WAF-A-MOLE	41
SqlMap	67
WAFuzz (UCT)	31
WAFuzz (UCT-Rand)	22

Table 6 presents the Query Count (QC) results for the three tools. WAFuzz with the UCT-Rand strategy achieved the best performance, requiring only 22 queries on average, compared to 31 for WAFuzz (UCT), 41 for WAF-A-MOLE, and 67 for SqlMap. This indicates that WAFuzz is more efficient in the mutation phase, generating effective payloads with fewer interactions.

(3) WAFuzz Test Results Analysis

To evaluate WAFuzz's performance against various WAF, Table 7 presents a comparison of WAFuzz with other tools, showing whether each tool was able to find failure scenarios in the tested WAF. Table 8 provides examples of effective attack payloads discovered by WAFuzz that can bypass ModSecurity (CRS 3.2.3).Since WAF-A-MOLE and SqlMap support only SQL injection attacks and do not test for XSS vulnerabilities, the comparison for XSS and RCE vulnerabilities in Table 7 uses Gotestwaf and WAF-bypass alongside WAFuzz.

As shown in Table 7, WAFuzz outperforms other tools by successfully bypassing multiple WAFs in both SQL injection and XSS tests, with the exception of ModSecurity (CRS 3.3.4). Other tools like WAF-A-MOLE, SqlMap, Gotestwaf, and WAF-bypass also failed to bypass ModSecurity. Ta-

Table 7: WAF Test Results.

WAF Under Test	SQLI			XSS		
	WAF-A-MOLE	SqlMap	WAFuzz	Gotestwaf	Waf-bypass	WAFuzz
ModSecurity	×	×	●	×	×	●
ModSecurity	×	×	●	×	×	×
uuwaf	●	×	●	●	●	●
Safedog	●	●	●	●	●	●
Safeline	×	×	●	●	●	●
Yunsuo	●	●	●	●	●	●

ble 8 lists sample payloads that successfully bypassed ModSecurity (CRS 3.2.3). While other WAFs were also bypassed, their non-open-source nature prevents disclosure of specific payloads. Analysis of these payloads reveals several noteworthy patterns.

Table 8: WAF Payload Layer Bypass Example.

Attack Type	Effective Attack Payload	Request Method
XSS	<a%0ahref%0a=%0a”jav%0Dascript%26colon;alert(1)”%0dx//XSS	GET
SQLI	1’%09or%0d‘if‘+0x1=1%09or%270	GET
SQLI	1.0and%09extractvalue(0x1,concat(0x7e%2c@@version,0x7e),0x1)–%0c”	JSON

In addition to the current experimental results, the proposed method has also successfully identified a number of effective payloads against various real-world commercial WAFs. These payloads will be disclosed publicly after the corresponding vulnerabilities are confirmed and patched.

5. Conclusion

This paper presents WAFuzz, a fuzzing-based system for evaluating the protection capabilities of WAFs against SQL Injection (SQLI) and Cross-Site Scripting (XSS) attacks. It integrates context-free grammar (CFG)-based payload generation with a Monte Carlo Tree Search (MCTS)-guided mutation strategy to ensure semantic validity and efficiently explore evasion paths. The improved MCTS algorithm further optimizes the mutation process for better testing performance.

Experimental results show that WAFuzz outperforms existing tools in discovering WAF weaknesses, achieving higher protection failure rates and lower invalid payload rates. It also significantly reduces query interactions, validating its effectiveness and efficiency in real-world testing scenarios.

References

A. Alanda, D. Satria, and H. A. Mooduto. Cross-site scripting (xss) vulnerabilities in modern web applications. In *2024 11th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, pages 270–276, 2024. doi: 10.1109/EECSI63442.2024.10776461.

M. Amouei, M. Rezvani, and M. Fateh. Rat: reinforcement-learning-driven and adaptive testing for vulnerability discovery in web application firewalls. *IEEE Transactions on Dependable and Secure Computing*, 19(5):3371–3386, 2021. doi: 10.1109/TDSC.2021.3095417.

D. Appelt, C. D. Nguyen, A. Panichella, et al. A machine-learning-driven evolutionary approach for testing web application firewalls. *IEEE Transactions on Reliability*, 67(3):733–757, 2018. doi: 10.1109/TR.2018.2805763.

B. Damele and M. Stampar. Sqlmap: Automatic sql injection and database takeover tool. sqlmap.org. Accessed 2025-04-16.

L. Demetrio, A. Valenza, G. Costa, et al. Waf-a-mole: evading web application firewalls through adversarial machine learning. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 1745–1752, 2020. doi: 10.1145/3341105.3373962.

DigiNinja. Dvwa (damn vulnerable web application). GitHub. Accessed 2025-04-16.

digininja. Damn vulnerable web application (dvwa). GitHub, 2025. Accessed 2025-04-16.

J. Esparza, P. Rossmanith, and S. Schwoon. A uniform framework for problems on context-free grammars. *arXiv preprint arXiv:2410.19386*, 2024. doi: 10.48550/arXiv.2410.19386.

G. Heyes. Cross-site scripting (xss) cheat sheet (2024 edition). PortSwigger, 2024. Accessed 2025-04-16.

A. Kumar, S. Dutta, and P. Pranav. Analysis of sql injection attacks in the cloud and in web applications. *Security and Privacy*, 7(3):e370, 2024. doi: 10.1002/spy2.370.

nidnogg. sqliv5-dataset: An expanded version of the kaggle sqliv3 sql injection dataset, using waf-a-mole. GitHub, 2023. Accessed 2025-04-16.

Z. Qu, X. Ling, and C. Wu. Autospear: Towards automatically bypassing and inspecting web application firewalls. Black Hat Asia Conference, 2022. Accessed 2025-04-16.

T. Someya, R. Yoshida, and Y. Oseki. Targeted syntactic evaluation on the chomsky hierarchy. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 15595–15605, 2024. doi: 10.18653/v1/2024.lrec-main.1356.

M. Tremante, D. Belson, and S. Zejnilovic. The state of application security in 2023. Cloudflare Blog, 2023. Accessed 2025-04-16.

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*, pages 347–357, 2013. doi: 10.1145/2483760.2483776.

R. Vallabhaneni, S. Pillai, S. A. Vaddadi, et al. Secured web application based on capsulenet and owasp in the cloud. *Indonesian Journal of Electrical Engineering and Computer Science*, 35(3):1924–1932, 2024. doi: 10.11591/ijeecs.v35.i3.pp1924-1932.

Wallarm. Gotestwaf: An open-source project to test wafs and api security solutions. GitHub, 2025. Accessed 2025-04-16.

M. Świechowski, K. Godlewski, B. Sawicki, et al. Monte carlo tree search: A review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562, 2023. doi: 10.1007/s10462-022-10228-y.