

A Python-based web application firewall for SQLi protection in heterogeneous web server environments

Muhammad Nur Yasir Utomo, Eddy Tungadi, Muh. Ahyar

Department of Informatics and Computer Engineering, Politeknik Negeri Ujung Pandang, Makassar, Indonesia

Article Info

Article history:

Received Oct 16, 2025

Revised Mar 31, 2026

Accepted Aug 28, 2026

Keywords:

Structured query language injection

Web application firewall

Web environments

Web security

Web server

ABSTRACT

Structured query language injection (SQLi) remains one of the most critical threats to web applications. Conventional web application firewall (WAF) such as ModSecurity provide protection but face limitations in heterogeneous environments and often require additional configurations on web servers. This issue is problematic because adding new configurations demands considerable effort and may disrupt stable services. To address this challenge, this research proposes PyWAF, a Python-based WAF developed as a reverse proxy that offers flexible protection across heterogeneous web servers against SQLi attacks. PyWAF employs a weighted core rule set (CRS) with a scoring mechanism and gate flags to detect various SQLi patterns. The proposed method was evaluated in compatibility testing and reliability testing across three popular web server environments namely Apache, Nginx, and OpenLiteSpeed. The reliability evaluation utilized 30 SQLi payloads across three different scenarios, namely GET, POST, and API payload. The results of compatibility test show that the proposed method achieved seamless integration with all web servers without requiring additional server configuration. In term of reliability, PyWAF successfully blocked 100% of SQLi attacks, while ModSecurity blocked 88.9%, demonstrating an improvement of 11.1% in SQLi detection capability.

This is an open access article under the [CC BY-SA](#) license.



Corresponding Author:

Muhammad Nur Yasir Utomo

Department of Informatics and Computer Engineering, Politeknik Negeri Ujung Pandang

Perintis Kemerdekaan Rd., KM 10, Tamalanrea, Makassar, Indonesia

Email: yasirutomo@poliupg.ac.id

1. INTRODUCTION

Web application security is a critical aspect as cyberattack threats continue to increase. One of the most common and dangerous threats is the structured query language injection (SQLi) attack [1]. This attack exploits input validation vulnerabilities to inject malicious structured query language (SQL) commands, which may lead to data leakage or even full database compromise [2]. SQLi remains a popular attack vector frequently used to compromise website security [3], and has consistently ranked among the most severe web vulnerabilities each year [4]. To mitigate such threats, many organizations implement a web application firewall (WAF) as an additional defensive layer in front of their web applications [5], [6].

WAF has been widely adopted as a defensive mechanism to filter and block malicious traffic before it reaches the web server [7]. One of the most popular and widely used WAF is ModSecurity, which can be employed to prevent SQLi attacks [8]. However, ModSecurity still has two major limitations which are integration challenges within heterogeneous web backend environments [9] and detection errors, particularly when dealing with new or evolving attack patterns [10]. Recent studies have shown that ModSecurity's default configuration achieves only 63.85% to 66.82% true positive (TP) rate at 1% false positive rate, and

performance further degrades to 55.55% when facing adversarial SQLi attacks [11]; this highlights significant gaps in detection effectiveness.

To enhance detection capabilities, several studies have proposed various improvements, such as adding new rules through proxy grammar integration [3], configuring filtering policies to improve effectiveness [12], integrating rules with autoencoders, and modifying usability aspects [13]. Nevertheless, the detection capability of SQLi attacks, particularly against new or emerging attack patterns, remains an ongoing challenge that has not been fully resolved.

In addition to challenges related to new attack patterns, ModSecurity also faces issues concerning integration within heterogeneous backend server infrastructures [14]. In practice, web applications often operate on diverse backend platforms such as Apache, Nginx, and OpenLiteSpeed, each with distinct configurations and architectures [15], [16]. ModSecurity's dependency on server-specific modules makes its implementation inconsistent across platforms. For instance, ModSecurity can be easily deployed on Apache web servers [9], [17], but requires additional integration through connectors or external modules on other web servers, increasing configuration complexity and potentially disrupting the stability of existing services [18].

This server-dependency challenge is not unique to ModSecurity but represents a fundamental limitation of most conventional WAF solutions. Existing WAF implementations typically rely on server-specific modules or connectors, requiring tailored configurations for each web server platform [19]. While previous research has focused primarily on improving detection algorithms or rule sets, few studies have addressed the practical challenge of deploying WAFs across heterogeneous web server environments. The lack of server-agnostic WAF solutions creates deployment complexity for organizations operating mixed infrastructure, where different applications may run on different web server platforms [20].

This study fills this gap by developing and validating PyWAF, a Python-based WAF designed for heterogeneous backend environments. Unlike conventional server-dependent WAFs, PyWAF utilizes a reverse proxy architecture that operates independently of the backend web server type, eliminating the need for server-specific modules or configurations while maintaining robust SQLi detection capabilities. This approach ensures consistent protection and simplifies deployment, allowing organizations to strengthen web security without altering the stability or structure of their existing systems.

The primary objective of this research is to develop and evaluate a WAF solution that achieves superior SQLi detection accuracy while maintaining deployment flexibility across heterogeneous web server infrastructures. The specific objectives and contributions of this research are:

- Development of a server-agnostic reverse proxy architecture enabling flexible SQLi protection across heterogeneous web server environments.
- Design of weighted core rule set (CRS) with scoring mechanism capable of detecting various SQLi patterns across GET, POST, and API payload attack scenarios.
- Empirical evaluation demonstrating that the proposed WAF can be seamlessly integrated into heterogeneous backend environments while maintaining strong SQLi detection performance.

Through these contributions, this study offers a flexible and practical security solution to strengthen web protection against SQLi threats, particularly in heterogeneous web infrastructure environments.

2. METHOD

This section presents the proposed method system design, experimental environment configuration, and evaluation method for assessing compatibility of proposed method across heterogeneous web server platforms and assessing reliability of SQLi detection.

2.1. System design

The proposed WAF, named PyWAF (<https://github.com/yasirutomo/pywaf>), is designed as a proxy for the web server, ensuring that all traffic passes through the WAF before reaching the backend server. Figure 1 illustrates the PyWAF architecture and data flow design, where the WAF is positioned as a reverse proxy serving as the primary defense layer that filters incoming HTTP/HTTPS requests from users before forwarding them to the backend servers. The backend environment consists of three web servers namely Nginx, Apache, and OpenLiteSpeed which represent a heterogeneous infrastructure protected by the WAF. During the filtering process, PyWAF inspects each request using its CRS. Detected anomalies or blocked requests are recorded in the log files, while legitimate requests are forwarded to the backend servers and processed by their respective applications and databases.

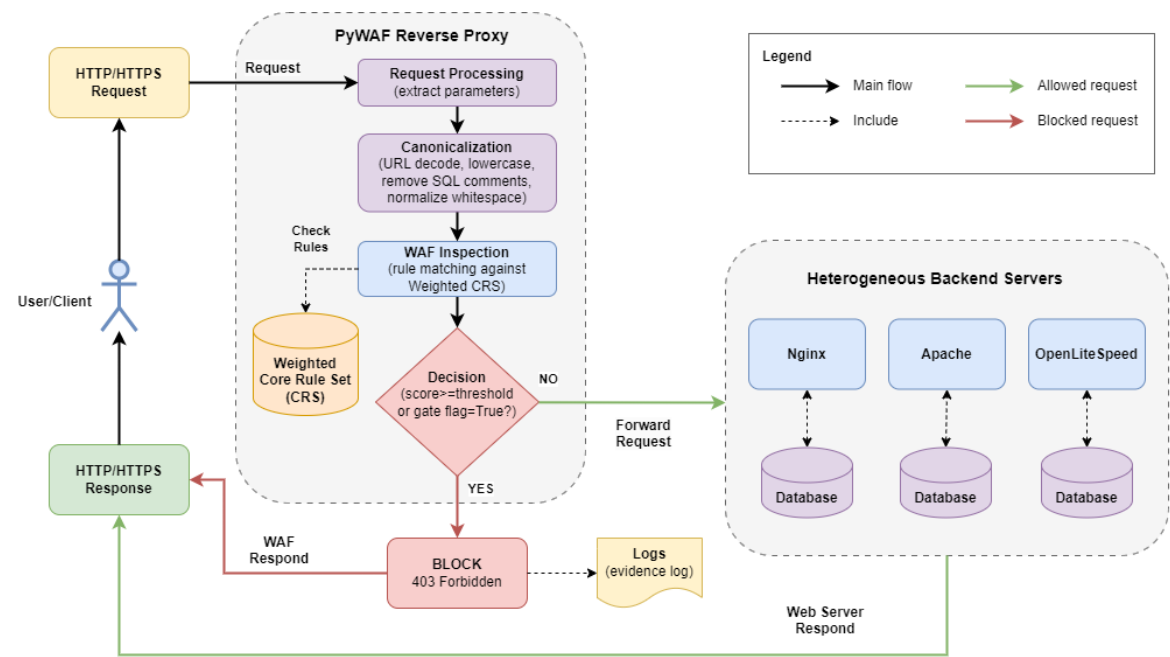


Figure 1. Proposed PyWAF architecture as a reverse proxy, covering request processing, canonicalization, rule matching against the weighted CRS, scoring/decision in heterogeneous backend servers

2.2. System environment

The experimental environment was built on a virtual infrastructure consisting of five virtual machines (VMs) running Ubuntu Server. These five VMs were categorized into two types, namely WAF VMs and backend VMs. The WAF VMs consisted of two machines, one running the proposed PyWAF and another running ModSecurity, which served as the baseline for comparison. The configuration details of the WAF VMs are presented in Table 1.

Table 1. WAF VM environment			
No.	VM name	Role	Config
1	Server-1	Python-based WAF (PyWAF)	Docker container
2	Server-2	ModSecurity WAF	Host-based config

Meanwhile, the remaining three VMs were used to simulate backend servers with varying configurations to evaluate the compatibility of the WAF across different types of web servers. The configuration details of each backend server are presented in Table 2.

Table 2. Backend VM environment					
No.	VM name	Role	Web server	Hosted web	Web name
1	Server-3	Backend web	OpenLiteSpeed	Wordpress	Web 1
2	Server-4	Backend web	Apache	Native PHP	Web 2
3	Server-5	Backend web	Nginx	Web API	Web 3

2.3. Core rule set development

CRS is a collection of pre-defined rules designed to detect and block SQLi attacks [21], [22]. The CRS in PyWAF is designed to detect SQLi attack patterns using a collection of regular expression-based rules. Each rule is structured in the format of rule_name, regex_pattern, score, and gate_flag. Figure 2 illustrates the example of CRS stored in PyWAF.

The implementation is represented in Python code format, where each rule consists of a unique identifier, regular expression pattern for detection, assigned score, and gate flag indicator. The assigned score was designed considering two main factors, namely potential impact if the attack is successfully executed and

false positive risk. Database destruction operations received the highest score (6) as they can cause immediate and irreversible damage to system integrity. Data extraction patterns were assigned scores of 4-5 as they enable attackers to steal sensitive information including credentials and personal data. Query manipulation patterns received scores of 2-3 as they typically serve as preparatory steps for more sophisticated attacks. Auxiliary indicators were assigned the lowest scores (1-2) because they may occasionally appear in legitimate queries, thus, lower weights minimize false positives while still contributing to cumulative detection when combined with other patterns. This graduated scoring approach balances detection sensitivity with precision, reducing the risk of blocking legitimate requests.

```

_Sqli_RULES_SPEC = [
    ("UNION_SELECT",      r"\bunion(?:[\\s\\/*!-]{0,8})all?(?:[\\s\\/*!-]{0,8})select\b",      5, True),
    ("SELECT_FROM",      r"(\<![a-z0-9_])select\s+.{0,200}?\s+from\b",      3, False),
    ("ORDER_BY_NUM",     r"\borders+by\s+d+\b",      2, False),
    ("TAUTOLOGY_EQ",     r"(\<^|^[^a-z0-9_])(?:or|and)\s+([\\s]*'?'1?'?'\s*=\s*'?'1?'?",      4, False),
    ("INFO_SCHEMA",     r"\binformation_schema\b",      4, False),
    ("INTO_OUTFILE",     r"\binto\s+outfile\b",      5, True),
    ("SLEEP_FN",        r"\b(?:sleep|pg_sleep)\s*(\\s*d+\\s*)|\bwaitfor\s+delay\b",      4, True),
    ...,
    ("STACKED_DDL",     r";\s*(?:drop|truncate)\b",      6, True),
    ("DML_DELETE",      r"\bdelete\s+from\b",      4, False),
    ("DML_INSERT",      r"\binsert\s+into\b",      3, False),
    ("ORDER_BY_FUNC",   r"\borders+by\b.{0,120}\b(?:sleep|pg_sleep|benchmark|md5\s*\\()",      5, True),
    ("DROP_FUNCTION",   r"\bdrop\s+function\b",      6, True),
    ("UPSERT_SELECT",   r"\binsert\s+into\b[\\sS]{0,120}\bselect\b",      4, True),
    ("DELETE_USING",    r"\bdelete\s+from\b\s+w+(?:\s+using\b|\s+where\b)",      3, False),
]

```

Figure 2. Weighted CRS for SQLi detection in PyWAF

In total, the complete CRS consists of 54 rules targeting various SQLi patterns, with severity scores ranging from 1 to 6 based on attack impact and false positive risk. Among these, 15 rules are marked with gate flags for immediate blocking of high-confidence attack indicators such as DROP, SLEEP, and UNION SELECT combinations. The cumulative score threshold is set to 1, meaning any request matching at least one suspicious pattern triggers blocking unless it is a low-scored pattern without a gate flag. This low threshold prioritizes detection sensitivity, with the gate flag mechanism providing precision for critical patterns. The complete list of CRS rules is available in the PyWAF source code repository on GitHub (<https://github.com/yasirutomo/pywaf/blob/main/waf/app-v2.2.py>).

All the rules are compiled and stored in an array structure for efficient pattern matching during request inspection. Table 3 presents concrete examples of these CRS rules with actual SQLi payloads, assigned scores, and their attack purposes.

Table 3. Examples of CRS rules with SQLi queries

No.	CRS rule name	Example query	Score	Gate flag	Description/purpose
1	UNION_SELECT	admin' UNION SELECT NULL, 'admin', --	5	True	Data extraction by combining query results to inject fake credentials and bypass authentication.
2	DROP_FUNCTION	1'; DROP FUNCTION validate_user; --	6	True	Database destruction by removing stored functions, disabling authentication mechanisms.
3	TAUTOLOGY_EQ	admin' OR 1=1 --	4	False	Authentication bypass using always-true conditions to access system without valid credentials.
4	SLEEP_FN	product_id=1 AND IF(SUBSTRING(password,1,1)='a', SLEEP(5), 0)	4	True	Blind SQLi via time delays to extract data character by character through response timing.
5	ORDER_BY_NUM	id=1 ORDER BY 5 --	2	False	Column enumeration to determine table structure count for preparing UNION-based attacks.

As shown in Table 3, each rule is assigned a weight (score) representing the severity or potential impact of the detected SQLi pattern. The scoring was determined based on the risk level and execution potential of each SQL keyword or function if exploited by an attacker. Patterns associated with critical

actions such as database modification or system command execution (e.g., `DROP_FUNCTION`) were assigned the highest score (6), indicating high-risk operations that can directly compromise data integrity or system availability. Union-based injection patterns (e.g., `UNION_SELECT`) and time-based blind injection patterns (e.g., `SLEEP_FN`) were assigned scores of 5 and 4, respectively, as they enable direct data extraction or information gathering attacks. Query manipulation patterns such as `ORDER_BY_NUM` were assigned lower scores (2) as they primarily serve as reconnaissance techniques to prepare for more advanced attacks.

A request is blocked when the cumulative score reaches or exceeds the predefined threshold of 1, set deliberately low to maximize detection sensitivity. Additionally, some rules are marked with a `gate_flag` set to True, indicating that a single match is sufficient to trigger immediate blocking regardless of the cumulative score. Rules with True `gate_flag` (such as `UNION_SELECT`, `DROP_FUNCTION`, and `SLEEP_FN` shown in Table 3) represent high-confidence SQLi indicators that warrant immediate blocking. This combined scoring and gating approach in the developed CRS provides flexibility in blocking decisions while maintaining high detection accuracy.

2.4. Deployment

To implement the CRS developed in the previous section, the proposed WAF (PyWAF) was implemented using the Python programming language. Python was selected due to its extensive support for string manipulation, regular expression processing, and logging mechanisms, which are essential for implementing rule-based inspection. Moreover, Python provides flexibility for rapid prototyping, and through Docker containerization, it ensures high portability and seamless deployment across different environments.

The CRS logic was integrated into the WAF as a global component that performs real-time request inspection. The filtering workflow applied by the CRS in PyWAF is presented in Algorithm 1, which illustrates the sequential process of canonicalization, rule matching, scoring, and decision-making during request inspection.

Algorithm 1. Request filtering flow with weighted CRS

Input: HTTP request req
Output: Decision $\in \{\text{BLOCK}, \text{FORWARD}\}$

```

1 Initialize total_score  $\leftarrow 0$ 
2 Initialize evidence  $\leftarrow \emptyset$ 
3 Initialize gated_flag  $\leftarrow \text{false}$ 
4 Initialize threshold  $\leftarrow 1$ 
  // Inspection stage
5 foreach field in request do
6   canon  $\leftarrow \text{canonicalize}(\text{field})$ 
7   foreach rule in CRS do
8     if rule.regex matches canon then
9       total_score  $\leftarrow \text{total\_score} + \text{rule.score}$ 
10      evidence  $\leftarrow \text{evidence} \cup \text{rule.id}$ 
11      if rule.gate = true then
12        gated_flag  $\leftarrow \text{true}$ 
  // Decision stage
13 if gated_flag = true or total_score  $\geq$  threshold then
14   Write evidence
15   return BLOCK
16 else
17   Forward request to backend
18   return FORWARD

```

Algorithm 1 illustrates that the request filtering process begins with the extraction of request parameters, which may include query parameters, form data, or JSON bodies. Each extracted value is normalized through a canonicalization process consisting of URL decoding, lowercasing, SQL comment removal, and whitespace normalization. This step is crucial to mitigate the effects of obfuscation techniques commonly used by attackers.

The standardized values are then matched against the CRS patterns. When a match is found, a local score is added, and the match information is recorded as evidence. After all inputs are inspected, the total score is computed and compared to the threshold. If the score exceeds the threshold or if a gate flag is triggered, the request is immediately blocked by returning a 403 forbidden status, while detailed evidence, triggered rules, and requested input parameter are logged in the evidence log. Otherwise, the request is forwarded to the backend server.

In terms of computational complexity, let N denote the number of extracted request parameters, M the number of CRS rules ($M = 54$), and L the average length of a parameter value. For each parameter, canonicalization requires $O(L)$ time, and matching against all CRS rules requires $O(M \times L)$ time using regex-

based pattern matching. Since this process is repeated for all N parameters, the overall computational complexity per request is $O(N \times M \times L)$. As M is a fixed constant independent of traffic volume, the effective complexity scales linearly with request size ($O(N \times L)$), supporting the lightweight nature of the proposed rule matching and scoring mechanism.

2.5. Evaluation

The system performance evaluation was conducted in two main stages. The first stage focused on compatibility evaluation, aimed at assessing the WAF's ability to operate in front of heterogeneous backend servers. The testing was carried out by deploying WAF with three types of web servers, namely Apache running a native PHP application, Nginx hosting a Web API application, and OpenLiteSpeed serving WordPress. At this stage, the primary indicator observed was whether the WAF could operate without causing operational disruptions or requiring significant configuration changes on each backend server.

The second stage involved evaluating the reliability of the WAF in preventing SQLi attacks. The testing included various attack scenarios, covering GET requests, POST form data, and API payloads in JSON format. An illustration of these attack scenario variations is presented in Table 4.

Table 4. SQLi test scenario

No	Scenario	Method	Query example
1	URL access test	URL (GET): attacks embedded in web URL [23], [24]	<code>https://contoh.com/product?id=1' AND EXTRACTVALUE(1,CONCAT(0x0a, USER())) --</code>
2	Form data test	Form data (POST): attacks embedded in web form input [25]	<code>username=admin' --&password=test</code>
3	API payload test	API payload: attacks inserted into API requests [26]	<code>{ "username": "admin%27%20OR%201=1%20--", "password": "test" }</code>

In each scenario, the SQLi attacks included five common injection techniques, namely tautology-based (Boolean-based), union-based, error-based, blind SQL, and time-based [27], [28]. The queries are publicly available in GitHub (<https://github.com/yasirutomo/sql-injection-test>). Table 5 shows the detail of SQLi payloads used.

Table 5. Detail of SQLi payload for testing

No.	Attack technique	Number of test queries	Example queries	Description
1	Tautology-based (Boolean-based)	6	<code>OR 1=1, AND 'a'='a'</code>	Converts a conditional expression into a tautology, causing the condition to always evaluate to true. Commonly used to bypass authentication or broaden query results [29].
2	Union-based	6	<code>UNION SELECT NULL, username, password FROM users</code>	Exploits the UNION operator to combine attacker-controlled SELECT results with legitimate application query. Effective when the application directly displays query output [30].
3	Error-based	6	<code>' AND (SELECT 1 FROM (SELECT COUNT(*) , CONCAT(...) GROUP BY x)a)</code>	Triggers database errors or exploits verbose error messages to disclose sensitive data through returned error output [31].
4	Blind SQL	6	<code>' AND SUBSTRING(database(),1,1)='a</code>	Attacker probes boolean conditions and observes changes in application behavior to extract information [32].
5	Time-based	6	<code>'; IF(1=1, SLEEP(5), 0) -</code>	Inferential technique using delay mechanisms (e.g., SLEEP(), WAITFOR) to extract data by observing differences in response times when conditions are true or false [33], [34].

Based on Tables 4 and 5, each scenario consists of 30 SQLi payloads. These 30 payloads were tested across three different scenarios (GET, POST, and API payload), resulting in a total of 90 SQLi payloads evaluated in this study. The test results were recorded and analyzed to measure the recall (sensitivity) of SQLi detection using (1):

$$Recall(s) = \frac{TP(s)}{TP(s) + FN(s)} \quad (1)$$

where TP represents the SQLi payloads that were successfully blocked, FN represents the SQLi payloads that bypassed the WAF, and $s \in \{non - WAF, ModSecurity, PyWAF\}$. After obtaining the recall values for each condition, the comparative security improvement between non-WAF, ModSecurity, and the proposed WAF can be calculated using (2):

$$\Delta Recall(w, x) = Recall(w) - Recall(x) \quad (2)$$

where w represents the proposed WAF and ModSecurity, which are compared against x (non-WAF). Ultimately, the evaluation results in this study demonstrate the degree of security improvement achieved by NonWAF, ModSecurity, and PyWAF.

3. RESULTS AND DISCUSSION

This section presents the compatibility results, SQLi detection evaluation, comparative analysis and discussion of the proposed PyWAF, along with its comparison to ModSecurity as the baseline across three different attack scenarios namely, GET, POST, and API payload.

3.1. Compatibility results

Compatibility evaluation was conducted by placing the WAF in front of three different backend servers which were Apache with a WordPress application, Nginx with a Web API application, and OpenLiteSpeed with a native PHP application. The integration of the WAF with each backend was tested without additional configuration on the web server side, in order to simulate real-world conditions where many web applications operate using heterogeneous web servers, technologies, and configurations. Adding new configurations to existing web servers often requires significant effort and may potentially disrupt the stability of already operational web services.

Table 6 presents the compatibility test results, showing that PyWAF operated successfully across all backend servers without requiring additional configuration modifications on the backend web servers. All legitimate requests were successfully forwarded to the applications without causing operational disruptions. This server-agnostic capability is particularly valuable for organizations operating heterogeneous web infrastructures, as it eliminates the need for server-specific WAF installations and reduces deployment complexity. In contrast, ModSecurity requires server-specific modules and can only protect Apache-based servers in our test environment, limiting its applicability in diverse infrastructure scenarios. Therefore, PyWAF offers higher flexibility, as it can protect various types of backend servers through a reverse proxy approach without altering existing web configurations.

Table 6. Compatibility results

No	WAF	Web server		
		OpenLiteSpeed (web 1)	Apache (web 2)	Nginx (web 3)
1	ModSecurity	-	✓	-
2	PyWAF	✓	✓	✓

3.2. Structured query language injection detection results

The system reliability evaluation was conducted by testing 30 SQLi queries representing five common techniques (tautology-based, union-based, error-based, blind SQL, and time-based) across GET, POST, and API Payload scenarios. Each query for every scenario was executed using Postman, adjusting the HTTP method and request body to match the scenario. The target application was a native PHP web application intentionally made vulnerable to SQLi.

3.2.1. GET scenario test results

The tests were conducted using Postman, where each SQLi payload was embedded in the URL query string. This setup aimed to simulate attacks exploiting GET parameters commonly found in web applications. The detection results for this scenario are summarized in Table 7.

Table 7 shows that the non-WAF setup was affected by all 30 SQLi payloads. ModSecurity managed to block 27 payloads (90%), while three attacks still succeeded in passing through. In contrast, PyWAF successfully blocked all attacks (100%) in the controlled test environment. These demonstrated PyWAF's effectiveness compared to the baseline in providing protection against various categories of SQLi attacks.

Table 7. SQLi detection on GET scenario

No	SQLi category	Number of query	Non-WAF			ModSecurity			PyWAF		
			Pass	Block	%	Pass	Block	%	Pass	Block	%
1	Tautology-based	6	6	0	0.00	2	4	66.67	0	6	100.00
2	Union-based	6	6	0	0.00	0	6	100.00	0	6	100.00
3	Error-based	6	6	0	0.00	1	5	83.33	0	6	100.00
4	Blind SQL	6	6	0	0.00	0	6	100.00	0	6	100.00
5	Time-based	6	6	0	0.00	0	6	100.00	0	6	100.00
	Total	30	30	0	0.00	3	27	90.00	0	30	100.00

3.2.2. POST scenario test results

The tests were performed using Postman with the POST method, where each SQLi payload was placed in the request body as form-data. This configuration replicates injection attempts through POST-based input fields commonly used in web forms. The detection results for this scenario are presented in Table 8.

Table 8. SQLi detection on POST scenario

No	SQLi category	Number of query	Non-WAF			ModSecurity			PyWAF		
			Pass	Block	%	Pass	Block	%	Pass	Block	%
1	Tautology-based	6	6	0	0.00	3	3	50.00	0	6	100.00
2	Union-based	6	6	0	0.00	0	6	100.00	0	6	100.00
3	Error-based	6	6	0	0.00	1	5	83.33	0	6	100.00
4	Blind SQL	6	6	0	0.00	0	6	100.00	0	6	100.00
5	Time-based	6	6	0	0.00	0	6	100.00	0	6	100.00
	Total	30	30	0	0.00	4	26	86.67	0	30	100.00

Table 8 presents the results of SQLi testing for the POST scenario. The non-WAF web server was affected by all 30 SQLi payloads. ModSecurity demonstrated better performance by blocking 26 payloads (86.7%), although four attacks, particularly in the tautology-based and error-based categories, still managed to pass through. In contrast, PyWAF successfully blocked all payloads in the dataset (100%). These results demonstrate that PyWAF is more effective in handling SQLi attacks in the POST request scenario.

3.2.3. API payload scenario test results

The tests were carried out using Postman with the POST method, where each SQLi payload was embedded in the raw JSON-formatted request body. This scenario emulates attacks targeting API endpoints that transmit data in JSON format. The detection results for this scenario are summarized in Table 9. The result shows that none of the 30 queries were blocked when no WAF was used. ModSecurity was able to block 27 out of 30 payloads (90%), leaving three attacks that still passed through. In contrast, PyWAF successfully blocked 100% of SQLi queries dataset, demonstrating its effectiveness in handling SQLi attack variations under the API/JSON scenario.

Table 9. SQLi detection on API payload scenario

No	SQLi category	Number of query	Non-WAF			ModSecurity			PyWAF		
			Pass	Block	%	Pass	Block	%	Pass	Block	%
1	Tautology-based	6	6	0	0.00	2	4	66.67	0	6	100.00
2	Union-based	6	6	0	0.00	0	6	100.00	0	6	100.00
3	Error-based	6	6	0	0.00	1	5	83.33	0	6	100.00
4	Blind SQL	6	6	0	0.00	0	6	100.00	0	6	100.00
5	Time-based	6	6	0	0.00	0	6	100.00	0	6	100.00
	Total	30	30	0	0.00	3	27	90.00	0	30	100.00

Notably, the detection results for GET scenario (Table 7) and API payload scenario (Table 9) are identical, with both scenarios showing the same three attacks bypassing ModSecurity in the tautology-based and error-based categories. This consistency indicates that ModSecurity exhibits similar detection behavior across different request types, with persistent weaknesses in handling specific tautology-based and error-based injection patterns. These patterns typically involve always-true conditions (e.g., OR 1=1) and error message exploitation that ModSecurity's generic rules struggle to identify effectively. Conversely, PyWAF maintained 100% detection across all scenarios, demonstrating robust protection regardless of the request method used. This reliability is critical for modern web applications that expose multiple attack surfaces through different input channels (URL parameters, form data, and API endpoints).

3.3. Comparative analysis

A comparative analysis was conducted by calculating the sensitivity (recall) and the security improvement achieved by ModSecurity compared to PyWAF. For each test scenario, the TP value represents the number of SQLi queries successfully blocked, while the false negative (FN) value represents the number of queries that were not blocked. The comparison results of SQLi detection recall/sensitivity are presented in Table 10.

Table 10. Comparison of SQLi detection sensitivity

System	Total SQLi query	TP	FN	Recall (%)
Non-WAF	90	0	90	0
ModSecurity	90	80	10	88.9
PyWAF	90	90	0	100

Table 10 shows that PyWAF's SQLi detection sensitivity outperforms ModSecurity, with values of 100% and 88.9%, respectively. The improvement represents 10 additional attacks blocked. In a production environment processing thousands of requests daily, this improvement could translate to preventing dozens of successful SQLi attempts that would otherwise compromise sensitive data. The 10 attacks that bypassed ModSecurity included critical injection types such as authentication bypass (tautology-based) and data extraction (error-based), which could lead to unauthorized access or data breaches if left undetected. This comparative performance is further visualized in Figure 3, which illustrates the security improvement ($\Delta Recall(w, x)$) achieved by each WAF system compared to the non-WAF baseline.

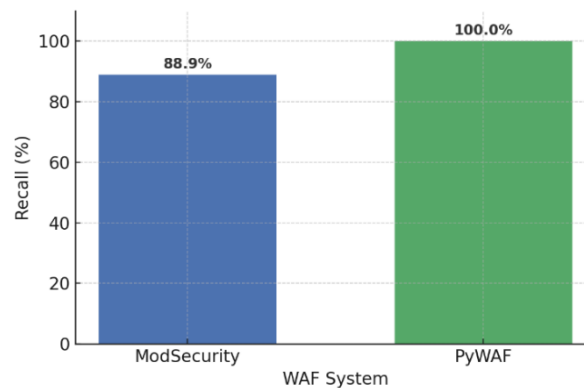


Figure 3. Comparison of security improvement against SQLi attacks

Based on Figure 3, the proposed WAF, PyWAF, outperforms ModSecurity by 11.1% in detecting and preventing SQLi attacks. These results demonstrate that PyWAF successfully addresses both primary objectives namely achieving superior detection accuracy (100% vs. 88.9%) and providing server-agnostic deployment across heterogeneous web infrastructures without configuration overhead.

3.4. Discussion

The evaluation results indicate that PyWAF successfully addresses the limitations of conventional WAFs such as ModSecurity in two key aspects which are compatibility and reliability. In terms of compatibility, PyWAF is capable of protecting various backend servers, including Apache, Nginx, and OpenLiteSpeed, without requiring additional host configurations in existing web server. This server-agnostic capability stems from the fundamental difference between reverse proxy and host-based architectures. As a reverse proxy, PyWAF operates as an independent intermediate layer that intercepts HTTP/HTTPS traffic at the network level before it reaches backend servers.

In contrast, host-based WAFs like ModSecurity require deep integration through server-specific connectors (e.g., ModSecurity on Apache uses mod_security module or Nginx requires a separately compiled ModSecurity-nginx connector). Each web server platform has distinct architectures, necessitating customized integration for each platform. This tight coupling creates compatibility constraints evident in ModSecurity's inability to protect OpenLiteSpeed and Nginx in our environment without additional compilation and configuration. PyWAF's reverse proxy approach avoids these issues by inspecting requests at the HTTP protocol level and forwarding legitimate traffic via standard HTTP proxying that all servers support natively.

This eliminates the need for server-specific modules or configuration changes, enabling seamless deployment across heterogeneous infrastructures. The processing overhead remains minimal as PyWAF performs efficient pattern matching without deep server integration. This architectural advantage makes the reverse proxy approach better suited for mixed backend infrastructures.

In terms of reliability, PyWAF achieved a 100% detection rate across all tested SQLi scenarios, including GET, POST, and API payload, using the evaluation dataset. In contrast, ModSecurity allowed 10 payloads (11.1%) to bypass detection, primarily in the tautology-based and error-based categories. This performance difference stems from PyWAF's dual-mechanism detection approach. The weighted scoring system (values 1-6 based on attack severity) enables cumulative evidence aggregation across multiple pattern matches within a single request, while gate flags provide immediate blocking for high-confidence indicators like DROP or UNION SELECT combinations.

In contrast, ModSecurity's default CRS relies on individual rule matching without weighted scoring, requiring each rule to independently trigger a block decision. The 10 bypassed attacks, particularly tautology-based ('1'='1') and error-based injections, succeeded against ModSecurity because individual pattern fragments did not meet blocking criteria, whereas PyWAF's cumulative scoring detected the anomalous combination of suspicious elements. This weighted approach with gating provides more comprehensive protection against sophisticated attacks that distribute malicious content across multiple parameters or employ obfuscation techniques. The 11.1% improvement demonstrates that this strategy effectively identifies attack patterns that evade generic rule-based detection while maintaining low false positive rates.

Beyond this comparison with ModSecurity, PyWAF's static rule-based design also offers higher explainability compare to machine learning (ML) approaches. Each detection decision in PyWAF can be directly traced to a specific rule and its assigned score, whereas ML/DL models, particularly deep learning architectures, often operate as black boxes that are difficult to audit. PyWAF also incurs lower computational overhead, since detection relies on lightweight regex-based pattern matching without the training or inference costs associated with ML/DL models. However, PyWAF's static rules require manual updates to address novel attack patterns, whereas ML/DL approaches can potentially generalize to previously unseen SQLi variants. This adaptability trade-off is one of several limitations that must be acknowledged in this study.

Other limitations relate to the evaluation method and scope. First, the evaluation was conducted using a controlled dataset of five types of SQLi payloads with a total of 30 queries in each scenario, which may not fully represent the complexity and diversity of real-world attack patterns encountered in production systems. Additionally, false positive rates were not systematically measured, evaluation with diverse legitimate traffic datasets is needed to quantify the trade-off between detection sensitivity and operational usability. Validation against larger standardized test suites such as OWASP SQLi test cases would provide more comprehensive assessment of detection capabilities. Second, a detailed performance benchmark under high traffic conditions has not yet been conducted and remains part of future work. These limitations suggest that while PyWAF demonstrates promising results in controlled settings, further validation with larger datasets, real-world traffic patterns, and performance evaluations under high traffic conditions is necessary to confirm its effectiveness and reliability for widespread adoption.

4. CONCLUSION

This study has proposed and implemented PyWAF, a Python-based WAF that functions as a reverse proxy for backend web servers against SQLi attacks. PyWAF implements a weighted CRS to score the possibility of whether a request is an SQLi attack or not. Based on evaluation conducted across various backend servers (Apache, Nginx, and OpenLiteSpeed), it achieved a 100% detection rate against 30 SQLi payloads across three different scenarios, namely GET, POST, and API payload (90 payloads in total). In contrast, ModSecurity achieved a recall of 88.9% and still allowed several attacks to pass through, particularly in the tautology-based and error-based categories. These results demonstrate: i) the advantage of the proposed reverse proxy-based architecture in providing flexible protection for heterogeneous web server environments, ii) seamless compatibility across Apache, Nginx, and OpenLiteSpeed without requiring server-specific configurations, and iii) a notable 11.1% improvement in security against SQLi attacks.

Although these results demonstrate PyWAF's advantages in terms of compatibility and detection reliability within the tested scope, the study is still limited to: i) controlled testing scenarios which may not fully represent production attack diversity, ii) a CRS that relies on static rule-based detection, which may become less effective against evolving or previously unseen SQLi techniques, and iii) the absence of a component-level ablation study on the system's individual components. Therefore, further validation using real-world traffic, larger datasets, broader interoperability testing across additional backend technologies and component-level ablation studies would strengthen these findings. Future research may also incorporate adaptive machine learning and anomaly detection techniques, as well as external threat intelligence feeds, to support automatic rule evolution against emerging and polymorphic SQLi attacks.

FUNDING INFORMATION

This work is supported by the Ministry of Higher Education, Science, and Technology of the Republic of Indonesia under the Fundamental Research scheme, with contract number 190/C3/DT.05.00/PL-BATCH II/2025.

AUTHOR CONTRIBUTIONS STATEMENT

This journal uses the Contributor Roles Taxonomy (CRediT) to recognize individual author contributions, reduce authorship disputes, and facilitate collaboration.

Name of Author	C	M	So	Va	Fo	I	R	D	O	E	Vi	Su	P	Fu
Muhammad Nur Yasir Utomo	✓	✓	✓	✓	✓	✓		✓	✓	✓			✓	✓
Eddy Tungadi			✓			✓		✓		✓	✓	✓		✓
Muh. Ahyar		✓		✓		✓	✓		✓	✓	✓		✓	

- C : Conceptualization
M : Methodology
So : Software
Va : Validation
Fo : Formal analysis
- I : Investigation
R : Resources
D : Data Curation
O : Writing - Original Draft
E : Writing - Review & Editing
- Vi : Visualization
Su : Supervision
P : Project administration
Fu : Funding acquisition

CONFLICT OF INTEREST STATEMENT

The authors declare no conflict of interest.

DATA AVAILABILITY

The authors confirm that the data supporting the findings of this study are available within the article.

REFERENCES

[1] F. M. Alotaibi and V. G. Vassilakis, "Toward an SDN-Based Web Application Firewall: Defending against SQL Injection Attacks," *Future Internet*, vol. 15, no. 5, p. 170, Apr. 2023, doi: 10.3390/fi15050170.

[2] F. F. Fadlalla and H. T. Elshoush, "Input Validation Vulnerabilities in Web Applications: Systematic Review, Classification, and Analysis of the Current State-of-the-Art," *IEEE Access*, vol. 11, pp. 40128–40161, 2023, doi: 10.1109/ACCESS.2023.3266385.

[3] A. Coscia, V. Dentamaro, S. Galantucci, A. Maci, and G. Pirlo, "PROGESI: A PROxy Grammar to Enhance Web Application Firewall for SQL Injection Prevention," *IEEE Access*, vol. 12, pp. 107689–107703, 2024, doi: 10.1109/ACCESS.2024.3438092.

[4] A. Paul, V. Sharma, and O. Olukoya, "SQL injection attack: Detection, prioritization & prevention," *Journal of Information Security and Applications*, vol. 85, pp. 1–21, Sep. 2024, doi: 10.1016/j.jisa.2024.103871.

[5] J. Román-Gallego, M. Pérez-Delgado, M. L. Viñuela, and M. Vega-Hernández, "Artificial Intelligence Web Application Firewall for advanced detection of web injection attacks," *Expert Systems*, vol. 42, no. 1, pp. 1–18, Jan. 2025, doi: 10.1111/exsy.13505.

[6] R. Ala et al., "Database Under Siege: The Hidden Menace of SQL Injection Attacks," in *International Conference on Broadband Communications, Networks and Systems*, Cham: Springer Nature Switzerland, 2025, pp. 144–153, doi: 10.1007/978-3-031-81171-5_15.

[7] S. Applebaum, T. Gaber, and A. Ahmed, "Signature-based and Machine-Learning-based Web Application Firewalls: A Short Survey," *Procedia Computer Science*, vol. 189, pp. 359–367, Jan. 2021.

[8] B. I. Mukhtar and M. A. Azer, "Evaluating the Modsecurity Web Application Firewall Against SQL Injection Attacks," in *2020 15th International Conference on Computer Engineering and Systems (ICCES)*, Cairo, Egypt: IEEE, Dec. 2020, pp. 1–6, doi: 10.1109/ICCES51560.2020.9334626.

[9] A. A. Bashir and N. Tariq, "Comparison of Mod-Evasive and Modsecurity Modules for Apache Web Server in Lamp Environment," in *2024 IEEE 9th International Conference on Engineering Technologies and Applied Sciences (ICETAS)*, Bahrain, Bahrain: IEEE, Nov. 2024, pp. 1–10, doi: 10.1109/ICETAS62372.2024.11120102.

[10] B. Dawadi, B. Adhikari, and D. Srivastava, "Deep Learning Technique-Enabled Web Application Firewall for the Detection of Web Attacks," *Sensors*, vol. 23, no. 4, p. 2073, Feb. 2023, doi: 10.3390/s23042073.

[11] G. Floris et al., "ModSec-AdvLearn: Countering Adversarial SQL Injections With Robust Machine Learning," *IEEE Transactions on Information Forensics and Security*, vol. 20, pp. 6693–6705, 2025, doi: 10.1109/TIFS.2025.3583234.

[12] V. Clincy and H. Shahriar, "Web Application Firewall: Network Security Models and Configuration," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Jul. 2018, pp. 835–836, doi: 10.1109/COMPSAC.2018.00144.

[13] 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 - SoICT 2018*, Danang City, Viet Nam: ACM Press, 2018, pp. 416–421, doi: 10.1145/3287921.3287946.

- [14] M. Alagoz, M. S. Tok, and K. Bicakci, "Exploring and Improving the Usability of ModSecurity Web Application Firewall," in *2021 International Conference on Information Security and Cryptology (ISCTURKEY)*, Ankara, Turkey: IEEE, Dec. 2021, pp. 51–56, doi: 10.1109/ISCTURKEY53027.2021.9654294.
- [15] I. M. Ibrahim *et al.*, "Web Server Performance Improvement Using Dynamic Load Balancing Techniques: A Review," *Asian Journal of Research in Computer Science*, pp. 47–62, Jun. 2021, doi: 10.9734/ajrcos/2021/v10i130234.
- [16] W. Wicoksono, H. A. Mustaqhim, P. P. Anwas, and L. N. L. Badratul, "Performance Comparison of NGINX, Apache, and Lighttpd Using WRK on a Debian," *Bit-Tech*, vol. 8, no. 1, pp. 670–680, Aug. 2025, doi: 10.32877/bt.v8i1.2661.
- [17] T.-C.-H. Nguyen, M.-K. Le-Nguyen, D.-T. Le, V.-H. Nguyen, L.-P. Tôn, and K. Nguyen-An, "Improving ModSecurity WAF Using a Structured-Language Classifier," in *International Conference on Future Data and Security Engineering*, Singapore: Springer Singapore, 2021, vol. 1500, pp. 89–104, doi: 10.1007/978-981-16-8062-5_6.
- [18] L. Mirabella, C. Bernardeschi, G. Lettieri, F. Lucattini, and S. Montanarella, "A Portable Research HoneyPot for Corporate Networks," in *International Workshop on Security and Trust Management*, Cham: Springer Nature Switzerland, 2025, pp. 128–139, doi: 10.1007/978-3-031-76371-7_9.
- [19] T. L. Kumari and A. R. Babu, "Control of Web Traffic Using Web Application Firewall: Techniques and Challenges," *International Journal of Scientific Research and Technology*, pp. 728–735, Jun. 2023, doi: 10.32628/IJSRST523102130.
- [20] D. R. Chittibala, "Web Application Firewalls (WAF) Integration in DevOps Practices: A Scholarly Exploration of Security, Automation and Continuous Protection," *Journal of Artificial Intelligence & Cloud Computing*, pp. 1–5, Dec. 2022, doi: 10.47363/JAICC/2022(1)210.
- [21] Z. Qu, X. Ling, T. Wang, X. Chen, S. Ji, and C. Wu, "AdvSQLi: Generating Adversarial SQL Injections Against Real-World WAF-as-a-Service," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 2623–2638, 2024, doi: 10.1109/TIFS.2024.3350911.
- [22] A. Ikramaputra, A. Sudarsono, and N. Ningsih, "Enhancing Web Application Firewall with Ensemble Machine Learning to Detect SQL Injection Attacks," in *2025 International Electronics Symposium (IES)*, Surabaya, Indonesia: IEEE, Aug. 2025, pp. 521–526, doi: 10.1109/IES67184.2025.11160726.
- [23] S. Venkatramulu, M. S. Waseem, A. Taneem, S. Y. Thoutam, S. Apuri, and Nachiketh, "Research on SQL Injection Attacks using Word Embedding Techniques and Machine Learning," *Journal of Sensors, IoT & Health Sciences*, vol. 2, no. 1, pp. 55–64, Mar. 2024, doi: 10.69996/jsihs.2024005.
- [24] D. Lu, J. Fei, and L. Liu, "A Semantic Learning-Based SQL Injection Attack Detection Technology," *Electronics*, vol. 12, no. 6, p. 1344, Mar. 2023, doi: 10.3390/electronics12061344.
- [25] D. Mehta, H. Suhagiya, H. Gandhi, M. Jha, P. Kanani, and A. Kore, "SQLIML: A Comprehensive Analysis for SQL Injection Detection Using Multiple Supervised and Unsupervised Learning Schemes," *SN Computer Science*, vol. 4, no. 3, p. 281, Mar. 2023, doi: 10.1007/s42979-022-01626-8.
- [26] M. Idris, I. Syarif, and I. Winarno, "Web Application Security Education Platform Based on OWASP API Security Project," *EMITTER International Journal of Engineering Technology*, pp. 246–261, Dec. 2022, doi: 10.24003/emitter.v10i2.705.
- [27] W. G. J. Halfond, J. Viegas, and A. Orso, "A Classification of SQL Injection Attacks and Countermeasures," in *IEEE International Symposium Secure Software Engineering*, Arlington, VA, USA, 2006, pp. 13–15.
- [28] V. Babaey and A. Ravindran, "GenSQLi: A Generative Artificial Intelligence Framework for Automatically Securing Web Application Firewalls Against Structured Query Language Injection Attacks," *Future Internet*, vol. 17, no. 1, p. 8, Dec. 2024, doi: 10.3390/fi17010008.
- [29] A. Kumar, S. Dutta, and P. Pranav, "Analysis of SQL injection attacks in the cloud and in WEB applications," *Security and Privacy*, vol. 7, no. 3, p. e370, 2024, doi: 10.1002/spy2.370.
- [30] M. Alghawazi, D. Alghazzawi, and S. Alarifi, "Detection of SQL Injection Attack Using Machine Learning Techniques: A Systematic Literature Review," *Journal of Cybersecurity and Privacy*, vol. 2, no. 4, pp. 764–777, Sep. 2022, doi: 10.3390/jcp2040039.
- [31] M. Nasereddin, A. ALKhamaiseh, M. Qasaimeh, and R. Al-Qassas, "A systematic review of detection and prevention techniques of SQL injection attacks," *Information Security Journal*, vol. 32, no. 4, pp. 252–265, Jul. 2023, doi: 10.1080/19393555.2021.1995537.
- [32] J. Zheng, J. Li, C. Li, and R. Li, "A SQL Blind Injection Method Based on Gated Recurrent Neural Network," in *2022 7th IEEE International Conference on Data Science in Cyberspace (DSC)*, Jul. 2022, pp. 519–525, doi: 10.1109/DSC55868.2022.00078.
- [33] B. Brindavathi, A. Karrothu, and C. Anilkumar, "An Analysis of AI-based SQL Injection (SQLi) Attack Detection," in *2023 Second International Conference on Augmented Intelligence and Sustainable Systems (ICAISS)*, Aug. 2023, pp. 31–35, doi: 10.1109/ICAISS58487.2023.10250505.
- [34] V. Mannuru, A. Al-Sinayyid, S. Kadiyala, and R. R. Battula, "Real-Time Detection of Time-Based Blind SQL Injection Using Machine Learning for Critical Infrastructure," in *Leadership, Business and Management in STEM*, Cham: Springer Nature Switzerland, 2026, pp. 38–57, doi: 10.1007/978-3-032-10539-4_4.

BIOGRAPHIES OF AUTHORS



Muhammad Nur Yasir Utomo    was born in Ujung Pandang, South Sulawesi, Indonesia, in 1992. He received his bachelor's degree in computer and network engineering from Politeknik Negeri Ujung Pandang in 2014, and his master's degree from Universitas Gadjah Mada in 2018. His current research interests include information technology, information security, and web performance. He is currently a lecturer in the Department of Informatics and Computer Engineering at Politeknik Negeri Ujung Pandang. He can be contacted at email: yasirutomo@poliupg.ac.id.



Eddy Tungadi    was born in Ujung Pandang, South Sulawesi, Indonesia, in 1979. He received his bachelor's degree in Electrical Engineering from Hasanuddin University in 2003 and his master's degree in Electrical Engineering from the same university in 2009. His current research interests are software engineering. He is currently a lecturer in the Department of Informatics and Computer Engineering at Politeknik Negeri Ujung Pandang. He can be contacted at email: eddy.tungadi@poliupg.ac.id.



Muh. Ahyar    received the bachelor's degree in Applied Science (Telecommunications Engineering) from the Politeknik Elektronika Negeri Surabaya, in 2008, and the master's degrees in Computer Engineering from Universitas Indonesia, in 2013. He is currently an associate professor with the Department of Informatics and Computer Engineering, Politeknik Negeri Ujung Pandang. His research interests include wireless sensor network, IoT, and computer networks. He can be contacted at email: ahyar@poliupg.ac.id.