SQLiFuzz: Uncovering SQL Injection in Any Web Applications

I PUTU ARYA DHARMAADI, University of Groningen, Netherlands and Udayana University, Indonesia
VAN-THUAN PHAM, University of Melbourne, Australia
FADI MOHSEN, University of Groningen, Netherlands
FATIH TURKMEN, University of Groningen, Netherlands

SQL injection (SQLi) is one of the most critical and prevalent security vulnerabilities, as it enables attackers to manipulate backend databases, bypass authentication, and even gain complete control of the underlying system. Since web applications are the primary targets of SQLi, they must be thoroughly tested to ensure they are free of this vulnerability. Recently, several fuzz testing solutions tailored to SQLi vulnerabilities have been developed; however, our preliminary analysis reveals key limitations that hinder their effectiveness: they primarily focus on GUI-based inputs while neglecting API endpoints, rely on less effective request selection and generation strategies, and require complex configurations to be deployed in practice.

To address these gaps, we propose **SQLiFuzz**, a universal and simple-to-deploy SQL injection fuzzer that operates across both GUI (web pages) and API entry points. SQLiFuzz introduces three key distinguishing features: (i) a reverse proxy that unifies request collection and fuzzing, and allows seamless integration with existing crawlers and API scanners, (ii) a database proxy that enables request–query matching and serves as a reliable oracle, and (iii) a feedback-driven fuzzer that prioritizes potentially effective requests and parameters, and validates exploitability through database responses. We evaluated SQLiFuzz on six security benchmarks and ten real-world applications. SQLiFuzz successfully detects the majority of known SQLi cases in benchmarks and uncovered nine new vulnerabilities that had been overlooked by state-of-the-art tools in real-world applications. These results highlight SQLiFuzz's ability to detect SQL injection across diverse web application frameworks and architectures while maintaining practicality and ease of deployment.

CCS Concepts: • **Security and privacy → Web application security**; **Software security engineering**.

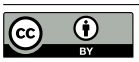
Additional Key Words and Phrases: web fuzzing, SQL injection, web GUI, web API, database proxy

ACM Reference Format:
I Putu Arya Dharmaadi, Van-Thuan Pham, Fadi Mohsen, and Fatih Turkmen. 2026. SQLiFuzz: Uncovering SQL Injection in Any Web Applications. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE142 (July 2026), 22 pages. https://doi.org/10.1145/3808149

1 Introduction

Web applications have become the backbone of modern digital services, and their central role in handling sensitive information makes them prime targets for cyberattacks. Among these threats, SQL injection (SQLi) stands out as one of the most prevalent and persistent vulnerabilities. SQLi arises when an application fails to properly sanitize user-supplied input before including it in SQL queries, allowing attackers to manipulate the database backend. A successful SQLi attack can lead to unauthorized access, disclosure of sensitive information, or even complete takeover of

Authors' Contact Information: I Putu Arya Dharmaadi, University of Groningen, Groningen, Netherlands and Udayana University, Badung, Indonesia, arya.dharmaadi@rug.nl; Van-Thuan Pham, University of Melbourne, Melbourne, Australia, thuan.pham@unimelb.edu.au; Fadi Mohsen, University of Groningen, Groningen, Netherlands, f.f.m.mohsen@rug.nl; Fatih Turkmen, University of Groningen, Groningen, Netherlands, f.turkmen@rug.nl.



ACM 2994-970X/2026/7-ARTFSE142
https://doi.org/10.1145/3808149

the underlying system. To mitigate such risks, developers must ensure that web applications are thoroughly tested and free of SQL injection vulnerabilities before deployment.

A variety of automated tools and techniques have been developed to detect SQLi, including black-box, grey-box, and white-box approaches. Widely used web scanners and fuzzers, such as OWASP ZAP [31], Burp Suite [34], sqlmap [35], Witcher [37], Atropos [12], Phuzz [27], and Predator [40], have contributed significantly to vulnerability discovery in web under test (WUT). However, according to our analysis (Section 2), these tools still face critical limitations, making the detection of SQLi vulnerabilities challenging in practice.

First, most existing tools focus primarily on conventional attack vectors exposed through web pages or graphical user interfaces (GUIs). Modern web applications, however, increasingly rely on APIs to support mobile clients, microservices, and third-party integrations. While web developers often enforce stronger sanitization on GUI inputs, API endpoints may be less carefully analyzed, creating a growing blind spot for SQLi related issues as the WUT evolves. Recent vulnerabilities (i.e., CVE-2025-7670, CVE-2025-6783, and CVE-2025-50979) illustrate this issue: in each case, applications carefully validated the inputs through the GUI but applied insufficient checks to inputs submitted via the APIs.

Beyond this gap, there are also other limitations related to effectiveness and usability. Current tools suffer from ineffective request selection where simplistic heuristics risk missing vulnerable endpoints (explained further in Section 2.2.2). They also lack robust exploitable analysis which might flag benign errors as SQLi. Grey-box and white-box approaches are particularly sensitive to code complexity. Specifically, static code analysis often fails in large codebases and require direct access to the WUT. Finally, these grey-box and white-box approaches demand complex and less scalable configurations which hinders their employment in modern microservice environments with diverse technology stacks.

These limitations highlight the need for a simple-to-deploy approach that still maintains high accuracy of SQLi detection. In addition, this simple-to-deploy approach is crucial as the recent study of Nourry et al. [29] states that *"fuzzers are very complex tools to set up and build before they can be used"*. Consequently, we address these three research problems: ***(i) how to design a SQLi fuzzer that is applicable across diverse web architectures and platforms***, ***(ii) how to identify and fuzz the most relevant HTTP requests efficiently***, and ***(iii) how to reliably detect successful exploitation in heterogeneous execution environments***.

To tackle these problems, this paper introduces SQLiFuzz, a simple-to-deploy grey-box web fuzzer designed for effective testing of SQLi vulnerabilities in diverse types of web applications, including GUI-based (or HTML-based) and API-based. SQLiFuzz analyses potentially vulnerable HTTP requests by crawling the WUT (for GUI-based) or generating requests from OpenAPI specifications (for API-based), and intercepting the communication between the WUT and the database via a DB proxy. When a parameter value in the submitted request appears in the captured database query, the request is classified as potentially effective (i.e., exploitable) because the value may not be well sanitised. SQLiFuzz then systematically injects malicious characters into the request to trigger SQL syntax errors in the DBMS, which are also captured by the proxy. The presence of an SQL syntax error caused by this unexpected character indicates that the parameter value was not properly sanitized, which in turn suggests the possibility of SQL injection (SQLi).

SQLiFuzz is designed to be lightweight, modular, and highly adaptable, and can be seamlessly applied across diverse web environments, including microservice-based systems. Evaluations across six security benchmarks and ten real-world applications demonstrated that SQLiFuzz can successfully detect the majority of known SQLi cases in the benchmarks without modification to the source code. More importantly, our evaluations uncovered nine new vulnerable endpoints, which state-of-the-art tools had missed, in real-world applications. These results demonstrate SQLiFuzz's

capability to uncover previously unknown vulnerabilities while offering more practicality and ease of deployment.

1.1 Contributions

In summary, we make the following contributions.

(1) **New Web Fuzzer**: To reveal SQL injection in modern web applications, we propose SQLiFuzz, which works in a very light grey-box setup and fits well in diverse web environments. It also covers more attack vectors than existing proposals. The evaluations show that SQLiFuzz can even detect more SQLi than existing approaches. To support further research on this topic, we release the implementation of SQLiFuzz and the WUT datasets in a GitHub repository[1].
(2) **New Potentially Vulnerable Request Analysis**: To help the proposed fuzzer select only potentially vulnerable HTTP requests to fuzz, we propose a new request analysis. This component utilizes a DB proxy to capture SQL queries and compares the queries to the values submitted in the request. When matched, the values are then fuzzed to observe the DBMS behaviour.
(3) **New Oracle Detector Implementation**: To detect whether the fuzzed requests successfully raise a DBMS error, we propose a vulnerability detector. This component utilizes the DB proxy to capture DBMS responses after the web application receives the fuzzed requests. It generates a report when the fuzzed value in the request causes the DBMS to return an error parsing message due to a malformed query. This malformed query indicates that the parameter value is not properly sanitized, making SQLi possible. Implementing this detector in the DB proxy offers several benefits, including ease of deployment and maintenance.
(4) **New Vulnerability Dataset**: Apart from existing security benchmarks, which contain artificial bugs, we curate three real SQL injection cases (sourced from recent CVE reports) in more complex web applications.

2 Preliminary Analysis

This section presents our preliminary analysis of existing approaches for detecting SQL injections in recently reported vulnerabilities. The findings highlight their limitations and motivate the development of a more effective SQLi fuzzer.

2.1 Recent SQL Injection Cases

We analyze three recent vulnerabilities: CVE-2025-7670 [26], CVE-2025-6783 [25], and CVE-2025-50979 [24], as they were recently discovered in popular open-source applications. The first vulnerability affects WordPress [41] that uses the JS Archive List plugin [22] and is susceptible to time-based SQL injection. The flaw originates in the *build_sql_where()* function, which is accessible through both the web GUI and the web API. While the GUI interface applied proper input sanitization, the API interface did not, leaving it vulnerable. A representative exploit request is: */wp-json/jalw/v1/archive/2025/08/?year=«sql_injection»*. Since this endpoint constructs a `SELECT` query, an attacker can inject payloads to manipulate the query logic, such as union-based injections to retrieve data from other tables. In cases where the DBMS permits stacked queries and sufficient privileges are available, the attacker may also execute additional statements (e.g., `DROP TABLE`), potentially leading to destructive effects.

The second vulnerability also affects WordPress, but through the GoZen Forms plugin [30], while the third targets NodeBB [28]. Despite affecting different systems, these three vulnerabilities share

[1]Code and dataset can be accessed at this link:https://github.com/websecfuzz/SQLiFuzz

the same root cause: applications validate user input carefully through the web GUI but apply weaker checks for inputs received via web APIs.

None of the existing tools, such as OWASP ZAP [31], Burp Suite [34], sqlmap [35], or Phuzz [27], can detect these issues because they primarily focus on web GUIs rather than APIs. Extending them to handle APIs is nontrivial, as it requires parsing OpenAPI specifications or Swagger documents [7]. Existing tools that target APIs, such as EvoMaster [2] and Schemathesis [14], are not designed to detect SQL injections and do not work on GUIs. Similarly, Witcher [37], Atropos [12], and Predator [40] are ineffective in this context, as they require mapping the vulnerable URL to a specific PHP file, which is difficult when the exploitation URL path neither reveals a *.php* extension nor refers to a specific PHP file.

These examples highlight a critical gap in current SQL injection testing approaches. Recent real-world vulnerabilities increasingly emerge at less visible entry points, such as APIs, where the attack surface is broader and more challenging to analyze. This motivates the design of a new approach capable of identifying vulnerable attack vectors across both GUIs and APIs, while efficiently confirming successful exploitation.

2.2 Current Approach Limitations

A deeper analysis of the aforementioned approaches reveals other limitations that restrict their effectiveness in practice. All of the limitations are summarized as follows.

2.2.1 Only Focus on Conventional Attack Vectors. Most existing SQL injection testing tools and fuzzers are designed primarily for conventional attack vectors, namely, GUI-based or HTML-based web interfaces. This design assumption reflects earlier generations of web applications, where user interaction was done through graphical forms. However, apart from web GUIs, modern web applications increasingly expose functionality through programmatic interfaces such as RESTful APIs, GraphQL endpoints, or gRPC services. These APIs are now a dominant means of integration between services, mobile applications, and third-party clients.

Despite this shift, the existing tools rarely extend their scanning capabilities beyond traditional front-end interfaces. As a result, they leave critical parts of the attack surface underexplored. Our motivating examples illustrate this point clearly: web developers tend to enforce stricter validation and sanitization on GUI inputs, since these are visible to end-users and more thoroughly tested. In contrast, API endpoints often receive less attention, either because they are perceived as "internal" or because their traffic is less transparent to conventional penetration-testing tools. This lack of attention by API programmers creates an opportunity for attackers to exploit.

The lack of methods for thoroughly testing SQL injection vulnerabilities in both web GUIs and web APIs represents a significant gap in the existing literature. To address this issue, a new approach (described in Section 3.1) is required—one that extends beyond merely validating web page inputs. It should also incorporate a systematic examination of APIs, ensuring that both interfaces are given equal consideration.

2.2.2 Ineffectiveness of Request Selection. While Witcher [37] has higher accuracy than the other approaches because of its coverage-guided feedback and effective oracle, it has some limitations on the request selection process. Witcher works by crawling the WUT to collect submitted requests. After storing all collected requests, Witcher only selects those with specific URL extensions (e.g., .php) for fuzzing. The main reason is that Witcher uses a PHP-CGI system, which targets the actual PHP file to run, rather than alias-based URLs, due to fast processing. In addition, the Witcher's paper reasoned that a request URL without a PHP extension is excluded because it looks like a directory listing. While such filtering reduces noise and improves fuzzing efficiency, it creates a drawback: in modern web applications, where routing often relies on complex URL

```
SELECT SCHEMA_NAME FROM INFORMATION_SCHEMA.SCHEMATA WHERE SCHEMA_NAME = 'appwrite' **Result Set
SET STATEMENT max_statement_time = 15 FOR CREATE TABLE `appwrite`.`logsV1__metadata` (_id INT(11) UN
   **ERROR 1050: 2S01Table 'logsV1__metadata' already exists
SELECT SCHEMA_NAME FROM INFORMATION_SCHEMA.SCHEMATA WHERE SCHEMA_NAME = 'appwrite' **Result Set
SET STATEMENT max_statement_time = 15 FOR /* host: 6fd59c6b985a */ /* project: console */
```

(a) Error because a table name already exists in the Appwrite application.

```
[COM_STMT_EXECUTE] 688 ['built-in', 'admin']
  **Result Set
INSERT INTO `syncer` (`owner`,`name`,`created_time`,`organization`,`type`,`database_type`,`ssl_mode`,`ssh_t
  **OK 689 26
[COM_STMT_EXECUTE] 689 ['admin', '', '', 'u_nKnF9beFbp3q-wZbzCDiABjkX67gdR86OiTfzUnUHfHv5fc-7c3ImSZnY5rrKTC
  **ERROR 1406: 2001Data too long for column 'organization' at row 1
SELECT `owner`, `name`, `created_time`, `organization`, `url`, `method`, `content_type`, `headers`, `events
  **Result Set
```

(b) Error because the column name is too long in the Casdoor application.

```
SELECT "cid", "data", "created", "expire", "serialized", "tags", "checksum" FROM "cache_config
   **Result Set
INSERT INTO "semaphore" ("name", "value", "expire") VALUES ('state:Drupal\\Core\\Cache\\CacheC
   **OK
INSERT INTO "semaphore" ("name", "value", "expire") VALUES ('state:Drupal\\Core\\Cache\\CacheC
   **ERROR 1062: 3000Duplicate entry 'state:Drupal\Core\Cache\CacheCollector' for key 'semapho
SELECT "cid", "data", "created", "expire", "serialized", "tags", "checksum" FROM "cache_bootst
```

(c) Error of a duplicated entry in the Drupal application.

Fig. 1. Some observed SQL queries and responses (with the format of ***«query»*** ** ***«response»***) during preliminary analysis using a DB proxy. When a user sends certain requests, the DBMS replies with error messages without directly showing them on the screen. The errors are triggered because the WUT executes the request, but the errors are not exploitable because of no direct correlation between the errors and the submitted data. Therefore, it is important to only include exploitable errors as SQL injection vulnerabilities.

rewriting and frameworks that hide file extensions, many vulnerable endpoints may be overlooked. Consequently, Witcher's extension-based filtering strategy risks missing a substantial number of fuzzing opportunities. Therefore, relying solely on URL extensions for finding potentially vulnerable requests is insufficient. A more flexible and reliable strategy is required, which will be described in Section 3.2.

2.2.3 Lack of Exploitable Analysis. Another limitation of existing work lies in the lack of exploitable analysis. For example, Witcher [37] classifies a request as exploitable once it observes that the request triggers SQL query errors in the database. However, this assumption is simplistic because a request that produces SQL errors might not indicate an exploitable SQL injection.

In practice, SQL errors may arise from a variety of causes, and some of them are not controlled by user inputs. For instance, developers may accidentally introduce faulty query statements, mismatched data types, or misconfigured database schemes that result in SQL errors (see Figure 1). Such cases represent implementation bugs but are not exploitable vulnerabilities, since the errors cannot be controlled by malicious inputs. If the error originates solely from developer mistakes and not from user-provided values, it should not be considered a SQL injection case.

Without analyzing the relationship between request parameters and the resulting SQL errors, tools may incorrectly label non-exploitable bugs as injection vulnerabilities. Therefore, it is essential to complement error detection with exploitable analysis (will be described in Section 3.4.3), which

can identify which specific part of the submitted request contributes to query construction and whether the observed SQL errors are indeed controlled by attacker-supplied inputs. Only by establishing this causal link can we reliably distinguish true SQL injection vulnerabilities from benign developer mistakes.

2.2.4 Sensitive to Code Complexity. A recent work from Wang et al. [40] that combines white-box analysis and dynamic testing shows promising results in finding SQL injection quickly. Predator, their proposed tool, uses advanced static analysis to infer relevant URLs and parameters needed to reach specific code locations, instead of employing a crawler. This analysis further computes block distances that estimate how close an execution path is to the identified targets. Finally, selective dynamic instrumentation leverages these block distances to guide the fuzzer more effectively toward vulnerable code.

Despite these innovations, Predator's reliance on static analysis introduces new limitations. The accuracy of its approach depends heavily on the quality of the underlying static analysis tools. In applications with highly complex codes, the analysis may fail to capture critical execution paths. As a result, Predator can miss vulnerable targets, especially in modern large-scale web applications. We address this limitation with our proposed approach, which will be described in Section 3.3.

2.2.5 Less Scalable Configuration. Witcher [37], Phuzz [27], and Predator [40] all require carefully configured execution environments for WUT. In practice, this setup often involves instrumenting WUT's source code or modifying server configurations. Such requirements make these tools well-suited to monolithic applications where the entire codebase and execution stack can be centrally controlled.

However, this assumption does not hold in modern deployment environments. Many modern web applications follow distributed microservice architectures, where functionality is spread across multiple services written in different programming languages and deployed on heterogeneous platforms. Under such conditions, setting up the specialized configurations required by existing tools across all environments is either less practical or even infeasible. For example, configuring dynamic instrumentation across several microservices would demand extensive engineering effort. In addition, different frameworks, programming models, and API protocols mean that a configuration strategy designed for one environment (e.g., PHP-based monoliths) does not generalize easily to others. This lack of portability limits the practical adoption of current grey-box or white-box SQLi fuzzing tools in real-world industrial settings.

Therefore, there is a need for a simple-to-deploy approach that scales across architectures and technology stacks. Such a solution (will be described in Section 3.3) should minimize the configuration burden on security testers, integrate seamlessly with distributed and service-oriented deployments, and remain agnostic to specific programming languages or frameworks.

3 Proposed Approach: SQLiFuzz

According to the limitations explained in the previous section, this paper proposes a new fuzzer, called SQLiFuzz, to reveal SQL injection in complex web applications efficiently. Illustrated in Figure 2, SQLiFuzz provides three core components: a reverse web proxy that intercepts HTTP requests and responses, a fuzzer that mutates the intercepted requests, and a database proxy to prove successful exploitations. The system is designed to be lightweight and modular: it does not include its own crawler or API scanner, but instead integrates seamlessly with existing tools, such as OWASP Zaproxy or EvoMaster. This design allows SQLiFuzz to complement existing testing workflows rather than replace them, enabling broader applicability across different environments.

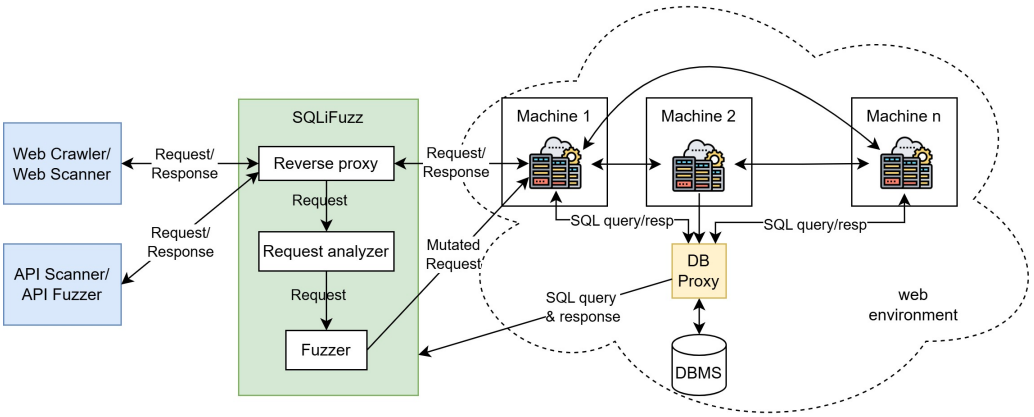


Fig. 2. Proposed fuzzer (green-box) and DB proxy (yellow-box) for revealing SQL injection in a complex web under test (WUT). Instead of connecting directly to the WUT, existing web scanners and API fuzzers (blue box) send HTTP requests through the reverse proxy, which records requests and maps them to SQL queries observed by the DB proxy. When a parameter value matches a query, the request proceeds to fuzzing. If a fuzzed request triggers a DBMS syntax error captured by the DB proxy, SQLiFuzz flags the endpoint as a SQL injection vulnerability.

3.1 Reverse Web Proxy

To overcome the limitation of restricted attack vectors (Section 2.2.1), SQLiFuzz introduces a reverse web proxy that integrates seamlessly with both web crawlers and API scanners. The proxy is positioned in front of the WUT to intercept and collect all HTTP requests generated by these crawlers and scanners. Each intercepted request is forwarded to the Request Analyzer, and, if classified as interesting, is subsequently passed on to the fuzzer campaign.

This design differs from existing approaches such as Witcher and Phuzz, which work exclusively with their web crawlers and treat crawling and fuzzing as two separate phases. In those tools, the crawler first exhaustively explores the WUT and stores all valid requests. Only after this collection phase is complete does fuzzing begin on the stored request set. In contrast, SQLiFuzz unifies request collection and fuzzing into a single continuous process. This integration addresses challenges in complex applications, such as WordPress [41] or Appwrite [1], where dynamic security measures (e.g., tokens, nonces, or session-specific keys) are generated at runtime whenever URLs are accessed. When crawling and fuzzing processes are separated, the stored requests often become invalid because the required tokens expire, forcing the fuzzer to look for valid tokens repeatedly. This process not only slows testing but may also prevent certain functions from being exercised at all.

By combining request collection with fuzzing into one process, SQLiFuzz is able to reuse dynamically generated tokens within their valid time frame. As a result, the fuzzing campaign operates more efficiently and effectively, ensuring higher coverage of dynamic and token-protected functionalities.

3.2 Request Analyzer

To ensure that the fuzzing process is guided toward meaningful attack surfaces and to replace the conventional request selection process (explained in Section 2.2.2), we propose a request analyzer. This request analyzer module is called when the reverse web proxy catches an HTTP request. This module is responsible for correlating incoming HTTP requests with the SQL queries

executed by the WUT. It systematically extracts the associated parameters of the requests. In this context, HTTP parameters refer to all name-value pairs that appear in the HTTP header, the URL query string, or the HTTP body. Simultaneously, the request analyzer obtains the SQL queries submitted to the database management system (DBMS) via the DB proxy. By examining both data sources, it determines whether any parameter values from the HTTP request are reflected in the corresponding SQL queries. When a match is observed, the request is classified as potentially vulnerable, since the presence of user-supplied data in SQL statements might indicate the possibility of insufficient sanitisation. Requests identified as potentially vulnerable are then forwarded to the fuzzing campaign.

This design differs from previous works. If we simply adopt state-of-the-art methods (i.e., request selection employed by Witcher [37] and Predator [40]), fuzzing would be inefficient and scale poorly. Specifically, in cases where URLs do not contain explicit file extensions (e.g., "*.php*"), a naive solution would be to fuzz all URLs exhaustively, which will be time-consuming. This raises a key challenge: how to efficiently identify potentially vulnerable requests from a large set of candidates. The proposed Request Analyzer addresses this challenge by prioritizing only those requests whose inputs directly influence SQL query construction.

3.3 DB Proxy

The DB Proxy is a core component that enables SQLiFuzz to implement a lightweight grey-box SQL injection oracle. It is proposed to address the problems of WUT complexity (Section 2.2.4) and scalability (Section 2.2.5). Functionally, it operates similarly to the Reverse Web Proxy, but instead of mediating HTTP traffic, it intercepts SQL queries and responses between WUT and DBMS. The DB Proxy is deployed transparently in front of the DBMS, using the same port but bound to a different IP address. This setup offers a practical deployment advantage: most modern web applications provide administrator interfaces for modifying database connection settings without requiring any changes to the application's source code. Leveraging this mechanism, users of SQLiFuzz can simply reconfigure the application to connect to the DB Proxy rather than directly to the DBMS. No source-level instrumentation or invasive code modification is required, ensuring ease of integration even in complex applications.

3.3.1 Challenge. Although using a DB proxy eliminates the need for WUT recompilation, it also introduces a significant challenge: correlating and mapping HTTP requests with their associated SQL queries. It is challenging because they appear at different layers/operations of the WUT stack (i.e., HTTP requests are handled by the web server and SQL queries by DBMS). Addressing this correlation problem is essential for identifying potentially vulnerable queries and for determining whether injected inputs are successfully propagated to the DBMS.

3.3.2 Solution: Request-Query Matching. To correlate HTTP requests with the SQL queries they trigger, the DB proxy records both submission and response times and employs a matching module. Specifically, when the reverse web proxy forwards an HTTP request to WUT, the submission timestamp is logged. When the end stream of the corresponding HTTP reply is received, the response timestamp is also stored. The interval between these two timestamps defines a time window within which the SQL queries generated by the request are expected to appear in the proxy log.

Because the web crawler may employ multiple concurrent workers to explore the WUT, ephemeral port data is also incorporated to differentiate overlapping requests. Ephemeral port data is a short-lived transport protocol port number used by the WUT to establish a TCP/IP connection with the DBMS. In a short time period, one request is assigned a unique ephemeral port number, and this port number is only used by the assigned request. Then, by observing these port numbers in the

SQL queries, the matching module collects the port number that appears only in the observed time window. Queries that fall outside the time window, or that use a different port number, are excluded. By combining time-window filtering with port-based checking, the DB proxy can reliably associate each SQL query with the specific HTTP request that triggered it.

3.3.3 Handling Prepare Statements. In addition to standard queries, the DB proxy is capable of handling prepared-statement queries. Prepared statements enforce a clear separation between the SQL query structure and the associated data, which requires the WUT to transmit a query at least twice: once to convey the query structure containing data placeholders (e.g., ? in MySQL and $ in PostgreSQL), and once to supply the actual data values. Under these conditions, the DB proxy captures all submitted queries without combining the actual data with the placeholders in earlier queries, because the request analyzer only needs to systematically compare each submitted query string with the submitted request. Therefore, when a submitted value appears in either query with placeholders or query with actual data, the submitted request is marked as potentially vulnerable and then forwarded to the fuzzing campaign.

3.4 Fuzzer

The fuzzer is activated only when the request analyzer identifies a potentially vulnerable request. Its role is to systematically attempt exploitation by selecting parameters, mutating their values, and verifying the outcomes against the oracle provided by the DB Proxy.

3.4.1 Parameter Selection. The fuzzer begins by selecting one of the HTTP parameters to mutate. Parameters are drawn from the request's headers, URL query string, or body. To improve efficiency, parameters previously flagged as potentially vulnerable are assigned higher selection weights, ensuring they are prioritized during fuzzing while still allowing random exploration of other parameters.

3.4.2 Mutation. Inspired by prior approaches such as Witcher, Phuzz, and Predator, SQLiFuzz employs mutation strategies based on special characters that commonly trigger SQL parsing errors. These special characters form a manually curated set of non-alphanumeric and extended Unicode characters. The set includes, for example, characters used as identifier delimiters (`'` `"` `` ` `` `[`), punctuation marks (`,` `.` `;`), comment indicators (`--` and `/*`), wildcard symbols (`%` and `_`), and Unicode (`£` and `¥`). Our mutator prioritizes choosing the apostrophe (`'`) or the asterisk (`*`) characters because they are more likely to break the SQL query structure, according to our preliminary analysis.

The selected character is then inserted into the value of a chosen parameter at different positions (beginning, middle, or end), or used to fully replace the original value. This altered parameter is put back in the request, which is subsequently sent to the WUT.

3.4.3 Oracle Verification. To overcome the lack of exploitable analysis (Section 2.2.3), SQLiFuzz introduces a new oracle verification. This verification module works in the DB Proxy, which records both the SQL queries executed and the corresponding responses from the DBMS. These responses are classified into three categories: `OK`, `Result Set`, and `Error`. During oracle verification, the fuzzer checks whether the mutated value appears in the generated SQL query. If it does and the DBMS returns a **SQL syntax error**, the request is marked as having successfully exploited a SQL injection vulnerability.

3.4.4 Mutation Feedback. To guide the fuzzer more effectively toward discovering SQLi vulnerabilities, SQLiFuzz leverages feedback from both the DB Proxy and HTTP response codes. When a parameter value from a submitted request is observed in a corresponding SQL query, the request

is stored in the Corpus for further fuzzing. In addition, requests that trigger server-side errors, indicated by 5xx HTTP response codes, are preserved.

3.5 Custom Web Crawler

Our preliminary analysis revealed that existing web scanners, such as OWASP ZAP and Burp Suite, often collect a limited number of unique HTTP requests. This restriction reduces overall coverage and, in turn, diminishes the effectiveness of fuzzing. As a result, most existing fuzzers, such as Witcher, Phuzz, WebFuzz [38], and BACFuzz [6], have developed their own crawling mechanisms to compensate for these limitations.

In SQLiFuzz, we also implement a custom web crawler to maximize request collection and ensure broader exploration of the WUT. However, because this crawler primarily builds upon established techniques and does not introduce novel ideas, we do not consider it part of the paper's core contribution. Instead, it serves as a supporting component that enables SQLiFuzz to operate with higher coverage. The crawler performs a depth-first traversal of the WUT by systematically following hyperlinks, submitting HTML forms, clicking buttons, and invoking JavaScript-based navigation where applicable. The crawler also traverses new tabs, modals, and pop-up pages opened by the WUT.

4 Design and Usability Comparison

To better illustrate the novelty and advantages of SQLiFuzz, we compare its design and usability with existing SQL injection testing tools, including OWASP ZAP, Burp Suite, SQLMap, Witcher, Phuzz, Atropos, and Predator.

4.1 Design

The design comparison is based on the following criteria: testing approach (black-box, grey-box, or white-box), target interface (web GUI, web API, or file path), target configuration, scalability to microservice-based architectures, and accuracy in detecting SQLi. Table 1 shows the overall comparison. In general, grey-box and white-box approaches bring more satisfactory results because they can extract more knowledge and internal information about WUT. However, the approaches that use OS-level or interpreter-level hooks require configuring specific execution environments and are commonly tied to monolithic architectures, making them less applicable in distributed microservice settings. SQLiFuzz does not require the WUT to use any runtime hooks or to modify any web environment. It only replaces the actual DBMS to be a DB proxy, making it lightweight and broadly applicable across both monolithic and microservice-based applications.

Table 1. Comparison of SQL injection testing tools.

Tool	Testing Approach	Target Interface	Need WUT Configuration?	Scalable to Microservice?	Detection Accuracy
OWASP ZAP	Black-box	Web GUI	No	High	Medium
Burp Suite	Black-box	Web GUI	No	High	Medium
SQLMap	Black-box	Web GUI	No	High	Medium
Witcher	Grey-box	File Path	Yes (OS-level hooks)	Low	High
Phuzz	Grey-box	Web GUI	Yes (PHP-level hooks)	Low-Med	High
Atropos	Grey-box	File Path	Yes (OS-level hooks)	Low	High
Predator	White-box	File Path	Yes (OS-level hooks)	Low	High
SQLiFuzz	**Grey-box**	**GUI & API**	**Yes (DB proxy)**	**High**	**High**

4.2 Usability

In addition to design comparison, we compare the usability of SQLiFuzz with existing tools. Usability is a crucial dimension, as a fuzzer usually requires complex configurations that limit its adoption, regardless of its technical performance [33]. We assess usability along two criteria: Configuration Effort and Integration Flexibility.

Configuration effort is related to the number of steps and manual interventions required to set up the tool for testing a new WUT. Witcher, Phuzz, Atropos, and Predator require users to configure application-specific execution environments, such as instrumented PHP runtimes, hooked libraries, or language-specific analyzers. This setup is more feasible for monolithic applications but becomes less practical in microservice environments with heterogeneous technology stacks. Furthermore, when the WUT continuously grows and requires more updated components, it is impractical to repeatedly configure the updated web environments. By contrast, SQLiFuzz only requires redirecting traffic through the DB proxy, which can be configured without modifying source code or the web environment. As most web applications provide settings for database connections and proxy configurations, the setup effort is significantly lower.

On the other hand, integration flexibility is the ability to interoperate with external components such as crawlers, scanners, or API fuzzers. Existing web fuzzers are often tightly coupled with their own crawlers, limiting their ability to leverage external scanners. For instance, Witcher depends on its own crawler and uses a specific format of crawled results, making it difficult to incorporate an alternative crawler. SQLiFuzz addresses this limitation by design. It operates transparently with both GUI-based web scanners (e.g., OWASP ZAP, Burp Suite) and API fuzzers (e.g., EvoMaster, Schemathesis), while also supporting a new custom crawler. This modularity enables users to select the most suitable crawling or scanning tool depending on the WUT's architecture.

5 Empirical Evaluation

To comprehensively evaluate the effectiveness of SQLiFuzz, we conduct experiments on both popular security benchmarks and real-world web applications containing known and unknown SQL injection vulnerabilities.

5.1 Fuzzer Implementation

We implement SQLiFuzz using Python to maintain compatibility with MITMProxy [23], which serves as the reverse proxy in front of the WUT. MITMProxy is chosen because it is widely adopted and provides the extensibility we require through its addon scripting features. SQLiFuzz is integrated with MITMProxy via these addons, enabling the interception and forwarding of HTTP requests to the request analyzer and fuzzer. In addition, we develop a dedicated DB Proxy in Python to capture SQL queries and DBMS responses.

As explained in Section 3.5, we also developed a custom web crawler in Python. The crawler utilizes the Playwright [21] library to open and manipulate web pages using a controlled browser. Similar to the work of Khodayari et al. [16], which crawls around 1M web pages, our crawler follows a depth-first strategy, visiting the most recently discovered link. Furthermore, our crawler uses URL components as the basis for page similarity, meaning it only visits pages with different URL components [36].

5.2 Web Scanners and API Fuzzers

Even though we implement a custom crawler to complement SQLiFuzz, SQLiFuzz can interoperate with existing web scanners and API fuzzers. For this evaluation, we incorporate OWASP ZAP and Burp Suite, which are widely used for crawling and vulnerability scanning. In addition, as

API scanners, we employ EvoMaster [2] and Schemathesis [14], which have demonstrated strong coverage among black-box fuzzers according to the surveys of Kim et al. [18] and Zhang et al. [43].

5.2.1 OWASP Zaproxy. We use OWASP ZAP version 2.16.1, enabling the Spider, AjaxSpider, and Active Scan modules for SQL injection testing. Authentication credentials for each WUT are configured in advance to allow full access during scanning. After execution, SQL injection alerts are extracted from the generated report files. We do not use a specific time-out to limit OWASP ZAP because this tool typically continues until all links are visited.

5.2.2 Burp Suite. We employ Burp Suite Professional version 2025.7.4, select the Deep Scan mode, and use the crawl-and-audit feature without AI enhancements. As with ZAP, administrator credentials are stored in Burp's authentication settings to ensure proper access. Upon completion, SQL injection alerts are collected from the issues tab in the report screen. We do not use a specific time-out to limit Burp Suite because this tool typically continues until all links are visited.

5.2.3 EvoMaster and Schemathesis. We use EvoMaster version 4.0.0 (black-box mode) and Schemathesis version 4.1.4. Both tools are configured with our login scripts that authenticate as administrators and store session cookies. The cookies are passed to the tools using the `-header` option to enable authenticated requests. EvoMaster additionally requires the specification of a testing duration, which we set to five hours.

5.2.4 Witcher and Phuzz. For comparison with state-of-the-art fuzzers, we use Witcher and Phuzz. Among the platforms covered by our evaluated WUTs, Witcher supports only PHP, Node.js, and Ruby applications through instrumented Docker base images, while Phuzz supports only PHP. As a result, both tools cannot be applied to applications built on unsupported platforms. Since crawler performance critically affects fuzzing outcomes, all tools (including SQLiFuzz) are provided with the same request set generated by our custom crawler. To ensure compatibility, our crawler exports requests in the formats required by each tool: *request_data.json* for Witcher and *HAR* files for Phuzz. This setup ensures that each fuzzer operates on an identical set of HTTP requests as the fuzzing seeds. Each fuzzing campaign is executed for 24 hours. Since various crashes are reported, only crashes that involve DBMS syntax errors are reported in the evaluation.

5.3 WUT

We classify the WUT into two groups: security benchmarks and popular real-world applications. The first group is used to validate whether SQLiFuzz can successfully detect known SQL injection vulnerabilities, while the second group evaluates its capability to uncover previously unknown flaws.

5.3.1 Security Benchmarks. We first evaluate SQLiFuzz against established security benchmarks that are widely used for vulnerability testing and training. Specifically, we use DVWA [10], XVWA [42], and bWAPP [20], each of which includes SQL injection cases. In DVWA and XVWA, we identified two SQL injection test cases per benchmark. In bWAPP, we identified 17 test cases; however, three rely on SQLite, which falls outside the scope of our current implementation. Therefore, we include only the remaining 14 cases in our evaluation.

In addition to these benchmarks, we incorporated real-world vulnerabilities from recent CVE reports discussed in Section 2.1, namely CVE-2025-7670, CVE-2025-6783, and CVE-2025-50979. These cases ensure that SQLiFuzz is assessed not only on synthetic benchmarks but also against real vulnerabilities. Table 2 summarizes the benchmarks used in our experiments.

5.3.2 Popular Applications. To further evaluate SQLiFuzz in realistic settings, we focus on open-source applications that support both HTML-based GUIs and OpenAPI specifications. This choice

Table 2. Security benchmarks sorted by popularity. While the first three are widely used security benchmarks, the other three are our new benchmarks from recent CVE reports.

No.	WUT	Version	Language/ Platforms	Database	Line of Code	Github Star
A1	DVWA	2.5	PHP	MySQL	7K	11.7K
A2	XVWA	fb30fa5	PHP	MySQL	17K	1.7K
A3	bWAPP	21c7c9d	PHP	MySQL	31K	42
A4	CVE-2025-7670 (Wordpress + JS Archive List)	6.8.1 6.1.5	PHP	MySQL	737K	20.4K
A5	CVE-2025-6783 (Wordpress + GoZen Forms)	6.8.1 1.1.5	PHP	MySQL	737K	20.4K
A6	CVE-2025-50979 (NodeBB)	4.3.0	Node.js	PostgreSQL	110K	14.7K

allows us to test the fuzzer across both user-facing and API-driven attack surfaces. Because SQLiFuzz requires minimal configuration on the WUT side, we are able to target complex applications that share their deployment configurations through Dockerfiles available on Docker Hub. We collected ten such applications, as listed in Table 3, and deployed the latest version of each. This setup enables us to assess SQLiFuzz's ability to reveal new SQL injection vulnerabilities in widely used applications.

Table 3. Popular applications sorted by popularity. They are selected due to these criteria: providing both HTML-based GUI and OpenAPI specifications and connected to either MariaDB, MySQL, or PostgreSQL.

No.	WUT	Version	Language/ Platforms	Database	Line of Code	Github Star
B1	Appwrite	1.7.4	Typescript	MariaDB	184K	52.6K
B2	Gitea	1.24.5	Go	MySQL	370K	50.5K
B3	NextCloud	31.0.8	PHP	MySQL	778K	30.7K
B4	Bagisto	2.3.6	PHP	MySQL	1.3M	21.3K
B5	WordPress	6.8.2	PHP	MySQL	904K	20.4K
B6	NodeBB	4.5.1	Node.js	PostgreSQL	112K	14.7K
B7	Casdoor	2.37.0	Go	MySQL	77K	12.2K
B8	Prestashop	9.0.0	PHP	MySQL	2.5M	8.8K
B9	Gitlab	18.2	Ruby	PostgreSQL	4.7M	5.7K
B10	Redmine	6.0.6	Ruby	MySQL	151K	5.7K

5.4 Experimental Environment

All experiments are conducted in a cloud server, which is equipped with an Intel(R) Xeon(R) Platinum 8358P CPU @ 2.60GHz (8 cores), 32 GB of RAM, and a 1 TB SSD. The operating system is Ubuntu 24.04.3 LTS (64-bit), running Docker Engine 28.3.3 for containerized deployments. Each WUT is deployed in its own Docker container, configured with the appropriate runtime environment (e.g., PHP 8.4 with Apache) and connected to a DB proxy container, which then forwards the query

Table 4. The number of SQL injection vulnerabilities revealed by SQLiFuzz and other tools. In A3 (bWAPP), SQLiFuzz detected a slightly smaller number (explained in Section 5.5.2), but revealed more vulnerabilities in other applications. In total, SQLiFuzz uncovers previously unknown vulnerabilities in 9 endpoints from B4 (Bagisto) and B7 (Casdoor). Some WUTs do not provide OpenAPI specifications, making them not available (NA) for testing.

WUT	Known	Without SQLiFuzz				SQLiFuzz with				SQLiFuzz
No	SQLi	ZAP	Burp	Witc	Phuzz	ZAP	CustCraw	EvoM	Sche	Total
A1	2	0	0	2	2	0	2	NA	NA	2
A2	2	0	1	2	2	0	2	NA	NA	2
A3	14	0	0	12	12	0	11	NA	NA	11
A4	1	0	0	0	0	0	0	1	1	1
A5	1	0	0	0	0	0	0	1	1	1
A6	1	0	0	0	0	0	0	1	1	1
TOTAL	**21**	**0**	**1**	**16**	**16**	**15**		**3**		**18**
B1	-	0	0	0	0	0	0	0	0	0
B2	-	0	0	0	0	0	0	0	0	0
B3	-	0	0	0	0	0	0	0	0	0
B4	-	0	0	0	0	0	0	7	8	8
B5	-	0	0	0	0	0	0	0	0	0
B6	-	0	0	0	0	0	0	0	0	0
B7	-	0	0	0	0	0	0	0	1	1
B8	-	0	0	0	0	0	0	0	0	0
B9	-	0	0	0	0	0	0	0	0	0
B10	-	0	0	0	0	0	0	0	0	0
TOTAL NEW		**0**				**0**		**9**		**9**

Legend: Witc = Witcher, EvoM = EvoMaster, Sche = Schemathesis, CustCraw = Our Custom Crawler.

to a MySQL or PostgreSQL, depending on application requirements. SQLiFuzz and MITMProxy (Reverse proxy) run directly on the host machine in Python 3.12.3, while the DB Proxy operates in a Docker container and is connected to the same Docker network as WUT containers. The DB proxy shares the queries and DBMS responses via a mounted volume.

Our SQLiFuzz experiment runs without time restrictions, finishing when all processes are complete. Crawlers and API scanners generally do not require the user to set a specific timeout to limit their processes, as they typically continue until all links are visited. In the fuzzing phase, we repeated the mutation 70 times for each potentially vulnerable request, corresponding to the 70 unusual characters and common SQLi payloads that typically trigger SQL syntax errors (see Section 3.4.2).

5.5 Result and Discussion

We evaluate the proposed fuzzer's ability to detect previously reported vulnerabilities and to uncover previously unknown SQLi vulnerabilities. The results are summarized in Table 4. In addition, we manually analyze each target application to identify cases where the fuzzer did not succeed in detecting existing vulnerabilities.

5.5.1 Detection of Known Vulnerabilities. We first assess whether SQLiFuzz can successfully detect documented SQL injection vulnerabilities from security benchmark datasets. In bWAPP, which

contains 14 suitable SQLi cases, SQLiFuzz successfully detected 11. Witcher and Phuzz performed slightly better, each detecting 12 cases. In contrast, OWASP ZAP and Burp Suite failed to detect any vulnerabilities. We attribute this to their crawler limitations: both tools might not retrieve all web pages in bWAPP, which prevented them from reaching the vulnerable endpoints. On the other two applications, DVWA and XVWA, SQLiFuzz detected the same number of vulnerabilities as Witcher and Phuzz.

Secondly, we evaluate the ability of SQLiFuzz to detect known SQL injection vulnerabilities from recent CVE reports. In the WordPress cases (CVE-2025-7670 and CVE-2025-6783) and NodeBB case (CVE-2025-50979), SQLiFuzz was able to identify the malformed queries triggered by malformed API requests that were missed by the other tools. None of the other tools detected these vulnerabilities, either because they could not collect the vulnerable requests or because their internal heuristics excluded them from testing.

5.5.2 Cases Missed by SQLiFuzz. We manually verified three missed cases in A3 (bWAPP): *GET /sqli_5.php*, *POST /sqli_8-2.php*, and *GET /sqli_9.php*. Each involves mechanisms outside SQLiFuzz's current scope. The first case uses a WS/SOAP interface, where the submitted SQL query (e.g., *"SELECT * FROM movies"*) contains no directly injected parameter data. The next case encodes the HTTP body in XML without declaring a `Content-Type` header, preventing SQLiFuzz from correctly extracting parameter values. The last one protects access with CAPTCHA validation, which blocked the crawler from reaching the vulnerable page. These cases illustrate practical limitations in handling SOAP protocols, undeclared specific data formats, and pages guarded by human-interaction challenges, which will be explained in Section 6.

> According to the experiments about known vulnerabilities, out of 21 evaluated cases, SQLiFuzz raised 18 True Positive (TP) Results and 3 False Negative (FN) Results (missed vulnerability), without having False Positive (FP) Results. This corresponds to a TP rate of 85.7%, a FP rate of 0%, and a FN rate of 14.3%. These results indicate that SQLiFuzz achieves high detection accuracy without incorrectly flagging non-vulnerable cases, which are largely attributed to the use of input-query correlation and DBMS parsing error signals.

5.5.3 Discovery of New Vulnerabilities. During the evaluation of popular open-source applications, SQLiFuzz uncovered 9 previously unreported SQL injection vulnerabilities, even though these vulnerabilities are hard to exploit. Specifically, SQLiFuzz found 8 vulnerable endpoints in Bagisto (B4) and 1 vulnerable endpoint in Casdoor (B7). Interestingly, these issues were triggered through the web API, not the web GUI. These findings align with our preliminary analysis, which emphasises that recent real-world vulnerabilities increasingly emerge at less visible entry points, such as APIs. Other tools likely missed these cases due to their focus on GUI-based requests or URL-extension–based request selection.

All newly discovered vulnerabilities have been responsibly disclosed to the respective maintainers. Bagisto maintainers have acknowledged these issues, which are SQL syntax errors caused by fuzzed values successfully modifying the query structures through vulnerable endpoints. However, they do not consider these errors exploitable SQL injection vulnerabilities due to the difficulty of exploitation. To further investigate, we conducted a deep analysis and found that Bagisto applies backtick-based sanitization to all user-controlled values across these vulnerable endpoints. While this mechanism mitigates many straightforward attacks, SQLiFuzz has proved that it is not fully robust: **a single asterisk (*) character was sufficient to bypass the sanitization and alter the query structure**. Although we are unable to construct a full exploit because the alphanumeric characters are sanitized, this behavior indicates improper input handling and suggests latent injection risks. We are currently

awaiting a response from the Casdoor maintainers, where we observed similar patterns related to the use of this backtick-based sanitization.

5.5.4 Result Stability. Repeated three times, our experiments show that there was no significant variance in the number of detected SQLi cases. It is primarily caused by the stability of the crawlers or API scanners used, as well as certain character prioritization in the mutation phase. Although some crawlers or API scanners generate more requests than others due to their crawling/scanning strategies, they, along with the request analyser, yield a quite consistent number of potentially vulnerability-triggering requests across all experiments. To make these requests produce a syntax error in the DBMS, the fuzzer uses the apostrophe or the asterisk as the prioritized character to be inserted in the mutation cycle. This workflow brings a consistent result.

5.5.5 Proxy Overhead. We evaluate the runtime overhead introduced by the DB proxy when capturing submitted queries and DBMS responses. To this end, we use the final corpus of our last experiments (Section 5.5.1 and 5.5.3), which contains requests whose parameter values are reflected in the generated SQL queries (as described in Section 3.2). In this overhead experiment, these requests serve as the seed, which constitutes the workload for our overhead measurement, as they have been confirmed to be processed by the DBMS. To further increase the workload size, we also include the fuzzed requests generated during the mutation phase of the previous experiments. We compare two experimental configurations: one using the full SQLiFuzz setup with the DB proxy enabled, and another in which the WUT connects directly to the DBMS without the proxy. Each request from the seed is submitted to both configurations, and the end-to-end processing time is measured from the client side using the `perf_counter` method. In addition, the number of query-response pairs captured by the DB proxy is recorded to observe the DBMS activity. To eliminate interference from concurrent execution, requests are submitted sequentially to the WUT.

As shown in Table 5, the introduction of the DB proxy results in only a slight increase in response time (around 0.02145 seconds on average), indicating that the proxy incurs minimal performance overhead. The overhead is introduced principally because the DB proxy logs (i.e., writes to a text file) the submitted queries and the corresponding responses from the DBMS. The overhead varies across requests because each request processed by the WUT can trigger a different number of database queries, with varying lengths.

6 Current Limitations of the Approach

In this section, we highlight three limitations of SQLiFuzz, mostly based on the experimental results, that need to be addressed in future work.

6.1 Web Protocol Generality

Due to limitations in its protocol generality, SQLiFuzz failed to detect two known SQL injection cases, namely *GET /sqli_5.php* and *POST /sqli_8-2.php* (see Section 5.5.2). This limitation stems from the assumptions about communication protocols and request formats, which constrain the approach's generality in certain scenarios. SQLiFuzz is currently designed to operate on widely used HTTP request formats, including URL parameters, JSON bodies, and form-encoded data. Interfaces that rely on alternative protocols or encodings, such as SOAP or XML, often embed parameters in structured payloads, which require protocol-specific parsing and semantic understanding to be implemented. Without explicit support for these formats, the fuzzer cannot reliably identify injection points or generate valid mutated requests. While extending SQLiFuzz to support additional protocols is feasible, doing so would require incorporating specialized parsers, schema awareness, and protocol-specific mutation strategies. Addressing richer protocol support remains an important direction for future research.

Table 5. Average request processing time (in seconds) of the considered WUTs, including the overhead introduced by the DB proxy. Overall, the overhead per-request is minimal and varies depending on the number of executed SQL queries, their lengths, and the size of the responses captured by the proxy.

WUT No	Num. Request	Num. SQL Pairs of Query-Response	Average Processing Time		Overhead per Request
			With Proxy	W/O Proxy	
A1	901	1,185	0.01512	0.01068	0.00444
A2	1,840	677	0.01572	0.01023	0.00549
A3	29,549	1,772	7.24518	7.24508	0.00010
A4	11,645	420,621	0.08683	0.04659	0.04024
A5					
A6	12,804	231,811	0.01735	0.01307	0.00428
B1	11,900	252,921	0.02007	0.01559	0.00448
B2	19,500	72,354	0.01578	0.00768	0.00810
B3	27,000	152,404	0.04935	0.03658	0.01277
B4	18,775	461,178	0.14779	0.14094	0.00685
B5	32,618	1,966,831	0.07011	0.04309	0.02702
B6	13,100	196,230	0.01254	0.00615	0.00639
B7	119,141	626,845	0.01550	0.00646	0.00904
B8	14,253	4,243,297	1.05910	0.87568	0.18342
B9	10,138	136,031	0.03927	0.03412	0.00515
B10	119,601	421,185	0.01450	0.01051	0.00399

6.2 Dependence on Crawler- and Scanner-Generated Traffic

Apart from the two failed cases due to protocol limitations, SQLiFuzz also failed to reveal a known SQL injection vulnerability in the endpoint *GET /sqli_9.php*. The reason is that this endpoint is protected by a CAPTCHA mechanism. Because the web crawler cannot automatically solve CAPTCHA challenges, it cannot access the vulnerable page. As a result, no corresponding HTTP request is generated and forwarded to SQLiFuzz for analysis and fuzzing, giving SQLiFuzz no opportunity to detect the vulnerability.

This case implies that the overall effectiveness of SQLiFuzz is generally bounded by the coverage achieved by the underlying crawling and API exploration tools. As explained in Section 3, SQLiFuzz relies on the web traffic generated by external web crawlers and API scanners to discover and test potential attack surfaces. The *Request Analyzer* component operates on HTTP requests that have already been generated during the crawling or scanning phase. Consequently, if a crawler or scanner fails to reach a vulnerable endpoint, SQLiFuzz will not receive the corresponding request for further analysis and fuzzing. Specifically, endpoints that require complex authentication flows, multi-step workflows, or specific input formats may remain unexplored if the crawler or scanner cannot satisfy such preconditions.

Enhancing crawler and scanner capabilities, or integrating more advanced exploration techniques, would directly increase the effectiveness of SQLiFuzz. We consider the integration of more advanced exploration techniques an important direction for future work.

6.3 Dependence on API Specification Completeness

Modern web applications often expose RESTful or GraphQL APIs alongside traditional GUI-based entry points. Although API specifications such as OpenAPI or Swagger are intended to facilitate

```
/jalw/v1/archive/{year}:
  get:
    responses:
      '200':
        description: OK
    parameters:
      - name: year
        in: path
        description: ''
        required: true
        schema: {}
```

```
/jalw/v1/archive/{year}/{month}:
  get:
    responses:
      '200':
        description: OK
    parameters:
      - name: year
        in: path
        description: ''
        required: true
        schema: {}
      - name: month
        in: path
        description: ''
        required: true
        schema: {}
```

Fig. 3. Examples of missing `example` fields in an OpenAPI specification. Without these hints, API scanners struggle to generate valid values for `year` and `month`. When the server enforces strict validity checks (e.g., 4-digit years, 2-digit months), the scanners consistently submit invalid values that lead to server rejection.

automated testing, they are often incomplete or outdated, as also recently observed by Pan et al. [32]. Undocumented endpoints or incomplete schema definitions limit API scanners' coverage, preventing them from accessing certain functionalities, which eventually affects SQLiFuzz's performance.

In addition to incomplete specifications, current API scanners rely heavily on the *example* fields of API specifications to generate valid parameter values or request bodies. When these fields are missing, scanners often fail to produce acceptable values that are in turn rejected by the WUT, leaving critical functions untested. For instance, in the case of CVE-2025-7670 (see Figure 3), the vulnerable endpoint expects a year parameter in a specific format. While a human tester can easily infer the correct input, neither EvoMaster nor Schemathesis could generate valid values without explicit examples in the specification. When the server performs validity checks on each value (i.e., 4-digit years and 2-digit months), both tools consistently received server rejections for invalid URL paths, thereby missing the vulnerability.

In summary, the effectiveness of API scanners is tightly coupled with the completeness and quality of the provided API documentation. To further enhance coverage in complex API-driven applications, future work can combine API fuzzing with large language models (LLMs) to infer plausible parameter values when API specifications are incomplete.

7 Threats to Validity

While this study demonstrates the effectiveness of SQLiFuzz, some threats to validity must be considered when interpreting the evaluation results.

7.1 Internal Validity

The internal threat comes from a slight modification to the API specifications officially released by the WUT developers. This modification is necessary because the specifications are often incomplete, preventing API scanners from reaching the expected endpoints, as explained in Section 6.3. As another example, beyond the case discussed in the previous section, the vulnerable endpoint `api/v3/search/categories?search=«keyword»` in *NodeBB* (A6) is not documented in the official specification. To proceed, we had to manually extend the specification by adding the missing part. We used Generative AI services (i.e., GitHub Copilot) to analyze CVE reports and affected code, and produce the endpoint specification, which we then added to the official specification.

This manual intervention represents a threat to validity because it slightly reduces the degree of automation highlighted by API-based scanning approaches. However, this effort was limited

to completing the specification and did not involve the modification of SQLiFuzz or the WUT. Moreover, this limitation is not specific to SQLiFuzz but reflects a broader dependency of current API scanners on the correctness of available API specifications.

7.2 External Validity

An external validity threat arises from the dataset selection. Several of the evaluated SQL injection vulnerabilities (i.e., three of which are discussed in Section 2.1) are exposed through APIs rather than traditional GUI-based interfaces. Therefore, existing scanners, which are primarily designed for GUI-centric testing, may appear disadvantaged in these scenarios.

This dataset selection is intentional to evaluate the effectiveness of current SQL injection testing tools against recent and increasingly prevalent attack vectors in modern web applications. As web development practices shift toward API-driven and microservice-based architectures, vulnerabilities are more likely to arise in non-GUI entry points that fall outside the original design scope of many existing scanners. To mitigate potential bias, the dataset is not exclusively composed of API-based vulnerabilities. Our evaluation also includes several testbed applications commonly used in prior SQL injection testing studies, enabling comparison with existing tools in more traditional, GUI-centric settings.

Nonetheless, our current dataset may still introduce bias because it does not include a broader set of recent CVEs, which affects the generality of our conclusions.

8 Related Works

Several lines of research are closely related to SQLiFuzz. In the context of testing for SQL injection-related vulnerabilities, prominent tools include OWASP ZAP [31], Burp Suite [34], sqlmap [35], Phuzz [27], Witcher [37], Atropos [12], and Predator [40]. While these tools have demonstrated effectiveness in conventional settings, they face limitations that SQLiFuzz seeks to overcome (as explained in Section 2.2).

For a broader discussion, several prior works have proposed components similar to those used in SQLiFuzz, although their goals and assumptions differ. In terms of the proxy component, our work is similar to SQLProb [19], which aims to prevent SQL injection attacks by deploying a DB proxy that filters out malicious queries. It extracts user inputs from submitted queries and applies a genetic algorithm to validate them based on some predefined attack patterns [13], ensuring only benign inputs are passed to the DBMS. In contrast, SQLiFuzz focuses on vulnerability testing rather than prevention (through filtering) and does not rely on prior knowledge of attack patterns. Our proxy component forwards all queries to the DBMS and leverages the DBMS parser as an oracle by injecting lightweight mutations (e.g., unexpected characters) into user inputs to trigger parsing errors. These parsing errors are suitable for detecting SQLi vulnerabilities, since they indicate that the query structure has been unexpectedly changed.

Regarding oracle verification, our work is similar to CANDID [4], which detects SQL injection attacks by comparing the structure of the program's intended query with the structure of the actual query issued. CANDID modifies the WUT to compute a symbolic query that captures the intention of the programmer. Another related work, Rivulet [15], performs a semantic check on a tainted query to verify whether the tainted query becomes part of the SQL structure, thus indicating SQL injection vulnerabilities. In contrast to checking the structure or semantics of a query using a new method, our work checks the query syntax using the DBMS parser, which is faster and more efficient because it does not involve code modification.

In terms of mutation strategy, our work is closely related to Ardilla [17] and Rivulet [15], both of which mutate only parts of the input reflected in SQL queries. To identify these parts, these tools track the flow of tainted data through the application and determine whether tainted data can

reach sensitive sinks. Our work adopts the same underlying concept however it employs string comparison rather than dynamic taint tracking in doing so. Although string comparison is less precise, it is significantly faster and well-suited to a proxy-based setting, making it effective for narrowing down candidates of potentially vulnerable HTTP requests. Furthermore, while string comparison is not applicable when inputs are encoded, hashed, or converted into binary formats, this limitation is acceptable in our context because such transformations typically yield sanitized or structurally constrained strings that are unlikely to induce SQL syntax errors.

In the field of web fuzzing, SQLiFuzz shares similarities with VoAPI^2 [9] and Nautilus [5], which analyze API interfaces for potential vulnerabilities. However, while VoAPI^2 and Nautilus focus primarily on OpenAPI specifications, SQLiFuzz leverages internal SQL queries and database responses to identify potentially vulnerable APIs. In addition, SQLiFuzz works similarly to existing tools that target APIs, such as EvoMaster [2], Restler [3], RestTestGen [39], and Schemathesis [14]; however, they are designed to detect web crashes rather than SQLi. SQLiFuzz is also related to work on fuzzing microservice-based web applications, such as MacroHive [11] and MicroFuzz [8]. These approaches monitor communication across distributed services, whereas SQLiFuzz specifically observes the interaction between services and the DBMS, thereby reducing configuration overhead and improving general applicability.

9 Conclusion and Future Work

In this paper, we presented SQLiFuzz, a universal SQL injection fuzzer designed to detect vulnerabilities across both GUI-based and API-based web applications. Unlike existing approaches, SQLiFuzz introduces three key distinguishing features: (1) a reverse proxy that integrates request collection and fuzzing into a unified process, (2) a DB proxy that enables reliable request–query matching and serves as a simple-to-deploy SQL injection oracle, and (3) a feedback-driven fuzzing campaign that prioritizes potentially vulnerable requests and parameters. Our empirical evaluation on security benchmarks and real-world applications demonstrates the effectiveness of SQLiFuzz. It successfully detected the majority of known SQL injection vulnerabilities and uncovered previously unknown vulnerabilities in popular open-source projects that existing tools missed.

For future work, combining SQLiFuzz with an AI-assisted API scanner could be a promising research direction because it could reduce the reliance on example fields and improve testing coverage.

Acknowledgements

This work was financially supported by the Indonesian Education Scholarship (BPI) from the Center for Higher Education Funding and Assessment (PPAPT) and the Indonesia Endowment Fund for Education (LPDP).

Data Availability

To promote transparency and reproducibility, we follow the ACM SIGSOFT Open Science policies.

(1) ***Artifacts***: The implementation of SQLiFuzz, including the fuzzer, reverse proxy, and DB proxy, is available in our public repository (https://github.com/websecfuzz/SQLiFuzz). The repository also contains scripts for setting up the evaluation environment.
(2) ***Benchmarks***: We evaluate SQLiFuzz on publicly available benchmark applications, including bWAPP, DVWA, and XVWA. Instructions and setup scripts are included in the artifact package.
(3) ***Real-world applications***: For all real-world applications, we provide source code, configuration files, and instructions for reproducing our experiments with SQLiFuzz.

References

[1] Appwrite Contributors. 2025. Appwrite: End-to-End Backend Server for Web, Mobile, and Flutter Developers. https://appwrite.io. Accessed: 2025-09-12.

[2] Andrea Arcuri. 2018. EvoMaster: Evolutionary Multi-context Automated System Test Generation. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Västerås, Sweden, 394–397. doi:10.1109/ICST.2018.00046

[3] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, Canada, 748–758. doi:10.1109/ICSE.2019.00083 ISSN: 1558-1225.

[4] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. 2007. CANDID: preventing sql injection attacks using dynamic candidate evaluations. In *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, Alexandria Virginia USA, 12–24. doi:10.1145/1315245.1315249

[5] Gelei Deng, Zhiyi Zhang, Yuekang Li, Yi Liu, Tianwei Zhang, Yang Liu, Guo Yu, and Dongjin Wang. 2023. NAUTILUS: Automated RESTful API Vulnerability Detection. In *Proceedings of the 32nd USENIX Security Symposium*. USENIX Association, Anaheim, CA, USA, 5593–5609. https://www.usenix.org/conference/usenixsecurity23/presentation/deng-gelei

[6] I Putu Arya Dharmaadi, Mohannad Alhanahnah, Van-Thuan Pham, Fadi Mohsen, and Fatih Turkmen. 2025. BACFuzz: Exposing the Silence on Broken Access Control Vulnerabilities in Web Applications. arXiv:2507.15984 [cs.CR] https://arxiv.org/abs/2507.15984

[7] I Putu Arya Dharmaadi, Elias Athanasopoulos, and Fatih Turkmen. 2025. Fuzzing frameworks for server-side web applications: a survey. *International Journal of Information Security* 24, 2 (Feb. 2025), 73. doi:10.1007/s10207-024-00979-w

[8] Peng Di, Bingchang Liu, and Yiyi Gao. 2024. MicroFuzz: An Efficient Fuzzing Framework for Microservices. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24)*. Association for Computing Machinery, New York, NY, USA, 216–227. doi:10.1145/3639477.3639723

[9] Wenlong Du, Jian Li, Yanhao Wang, Libo Chen, Ruijie Zhao, Junmin Zhu, Zhengguang Han, Yijun Wang, and Zhi Xue. 2024. Vulnerability-oriented Testing for RESTful APIs. In *Proceedings of the 33rd USENIX Security Symposium*. USENIX Association, Philadelphia, PA, USA, 739–755. https://www.usenix.org/conference/usenixsecurity24/presentation/du

[10] DVWA. 2009. Damn Vulnerable Web Application (DVWA). https://github.com/digininja/DVWA. Accessed: 2025-05-04.

[11] Luca Giamattei, Antonio Guerriero, Roberto Pietrantuono, and Stefano Russo. 2024. Automated functional and robustness testing of microservice architectures. *Journal of Systems and Software* 207 (Jan. 2024), 111857. doi:10.1016/j.jss.2023.111857

[12] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, Cemal Kaygusuz, and Thorsten Holz. 2024. Atropos: Effective Fuzzing of Web Applications for Server-Side Vulnerabilities. In *Proceedings of the 33rd USENIX Security Symposium*. USENIX Association, Philadelphia, PA, USA, 4765–4782. https://www.usenix.org/conference/usenixsecurity24/presentation/gÃijler

[13] William G. J. Halfond and Alessandro Orso. 2005. AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (Long Beach, CA, USA) *(ASE '05)*. Association for Computing Machinery, New York, NY, USA, 174–183. doi:10.1145/1101908.1101935

[14] Zac Hatfield-Dodds and Dmitry Dygalo. 2022. Deriving semantics-aware fuzzers from web API schemas. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 345–346. doi:10.1145/3510454.3528637

[15] Katherine Hough, Gebrehiwet Welearegai, Christian Hammer, and Jonathan Bell. 2020. Revealing injection vulnerabilities by leveraging existing tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, Seoul South Korea, 284–296. doi:10.1145/3377811.3380326

[16] Soheil Khodayari, Thomas Barber, and Giancarlo Pellegrino. 2024. The Great Request Robbery: An Empirical Study of Client-side Request Hijacking Vulnerabilities on the Web. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 166–184. doi:10.1109/SP54263.2024.00098

[17] Adam Kieyzun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. 2009. Automatic creation of SQL Injection and cross-site scripting attacks. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, Vancouver, BC, Canada, 199–209. doi:10.1109/ICSE.2009.5070521

[18] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated test generation for REST APIs: no time to rest yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Virtual South Korea, 289–301. doi:10.1145/3533767.3534401

[19] Anyi Liu, Yi Yuan, Duminda Wijesekera, and Angelos Stavrou. 2009. SQLProb: a proxy-based architecture towards preventing SQL injection attacks. In *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, Honolulu Hawaii, 2054–2061. doi:10.1145/1529282.1529737

[20] Malik Mesellem. 2014. bWAPP, an extremely buggy web app. http://www.itsecgames.com/download.htm. Accessed: 2025-05-04.
[21] Microsoft. 2025. Playwright for Python. Available at: https://playwright.dev/python/ (Accessed: 2025-01-21).
[22] Miguel Useche. 2025. JS Archive List. Available at: https://wordpress.org/plugins/jquery-archive-list-widget/ (Accessed: 2025-05-02).
[23] mitmproxy Project. 2024. mitmproxy - an interactive HTTPS proxy. Available at: https://mitmproxy.org/ (Accessed: 2024-11-01).
[24] MITRE. 2025. CVE-2025-50979: SQL Injection in NodeBB Search-Categories API Endpoint. https://nvd.nist.gov/vuln/detail/CVE-2025-50979. NodeBB v4.3.0 vulnerable; search parameter not properly sanitized. CVSS v3.1 base score 8.6 (HIGH). Published 2025-08-27..
[25] MITRE. 2025. CVE-2025-6783: SQL Injection in GoZen Forms WordPress Plugin. https://nvd.nist.gov/vuln/detail/CVE-2025-6783. Versions 1.1.5 of the GoZen Forms plugin vulnerable. Published 2025-07-04..
[26] MITRE. 2025. CVE-2025-7670: Time-based SQL Injection in JS Archive List WordPress Plugin. https://nvd.nist.gov/vuln/detail/CVE-2025-7670. JS Archive List 6.1.5 plugin vulnerable. CVSS 3.1 base score 7.5 (HIGH). Published 2025-08-19..
[27] Sebastian Neef, Lorenz Kleissner, and Jean-Pierre Seifert. 2024. What All the PHUZZ Is About: A Coverage-guided Fuzzer for Finding Vulnerabilities in PHP Web Applications. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. ACM, Singapore Singapore, 1523–1538. doi:10.1145/3634737.3661137
[28] nodeBB. 2025. NodeBB. Available at: https://github.com/NodeBB/NodeBB (Accessed: 2025-05-02).
[29] Olivier Nourry, Yutaro Kashiwa, Weiyi Shang, Honglin Shu, and Yasutaka Kamei. 2025. My Fuzzers Won't Build: An Empirical Study of Fuzzing Build Failures. *ACM Transactions on Software Engineering and Methodology* 34, 2 (Feb. 2025), 1–30. doi:10.1145/3688842
[30] optinlyhq. 2025. GoZen Forms. Available at: https://wordpress.org/plugins/gozen-forms/ (Accessed: 2025-05-02).
[31] OWASP Foundation. 2025. Zed Attack Proxy (ZAP). Available at: https://www.zaproxy.org/ (Accessed: 2025-01-15).
[32] Lianglu Pan, Shaanan Cohney, Toby Murray, and Van-Thuan Pham. 2025. Trailblazer: Practical End-to-end Web API Fuzzing (Registered Report). In *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Clarion Hotel Trondheim, Trondheim, Norway) *(ISSTA Companion '25)*. Association for Computing Machinery, New York, NY, USA, 143–152. doi:10.1145/3713081.3731717
[33] Stephan Plöger, Mischa Meier, and Matthew Smith. 2023. A Usability Evaluation of AFL and libFuzzer with CS Students. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 186, 18 pages. doi:10.1145/3544548.3581178
[34] PortSwigger. n.d.. Burp Suite – Application Security Testing Software. https://portswigger.net/burp. Accessed: 2025-05-04.
[35] sqlmap. 2024. sqlmap. Available at: https://github.com/sqlmapproject/sqlmap (Accessed: 2024-10-02).
[36] Aleksei Stafeev and Giancarlo Pellegrino. 2024. SoK: State of the Krawlers – Evaluating the Effectiveness of Crawling Algorithms for Web Security Measurements. In *Proceedings of the 33rd USENIX Security Symposium*. USENIX Association, Philadelphia, PA, USA, 719–737. https://www.usenix.org/conference/usenixsecurity24/presentation/stafeev
[37] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Yan Shoshitaishvili, and Adam Doupé. 2023. Toss a Fault to Your Witcher: Applying Grey-box Coverage-Guided Mutational Fuzzing to Detect SQL and Command Injection Vulnerabilities. In *2023 IEEE Symposium on Security and Privacy (SP) (44)*. IEEE Computer Society, SAN FRANCISCO, 2658–2675. doi:10.1109/SP46215.2023.00007
[38] Orpheas van Rooij, Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos. 2021. webFuzz: Grey-Box Fuzzing for Web Applications. In *Computer Security – ESORICS 2021 (Lecture Notes in Computer Science)*, Elisa Bertino, Haya Shulman, and Michael Waidner (Eds.). Springer International Publishing, Cham, 152–172. doi:10.1007/978-3-030-88418-5_8
[39] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. RESTTESTGEN: Automated Black-Box Testing of RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Porto, Portugal, 142–152. doi:10.1109/ICST46399.2020.00024 ISSN: 2159-4848.
[40] Chenlin Wang, Wei Meng, Changhua Luo, and Penghui Li. 2025. Predator: Directed Web Application Fuzzing for Efficient Vulnerability Validation. In *2025 IEEE Symposium on Security and Privacy (SP)*. 886–902. doi:10.1109/SP61157.2025.00066
[41] wordpress. 2025. WordPress. Available at: https://wordpress.org/ (Accessed: 2025-05-02).
[42] XVWA. 2013. Xtreme Vulnerable Web Application (XVWA). https://github.com/s4n7h0/xvwa. Accessed: 2025-05-04.
[43] Man Zhang and Andrea Arcuri. 2023. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *ACM Transactions on Software Engineering and Methodology* 32, 6 (Nov. 2023), 1–45. doi:10.1145/3597205

Received 2025-09-12; accepted 2026-03-24