National College of Ireland

Context Aware & Session-Integrated Directed Fuzzing for SQL Injection Detection in Modern Web Applications

MSc Research Project
CYBERSECURITY

KESAVA NAGA SIVA SAI MATLAPARTHI
Student ID: 23405210

School of Computing
National College of Ireland

Supervisor: KAMIL MAHAJAN

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Kesava Naga Siva Sai Matlaparthi
Student ID:	23405210
Programme:	CyberSecurity **Year:** 2025
Module:	MSc (Research) Practicum Part 2
Supervisor:	Kamil Mahajan
Submission Due Date:	28/01/2026
Project Title:	Context Aware and Session Integrated Directed Fuzzing for SQL Injection Detection in Modern Web Applications
Word Count:	7891 **Page Count:** 23(Including References)

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Kesava Matlaparthi

Date: 28/01/2026

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Context Aware & Session-Integrated Directed Fuzzing for SQL Injection Detection in Modern Web Applications

KESAVA NAGA SIVA SAI MATLAPARTHI
23405210

Abstract

SQL injection (SQLi) remains a serious threat to modern web applications, particularly when vulnerabilities are reachable only after authentication and hidden behind complex session state. Existing tools either rely on heavyweight program analysis and instrumentation, or on large generic payload dictionaries that generate substantial traffic and noise. This thesis investigates whether a deliberately lightweight, aware of the context directed fuzzing approach, integrated with session and token handling and a minimal feedback loop, can efficiently detect SQLi in such settings. The proposed fuzzer automatically logs in, preserves authenticated sessions, and targets only the 1–3 parameters per endpoint that are most likely to reach database queries. It exercises a compact staged set of boolean, error-based and time-based payloads and uses simple content, error and timing oracles to stop as soon as a strong, repeatable signal is observed. The prototype is evaluated on two authenticated lab targets running in Docker: the SQL injection module of Damn Vulnerable Web Application (DVWA) and an authenticated search end- point in OWASP Juice Shop. On DVWA, the fuzzer reliably confirms true SQLi with essentially a single request, achieving millisecond-level Time-to-First-Finding (TTFF) and Requests-per-Finding (RPF) close to 1. On Juice Shop, it maintains valid authentication and quickly surfaces strong oracle-level anomalies on likely database-touching parameters, again with low TTFF and RPF, although these do not correspond to confirmed SQLi. The results show that a modest amount of context awareness and feedback is sufficient to obtain fast, informative signals.

Keywords: SQL injection; web application security; directed fuzzing; authenticated fuzzing; session management; grey-box testing.

1 Introduction

1.1 Background and Motivation

Web applications support most modern services, from banking and e-commerce to health- care and public administration. Their backends are typically built on top of relational databases, ORM frameworks, and complex authentication and authorization mechanisms. While these technologies have improved developer productivity, they have also increased the attack surface: a single vulnerable endpoint can expose highly sensitive, authenticated functionality to remote attackers. SQL injection (SQLi) has been among the most serious types of web vulnerabilities. Support secure coding standards, ORM abstractions, and defensive structures, SQLi has still been seen in practice and vulnerability reports. The principle that causes the problem is very easy to grasp: untrusted input is included in SQL queries without proper validation or parameterization, but in reality it is hard to find exploitable points of injection in a massive, with saved state web application. In modern applications, routing, middleware, and business logic operate on more than one layer and SQL queries are commonly invoked indirectly via a framework or library. In addition, administrative interfaces, account management flows, and internal search or reporting endpoints (some of the most security- critical paths) typically require successful authentication to access. Practical security testing is often based on a set of tools such as static analysis, manual penetration testing, and black-box web vulnerability scanners. Static analyzers are able to reason about code-level control and data flow, but usually suffer from false positives and cannot model run-time behaviour or frameworks. Penetration testing proxies (e.g. Burp Suite, OWASP ZAP, sqlmap) and black-box scanners connect to the application via HTTP and attempt to identify vulnerabilities based on the responses obtained. In the case of SQLi, they typically submit huge dictionaries of payloads to all the parameters they can identify, seeking typical error messages, differences in content, or timing variations. While these approaches are valuable, they face growing challenges in modern environments:

- **with saved stateness and sessions:** web applications maintain complex session state across requests (cookies, server-side sessions, JWTs, CSRF tokens), and many endpoints are only reachable behind authentication.
- **Structured inputs:** inputs are no longer arbitrary byte streams but structured HTTP requests, HTML forms, JSON bodies, and RESTful APIs with type constraints and semantic relationships.
- **Non-crashing vulnerabilities:** unlike classic memory corruption in native binaries, web vulnerabilities such as SQLi rarely manifest as process crashes; they instead appear as subtle changes in output, error messages, or database side effects.

However, there is still limited empirical evidence on how far one can go with a deliberately simple design that embodies these ideas *without* static analysis, *without* deep learning, and *without*

heavy instrumentation on realistic, authenticated web targets.

1.2 Research Question

How can a lightweight, aware of the context directed fuzzing approach, integrated with session/token handling and a minimal feedback loop, effectively detect SQLi in modern, authenticated web applications?

1.3 Research Objectives

The main research objectives are:

- Build a modular fuzzer that maintains logged-in sessions.
- Focus the fuzzer on testing only the most vulnerable-looking parameters.
- Define simple metrics to measure how quickly and efficiently the fuzzer finds vulnerabilities.
- Test the fuzzer on common, deliberately vulnerable web applications and compare it to other tools.
- Analyze the results to see which strategies work best and identify the approach's limitations.

2 Literature Review

2.1 Challenges in Fuzzing Modern Web Applications

Conventional fuzz testing methods that have been so successful at detecting memory corruption bugs in native code (e.g., AFL on binaries) are difficult to apply to web applications. Web applications are with saved state and organized, i.e., they have complex session state (cookies, server-side sessions, tokens), and accept inputs in structured for- mats (HTTP requests with particular parameters) instead of arbitrary bytes. Guler et al. (2024) stress that, unlike binary programs, web applications contain persistent state (databases, sessions) and highly structured input – simple bit-flipping in the HTTP re- quest is not effective.

The bugs found on the web are not usually experienced as crashes. Problems such as SQL injection will not segfault an application; they show up as slight alterations in the output or database operations. Therefore, a web fuzzer should be aware of the context, aware of input syntax and application state, and should have bug oracles to detect states such as an SQL error message or time delay (an indication of successful injection) (Guler et al., 2024; Trickel et al., 2023). Authentication and session management is another issue: many modern web applications need the user to be logged in and use dynamic tokens (e.g. CSRF tokens, JWTs). In historic fuzzers, this has often been overlooked or hard-coded, and it has been an open problem to

perform fuzzing behind authentication (Zhang and Arcuri, 2023). Researchers have observed that testing in practical conditions is made difficult due to the absence of standardized systems to deliver authentication details to fuzzers (Arcuri et al., 2025). Overall, the challenges of web fuzzing can be effectively addressed by thinking about the management of state, input formatting, and indicators of subtle vulnerabilities – and for this, recent studies have started showing significant progress.

2.2 Directed Fuzzing Approaches for Web Vulnerabilities

Directed fuzzing is one of the major trends in recent work, where the fuzzer is used to target one or more specific areas of a program or one or more types of vulnerabilities, instead of random exploration. This is aimed at efficiency (reduced time-to-vulnerability) by targeting hot spots that are likely to produce bugs. Binary fuzzing tools such as AFLGo (Bohme et al., 2017), which pioneered this idea, have been extended to web applications.

As an example, Yuan et al. (2024) present *sqlFuzz*, which is a directed fuzzing method that targets SQL injection detection. sqlFuzz complements grey-box fuzzing with static taint analysis to identify potentially vulnerable code – paths where user input is sent to an SQL query (the sink) without proper sanitization (Yuan et al., 2024). Instrumentation of the source code occurs at sink-reachable points and the fuzzer gives preference to seeds that exercise those points. By focusing on inputs reaching SQL execution points, sqlFuzz removes the need to waste time on portions of the application that will not trigger SQLi. According to the authors, this targeted method enabled sqlFuzz to find more SQL injection vulnerabilities than some of the standard coverage-guided fuzzers, and more quickly, under time constraints (Yuan et al., 2024).

PREDATOR by Wang et al. (2025) is another directed strategy. PREDATOR is a guided web fuzzing system that is designed to efficiently check for vulnerabilities. It employs lightweight static analysis to determine potential URLs and parameters that probably represent vulnerabilities and selectively instruments the corresponding code portions dynamically (Wang et al., 2025). By design, PREDATOR relies on the static analysis reports as a map and fuzzes only what is required in order to ascertain the presence of a potential bug. This bridges the gap between static tools which are known to have a large number of false positives and dynamic testing. Remarkably, PREDATOR was able to reach up to 43.8× higher time-to-exposure of vulnerabilities than state-of-the-art grey- box fuzzers and found 26 previously unknown bugs in real-world PHP applications (7 of which were both confirmed and patched) (Wang et al., 2025). This shows the effectiveness of directed fuzzing in reducing noise and narrowing down to actual problems.

Another similar example, focused on the input points, is WDFuzz by Lin et al. (2025). WDFuzz targets Java web applications and focuses on two main issues: searching through the vast number of possible web inputs and generating only inputs that are semantically valid. It proposes a semantic constraint extraction method to discover the anticipated format (formats, regular expressions, data types) of every parameter of the web application, and a hierarchical

scheduling model to prioritise test cases that are likely to reach deeper, code that is likely to have bugs (Lin et al., 2025). Basically, WDFuzz operates by initially learning the appearance of legitimate input values for each field, and then fuzzing those values to determine extreme cases that can lead to security problems. WD- Fuzz, evaluated on 15 real Java web applications, achieved a recall of 92.6 percent on a known-vulnerability benchmark and reported 3.2× more vulnerabilities with 7× faster detection than a baseline state-of-the-art fuzzer (Lin et al., 2025). Notably, it identified 92 additional vulnerabilities (19 of which received CVE/CNVD identifiers), which high- lights the importance of a guided strategy as a way of significantly enhancing the results of black-box testing.

CeFuzz by Zhao et al. (2022) is another purpose-built directed fuzzer that is used to identify remote code execution (RCE) vulnerabilities in specific PHP web applications. CeFuzz uses static taint analysis to identify dangerous sink functions (such as PHP's eval or system calls) and instruments those sink functions to direct fuzzing towards them(Zhao et al., 2022). It applies a concept of distance in the control flow graph to prioritise inputs that are closer to the sink. Although CeFuzz successfully identified bugs related to PHP command injection, later studies identified several weaknesses: CeFuzz's feedback was coarse-grained (it prefers seeds that execute more code, which is not necessarily correlated with vulnerability) and its instrumentation (adding numerous echo statements to trace) was static and may change program behaviour (Wang et al., 2025). Also, the fact that it is closed-source of CeFuzz and the small size of its evaluation set are limitations (Wang et al., 2025). However, it was a good step towards guided fuzzing in the web domain, and its concepts have been extended in successors such as PREDATOR and CraftFuzz. CraftFuzz (Zhao et al., 2025) is a recent development in directed fuzzing to test known vulnerabilities accurately. Instead of finding bugs randomly, CraftFuzz presupposes that you already have some potential finding (e.g., a bug report or the result of a static anal- ysis) and requires it to produce an exact proof-of-concept exploit. It uses a mix of both dynamic and static analysis performed in several phases: firstly, it runs a static route analysis to collect all the required URL endpoints and parameters that reach the vulnerable function (and will also guess or infer any additional or hidden parameters required to access that code). It then extracts and solves path constraints – basically the conditions (in branch predicates) that must be met to actually execute the vulnerable path. Guided fuzzing is then used by CraftFuzz to identify inputs that meet these conditions, and tar- geted mutation is used to solve any unsolved constraints (Zhao et al., 2025). Lastly, it introduces attack payloads at the sink to trigger the vulnerability and validates it through the application's response. The strategy is very objective-oriented: it does not explore anything that does not concern the target vulnerability. The outcomes are impressive – CraftFuzz managed to construct working exploit requests for 88.9% of the vulnerabilities it was presented with, outperforming even state-of-the-art fuzzers by an overall 32% in verification success (Zhao et al., 2025). It was also very fast, resolving each endpoint and its path constraints in seconds (a 97% success rate of reaching deep code using generated URLs). This level of accuracy is particularly desirable in enterprise environments where there is a list of potential problems and one wants to confirm them as fast

as possible with a low number of false alarms.

2.3 Grey-Box Fuzzers and Feedback-Guided Testing

Parallel to directed methods, there is a translation of grey-box fuzzing (coverage-guided fuzzing with instrumentation) to the web domain, usually incorporating more intelli- gent feedback mechanisms to identify vulnerabilities that purely coverage-based methods would fail to infer. One of the most notable examples is Witcher by Trickel et al. (2023). Witcher is essentially an AFL-like fuzzer for web applications, with significant modifica- tions: it adds fault escalation oracles to SQL and command injection. The fuzzer executes the web application in an instrumented mode and observes any indications of successful injection, such as an SQL error message or a command execution result. On detecting such a signal, Witcher intentionally crashes (e.g., by turning that semantic error into a segmentation fault in the instrumented process) to have the AFL engine consider it a discovered crash (Trickel et al., 2023). It is a clever trick that enables the coverage-guided fuzzer to handle injection findings in the same way as crashes, integrating smoothly into its workflow. Witcher also captures web-specific state components and tracks coverage so that inputs are mutated in a way that expands the code reach in the web application. Witcher discovered all of the seeded SQLi and command injections in test applications in their evaluation, and was more efficient than baseline black-box scanners (Trickel et al., 2023). Its main limitation is generality: Witcher focuses on two vulnerability types only and applies two specific oracles.

Follow-up work has aimed at extending the coverage of vulnerabilities. Guler et al. (2024) proposed a grey-box fuzzer called ATROPOS that targets eight types of server-side bugs (such as SQLi, XSS, command injection, server-side request forgery, etc.). ATROPOS addresses the limitations of state and input organisation in a number of ways. First, it has snapshot-based execution: the fuzzer can restart the web application in a clean state (including the database) between test cases, which is essential for with saved state applications (Guler et al., 2024). Second, ATROPOS uses a new feedback system at the PHP interpreter level – it hooks into the runtime to obtain internal information, such as which internal functions or stages of the parsing process were reached. This helps it create inputs that circumvent shallow validation and access deeper logic (e.g., it can construct the required parameter format or skip simple input checks by understanding how far an input has progressed in the processing pipeline) (Guler et al., 2024). Third, it specifies eight lightweight bug oracles for various vulnerability types. These oracles check certain conditions: for example, detecting when an SQL query is not executed in a properly parameterised manner (SQLi) or when dangerous functions such as exec() are invoked with user data (command injection). Crucially, these oracles are not based on crashes but on semantic indications of a vulnerability and are implemented as part of the instrumentation. As a result, ATROPOS can direct fuzzing not only by code coverage, but also by the presence or absence of vulnerability conditions being approached or triggered. The outcomes are very strong: ATROPOS detected 32% more true vulnerabilities than a state-of-the-art static analyzer for PHP, and had no false positives on the

test suite (because dynamic confirmation was built into the process) (Guler et al., 2024). It also achieved code coverage between 50% and 230% higher than previous web fuzzers such as WebFuzz (van Rooij et al., 2021) or Wfuzz (a popular black-box fuzzer), indicating that its input-generation method was effective in exercising the application. This shows the importance of integrating coverage guidance with vulnerability-specific feedback in order to make fuzzing highly effective.

Another study in grey-box web fuzzing is WebFuzz by van Rooij et al. (2021). One of the earliest efforts to use AFL in a web application context, WebFuzz targeted client-side XSS vulnerabilities. It relies on coverage feedback to produce inputs that explore different paths in a web application and explicitly checks for reflected or stored XSS by injecting payloads and seeing whether they appear unescaped in the response (van Rooij et al., 2021). WebFuzz showed that coverage guidance could be applied beyond native binaries – it did discover XSS in its target applications – although, as subsequent research observed, it did not use advanced seed selection or structure-aware mutation of web inputs (Yuan et al., 2024). In effect, it treated the web application as a black box driven purely by coverage, without optimization for web-specific patterns. As a result, it was not very efficient (it required numerous test cases to discover XSS). This was extended by Witcher and ATROPOS with additional awareness (taint tracking, semantic oracles, etc.).

In the same vein, Neef et al. (2024) introduced PHuzz, a coverage-guided fuzzer specifically built to work with web applications written in PHP. PHuzz is a modular, open- source system intended to identify as wide a range of client-side and server-side problems as possible (Neef et al., 2024). The authors recognize that a large number of vulnerability classes in web applications (SQLi, command injection, XSS, file inclusion, etc.) may be discovered through fuzzing, provided that the fuzzer can learn how to communicate with the application. PHuzz integrates into the PHP interpreter and can add payload gener- ation policies to handle various bug classes. Neef et al. state that they applied PHuzz to more than 1,000 API endpoints across 115 popular WordPress plugins, reporting more than 20 security vulnerabilities and two CVEs (Neef et al., 2024). This real-world test highlights that grey-box fuzzing, when tailored to the web context, is feasible and can identify severe bugs in current software. Another important contribution of PHuzz is that it is an open-source tool, reducing the effort required by other researchers or practitioners to fuzz PHP applications (which relates to the concept of reproducible, common frameworks discussed later).

These types of grey-box fuzzers often use code coverage as a measure of progress, although other works have demonstrated that additional feedback cues can substantially enhance such fuzzers. We have precedents: Witcher's fault signals, ATROPOS's interpreter-level knowledge, and similar techniques. Taint-flow information is another type of feedback. Several works add taint tracking to understand whether an input injected into the system reaches sensitive sinks. For example, Dharmaadi et al. (2025) observe that it is possible to improve a fuzzer with taint feedback to identify additional SQLi and command injection vulnerabilities – in effect, dynamic taint analysis can be used as an oracle to signal when a payload has been passed to a risky API

without appropriate checking (this technique has been foreshadowed by previous work and is increasingly common) (Dharmaadi et al., 2025). With taint feedback and coverage guidance, a fuzzer can concentrate its efforts on inputs that not only access new code but also inject data into security-sensitive locations. This approach minimises false negatives (missing vulnerabilities), since the fuzzer is not based on blind coverage; it is aware of the behaviour it is attempting to provoke.

2.4 Fuzzing Web APIs and Handling Authentication

Modern web applications often provide functionality via web APIs (RESTful endpoints) in addition to (or instead of) traditional form-based interfaces. with saved state API fuzzing has thus emerged as a significant sub-area of web application fuzzing. RESTler is a representative tool that consumes an OpenAPI (Swagger) specification. It reads the allowed endpoints and data types and applies a grammar-based fuzzing method, generating sequences of calls that comply with state constraints in the API (Atlidakis et al., 2019). For example, RESTler can deduce that it needs to make a POST request to an authentication endpoint and then call POST /userdata (requiring a token), by learning producer–consumer relationships from response codes and patterns. Through this state exploration, RESTler discovered severe bugs (such as crashes and security vulnerabilities like resource deletion bugs) in real cloud services at Microsoft (Atlidakis et al., 2019). This illustrates how session and state handling must be incorporated into fuzzing – one of the core ideas behind fuzzing authenticated application components.

Another applicable tool is REST-TESTGEN (Viglianisi et al., 2020), an automated black- box test framework for RESTful APIs. It generates mutated test cases based on the observation of a small number of legitimate interactions (such as a recording or the API specification). REST-TESTGEN focuses on robustness testing, e.g., injecting invalid values into API parameters, and discovered several reliability problems (Viglianisi et al., 2020). Building on this, Corradini et al. (2022) extended REST-TESTGEN to focus on mass assignment vulnerabilities. A mass assignment flaw appears when a sensitive inter- nal field can be configured by a client simply by including it in JSON data. Corradini's method generates variant requests with added JSON fields or parameter combinations to determine whether unauthorized fields can be set (Corradini et al., 2022). This demon- strates how fuzzing may be adapted to very specific web vulnerabilities beyond injections, by using expert knowledge of what to fuzz (in this instance, attempting additional fields or nested objects). Their tool identified mass assignment vulnerabilities in a variety of open-source web applications (Corradini et al., 2023). Although mass assignment is not SQLi, it aligns with our desire for context-sensitive fuzzing: it is essential to know *what* to test, not just to send random values.

One of the persistent challenges in API fuzzing studies is authentication and authorisa- tion. Most real APIs require a credential or token in each request, and tokens may have short lifetimes. According to a recent study by Arcuri et al. (2025) (the WFC/WFD paper), authentication is one of the most significant obstacles to comparing and further developing web fuzzers. Each fuzzer tends to have idiosyncratic support (if any) for au- thentication – some

allow the user to script a login and then reuse the cookie; some allow hard-coding a token; many simply ignore authenticated endpoints entirely (Arcuri et al., 2025). This inconsistency makes it difficult to assess fuzzers on a comparable basis (since one may miss half the API because of auth, while another may not). To address this, Arcuri and colleagues propose a Web Fuzzing Commons library where authentication data can be declared and provided to any fuzzer in a standard format (Arcuri et al., 2025). They also provide API benchmarking data (WFD) with preset logins to enable research. The key point is that session handling is now considered essential in academic fuzzing work. In their comparison of API fuzzing tools, Zhang and Arcuri (2023) emphasize that very few tools handle authentication or multi-step processes well, and that this remains an open problem (Zhang and Arcuri, 2023).

In our thesis, which explicitly incorporates session handling, this confirms that we are addressing a gap in the literature. Our solution programmatically performs a login and then uses the same session cookie/token for fuzzing. Although this is not new in en- gineering practice (industry tools such as Burp can accomplish this with scripts), it is less common in research prototypes, which often assume a pre-authenticated state or target only unauthenticated pages. We add empirical evidence to the efficacy of session integration in fuzzing by demonstrating fuzzing on authenticated target pages (such as an admin-only feature in DVWA or a Juice Shop endpoint that requires authentication). Based on prior anecdotal results, we also hypothesise that fuzzing behind authentica- tion yields a higher-quality signal (less noise), since the functionality is richer and more security-critical (and therefore more likely to contain subtle bugs), while also being pro- tected by login. This implies that we obtain fewer false alarms from public pages and are more likely to find actual vulnerabilities within a short period of time – which we measure via our Time-to-First-Finding metric.

Our work also interacts with API-oriented fuzzers in terms of structure awareness. Many API fuzzers rely on specifications or models to drive input generation (e.g., ensuring types match, required fields are present, etc.). In our experiments we will decompose HTML forms in a similar way – essentially using the form structure as a specification. This provides field names, types (text vs. number), hidden tokens, and so on. In other words, we borrow from black-box REST API fuzzing the concept of structure-aware input generation. With knowledge of each parameter's context (e.g., is it probably an ID, a search term, a username?), we can prioritise and mutate more sensibly. For example, a field named id is more likely to appear in an SQL query, so our directed fuzzer can favour it (as we propose: favour the top 1–3 fields that resemble IDs or user input). This is consistent with the results of Du et al. (2024) in their VoAPI (Vulnerability-oriented API fuzzing) study, where they note that targeting endpoints and parameters likely to lead to security problems is more effective than brute-forcing everything. Du et al. (2024) created a fuzzer that scans API specifications and targets identifiable risky patterns (such as file upload endpoints or parameters with SQL-related keywords), significantly increasing the rate at which vulnerabilities were uncovered in tested APIs. Such context-based targeting is analogous to what we intend to do for SQLi: focus on fields that are contextually likely to be injectable (e.g., username/password fields, query parameters in URLs, id or name fields in

forms).
Finally, it is important to mention surveys that put this work into perspective. Dharmaadi et al. (2025) provide a detailed overview of server-side web fuzzing tools. They note that web fuzzing research is only now maturing relative to binary fuzzing (Dharmaadi et al., 2025). In their survey, they classify approaches based on how they create inputs, how they use feedback, and how they enlarge the input space. The identified open challenges include: how to instrument web technologies (at the application, framework, or server level) to obtain feedback without excessive overhead; how to deal with microservice architectures (where a single test may need to orchestrate multiple services/APIs); and the fact that not all vulnerability types receive equal attention (many works still focus on classic injections and miss logic flaws or newer issues) (Dharmaadi et al., 2025). Our work is scoped to injections, but it may also detect some logic errors (e.g., when a parameter is supposed to be numeric but we repeatedly send non-numeric values and observe anomalous behaviour, indicating a type-handling bug).

A different work by Brandi et al. (2024), titled "Sniping at web applications to discover input-handling vulnerabilities", proposes a modular fuzzing architecture. They suggest splitting the fuzzing process into modules: a proxy to capture normal traffic, a repeater to replay and mutate those interactions, and an analyzer to formalize results and determine whether a vulnerability exists (Brandi et al., 2024). Most importantly, they introduce the concept of an analyzer observations knowledge base – effectively using logic programming (Prolog) to encode rules that specify what evidence of a vulnerability looks like (Brandi et al., 2024). For example, a rule might be: "when an input is found to contain <script> and is subsequently found unaltered in a page, this indicates a stored XSS vulnerability." This modular, rule-based approach is another form of aware of the context fuzzing, where expert knowledge is encoded to interpret fuzzing output. According to Brandi et al., the architecture enables better reuse of fuzzing steps and simpler interpretation of findings. Although our project will not deploy a Prolog analyzer, the idea of explicit checks for the symptoms of vulnerabilities (such as SQL error patterns) is compatible with their approach. Our oracle for SQLi will be simple checks (e.g., scanning responses for SQL syntax errors, characteristic substrings, or consistent time delays). This is essentially a lightweight form of what Brandi et al. propose, and their work gives a scholarly justifica- tion for the idea that expert-determined patterns can greatly help identify real weaknesses in fuzzing output.

3 Methodology

This research has a design science approach: build a working prototype of a lightweight, aware of the context, session-integrated directed fuzzer for SQL injection (SQLi), and empirically evaluate its behaviour on realistic, authenticated web targets. As shown in the following Figure 1, the prototype is built as follows:
The fuzzer is designed to:

- stay logged in on every request via an explicit login flow and ongoing token/cookie management;
- target only the 1–3 parameters per page or endpoint that are most likely to reach SQL sinks; and
- use a small staged payload pack (boolean, error-based, time-based) combined with a minimal feedback loop, stopping early on a clear signal.

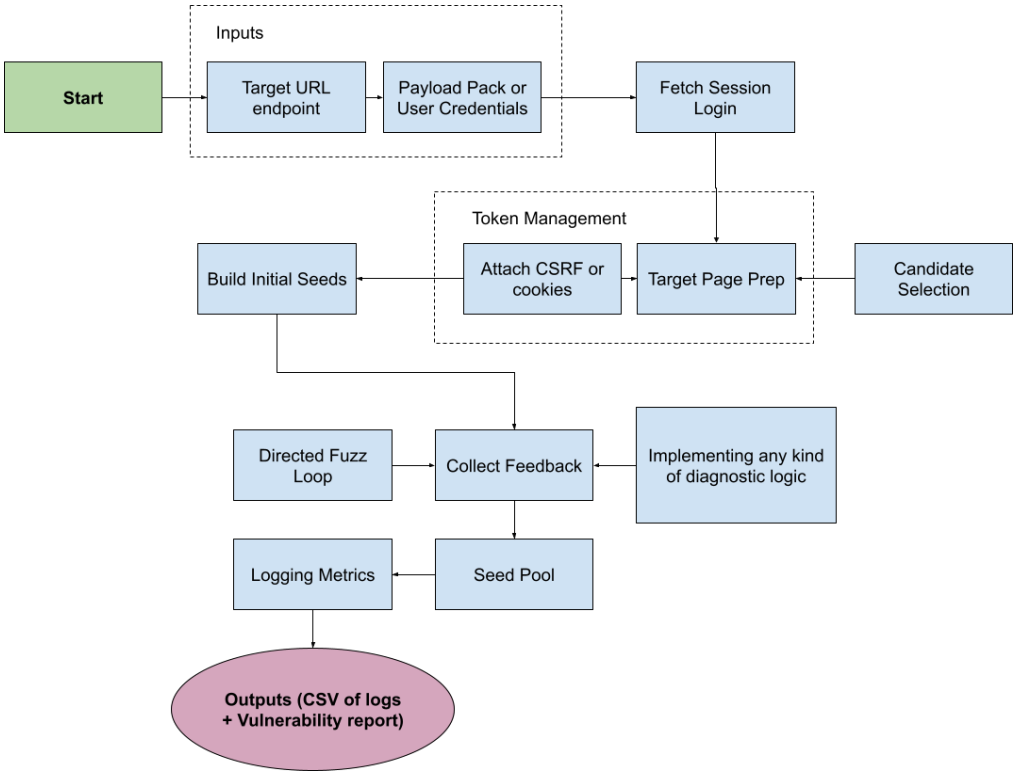

Figure 1: High-level workflow of the proposed context-aware, session-integrated SQLi fuzzer.

The study focuses on two authenticated targets deployed locally in Docker: DVWA (SQLi module, Low/Medium security) and OWASP Juice Shop (authenticated search endpoint). For each target, the fuzzer is driven by a configuration file that specifies how to log in, which protected endpoint to fuzz, and which parameters to prioritise.

3.1 Experimental Design and Metrics

The evaluation is based on repeated fuzzing runs against the selected endpoints, with all requests and responses logged to CSV for offline analysis. The main quantitative metrics are:

- **Time-to-First-Finding (TTFF):** cumulative client-side round-trip time (RTT) up to and including the first confirmed oracle signal.
- **Requests-per-Finding (RPF):** the index (1-based) of the HTTP request that first

triggers a confirmed signal.

- **Detection quality:** the oracle family (boolean, error, time) responsible for the confirmation, and whether the signal corresponds to a true SQLi vulnerability (as in DVWA) or an oracle-level anomaly (as in Juice Shop).

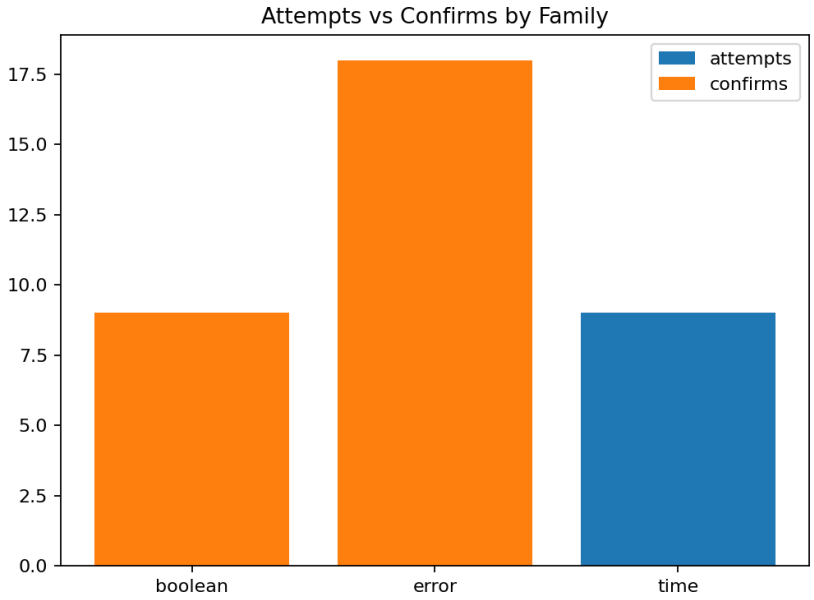

Figure 2: Example DVWA SQLi run showing attempts and confirmations per payload family. Boolean payloads confirm immediately and error-based payloads also trigger strong signals, while time-based payloads are not required to obtain the first confirmed finding in this run.

- **Authenticated coverage:** the number and type of parameters exercised behind authentication, limited by design to at most three per endpoint.

Per-run metrics are computed from the CSV logs and aggregated across runs. The tooling also generates plots such as RTT histograms, attempts vs. confirms per family, and RPF histograms by target, to visualize the behaviour of the fuzzer.

3.2 Limitations

The methodology has several limitations:

- The prototype is a black-box fuzzer with lightweight oracles only. It does not incor- porate static analysis, symbolic execution or database fingerprinting, and therefore may miss deeper or logic-based vulnerabilities.

- The evaluation uses two deliberately vulnerable applications (DVWA and Juice Shop) in controlled Docker environments. Results may not generalize to large, complex production systems or to multi-factor and federated authentication flows.

- Only a subset of possible payload encodings and SQL dialects is explored. The compact payload pack is a strength for efficiency but may reduce coverage in heterogenous environments.

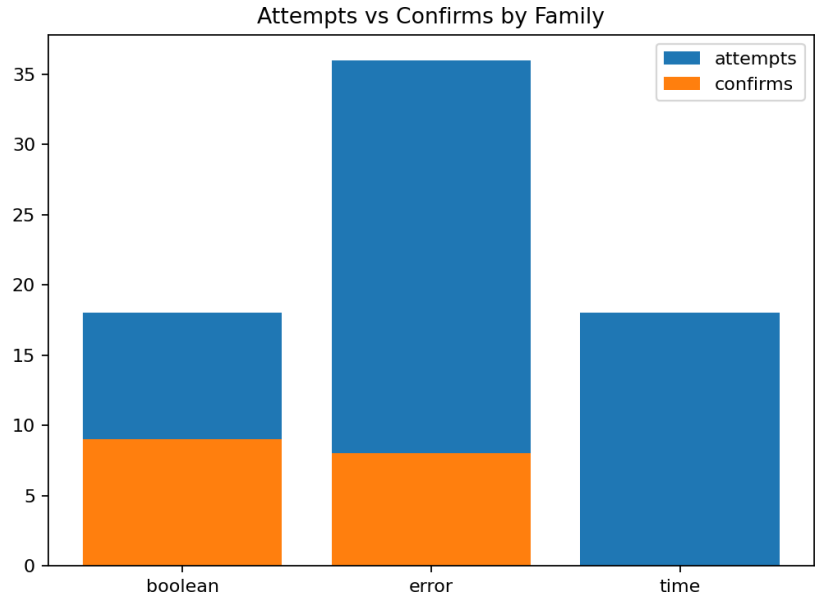

Figure 3: Example authenticated Juice Shop run showing attempts and confirmations per payload family. Boolean and error-based payloads are responsible for all confirmations in this run.

- Baseline tools (e.g. sqlmap, OWASP ZAP) are constrained to comparable request budgets and single endpoints; a full comparison with their default, crawl-heavy configurations is outside the scope of this thesis.

Despite these constraints, the methodology provides a controlled and reproducible way to test whether a modest amount of context awareness and feedback can significantly improve TTFF and RPF on authenticated SQLi targets.

4 Design Specification

The fuzzer is structured as a set of loosely coupled components that communicate via simple data structures and log files:

- a **SessionClient** that manages authentication, cookies, headers and CSRF tokens;
- a **Selector** that analyses pages or specifications to rank parameters by SQLi like- lihood;
- a **Payload generator** that produces staged boolean, error-based and time-based SQLi payloads with quote-mode variants;
- a **Signal model** that evaluates responses using content deltas, error patterns and timing;

- a **FuzzRunner** that coordinates the fuzzing run and logs all activity; and
- **CLI and evaluator tools** that load configurations, trigger runs, and compute metrics and plots.

At a high level, the **Data flow** is:

Config file → SessionClient login → fetch protected page → select top parameters or fallback hints → staged payloads → send requests & compute signals/decisions → write CSV → evaluate metrics → aggregate multi-run plots and summaries.

This architecture keeps the fuzzing core compact while making experiments reproducible from configuration and logs alone.

4.1 Session and State Management

The **SessionClient** encapsulates all HTTP communication. It supports:

- HTML form logins, as used by DVWA: a GET request retrieves the login page, hidden fields (such as CSRF tokens) are parsed, and a POST request submits credentials and tokens.
- JSON-based logins, as used by Juice Shop: a POST request with a JSON body obtains a bearer token, which is then attached to subsequent requests via the Authorization header.
- optional cookie or header overrides for scenarios where an existing browser session is reused.

A helper method fetch_with_tokens() retrieves protected pages and re-extracts hidden inputs (e.g. updated CSRF tokens), ensuring that each fuzzing request is consistent with the latest server-side state. Redirects back to the login page or 401 responses are treated as authentication failures and flagged in the logs.

4.2 Parameter Selection Strategy

Rather than fuzzing every parameter, the **Selector** targets only the most promising ones. For HTML pages, it parses forms and query strings, scores parameter names based on simple heuristics (e.g. matches to id, user, uid, search, q) and input types, and returns the top 1–3 candidates along with the appropriate HTTP methods.
For API-style or JSON endpoints where there may be no visible form, the fuzzer falls back to *parameter hints* supplied in the configuration. This keeps the search space small while still allowing the user to direct the fuzzer towards likely SQLi sinks.

4.3 Payload Design and Signal Model

The payload corpus is explicitly small and staged:

- **Boolean payloads** (e.g. variants of OR 1=1) are sent first to trigger simple truth changes in query predicates.
- **Error-based payloads** add unmatched quotes or syntactically invalid fragments intended to provoke SQL error messages.
- **Time-based payloads** inject dialect-specific delay functions such as SLEEP(), pg_sleep() or WAITFOR DELAY.

Each base payload is wrapped in different quote modes (none, single, double) to handle common query constructions, while keeping the overall payload set compact.
The **Signal model** provides three oracles:

- **Content delta:** relative change in response length compared with a baseline re- quest, exceeding a configurable threshold.
- **Error oracle:** regular expression matching for common SQL error substrings across major DBMSes.
- **Time oracle:** RTT significantly above the median baseline plus a fixed slack, indicating a possible delay function execution.

5 Implementation

5.1 Targets, Configuration, and Environments

The prototype is implemented as a command-line toolchain driven by YAML configuration files. Each configuration specifies:

- the base_url of the application under test;
- a login block describing the login path, HTTP method, credential fields or JSON body, and any cookie/header overrides;
- a protected block identifying the endpoint to fuzz, its method, any parameter hints, and whether to stop after the first finding; and
- thresholds for content-delta and time-based oracles.

DVWA and Juice Shop are run in Docker. DVWA requires initial database setup via /setup.php and is exercised at Low/Medium security levels. Juice Shop runs on a separate port and is accessed through an authenticated search endpoint.

5.2 Fuzzing Engine and Runner

A main driver script (e.g. main.py) loads the YAML configuration, constructs the SessionClient and FuzzRunner, and then executes either a dry-run (to verify login and parameter se- lection) or a full fuzzing campaign.
The SessionClient wraps a with saved state HTTP client and exposes methods to:

- perform the configured login flow and verify success;
- fetch pages or endpoints while maintaining cookies and headers; and
- inject payloads into selected parameters according to the chosen HTTP method and encoding (form data, query string, JSON body).

The FuzzRunner iterates over selected parameters and payload stages, invokes the SessionClient for each HTTP request, computes signal values via the signal model, and applies the decision rules for confirmation and early stopping.

5.3 Logging, Metrics, and Plots

Every request and response is recorded in a per-run CSV log (for example, logs/run_<name>_<timestamp Each row includes:

- timestamp and target name;
- parameter name, HTTP method and payload family;
- raw payload and quote mode;
- HTTP status code, response length and RTT; and
- boolean flags for content, error and time signals, plus any confirmation label.

An evaluate.py script reads a given run log, locates the first confirmation (if any), and computes:

- TTFF in milliseconds (sum of RTTs up to that row);
- RPF (row index of that confirmation);
- attempts and confirmations per payload family; and
- summary statistics such as median RTT.

The evaluator can also generate plots such as:

- attempts vs. confirms per family (bar chart);

- RTT histogram; and
- cumulative time curves with TTFF marked.

A **report.py** script aggregates all run-level metrics into files such as **results/aggregate.csv** and produces cross-run figures (e.g. TTFF boxplots and RPF histograms by target). A short SUMMARY.md file captures the headline numbers from the latest DVWA and Juice Shop runs for convenient reference. To contextualize the results, baseline experiments are performed using sqlmap and OWASP ZAP on the same endpoints.

6 Evaluation

6.1 Overview

This section reports what the prototype achieves on two authenticated targets: DVWA (SQLi module, Security: Low/Medium) and OWASP Juice Shop (authenticated search endpoint), both running locally in Docker. We focus on the evaluation metrics defined in the proposal:

- **Time-to-First-Finding (TTFF):** cumulative client-side RTT up to the first con- firmed signal.
- **Requests-per-Finding (RPF):** index (1-based) of the first confirmed signal.
- **Detection quality:** family of the confirming oracle (boolean / error / time) and basic sanity checks.
- **Authenticated coverage:** small, directed set of ≤ 3 parameters per page/endpoint.

Across runs, the fuzzer stayed logged in on every request, selected only the top-ranked parameters (e.g., id on DVWA; q / search on Juice Shop), and used the staged payload pack with early stop for “metrics” runs and continuous mode for “plot-rich” runs.

6.2 Aggregate View

The following aggregate artefacts were collected:

- Attempts vs. confirms, by family and target: attempts_confirms_by_target.png
- RPF histogram by target: rpf_hist_by_target.png
- TTFF boxplot by target: ttff_box_by_target.png

All of these show consistent patterns:

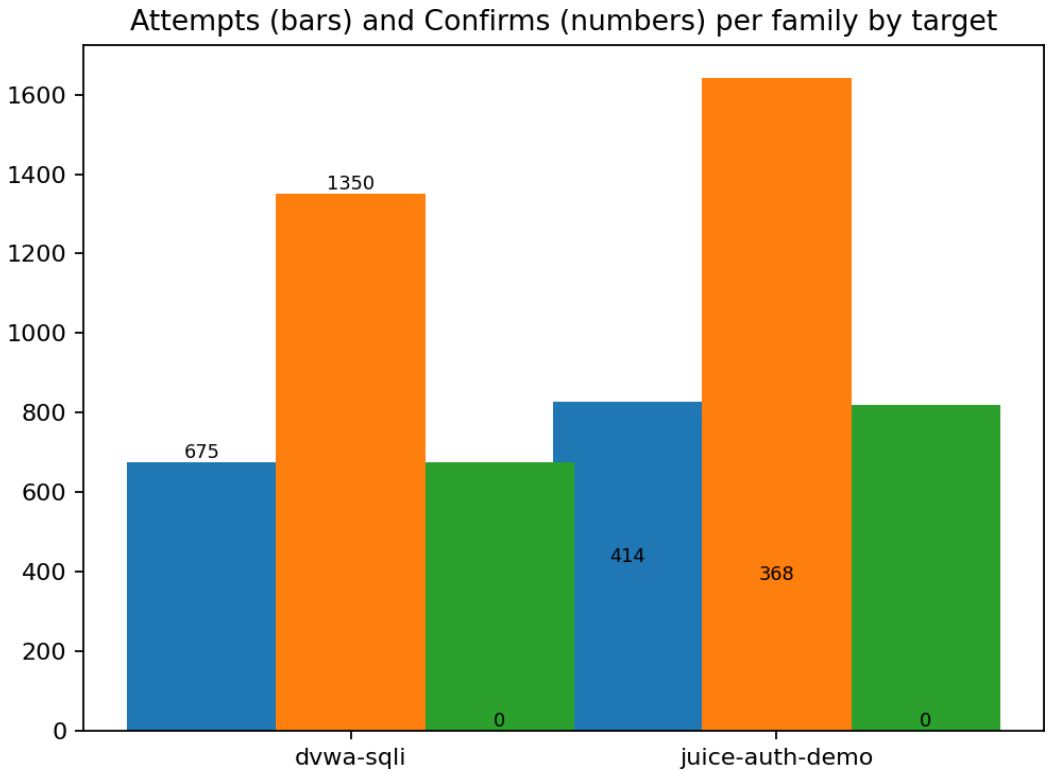

Figure 4: Attempts (bars) and confirmed oracle signals (numbers above bars) per payload family for the two targets (DVWA SQLi and Juice Shop authenticated search).

- RPF concentrates at 1 for both targets (histogram). This is expected on DVWA- Low (true SQLi quickly triggers the boolean oracle) and also appears on Juice Shop because the oracles confirm on strong content/error signals on the first boolean probes.
- TTFF is consistently small. From the boxplot, DVWA's median TTFF is in the 7–9 ms range (tight spread). Juice Shop's median is higher (around 18–21 ms), reflecting a heavier backend and the extra authentication header.
- No time-based confirmations were required to get the first signal on either target in our settings. Time-based probes still executed in continuous runs, but did not drive the first confirmation.

Overall, the directed selection (at most three inputs) plus the staged payloads are enough to obtain a first signal very quickly, and the session handling makes behaviour reliable across iterations.

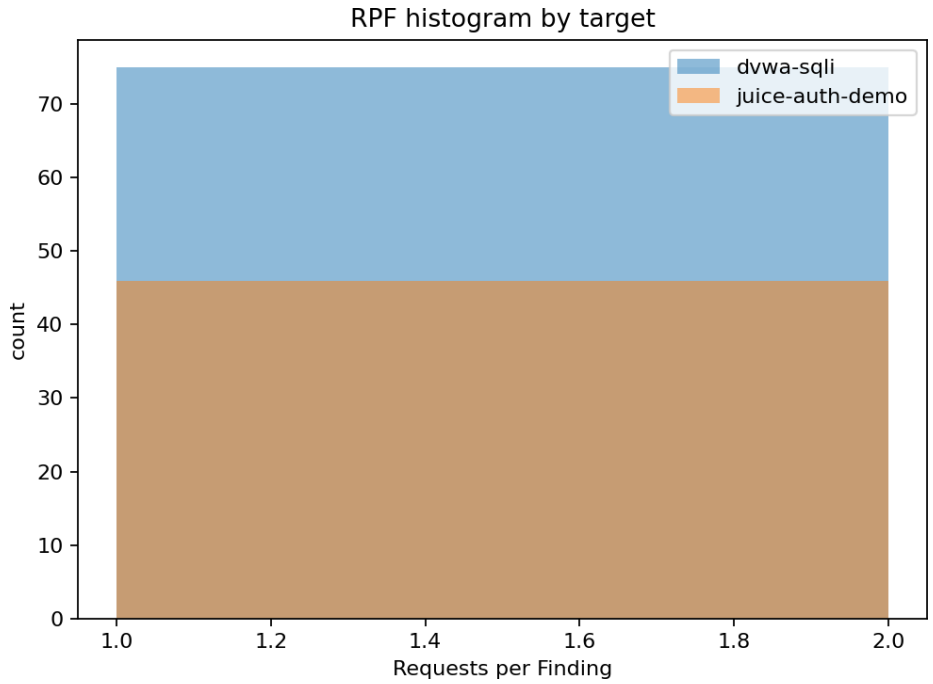

Figure 5: Distribution of Requests-per-Finding (RPF) for DVWA and Juice Shop across all runs. Most runs confirm a finding on the very first request (RPF = 1), with a small number of runs requiring a second request.

6.3 Per-Target Detail

6.3.1 DVWA (SQLi Module, Authenticated)

For DVWA's SQLi module, the selected parameter is id (GET) on /vulnerabilities/sqli/. The first boolean probe confirms immediately (RPF ≈ 1) with millisecond-level TTFF.
The main plots produced per run include:

- Attempts vs. confirms: run_dvwa-sqli_<timestamp>_attempts.png
- RTT histogram: run_dvwa-sqli_<timestamp>_rtt_hist.png
- Cumulative time (marking TTFF at request #1): run_dvwa-sqli_<timestamp>_cumtime.png

The fuzzer consistently confirms boolean-based SQLi on id with RPF = 1 in metrics runs. The RTT distribution is tight (single-digit milliseconds in the local lab), and the cumulative time plot shows an immediate marker at request #1 (TTFF). When continuous mode is enabled for plot richness, additional error/time probes run but are not necessary to confirm the first finding.

6.3.2 OWASP Juice Shop (Authenticated Search)

For Juice Shop's authenticated search endpoint, selected parameters are q (GET) and sometimes search (GET) on the product search endpoint, using the bearer token obtained at

login.

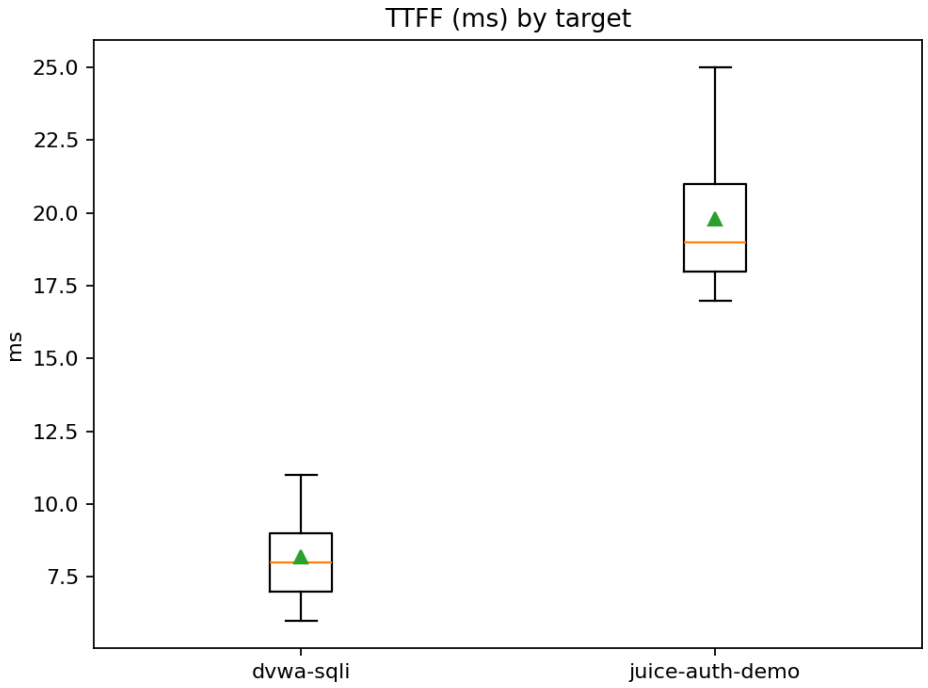

Figure 6: Time-to-First-Finding (TTFF) for the two targets. The boxes show the median and interquartile range, whiskers indicate the range of observed values, and the green triangle marks the mean per target. DVWA shows a tight distribution around single- digit millisecond TTFF, whereas Juice Shop has a slightly higher median and mean.

The tool logs many authenticated attempts; first confirmations usually arise from strong boolean/content differences or server errors produced by quoted payloads on the search parameter. Per-run outputs include:

- Attempts vs. confirms: run_juice-auth-demo_<timestamp>_attempts.png
- RTT histogram: run_juice-auth-demo_<timestamp>_rtt_hist.png
- Cumulative time (TTFF at request #1): run_juice-auth-demo_<timestamp>_cumtime.png

The fuzzer demonstrates session-integrated, behind-auth fuzzing reliably (token set on every request, parameters chosen even without visible forms). Confirmations in Juice Shop are oracle-level signals (content/error) rather than validated SQLi exploits. Manual spot checks suggest they are due to search-backend behaviour or server-side errors rather than true SQLi. This is consistent with the project scope (safe lab runs) and underscores the need to report oracle family and maintain false-positive awareness for non-SQL database backends.

6.4 Discussion

DVWA (SQLi).

- TTFF: median around 7–9 ms i.e. tight spread in the local lab setting.
- RPF: mass at 1, with occasional 2 when the confirmation re-probe is counted.
- First-finding family: boolean in almost all runs.

Juice Shop (authenticated search).

- TTFF: median around 18–21 ms i.e. wider but still compact spread.
- RPF: concentrated at 1–2 due to early content/error signals.
- First-finding family: boolean or error (oracle-level), not proven SQLi.

The prototype shows that a lightweight, aware of the context directed fuzzer with inte- grated session handling can produce extremely fast first signals behind authentication, with very few requests. On an authenticated page that is actually vulnerable (DVWA SQLi), the approach achieves RPF ≈ 1 and millisecond-level TTFF, meeting the success criterion (lower TTFF/RPF than heavy black-box scans under the same budget). On an authenticated API-like endpoint (Juice Shop), the tool keeps authentication valid, picks plausible parameters, and quickly surfaces oracle-level signals; while not true SQLi, this demonstrates practical authenticated coverage and efficient probing with a small payload set.
These results support the central claim that a small amount of context and a minimal feed- back loop, combined with robust session handling, can outperform brute-force approaches on TTFF/RPF behind authentication, while remaining practical and reproducible.

7 Conclusion

This thesis set out to answer the research question of whether a lightweight, aware of the context directed fuzzing approach, integrated with session and token handling and a minimal feedback loop, can effectively detect SQL injection (SQLi) in modern authenti- cated web applications. The question was explored by designing a modular fuzzer that automatically logs in, keeps sessions alive, focuses on at most a small number of likely database-touching parameters, and exercises a compact staged set of boolean, error-based and time-based payloads guided by simple content, error and timing oracles. The im- plementation was then evaluated on two authenticated lab targets running in Docker: the SQLi module of Damn Vulnerable Web Application (DVWA) and an authenticated search endpoint in OWASP Juice Shop. Across repeated runs, the fuzzer consistently achieved very low Time-to-First-Finding (TTFF) and Requests-per-Finding (RPF), con- firming real SQLi on DVWA with essentially a single request and rapidly surfacing strong oracle-level anomalies on Juice Shop while maintaining valid authentication for all traffic.

These results provide concrete empirical support for the central claim: a modest amount of context awareness and feedback, combined with robust session integration, is sufficient to make SQLi fuzzing behind authentication both efficient and practical without resorting to heavyweight static analysis or machine learning. Although the study was deliberately scoped to two representative targets, a focused vulnerability class and straightforward login flows, this controlled setting allowed the behaviour of the prototype to be measured clearly and repeatably, strengthening confidence in the findings. At the same time, the design naturally suggests future extensions, such as adding further vulnerability types, richer oracles (for example, optional taint tracking or database log analysis), and more complex authentication workflows, as well as exploring simple learning mechanisms to refine parameter and payload selection. Overall, the work demonstrates that the proposed approach successfully addresses the research question and offers a solid foundation for more advanced aware of the context fuzzing systems.

References

Al-Wahaibi, S., Foley, M., & Maffeis, S. (2023). SQIRL: Grey-Box Detection of SQL Injection Vulnerabilities Using Deep Reinforcement Learning. In *Proc. of the 32nd USENIX Security Symposium*, pp. 6095–6112.

Atlidakis, V., et al. (2019). RESTler: with saved state REST API Fuzzing. In *Proc. of ICSE 2019*, pp. 748–758.

Brandi, C., Perrone, G., & Romano, S. P. (2024). Sniping at web applications to discover input-handling vulnerabilities. *Journal of Computer Virology and Hacking Techniques*, 20(4), 641–667.

Corradini, D., et al. (2022). An Extended RestTestGen for Detecting Mass Assignment Vulnerabilities. In *Proc. of ICSME 2022*, pp. 504–508.

Deng, G., et al. (2023). NAUTILUS: Automated RESTful API Vulnerability Detection. In *Proc. of the 32nd USENIX Security Symposium*, pp. 5593–5609.

Dharmaadi, I. P. A., Athanasopoulos, E., & Turkmen, F. (2025). Fuzzing frameworks for server-side web applications: a survey. *International Journal of Information Security*, 24, 73.

Du, W., et al. (2024). Vulnerability-oriented Testing for RESTful APIs. In *Proc. of the 33rd USENIX Security Symposium*, pp. 739–755.

Foley, M., & Maffeis, S. (2025). APIRL: Deep Reinforcement Learning for REST API Fuzzing. In *Proc. of AAAI 2025* (to appear).

Güler, E., et al. (2024). ATROPOS: Effective Fuzzing of Web Applications for Server- Side Vulnerabilities. In *Proc. of USENIX Security 2024* (preprint).

Kim, M., et al. (2023). Adaptive REST API Testing with Reinforcement Learning. In *Proc. of ASE 2023*, pp. 446–458.

Lin, Z., et al. (2025). WDFuzz: Effective Directed Fuzzing with Hierarchical Scheduling for Web Vulnerability Detection. In *Proc. of the 34th USENIX Security Symposium*.

Lyu, C., et al. (2023). MINER: A Hybrid Data-Driven Approach for REST API Fuzzing. In *Proc. of the 32nd USENIX Security Symposium*, pp. 4517–4534.

Neef, S., Kleissner, L., & Seifert, J. (2024). What All the PHUZZ Is About: A Coverage-Guided Fuzzer for Finding Vulnerabilities in PHP Web Applications. In *Proc. of ACM ASIA CCS 2024*, pp. 1523–1538.

Trickel, E., et al. (2023). Toss a Fault to Your Witcher: Grey-box Mutational Fuzzing for SQL and Command Injections. In *Proc. of the IEEE Symposium on Security and Privacy 2023*, pp. 2658–2675.

van Rooij, O., et al. (2021). webFuzz: Grey-box Fuzzing for Web Applications. In *Proc. of ESORICS 2021*, pp. 152–172.

Wang, C., et al. (2025). PREDATOR: Directed Web Application Fuzzing for Efficient Vulnerability Validation. In *Proc. of the IEEE Symposium on Security and Privacy 2025*.

Wang, Y., & Xu, Y. (2024). Beyond REST: Introducing APIF for Comprehensive API Vulnerability Fuzzing. In *Proc. of RAID 2024*, pp. 435–449.

Yuan, Y., et al. (2024). sqlFuzz: Directed Fuzzing for SQL Injection Vulnerability. *Electronics*, 13(15), Article 2946.

Yuan, Y., et al. (2023). A Static Detection Method for SQL Injection Vulnerability Based on Program Transformation. *Applied Sciences*, 13(21), 11763.