Thesis Dissertation

LANGUAGE-AGNOSTIC FEEDBACK-DRIVEN WEB FUZZING

Julios Fotiou

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2025

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

Language-Agnostic Feedback-Driven Web Fuzzing

Julios Fotiou

Supervisor
Dr. Elias Athanasopoulos

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

May 2025

Acknowledgments

I would like to express my appreciation to Dr. Elias Athanasopoulos for his supervision throughout the course of this dissertation. His guidance and constructive feedback provided valuable support during the development of this work.

I also wish to thank the faculty members of the University of Cyprus, whose instruction has significantly contributed to my academic development and deepened my interest in the field of computer science.

Lastly, I am deeply grateful to my family and friends for their continued encouragement, understanding, and unwavering support throughout this endeavor.

Abstract

Modern web applications are increasingly built using diverse technologies, often spanning multiple programming languages and frameworks. This heterogeneity poses significant challenges for traditional fuzzing techniques that rely on instrumentation-based code coverage to scale effectively. Such methods can be intrusive, language-specific, and difficult to maintain across complex systems.

We present a language-agnostic web fuzzing framework that eliminates the need for instrumentation by leveraging system-level behavioral feedback. Our prototype was developed by modifying a pre-existing web fuzzer framework (webFuzz), by replacing its instrumentation-based feedback with system-level behavioral signals, specifically system calls and database query patterns, as a means of guiding the exploration process in a language-agnostic manner.

We evaluate our approach by integrating it into the original webFuzz framework and conducting a comparative analysis against the instrumentation-based version. This evaluation, performed on real-world web applications, focuses on code coverage and performance characteristics. Based on the results, our prototype demonstrates the feasibility of using system-level behavioral feedback as a practical and generalizable alternative to traditional instrumentation techniques in language-diverse web environments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In an era where *software vulnerabilities* continue to pose serious threats to digital infrastructure, *fuzzing* has emerged as one of the most powerful and widely adopted techniques for *automated vulnerability discovery*. Its power lies in its simplicity, by automatically generating and executing large numbers of inputs, fuzzers can trigger unexpected behaviors and crash-inducing conditions that developers may never have anticipated. In particular, *coverage-guided* fuzzing has demonstrated remarkable success in detecting memory corruption and logic bugs in native applications by leveraging lightweight feedback mechanisms, such as instrumentation-based code coverage [25]. Tools such as AFL [30] and libFuzzer [21] have significantly advanced the field by applying coverage feedback to guide fuzzing inputs more intelligently. This strategy has proven to be effective in exposing hidden bugs in complex and widely deployed software systems, including high-profile targets such as the Linux kernel, OpenSSL, and Mozilla Firefox [8, 29].

Although fuzzing has seen remarkable success in native software, its application to *web systems* remains relatively underexplored and limited [11, 23]. This is particularly concerning given the increasing complexity and critical role of *web applications* in modern computing. These applications are responsible for managing sensitive data, enforcing authentication, and controlling access to critical backend infrastructure. Vulnerabilities in such systems can have severe consequences, including data breaches, privilege escalation, and service disruption. As a result, enhancing fuzzing techniques for web applications has the potential to significantly improve the security posture of millions of real-world deployments. Several *white-box* web fuzzing techniques have been proposed, typically relying on *symbolic execution*, *taint analysis*, or *static code analysis* to reason about execution paths and generate targeted inputs [1, 4]. Although these approaches offer high precision, they are often difficult to scale because of the dynamic and framework-heavy nature of real-world web stacks. Many web applications use interpreted languages (e.g. PHP, JavaScript) and incorporate third-party libraries, making it hard for symbolic analysis to model runtime behavior accurately.

Consequently, the majority of practical web fuzzers adopt a *black-box* approach, sending HTTP requests without any visibility into the application's internal execution or control flow, and relying solely on externally observable responses, such as status codes or response content [3]. Although this approach is simple to deploy and effective in some scenarios, it provides little to no guidance for test case generation. As a result, *black-box* fuzzers often fail to reach deeper logic or explore complex state-dependent behavior, limiting their ability to detect *vulnerabilities*.

To address these limitations, *grey-box* web fuzzers have emerged, which incorporate lightweight runtime feedback, typically in the form of code coverage, to guide fuzzing more effectively. These tools instrument the application's source code or runtime environment, enabling them to prioritize inputs that explore new execution paths. A notable example is *WebFuzz*, which applies Abstract Syntax Tree (AST)-level instrumentation to PHP applications, to collect code coverage and improve fuzzing precision [23].

While instrumentation has played a crucial role in enhancing test guidance for fuzzing, it introduces significant practical limitations. Most instrumentation techniques are closely tied to the *implementation language*, which restricts the applicability of many fuzzing frameworks to a narrow range of platforms and technologies. In applications built using multiple languages or heterogeneous technology stacks, applying instrumentation consistently across all components may require substantial engineering effort.

These limitations underscore the need for a *language-agnostic* feedback-driven fuzzing approach that preserves the advantages of grey-box techniques, such as guided input generation, without relying on language-specific instrumentation. Decoupling the feedback mechanism from the internal structure of the application enables broader applicability across diverse web environments, particularly those composed of multiple programming languages or frameworks.

To address this, this thesis proposes a novel web fuzzing framework, built upon and extending the original WebFuzz [23] system. The modified framework leverages externally observable runtime behaviors, specifically *system calls* and *SQL queries*, as proxies for internal execution feedback. By passively collecting and analyzing these side effects during execution, the system can prioritize inputs that trigger new behaviors, enabling deeper exploration of application logic in a non-intrusive and scalable manner. This design maintains the effectiveness of *coverage-guided* fuzzing while improving its practicality and generalizability in real-world web environments, where traditional instrumentation poses significant practical challenges.

1.2 Contributions

The main contributions of this thesis are:

1. We propose a grey-box fuzzing approach for web applications that replaces traditional instrumentation-based feedback with system-level behavioral signals. By extracting *system calls* and *SQL queries* during execution, the framework eliminates the need for access to source code or language-specific internals, enabling input mutation guidance in a language-agnostic and non-intrusive manner.

2. We introduce a novel prioritization strategy based on *execution traces*, where system calls and SQL query keywords are abstracted into lightweight sequences. These traces are matched to individual HTTP requests using timestamp alignment and compared using edit distance, allowing the fuzzer to prioritize inputs that exhibit behavioral novelty.

3. We implement the proposed feedback mechanism as an extension to the WebFuzz framework [23], replacing its original AST-level PHP instrumentation with passive system-level monitoring, while preserving the overall architecture of the framework. The modified fuzzer operates sequentially to ensure accurate trace-to-request associations and enable precise feedback-driven input selection.

4. We evaluate the modified framework on three widely-used web applications, WordPress, DVWA, and Joomla, comparing its performance against both the original WebFuzz and a variation that incorporates periodic mutation from the queue. While our approach does not use instrumentation for feedback during fuzzing, we retain WebFuzz's instrumentation during evaluation solely to measure code coverage, enabling a fair and consistent comparison across all configurations.

At its core, our research aims to determine whether system calls and SQL queries can provide meaningful feedback for guiding the prioritization and mutation of fuzzing requests. Crucially, our work focuses on evaluating the feasibility of leveraging such traces as an alternative to traditional instrumentation-based feedback mechanisms.

Chapter 2

Background

In this chapter, we present the necessary *background* to support the design and development of our proposed *language-agnostic fuzzing* framework. We begin with an overview of fuzzing and its core classifications, *black-box, white-box, and gray-box*, based on the level of internal visibility available to the fuzzer. We then introduce *coverage-guided fuzzing*, a widely adopted *grey-box technique* that leverages runtime feedback to guide input generation. Next, we discuss the unique challenges of applying fuzzing to *web applications* and examine the limitations of *language-specific* instrumentation in real-world scenarios. We also provide an overview of the WebFuzz [23] framework, which serves as the foundation for our prototype, and highlight the aspects that are preserved or modified in our implementation. Finally, we introduce two key concepts that are central to our approach: *edit distance*, which we use to measure the similarity between execution traces, and *priority queues*, which help determine the scheduling of inputs based on behavioral novelty.

2.1 Fuzzing

Fuzzing is a software testing technique that involves automatically generating and executing large numbers of inputs with the goal of triggering unexpected behaviors such as *crashes, assertion failures, or memory errors* [13, 14, 16]. It is especially valuable in the context of *security testing*, where such behaviors can reveal *vulnerabilities* exploitable by attackers [22]. The simplicity of fuzzing allows it to be integrated into a wide range of testing pipelines, and its effectiveness has been demonstrated in uncovering both shallow and deep bugs across many types of applications.

At its core, a fuzzer operates in an iterative loop designed to uncover unexpected or erroneous behavior in software. This loop typically consists of four key stages: *input generation, execution, monitoring, and feedback* [31]. First, the fuzzer generates inputs, either randomly or based on a model or mutation of existing inputs, and feeds them to the target program. The program is then executed with these inputs, while the fuzzer monitors for abnormal behavior such as *crashes, hangs, assertion failures, or memory access violations*. In more advanced fuzzing techniques, this monitoring phase also collects *code coverage* or runtime information to guide the input generation process [30]. The results are then analyzed, and the feedback is used to refine or evolve subsequent inputs in a way that increases the likelihood of *triggering deeper* or more subtle *bugs*. The sophistication of this loop varies based on the type of fuzzing employed.

Fuzzing techniques are typically categorized into *black-box*, *white-box*, and *grey-box* approaches, depending on the level of internal information available about the program under test:

Black-box

Black-box fuzzers treat the application as opaque and generates inputs without any knowledge of the internal code or execution state, relying solely on outputs such as response codes or error messages [3, 13]. This model is easy to deploy and works well for closed-source or externally hosted systems, but often suffers from low code coverage and ineffective exploration of deep application logic.

White-box

White-box fuzzers have full access to source code and execution semantics, allowing them to reason about program paths using techniques like symbolic execution, concolic execution, or constraint solving [12, 32, 33]. While white-box fuzzing enables precise input generation and high path coverage, it is inherently computationally intensive, presents scalability challenges in large or complex programs, and requires significant engineering effort to apply in production environments [6].

Grey-box

Grey-box fuzzers represent a practical compromise between *black-box* and *white-box* approaches, leveraging lightweight runtime instrumentation to collect limited execution feedback, most commonly in the form of *code coverage* [30]. Their scalability, simplicity, and demonstrated success in real-world applications have made them a widely adopted choice in modern fuzz testing [5].

In practical settings, the choice of fuzzing technique is shaped by various factors, including the characteristics of the target application, available resources, and the constraints of the deployment environment. *Black-box* fuzzing is often favored for *web applications* and *closed-source* software, where internal visibility or access to source code is limited. *Grey-box* fuzzing, offering a balance between observability and performance, is widely used across different domains where partial *instrumentation* is feasible. *White-box* fuzzing, although capable of deep and comprehensive analysis, is typically reserved for *research* or *controlled environments* due to its high computational cost and implementation complexity. As a result, it is less suited for *large-scale* or *time-sensitive* testing scenarios.

2.2 Instrumentation

Instrumentation is a technique used to monitor the runtime behavior of a program by *injecting additional code or hooks* into its execution. In the context of fuzzing, instrumentation is commonly used to collect feedback, such as code coverage, or branch execution, which helps guide input generation toward unexplored or interesting program paths [2, 30].
There are generally two categories of instrumentation:

Static Instrumentation

Static Instrumentation involves modifying a program's binary before execution. This method allows analysts to insert additional code or markers into the binary, facilitating the observation of specific behaviors or the collection of metrics during runtime. It is particularly useful for scenarios where source code is unavailable, and modifications need to be made directly to the binary [2].

Dynamic Instrumentation

Dynamic Instrumentation refers to the analysis of a program during its execution. This technique enables real-time observation and modification of a program's behavior without altering the original binary. Tools like Pin and DynamoRIO are commonly employed for this purpose, allowing analysts to inject code, monitor execution paths, and gather runtime information dynamically [2].

While *dynamic instrumentation* offers flexibility and does not require source code access, it is less commonly used in practical fuzzing frameworks due to its higher runtime overhead and complexity. *Coverage-guided* fuzzers like *AFL* and *libFuzzer* rely heavily on static instrumentation to track edge or block coverage during execution [21, 29, 30]. This feedback is essential for driving the mutation engine, allowing the fuzzer to evolve inputs that exercise new parts of the codebase. Similarly, in the domain of web fuzzing, tools like *WebFuzz* perform instrumentation at the Abstract Syntax Tree (AST) level for interpreted languages like PHP, enabling fine-grained control over which paths have been explored [23].

Despite its advantages, instrumentation is not always practical or desirable in real-world deployments. It can introduce significant performance overhead, compatibility issues, or runtime instability, especially when applied to production systems or complex execution environments. In some cases, instrumentation relies on language-specific features or interpreter internals [23], which limits portability and confines fuzzing to a narrow set of applications. As a result, there is growing interest in developing a language-agnostic feedback mechanisms that enable effective fuzzing without requiring modifications to the target application.

2.3 Coverage-Guided Fuzzing

Coverage-guided fuzzing (CGF) is one of the most widely adopted *grey-box fuzzing* strategies. It uses lightweight instrumentation to gather feedback on which parts of the program are exercised by each test input. This runtime feedback is then used to guide the generation of new inputs, helping the fuzzer explore previously unexecuted code paths more effectively [25, 30]. A typical coverage-guided fuzzing loop includes generating inputs, executing them, collecting coverage information, and mutating promising inputs to reach deeper code regions. Inputs that exercise new program behaviors, such as novel control-flow paths, are prioritized for further mutation [24]. This process, illustrated in Algorithm 1, enables the fuzzer to incrementally expand the program state space it can reach, improving its ability to discover both shallow and deep bugs.

Algorithm 1: Coverage-Guided Fuzzing Loop, extracted from [24]

```
   Input: Program $P$, Seed corpus $C$
   Output: Crash-inducing inputs $D$
1  $D \leftarrow \emptyset$ ;                                    ▷ Crash reports
2  while fuzzing not terminated do
3      $i \leftarrow$ PickSeed($C$) ;                           ▷ Select input from corpus
4      $i' \leftarrow$ Mutate($i$) ;                            ▷ Apply random mutation
5      $(outcome, coverage) \leftarrow$ Run($P, i'$) ;          ▷ Execute and monitor behavior
6      if $outcome = crash$ then
7          $D \leftarrow D \cup \{i'\}$ ;                       ▷ Log crash-inducing input
8      if coverage is new then
9          $C \leftarrow C \cup \{i'\}$ ;                       ▷ Expand corpus
10 return $D$
```

Several types of coverage metrics are used in CGF:

- **Block coverage**: identifies whether specific basic blocks have been executed [30].
- **Edge coverage**: tracks transitions between basic blocks in the control-flow graph [30].
- **Path coverage**: records full execution paths [27].
- **Comparison feedback**: captures values in branch conditions, helping generate inputs that satisfy conditionals [14].

Widely used tools such as *AFL* and *libFuzzer* have shown the effectiveness of coverage-guided fuzzing by discovering thousands of bugs in real-world software [21, 29, 30].

2.4 Web Application Fuzzing

Web application fuzzing is a specialized form of software testing that aims to discover vulnerabilities in *server-side* web logic by automatically generating and sending crafted HTTP requests. It extends the core principles of fuzzing, which involve generating a large number of inputs and observing system behavior, to the web domain, where the inputs consist of structured network traffic and the target application is a remote server handling client requests [11, 23].

A typical web fuzzer operates by constructing HTTP requests that target parameters such as *query strings, form fields, cookies, and headers*. These inputs are then sent to the application, and the responses, such as HTTP status codes, response bodies, and error messages, are monitored for signs of unexpected behavior. Examples of such behavior include *server crashes, stack traces, anomalous output, or violations of expected logic*, such as *authentication bypass or database error leakage*. In particular, web fuzzers often attempt to detect vulnerabilities like *Reflected Cross-Site Scripting* (XSS) by injecting JavaScript or HTML payloads into input fields and observing whether they are improperly reflected in the server's response. The primary goal is to uncover vulnerabilities such as reflected cross-site scripting (XSS), SQL injection, file inclusion, broken access control, and other logic flaws that may compromise application security [15, 23].

Unlike native binaries, web applications are structured as multi-layered systems composed of routing logic, middleware, templating engines, and backends such as databases or APIs. Additionally, they often require stateful interactions: for example, reaching certain endpoints may require login credentials, cookies, or a specific sequence of requests. This makes fuzzing more complex, as it requires not only generating valid individual requests, but often maintaining realistic user workflows and session state to reach deeper functionality [23].

Despite these challenges, web application fuzzing remains a crucial part of modern security testing. Given the ubiquity of web interfaces and their direct exposure to untrusted users, any exploitable flaw in a web application can have significant consequences, from data leaks to full server compromise. As a result, improving the effectiveness and depth of fuzzing techniques in this domain continues to be a high-impact area of research.

2.5 WebFuzz Framework Overview

WebFuzz [23] is a grey-box fuzzing framework designed to uncover vulnerabilities in web applications by combining lightweight instrumentation with dynamic analysis. Specifically targeted at *PHP-based* environments, WebFuzz instruments the application's Abstract Syntax Tree (AST) to gather code coverage information during execution. This feedback is used to guide the mutation and selection of HTTP requests, allowing the framework to explore deeper execution paths and uncover vulnerabilities such as cross-site scripting (XSS). The framework includes components for automated crawling, request mutation, feedback-based scheduling, and vulnerability detection. Inputs are prioritized primarily based on their contribution to code coverage, and mutations are generated accordingly to maximize the likelihood of triggering previously untested behaviors.

In this thesis, the WebFuzz framework serves as the foundation for our prototype. While we retain its overall structure, we significantly modify its core feedback mechanism by removing the reliance on instrumentation. Instead of using AST-based coverage signals, our approach adopts a language-agnostic feedback model that leverages externally observable *system-level behaviors*.

2.6 Edit Distance and Trace Similarity

In the absence of traditional code coverage metrics, our approach relies on behavioral traces, such as system calls and SQL queries, to infer application behavior. To effectively use these traces for feedback, we require a way to measure how similar or different two execution traces are. For this purpose, we employ edit distance, a well-established metric used to quantify the difference between two sequences.

Edit Distance

Edit distance, also known as *Levenshtein distance*, is a metric that quantifies the minimum number of single-element operations required to transform one sequence into another, where these operations include *insertions, deletions, or substitutions*. In our context, each execution trace is represented as a sequence of abstracted system-level events, such as syscall names or SQL keywords. By calculating the edit distance between a newly observed trace and those previously encountered, we are able to estimate the behavioral novelty of new inputs.

Trace Similarity as a Coverage Proxy

This comparison serves as a *lightweight* and *language-independent* proxy for code coverage. By measuring how different a newly observed trace is from those seen previously, the fuzzer can estimate whether a new input triggers novel application behavior. Traces that differ substantially from prior executions are considered more interesting and are, in principle, prioritized for further exploration, while those that are very similar may be deprioritized to avoid redundant effort.

Adopting this trace-based similarity approach enables feedback-driven fuzzing without requiring instrumentation or access to the source code, thereby supporting a language-agnostic fuzzing strategy.

2.7 Priority Queue

A priority queue is a specialized data structure that organizes and retrieves elements based on their assigned priority, rather than the order in which they were added. Each element in the queue is associated with a priority value, and the element with the highest priority is always selected for processing next. This allows the system to focus on the most "interesting" or important items at any given time [10].

In the context of fuzzing, the priority queue acts as a scheduler for test cases. Instead of processing test cases in the order they are generated, the fuzzer uses the priority queue to always select the test case deemed most promising, according to some criterion, for the next mutation and execution cycle. This allows the fuzzer to focus computational resources on inputs that are most likely to yield new or interesting behaviors [23].

In our approach, each test case is assigned a priority based on two main factors: the behavioral novelty of its execution trace, and the frequency with which it has already been selected. The priority queue automatically maintains these relationships, ensuring that at each iteration, the next test case to be fuzzed is one that strikes the best balance between novelty and diversity. By using a priority queue, the fuzzer can efficiently manage thousands of candidate inputs, rapidly adapt to new discoveries, and avoid wasting resources on redundant or uninteresting paths.

Chapter 3

Architecture

3.1 Overview

The architecture of our prototype is designed to efficiently guide the fuzzer and establish a language-agnostic *feedback mechanism*, without relying on language-specific instrumentation. At its core, the system is feedback-driven: it uses *externally observable behaviors*, specifically, *system calls* and *SQL queries*, as proxies for internal program coverage. This approach enables the framework to operate across a broad spectrum of web technologies, such as applications composed of multiple programming languages.

This architecture treats the web application as an *opaque system*, focusing entirely on the external effects that each test input produces during execution. When an input, such as an HTTP request, is sent to the application, the framework passively *monitors the system's runtime activity*. It captures the sequences of system calls that represent all interactions with the operating system, such as file operations, network communication, and process management, as well as the SQL queries issued to backend databases. These observations provide a detailed picture of the application's real execution paths, without requiring modification of the application itself.

Each execution is abstracted into a behavioral trace: a simplified sequence of system-level events and database operations that characterizes how the application responded to a specific input. By comparing execution traces, the framework identifies whether a test case exercises *previously unexplored* code paths or follows paths that have already been covered.

In summary, the architecture is specifically designed to determine whether a web fuzzer can be effectively guided without relying on instrumentation, thereby enabling its use across a diverse range of web applications. By focusing on externally observable behaviors, such as *system calls and SQL queries*, our prototype aims to provide a flexible and language-agnostic feedback mechanism while maintaining minimal intrusion.

3.2 System Design

The architecture of our language-agnostic web fuzzing framework is organized into several main components, each fulfilling a distinct role within the fuzzing process. Together, these components form an iterative feedback-driven workflow that enables the efficient exploration and testing of web application behavior. The overall structure and interaction of the components that make up our fuzzing framework are illustrated in Figure 3.1. This high-level overview provides a visual summary of the system's iterative feedback-driven process, from input generation to behavioral monitoring and prioritization. In the following subsections, we describe each component in detail.

3.2.1 Input Generation and Mutation

The fuzzing process begins with a collection of valid HTTP requests, each representing a typical interaction with the target web application. As the automated crawler explores the application, it continuously discovers new endpoints and user actions, dynamically expanding the pool of candidate requests. In addition to these newly identified requests, the framework generates further test inputs by systematically mutating requests that have previously shown *promising* or *interesting results*. Mutation strategies include introducing crafted payloads, recombining elements from different requests, and altering the structure or content of request parameters. In some cases, certain parameters may be deliberately excluded from mutation to maintain necessary application state. This dual approach, which combines ongoing crawling with diverse mutation techniques applied to favorable requests, ensures that the fuzzer explores both new and previously encountered parts of the application, maximizing the likelihood of uncovering a wide range of potential vulnerabilities.

3.2.2 Automated Execution

Each input, whether discovered by the crawler or generated through mutation of favorable previous requests, is automatically submitted to the target web application in the form of an HTTP request. This fully automated process enables efficient and repeatable submission of a large number of test cases, systematically exercising different *entry points, endpoints, and workflows* within the application. By automatically exercising both newly discovered and promising mutated requests, the framework achieves broad and thorough coverage of the application's possible behaviors. This high degree of automation enhances the efficiency, repeatability, and scalability of the testing process, allowing security assessments to proceed with minimal manual effort, even in large or complex web application environments.

3.2.3 Behavioral Monitoring

As each input is processed by the web application, the framework passively monitors the application's runtime activity by externally capturing all relevant *system calls and SQL queries* generated during the handling of that request. This behavioral monitoring is non-intrusive and requires no modification of the application's source code, thereby preserving normal operations. The resulting *traces*, which consist of sequences of system calls and SQL queries, are collected for every test case. By comparing these traces, the framework can identify and retain those inputs that, based on their behavioral differences, are likely to have exercised previously unexplored execution paths. These prioritized inputs form the basis for subsequent mutations, allowing the fuzzer to progressively expand its exploration of the behavior and logic of the application.

3.2.4 Trace Association and Abstraction

To accurately analyze the application's behavior in response to each test input, our framework first associates system call and SQL query traces with their corresponding HTTP request. This is achieved by collecting all system-level and database events that occur within a defined *time window* spanning the request and its response. By focusing on this window of activity, the framework ensures that only the behaviors directly triggered by a specific input are considered for further analysis.

Next, the *raw* system calls and SQL queries captured during execution are abstracted into compact behavioral traces. Each trace succinctly summarizes the sequence of system-level and database operations performed by the application in response to a particular input, providing a high-level view of its internal processing. This abstraction process involves *filtering out unnecessary details* and focusing on the essential structural elements, such as *syscall names and key SQL query keywords*, thereby reducing the complexity and size of the traces. By transforming detailed execution logs into concise, comparable representations, the framework facilitates efficient analysis and enables the use of *sequence similarity metrics* to guide the fuzzing process. This approach not only streamlines trace comparison but also supports the *language-agnostic* nature of the framework, as it operates independently of the application's implementation language or internal architecture.

Finally, the abstracted syscall and SQL traces are combined into a unified *sequence of tokens* for each request. This unified sequence captures the full range of externally observable behaviors associated with the input and serves as the foundation for trace comparison, novelty assessment, and input prioritization during fuzzing.

3.2.5 Trace Comparison

After abstraction, the trace associated with each new input is compared against traces generated by *previous executions*. A similarity metric, such as *edit distance*, is used to measure the degree of difference between the new trace and the set of previously observed traces. If the trace associated with a new input is found to be *unique or significantly different*, it suggests that the input has driven the application along a new execution path, potentially exposing new behaviors, logic, or vulnerabilities.

This process enables the framework to quantitatively assess the novelty of the effect of each input on the application, guiding the prioritization of test cases for future mutations. By systematically seeking out and favoring inputs that yield novel traces, the framework continually expands its exploration of the state space of the application. This feedback-driven approach increases the likelihood of uncovering complex or deep-seated vulnerabilities that might be missed by approaches relying solely on random input generation.

3.2.6 Prioritization and Scheduling

All candidate inputs, along with their associated behavioral traces, are managed within a *priority queue*. This structure allows the fuzzer to continually assess and reprioritize test cases, ensuring that the *most promising* inputs are systematically selected for further *mutation and testing* as the fuzzing process progresses. The scheduling mechanism assigns higher priority to inputs that have demonstrated *behavioral novelty*, as determined by trace comparison, or to those that have been *selected less frequently* for mutation. This dual consideration helps ensure a *balanced exploration*, preventing the framework from focusing exclusively on a small subset of inputs while neglecting others that might still lead to interesting discoveries.

By continually updating priorities based on both novelty and selection frequency, the system adapts in real time to the evolving state of application exploration. Inputs that consistently produce unique or rarely observed traces are favored for subsequent mutation and testing, thus maximizing the likelihood of uncovering subtle or complex vulnerabilities. This adaptive scheduling is central to maintaining an effective and efficient fuzzing process, enabling the framework to systematically cover a wide spectrum of the application's logic while minimizing redundant testing.

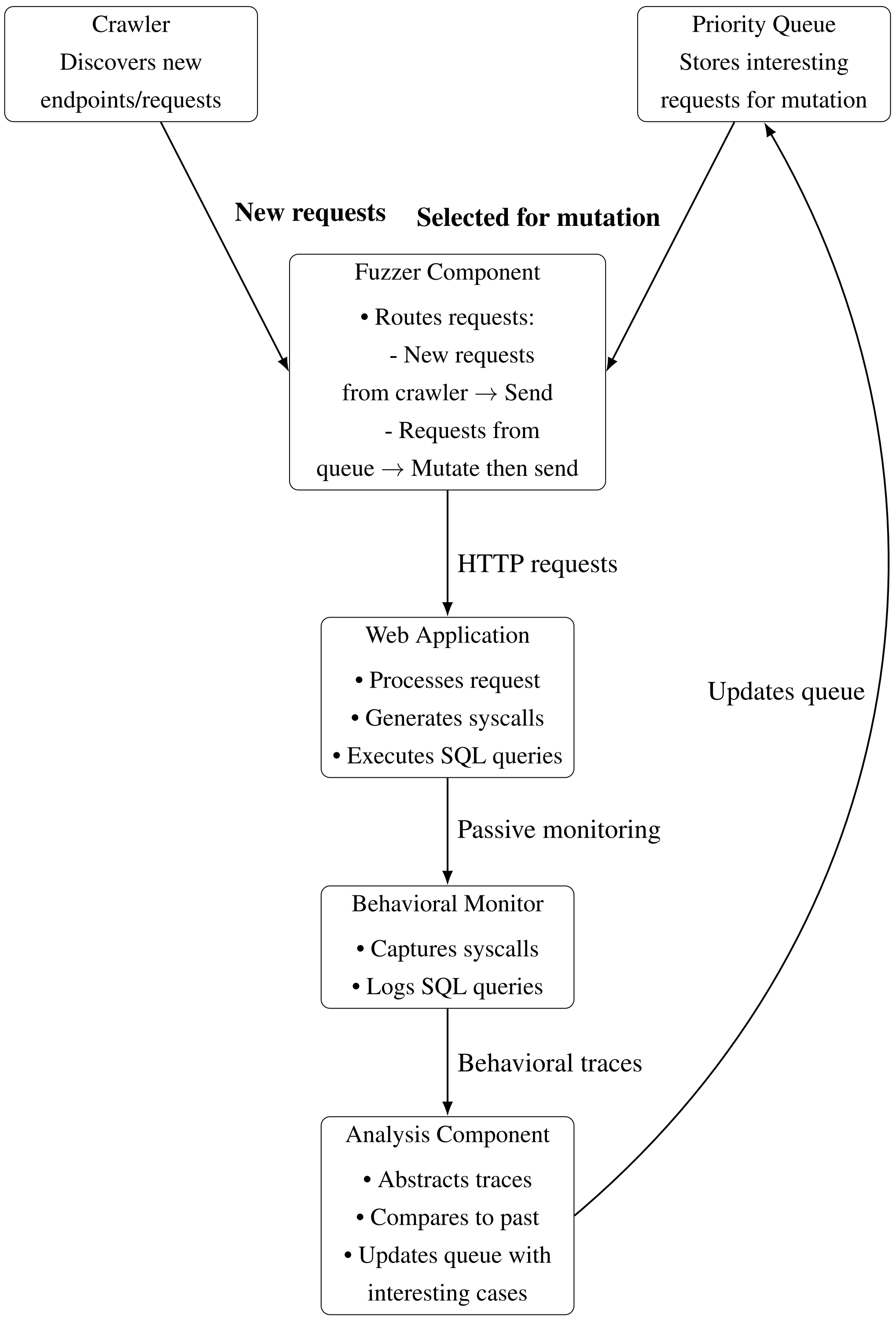

Figure 3.1: Architecture and feedback-driven workflow of the language-agnostic web fuzzing framework.

Chapter 4

Implementation

4.1 Overview

This chapter presents the implementation details of our *language-agnostic feedback-driven web fuzzing framework*. The primary goal of the implementation was to evaluate whether observable behavior at the system level, specifically *system calls* and *SQL queries*, can serve as meaningful feedback signals to guide the fuzzing process. This is done instead of traditional *code coverage*, which typically requires instrumentation at the source code level.

To achieve this, we adapted the existing WebFuzz framework [23] by removing its dependency on instrumentation-based coverage collection. In its place, we introduced a new mechanism that *collects, abstracts, and analyzes* externally observable behavioral traces generated by the application during its execution. This mechanism operates independently of the underlying implementation language, making it suitable for use with a wide range of real-world web applications.

The resulting prototype enables us to perform a direct comparison against the original WebFuzz system, which relies on instrumentation-based feedback to guide the fuzzing process. This side-by-side evaluation enables a systematic assessment of whether trace-based behavioral signals, derived from system calls and SQL queries, can effectively substitute for traditional code coverage metrics in practice.

4.2 Tracing and Behavioral Monitoring

A core component of our implementation is the ability to observe the runtime behavior of the web application in response to each fuzzing input. Rather than relying on internal instrumentation, we employ an external tracing mechanism that captures two categories of system-level activity: *system calls* made by the application and *SQL queries* executed by its back-end database. This design ensures full language independence for feedback guiding,enabling the fuzzer to prioritize requests for future mutation without requiring source code instrumentation.

Tracing is active throughout the entire fuzzing session. From the moment the fuzzer is launched until it terminates, system activity is continuously recorded in the background. To correlate this global stream of behavior with specific inputs, we define a bounded time window around each HTTP request. This window spans from just before a request is issued to shortly after the corresponding response is received. All system-level activity observed during this interval is attributed to the handling of that particular input.

System calls are captured using *strace*, a dynamic tracing tool that intercepts and logs all interactions between a running process and the operating system kernel. By attaching *strace* to the relevant worker processes that serve the web application, we record low-level operations. This provides a detailed view of how the application interacts with the underlying system in response to each test input.

In parallel, SQL query monitoring is achieved by enabling query logging at the database level. All queries issued by the application during the same bounded-time window are extracted from the logs and associated with the corresponding input. This dual-tracing approach yields a rich behavioral profile for each request, combining operating system activity with database interactions. By isolating relevant behavior within this time window, we obtain a pair of input-specific raw trace that accurately reflect how the application responded to a given test case.

4.3 Trace Abstraction

Once the system call and SQL query traces are captured for each input, they are processed to produce an *abstract, uniform* representation that facilitates behavioral comparison. The purpose of this abstraction is to reduce trace complexity while preserving the structural and semantic information relevant to the program behavior.

The abstraction process begins by extracting the relevant syscall and SQL query segments from the raw trace logs based on the time window associated with each request. These segments are parsed and transformed individually:

- **System calls:** From each syscall entry, only the *syscall name* is retained, discarding arguments, return values and metadata. This results in a simplified sequence that reflects the types of system-level operations triggered by the request.
- **SQL queries:** Each SQL statement is reduced to a sequence of capitalized keywords. These keywords are then concatenated into a single token per query, effec-

tively representing the structural pattern of the SQL operation. Table names, field names, literals, and values are omitted to ensure generality and reduce variability from input-dependent data.

After individually abstracting both traces,they are merged into a single unified sequence of tokens that characterizes the overall behavior of the input. To preserve meaningful structure, we maintain the original ordering within each trace: system calls are listed in the order they occurred during execution, and SQL queries follow in the order they were logged by the database. However, rather than interleaving system calls and SQL queries in their original chronological order, we concatenate the two sequences by appending the entire list of abstracted system calls first, followed by the corresponding SQL query tokens. This design choice was made empirically: during testing, we observed that separating the two trace types in this way led to more stable and consistent similarity measurements. Specifically, when sending the same HTTP request multiple times, the resulting traces produced lower edit distances under this ordering scheme, indicating stronger alignment and reduced noise in the comparison process.

Listings 4.1 through 4.3 illustrate the transformation process applied to raw execution events. The workflow begins with the full chronological trace Listing 4.1, followed by the separation of system calls and SQL queries Listing 4.2, and concludes with the abstraction and final composition of the unified trace used for novelty detection and prioritization Listing 4.3.

```
- read(11, "GET␣/wp-admin/load-styles.php?c="..., 8000) = 1101
- fstat(12, {st_mode=S_IFREG|0644, st_size=55401, ...}) = 0
- SELECT option_name, option_value FROM wp_options WHERE autoload = 'yes'
- munmap(0x79235a600000, 2097152) = 0
- SELECT * FROM wp_users WHERE user_login = 'username' LIMIT 1
- close(11) = 0
```

Listing 4.1: Raw execution trace in chronological order

```
System calls:
  - read(11, "GET␣/wp-admin/load-styles.php?c="..., 8000) = 1101
  - fstat(12, {st_mode=S_IFREG|0644, st_size=55401, ...}) = 0
  - munmap(0x79235a600000, 2097152) = 0
  - close(11) = 0

SQL queries:
  - SELECT option_name, option_value FROM wp_options WHERE autoload = 'yes'
  - SELECT * FROM wp_users WHERE user_login = 'username' LIMIT 1
```

Listing 4.2: Separated system calls and SQL queries

```
System calls:
  - read
  - fstat
  - munmap
  - close
```

```
SQL queries:
  - SELECTFROMWHERE
  - SELECTFROMWHERELIMIT

Final combined trace:
    read fstat munmap close SELECTFROMWHERE SELECTFROMWHERELIMIT
```

Listing 4.3: Abstracted and combined trace

4.4 Trace Comparison and Novelty Estimation

Once the behavioral trace of a given input is abstracted into a unified sequence of tokens, the next step involves determining whether this trace exhibits meaningful differences compared to those already observed. To achieve this, our framework performs a systematic comparison between the new trace and the set of previously stored traces, with the goal of identifying novel behaviors that may indicate unexplored execution paths.

We measure similarity using the *edit distance metric*, which quantifies the minimum number of operations required to transform one sequence into another. These operations include insertions, deletions, and substitutions. In our implementation, the edit distance is computed at the *token level* instead of the character level. This approach aligns with the structure of our abstracted traces, which are composed of discrete tokens such as syscall names and SQL keyword sequences, rather than continuous character strings.

For each new trace, we compute its edit distance against all previously recorded traces in the corpus. Among these, we retain the smallest edit distance, which represents the closest behavioral match to the new input. A *low minimum distance* suggests that the current input has produced a response that closely resembles prior behavior, while a larger distance implies that the input may have triggered new or distinct behavior within the application.

This minimum-distance value is used as the *novelty score* of the input. Inputs with higher novelty scores are assumed to be more valuable, as they are more likely to explore code regions that have not yet been exercised. These inputs are then selected for future mutation and retained in the priority queue.

By leveraging this trace-based comparison strategy, the fuzzer can intelligently and progressively explore the behavioral space of the application without relying on instrumentation. This approach establishes a fully language-agnostic feedback mechanism that is both generalizable and effective across a wide range of web technologies.

4.5 Prioritization and Scheduling

After traces are abstracted and analyzed, the framework must determine which inputs should be *prioritized* for *mutation and re-execution*. This prioritization mechanism is fundamental to the effectiveness of the fuzzing strategy, as it directs computational resources toward inputs that are more likely to trigger unexplored execution paths.

All candidate inputs are maintained in a *dynamic priority queue*, implemented using a *min-heap* data structure. This structure enables efficient insertion, retrieval, and reordering of inputs based on their priority scores. In a min-heap, the element with the *lowest score* is always located at the root of the heap and is *selected first* during extraction operations. Consequently, the input deemed most valuable, based on the prioritization criteria, is always readily accessible at the top of the queue, enabling the framework to make prompt and informed scheduling decisions.

Each input is assigned a numerical priority score derived from a combination of two criteria: behavioral novelty and selection frequency. Behavioral novelty is measured using the edit distance between the abstracted trace of the current input and the previously observed traces. A larger edit distance implies that the input has triggered behavior that deviates significantly from past executions, suggesting the exploration of new code paths. Conversely, selection frequency tracks how many times a given input has already been used for mutation. Inputs that have been selected *less frequently* are slightly favored to promote diversity and avoid stagnation in local areas of the input space.

To integrate these two metrics into a single comparable value, the system employs a weighted scoring function. For each component, a weighted difference is computed using the formula:

$$\text{WeightedDifference}(v_1, v_2) = \frac{w \cdot (v_2 - v_1)}{\left|\frac{v_1 + v_2}{2}\right|}$$

where `v1` and `v2` represent the metric values of two competing nodes, and w is the weight assigned to the corresponding component. The total priority score is computed by aggregating the results of these individual weighted comparisons.

The comparison logic is implemented directly in the node structure used by the priority queue. When two nodes are compared, such as during insertion or heap rebalancing, the system evaluates their respective scores using this function. Inputs that exhibit greater novelty and lower selection frequency produce *lower composite scores* and are therefore *favored* by the min-heap. This ordering ensures that the input with the most desirable combination of properties is automatically positioned at the top of the queue and selected next for mutation and execution.

The weights assigned to novelty and frequency were empirically determined through experimentation. The system places greater emphasis on behavioral novelty, reflecting the framework's objective of maximizing behavioral diversity and exploring new execution paths. Selection frequency is assigned a smaller weight to ensure that promising but underexplored inputs are not neglected.

As the fuzzing process continues, this adaptive prioritization mechanism enables the system to dynamically re-rank inputs based on updated trace comparisons and usage statistics. New inputs are inserted into the heap with scores calculated using the same criteria, while existing inputs may change in priority as new behavioral insights emerge. By prioritizing inputs that exhibit the highest novelty and minimal prior usage, the system effectively enhances both the depth and breadth of application exploration. This approach increases the likelihood of uncovering rare or complex execution paths.

4.6 Cyclic Request Scheduling

A key objective of this work is to evaluate the effectiveness of a language-agnostic feedback mechanism for guiding the fuzzing process. However, the default scheduling behavior in the original WebFuzz framework favors exploration by prioritizing *newly discovered* requests from the crawler. Inputs from the mutation queue, those selected based on behavioral traces, are only used when the crawler exhausts all available paths. Although effective in maximizing application coverage, this policy limits the influence of the prioritization mechanism, making it difficult to assess its true value.

To enable a more meaningful evaluation of the feedback mechanism, we modified the input scheduling policy to introduce a *cyclic request selection strategy*. Under this scheme, inputs are selected from the mutation queue at regular intervals, even if new requests from the crawler are still available. Specifically, after a fixed number of crawler-generated inputs, a request is deliberately selected from the mutation queue to ensure that trace-based prioritization actively contributes to the fuzzing process.

This modification was essential to observe the impact of our feedback mechanism. By ensuring that prioritized inputs are evaluated alongside newly discovered requests, we are able to monitor their performance, contribution to behavioral coverage, and ability to uncover unexplored execution paths. Additionally, this scheduling policy retains the original system's fallback behavior. When the crawler discovers no new inputs, the scheduler automatically shifts to using the mutation queue exclusively. This ensures continuity in the fuzzing process while maintaining the balance between exploration and exploitation.

To visually summarize the cyclic selection strategy and its decision logic, Figure 4.1 presents the request scheduling workflow. It shows how the framework alternates between crawler-generated and queue-based requests, and how fallback to the mutation queue occurs when no new crawler inputs are available.

In summary, cyclic request scheduling was introduced not as an optimization, but as a deliberate design choice to support a fair and rigorous evaluation of our proposed trace-based feedback approach. It allows the system to test the inputs selected by behavioral prioritization more consistently, providing a stronger foundation for comparing their effectiveness relative to purely exploratory strategies.

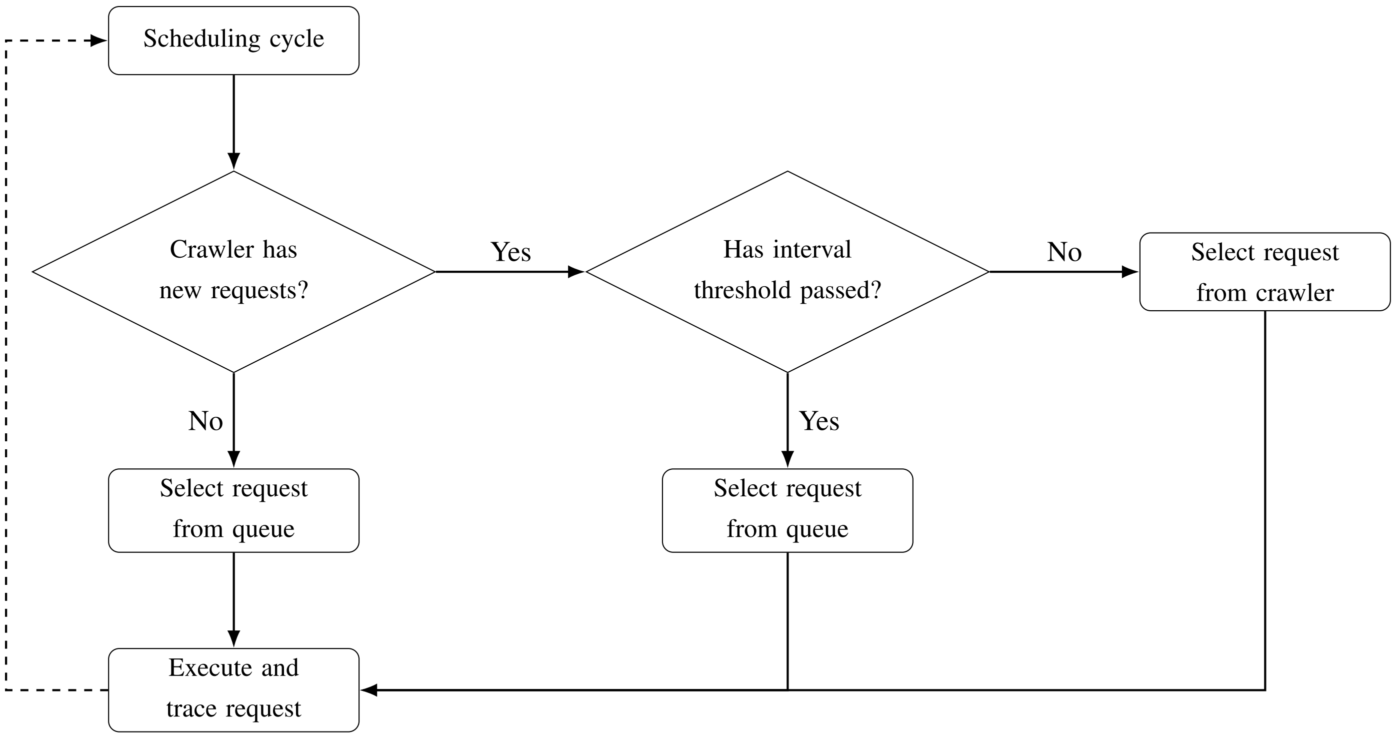
Scheduling cycle
Crawler has new requests?
Yes
Has interval threshold passed?
No
Select request from crawler
No
Yes
Select request from queue
Select request from queue
Execute and trace request

Figure 4.1: Cyclic scheduling logic combining crawler exploration and feedback-guided mutation.

Chapter 5

Evaluation

5.1 Evaluation Setup

All experiments were conducted on a single Ubuntu 24.10 Linux machine equipped with an 11th Gen Intel® Core™ i7-1165G7 processor running at 2.80 GHz, featuring 4 physical cores and 8 threads, and supported by 16 GB of RAM. The web server used to host the applications is Apache 2.4.62 and we used MySQL 8.0.42 for the database back-end.

To evaluate the effectiveness and efficiency of the proposed trace-based feedback mechanism, we selected three widely used web applications:

- WordPress 6.1.1, a popular content management system (CMS).
- DVWA 2.5 (Damn Vulnerable Web Application), a deliberately insecure web app for security testing.
- Joomla 5.0.3, another widely deployed and feature-rich CMS.

Each application was deployed in a controlled environment, configured with default settings where applicable, and isolated from external traffic to ensure consistent measurements.
We compare the following fuzzing configurations:

- The original WebFuzz framework, which uses instrumentation-based feedback for guidance.
- WebFuzz with Cyclic Iteration, which periodically selects inputs from the mutation queue.
- Our proposed Trace-based Feedback variant, which relies on system-level signals to guide input selection in a language-agnostic manner.

Each experiment was executed for a total duration of 600 minutes to provide sufficient time for the fuzzers to explore the application and stabilize their performance. All configurations were initially evaluated under identical time constraints to ensure fairness when comparing real-world execution characteristics such as throughput and long-term behavior. In addition, selected experiments were also evaluated under a fixed number of requests, based on the request counts observed in our trace-based method, to allow for a fair assessment of code coverage across fuzzers independent of execution speed.
We evaluated each fuzzer using the following key metrics:

- **Code Coverage:** The extent to which the application's codebase is exercised, measured using instrumentation. This includes both overall and mutated request coverage.
- **Feedback Mechanism Effectiveness:** The ability of mutated (non-crawler) requests to contribute uniquely to code coverage, demonstrating how well each fuzzer prioritizes meaningful mutations.
- **Request Generation Behavior:** This aspect of evaluation captures how efficiently each fuzzer generates inputs over time. It includes both the throughput (requests per second) and the request activity observed during specific execution intervals. Together, these measurements provide insight into execution efficiency, responsiveness, and the impact of feedback mechanisms or application complexity on request generation trends.

5.2 Fuzzer Performance Analysis

5.2.1 Overall Code Coverage

Code coverage is a key metric for evaluating the effectiveness of feedback mechanisms in guiding exploration during fuzzing. In our experiments, we measured the percentage of unique code paths exercised by each fuzzing strategy across three widely used web applications: WordPress, DVWA, and Joomla.

Table 5.1 presents the mean code coverage (%) achieved by each fuzzer across three real-world web applications over the full duration of the experiment.
The original WebFuzz achieved the highest mean coverage on WordPress (24.5%) and DVWA (31.0%), with moderately lower performance on Joomla (19.5%). Incorporating cyclic iteration led to only marginal improvements in WordPress (25.0%) and Joomla (20.0%), while having no observable impact on DVWA.
In comparison, our trace-based feedback approach yielded slightly lower coverage across all three applications: 19.0% for WordPress, 29.5% for DVWA, and 17.0% for Joomla.

This reduction is expected, given that our method does not rely on instrumentation-based coverage and instead leverages system-level traces to guide input selection. While this approach may miss certain code paths captured by traditional instrumentation, it offers a viable and generalizable alternative that is language-agnostic.

Fuzzer	WordPress (%)	DVWA (%)	Joomla (%)
Original WebFuzz	24.5 ± 0.5	31.0 ± 0.1	19.5 ± 0.5
WebFuzz + Cyclic Iteration	25.0 ± 0.6	31.0 ± 0.2	20.0 ± 1.0
Trace-based Feedback (Ours)	$\mathbf{19.0 \pm 1.0}$	$\mathbf{29.50 \pm 0.2}$	$\mathbf{17.0 \pm 1.0}$

Table 5.1: Mean code coverage (%) ± standard deviation across runs per web application.

5.2.2 Coverage by Request Type

To gain deeper insight into how each fuzzer configuration achieves its overall code coverage, we analyze the contributions of two distinct request types: *crawler-generated* requests, which explore the application through predefined or automatically discovered paths, and *mutated* requests, which are generated and prioritized based on feedback mechanisms.

Figures 5.1, 5.2, and 5.3 illustrate the code coverage achieved by each request type across WordPress, DVWA, and Joomla, respectively. In all cases, crawler-generated requests are responsible for the majority of total coverage. This is expected, as the crawler quickly discovers and explores accessible endpoints, covering common or shallow application logic.

However, although smaller in magnitude, the contribution from mutated requests remains important, as these inputs often reach deeper or less accessible code paths that the crawler alone cannot uncover. These requests reflect the effectiveness of each fuzzer's feedback mechanism. As shown in Table 5.2, when comparing mutated coverage under the same execution time, our trace-based feedback method achieves competitive results. For instance, our approach yields $1.35\% \pm 0.95$ mutated coverage on WordPress and $1.03\% \pm 0.03$ on DVWA, which, while slightly lower, remains comparable to WebFuzz with cyclic iteration, achieving $1.95\% \pm 1.15$ and $1.58\% \pm 0.40$ on the same applications, respectively. This is particularly noteworthy given that our method operates under stricter execution constraints. While mutated coverage on Joomla is notably lower at $0.12\% \pm 0.02$, this may be attributed to the complexity of the application and fewer opportunities for effective mutation.

Although our method does not attain the same level of mutated coverage as instrumentation-based techniques such as WebFuzz with cyclic iteration, it delivers comparable results and demonstrates the practical viability of a language-agnostic, instrumentation-free approach. In cases where coverage is lower, this is likely due to factors such as application complexity and fewer opportunities for effective mutation, suggesting that the effectiveness of the method may vary with the nature of the target application.

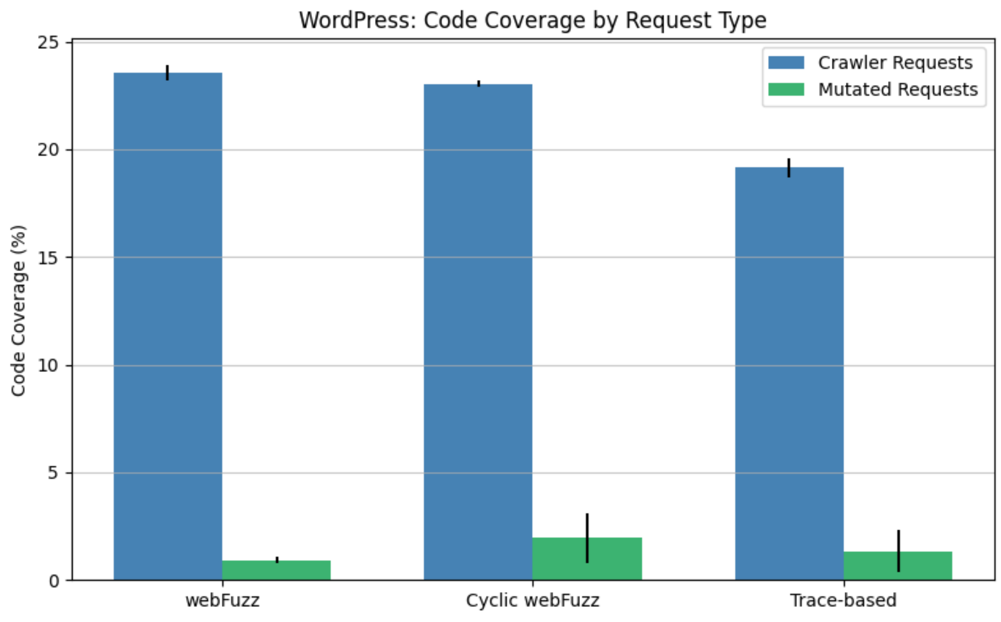

Figure 5.1: WordPress: Code coverage by request type.

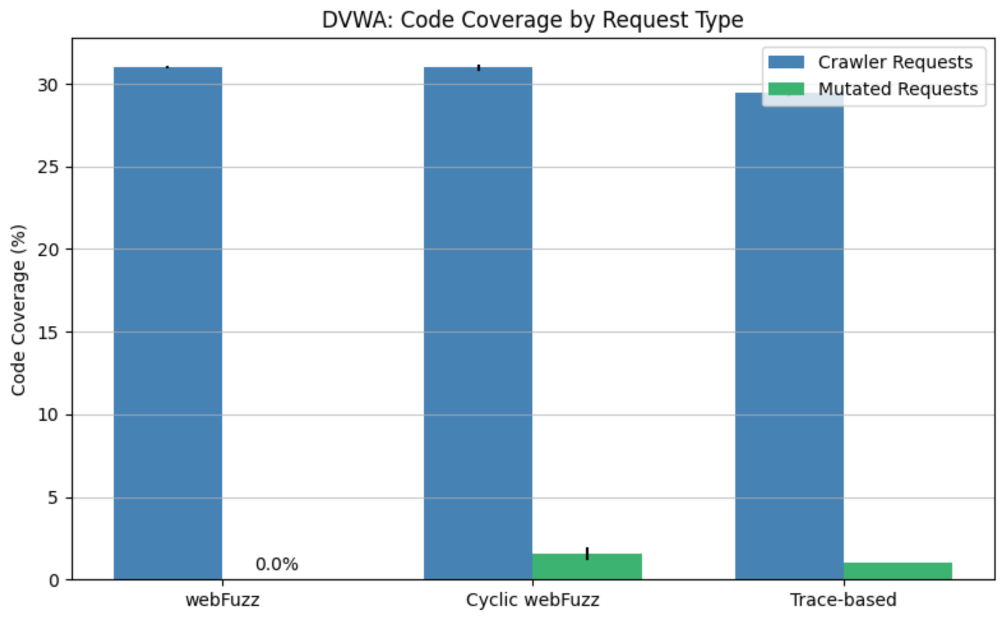

Figure 5.2: DVWA: Code coverage by request type.

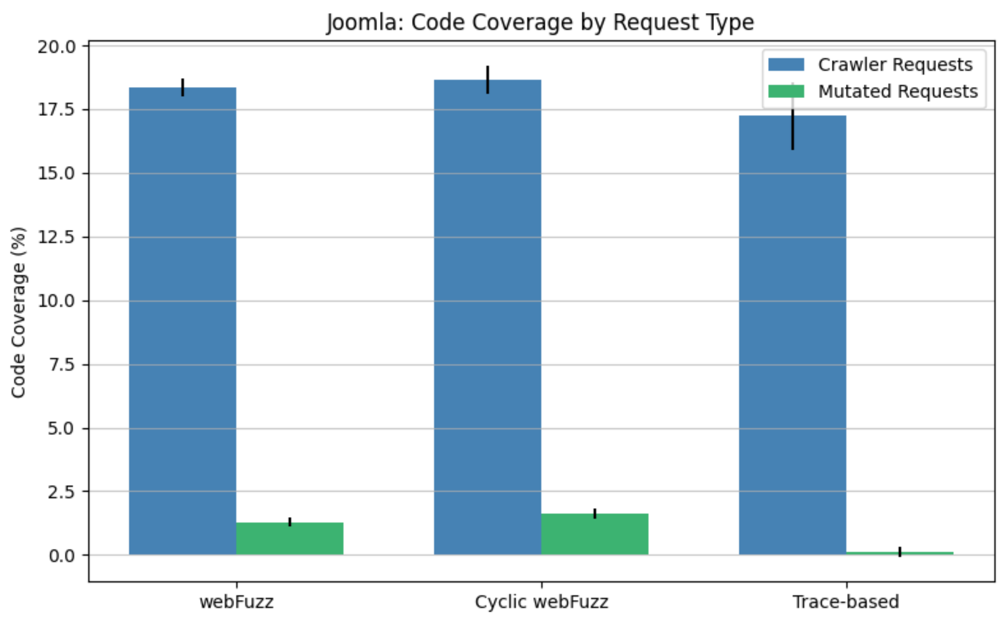

Figure 5.3: Joomla: Code coverage by request type.

Fuzzer	**WordPress (%)**	**DVWA (%)**	**Joomla (%)**
Original WebFuzz	0.93 ± 0.13	0.0 ± 0.0	1.29 ± 0.19
WebFuzz + Cyclic Iteration	1.95 ± 1.15	1.58 ± 0.40	1.64 ± 0.21
Trace-based Feedback (Ours)	$\mathbf{1.35 \pm 0.95}$	$\mathbf{1.03 \pm 0.03}$	$\mathbf{0.12 \pm 0.02}$

Table 5.2: Mean mutated code coverage (%) ± standard deviation for each fuzzer under the same execution time budget.

5.2.3 Code Coverage under Equal Request Counts

Because each fuzzer variant operates at a different speed, the total number of HTTP requests sent during the same time period varies across configurations. The original WebFuzz being instrumentation-based, achieves high throughput and is capable of sending hundreds of thousands of requests within the allotted execution time. In contrast, our feedback mechanism introduces additional overhead due to its trace-oriented design, resulting in significantly fewer requests being processed in the same time window, as illustrated in Figure 5.4. A more detailed analysis of requests over time is provided in Section 5.2.4

This discrepancy in the request volume impacts the fairness of direct coverage comparisons. High-throughput fuzzers have more opportunities to explore the application, which may inflate their overall coverage metrics, not necessarily due to better input quality, but due to sheer volume. Therefore, comparing coverage results without normalizing the number of requests can misrepresent the effectiveness of different feedback mechanisms.

To address this discrepancy, we present a normalized comparison in Tables 5.3 and 5.4, where all fuzzers are evaluated under the same total number of requests. Table 5.3 shows the overall code coverage, while Table 5.4 isolates the coverage achieved from mutated requests only. This normalization allows for a fair comparison of each feedback strategy's effectiveness without being skewed by differences in request volume.

The fixed number of request for each web application in these tables is determined by the number of requests our trace-based feedback method was able to send within the execution time window. As shown in Figure 5.4, this corresponds to 4,500 requests for WordPress, 18,500 for DVWA, and 1,900 for Joomla. These values serve as the fixed request count across all fuzzer configurations to ensure fairness. In other words, the instrumentation-based fuzzers, despite typically operating at higher throughput, were limited to the same number of requests as our trace-based method for the purpose of these comparisons. This setup highlights the efficiency of our approach in selecting high-impact inputs and achieving competitive coverage under identical request constraints.

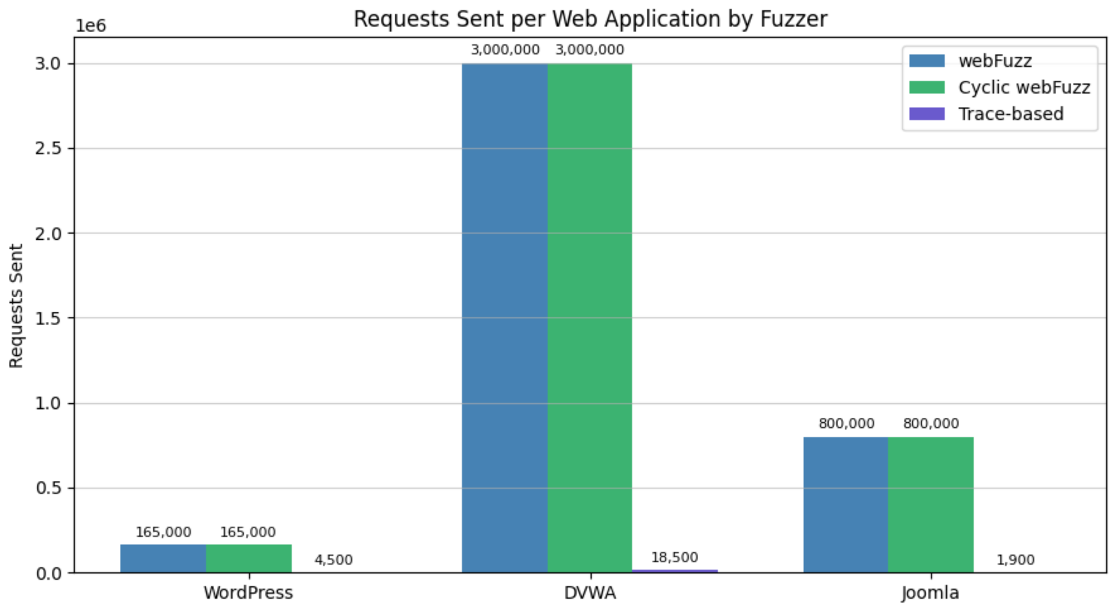

Figure 5.4: Mean number of requests issued by each fuzzer under an equal time budget.

The results presented in Tables 5.3 and 5.4 highlight the effectiveness of each fuzzer when constrained to the same number of requests. In terms of total coverage, both WebFuzz with cyclic iteration and our trace-based feedback approach perform similarly across all three applications. For example, on WordPress and Joomla, our method achieves 19.0% $\pm$ 1.0 and 17.0% $\pm$ 1.0, respectively, only slightly lower than the cyclic variant. On

DVWA, the difference is more pronounced, with the cyclic variant reaching $31.0\% \pm 0.2$ and the trace-based method achieving $29.50\% \pm 0.2$, though the gap remains relatively small.

The mutated request coverage in Table 5.4 offers further insight into how effectively each approach prioritizes new inputs. The original WebFuzz records 0% mutated coverage across all applications. This behavior is expected, as it only begins mutating requests once all crawling paths have been explored, something that does not occur under the limited request count used in this comparison. As a result, no mutated requests are sent.

In contrast, WebFuzz with cyclic iteration introduces mutations earlier by alternating between crawling and mutation phases. It also benefits from instrumentation-based feedback to prioritize inputs more effectively. As a result, it achieves the highest mutated coverage on WordPress ($1.55\% \pm 0.95$) and DVWA ($1.42\% \pm 0.1$). Although our trace-based method yields lower mutated coverage, it performs comparably on these two targets, achieving $1.35\% \pm 0.95$ and $1.03\% \pm 0.03$, respectively. The slightly lower results are expected, given that our method relies on system-level traces rather than direct instrumentation, and operates under stricter overhead and throughput limitations. On Joomla, mutated coverage is generally low across all fuzzers. The cyclic variant reaches $0.14\% \pm 0.04$, while our approach achieves $0.12\% \pm 0.02$, again showing close performance despite the lack of instrumentation.

This suggests that while our method may not reach as many code paths as instrumentation-based approaches, it remains competitive and demonstrates strong potential for guiding effective mutations through language-agnostic, trace-based feedback.

Fuzzer	**WordPress (%)**	**DVWA (%)**	**Joomla (%)**
Original WebFuzz	16.65 ± 0.65	31.0 ± 0.1	16.85 ± 1.5
WebFuzz + Cyclic Iteration	19.5 ± 1.5	31.0 ± 0.2	17.5 ± 0.7
Trace-based Feedback (Ours)	$\mathbf{19.0 \pm 1.0}$	$\mathbf{29.50 \pm 0.2}$	$\mathbf{17.0 \pm 1.0}$

Table 5.3: Mean code coverage (%) ± standard deviation across runs for each fuzzer. All fuzzers were run with the same number of requests per web application.

Fuzzer	**WordPress (%)**	**DVWA (%)**	**Joomla (%)**
Original WebFuzz	0.0	0.0	0.0
WebFuzz + Cyclic Iteration	1.55 ± 0.95	1.42 ± 0.1	0.14 ± 0.04
Trace-based Feedback (Ours)	$\mathbf{1.35 \pm 0.95}$	$\mathbf{1.03 \pm 0.03}$	$\mathbf{0.12 \pm 0.02}$

Table 5.4: Mean code coverage (%) ± standard deviation from **mutated requests only**, for each fuzzer. All fuzzers were run with the same number of total requests per web application.

5.2.4 Request Generation Behavior

To assess the execution behavior of each fuzzer throughout the testing period, we analyze both request throughput and request activity over time. Throughput is defined as the number of HTTP requests issued per second and reflects how efficiently a fuzzer processes inputs and interacts with the target application. In addition to overall throughput, we examine the number of requests issued within fixed time intervals, offering a more detailed view of how request generation evolves and fluctuates during the 600-minute execution window.

At fixed checkpoints during each run, we compute the mean throughput across multiple independent executions. These results are shown in Figures 5.5, 5.6, and 5.7 for WordPress, DVWA, and Joomla, respectively. Additionally, Figures 5.8, 5.9 and 5.10 present the number of requests issued within selected time intervals during the execution. By examining request counts over fixed intervals, we capture how each fuzzer's activity changes throughout the execution.

The original WebFuzz and its cyclic variant demonstrate similar performance in terms of request generation across all applications. Both maintain high throughput throughout the execution, with only minimal differences between them. Among the tested targets, DVWA consistently results in the highest number of requests and throughput for both fuzzers, followed by WordPress and then Joomla. This trend reflects the relative complexity of the target applications: less complex systems like DVWA facilitate faster request processing, whereas more feature-rich platforms such as Joomla exhibit lower throughput due to greater processing overhead and more intricate server-side behavior.

In contrast, the trace-based feedback method issues significantly fewer requests and exhibits a gradual decline in request activity across all applications. This behavior is primarily attributed to the tracing-driven feedback mechanism, which introduces latency during input evaluation. A key contributing factor to the decline in request activity over time is the increasing cost of computing the minimum edit distance between each new trace and the set of previously collected traces. As the number of traces grows, this operation becomes increasingly computationally expensive, thereby reducing throughput over time.

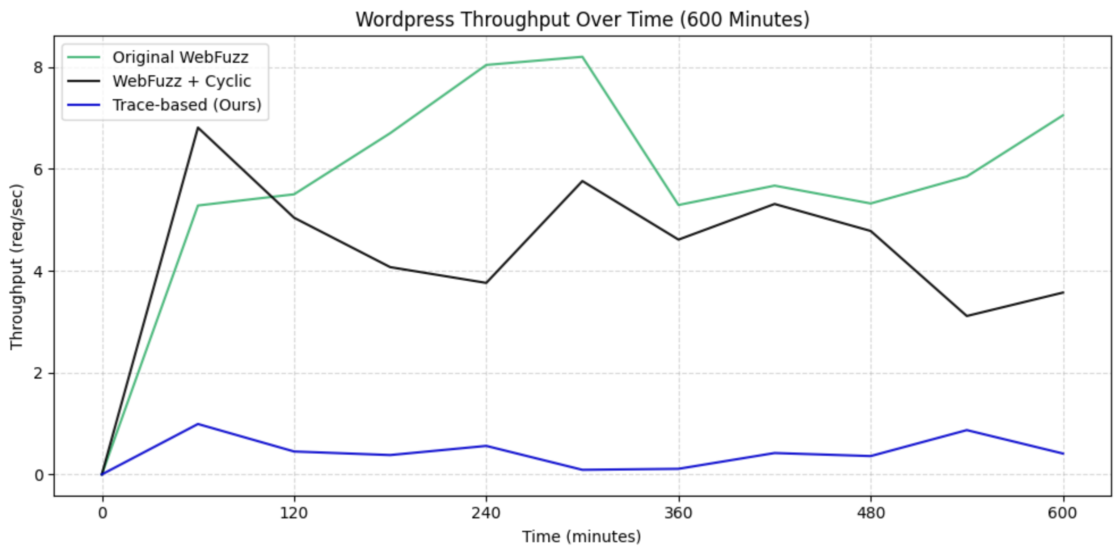

Figure 5.5: Mean request throughput over time for WordPress, based on averaged values at fixed time checkpoints.

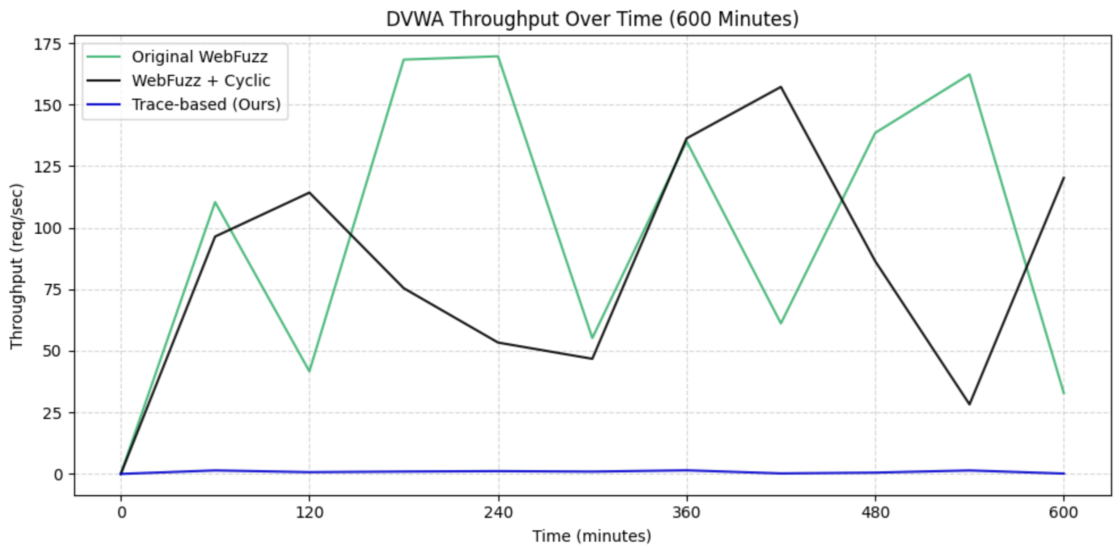

Figure 5.6: Mean request throughput over time for DVWA, based on averaged values at fixed time checkpoints.

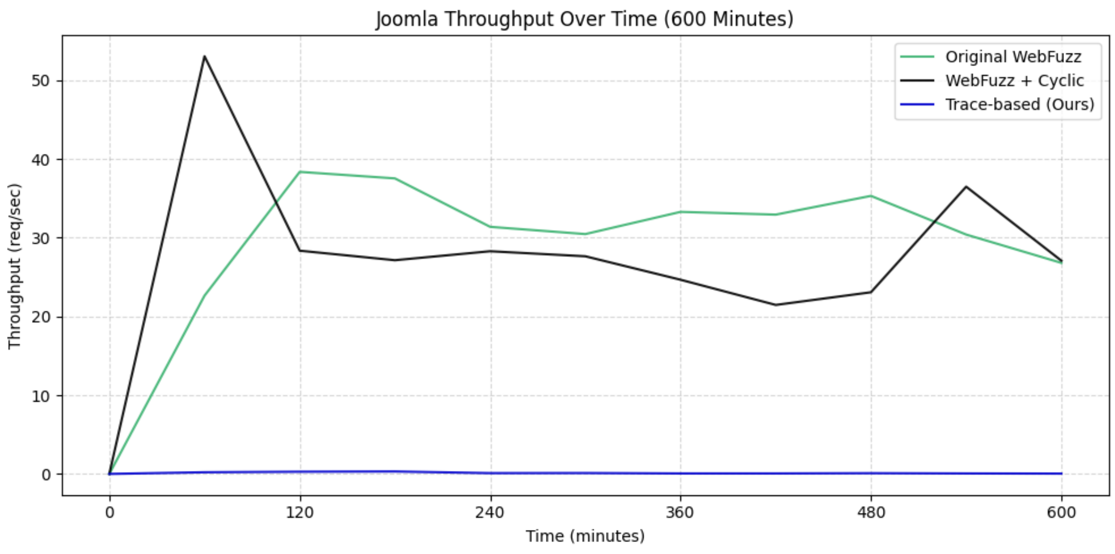

Figure 5.7: Mean request throughput over time for Joomla, based on averaged values at fixed time checkpoints.

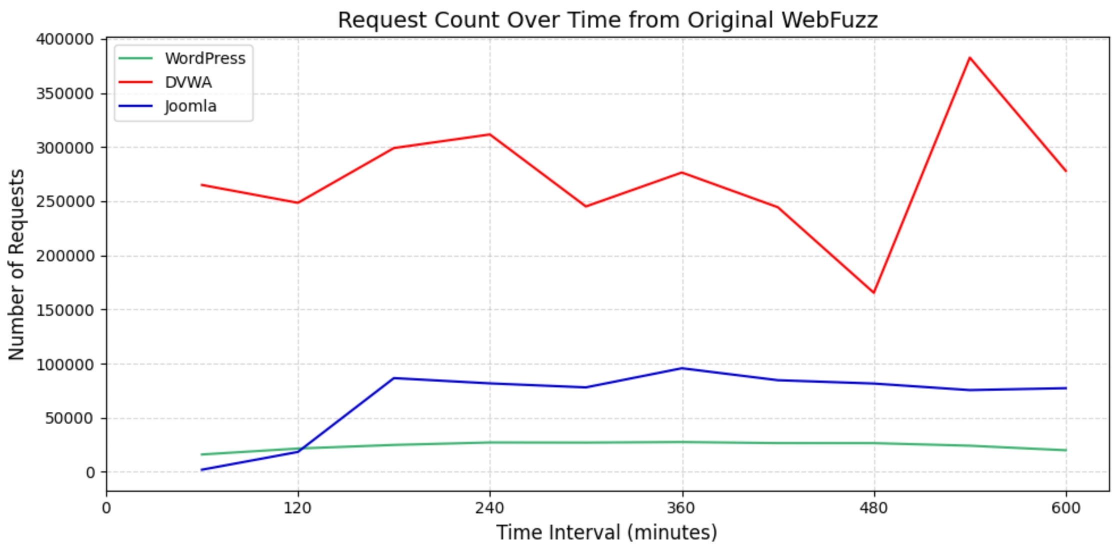

Figure 5.8: Request count over time from Original WebFuzz.

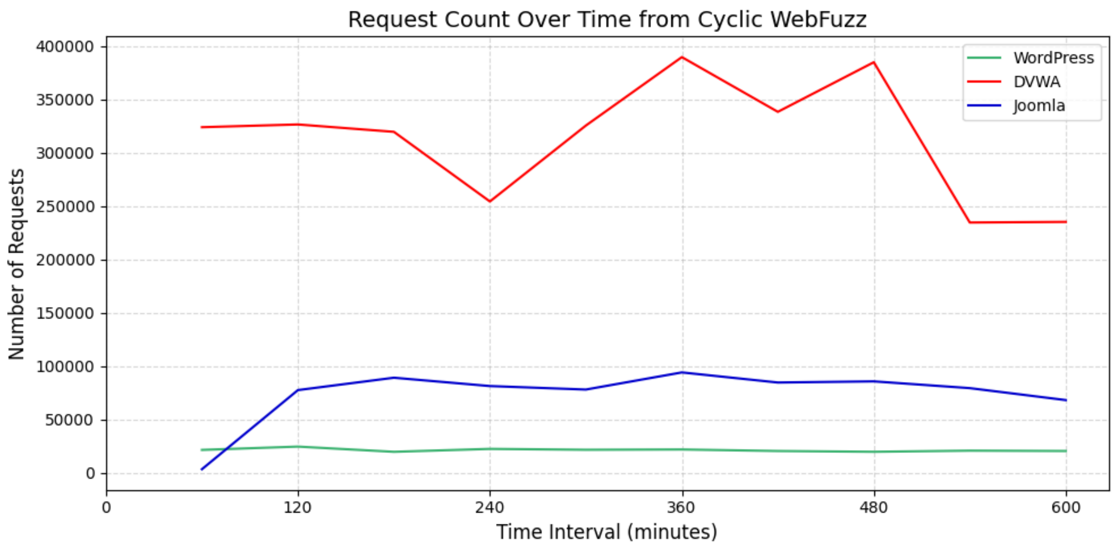

Figure 5.9: Request count over time from Cyclic WebFuzz.

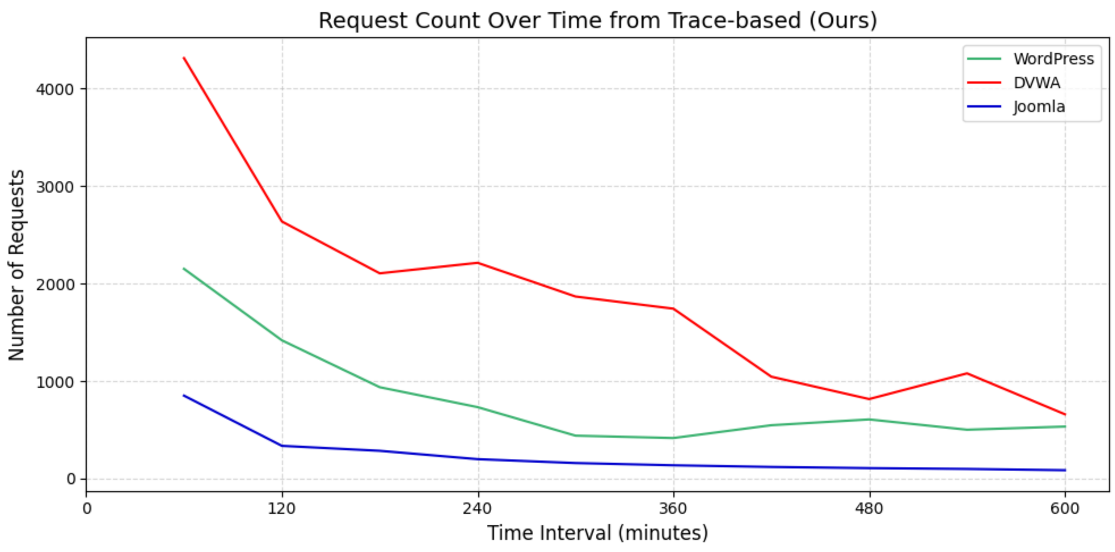

Figure 5.10: Request count over time from Trace-based WebFuzz (Ours).

5.3 Evaluation Summary and Key Findings

The experimental evaluation assessed the effectiveness and efficiency of the proposed trace-based feedback mechanism in comparison to traditional instrumentation-based fuzzing configurations. While the trace-based approach did not reach the same level of code coverage as instrumentation-driven methods when evaluated over equal execution time, it demonstrated competitive performance when normalized by the number of requests. Notably, it achieved comparable mutated request effectiveness to WebFuzz with cyclic iteration, without depending on instrumentation-based feedback.

Throughput analysis confirmed the anticipated trade-off between system-level behavioral tracing and execution speed. The trace-based method issued fewer requests due to the latency introduced by the collection and processing of system-level traces. Nevertheless, it consistently generated high-value inputs that exercised deeper and less accessible parts of the application logic, reinforcing its potential as a practical, language-agnostic alternative to traditional feedback mechanisms.

The evaluation also highlighted areas for improvement in long-running or large-scale fuzzing scenarios, where the cost of trace comparisons can gradually impact performance. Addressing these challenges will be important for ensuring scalability and sustained effectiveness over time.

Chapter 6

Discussion

6.1 Limitations

Despite demonstrating the feasibility of using system-level behavioral signals for feedback-driven fuzzing, the proposed framework faces several practical limitations that constrain its scalability and efficiency.

6.1.1 Overhead from External Tracing Tools

The current implementation relies on external tools such as *strace* to monitor system calls and parses SQL queries from database log files. While these approaches are effective for extracting behavioral traces without modifying the application, they introduce non-trivial runtime overhead. In particular, *strace* incurs significant performance penalties due to its syscall-level tracing, and frequent log parsing introduces delays in capturing SQL query activity. Consequently, the system runs more slowly than the original instrumentation-based WebFuzz, reducing fuzzing throughput during large-scale or time-sensitive testing.

6.1.2 Sequential Execution Constraint

To ensure accurate association between HTTP requests and their corresponding traces, the system processes inputs *sequentially*. This restriction arises from the challenge of matching trace data to specific inputs when multiple requests are in flight concurrently. Running multiple workers in parallel would introduce ambiguity in trace attribution, making it difficult to maintain the accuracy of feedback. Consequently, the prototype sacrifices concurrency for trace precision.

6.1.3 Trace Alignment Sensitivity

The framework uses time-based heuristics to associate system-level behaviors with individual requests. While effective in controlled conditions, this method may misattribute behaviors in the presence of background noise or overlapping activity, particularly in multi-threaded or high-load environments. This can affect the fidelity of feedback and reduce the effectiveness of input prioritization.

6.1.4 Computational Overhead from Edit Distance Calculations

The trace-based feedback mechanism relies on computing the edit distance between each new execution trace and all previously collected traces. This comparison is crucial for assessing behavioral novelty and prioritizing inputs. However, as the number of stored traces grows, the cumulative cost of these comparisons increases significantly. This leads to higher latency during input evaluation and, as observed in our evaluation, a gradual decline in the request generation rate. The unbounded growth in trace comparisons introduces a scalability bottleneck, especially in long-running fuzzing sessions or large-scale deployments, where performance degradation can undermine the overall effectiveness of the fuzzing process.

6.2 Future Work

Although the proposed language-agnostic feedback mechanism demonstrates promising results, several avenues remain open for improvement and further exploration.

6.2.1 Timestamp-Free Trace Association

A key challenge in the current implementation is accurately aligning captured system-level traces with their corresponding HTTP requests using timestamps. This process can be imprecise, especially under concurrent or high-throughput conditions. A potential direction for future work is to eliminate reliance on timing-based correlation altogether. One proposed solution involves introducing a *server on/off mechanism* that temporarily disables request handling between fuzzing iterations. This would provide the server sufficient time to flush all traces related to a single request to persistent storage before accepting a new one, improving the fidelity of trace-to-request association.

6.2.2 Low-Overhead Tracing

The current prototype collects system-level traces using external tools such as *strace* for system calls and database logs for SQL queries. While effective, both approaches introduce notable overhead and can impact performance during fuzzing.

To reduce this overhead, future work could explore more efficient alternatives. For system call monitoring, this may involve developing a lightweight custom tracer using *ptrace* or *kernel-level hooks* to capture only essential syscall metadata. Similarly, SQL query monitoring could be improved by deploying *SQL proxy* tools that transparently intercept queries between the application and the database. These proxies can intercept and extract queries in real time, allowing for low overhead and more reliable SQL trace collection. Together, these enhancements would maintain the non-intrusive and language-agnostic nature of the framework while significantly improving trace collection performance and responsiveness.

6.2.3 Parallelization for Scalable Fuzzing

The current implementation enforces a sequential request execution model to ensure accurate association between each HTTP request and its corresponding system-level traces. While this design guarantees trace integrity, it inherently limits the system's throughput and scalability. A promising direction for future work is to explore opportunities for *parallelizing* components of the fuzzing *pipeline*, notably trace processing and comparison, without compromising the correctness of trace attribution. Enabling concurrent execution while preserving one-to-one request-trace mapping could significantly improve fuzzing efficiency, making the system more suitable for large-scale or time-constrained deployments.

6.2.4 Generalization Beyond WebFuzz Components

Although the prototype modifies key components of WebFuzz, it still inherits parts of the original architecture. Future work could involve fully re-implementing or generalizing those components to eliminate any remaining dependencies on instrumentation-specific logic. This would result in a truly independent, language-agnostic fuzzing framework capable of functioning across a broader range of environments and application types.

Chapter 7

Related Work

7.1 Instrumentation-Based Web Fuzzers

Instrumentation-based web fuzzers rely on modifying the application or its execution environment to collect detailed runtime information, most commonly, code coverage. This feedback is then used to guide input selection and mutation strategies, enabling more targeted and efficient exploration of the program's behavior.

One prominent example is *WebFuzz* [23], a grey-box fuzzing framework designed for PHP-based web applications. WebFuzz performs instrumentation at the Abstract Syntax Tree (AST) level, enabling fine-grained tracking of which parts of the PHP code have been executed. This feedback allows it to prioritize and mutate inputs that are more likely to exercise unexplored code paths, improving the chances of discovering server-side vulnerabilities such as reflected cross-site scripting (XSS). WebFuzz also integrates a crawler, mutator, scheduler, and detector into a unified system, making it a comprehensive solution for grey-box web fuzzing.

Another instrumentation-based tool is PHP-Fuzzer [20], a lightweight coverage-guided fuzzer specifically built for PHP applications. PHP-Fuzzer instruments PHP code using hooks into the interpreter to monitor executed lines or branches during test execution. Then it uses this information to direct the mutation process. Although limited to PHP environments, it demonstrates the practical benefits of combining runtime feedback with input generation, even in interpreted languages.

A more recent advancement is PHUZZ [17], a modular, and open-source coverage-guided fuzzer specifically designed for PHP web applications. Unlike traditional fuzzing approaches that often rely on intrusive instrumentation, such as modifying the application's source code or altering the PHP interpreter, PHUZZ employs a lightweight and non-invasive instrumentation strategy. It utilizes existing PHP extensions, including PCOV and Xdebug, to gather runtime code coverage information, and UOPZ to dynamically hook into and override security-critical functions, such as those handling authentication. This design allows PHUZZ to monitor execution paths and explore deeper application logic without disrupting normal system behavior. While this targeted approach enhances its effectiveness in identifying vulnerabilities within PHP applications, it inherently ties the tool to the PHP ecosystem, thereby limiting its applicability to other programming languages and web application frameworks.

Although instrumentation-based fuzzers are highly effective at guiding input generation through detailed execution feedback, their applicability is often constrained by their language-specific design. These tools are typically tailored to a particular language, limiting their use to a narrow range of web applications built within those ecosystems. This restricts their utility in heterogeneous environments where applications may span multiple languages, components or technologies. The approach proposed in this thesis addresses these limitations by enabling feedback-driven fuzzing without relying on language-specific instrumentation, making it applicable across a broader and more diverse set of web applications.

7.2 Black-Box Web Fuzzers

Black-box web fuzzers operate without any internal knowledge of the application under test. They generate HTTP requests and analyze the resulting responses to infer potential vulnerabilities [3, 13]. Unlike grey-box or white-box fuzzers, they do not rely on instrumentation, code access, or execution tracing. As a result, black-box approaches are highly portable and easy to deploy, but they often suffer from limited visibility into application behavior, making deep or complex bugs harder to detect.

Traditional black-box fuzzing tools like Wfuzz and OWASP ZAP perform large-scale request generation by injecting payloads into various components of HTTP requests, including query strings, form fields, headers, and cookies. Wfuzz is a command-line utility optimized for parameter discovery and brute-force fuzzing of web input vectors [9, 26]. It relies primarily on analyzing coarse-grained HTTP response features, such as status codes and content length, to detect anomalies and potential vulnerabilities. Similarly, OWASP ZAP offers a comprehensive black-box fuzzing framework with an integrated proxy, scanner, and fuzzer [18, 19]. It supports both built-in and customizable payload sets and provides scripting capabilities for tailoring fuzzing campaigns. Despite its flexibility and utility in interactive testing, ZAP also depends on surface-level HTTP response feedback, lacking the ability to reason about deeper application behavior.

Recent advancements in black-box fuzzing, such as Firefly, improve exploration depth by introducing more structured and semantically aware feedback mechanisms [7, 28]. Firefly analyzes full HTTP responses, including status codes, headers, body content, and

DOM structure, and uses a similarity oracle to compare textual and structural elements across responses. This helps prioritize inputs that cause observable changes in behavior, all without requiring instrumentation. Although effective, Firefly's analysis remains limited to surface-level output. In contrast, our approach captures system-level traces, such as system calls and SQL queries, providing deeper insight into the internal execution of the application. This enables more precise feedback and better generalization across diverse web technologies.

In summary, black-box fuzzers have made notable strides by incorporating response-based heuristics and semantic analysis to guide exploration. However, their reliance on externally visible outputs limits the depth of behavioral insight they can achieve. Using system-level traces as feedback, our approach bridges the gap between black-box simplicity and grey-box effectiveness. It enables deeper language-agnostic introspection of application behavior without requiring access to source code or internal instrumentation, offering a practical alternative for fuzzing in heterogeneous and constrained web environments.

Chapter 8

Conclusion

This thesis explored a language-agnostic approach to grey-box fuzzing by replacing traditional instrumentation-based feedback with externally observable system-level behaviors, specifically system calls and SQL queries. The primary motivation was to address the limitations of instrumentation, which is often tightly coupled to specific programming languages or runtime environments, making it impractical for use in diverse systems.

To this end, we extended the WebFuzz framework by redesigning its feedback mechanism. Rather than relying on internal code coverage, our prototype collects and abstracts system-level execution traces. These traces are used to guide the fuzzing process by estimating behavioral novelty, with input prioritization based on edit distance to previously observed traces. This strategy favors inputs that are more likely to trigger previously unexplored behavior.

The evaluation compared the proposed trace-based approach against both the original instrumentation-based webFuzz and a variant employing cyclic input selection. Experimental results across three real-world web applications, WordPress, DVWA, and Joomla, demonstrated that although the trace-based method did not achieve the same raw code coverage within equal execution time, it remained competitive when normalized by the number of requests issued. These findings indicate that, despite the reduced throughput introduced by the trace-based feedback mechanism, the core strategy remains effective in uncovering meaningful behaviors and guiding application exploration.

These findings underscore the potential of system-level behavioral feedback as a practical alternative to traditional instrumentation-based guidance. By leveraging externally observable execution signals, the proposed approach provides a generalized input prioritization mechanism that is adaptable across a broad spectrum of technologies. Its ability to effectively select mutated requests based solely on trace analysis further demonstrates its capacity to support deeper and more meaningful exploration of application behavior in diverse execution environments.

Overall, the evaluation results demonstrate the feasibility of guiding feedback-driven fuzzing through system-level behavioral traces. Although further work is required to improve scalability and reduce runtime overhead, the findings validate the core concept and provide a strong foundation for the continued development of lightweight, instrumentation-free fuzzing frameworks. Future enhancements such as adopting more efficient tracing mechanisms, improving trace-to-request alignment, and fully decoupling the system from instrumentation-based components could enable broader applicability in real-world, multi-language web environments. Additionally, addressing the computational overhead introduced by distance-based trace comparison will be essential for improving scalability and sustaining performance in long-running fuzzing campaigns.

Bibliography

[1] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan. {NAVEX}: Precise and scalable exploit generation for dynamic web applications. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 377–392, 2018.

[2] D. Andriesse. *Practical binary analysis: build your own Linux tools for binary instrumentation, analysis, and disassembly*. no starch press, 2018.

[3] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *2010 IEEE symposium on security and privacy*, pages 332–345. IEEE, 2010.

[4] P. Bisht, T. Hinrichs, N. Skrupsky, and V. Venkatakrishnan. Waptec: whitebox analysis of web applications for parameter tampering exploit construction. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 575–586, 2011.

[5] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[6] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 122–131. IEEE, 2013.

[7] Brum3ns. Firefly - github repository. `https://github.com/Brum3ns/firefly`.

[8] O. Chang, A. Arya, K. Serebryany, and J. Armour. OSS-Fuzz: Five Months Later, and Rewarding Projects. Google Open Source Blog, 2017. `https://opensource.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html`.

[9] CheckOps. Wfuzz - web application fuzzer. `https://www.checkops.com/wfuzz/`.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 3rd edition, 2009.

[11] I. P. A. Dharmaadi, E. Athanasopoulos, and F. Turkmen. Fuzzing frameworks for server-side web applications: a survey. *International Journal of Information Security*, 24(2):73, 2025.

[12] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft. *Queue*, 10(1):20–27, Jan. 2012.

[13] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[14] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[15] A. Marchand-Melsom and D. B. Nguyen Mai. Automatic repair of owasp top 10 security vulnerabilities: A survey. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 23–30, 2020.

[16] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[17] S. Neef, L. Kleissner, and J.-P. Seifert. 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*, pages 1523–1538, 2024.

[18] OWASP. Fuzzing overview. `https://owasp.org/www-community/Fuzzing`.

[19] OWASP. Owasp zap - zed attack proxy. `https://www.zaproxy.org/docs/`.

[20] N. Popov. PHP Fuzzer. `https://github.com/nikic/PHP-Fuzzer`.

[21] K. Serebryany. libFuzzer–a library for coverage-guided fuzz testing. LLVM Project, 2015. `https://llvm.org/docs/LibFuzzer.html`.

[22] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2018.

[23] O. van Rooij, M. A. Charalambous, D. Kaizer, M. Papaevripides, and E. Athanasopoulos. webfuzz: Grey-box fuzzing for web applications. In *Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26*, pages 152–172. Springer, 2021.

[24] V. Vikram, I. Laybourn, A. Li, N. Nair, K. OBrien, R. Sanna, and R. Padhye. Guiding greybox fuzzing with mutation testing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 929–941, 2023.

[25] M. Wang, J. Liang, C. Zhou, Z. Wu, J. Fu, Z. Su, Q. Liao, B. Gu, B. Wu, and Y. Jiang. Data coverage for guided fuzzing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2511–2526, Philadelphia, PA, Aug. 2024. USENIX Association.

[26] Wfuzz Developers. Wfuzz documentation. `https://wfuzz.readthedocs.io/en/latest/`.

[27] S. Yan, C. Wu, H. Li, W. Shao, and C. Jia. Pathafl: Path-coverage assisted fuzzing. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 598–609, 2020.

[28] YesWeHack. Firefly - a smart black-box fuzzer for web applications. `https://www.yeswehack.com/learn-bug-bounty/firefly-v1-1-0-a-smart-black-box-fuzzer-for-testing-web-applications`.

[29] M. Zalewski. American fuzzy lop trophy case. `https://lcamtuf.coredump.cx/afl/`, 2015.

[30] M. Zalewski. More about afl - afl 2.53b documentation. `https://afl-1.readthedocs.io/en/latest/about_afl.html`, 2019.

[31] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. Part ii: Lexical fuzzing. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2023. `https://www.fuzzingbook.org/html/02_Lexical_Fuzzing.html`.

[32] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. Concolic fuzzing. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. `https://www.fuzzingbook.org/html/ConcolicFuzzer.html`.

[33] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. Symbolic fuzzing. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2025. `https://www.fuzzingbook.org/html/SymbolicFuzzer.html`.