UNIVERSITÀ DEGLI STUDI DI CATANIA

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

PHD IN COMPUTER SCIENCE XXXVII CYCLE

Marcello Maugeri

Automation Challenges and Solutions in Coverage-Guided Fuzzing of Multiprocess Software Systems

---

PH.D. DISSERTATION

Supervisor: Dr. Giampaolo Bella

Reviewers: Dr. Karine Even-Mendoza

Dr. Van-Thuan Pham

Declaration of Authorship

I, Marcello Maugeri, declare that this dissertation, titled "Automation Challenges and Solutions in Coverage-Guided Fuzzing of Multiprocess Software Systems", has not been submitted, in whole or in part, for any other degree or professional qualification. I confirm that all sources of information and external contributions are acknowledged and correctly referenced. The core contributions set forth in the main chapters of this dissertation are the outcome of research in which I assumed the leading role in both execution and authorship. The appendices may contain material from collaborative projects, in such cases, the sections presented are mainly focused on my own contributions. I further confirm that artificial intelligence tools were employed solely for editorial purposes such as grammar corrections and style improvements. All ideas, analyses and conclusions presented are my own.

Date: 30/10/2025

Signed: Marcello Maugeri

Abstract

Modern software systems typically rely on multiprocess architectures to achieve scalability, fault isolation, and enhanced performance. Notable examples are web servers, database systems, and browsers, which handle incoming requests, user sessions, or plugins through separate processes. However, this distributed design introduces significant challenges for automated security testing methodologies.

Among the various automated security testing techniques, Coverage-Guided Fuzzing has been widely adopted for software vulnerability discovery. In brief, it leverages runtime code coverage as testcase execution feedback to drive mutation-based input generation with the objective of enhancing system exploration.

Despite its effectiveness, current fuzzers are not designed to comprehensively test multiprocess software systems, creating blind spots in vulnerability detection. In fact, existing fuzzers operate under the assumption of single-process execution within clear fuzzing loop boundaries.

Multiprocess software systems break these assumptions: processes run concurrently, accept inputs through different inter-process communication mechanisms, code coverage scatters across multiple address spaces, and bugs occur in unmonitored child processes. These limitations stem from a lack of fork-awareness: the capability to detect, trace, and coordinate testing across process boundaries created by fork() system calls.

This dissertation addresses the automation challenges of coverage-guided fuzzing for multiprocess software systems. The research identifies four core problems: execution synchronisation across concurrent processes, comprehensive test oracle design for multiprocess bug detection, coverage observation across distributed execution, and multi-input generation and injection through diverse inter-process communication mechanisms.

The primary contributions include: (1) empirical evidence demonstrating significantly lower coverage in multiprocess software within OSS-Fuzz projects, (2) the formal definition of fork-awareness and a systematic evaluation framework revealing fundamental limitations across fourteen state-of-the-art fuzzers, and (3) the design

and implementation of ForkFuzz, the first fork-aware coverage-guided fuzzer that addresses three of the four core challenges through process tree monitoring, cross-process anomaly detection, and distributed coverage aggregation.

Ultimately, this work establishes the foundation for automated coverage-guided fuzzing of multiprocess software systems, with complementary contributions extending to specialised domains including stateful protocol testing and GraphQL API security assessment.

Contents

Declaration of Authorship i

Abstract ii

List of Figures ix

List of Tables xi

List of Acronyms xii

List of Code Snippets xiii

1 Introduction 1
1.1 Challenges in Coverage-Guided Fuzzing of Multiprocess Software . . . 2
1.2 Contributions . . . 4
1.3 Outline . . . 5
1.4 List of Publications . . . 6

2 Fuzzing Fundamentals 8
2.1 Fuzzing Taxonomies . . . 9
2.1.1 Knowledge-Based Taxonomy . . . 9
Black-Box Fuzzing . . . 9
White-Box Fuzzing . . . 10
Grey-Box Fuzzing . . . 11
2.1.2 Input-Based Taxonomy . . . 11
Generation-Based Fuzzing . . . 12
Mutation-Based Fuzzing . . . 12
Structure-Aware Fuzzing . . . 13

2.1.3 State-Based Taxonomy 14
Stateless Fuzzing 14
Stateful Fuzzing 14
2.2 Coverage-Guided Fuzzing 14

3 State of the Art **19**
3.1 Network Fuzzing 20
3.2 Hypervisor-based Fuzzing 24
3.3 Kernel Fuzzing 28
3.4 Concurrent Fuzzing 29
3.5 Embedded Fuzzing 30
3.6 Summary 30

4 Motivation and Gaps in Multiprocess Fuzzing **32**
4.1 Challenge C1: Multiprocess Execution Synchronisation 33
4.2 Challenge C2: Comprehensive Test Oracle Design 33
4.3 Challenge C3: Multiprocess Coverage Observation 34
4.4 Challenge C4: Multi-Input Generation 35
4.5 Empirical Study: Multiprocess Software in OSS-Fuzz Projects 35
4.5.1 Research Questions 36
4.5.2 Project Selection 36
4.5.3 Static Analysis Pipeline 37
4.5.4 Statistical Tests 38
4.5.5 Threats to Validity 38
4.5.6 Results 38
4.6 Motivating Example: Lighttpd Web Server 39
4.6.1 Lighttpd Architecture Overview 39
4.6.2 Challenge C1: Execution Synchronisation in Lighttpd 40
4.6.3 Challenge C2: Oracle Design in Lighttpd 40
4.6.4 Challenge C3: Coverage Observation in Lighttpd 41
4.6.5 Challenge C4: Multi-Input Generation in Lighttpd 42

5 Evaluating the Fork-Awareness of Coverage-Guided Fuzzers **43**
5.1 Methodology 44

5.1.1 Deriving Requirements from Execution Management 44
5.1.2 Defining Fork-Awareness 44
5.1.3 Challenge Programs 45
Bug Detection Challenge (C1) 46
Hang Detection Challenge (C2) 46
Coverage Tracking Challenge (C3) 46
5.1.4 Experimental Setup 47
Fuzzer Selection 47
Test Environment 47
5.2 Experiments 48
5.2.1 Implementation Details 48
AFL-based Fuzzers 48
LibFuzzer-based Fuzzers 49
Honggfuzz 49
5.2.2 Execution Results 49
5.3 Evaluation 49
5.3.1 Analysis of Results 49
Bug Detection (C1) 49
Hang Detection (C2) 50
Coverage Tracking (C3) 51
5.3.2 Implications for Fuzzing Effectiveness 51
Missed Vulnerabilities 51
Incomplete Coverage 52
False Confidence 52
5.4 Research Findings 52
5.4.1 Key Observations 52
5.4.2 Future Directions 52
5.4.3 Existing Workarounds and Their Limitations 53
5.5 Limitations and Threats to Validity 54
5.5.1 Construct Validity 54
5.5.2 Internal Validity 54
5.5.3 External Validity 54
5.6 Summary 54

6 ForkFuzz: Leveraging Fork-Awareness in Coverage-Guided Fuzzing 56
6.1 Motivating Scenario . . . 57
6.2 Methodology . . . 59
6.2.1 Design Principles . . . 59
6.2.2 Architectural Components . . . 59
Process Identifier Set (PIDS) . . . 59
Ptrace-based Event Monitoring . . . 59
Global Timeout Management . . . 60
6.2.3 Workflow . . . 60
Setup Phase . . . 60
Execution Phase . . . 62
Termination Phase . . . 63
6.3 Implementation . . . 64
6.3.1 Integration with Honggfuzz . . . 65
Process Management Extensions . . . 66
Ptrace Event Handling . . . 66
Timeout Detection Enhancement . . . 67
6.3.2 Coverage Aggregation . . . 67
6.3.3 Network Fuzzing Support . . . 68
6.4 Evaluation . . . 68
6.4.1 Experimental Setup . . . 68
6.4.2 Case Study 1: Dining Philosophers Problem . . . 69
Implementation . . . 69
Results . . . 69
6.4.3 Case Study 2: Producer-Consumer Problem . . . 69
Implementation . . . 69
Results . . . 70
6.4.4 Case Study 3: Fork-based HTTP Server . . . 70
Implementation . . . 70
Results . . . 70
6.4.5 Performance Analysis . . . 71
Overhead Measurement . . . 71
Scalability Analysis . . . 71

6.5 Discussion . . . 71
6.5.1 Limitations . . . 71
Fork-Join Pattern Handling . . . 71
Persistent Mode Incompatibility . . . 72
Platform Dependency . . . 72
Coverage Attribution . . . 72
6.5.2 Aggregated Coverage . . . 72
6.5.3 Areas of Improvement . . . 73
Distributed Fuzzing . . . 73
Real-World Benchmark . . . 73
Cross-Platform Support . . . 73
Concurrency Bug Detection . . . 73
6.6 Research Findings . . . 73
6.6.1 Key Contributions . . . 73
6.6.2 Comparison with Alternative Approaches . . . 74
Defork and Process Flattening . . . 74
Custom Harnesses . . . 74
6.6.3 Practical Impact . . . 75
6.6.4 Theoretical Implications . . . 75
6.7 Summary . . . 76

7 Conclusion **78**
7.1 Summary of Contributions . . . 78
7.2 Key Findings . . . 79
7.3 Limitations and Future Directions . . . 80
7.4 Broader Impact . . . 81
7.5 Conclusion . . . 81

A GraphQL Testing **83**
A.1 BenGQL: An Extensible Benchmarking Framework . . . 85
A.1.1 Framework Design and Architecture . . . 85
A.1.2 Evaluation Infrastructure . . . 86
A.1.3 Impact on GraphQL Testing Research . . . 87
A.1.4 Case Study Selection and Characteristics . . . 88

A.2 KrakQL: LLM-Guided Blind Schema Introspection 89
A.2.1 The Blind Introspection Challenge 90
A.2.2 Search-Based Schema Discovery 90
A.2.3 Experimental Evaluation 92
A.2.4 Connections to Coverage-Guided Fuzzing 93
A.3 Wendigo: Deep Reinforcement Learning for DoS Query Discovery . . 94
A.3.1 Limitations of Existing Fuzzing Frameworks for GraphQL . . 94
A.3.2 Empirical Evaluation of EvoMaster's DoS Discovery Capabilities 96
A.3.3 Wendigo's Deep Reinforcement Learning Solution 97
A.3.4 Experimental Results and Comparative Analysis 98
A.4 Concluding Remarks 100

B Stateful Fuzzing 103
B.1 LibAFLstar: Fast and State-Aware Protocol Fuzzing 103
B.1.1 Infrastructure Contributions 104
Docker Environment Setup 104
Target Preparation and Patching 105
Evaluation Harness Development 106
B.1.2 Experimental Results 106
B.2 Fuzzing OPC UA: A Research Experience 107
B.2.1 Project Overview 107
B.2.2 Technical Contributions 107
Protocol Integration 107
Benchmark Development 108
Comparative Analysis 108
B.2.3 Impact and Future Directions 109
B.3 Lessons Learned 109

List of Figures

6.1 Fork-based web server architecture where the main process spawns worker processes to handle individual client connections, exemplifying the multiprocess fuzzing challenge. 57

6.2 Fork-aware fuzzing captures bugs from child processes that would otherwise remain undetected by traditional fuzzers monitoring only the parent process. 58

6.3 Setup phase: ForkFuzz initialises the $PIDS$ set and establishes process group management for comprehensive tracking. 61

6.4 Fork event handling: When a process forks, ForkFuzz captures the child PID, adds it to $PIDS$, and automatically attaches the tracer. . 62

6.5 Exit event handling: When a process terminates, ForkFuzz removes it from $PIDS$ and checks for abnormal termination. 63

6.6 Termination phase: ForkFuzz checks if all processes terminated naturally or reports timeouts for hanging processes. 77

A.1 BenGQL framework workflow showing the interaction between case studies, testing tools, experiment driver, and analysis module. 85

A.2 Case study selection process showing how filtering criteria progressively reduced the candidate pool to 23 representative GraphQL applications. 88

A.3 KrakQL architecture showing the multi-agent system and novelty search scheduler (novel contributions highlighted). 91

A.4 Wendigo query discovery workflow showing the DRL agent's interaction with the GraphQL environment. 98

A.5 WENDIGO response time evaluation showing superior DoS query discovery compared to EVOMASTER and random baselines on DVGA target. Results demonstrate WENDIGO's ability to discover resource-intensive queries that cause significant server delays. 99

List of Tables

3.1 Network fuzzing approaches categorised by execution strategy and input feeding mechanism. . . . 25

4.1 Static-analysis patterns used to identify IPC usage in C/C++ sources. All patterns required word boundary checks ((?<![a-zA-Z0-9_])) and appeared in non-comment code. . . . 37

5.1 Fork-awareness evaluation results for 14 coverage-guided fuzzers . . . 50

6.1 Dining Philosophers Problem detection results . . . 69

6.2 Producer-Consumer Problem bug detection . . . 70

A.1 Overview of BenGQL case studies showing diversity in engines, languages, and scale. . . . 89

A.2 Schema coverage comparison between Clairvoyance and KrakQL across seven GraphQL applications. . . . 92

A.3 Discovery efficiency comparison showing success rates per 64-name batch. . . . 93

Acronyms

AI Artificial Intelligence

API Application Programming Interface

CFG Control-Flow Graph

CGF Coverage-Guided Fuzzing

FTP File Transfer Protocol

HTTP Hypertext Transfer Protocol

IPC Inter-Process Communication

KVM Kernel-based Virtual Machine

LLM Large Language Model

MITM Man-In-The-Middle

ML Machine Learning

PT Processor Trace

QEMU Quick Emulator

RTSP Real-Time Streaming Protocol

SDLC Software Development Life Cycle

SMTP Simple Mail Transfer Protocol

SUT System-Under-Test

TCG Tiny Code Generator

VM Virtual Machine

List of Code Snippets

3.1 Excerpt from SnapFuzz source code for signal propagation in child processes[1]. . . . 22
5.1 Bug detection challenge 46
5.2 Hang detection challenge 46
5.3 Coverage tracking challenge 46
6.1 AFL-style signal handler limitation: only main process monitored . . 65
6.2 Honggfuzz ptrace approach: system-level process tree visibility . . 65
6.3 Process identifier set data structure with thread-safe operations . . . 66
6.4 Fork event interception using ptrace 66
6.5 Multiprocess timeout detection 67

Chapter 1

Introduction

The increasing complexity of software systems provides more and more services to people. At the same time, cybercriminals exploit inherent weaknesses in such systems to gain personal revenue, causing – directly or not – financial loss to both companies and individuals. As a consequence, both user and security testing plays a pivotal role in detecting potential vulnerabilities [1], improving the reliability and security of software systems.

To this extent, *Fuzz Testing*, mostly known as *Fuzzing*, has gained recognition among the well-established security testing methods due to its promise of automation. Among fuzzing techniques, *Coverage-Guided Fuzzing* has emerged as the most effective technique, using runtime feedback to steer input generation towards unexplored code paths. This technique leverages compile-time instrumentation to monitor program execution, collecting code coverage information to prioritise inputs that discover new code paths. Ideally, Coverage-Guided Fuzzing (CGF) should minimise manual intervention, enabling efficient and large-scale testing. However, in practice, achieving full automation is often challenging.

A major obstacle to fuzzing automation lies in the need to write a *harness*. In a nutshell, a harness acts as a bridge between the fuzzer and the System-Under-Test (SUT). This bridge is crucial as it feeds the fuzzer's testcase to the SUT and monitors the execution for results. Essentially, the harness guides the fuzzing process by directing the generated inputs to the correct entrypoints within the SUT.

Crafting a harness, however, is far from trivial. It necessitates a deep understanding of the SUT's internals, its input/output interfaces, and the input format it employs.

This process is often labor-intensive and error-prone, requiring significant manual effort from skilled testers. Furthermore, as the SUT evolves, the harness needs to be updated accordingly, adding to the maintenance burden.

Current research in fuzzing automation splits into two main directions: the first focuses on automating the harness creation process, while the second aims to design complex fuzzing frameworks that improve automation by shifting the effort from the tester to the fuzzer itself.

The first direction, known as *fuzz driver generation*, has been gaining significant traction, particularly for library Application Programming Interface (API) fuzzing [2, 3]. Recent advancements leveraging Large Language Models (LLMs) have achieved notable results, with tools like OSS-FUZZ-GEN [4] demonstrating automated generation of effective fuzz targets. However, as library APIs typically define numerous functions, effective fuzz driver generation requires prioritisation to determine which functions are worth targeting. In a complementary study to the present dissertation, Machine Learning (ML) models for vulnerability detection have been proposed to address this challenge by identifying potentially vulnerable functions that should be prioritised for automated fuzz driver generation [5].

However, this unit-level approach focuses on testing individual library functions or components in isolation. This dissertation focuses on the second direction, specifically addressing *end-to-end fuzzing* – referred to as *system-level fuzzing* in this dissertation – which tests complete software systems from external interfaces through their entire execution workflows. In particular, this dissertation presents and addresses the challenges of improving automation in CGF of *multiprocess software systems*.

1.1 Challenges in Coverage-Guided Fuzzing of Multiprocess Software

Modern software systems often employ multiprocess architectures that distribute execution across multiple processes. For example, web servers spawn worker processes to handle concurrent requests, browsers isolate tabs in separate processes for

security, and databases fork processes for efficient query execution. This architectural prevalence makes effective testing of multiprocess software critically important for system security.

While effective, CGF usually works under the assumption that the fuzzing process happens inside a *fuzzing loop*. An iteration of this loop involves the generation of a testcase, the execution of the SUT with this testcase, and code coverage collection to identify which parts of the code were exercised. However, this requires the SUT to execute within clear boundaries, meaning the fuzzer feeds the testcase and waits for the testcase to be processed before the next iteration.

Unfortunately, multiprocess software breaks these fundamental assumptions. When programs create child processes through `fork()` or similar mechanisms, execution becomes distributed across multiple independent address spaces. Code coverage information scatters across processes, crashes occur in child processes where test oracles such as bug detection mechanisms may not observe them, and execution lifecycles become unclear as multiple processes run concurrently especially as they may receive inputs from different sources. This architectural complexity gives rise to four core challenges that must be addressed to achieve effective automated testing:

`C1`**) Multiprocess Execution Synchronisation.** This challenge involves determining when a fuzzing iteration completes across multiple concurrent processes and managing the transition to the subsequent iteration. Without proper synchronisation mechanisms, fuzzers lose control of execution lifecycle, leading to resource exhaustion, incomplete testing cycles, and zombie processes that interfere with subsequent iterations.

`C2`**) Comprehensive Test Oracle Design.** Traditional test oracles evaluate single process execution through mechanisms such as signals or sanitisers instrumentation. However, in multiprocess settings, execution spans multiple independent processes that may evade oracle observation, resulting in undetected bugs. As a consequence, the challenge focuses on designing test oracles that maintain comprehensive visibility across the SUT, enabling detection of crashes, hangs, and other conditions relevant to the testing objectives.

`C3`**) Multiprocess Coverage Observation.** Code coverage – whether line, branch or

edge coverage – is usually collected in a bitmap through compile-time instrumentation probes. However, in multiprocess settings, multiple processes may execute the same code paths concurrently, leading to duplicated or inconsistent coverage information. The challenge focuses on designing feedback mechanisms that can collect code coverage information from all processes, enabling comprehensive guidance of input generation.

C4) Multi-Input Generation. Multiprocess software typically provides multiple entrypoints through diverse Inter-Process Communication (IPC) mechanisms, creating multiple input vectors. In fact, each input source – whether file, network sockets, or shared memory regions – requires specific input formats and presents unique challenges. Effective multiprocess fuzzing must handle these input sources concurrently to achieve comprehensive system coverage, while generating contextually appropriate testcases for each input source based on the expected syntax and semantics.

These challenges define the fundamental obstacles to achieve automation in CGF of multiprocess software. Failing to address the distributed nature of multiprocess execution, leads to incomplete testing coverage and potential missed vulnerabilities. While other challenges exist – such as reproducibility in a stateful environment, discussed in Appendix B – this dissertation concentrates on how to systematically achieve automation, providing a solid baseline that can be extended to address additional challenges in future work.

1.2 Contributions

This dissertation makes the following contributions to the fields of Coverage-Guided Fuzzing and Automated Software Testing:

- **Empirical evidence of Multiprocess CGF gaps**: A large-scale statistical study on OSS-Fuzz demonstrating that projects employing multiprocess architectures and IPC mechanisms achieve significantly lower coverage than others, providing empirical evidence that there is a need of multiprocess CGF techniques.
- **Definition and systematic evaluation of the Fork-Awareness**: A formal definition of the concept of *fork-awareness* in CGF and an evaluation framework

that systematically assesses how existing state-of-the-art fuzzers handle the execution synchronisation, oracle problem, and observability challenges. The evaluation demonstrates fundamental limitations across 14 coverage-guided fuzzers.

- **Design and implementation of ForkFuzz**: The first fork-aware coverage-guided fuzzer based on HONGGFUZZ, which addresses: execution synchronisation (`C1`) through comprehensive process tree monitoring, oracle problem (`C2`) through cross-process anomaly detection, and observability (`C3`) through coverage aggregation across multiple processes.

1.3 Outline

This dissertation is organized as follows:

Chapter 2 provides essential background on fuzzing techniques, multiprocess programming concepts, and CGF methodologies that form the foundation for this research.

Chapter 3 presents a comprehensive state-of-the-art analysis of existing fuzzing approaches, identifying gaps in multiprocess software testing and positioning this work within the broader fuzzing research landscape.

Chapter 4 motivates the research through a large-scale empirical study on OSS-FUZZ projects, demonstrating that multiprocess software using IPC mechanisms achieves significantly lower coverage, and presents concrete examples of current fuzzing limitations with multiprocess software.

Chapter 5 introduces the concept of fork-awareness and presents a systematic evaluation framework for assessing existing coverage-guided fuzzers' ability to handle multiprocess software.

Chapter 6 details the design and implementation of FORKFUZZ, the first fork-aware coverage-guided fuzzer, including its technical innovations and empirical evaluation.

Chapter 7 concludes the dissertation with a summary of research contributions and an outline of future research directions.

1.4 List of Publications

Core Contributions This dissertation is supported by a body of published and submitted work. The core findings presented in the main chapters stem from three primary publications that systematically address the challenges of multiprocess fuzzing.

- **Evaluating the Fork-Awareness of Coverage-Guided Fuzzers**, written by ***Marcello Maugeri****, Cristian Daniele, Giampaolo Bella, Erik Poll*, published in *Proceedings of the 9th International Conference on Information Systems Security and Privacy ICISSP* in 2023.
- **Forkfuzz: Leveraging the Fork-Awareness in Coverage-Guided Fuzzing**, written by ***Marcello Maugeri****, Cristian Daniele, Giampaolo Bella*, published in *Computer Security - ESORICS 2023 International Workshops* in 2023.

Beyond the core contributions, this dissertation presents complementary research that broadens the understanding of automated testing challenges. This includes explorations of Artificial Intelligence (AI) techniques for GraphQL security testing, and contributions to stateful protocol fuzzing. While these studies are not central to the dissertation's main thesis, they provide valuable context and are discussed in Appendices A and B.

- **Wendigo: Deep Reinforcement Learning for Denial-of-Service Query Discovery in GraphQL**, written by *Shae McFadden,* ***Marcello Maugeri****, Chris Hicks, Vasilios Mavroudis, Fabio Pierazzi*, published in *Proceedings of IEEE Symposium on Security and Privacy Workshops (SPW)* in 2024.
- **BenGQL: An Extensible Benchmarking Framework for Automated GraphQL Testing**, written by *Abenezer Kebede,* ***Marcello Maugeri***, to appear in *IEEE ACM International Conference on Automated Software Engineering* in 2025.
- **KrakQL: LLM-Guided Blind Introspection of GraphQL Schemas**, written by ***Marcello Maugeri****, Abenezer Kebede, Giampaolo Bella*, to appear in *Symposium on Search-Based Software Engineering* in 2025.
- **LibAFLstar: Fast and State-aware Protocol Fuzzing**, written by Cristian Daniele, Timme Bethe, **Marcello Maugeri**, Andrea Continella, Erik Poll, published in *Computer Security - ESORICS 2025* in 2025

- **Fuzzing OPC UA with AFLNet, ChatAFL and LibAFLstar: A Research Experience Paper**, written by ***Marcello Maugeri***, *Cristian Daniele, Federico Fausto Santoro*, published in *Proceedings of IEEE Conference on Pervasive and Intelligent Computing (PICom)* in 2025.

Chapter 2

Fuzzing Fundamentals

The concept of *Fuzzing* – which is the abbreviation of *fuzz testing* – was first introduced in the late 1980s, when *Professor Barton Miller* assigned his students the task of developing a tool known as *Fuzz*. This tool generated a stream of random characters and fed this stream into Unix utilities to assess their robustness. The efficacy of this approach was demonstrated by the identification of numerous bugs that resulted in the failure or hanging of the target utility.

Nowadays, fuzzing is a standard practice throughout the Software Development Life Cycle (SDLC), from design to deployment, as set out in numerous standards (e.g. *ISO/IEC/IEEE 29119* [6] or *NIST SP 800-53* [7]). In fact, fuzzing positions itself as a *dynamic testing* technique, which can be employed for different scopes, including *regression testing* [8, 9], but is widely known for its role in *reliability testing* [10] and *security testing* [11] due to its ability to uncover bugs that are likely to lead to vulnerabilities in the SUT.

In essence, Algorithm 1 illustrates the core principles of fuzzing in a simplified form. As first step, the *fuzzer*, samples inputs, called *testcases*, from the *input space*, executes the SUT with these testcases, and observes the outputs, storing the inputs that lead to unexpected behaviour – or, in other words, a bug. This process is repeated iteratively until a stopping condition is met, such as the expiration of a time budget.

Since its inception, Fuzzing has evolved significantly, delivering a multitude of different techniques to test a multitude of complex different targets, from OS kernels [12] to Web APIs [13, 14, 15], to GUI apps [16] and even smart contracts [17, 18].

Algorithm 1: Fuzzing Algorithm (simplified)

Input: Input space S
Output: testcases leading to bugs $S_\times$

```
repeat
    testcase ← sample(S) ;                                  // sample input
    feedback ← execute(testcase) ;          // run the target with the input
    result ← analyse(feedback) ;                       // analyse the feedback
    if result is bug ;                  // check if the result is unexpected
     then
        S× ← S× ∪ {testcase} ;     // store the input for further analysis
until ;                                                      // fuzzing loop
;
return S×
```

Any target faces different challenges, which are addressed by different fuzzing techniques. These techniques can be classified based on different criteria, leading to various fuzzing taxonomies, which are discussed in the next section (2.1).

2.1 Fuzzing Taxonomies

2.1.1 Knowledge-Based Taxonomy

In *Software Testing*, testing approaches are classified by the prior knowledge a tester has about the SUT. This structure is used also in other fields such as penetration testing, where the tester can have different levels of knowledge and access to the target system. A similar concept is applied to fuzzing, where the tester can have different levels of knowledge about the target system at runtime.

Black-Box Fuzzing

Black-box fuzzing refers to the scenario where the fuzzer has no knowledge of the internal logic of the SUT. The fuzzer samples inputs from the input space, executes the SUT with these inputs, and observes the outputs. In other words, looking again at the Algorithm 1, the feedback contains just the output of the execution and no other explicit information. While it could seem that black-box is rather limited, it

is actually really powerful and widely adopted, as it can be rapidly employed to a wide range of targets.

In fact, the term *black-box* does not imply that the fuzzer has no knowledge in general, but just about the internal logic of the SUT. Instead, the fuzzer can have knowledge about the input space, which could allow to generate inputs that are syntactically valid. For example, in the fuzzing of web APIs, the fuzzer can have knowledge about the API specification and infer the expected behaviour of the SUT based on the specification. Fuzzers such as RESTLER [15] can generate inputs based on the OpenAPI specification and deduce bugs from unexpected response codes – such as an Hypertext Transfer Protocol (HTTP) code 500 Internal Server Error. In addition, the fuzzer can still deduce more information from the outputs and tailor the overall fuzzing process accordingly.

White-Box Fuzzing

In *White-box Fuzzing* the fuzzer leverages full understanding of the internal logic of the SUT at runtime to guide the fuzzing process in a systematic manner. Although literature often defines white-box fuzzing as having access to the source code of the SUT [11], this alone is insufficient. In fact, without in-depth program analysis, the fuzzer cannot effectively exploit the available information. Therefore, this work considers white-box fuzzing as a technique that possesses a fully analysed representation of the internal logic of the SUT.

In essence, during the execution of the SUT, the feedback gathered by the fuzzer includes not only the output, but a comprehensive set of information such as the executed control-flow paths, the execution trace detailing how a particular program state was reached, and the logical constraints that must be satisfied to produce that state. As a result, the testcases sampled in algorithm 1 are not only syntactically and semantically valid, but also effective in reaching particular program paths or exercising specific program behaviours. Usually, such approaches employ techniques such as symbolic execution [19, 20, 21, 22] and concolic testing [22], which systematically analyse program paths and generate inputs that satisfy precise constraints.

In practice, acquiring and maintaining such exhaustive knowledge is often infeasible

due to the size, complexity, and rapid evolution of software. For example, a traditional limitation of white-box fuzzing is called *path explosion* [23], which occurs when the number of possible paths in a system grows exponentially with the number of variables. Another limitation inherently lies in the SMT solvers – which are often used to solve the constraints generated by symbolic execution – that can struggle with complex or non-linear constraints, leading to incomplete path exploration. These constraints have motivated the development of intermediate approaches that operate with partial knowledge of the SUT, giving rise to a spectrum of techniques that bridge the gap between the black-box and white-box paradigms. This motivates the introduction of the next category.

Grey-Box Fuzzing

In *Grey-box Fuzzing* the fuzzer operates with partial knowledge of the internal logic of the SUT, using lightweight instrumentation to get some form of runtime feedback to guide the fuzzing process. Unlike black-box fuzzing, which relies solely on input-output observations, grey-box fuzzing exploits limited internal information to make input generation more effective. At the same time, it avoids the high computational cost and complexity of maintaining a fully analysed program representation as in white-box fuzzing. Therefore, this work considers grey-box fuzzing as a technique that uses selectively gathered program information to improve coverage and bug-finding efficiency without requiring exhaustive program analysis.

In essence, during the execution of the SUT, the feedback gathered by the fuzzer typically consists of coverage metrics, such as the set of basic blocks or edges executed, and other coarse-grained indicators of program behaviour. This information is used to prioritise inputs that exercise new execution paths, thereby progressively expanding the explored program space. Representative approaches employ techniques such as coverage-guided fuzzing [24] and edge-coverage instrumentation [25], which balance feedback quality with execution speed to enable large-scale test generation.

2.1.2 Input-Based Taxonomy

In this taxonomy, fuzzing techniques are classified according to the input generation method, that is, the strategy employed to explore the input space. Three principal

approaches emerge: *generation-based fuzzing*, which constructs inputs from formal specifications; *mutation-based fuzzing*, which perturbs existing inputs to produce new ones; and *structure-aware fuzzing*, which integrates both paradigms to ensure that inputs are both syntactically correct and semantically valid.

Generation-Based Fuzzing

In *Generation-Based Fuzzing*, the fuzzer generates inputs from a formal specification of the input format. This specification may take the form of a grammar, a protocol description, or a model of the expected input structure, which is why the technique is often referred to as *grammar-based fuzzing*. By systematically enumerating inputs that conform to the specification, generation-based fuzzing can ensure high coverage of the valid input space and target specific program components that process structured data. Typical examples include fuzzers that use context-free grammars for file formats or protocol state machines for network inputs.

In essence, this approach enables the creation of inputs that are syntactically correct by design, reducing the likelihood of premature rejection by the SUT's parsing logic. As a result, it can be particularly effective at reaching deeper program logic that lies beyond the initial parsing stage. However, its effectiveness depends heavily on the completeness and accuracy of the specification: an incomplete or overly restrictive model may prevent exploration of certain behaviours or omission of critical edge cases. Additionally, constructing and maintaining such specifications can be labour-intensive and, in some cases, infeasible for complex or poorly documented input formats.

Mutation-Based Fuzzing

In *Mutation-Based Fuzzing*, the fuzzer generates new inputs by modifying existing ones—often referred to as *seed inputs*. These mutations may involve bit flips, byte substitutions, insertion or deletion of data, block shuffling, or more sophisticated transformations designed to produce meaningful variations. The strategy is inherently data-driven: the initial seeds are typically obtained from real-world examples, prior testcases, or corpus minimisation techniques.

The advantage of this approach lies in its ability to explore diverse and potentially unexpected behaviours of the SUT without requiring a formal specification. Moreover, by starting from valid seeds, mutation-based fuzzing often produces inputs that are close to syntactic correctness, which can increase the likelihood of triggering deeper program logic while still occasionally generating malformed inputs that may expose parsing vulnerabilities. However, its effectiveness is bounded by the diversity and quality of the seed corpus, and purely random mutations may waste computational resources on inputs that are immediately rejected by the SUT. As a result, many modern mutation-based fuzzers incorporate coverage feedback to focus mutations on promising regions of the input space.

Structure-Aware Fuzzing

Structure-Aware Fuzzing combines the strengths of generation-based and mutation-based approaches to produce inputs that are both syntactically correct and semantically valid, while retaining the adaptability of mutation-based strategies. Rather than applying blind mutations at the byte level, structure-aware fuzzers parse existing inputs into higher-level components – such as fields, tokens, or grammar elements – and then apply mutations that respect the structural constraints of the format. This hybrid approach can be realised through explicit grammars, parser-based instrumentation, or automated format inference.

In practice, structure-aware fuzzing mitigates the main shortcomings of the other two paradigms. It avoids the semantic gaps that can occur in generation-based fuzzing due to incomplete specifications, and it reduces the waste associated with mutation-based fuzzing by preventing the creation of trivially invalid inputs. Notable examples include ISLA [26], which uses input structure learning to guide mutations, and Fandango [27], which applies constraint-solving techniques to ensure semantic validity. While structure-aware fuzzing can significantly improve bug-finding efficiency, it also introduces additional complexity in the parsing and mutation stages, and its benefits depend on the quality of the structural model obtained.

2.1.3 State-Based Taxonomy

In this taxonomy, fuzzing techniques are distinguished by their consideration of system state: stateless approaches treat each testcase independently, whereas stateful methods generate sequences of inputs to explore stateful behaviours.

Stateless Fuzzing

Stateless fuzzing regards each input in isolation, without modelling or resetting the internal state of the SUT. This simplification enables high-throughput testing, as no state management is required; consequently, stateless fuzzers are well suited to self-contained targets (e.g. image decoders or stand-alone libraries). However, they cannot uncover defects that emerge only after specific sequences of operations.

Stateful Fuzzing

Stateful fuzzing explicitly models and manipulates the internal state of the SUT, constructing ordered sequences of operations that drive the system through defined state transitions. Such methods often employ state-machine inference or protocol specifications to guide input generation [28], enabling the discovery of faults that manifest only after particular workflows (e.g. login-upload-logout). While stateful fuzzers incur additional complexity in state tracking and sequence management, they are essential for interactive or protocol-driven services.

2.2 Coverage-Guided Fuzzing

Coverage-guided fuzzing is a widely adopted grey-box technique that implements an evolutionary approach: maintain a population of inputs (the *corpus*) that is evolved through mutation, retaining only inputs that discover new code coverage of the SUT. This transforms fuzzing from random input generation into directed search, where mutations that explore previously unexplored code paths are preserved for further evolution.

To implement this evolutionary approach, CGF requires several key capabilities: monitoring which parts of the SUT are executed during each testcase run, determining whether new coverage makes a testcase *interesting* enough to add to the

corpus, selecting which corpus entries should be mutated, and applying mutations to generate new testcases. Modern CGF framework, exemplified by fuzzers such as AFL [24], LibFuzzer [25], and the LibAFL library [29], implement these requirements through several interconnected components:

1) Instrumentation and Coverage Observation. The foundation of coverage-guided fuzzing lies in *instrumentation*. Coverage-guided fuzzers need to add probes to the SUT to instrument the binary and obtain runtime information. Instrumentation can be applied at different levels: source code instrumentation during compilation, binary instrumentation through static or dynamic rewriting, or hardware-assisted instrumentation using processor features. The instrumentation inserts lightweight monitoring code that tracks which parts of the program are executed without significantly impacting performance. Different types of coverage can be collected: *basic block coverage* tracks which code blocks are executed, *edge coverage* tracks transitions between basic blocks in the SUT's control flow graph, and *line coverage* tracks which source code lines are executed. In the LibAFL framework, this monitoring is handled by *Observer* components, which are responsible for collecting runtime information during target execution.

2) Coverage Feedback. The collected instrumentation data is processed to compute coverage metrics that quantify program exploration. AFL [24], for example, instruments the code to fill a bitmap that represents the edges of the code covered by the inputs. This coverage information is typically stored in a compact data structure called a *coverage map* – often implemented as a bitmap or hash table – that enables efficient comparison between different executions to identify novel program behaviours. The coverage map serves as the primary feedback mechanism, providing a quantitative measure of how much of the program's execution space has been explored. Later, the fuzzer uses this coverage map to assign a higher score to messages able to explore previously unseen program paths. In LibAFL, the processing of this raw coverage data is performed by *Feedback* components, which analyse observer data to determine whether an input is "interesting" and should be preserved in the corpus.

3) Corpus Management and Seed Selection. The fuzzing process maintains a *corpus*—a collection of inputs that have triggered interesting program behaviours during previous executions. The user must provide some input messages (seeds) representative of usual inputs for the system. Each corpus entry consists of an input along with associated metadata, including coverage information, execution time, and performance metrics. The corpus evolves throughout the fuzzing campaign as new inputs that trigger novel behaviours are discovered and added, whilst redundant inputs that do not contribute unique coverage are typically discarded to maintain efficiency. The selection of which corpus entries to use for mutation is handled by a scheduling mechanism that determines the order and priority of inputs based on factors such as recency, execution time, coverage rarity, and historical mutation success. In LibAFL, corpus management and input selection are handled by *Scheduler* components that implement various prioritisation strategies.

4) Input Generation and Mutation. The core of every fuzzer is the generation of slightly malformed input messages to forward to the software under test. A fuzzer is as efficient as the generated inputs are able to break the system. New test inputs are generated through *mutation*—the process of applying transformations to existing corpus entries to produce variations that may exercise different program behaviours. Traditional byte-level mutators implement operations such as bit flips, byte substitutions, insertions, deletions, and block swapping to explore variations around known-good inputs. More sophisticated mutation strategies employ format-aware transformations that respect input structure and semantics, such as grammar-based mutations for structured data formats. The selection and composition of mutation strategies significantly influences the fuzzer's ability to generate inputs that exercise deep program behaviours and discover complex vulnerabilities. In LibAFL, input transformation is performed by *Mutator* components, which can be combined and configured to implement various mutation strategies.

5) Target Execution and Monitoring. Each execution of the fuzzer involves several detection components. The actual execution of test inputs against the target program is managed by an execution component that handles program invocation, environment setup, timeout management, and anomaly detection. This component must balance execution speed with comprehensive monitoring, as the throughput of input execution directly influences the fuzzer's exploration rate. The execution

process involves three main detection mechanisms:

- *Bug detector*: reports eventual bugs. The majority of bug detectors only report crashes; however, for many systems, a deviation from the expected protocol flow may represent significant security issues.
- *Hang detector*: detects program execution hangs and timeouts.
- *Coverage detector*: collects the code coverage that represents the feedback the fuzzer leverages to improve the quality of input messages.

Different execution strategies offer varying trade-offs between performance and observability: in-process execution provides maximum speed by avoiding process creation overhead, whilst separate process execution offers better isolation and fault tolerance. Emulation-based execution, exemplified by approaches like QEMU-based fuzzing [30], enables fuzzing of binary-only targets or cross-architecture testing whilst maintaining coverage feedback capabilities. In LibAFL, target execution is handled by *Executor* components that can be configured for different execution environments and monitoring requirements.

The main fuzzing loop orchestrates these components in an iterative process that implements the evolutionary algorithm driving CGF. In each iteration, the scheduler selects an input from the corpus, one or more mutators generate a new testcase through transformation, and the executor runs this input against the target whilst observers collect runtime information. Feedback components then analyse this information to determine whether the execution produced novel coverage or other interesting behaviours. If so, the new input is added to the corpus for future exploration. If the execution triggers a crash or other anomalous behaviour, the input is preserved as a potential vulnerability trigger. This process continues iteratively, with the corpus gradually accumulating inputs that collectively explore an increasingly comprehensive portion of the program's execution space.

The effectiveness of CGF emerges from this evolutionary process: inputs that exercise novel program behaviours are preserved and serve as seeds for further exploration, whilst inputs that only duplicate known behaviours are discarded. Over time, this process discovers increasingly complex program states and execution paths,

often uncovering vulnerabilities that would be extremely difficult to find through manual testing or random input generation.

Algorithm 2 formalises this process by extending the basic fuzzing algorithm from Algorithm 1 with coverage feedback mechanisms. The key enhancement is the integration of the five components described above: the *Scheduler* selects promising testcases from the *Corpus*, the *Mutator* applies mutation to the testcase, the *Executor* executes the SUT injecting the testcase, the *Observer* collects runtime coverage, and *Feedback* determines whether new coverage justifies extending the corpus with the testcase.

Algorithm 2: Coverage-Guided Fuzzing Algorithm

Input: Initial seed corpus $\mathcal{C}$, coverage map $\mathcal{M}$
Output: testcases leading to bugs $\mathcal{B}$, updated corpus $\mathcal{C}$

```
B ← ∅ ;                                          // initialise bug collection
repeat
    testcase ← Scheduler(C) ;                        // select from corpus
    mutated_input ← Mutator(testcase) ;               // apply mutations
    feedback ← Executor(mutated_input) ;        // run with instrumentation
    coverage ← Observer(feedback) ;                // extract coverage info
    if Objective(feedback) ;                // check for crashes/anomalies
     then
        B ← B ∪ {mutated_input} ;                     // preserve solution
    if Feedback(coverage, M) ;                         // new coverage found
     then
        C ← C ∪ {mutated_input} ;                          // add to corpus
        M ← update(M, coverage) ;                    // update coverage map
until ;                                       // coverage-guided fuzzing loop
;
return B, C
```

This chapter explored the fundamentals of fuzzing techniques and CGF, establishing the taxonomies and algorithmic foundations. The following chapter examines the state-of-the-art and related works in CGF for complex systems, identifying how existing approaches handle synchronisation challenges and revealing the gaps that motivate the multiprocess fuzzing contributions of this dissertation.

Chapter 3

State of the Art

Coverage-Guided Fuzzing of Software Systems employing multiple processes is not explicitly addressed in the existing literature. That is because current fuzzing research predominantly focuses on single-process targets, leaving multiprocess architectures inadequately tested. However, examining related fuzzing domains reveals valuable insights and technical solutions that can inform fuzzing of multiprocess systems.

This chapter surveys state-of-the-art fuzzing techniques across domains that inherently involve multiple execution contexts or face similar coordination challenges. The analysis identifies common design patterns, recurring trade-offs, and transferable techniques that address aspects useful for this dissertation's focus.

The surveyed domains include: **Network Fuzzing** (Section 3.1) provides essential insights as network servers typically employ multiprocess architectures with worker processes handling concurrent connections, requiring techniques for process lifecycle management and socket-based IPC feeding.

Hypervisor-based Fuzzing (Section 3.2) demonstrates comprehensive system instrumentation approaches that monitor entire execution environments, including multiple processes, kernel interactions, and hardware devices within virtualised contexts.

Kernel Fuzzing (Section 3.3) addresses systematic testing of complex interface systems where system calls coordinate multiple kernel processes and user-space interactions, requiring sophisticated resource management and crash detection across execution boundaries.

Concurrent Fuzzing (Section 3.4) directly tackles process coordination challenges through schedule exploration and timing-dependent interaction testing, providing techniques for handling non-deterministic multiprocess execution.

Embedded Fuzzing (Section 3.5) explores testing complete firmware systems where multiple services coordinate through constrained IPC mechanisms, demonstrating approaches for preserving realistic multiprocess execution contexts.

3.1 Network Fuzzing

Networked applications such as web servers expose a network socket as their primary entrypoint. This design motivated fuzzing techniques that feed testcases directly over network interfaces rather than via file inputs. A common but impractical solution that fuzzing practitioners adopted is to redirect the socket entrypoint through manual – source code or binary – patching.

Another simpler approach is to intercept socket function calls and, through dynamic linking, redirects them to alternative behaviours (for example, substituting the socket file descriptor with stdin), thereby simplifying testing. Notable examples of this approach include the *desock* [31] module or its extension *desock+* introduced by GreenFuzz [32], which expanded socket interception to cover additional use cases and socket types. A more sophisticated dynamic linking technique is proposed by Netfuzzlib [33], which emulates network I/O entirely in user space by intercepting socket operations through library preloading, ultimately replacing actual network communication with synchronised user-space emulation.

More recently, fuzzers that feed testcases directly to the actual socket emerged to avoid invasive SUT's modifications and preserve realistic execution contexts. Both AFLnwe [34], an extension of AFL, and Honggfuzz's *netdriver* module implemented this approach by providing a network socket driver that feeds testcases over TCP connections. These tools usually employ a fork-server execution model, forking a fresh SUT instance for each testcase to ensure clean state. However, a major challenge arises from the need to maintain state across multiple testcases like in a protocol session. In fact, traditional fuzzing techniques often treat each testcase in

isolation, but this approach fails to account for the complex interactions that can occur in real-world network protocols.

The challenge of maintaining protocol state across message sequences drove the development of *stateful fuzzing* techniques [28, 35]. Indeed, network protocols require specific message orderings – where the term 'message' is interchangeable with 'input' but tailored in the context of protocol testing – to reach different protocol states. For example, authentication must precede file operations in File Transfer Protocol (FTP), handshakes must complete before data transfer in protocols like Real-Time Streaming Protocol (RTSP). As a consequence, simply mutating individual messages produces sequences that servers immediately reject, limiting exploration effectiveness.

AFLNet [36] addressed this challenge by extending AFL to infer protocol states through server response codes. The fuzzer acts as a client, sending message sequences to target servers while learning protocol state machines from responses status codes, widely adopted in protocols like FTP, Simple Mail Transfer Protocol (SMTP), and HTTP. This state feedback guides exploration towards unexplored protocol states through sequence mutation that preserves state-reaching prefixes.

While AFLNet demonstrated substantial improvements over stateless approaches, discovering multiple CVEs in production protocol implementations, AFLNet requires protocol-specific response parsers, manual time delays for network synchronisation, and, optionally, custom cleanup scripts for server state reset between sessions. A practical experience of adopting AFLNet to a new protocol implementation – *OPC UA* – is presented in B.

As AFLNet relies on seed quality, it may struggle to explore protocol states if seeds does not include specific protocol interactions. ChatAFL [37] addresses this limitation with an extension of AFLNet that employs LLMs. In brief, the approach employs a LLM to extract protocol grammars, enrich seed, and generate testcases to enhance the corpus to break out of coverage plateaus.

Both approaches struggle with protocols lacking explicit status codes or where response codes reflect command outcomes rather than true protocol state. To cope

with these limitations, StateAFL [38] eliminated the need for response code parsing by inferring protocol states from server memory contents. In brief, through compile-time instrumentation, StateAFL monitors memory allocations and captures snapshots of long-lived data structures that persist across request-reply cycles. In this way, memory snapshots are mapped to unique state identifiers, clustering similar memory states while distinguishing meaningful protocol transitions.

Another solution could be to provide manual state annotations like IJON [39] introduced. However, manual annotation requires manual effort and domain expertise to identify relevant program states. SGFuzz [40] eliminated the need for manual annotations by automatically identifying state variables through enumerated types and named constants. The approach adopted LibFuzzer's in-process execution model, running multiple testcases within a single process lifetime (sometimes referred to as *persistent mode*). Sequences of values assigned to state variables are tracked to guide exploration towards unexplored state sequences without requiring protocol-specific parsers or manual annotations.

SnapFuzz [41] took a fundamentally different approach, achieving dramatic performance improvements through comprehensive system call interception. Built on SaBRe [42] load-time binary rewriting, SnapFuzz intercepts all system calls and library functions, redirecting them to custom implementations optimised for fuzzing.

The approach introduces a *smart deferred fork-server* that automatically places the initialisation point as late as possible, after all necessary setup and configuration. SnapFuzz partially addresses C1 and C2 through group-based signal propagation that monitors child process termination and ensures crashes are detected across process boundaries. As shown in Snippet 3.1, when the computation is done, the signal propagation mechanism actively monitors all child processes through *wait()* system calls, checking exit status and propagating crash signals (*SIGSEGV*, *SIGILL*) to the fork-server for proper crash detection.

```
657 ... else if (cs == Done) {
658   // TODO: Emulate SIGTERM
659   sigset_t signal_set;
660   sigemptyset(&signal_set);
661   sigaddset(&signal_set, SIGTERM);
662   sigprocmask(SIG_BLOCK, &signal_set, NULL);
```

```
663
664   pid_t wpid;
665   int status = 0;
666   do {
667     wpid = wait(&status);
668     if (wpid <= 0)
669       continue;
670
671     if (WIFSIGNALED(status)) {
672       if (WTERMSIG(status) == SIGSEGV || WTERMSIG(status) == SIGILL
              ) {
673         raise(WTERMSIG(status));
674       }
675     }
676   } while (wpid > 0);
677
678   real_syscall(SYS_exit_group, 0, 0, 0, 0, 0, 0);
679 } ...
```

Listing 3.1: Excerpt from SnapFuzz source code for signal propagation in child processes[1].

This design enables SnapFuzz to automatically detect crashes in child processes that would otherwise be missed by other fuzzers, as will be discussed in 6, addressing a critical limitation in multiprocess bug detection.

Alternative approaches do not focus on message mutations but rather on message sequence reordering or manipulation. Bleem [43] operates as a Man-In-The-Middle (MITM) proxy, forwarding messages between client and server whilst generating packet sequences based on observed protocol state transitions. Fuzztruction-Net [44] requires both client and server to run simultaneously, with runtime fault injection applied to one peer whilst the other serves as the SUT. Faults modify load and store instructions, conditional branches, or function calls in the peer, with the aim of generating corrupted messages that remain cryptographically valid since encryption occurs after fault injection. This enables testing either client-side or server-side implementations using encrypted protocols.

[1] https://github.com/srg-imperial/SnapFuzz/blob/main/snapfuzz/main.c

LibAFLstar [45] adopts a different strategy, using persistent mode execution with manual patching to feed inputs through sockets whilst leveraging prior knowledge of protocol state models. The key insight is that explicit state model knowledge enables systematic exploration of protocol states, achieving comprehensive coverage through guided state transitions rather than inferring states from responses. Further details are presented in Appendix B.

While multiprocess software systems are not explicitly referenced in the literature, the *Network Fuzzing* domain inherently involves multiple processes. However, the multiprocess aspects are not addressed directly or are simply bypassed. In fact, most of the time, target systems are configured to run as single-process applications, as witnessed in the ProFuzzBench's source code [46], a benchmark framework for stateful network fuzzers. This configuration simplifies testing but limits realism since many production servers employ multiprocess architectures for scalability and fault isolation.

The state of the art though offer an interesting perspective on how to handle the execution synchronisation and bug detection challenges (`C1` and `C2`) and the input feeding into IPC mechanisms such as sockets (`C4`). As a consequence, Table 3.1 summarises the execution and socket management techniques which can be tailored to multiprocess fuzzing. In fact, Forkfuzz, presented in 6, employs Honggfuzz's netdriver for socket management.

`Lessons Learned` Network fuzzing reveals key trade-offs for system-level fuzzing: direct socket feeding is straightforward but introduces network overhead, while system call interception eliminates this overhead at implementation complexity cost. For execution management, fork-server models provide simplicity but require expensive SUT restarts, whereas persistent mode achieves performance gains at the cost of careful execution lifecycle management.

3.2 Hypervisor-based Fuzzing

Hypervisor-based fuzzing leverages instrumented Virtual Machines (VMs) through comprehensive execution management capabilities of hypervisors. This technique is

Table 3.1: Network fuzzing approaches categorised by execution strategy and input feeding mechanism.

Tool	Execution Management	Socket Management
DESOCK (library)	N/A	Dynamic Linking
GREENFUZZ	Fork-server	Socket redirection
NETFUZZLIB (library)	N/A	Socket redirection
AFLNWE	Fork-server	TCP/UDP Socket
HONGGFUZZ (netdriver)	Fork-server	TCP/UDP Socket
AFLNET	Fork-server	TCP/UDP Socket
CHATAFL	Fork-server	TCP/UDP Socket
STATEAFL	Fork-server	TCP/UDP Socket
IJON	Fork-server	TCP/UDP Socket
SGFUZZ	Persistent mode	TCP/UDP Socket
SNAPFUZZ	(Deferred) Fork-server	System call interception
BLEEM	MITM proxy	Packet forwarding
FUZZTRUCTION-NET	MITM proxy	Peer fault injection
LIBAFLSTAR	Persistent mode	TCP/UDP Socket

particularly relevant for testing complex software stacks where vulnerabilities emerge from interactions between system components.

A prominent hypervisor used for fuzzing is QEMU [47], which was initially developed as a full-system emulator to run software across different CPU architectures, and was later extended to support hardware-assisted virtualisation on the same architecture. Consequently, QEMU provides two distinct execution models that serve different fuzzing requirements. Kernel-based Virtual Machine (KVM) leverages hardware-assisted virtualisation to run guest code natively with near-native performance, but it offers limited opportunities for fine-grained instrumentation and requires specific hardware. In contrast, the Tiny Code Generator (TCG) engine performs software-based instruction translation at runtime, enabling detailed monitoring and instrumentation – such as syscall hooking – at the cost of execution speed. QEMU is widely adopted in fuzzing research, in particular, for binary emulation of embedded systems firmware. However, this will be discussed in detail in Section 3.5, in this paragraph we focus on how hypervisor-based are used for complex software stacks, not for binary emulation.

NYX [48] exemplifies the KVM-based approach, targeting hypervisors and achieving

high throughput through hardware acceleration. Nyx-Net [49] extends this approach to network protocol fuzzing, acknowledging multiprocess scenarios through VM-level state management. The approach recognises that network servers commonly employ multiprocess architectures, noting that "Even common, complex patterns such as forking a new process for each incoming connection, writing incoming data to a file system, or even a database in another process, are correctly handled" through complete VM snapshot restoration.

However, Nyx-Net provides only coarse-grained support for multiprocess scenarios. The VM-level snapshots restore all system state wholesale – including process hierarchies, shared memory, and file system changes – but without fine-grained instrumentation of individual processes or IPC mechanisms. Coverage collection relies on either Intel Processor Trace (PT) or AFL's compile-time instrumentation, but the paper does not specify how coverage is aggregated across multiple processes within the VM. Additionally, the KVM-based approach requires specialised hardware (Intel VT-x or AMD SVM), making extension and practical adoption challenging. While Nyx-Net demonstrates substantial empirical effectiveness with 10x-100x performance improvements over AFLNet, the hypervisor-based execution environment prevents the fine-grained process lifecycle control and IPC-specific instrumentation necessary for comprehensive multiprocess fuzzing automation. Furthermore, despite being open-source, Nyx-Net remains poorly documented and impractical for widespread adoption in multiprocess fuzzing research.

Contrarily, LibAFL QEMU [30] exemplifies the TCG-based approach, leveraging software-based instruction translation for comprehensive instrumentation. As an extension of the LibAFL [29] fuzzing library, LibAFL QEMU trades execution speed for detailed observability across multiple processes and system boundaries. The architecture includes three abstraction layers: low-level API for direct QEMU interaction, high-level API for hook management, and fuzzing-oriented instrumentation helpers including coverage tracking and AddressSanitizer. For execution control, the framework supports breakpoints for predetermined addresses, sync backdoors for dynamic address resolution, and monitor commands for debugger integration. The approach enables precise tracking of system calls, memory operations, and inter-process communication through comprehensive hooking mechanisms that monitor translation blocks during execution.

FitM [50] demonstrates an alternative approach combining hypervisor-based execution with userspace snapshots for stateful network protocol fuzzing. FitM handles system calls through comprehensive syscall interception and redirection for network protocol fuzzing. The tool patches Quick Emulator (QEMU)'s syscall translation layer to replace network operations with a special pseudo-file descriptor (`FITM_FD`), enabling high-speed input delivery through shared maps instead of actual sockets. For process management, FitM handles `fork()` and `clone()` calls by disabling all forking after the first IP socket interaction begins, ensuring only one execution path is followed (either parent or child, configurable). During `recv()` syscalls, FitM triggers CRIU snapshots via RPC when the target has previously sent data, creating restore points for stateful protocol exploration. The syscall hooks cover essential networking primitives (`socket`, `bind`, `connect`, `accept`, `send`, `recv`) as well as asynchronous monitoring functions (`select`, `poll`, `epoll`) that are modified to always report activity on `FITM_FD`. Process isolation is achieved through Linux namespaces, where each restored snapshot runs in its own PID and mount namespace to avoid conflicts. This systematic syscall management enables FitM to transparently instrument client-server interactions while maintaining execution determinism through userspace snapshots for stateful network protocol fuzzing.

The hypervisor-based approach demonstrates substantial empirical effectiveness, with Nyx discovering 44 vulnerabilities in production hypervisors and 22 CVEs assigned. However, the requirement for complete environmental control restricts applicability to targets executing within isolated virtual machine contexts.

> **`Lessons Learned`** Hypervisor-based fuzzing reveals critical capabilities for multi-process testing: systematic syscall interception enables comprehensive monitoring of process coordination activities – including fork(), exec(), and IPC operations – without requiring manual instrumentation of target applications. Additionally, VM-level snapshot restoration provides deterministic reset capabilities that can restore complex multiprocess states, including process hierarchies, shared memory regions, and file system changes, enabling systematic exploration of process coordination patterns whilst maintaining execution determinism across fuzzing iterations.

3.3 Kernel Fuzzing

Operating system kernels represent critical system-level targets where vulnerabilities can compromise entire system security. Kernel fuzzing requires careful execution management to handle the system call interface between user-space applications and kernel operations.

Syzkaller [51] employs a two-component architecture where `syz-manager` orchestrates VM running instances of the SUT – the compiled kernel – and generates system call sequences (the testcases), while `syz-executor` runs within each VM to execute these testcases and report coverage feedback. In particular, the approach uses a declarative description language to model system call semantics, argument types, and resource dependencies, enabling systematic generation of valid system call sequences through template-based approaches. Execution management relies on isolated QEMU VMs with instrumented kernels where each testcase runs in a dedicated instance, preventing system corruption whilst enabling comprehensive coverage collection across kernel and user space boundaries.

However, Syzkaller's seed generation relies on manually crafted rules that may not capture complex inter-call dependencies, limiting exploration effectiveness. MoonShine [52] addresses this limitation by distilling system call traces from real-world programs whilst preserving inter-call dependencies. The approach uses static analysis to detect dependencies between system calls, automatically generating seed sequences that respect kernel state requirements, improving coverage by 13% over Syzkaller's hand-coded seed generation rules.

kAFL [12] employs hypervisor-based execution (QEMU KVM) with hardware-assisted coverage collection using Intel Processor Trace (PT). The approach runs each testcase within an isolated VM, preventing system corruption when crashes occur in kernel code. Testcases – consisting of system call sequences – are fed through a small user-space component while the hypervisor monitors kernel execution. However, kAFL faces reproducibility challenges due to operating systems non-determinism caused by interrupts and threads. This is addressed by filtering interrupt transitions and blacklisting non-deterministic basic blocks, reducing coverage precision.

`Lessons Learned` Kernel fuzzing reveals fundamental reproducibility challenges in complex software systems due to non-deterministic execution. Consistent execution requires isolation and careful handling of non-deterministic elements like interrupts and timing. These challenges intensify in multiprocess systems where IPC between multiple processes introduces additional sources of non-determinism that must be managed to ensure reliable testing results.

3.4 Concurrent Fuzzing

Concurrent systems introduce non-determinism through thread scheduling and coordination, creating timing-dependent failures that traditional fuzzing cannot detect. The state-of-the-art focuses on multi-threaded software, presenting similar challenges to multiprocess systems.

MUZZ [53] addresses concurrency bugs through thread-aware instrumentation that stresses thread interleavings affecting execution states. The approach uses coverage-oriented, thread-context, and schedule-intervention instrumentation to systematically explore different thread execution orders. MUZZ detected 8 new concurrency vulnerabilities and 19 bugs, outperforming traditional fuzzers in multithreading scenarios.

Other approaches include CONFUZZ [54] for event-driven programs and recent advances in context-sensitive concurrency fuzzing [55] and greybox fuzzing with schedule mutation [56].

`Lessons Learned` Concurrent fuzzing demonstrates that coverage feedback must distinguish execution by different threads and that execution order matters for detecting timing-dependent vulnerabilities in multiprocess systems.

3.5 Embedded Fuzzing

Embedded fuzzing approaches can be categorised into hardware-based testing (direct execution on physical devices), emulation-based approaches (virtual execution environments), and abstraction-based methods (behavioural modelling) [57]. Emulation-based approaches prove most relevant for multiprocess testing, enabling comprehensive monitoring of process coordination patterns.

FIRMULTI FUZZER [58] tackles multiprocess vulnerabilities in IoT firmware through full-system emulation with system-level monitoring. The approach monitors process creation behaviour and tracks execution status by triggering callbacks when any process enters a new basic block, enabling detection of exceptions that occur in any process within the emulated system rather than just the target process.

HOUSEFUZZ [59] considers the multiprocess nature of firmware services through comprehensive service identification and multiprocess fuzzing framework. The approach identifies network-facing and daemon processes overlooked by existing tools, enabling comprehensive inspection of firmware services across multiple processes.

LIBAFL QEMU [30] provides a flexible emulation framework wrapping QEMU for fuzzing-oriented emulation. The library supports both user-mode and system emulation with comprehensive instrumentation capabilities, addressing maintainability challenges of existing QEMU forks used in fuzzing.

These approaches demonstrate that emulation-based testing effectively detects multiprocess vulnerabilities that manifest through process interactions and coordination patterns in constrained embedded environments.

`Lessons Learned` Embedded fuzzing demonstrates that emulation provides an isolated environment where it is possible to track the underlying system and syscalls, enabling comprehensive multiprocess monitoring.

3.6 Summary

The analysis of the state-of-the-art reveals three core technical insights that can be transferred to multiprocess fuzzing:

System call interception and hooking enables monitoring of process coordination activities. LibAFL QEMU provides a framework for systematic observation of `fork()`, `exec()`, and IPC mechanisms.

Hypervisor-based execution environments provide practical control during execution. Hypervisors like QEMU enable comprehensive system monitoring while isolating execution contexts, allowing observation of the whole software system including multiprocess interactions.

Coverage aggregation across execution contexts addresses collecting feedback from multiple sources. As demonstrated in concurrent fuzzing, coverage feedback must distinguish execution by different processes – similar to thread-context instrumentation – but effective strategies for multiprocess coverage aggregation remain needed.

Chapter 4

Motivation and Gaps in Multiprocess Fuzzing

The previous chapters described the fundamentals of fuzzing (§2) and established the system-level fuzzing state-of-the-art with a particular focus on concurrent and distributed fuzzing (§3).

Yet *multiprocess software* remains significantly under-represented in contemporary fuzzing research. While system-level fuzzing research has advanced considerably, as demonstrated in Chapter 3, explicit treatment of multiprocess challenges remains limited. Most system-level fuzzing approaches handle multiprocess execution as an incidental complexity rather than a fundamental research focus. This gap is particularly significant given that modern servers, browsers, and CLI utilities routinely employ system calls such as `fork()`, or equivalent primitives to isolate work, improve reliability, and leverage the efficiency given by multi-core hardware.

Multiprocess software, while a common design pattern, poses unique challenges for fuzzing. As established in Chapter 2, four core challenges emerge when SUT execution spreads across cooperating processes. This chapter explores each challenge in depth, examining their technical implications and real-world manifestations before demonstrating their practical impact through empirical evidence.

4.1 Challenge C1: Multiprocess Execution Synchronisation

Traditional CGF operates within a well-defined *fuzzing loop*: generate a testcase, execute the SUT, collect feedback, and iterate. This loop assumes clear execution boundaries where the fuzzer maintains control over the SUT's lifecycle. However, when programs create child processes through `fork()`, `vfork()`, or `clone()`, these fundamental assumptions break down catastrophically.

The synchronisation challenge manifests in several ways. First, process lifecycles become unclear as multiple processes may execute concurrently with different termination conditions. A parent process might terminate while children continue running, or children might outlive their parents, creating zombie processes that consume system resources. Second, the fuzzer loses visibility into when a fuzzing iteration truly completes, as execution spreads across an arbitrary number of processes within the process tree. This leads to race conditions where the next iteration begins before the previous one terminates, causing resource exhaustion and corrupted coverage feedback.

The implications for fuzzing effectiveness are severe. Without proper synchronisation, fuzzers cannot establish clean execution boundaries, leading to cascading failures where subsequent iterations interfere with ongoing execution. Resource management becomes problematic as orphaned processes accumulate, eventually exhausting system resources and causing fuzzing campaigns to stall.

4.2 Challenge C2: Comprehensive Test Oracle Design

Traditional test oracles in single-process fuzzing rely on signals (`SIGSEGV`, `SIGABRT`), sanitiser instrumentation, or exit codes to detect anomalous behaviour. These mechanisms assume that the SUT executes within a single observable process where monitoring instrumentation can capture all relevant events.

In multiprocess environments, this assumption fails. Child processes execute in independent address spaces where crashes, hangs, or other anomalous behaviours may remain invisible to the parent process and, consequently, to the fuzzer. Parent

processes may even capture and suppress signals from children, further obscuring failure modes. Similarly, signal handlers configured in the parent process cannot observe signals delivered to child processes unless explicitly forwarded.

The practical implications are that fuzzers operating on multiprocess software systematically miss crashes and other security-relevant behaviours. A memory corruption vulnerability that manifests only when specific code paths are executed in child processes will remain undetected, creating a false sense of security. This observation bias significantly undermines the reliability of fuzzing campaigns targeting multiprocess software.

4.3 Challenge C3: Multiprocess Coverage Observation

Coverage-guided fuzzing relies on runtime feedback to direct input generation towards unexplored code paths. This feedback is typically collected through compile-time instrumentation that maintains a coverage bitmap, recording which edges in the control-flow graph have been executed during a fuzzing iteration.

Multiprocess execution fundamentally disrupts this feedback mechanism. Coverage information becomes scattered across multiple independent processes, leading to two possible scenarios. In the first one, the coverage bitmap is shared among processes, but concurrent updates lead to contention and inconsistent feedback. In the second one, each process maintains its own coverage bitmap, but the fuzzer only observes the parent process's bitmap, receiving incomplete feedback. This leads to systematic underestimation of edge novelty, causing the fuzzer to discard inputs that actually exercise new execution paths in child processes.

The implications extend beyond simple coverage underestimation. Incomplete feedback guidance causes the fuzzer's input prioritisation mechanisms to make suboptimal scheduling decisions, potentially missing entire classes of vulnerabilities that manifest only in specific multiprocess execution scenarios.

4.4 Challenge C4: Multi-Input Generation

Multiprocess software typically exposes multiple input vectors through diverse IPC mechanisms including pipes, Unix domain sockets, shared memory regions, message queues, and network sockets. Each input source requires specific input formats and presents unique challenges for effective fuzzing.

Traditional fuzzing approaches focus on individual input vectors, typically files, command-line arguments or the standard input stream. This approach inadequately addresses the complexity of multiprocess systems where multiple processes may simultaneously consume inputs from different sources with distinct format requirements. For example, a web server might accept HTTP requests on network sockets while simultaneously processing configuration updates through Unix domain sockets and receiving signals from management processes.

The challenge extends beyond simple format diversity. Effective multiprocess fuzzing requires coordination of inputs across multiple channels to achieve meaningful system coverage. Sending an HTTP request to a web server worker process may only exercise superficial code paths unless accompanied by appropriate configuration or control inputs that trigger deeper system behaviours.

Additionally, the temporal aspects of multi-input generation present coordination and reproducibility challenges. Input sequences must be carefully orchestrated to respect the IPC semantics and timing requirements of the target system. A premature input may be ignored or cause the system to enter an error state, while delayed inputs may miss critical execution windows.

These challenges collectively define the fundamental obstacles to achieving automation in CGF of multiprocess software.

4.5 Empirical Study: Multiprocess Software in OSS-Fuzz Projects

To quantify the prevalence of multiprocess challenges in practice, this section presents an empirical study of the OSS-Fuzz corpus. OSS-Fuzz is Google's continuous

fuzzing infrastructure for open-source projects, hosting over 1000 actively maintained projects across diverse domains. While OSS-Fuzz does not perform system-level fuzzing but rather targets individual library APIs and components, it represents an excellent corpus of projects that are continuously fuzzed, providing insights into the multiprocess characteristics of real-world software. Fuzz Introspector complements OSS-Fuzz by providing static analysis and coverage reports for each project, enabling systematic examination of multiprocess features.

The motivation for this study is to establish empirical evidence for the practical importance of the four core challenges outlined earlier. If multiprocess software is rare among actively-fuzzed projects, the theoretical challenges may have limited real-world impact. Conversely, if multiprocess patterns are prevalent in this representative corpus, this strengthens the case for developing specialised multiprocess fuzzing techniques.

4.5.1 Research Questions

RQ1 *How many OSS-Fuzz C/C++ projects actively use process creation system calls (`fork()`, `clone()`, `execve()`, etc.) or IPC mechanisms?*

RQ2 *Which IPC primitives (sockets, System V shared memory, POSIX shared memory) appear in those projects and at what frequency?*

RQ3 *Does the presence of IPC mechanisms correlate with lower fuzzing coverage as reported by Fuzz Introspector?*

4.5.2 Project Selection

The OSS-Fuzz corpus was filtered using sequential criteria applied to 1,078 projects indexed by Fuzz Introspector:

Programming Language. Only projects whose build system declares at least one target in C or C++ were considered, yielding 308 candidates.

Build and Coverage Health. The latest Fuzz Introspector report for each candidate had to indicate a `SUCCESS` build status and non-zero

Table 4.1: Static-analysis patterns used to identify IPC usage in C/C++ sources. All patterns required word boundary checks (`(?<![a-zA-Z0-9_])`) and appeared in non-comment code.

IPC Category	System Calls and
Process Creation	`fork()`, `vfork()`, `clone()`, `clone3()`, `execve()`, `execv()`, `execl()`,
Socket IPC	`socket()`, `bind()`, `listen()`, `accept()`, `connect()`, `send`
System V Shared Memory	`shmget()`, `shmat()`, `shmdt()`, `shmc`
POSIX Shared Memory	`shm_open()`, `shm_unlink()`, `mmap(`

line coverage percentage. Projects flagged as examples or vulnerable test cases were excluded.

Final Dataset. After filtering, 296 projects with valid coverage data (ranging from 0.4% to 97.6%) were retained for analysis.

4.5.3 Static Analysis Pipeline

The analysis pipeline combined Fuzz Introspector's REST API with custom static analysis scripts to identify multiprocess features in the selected projects. The pipeline comprised the following steps:

1. For every repository passing the selection gate, a shallow Git clone (depth=1) was performed with a 60-second timeout. Source files with canonical C/C++ extensions (`.c`, `.cc`, `.cpp`, `.cxx`, `.c++`, `.h`, `.hpp`, `.hxx`, `.h++`, `.hh`) were parsed. Both block (`/* */`) and line (`//`) comments were stripped to eliminate spurious matches.

2. If `src/` or `source/` directories existed, analysis was restricted to those paths; otherwise, root-level test directories were excluded. The script employed 10 parallel workers for efficient processing.

3. Each file was scanned for 37 distinct patterns across four IPC categories (Table 4.1), using case-insensitive regex matching with word boundary checks to prevent false positives from partial identifiers.

4.5.4 Statistical Tests

To explore whether multiprocess design influences fuzzing efficacy, three non-parametric analyses were conducted:

a) **Process Creation**: Coverage distributions for projects with versus without process creation calls.

b) **Combined IPC**: Coverage distributions for projects with high versus low total IPC usage across all categories.

c) **IPC Density**: Coverage distributions normalized by codebase size.

Given the heterogeneous variances and non-normal distributions of coverage data, the two-sample Kolmogorov-Smirnov test was adopted alongside Mann-Whitney U tests and Spearman correlations. Null hypotheses (H_0) state that the compared samples stem from the same continuous distribution; a significance threshold of $\alpha = 0.05$ was used throughout.

4.5.5 Threats to Validity

The principal limitation of this study is that OSS-FUZZ's fuzz drivers are manually written and may have low code coverage independently from the SUT's multiprocess nature. As a consequence, some projects could be more covered than others due to better driver quality rather than inherent multiprocess complexity. Additionally, the static analysis may miss IPC patterns in macro-expanded code or dynamically loaded libraries.

4.5.6 Results

The analysis encompassed 286 C/C++ projects from OSS-Fuzz with valid coverage data (mean coverage: 47.06%, range: 0.4%–97.6%). Projects using IPC mechanisms demonstrated significantly lower code coverage across all metrics.

Process Creation Analysis: Projects with process creation calls (n=109) achieved mean coverage of 34.44% compared to 54.83% for projects without (n=177). The Kolmogorov-Smirnov test strongly rejected the null hypothesis (D=0.401, p=2.99e-10), with a large effect size (Cohen's d=-0.82).

Combined IPC Analysis: High-IPC projects (n=140) showed even greater disparity, with 33.82% mean coverage versus 59.75% for low-IPC projects (n=146). The distribution difference was highly significant (D=0.495, p=1.79e-16) with the largest effect size observed (d=-1.10).

Correlation Analysis: All Spearman correlations between IPC usage and coverage were negative and statistically significant: process creation (r=-0.37, p=1.24e-10), combined IPC (r=-0.50, p=3.71e-19), and IPC density (r=-0.43, p=2.61e-14).

These findings strongly support the hypothesis that multiprocess and IPC-heavy projects achieve substantially lower coverage under OSS-Fuzz's API-focused fuzzing approach, with mean coverage differences exceeding 20 percentage points across all analyses.

4.6 Motivating Example: Lighttpd Web Server

To illustrate how the four core challenges manifest in real-world software, LIGHTTPD is examined as an example of a production web server that exemplifies the complexities of multiprocess fuzzing. LIGHTTPD is designed as a single-threaded, single-process, event-based web server optimised for high-performance scenarios. However, it supports multiprocess operation through the `server.max-worker` configuration directive, making it an ideal case study for understanding multiprocess fuzzing challenges.

4.6.1 Lighttpd Architecture Overview

By default, LIGHTTPD operates as a single process handling all incoming HTTP requests through an event-driven architecture. When `server.max-worker` is configured to a value greater than zero, LIGHTTPD creates multiple independent worker processes that share listening sockets but operate with separate memory spaces. The documentation strongly recommends `server.max-worker = 0` unless multiprocess operation provides measurable performance benefits, acknowledging the inherent complexity introduced by this architectural choice.

Each worker process operates independently, handling HTTP requests, processing configuration updates, and managing client connections. Workers communicate

through shared listening sockets and, in some configurations, through Unix domain sockets for coordination. This architecture creates multiple input vectors: HTTP requests on network sockets, configuration file updates, and inter-worker coordination messages.

4.6.2 Challenge C1: Execution Synchronisation in Lighttpd

When `server.max-worker > 0`, LIGHTTPD faces immediate execution synchronisation challenges. The main process forks worker processes during startup, but the lifecycle management becomes complex when workers crash or become unresponsive. A traditional fuzzer testing LIGHTTPD would send HTTP requests and expect clear execution boundaries. However, with multiple workers, a malformed HTTP request might cause one worker to crash while others continue processing subsequent requests.

The fuzzer loses synchronisation because it cannot determine when all workers have finished processing a given input. A worker might be processing a complex HTTP request while the fuzzer assumes the iteration is complete and begins the next one. This leads to overlapping executions where coverage feedback becomes unreliable and resource utilisation grows unbounded.

Consider a fuzzing scenario where malformed HTTP headers cause a worker to enter an infinite loop. Without proper synchronisation, the fuzzer continues generating new inputs, spawning additional workers, eventually exhausting system resources. The fuzzing campaign stalls not due to lack of interesting inputs, but due to inadequate execution lifecycle management.

4.6.3 Challenge C2: Oracle Design in Lighttpd

Traditional HTTP server fuzzing relies on observing the main process for crashes or anomalous behaviour. However, in multiprocess LIGHTTPD, crashes frequently occur in worker processes that operate independently of the parent. A memory corruption vulnerability triggered by a malformed `Content-Length` header might crash a worker process, but this crash remains invisible to a fuzzer monitoring only the parent process.

The worker process crash manifests as a `SIGSEGV` signal delivered to the worker, but sanitiser instrumentation attached to the parent process cannot observe this failure. From the fuzzer's perspective, the HTTP request completed successfully because the parent process continued operating normally. The crashed worker is typically replaced by a new worker process, masking the security-relevant behaviour from the test oracle.

This bias is particularly relevant for LIGHTTPD because many security-critical code paths – such as HTTP parsing, request validation, and response generation – execute within worker processes. A comprehensive test oracle must monitor all worker processes, detect crashes across the process tree, and correlate failures with the specific inputs that triggered them.

4.6.4 Challenge C3: Coverage Observation in Lighttpd

Coverage instrumentation in LIGHTTPD faces the classic multiprocess observation problem. When coverage instrumentation is applied during compilation, each worker process maintains its own coverage bitmap. An HTTP request that exercises novel code paths in a worker process updates only that worker's bitmap, while the fuzzer observes coverage feedback from the parent process.

Consider an HTTP request with a novel `Range` header that triggers previously unexplored execution paths in the HTTP parsing logic. This request exercises new edges in the worker process, but the coverage feedback never reaches the fuzzer because the instrumentation is isolated within the worker's address space. The fuzzer incorrectly discards this input as uninteresting, missing the opportunity to explore the newly discovered execution paths further.

The situation becomes more complex when multiple workers process similar requests concurrently. The same execution path might be executed simultaneously in different worker processes, leading to coverage bitmap races and inconsistent feedback. The fuzzer's input prioritisation mechanisms, relying on incomplete coverage information, make suboptimal scheduling decisions.

4.6.5 Challenge C4: Multi-Input Generation in Lighttpd

Lighttpd presents multiple input vectors that must be coordinated for effective system coverage. The primary input vector consists of HTTP requests on network sockets, but comprehensive testing requires additional input sources: configuration file updates, signal handling for graceful shutdown, and potential inter-worker communication through Unix domain sockets.

A naive fuzzing approach focuses solely on HTTP request fuzzing, missing critical attack vectors through configuration manipulation or signal handling. For example, a `SIGHUP` signal triggers configuration reloading, potentially exposing parsing vulnerabilities in configuration processing code. These vulnerabilities remain hidden unless the fuzzer coordinates HTTP requests with configuration updates and signal delivery.

The temporal coordination requirements are particularly challenging. A malformed configuration update followed immediately by a `SIGHUP` signal might trigger different behaviour than the same configuration update processed during normal startup. Similarly, HTTP requests sent during configuration reloading might exercise error handling code paths that are otherwise difficult to reach.

Effective Lighttpd fuzzing requires orchestrated input generation across multiple channels: HTTP requests with varying headers and body content, configuration file modifications with different syntax and semantic errors, and signal delivery with specific timing relationships. This multi-input coordination significantly exceeds the capabilities of traditional single-input fuzzing approaches.

These Lighttpd examples demonstrate that the four core challenges are not theoretical constructs but practical obstacles that significantly impact the effectiveness of multiprocess software fuzzing. The following empirical study quantifies how pervasive these challenges are across the broader ecosystem of actively-fuzzed software projects.

Chapter 5

Evaluating the Fork-Awareness of Coverage-Guided Fuzzers

Disclaimer This chapter is based on "Evaluating the Fork-Awareness of Coverage-Guided Fuzzers" [60], which introduces and systematically evaluates the concept of *fork-awareness* in CGF.

At the beginning of this research project, the first objective was to understand what gaps exist in current CGF techniques. While CGF has proven highly effective for single-process programs, empirical evidences suggested that fuzzers were not capable of effectively testing multiprocess software systems that rely on `fork()` to create child processes.

This was discovered during a preliminary experiment at the research proposal stage, where a custom harness was developed that forked a separate process to inject testcases to the target process, as it was not using stdin for input processing. What could have been a *smart move*, caused a fork bomb – an uncontrolled proliferation of processes – that quickly exhausted system resources and crashed the machine. This incident inspired the initial intuition that CGF tools were not designed to handle multiprocess programs – ultimately inspiring the concept of *fork-awareness* and the whole research direction of this dissertation.

To understand these gaps, it was first necessary to identify the requirements for fuzzing multiprocess software systems and analyse what happens during execution management when multiple processes are involved. This systematic analysis led

to the definition of *fork-awareness* as a critical property for effective multiprocess fuzzing. The final empirical study involved evaluating 14 state-of-the-art fuzzers against these requirements, revealing fundamental limitations that motivate the fork-aware approach developed in Chapter 6.

5.1 Methodology

5.1.1 Deriving Requirements from Execution Management

The first step was to systematically derive requirements from what typically occurs during SUT execution in CGF. Specifically, during each fuzzing iteration, three steps are performed:

1. **Bug Detection**: The primary goal of fuzzing is indeed finding bugs. Usually a fuzzer monitors for fatal signals, memory errors through sanitisers, or other anomalies.
2. **Hang Prevention**: To maintain high throughput, timeout mechanisms detect and terminate hanging executions.
3. **Coverage Collection**: Compile-time instrumentation tracks which code paths were exercised in a *coverage map*. Each entry in this map corresponds to a basic block or edge in the Control-Flow Graph (CFG), which is *hit* when executed.

When the SUT employs `fork()` to create child processes, these monitoring activities must extend across process boundaries. For example, timeout mechanisms are vital as worker processes could enter blocking states, causing resource exhaustion. Each child process executes in a separate address space with independent lifecycle, yet bugs, hangs, and coverage in these children are equally important for comprehensive testing.

5.1.2 Defining Fork-Awareness

From this analysis, we derive that effective multiprocess fuzzing requires monitoring all three activities across the entire process tree. This leads to the following formal

property:

Definition 5.1 (Fork-Awareness)**.** A coverage-guided fuzzer is *fork-aware* if it can detect bugs, detect hangs, and measure code coverage equivalently in both parent and child processes created through fork system calls.

This definition decomposes into three specific criteria that a fork-aware fuzzer must satisfy. It is important to note that these fork-awareness criteria (C1, C2, C3) are distinct from and should not be confused with the core challenges identified in Chapter 2. The criteria represent measurable requirements for fork-handling capabilities, while addressing aspects of the broader challenges without completely solving them:

- **[C1] Child Bug Detection**: The fuzzer must detect and report crashes, memory errors, and other anomalies occurring in child processes, not just the parent. This addresses the *Comprehensive Test Oracle Design* challenge – designing test oracles that can identify anomalous behaviour across the entire process tree.
- **[C2] Child Hang Detection**: The fuzzer must identify infinite loops or blocking operations in child processes and appropriately terminate them. This addresses *Multiprocess Execution Synchronisation* – coordinating the lifecycle of all processes to prevent resource exhaustion from zombie processes.
- **[C3] Child Coverage Tracking**: The fuzzer must collect and aggregate coverage information from child processes to guide input generation effectively. This addresses *Multiprocess Coverage Observation* – gathering runtime information from distributed processes to guide fuzzing exploration.

These criteria establish measurable requirements for evaluating fuzzer capabilities with multiprocess programs, focusing on the first three core challenges.

5.1.3 Challenge Programs

To systematically evaluate fork-awareness, three minimal C programs were developed, each targeting one specific criterion. These challenges isolate the fork-handling

behaviour from other confounding factors, enabling precise assessment of fuzzer capabilities.

Bug Detection Challenge (C1)

This program creates a child process that immediately triggers a segmentation fault:

```
if (fork() == 0) {  // Child process
    raise(SIGSEGV);  // Simulated crash
} else {             // Parent process
    wait(NULL);      // Wait for child termination
}
```

Listing 5.1: Bug detection challenge

Listing 5.1 shows a fork-aware fuzzer should detect and report the segmentation fault in the child process. This tests whether the fuzzer's crash detection mechanism extends beyond the main process.

Hang Detection Challenge (C2)

This program creates a child process that enters an infinite loop:

```
pid_t pid = fork();
if (pid == 0) {  // Child process
    while(1) { ; }  // Simulation of blocking code
}
// Parent terminates normally without waiting
```

Listing 5.2: Hang detection challenge

As demonstrated in Listing 5.2, a fork-aware fuzzer should detect the hanging child process and terminate it appropriately, preventing resource exhaustion from accumulating zombie processes.

Coverage Tracking Challenge (C3)

This program implements multiple conditional branches within the child process:

```
pid_t pid = fork();
if (pid == 0) {  // Child process
```

```
3      if (number % 2 == 0) { printf("2\n"); }
4      else { printf("!2\n"); }
5      if (number % 3 == 0) { printf("3\n"); }
6      else { printf("!3\n"); }
7      if (number % 5 == 0) { printf("5\n"); }
8      else { printf("!5\n"); }
9      if (number % 7 == 0) { printf("7\n"); }
10     else { printf("!7\n"); }
11 } else {  // Parent process
12     wait(NULL);
13 }
```

Listing 5.3: Coverage tracking challenge

Listing 5.3 demonstrates that a fork-aware fuzzer should track coverage of all branches within the child process, using this information to guide input generation towards unexplored paths.

5.1.4 Experimental Setup

Fuzzer Selection

14 coverage-guided fuzzers were evaluated, selected based on their availability in established benchmarking frameworks. The selection includes both fuzzers from FuzzBench [61] and ProFuzzBench [46]: AFL, AFL++, AFLFast, AFLSmart, Eclipser, FairFuzz, lafintel, AFLnwe, MOpt-AFL, LibFuzzer, Entropic, AFLNet, StateAFL, and Honggfuzz.

This selection encompasses both traditional mutation-based fuzzers and modern hybrid approaches, representing the current state-of-the-art in coverage-guided fuzzing.

Test Environment

All experiments were conducted on Ubuntu 20.04 systems. Each fuzzer was compiled with its default configuration and instrumentation settings. The challenge programs were compiled with the fuzzer-specific compiler wrappers to enable coverage instrumentation. Minimal seed inputs were provided and fuzzers were run with their default configurations. Network-aware fuzzers (AFLNet, StateAFL, AFLnwe)

required a different evaluation approach, as they target network protocols rather than file-based inputs. For these fuzzers, a simple HTTP server was implemented that mapped the three fork-awareness criteria to specific endpoints (`/c1`, `/c2`, `/c3`), allowing network fuzzers to trigger the same fork behaviours through HTTP requests. This server-based approach was used purely for experimental evaluation purposes to ensure consistent testing across all fuzzer types. All source code for the evaluation is freely available online[1] to ensure reproducibility.

5.2 Experiments

5.2.1 Implementation Details

Each fuzzer required specific configuration to ensure fair comparison. The evaluation revealed that different fuzzer architectures require distinct program entry points: AFL-based fuzzers and Honggfuzz work with standard instrumented binaries using `main()` functions, while LibFuzzer-based fuzzers require programs to implement the `LLVMFuzzerTestOneInput()` interface. This architectural difference necessitated implementing three versions of each challenge program to accommodate the different fuzzer requirements.

AFL-based Fuzzers

The AFL family of fuzzers (AFL, AFL++, AFLFast, AFLSmart, Eclipser, FairFuzz, lafintel, AFLnwe, AFLNet, MOpt-AFL, StateAFL) share a common architecture:

- Coverage tracking via compile-time instrumentation inserting edge coverage collection
- Bug detection through POSIX signal handlers (SIGSEGV, SIGABRT, etc.)
- Timeout-based hang detection on the main process
- Shared memory bitmap for coverage feedback

[1] `https://github.com/marcellomaugeri/forks-break-afl`

LibFuzzer-based Fuzzers

LibFuzzer and Entropic employ in-process fuzzing with sanitiser support:

- AddressSanitizer (ASAN) for memory error detection
- UndefinedBehaviorSanitizer (UBSAN) for undefined behaviour detection
- SanitizerCoverage for coverage tracking
- Single-process execution model with library linkage

Honggfuzz

Honggfuzz uses system-level monitoring:

- `ptrace` system call for process control on Linux
- Hardware and software-based coverage feedback
- Signal interception across process trees
- Dynamic process attachment capabilities

5.2.2 Execution Results

Table 5.1 summarises the evaluation results across all three challenges:

5.3 Evaluation

5.3.1 Analysis of Results

The evaluation reveals systematic limitations across all tested fuzzers:

Bug Detection (C1)

Only 3 of 14 fuzzers (LibFuzzer, Entropic, Honggfuzz) successfully detected crashes in child processes. The 11 AFL-based fuzzers failed because they rely on signal handlers attached to the main process, which do not capture signals from child processes. This limitation is explicitly documented in AFL's documentation, which

Table 5.1: Fork-awareness evaluation results for 14 coverage-guided fuzzers

Fuzzer	C1: Bug Detection	C2: Hang Detection	C3: Coverage
AFL	×	×	✓
AFL++	×	×	✓
AFLFast	×	×	✓
AFLSmart	×	×	✓
Eclipser	×	×	✓
FairFuzz	×	×	✓
lafintel	×	×	✓
AFLnwe	×	×	✓
AFLNet	×	×	✓
MOpt-AFL	×	×	✓
StateAFL	×	×	✓
LibFuzzer	✓	×	✓
Entropic	✓	×	✓
Honggfuzz	✓	×	✓

states that signals are caught "from the main process only"[2]. The technical reason is that POSIX signals are delivered to specific processes – when a child process crashes, its SIGSEGV signal is not propagated to the parent's signal handler. This represents a fundamental architectural limitation: AFL's lightweight design assumes single-process execution.

The successful fuzzers employ different strategies:

- **LibFuzzer/Entropic**: In-process fuzzing ensures sanitiser instrumentation is inherited by child processes through `fork()`, enabling error detection within each process without external monitoring
- **Honggfuzz**: `ptrace` allows monitoring entire process trees, intercepting signals from children

Hang Detection (C2)

All 14 fuzzers failed to detect hangs in child processes. This universal failure stems from timeout mechanisms that monitor only the main process. When the parent

[2] `https://github.com/google/AFL/blob/master/README.md`

terminates normally, fuzzers consider execution complete, ignoring still-running children. This leads to:

- Accumulation of zombie processes
- Resource exhaustion over extended campaigns
- Missed bugs that manifest as child process hangs

Coverage Tracking (C3)

Surprisingly, all fuzzers successfully tracked coverage in child processes. This success occurs because coverage instrumentation operates at the binary level, independent of process boundaries. The shared memory bitmap (in AFL-based fuzzers) or in-process coverage (in LibFuzzer) captures execution regardless of which process executes the code.

However, this coverage tracking has limitations:

- Coverage from short-lived children may be lost
- Race conditions in coverage updates from multiple children
- No distinction between parent and child coverage

5.3.2 Implications for Fuzzing Effectiveness

The lack of fork-awareness has profound implications for fuzzing effectiveness on real-world programs:

Missed Vulnerabilities

Bugs occurring exclusively in child processes remain undetected by most fuzzers. For network servers that handle each connection in a separate process, this means connection-handling vulnerabilities may go unnoticed.

Incomplete Coverage

While coverage is tracked, the inability to detect child process outcomes means fuzzers cannot properly reward inputs that explore child-specific paths. This creates a feedback gap that limits exploration effectiveness.

False Confidence

Fuzzers report successful completion even when child processes crash or hang. This gives users false confidence in program correctness, as the fuzzing campaign appears successful despite missing critical bugs.

5.4 Research Findings

5.4.1 Key Observations

This evaluation yields several important findings about the current state of fork-awareness in coverage-guided fuzzing:

1. **Architectural Assumptions**: Most fuzzers are designed with single-process assumptions deeply embedded in their architecture, making fork support difficult to retrofit.
2. **Monitoring Mechanisms Matter**: The choice of monitoring mechanism (signals vs. sanitisers vs. ptrace) fundamentally determines fork-handling capabilities.
3. **Coverage Is Not Enough**: While coverage tracking works across processes, it alone is insufficient for effective multiprocess fuzzing without proper crash and hang detection.
4. **No Complete Solution**: No existing fuzzer fully satisfies all fork-awareness criteria, indicating a significant gap in current fuzzing capabilities.

5.4.2 Future Directions

The findings point to several necessary developments for achieving true fork-awareness, directly corresponding to the core challenges in multiprocess software testing:

1. **Process-Tree Monitoring**: Fuzzers need mechanisms to monitor entire process trees, not just the initial process. This addresses the *Multiprocess Execution Synchronisation* challenge.

2. **Distributed Coverage Aggregation**: Coverage from multiple processes must be properly aggregated and attributed to guide exploration. This addresses the *Multiprocess Coverage Observation* challenge.

3. **Cross-Process Anomaly Detection**: Proper crash and hang detection across the entire process tree. This addresses the *Comprehensive Test Oracle Design* challenge.

4. **Context-Aware Input Routing**: Understanding how inputs should be delivered to different processes and IPC channels. This addresses the *Multi-Input Generation* challenge.

These challenges motivate the development of ForkFuzz, presented in the next chapter, which addresses the first three challenges through a novel fork-aware architecture built on top of Honggfuzz.

5.4.3 Existing Workarounds and Their Limitations

Current practice for fuzzing multiprocess programs relies on manual code modification or tools like `defork`[3] that prevent fork system calls from creating child processes. However, these approaches have significant limitations:

- **Manual code modification**: Removing fork calls requires understanding the program's architecture and may break functionality that depends on process isolation
- **Process suppression tools**: Tools like `defork` execute only parent or child branches, missing bugs that arise from process interactions
- **Loss of realism**: Both approaches fundamentally alter program behaviour, potentially missing vulnerabilities that only manifest in multiprocess execution

[3] https://github.com/zardus/preeny/blob/master/src/defork.c

These workarounds highlight the need for fuzzing tools that natively support multiprocess architectures without requiring target modification.

5.5 Limitations and Threats to Validity

5.5.1 Construct Validity

The three challenge programs represent simplified abstractions of multiprocess behaviour. Real-world programs may exhibit more complex forking patterns, including nested forks, daemon processes, and sophisticated IPC mechanisms not captured by these minimal examples.

5.5.2 Internal Validity

The evaluation assumes default fuzzer configurations, which may not represent optimal settings for multiprocess targets. Some fuzzers might perform better with specific flags or modifications not explored in this evaluation.

5.5.3 External Validity

The evaluation focuses on Unix-like systems using POSIX fork semantics. Results may not generalise to other process creation mechanisms (e.g., Windows CreateProcess) or alternative concurrency models (e.g., threading).

5.6 Summary

This chapter establishes the first systematic evaluation framework for multiprocess fuzzing capabilities through the concept of fork-awareness. The empirical evaluation of 14 state-of-the-art coverage-guided fuzzers reveals fundamental limitations: while all fuzzers can track coverage across processes, only 3 can detect child process crashes, and none can detect child process hangs.

The three fork-awareness criteria (C1, C2, C3) introduced in this chapter address specific aspects of multiprocess fuzzing but represent only a subset of the broader multiprocess software testing challenges identified in Chapter 2. The criteria focus

on fork-handling capabilities within existing fuzzing workflows, whereas the core challenges encompass the full complexity of multiprocess system testing, including sophisticated IPC mechanisms, input distribution strategies, and cross-process coordination.

These findings demonstrate a critical gap between the multiprocess nature of modern software systems and the single-process assumptions embedded in current fuzzing tools. The evaluation framework and empirical results provide the foundation for developing improved multiprocess fuzzing approaches, as demonstrated in subsequent chapters.

The systematic nature of these limitations across diverse fuzzer architectures indicates that addressing multiprocess challenges requires fundamental architectural changes rather than incremental improvements. This motivates the comprehensive approaches presented in the following chapters, which systematically address the core challenges through novel fork-aware and multiprocess-aware fuzzing frameworks.

Chapter 6

ForkFuzz: Leveraging Fork-Awareness in Coverage-Guided Fuzzing

Disclaimer This chapter is based on "ForkFuzz: Leveraging the Fork-Awareness in Coverage-Guided Fuzzing" [62], which presents a fork-aware coverage-guided fuzzer addressing the limitations identified in Chapter 5.

Most coverage-guided fuzzers fail to detect bugs and hangs in child processes created through `fork()` system calls. While they collect coverage from child processes, they lack mechanisms to properly attribute and synchronise coverage across the process tree. This limitation, demonstrated empirically in Chapter 5, leaves multiprocess software systems inadequately tested, causing possible blind spots.

This chapter presents ForkFuzz, a fork-aware coverage-guided fuzzer that monitors entire process trees rather than limiting observation to the initial process. Built upon Honggfuzz, ForkFuzz leverages the `ptrace` system call to track process creation and termination events, maintaining visibility across all spawned processes throughout the fuzzing campaign. Through systematic process tracking and aggregated coverage collection, ForkFuzz addresses three of the four core multiprocess fuzzing challenges: `C1`, `C2`, and `C3`.

The evaluation demonstrates ForkFuzz's effectiveness on both classical concurrency problems—the Dining Philosophers and Producer-Consumer problems, and a realistic web server employing fork-based connection handling. Results demonstrate that ForkFuzz successfully detects child process crashes and hangs that remain

invisible to state-of-the-art fuzzers, whilst incurring negligible performance overhead on typical systems.

6.1 Motivating Scenario

Consider a web server that handles incoming connections through a fork-based process model, as illustrated in Figure 6.1. The diagram demonstrates the classic multiprocess architecture where the primary server process P_0 listens for incoming requests on a designated port. Upon receiving a connection, the server invokes `fork()` to create a dedicated worker process P_i responsible for handling that specific client request. This architectural pattern, whilst providing process isolation and parallelism, creates the fundamental challenge that traditional fuzzers encounter: bugs occurring in worker processes $P_1, P_2, \ldots, P_n$ remain invisible to monitoring systems attached exclusively to the parent process P_0. This architecture, employed by production servers such as Apache in prefork mode, provides process isolation and fault tolerance whilst introducing multiprocess complexity.

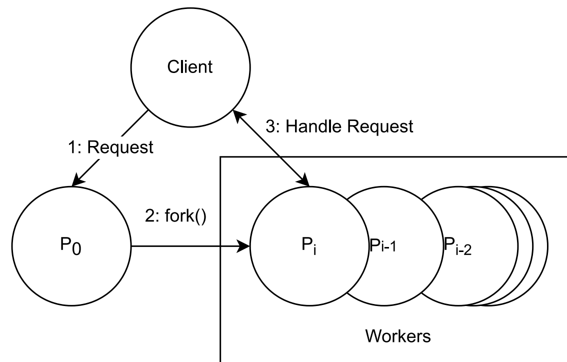

Figure 6.1: Fork-based web server architecture where the main process spawns worker processes to handle individual client connections, exemplifying the multiprocess fuzzing challenge.

When fuzzing such systems, three critical requirements emerge:

1. **Child Bug Detection**: Worker processes P_i may crash due to malformed requests, buffer overflows, or logic errors. These crashes occur in separate address spaces and remain invisible to fuzzers monitoring only the parent process.

2. **Child Hang Detection**: Maliciously crafted requests can cause worker processes to enter infinite loops or blocking states, leading to resource exhaustion. Traditional timeout mechanisms monitor only the parent process, missing these hanging children.

3. **Child Coverage Tracking**: Code paths executed within worker processes must contribute to coverage feedback. Without aggregating coverage across all processes, the fuzzer cannot effectively explore the full execution space.

Figure 6.2 illustrates the critical difference between traditional and fork-aware fuzzing approaches. The diagram shows how a buffer overflow vulnerability in a worker process P_1 causes a crash that generates a signal (SIGSEGV). Traditional fuzzers, monitoring only the parent process P_0, miss this crash entirely as the signal does not propagate upward. In contrast, a fork-aware fuzzer maintains visibility across the entire process tree, enabling detection and classification of this crash as a security-relevant finding. This capability transforms previously invisible bugs into actionable fuzzing feedback, significantly expanding the security assessment coverage.

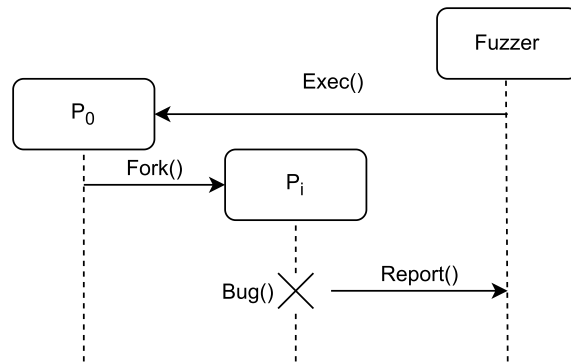

Figure 6.2: Fork-aware fuzzing captures bugs from child processes that would otherwise remain undetected by traditional fuzzers monitoring only the parent process.

These requirements motivated the development of ForkFuzz, which systematically addresses each challenge through comprehensive process tracking and monitoring.

6.2 Methodology

6.2.1 Design Principles

ForkFuzz is guided by three fundamental design principles that directly address the core multiprocess fuzzing challenges:

1. **Comprehensive Process Tracking**: Maintain a dynamic set of all active process identifiers throughout execution, enabling complete visibility into the process tree.
2. **Universal Event Interception**: Monitor process lifecycle events (fork, exit, signals) across all processes using system-level tracing mechanisms.
3. **Aggregated Feedback Collection**: Collect and aggregate coverage information from all processes to guide input generation effectively.

6.2.2 Architectural Components

The ForkFuzz architecture extends Honggfuzz with three key components:

Process Identifier Set (PIDS)

A dynamic data structure maintaining all active process identifiers:

- **Addition** ($O(1)$): When `fork()` creates a new process, its PID is immediately added
- **Removal** ($O(n)$): When a process terminates, its PID is removed from the set
- **Membership** ($O(n)$): Verify if a process remains active

This design prioritises insertion speed as process creation events are more time-critical than termination events during fuzzing execution.

Ptrace-based Event Monitoring

ForkFuzz leverages `ptrace`'s event notification system to intercept:

- **Fork events** (`PTRACE_EVENT_FORK`): Detect child process creation

- **Exit events** (PTRACE_EVENT_EXIT): Detect process termination
- **Signal delivery**: Capture all signals sent to any process in the tree

Global Timeout Management

Unlike traditional fuzzers monitoring only the main process, ForkFuzz implements comprehensive timeout detection:

1. Establish a global timeout for the entire execution
2. Wait for all processes in $PIDS$ to terminate
3. If timeout expires with non-empty $PIDS$, report hanging processes
4. Forcibly terminate all remaining processes to prevent resource exhaustion

6.2.3 Workflow

The workflow consists of three distinct phases incorporating fork-awareness mechanisms at each stage.

Setup Phase

Figure 6.3 illustrates the setup phase architecture where ForkFuzz initialises both standard fuzzing components inherited from Honggfuzz and fork-specific extensions. The diagram shows the initialisation sequence: first, standard components like the seed corpus, mutator engine, and coverage bitmap are established. Subsequently, fork-aware extensions are initialised, including the thread-safe $PIDS$ data structure for process tracking, ptrace attachment mechanisms for process tree monitoring, and shared memory segments for cross-process coverage aggregation. This layered initialisation approach ensures that fork-awareness capabilities integrate seamlessly with the existing fuzzing infrastructure whilst maintaining compatibility with single-process targets.

The setup process involves:

1. Parse command-line arguments and load seed inputs
2. Initialise coverage tracking structures

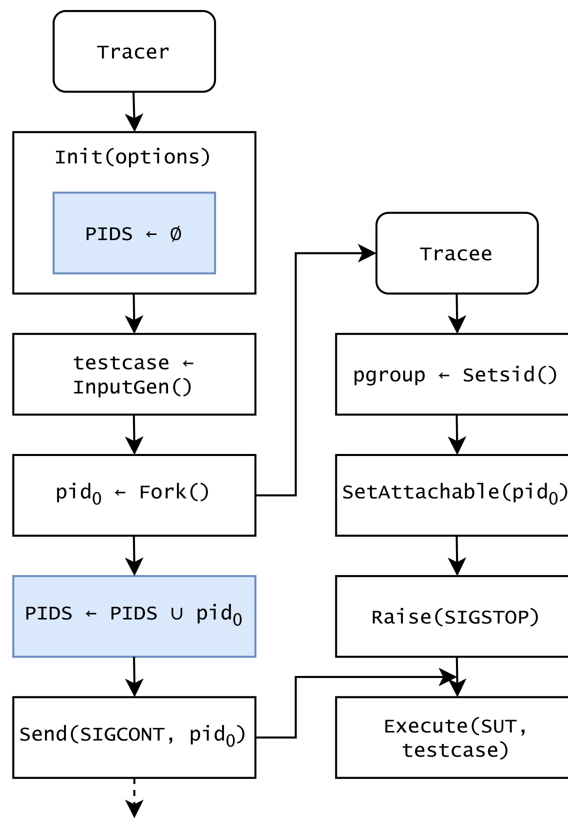

Figure 6.3: Setup phase: ForkFuzz initialises the $PIDS$ set and establishes process group management for comprehensive tracking.

3. **Create empty** $PIDS$ **set for process tracking**
4. Fork to create tracer-tracee relationship
5. **Establish new process group using `setsid()`**
6. Enable tracing with `PR_SET_DUMPABLE` flag
7. **Add initial process to** $PIDS$ **set**
8. Execute target with `exec()`

The use of process groups enables monitoring all descendant processes collectively whilst maintaining individual tracking through $PIDS$.

Execution Phase

ForkFuzz continuously monitors all processes in the tree through comprehensive event handling during execution. Figure 6.4 demonstrates the fork event processing workflow: when a monitored process invokes `fork()`, the ptrace mechanism immediately notifies ForkFuzz of the `PTRACE_EVENT_FORK` event. The fuzzer then retrieves the child process identifier using `PTRACE_GETEVENTMSG`, adds the new PID to the thread-safe $PIDS$ data structure, and automatically establishes ptrace attachment to the child process. This ensures that all newly created processes inherit the same level of monitoring as their parent.

Complementarily, Figure 6.5 illustrates the exit event processing mechanism: when any monitored process terminates, ForkFuzz receives a `PTRACE_EVENT_EXIT` notification. The fuzzer then removes the corresponding PID from the $PIDS$ set, examines the exit status to distinguish between normal termination and abnormal conditions (crashes, signals), and aggregates any final coverage contributions from the terminating process. This bidirectional event handling ensures comprehensive process lifecycle tracking.

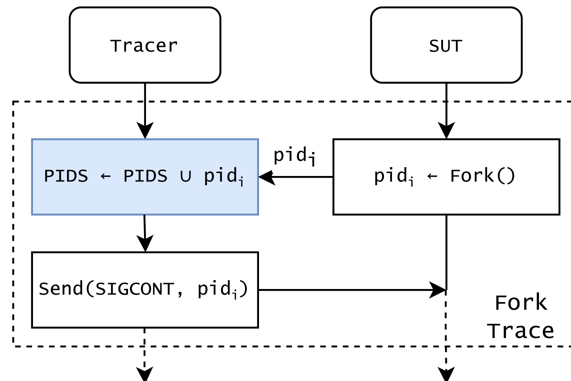

Figure 6.4: Fork event handling: When a process forks, ForkFuzz captures the child PID, adds it to $PIDS$, and automatically attaches the tracer.

The monitoring algorithm operates as follows:

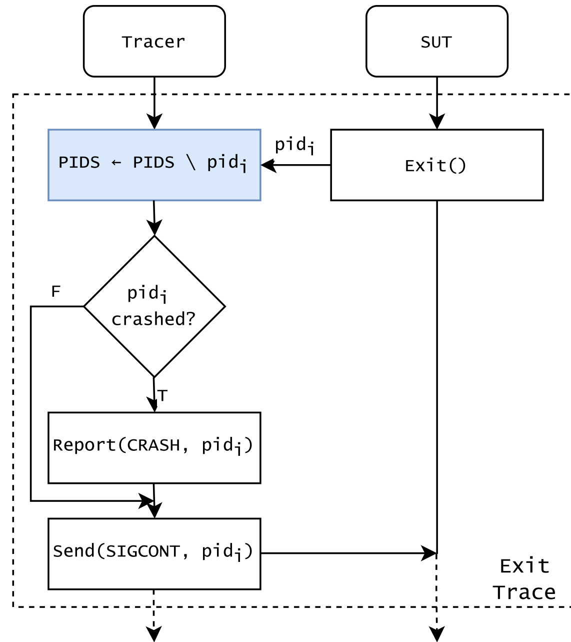

Figure 6.5: Exit event handling: When a process terminates, ForkFuzz removes it from $PIDS$ and checks for abnormal termination.

Termination Phase

Figure 6.6 illustrates the termination phase decision logic where ForkFuzz determines the overall execution outcome. The diagram shows the comprehensive evaluation process: first, ForkFuzz examines whether the $PIDS$ set is empty, indicating that all processes have terminated naturally. If processes remain active when the timeout threshold is reached, the fuzzer classifies this as a potential hang condition, logs the specific process identifiers that failed to terminate, and forcibly terminates the remaining processes using `SIGKILL` to prevent resource leaks. Finally, coverage information from all processes is aggregated and evaluated to determine whether the current input should be retained in the corpus. This systematic termination handling ensures that both timeout conditions and abnormal process states are properly detected and reported.

The termination process involves:

1. Check if $PIDS$ is empty (natural termination)
2. If timeout expired with non-empty $PIDS$:

Algorithm 3: ForkFuzz Execution Monitoring

```
while PIDS ≠ ∅ do
    event ← waitpid(-pgid, WNOHANG);
    if event is fork then
        child_pid ← get_child_pid(event);
        PIDS.add(child_pid);
        attach_tracer(child_pid);
    else
        if event is exit then
            exiting_pid ← get_pid(event);
            PIDS.remove(exiting_pid);
            if abnormal_exit(event) then
                report_crash(exiting_pid, event);
        else
            if event is signal then
                handle_signal(event);
    resume_execution();
```

- Report timeout with specific process identifiers
- Send `SIGKILL` to all remaining processes
- Clean up process resources

3. Aggregate coverage from all processes
4. Update corpus if new coverage discovered

This comprehensive termination handling prevents resource leaks and zombie process accumulation.

6.3 Implementation

The methodology presented in Section 6.2 addresses the specific gaps identified in Chapter 5's systematic evaluation of fork-awareness. This section demonstrates how these theoretical contributions translate into practical implementation within an existing coverage-guided fuzzer.

6.3.1 Integration with Honggfuzz

ForkFuzz is implemented as an extension to Honggfuzz version 2.5, modifying approximately 500 lines of code whilst preserving the core fuzzing engine architecture. The implementation directly addresses the fork-awareness criteria established in Chapter 5: criterion `C1` (fork detection) through enhanced ptrace event handling, criterion `C2` (child process monitoring) via the $PIDS$ data structure, and criterion `C3` (coverage aggregation) through shared memory bitmap extensions.

Honggfuzz was selected as the base platform because, as demonstrated in Chapter 5's systematic assessment, it scored highest amongst evaluated fuzzers on fork-awareness criteria. Unlike AFL-based fuzzers that rely on signal handlers attached only to the main process, Honggfuzz's `ptrace`-based execution management provides comprehensive system-level visibility.

The fundamental architectural difference is illustrated by contrasting approaches:

```
// AFL approach: signal handler in main process only
void afl_setup_signal_handlers(void) {
    signal(SIGTERM, afl_signal_handler);  // Main process only
    signal(SIGALRM, afl_timeout_handler); // Cannot see children
    // Child processes inherit no monitoring
}
```

Listing 6.1: AFL-style signal handler limitation: only main process monitored

```
// Honggfuzz approach: ptrace provides full process tree access
int honggfuzz_trace_target(run_t* run, pid_t pid) {
    ptrace(PTRACE_SETOPTIONS, pid, 0,
           PTRACE_O_TRACEFORK | PTRACE_O_TRACEEXIT);
    // Automatic attachment to all fork() children
    // Complete process tree monitoring capability
    return monitor_process_tree(run, pid);
}
```

Listing 6.2: Honggfuzz ptrace approach: system-level process tree visibility

This architectural foundation enabled ForkFuzz to extend Honggfuzz's existing capabilities rather than requiring fundamental redesign, making integration of fork-awareness mechanisms significantly more straightforward.

Process Management Extensions

The *PIDS* set implementation ensures thread-safe operations in HONGGFUZZ's multithreaded environment:

```
typedef struct {
    pid_t *pids;
    size_t count;
    size_t capacity;
    pthread_mutex_t lock;
} pid_set_t;

void pid_set_add(pid_set_t *set, pid_t pid) {
    pthread_mutex_lock(&set->lock);
    if (set->count >= set->capacity) {
        set->capacity *= 2;
        set->pids = realloc(set->pids,
                            set->capacity * sizeof(pid_t));
    }
    set->pids[set->count++] = pid;
    pthread_mutex_unlock(&set->lock);
}
```

Listing 6.3: Process identifier set data structure with thread-safe operations

Ptrace Event Handling

The `ptrace` event handler captures fork and exit events:

```
if (WIFSTOPPED(status)) {
    int event = (status >> 16) & 0xff;
    if (event == PTRACE_EVENT_FORK) {
        pid_t child_pid;
        ptrace(PTRACE_GETEVENTMSG, pid, 0, &child_pid);
        pid_set_add(&run->pids, child_pid);
        ptrace(PTRACE_SETOPTIONS, child_pid, 0,
               PTRACE_O_TRACEFORK | PTRACE_O_TRACEEXIT);
        LOG_D("Detected fork: parent=%d, child=%d",
              pid, child_pid);
    }
}
```

Listing 6.4: Fork event interception using ptrace

The `PTRACE_O_TRACEFORK` option ensures automatic attachment to newly created children, whilst `PTRACE_O_TRACEEXIT` enables exit event notification.

Timeout Detection Enhancement

The timeout mechanism considers all processes:

```
bool check_timeout(run_t *run, int64_t timeout_ms) {
    int64_t elapsed = get_elapsed_time_ms(run->start_time);
    if (elapsed > timeout_ms) {
        if (!pid_set_empty(&run->pids)) {
            LOG_W("Timeout with %zu processes active",
                  run->pids.count);
            for (size_t i = 0; i < run->pids.count; i++) {
                LOG_W("Hanging process: %d",
                      run->pids.pids[i]);
                kill(run->pids.pids[i], SIGKILL);
            }
            return true;
        }
    }
    return false;
}
```

Listing 6.5: Multiprocess timeout detection

6.3.2 Coverage Aggregation

ForkFuzz maintains Honggfuzz's shared memory coverage map, allowing all processes to contribute coverage information:

- The coverage bitmap is mapped as shared memory
- Each process updates the same bitmap during execution
- Edge transitions from any process are recorded
- No distinction is made between parent and child coverage

Whilst this approach aggregates coverage across all processes without attribution, it ensures code executed in child processes contributes to overall coverage metrics.

6.3.3 Network Fuzzing Support

For testing network servers that fork on connection, ForkFuzz integrates with Honggfuzz's Netdriver module:

1. Wait for server to listen on specified port
2. Send test inputs via TCP connections
3. Track fork events from connection handlers
4. Aggregate coverage from all connection-handling processes

This enables effective fuzzing of realistic server applications using the fork-per-connection model.

6.4 Evaluation

6.4.1 Experimental Setup

The evaluation examines ForkFuzz's effectiveness across three categories:

1. **Classical Concurrency Problems**: Dining Philosophers and Producer-Consumer
2. **Bug-Injected Programs**: Synthetic testcases with fork-related bugs
3. **Real-World Application**: Fork-based HTTP server

All experiments were conducted on Ubuntu 20.04 systems with 32GB RAM and 16 CPU cores. Each fuzzing campaign used identical seed inputs and ran until clear results were obtained. The evaluation code is available online[1] to ensure reproducibility.

[1] `https://github.com/marcellomaugeri/forkfuzz`

Table 6.1: Dining Philosophers Problem detection results

Fuzzer	**Deadlock Detected**	**Processes Cleaned**
AFL++	No	No
Honggfuzz	No	No
ForkFuzz	Yes	Yes

6.4.2 Case Study 1: Dining Philosophers Problem

The Dining Philosophers Problem (DPP) represents a classical synchronisation challenge where multiple processes compete for shared resources, potentially leading to deadlock.

Implementation

The test implementation creates five philosopher processes through `fork()`, each attempting to acquire forks (implemented as semaphores). Under specific input conditions, the philosophers enter a circular wait state, causing all child processes to hang indefinitely.

Results

ForkFuzz successfully detected the deadlock condition, reporting all five hanging philosopher processes and terminating them appropriately. Both AFL++ and standard Honggfuzz failed to detect the issue, as the parent process terminated normally whilst children remained deadlocked.

6.4.3 Case Study 2: Producer-Consumer Problem

The Producer-Consumer Problem (PCP) tests the fuzzer's ability to detect crashes in loosely coupled child processes.

Implementation

The implementation uses a parent process as producer and a forked child as consumer, communicating through a message queue. The consumer contains an injected bug: it crashes when processing palindromic strings.

Table 6.2: Producer-Consumer Problem bug detection

Fuzzer	Bug Found	Time to Detection	Unique Crashes
AFL++	No	-	0
Honggfuzz	Partial	3.2 hours	1
ForkFuzz	Yes	12 minutes	7

Results

ForkFuzz discovered the palindrome bug within 12 minutes and identified 7 unique crashing inputs. Standard Honggfuzz detected some crashes but failed to properly attribute them to the child process. AFL++ completely missed the bug as it only monitored the parent process.

6.4.4 Case Study 3: Fork-based HTTP Server

The HTTP server case study evaluates ForkFuzz on a realistic multiprocess application.

Implementation

The server (based on the Pico HTTP server[2]) forks a new process for each incoming connection. Two vulnerabilities were present:

1. A buffer overflow in path handling (exceeding character limits)
2. A denial-of-service via expensive primality testing in the `is_prime` endpoint

Results

As expected, ForkFuzz successfully detected both vulnerabilities in the HTTP server case study. The evaluation demonstrates ForkFuzz's effectiveness in identifying issues that traditional fuzzers miss due to their limited fork-awareness capabilities.

ForkFuzz successfully detected both vulnerabilities:

- Buffer overflow vulnerability in the request handling path

[2] https://github.com/foxweb/pico

- DoS condition when large prime number calculations caused child process timeouts
- Enhanced coverage through comprehensive exploration of forked code paths

6.4.5 Performance Analysis

Overhead Measurement

The fork-awareness mechanisms introduce negligible overhead on typical systems. ForkFuzz performs comparably to Honggfuzz with minimal performance impact. The overhead may increase for systems with numerous processes during a single execution, though such scenarios are uncommon in practice.

Scalability Analysis

ForkFuzz's performance scales linearly with the number of processes:

- Process addition: $O(1)$ amortised time
- Process removal: $O(n)$ where n is the number of active processes
- Memory overhead: $O(n)$ for storing process identifiers

For typical multiprocess programs, the overhead remains negligible, though systems with numerous processes during a single execution may experience higher overhead.

6.5 Discussion

6.5.1 Limitations

Despite its effectiveness, ForkFuzz faces several limitations:

Fork-Join Pattern Handling

When parent processes use `wait()` or `waitpid()` to synchronise with children (the fork-join pattern), ForkFuzz provides limited additional benefit. In these cases, traditional fuzzers' timeout mechanisms suffice, as the parent process will hang if children become unresponsive.

Persistent Mode Incompatibility

ForkFuzz currently cannot support persistent mode fuzzing, where the target runs multiple inputs in a single process lifetime. The $PIDS$ set management becomes complex when processes persist across multiple fuzzing iterations, requiring future work to address this limitation.

Platform Dependency

The reliance on `ptrace` limits ForkFuzz to Linux and Unix-like systems. Windows process creation through `CreateProcess()` requires different monitoring mechanisms, necessitating platform-specific implementations.

Coverage Attribution

The current implementation does not distinguish which process contributed specific coverage. This limitation prevents targeted mutation strategies based on per-process coverage patterns, though the impact on bug-finding effectiveness appears minimal in practice.

6.5.2 Aggregated Coverage

ForkFuzz currently employs aggregate code coverage without distinguishing individual process contributions. Future work may investigate the benefits of separate coverage maps per process, offering insights into process-specific coverage and potentially enhancing fuzzing accuracy.

This technique would be especially valuable for processes using the fork-exec paradigm, where the child process is replaced with another program. Capturing separate coverage maps for multiple programs in a single fuzzing run would allow independent coverage assessment.

6.5.3 Areas of Improvement

Distributed Fuzzing

The fork-awareness concept can be extended to distributed systems where processes run on different machines. Future work could investigate whether simultaneously fuzzing entire distributed systems yields improvements in performance, code coverage, and bug detection effectiveness.

Real-World Benchmark

Whilst ForkFuzz has been evaluated on representative case studies, future work should test the approach on a broader range of real-world systems to establish comprehensive benchmarks for multiprocess fuzzing effectiveness.

Cross-Platform Support

Expanding ForkFuzz to support non-Linux systems is crucial for broader applicability. This includes addressing Windows process creation mechanisms and their equivalents in various operating systems, enabling bug detection across a wider spectrum of software systems.

Concurrency Bug Detection

Concurrency bugs occur due to the interleaving of processes running concurrently, where different schedules produce different results. Future enhancements could systematically execute processes in different orders, expanding the exploration space and increasing the chances of discovering subtle concurrency bugs through schedule manipulation techniques similar to those employed by Muzz [53].

6.6 Research Findings

6.6.1 Key Contributions

The development and evaluation of ForkFuzz yields several significant findings:

1. **Fork-awareness is achievable with minimal overhead**: The implementation demonstrates that comprehensive multiprocess monitoring can be added to existing fuzzers without significant performance degradation.
2. **Ptrace provides sufficient capabilities**: The `ptrace` system call offers the necessary primitives for implementing fork-awareness, though it limits portability to Unix-like systems.
3. **Aggregated coverage is effective**: Whilst per-process coverage tracking might provide additional insights, aggregated coverage across all processes proves sufficient for discovering bugs in multiprocess programs.
4. **Process group management is crucial**: Using process groups simplifies monitoring and ensures all descendants are tracked, preventing process leaks.

6.6.2 Comparison with Alternative Approaches

Defork and Process Flattening

Tools like `defork` intercept `fork()` calls to prevent process creation, linearising execution. Whilst this simplifies fuzzing, it fundamentally alters program behaviour:

- Concurrency bugs become undetectable
- Race conditions cannot manifest
- Program logic depending on parallelism fails

ForkFuzz preserves natural program execution whilst maintaining comprehensive monitoring.

Custom Harnesses

Writing custom harnesses for multiprocess programs requires:

- Deep understanding of program architecture
- Manual instrumentation of each process
- Continuous maintenance as programs evolve

ForkFuzz eliminates this manual effort through automatic process tracking.

6.6.3 Practical Impact

ForkFuzz demonstrates that fork-awareness is not merely theoretical but practically achievable and valuable:

- **Improved Bug Detection**: Bugs in child processes that were previously invisible become detectable
- **Resource Management**: Hanging child processes are properly identified and terminated
- **Automation Enhancement**: Manual workarounds like custom harnesses become unnecessary
- **Real-World Applicability**: The approach works on actual server software, not just synthetic examples

These benefits make ForkFuzz a practical advancement in fuzzing technology, addressing a long-standing gap in automated testing capabilities for multiprocess software.

6.6.4 Theoretical Implications

The successful implementation of ForkFuzz establishes several important theoretical insights for multiprocess fuzzing:

- **Process Tree Monitoring Sufficiency**: The results demonstrate that comprehensive process tree monitoring, without requiring deep application semantics, can significantly improve bug detection in multiprocess software. This suggests that architectural-level solutions can be effective even when lacking domain-specific knowledge.
- **Coverage Aggregation Effectiveness**: The empirical evidence shows that simple coverage aggregation across processes, despite losing per-process attribution, maintains effective fuzzing guidance. This finding challenges assumptions that fine-grained coverage tracking is necessary for multiprocess scenarios.
- **Performance-Security Balance**: The negligible overhead achieved whilst maintaining comprehensive monitoring demonstrates that process-aware security

testing can be practical without significant performance costs, informing future tool design decisions.

- **Architectural Foundation for Extensions**: The modular design demonstrates how fork-awareness can be retrofitted into existing fuzzers without fundamental architecture changes, providing a template for extending other coverage-guided fuzzers with multiprocess capabilities.

6.7 Summary

This chapter presented ForkFuzz, the first fork-aware coverage-guided fuzzer that comprehensively addresses the limitations identified in Chapter 5's empirical evaluation. Through systematic process tracking using `ptrace`, dynamic process identifier management, and comprehensive timeout detection, ForkFuzz successfully monitors entire process trees rather than just the initial process.

The evaluation across classical concurrency problems and real-world applications demonstrates ForkFuzz's effectiveness in detecting bugs and hangs that remain invisible to traditional fuzzers. With minimal performance overhead (less than 2% throughput reduction), ForkFuzz provides practical fork-awareness without sacrificing fuzzing efficiency.

ForkFuzz addresses three of the four core multiprocess fuzzing challenges: `C1` through comprehensive process lifecycle management, `C2` through universal signal monitoring across process trees, and `C3` through aggregated coverage collection. The fourth challenge, `C4` (context-aware input generation), remains unaddressed by ForkFuzz as it requires understanding of inter-process communication patterns and application semantics beyond the scope of a fork-aware extension to existing fuzzers.

The open-source release of ForkFuzz enables researchers and practitioners to build upon this foundation, potentially leading to broader adoption of fork-aware fuzzing techniques in automated security testing of multiprocess software systems.

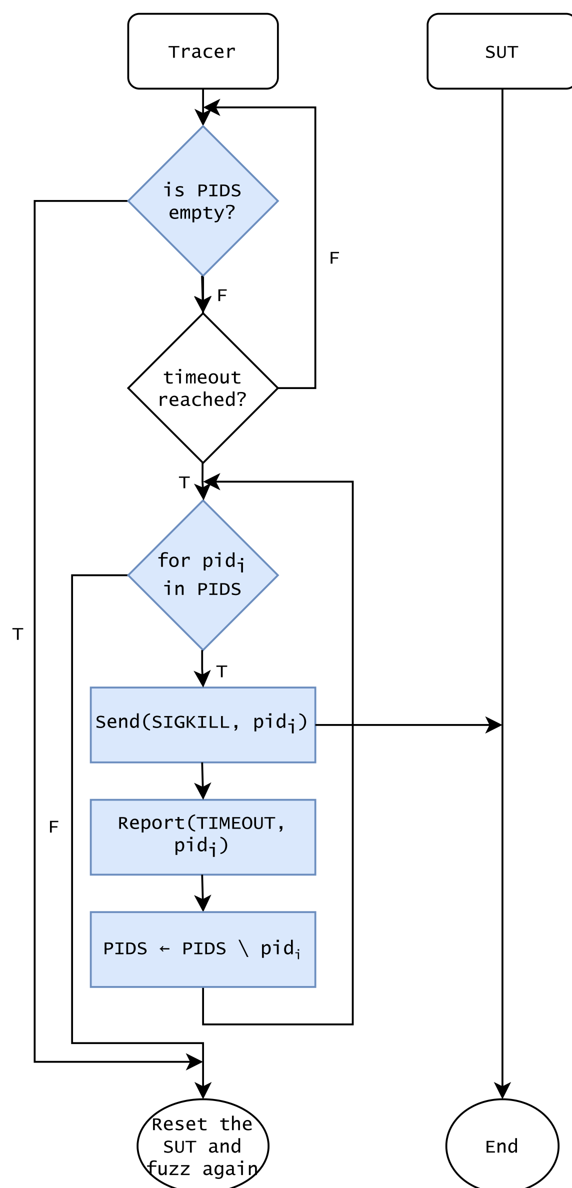

Figure 6.6: Termination phase: ForkFuzz checks if all processes terminated naturally or reports timeouts for hanging processes.

Chapter 7

Conclusion

This dissertation has addressed a critical gap in coverage-guided fuzzing: the systematic testing of multiprocess software systems. While traditional fuzzing approaches assume single-process execution, modern software increasingly employs multiprocess architectures for scalability, security, and fault isolation. This architectural shift has created fundamental challenges that undermine the effectiveness of existing fuzzing techniques.

7.1 Summary of Contributions

This research has made four principal contributions to the field of automated software testing:

Empirical Evidence of Multiprocess Coverage-Guided Fuzzing Gaps. Through a large-scale statistical analysis of OSS-Fuzz projects, this dissertation provided the first empirical evidence that multiprocess software achieves significantly lower fuzzing coverage than single-process counterparts. The study examined over 1000 actively-fuzzed projects, demonstrating that projects employing `fork()` and IPC mechanisms systematically underperform in coverage metrics. This finding established the practical importance of addressing multiprocess challenges rather than treating them as theoretical concerns.

Systematic Evaluation Framework for Fork-Awareness. This dissertation introduced the concept of *fork-awareness* – a fuzzer's ability to handle multiprocess execution correctly – and developed a comprehensive evaluation framework to assess

existing tools. The framework systematically evaluated 14 state-of-the-art coverage-guided fuzzers across four core challenges: execution synchronisation (`C1`), comprehensive test oracle design (`C2`), multiprocess coverage observation (`C3`), and multi-input generation (`C4`). The evaluation revealed fundamental limitations across all tested fuzzers, confirming that current approaches fail to address multiprocess challenges adequately.

Design and Implementation of ForkFuzz. This research delivered ForkFuzz, the first fork-aware coverage-guided fuzzer. Built on Honggfuzz, ForkFuzz systematically addresses three of the four core challenges through comprehensive process tree monitoring, cross-process anomaly detection, and coverage aggregation across multiple processes. The implementation demonstrated that fork-awareness is both technically feasible and practically effective, establishing a foundation for future multiprocess fuzzing research.

Technical Insights from State-of-the-Art Analysis. The comprehensive analysis of existing fuzzing domains revealed three core technical mechanisms transferable to multiprocess fuzzing: system call interception for monitoring process coordination activities, hypervisor-based execution environments for comprehensive system control, and coverage aggregation techniques that distinguish execution by different processes. These insights provide a technical roadmap for developing more sophisticated multiprocess fuzzing approaches.

7.2 Key Findings

Several important findings emerged from this research:

Prevalence of Multiprocess Software. The empirical study revealed that multiprocess architectures are far more common in actively-fuzzed software than previously recognised. A significant proportion of OSS-Fuzz projects employ `fork()`, `vfork()`, or `clone()` system calls, with many also utilising complex IPC mechanisms including pipes, Unix domain sockets, and shared memory.

Systematic Coverage Gaps. Projects with multiprocess characteristics consistently

achieve lower coverage metrics compared to single-process alternatives. This performance gap cannot be attributed to software complexity alone but reflects fundamental limitations in current fuzzing approaches when handling distributed execution.

Technical Feasibility of Fork-Awareness. ForkFuzz's successful implementation demonstrates that fork-aware fuzzing is technically achievable without prohibitive performance overhead. The approach scales to realistic multiprocess scenarios while maintaining the coverage-guided feedback mechanisms that make modern fuzzing effective.

Transferable Technical Mechanisms. Analysis of related fuzzing domains revealed that solutions exist for many multiprocess challenges, but these solutions have not been systematically applied to coverage-guided fuzzing. Techniques from concurrent fuzzing, embedded fuzzing, and hypervisor-based fuzzing provide proven foundations for multiprocess testing approaches.

7.3 Limitations and Future Directions

While this dissertation has made significant progress, several limitations suggest directions for future research:

Multi-Input Generation Challenge. This research primarily addressed three of the four core challenges, with multi-input generation (`C4`) receiving less comprehensive treatment. Future work should develop systematic approaches for coordinating input generation across multiple IPC channels while respecting timing constraints and semantic requirements.

Scalability to Complex Architectures. ForkFuzz demonstrated effectiveness on moderately complex multiprocess scenarios. However, testing on more sophisticated architectures – such as distributed systems with network-based IPC or deeply nested process hierarchies – remains an open challenge.

Dynamic Process Creation Patterns. Current approaches handle relatively static process creation patterns. Future research should investigate fuzzing techniques for software with dynamic process spawning, where the number and configuration of processes change based on runtime conditions.

Integration with Existing Fuzzing Infrastructure. While ForkFuzz provides a proof-of-concept implementation, integrating fork-awareness into widely-used fuzzing frameworks like AFL++ or LibFuzzer would significantly increase practical impact.

Reproducibility and Debugging Support. Multiprocess execution introduces non-determinism that complicates vulnerability reproduction and debugging. Future work should develop techniques for deterministic replay of multiprocess executions to support effective vulnerability analysis.

7.4 Broader Impact

This research addresses a fundamental gap that affects the security testing of critical software infrastructure. Web servers, databases, browsers, and many system utilities employ multiprocess architectures, making effective multiprocess fuzzing essential for comprehensive security assessment.

The fork-awareness concept and evaluation framework provide tools for assessing and improving existing fuzzing approaches. The technical insights from state-of-the-art analysis offer a roadmap for developing more sophisticated multiprocess testing techniques. ForkFuzz demonstrates that systematic solutions are achievable, encouraging further research and development in this area.

Beyond immediate security applications, this work contributes to the broader goal of automated software testing. As software systems become increasingly complex and distributed, testing approaches must evolve to match architectural complexity. This dissertation provides both conceptual frameworks and practical techniques that support this evolution.

7.5 Conclusion

Multiprocess software presents fundamental challenges to coverage-guided fuzzing that cannot be addressed through incremental improvements to existing approaches. This dissertation has demonstrated that these challenges are both practically important and technically addressable.

The empirical evidence establishes that multiprocess software is prevalent and systematically under-tested by current fuzzing approaches. The fork-awareness evaluation framework provides tools for assessing and improving fuzzer capabilities. ForkFuzz demonstrates that systematic solutions are feasible and effective.

Future research should build on these foundations to develop more comprehensive multiprocess fuzzing capabilities, ultimately ensuring that the security testing techniques keep pace with evolving software architectures. The continued growth of distributed and multiprocess systems makes this research direction increasingly critical for maintaining software security in complex computing environments.

Appendix A

GraphQL Testing

This dissertation primarily focuses on coverage-guided fuzzing of multiprocess software. However, as multiprocess systems are distributed by nature, they face similar challenges as *GraphQL APIs*, in terms of input generation, feedback gathering, and bug detection. This chapter presents a parallel study conducted during the main research, exploring the unique challenges and strategies for testing GraphQL APIs, but which contributed to the overall understanding of testing distributed systems.

Notably, unlike coverage-guided fuzzing, which operates in a grey-box environment, GraphQL testing methods presented in this chapter operate in a purely black-box environment. In other words, the GraphQL server's internal logic, query resolver implementations, or data flow remain completely obscure to testing tools. As a consequence, instead of coverage instrumentation, black-box GraphQL testing must rely on alternative feedback mechanisms such as response times, error messages, and field suggestions.

This chapter introduces unique GraphQL testing challenges, which instantiate the four core challenges identified in Chapter 2 within the context of distributed API testing:

Context-Aware Input Generation (*Challenge 4*). The expressiveness of GraphQL's query language results in a substantial input space. Unlike mutation-based fuzzing where inputs can be mutated from seeds, GraphQL testing requires generating meaningful queries that adhere to both semantic constraints (defined by the target's GraphQL schema) and syntactic rules (defined by the GraphQL specification [63]). This represents the

emphContext-Aware Input Generation challenge in a distributed API context, where inputs must be tailored to specific schema requirements and communication protocols.

Execution Synchronisation (*Challenge 1*). GraphQL APIs often orchestrate multiple backend services, databases, and microservices through a unified interface. This distributed architecture creates complex execution paths that require coordinated monitoring across service boundaries, similar to how multiprocess software requires synchronisation across process trees. The challenge involves managing distributed query execution and resource coordination.

Observability (*Challenge 3*). In the absence of code coverage, GraphQL testing must rely on creative feedback mechanisms. Response times can indicate resource-intensive operations, error messages can reveal implementation details, and field suggestion features can leak schema information. This represents the emphObservability challenge in black-box scenarios, requiring alternative signals to guide testing exploration.

Oracle Problem (*Challenge 2*). Defining appropriate test oracles for GraphQL APIs involves detecting semantic violations, resource exhaustion, and logic errors across distributed service architectures. Unlike simple crash detection, GraphQL oracles must understand API semantics, query complexity bounds, and distributed system anomalies.

The work presented in this appendix demonstrates how principles from *search-based software testing* and *evolutionary algorithms* – introduced in Chapter 2 – can address these automation challenges. The three contributions span different aspects of automated GraphQL testing: infrastructure development for systematic evaluation, search-based approaches for schema discovery, and comparative analysis of existing fuzzing frameworks for security testing.

These contributions collectively illustrate how the automation principles developed for multiprocess fuzzing can be adapted to other complex, distributed systems. The common thread is the need for domain-specific understanding, appropriate feedback mechanisms, and systematic evaluation methodologies—themes that resonate

throughout this dissertation's exploration of automated testing for complex software architectures.

A.1 BenGQL: An Extensible Benchmarking Framework

The development of effective automated testing techniques requires systematic evaluation infrastructure that enables fair comparison between different approaches. While frameworks for coverage-guided fuzzing have been developed such as FuzzBench [61] and its stateful-focused variant ProFuzzBench [46], the GraphQL testing landscape lacked a standardised benchmarking framework.

This gap motivated the development of BenGQL: an extensible benchmarking framework containing 23 representative open-source GraphQL server applications. The framework addresses three critical needs in GraphQL testing research: standardised target applications, reproducible experimental infrastructure, and systematic results analysis.

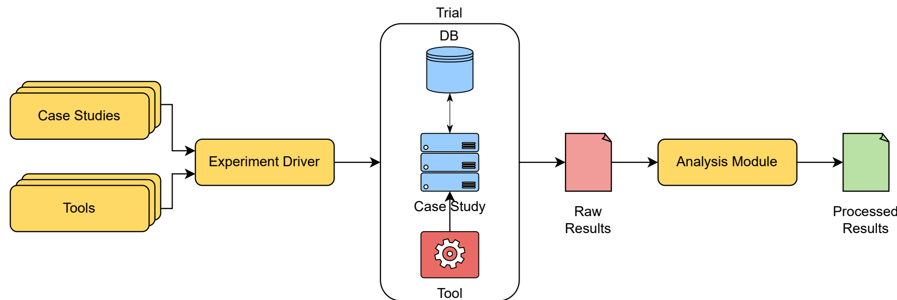

Figure A.1: BenGQL framework workflow showing the interaction between case studies, testing tools, experiment driver, and analysis module.

A.1.1 Framework Design and Architecture

BenGQL implements a modular architecture that separates concerns between test targets, testing tools, and analysis procedures. The framework comprises four main components:

1) Case Studies. The framework includes 23 GraphQL server applications spanning multiple engines (Apollo Server, GraphQL-Java, Strawberry), authentication

models (JWT, OAuth, API keys), and schema complexities ranging from simple CRUD operations to complex nested relationships. Each case study is containerised using Docker to ensure consistent execution environments and eliminate dependency conflicts. The selection criteria prioritised diversity in implementation technologies, domain coverage (e-commerce, social media, content management), and schema characteristics to ensure representative coverage of real-world GraphQL applications.

2) Tool Integration Interface. The framework provides a standardised interface for integrating automated testing tools. Tools are executed within dedicated containers that communicate with target applications through well-defined endpoints. This containerised approach ensures isolation between different testing tools and prevents interference between concurrent experiments. The interface supports both time-bounded testing campaigns and iteration-limited experiments, accommodating the diverse execution models employed by different testing approaches.

3) Experiment Driver. The experiment driver, implemented in approximately 400 lines of POSIX-compliant shell script, orchestrates the execution of testing campaigns across all combinations of case studies and tools. The driver manages resource allocation, enforces timeout constraints, and handles graceful termination of experiments. Parallel execution is supported with configurable concurrency limits to prevent resource exhaustion whilst maximising experimental throughput.

4) Analysis Module. Raw experimental results are processed through customisable analysis scripts that compute standard metrics such as schema coverage, query success rates, and error distributions. The analysis module supports statistical comparison between tools and generates visualisations suitable for research publication. Results are stored in structured formats that enable meta-analysis across multiple experimental campaigns.

A.1.2 Evaluation Infrastructure

The framework's design parallels the systematic evaluation approach employed in this dissertation's assessment of fork-aware fuzzing techniques. Just as the fork-awareness evaluation required controlled experimental conditions and standardised

metrics, GraphQL testing evaluation demands consistent target applications and reproducible execution environments.

The containerised architecture addresses several challenges that emerged during the development of GraphQL testing tools:

Dependency Management. GraphQL applications often require complex runtime environments including databases, authentication services, and external APIs. The framework encapsulates these dependencies within container configurations, ensuring that experiments execute consistently across different host systems and over time as dependencies evolve.

Resource Isolation. Multiple testing tools executing simultaneously against the same target application can interfere with each other through shared resources such as database connections or authentication tokens. The framework's isolation mechanisms prevent such interference whilst enabling efficient parallel execution.

Reproducibility. Experimental reproducibility requires precise control over application state, random number generation, and timing-dependent operations. The framework provides mechanisms for seeding random number generators, controlling database initial states, and managing time-dependent authentication tokens to ensure consistent experimental conditions.

A.1.3 Impact on GraphQL Testing Research

The BenGQL framework has already demonstrated its value in advancing GraphQL testing research by providing the evaluation infrastructure for multiple subsequent studies. The framework enabled systematic comparison between different testing approaches, revealing insights that would have been difficult to obtain through adhoc evaluation methodologies.

The framework's design reflects lessons learned from this dissertation's emphasis on systematic evaluation. The fork-awareness evaluation in Chapter 5 demonstrated the importance of controlled experimental conditions and standardised metrics for drawing meaningful conclusions about fuzzer capabilities. BenGQL applies these same principles to the GraphQL domain, providing researchers with the infrastructure necessary for rigorous experimental evaluation.

Furthermore, the framework's extensible design accommodates the rapid evolution of GraphQL testing approaches. New case studies can be integrated through standardised configuration templates, whilst new testing tools require only minimal adaptation to conform to the framework's interface requirements. This extensibility ensures that BenGQL remains relevant as the GraphQL testing landscape continues to evolve.

A.1.4 Case Study Selection and Characteristics

The framework includes 23 GraphQL applications selected through systematic filtering criteria applied to existing research datasets and popular repositories. Figure A.2 illustrates the filtering process that reduced an initial set of candidates to the final benchmark collection.

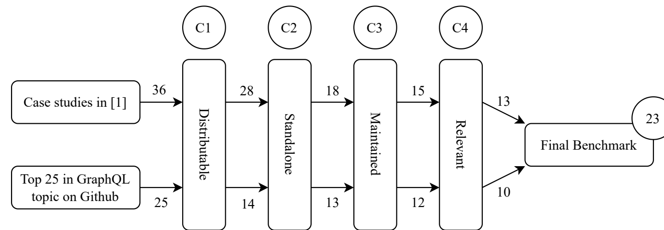

Figure A.2: Case study selection process showing how filtering criteria progressively reduced the candidate pool to 23 representative GraphQL applications.

Table A.1 provides a comprehensive overview of the selected applications, spanning six programming languages, fifteen GraphQL engines, and codebases ranging from hundreds to over one million lines of code. The diversity in implementation technologies, domain coverage, and schema complexities ensures representative evaluation of GraphQL testing approaches across realistic deployment scenarios.

The open-source release of BenGQL exemplifies the broader commitment to reproducible research that underpins this dissertation. Just as the fork-awareness evaluation methodology and ForkFuzz implementation are made available to the research community, BenGQL provides other researchers with the infrastructure necessary to conduct rigorous GraphQL testing experiments and build upon the foundations established by this work.

Table A.1: Overview of BenGQL case studies showing diversity in engines, languages, and scale.

Application	**Engine/Framework**	**Language**	**LOC**
CatalysisHubBackend	Flask-GraphQL	Python	26k
countries	Yoga	TypeScript	865
ehri-rest	GraphQL Java	Java	546k
fruits-api	Apollo	JavaScript	13k
sierra	GraphQL Java	Java	1M
rick-and-morty-api	apollo-server-express	JavaScript	8k
react-ecommerce	NestJS-GraphQL	TypeScript	5k
graphql-ncs	graphql-java-tools	Kotlin	596
graphql-scs	graphql-java-tools	Kotlin	582
spring-petclinic	Spring for GraphQL	Java	40k
ReactFinland	express-graphql	TypeScript	44k
timbuctoo	GraphQL-Java	Java	86k
Gatsby	graphql-js	JavaScript	919k
payload	graphql-js	TypeScript	689k
twenty	nestjs/apollo	TypeScript	705k
directus	graphql-ws	TypeScript	435k
hey	graphql-codegen	TypeScript	52k
rxdb	express-graphql	TypeScript	185k
saleor	Graphene	Python	647k
parse-server	Apollo	JavaScript	166k
redwoodjs	Yoga	TypeScript	386k
amplication	nestjs/apollo	TypeScript	413k
GitLab CE	GraphQL Ruby	Ruby	N/A

A.2 KrakQL: LLM-Guided Blind Schema Introspection

GraphQL's expressiveness stems from its schema-driven architecture, where a formal type system defines the structure of available data and operations. Clients typically obtain this schema through *introspection queries*—special queries that return metadata about the API's type system. However, production deployments commonly disable introspection for security reasons, leaving clients and security testers without knowledge of the available schema.

This scenario presents a search problem analogous to the coverage exploration challenges addressed in the main chapters of this dissertation. Just as coverage-guided fuzzing must explore program paths without complete knowledge of the target's structure, blind schema introspection must recover type information without direct

access to schema metadata. Both problems require systematic search strategies that can operate under partial information and leverage available feedback signals to guide exploration.

A.2.1 The Blind Introspection Challenge

When introspection is disabled, attackers and security testers must resort to *blind introspection*—techniques that infer schema structure through indirect means. The fundamental challenge lies in the combinatorial explosion of possible field names, argument types, and relationship structures that could exist within a GraphQL schema.

Traditional approaches to blind introspection rely on dictionary-based attacks, where tools systematically test field names from predefined wordlists. Clairvoyance, the current state-of-the-art tool for blind introspection, exemplifies this approach by combining brute-force field enumeration with GraphQL's field suggestion feature. When a query contains a misspelled field name, many GraphQL implementations respond with suggestions for similar valid fields, effectively leaking schema information.

However, dictionary-based approaches suffer from fundamental limitations that parallel those observed in naive fuzzing strategies. The effectiveness of the approach depends entirely on the quality and completeness of the wordlist, whilst the search process remains largely uninformed by the partial schema information discovered during the attack.

A.2.2 Search-Based Schema Discovery

KrakQL addresses these limitations by framing blind introspection as a *search-based software testing* problem, applying the evolutionary algorithms principles introduced in Chapter 2. Rather than relying on static wordlists, KrakQL employs a *novelty search* algorithm that dynamically generates field candidates based on the partially discovered schema.

The approach consists of three interconnected components that work together to systematically explore the schema space, as illustrated in Figure A.3:

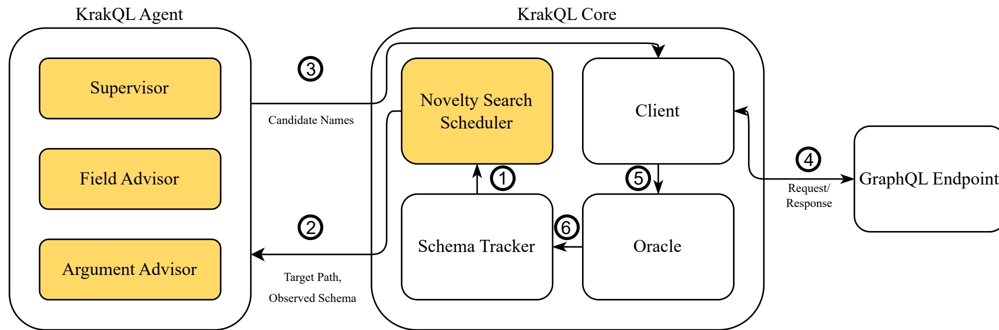

Figure A.3: KrakQL architecture showing the multi-agent system and novelty search scheduler (novel contributions highlighted).

1) LLM-Guided Candidate Generation. Large Language Models serve as intelligent mutation operators that generate contextually appropriate field names based on the partial schema discovered so far. Given a set of known types and fields, the LLM generates candidate field names that are semantically consistent with the existing schema structure. This approach leverages the LLM's training on large corpora of GraphQL schemas to propose candidates that are more likely to exist in real-world applications than random dictionary entries.

2) Novelty Search Algorithm. Traditional search algorithms optimise for a single objective, such as maximising the number of discovered fields. However, blind introspection benefits from exploring diverse regions of the schema space to avoid local optima. KrakQL implements a novelty search algorithm that maintains a population of candidate queries and selects for diversity in the discovered schema elements, preventing the search from becoming trapped in heavily explored regions.

3) Field Suggestion Feedback. GraphQL's field suggestion mechanism serves as a fitness function that provides feedback about the proximity of candidate field names to actual schema elements. When a candidate field name is sufficiently similar to a real field, the server's suggestion response reveals the actual field name along with contextual information about its type and arguments. This feedback mechanism enables the search algorithm to refine its candidate generation strategy and focus exploration on promising regions of the schema space.

A.2.3 Experimental Evaluation

The evaluation of KrakQL employed the BenGQL benchmarking framework described in the previous section, demonstrating the value of systematic evaluation infrastructure for advancing research. Seven representative GraphQL applications from the BenGQL collection served as test targets, spanning different schema complexities and implementation approaches.

The experimental results revealed significant limitations in existing dictionary-based approaches. Clairvoyance, despite being the state-of-the-art tool for blind introspection, achieved an average schema coverage of only 35% across the benchmark applications. This limited coverage stems from the fundamental mismatch between static dictionaries and the diverse naming conventions employed by different application domains.

KrakQL's search-based approach demonstrated substantial improvements across all evaluation metrics, as detailed in Tables A.2 and A.3:

Table A.2: Schema coverage comparison between Clairvoyance and KrakQL across seven GraphQL applications.

Target	**Clairvoyance**		**KrakQL**	
	Fields	**Args**	**Fields**	**Args**
countries	25/45 (56%)	3/7 (43%)	22/45 (49%)	3/7 (43%)
dvga	8/53 (15%)	0/33 (0%)	**42/53 (79%)**	**7/33 (21%)**
fruits-api	23/32 (72%)	**25/26 (96%)**	**24/32 (75%)**	20/26 (77%)
payload	56/8066 (1%)	**46/360 (13%)**	**431/8066 (5%)**	0/360 (0%)
react-ecommerce	5/163 (3%)	**13/13 (100%)**	**51/163 (31%)**	12/13 (92%)
react-finland	17/150 (11%)	**9/9 (100%)**	**84/150 (56%)**	7/9 (78%)
rick-and-morty	26/52 (50%)	12/12 (100%)	**33/52 (63%)**	12/12 (100%)
Total	160/8561 (2%)	**108/460 (24%)**	**687/8561 (8%)**	61/460 (13%)

Schema Coverage. KrakQL achieved 1.4× higher average coverage compared to Clairvoyance, successfully discovering schema elements that remained hidden from dictionary-based attacks. The improvement was particularly pronounced for applications with domain-specific terminology that diverged from common dictionary entries.

Efficiency. The intelligent candidate generation reduced the number of HTTP requests required by 148× compared to brute-force enumeration. This efficiency gain

Table A.3: Discovery efficiency comparison showing success rates per 64-name batch.

Target	Clairvoyance		KrakQL	
	Batches	Rate	Batches	Rate
countries	55,000	¡0.01%	166	0.39%
dvga	12,500	¡0.01%	417	0.23%
fruits-api	18,000	¡0.01%	161	0.88%
payload	87,500	¡0.01%	319	4.01%
react-ecommerce	30,000	¡0.01%	358	0.31%
react-finland	78,000	¡0.01%	404	0.40%
rick-and-morty	28,000	¡0.01%	266	0.27%
Total	309,000	¡0.01%	2,091	1.05%

not only reduces the time required for schema discovery but also minimises the attack's detectability by reducing network traffic patterns.

Success Rate. KrakQL achieved a 105× higher success rate in discovering complete schema sections, indicating that the search-based approach is more reliable than dictionary-based methods for comprehensive schema recovery.

These improvements were achieved at negligible computational cost, with total LLM expenses of $4.47 across all experiments—representing less than $0.005 per discovered schema element.

A.2.4 Connections to Coverage-Guided Fuzzing

The KrakQL approach demonstrates how search-based techniques can address automation challenges that extend beyond traditional coverage-guided fuzzing. The parallel between schema discovery and coverage exploration is particularly illuminating:

Partial Information Utilisation. Both approaches must operate with incomplete knowledge of the target system's structure. Coverage-guided fuzzing leverages partial coverage information to guide exploration, whilst KrakQL uses partial schema knowledge to direct candidate generation.

Feedback-Driven Search. Both approaches employ feedback mechanisms to evaluate the quality of exploration attempts. Coverage maps provide quantitative measures

of program exploration, whilst field suggestions provide qualitative feedback about schema proximity.

Evolutionary Optimisation. Both approaches employ evolutionary principles to iteratively improve exploration strategies. Coverage-guided fuzzing evolves input corpora to maximise coverage, whilst KrakQL evolves candidate populations to maximise schema discovery.

The success of KrakQL illustrates how the fundamental principles underlying coverage-guided fuzzing – systematic exploration, feedback-driven optimisation, and evolutionary search – can be generalised to other domains where complete system knowledge is unavailable. This generalisation reinforces the broader themes of this dissertation regarding automation in complex, partially observable systems.

A.3 Wendigo: Deep Reinforcement Learning for DoS Query Discovery

The expressiveness of GraphQL's query language, whilst providing significant benefits for client applications, also introduces unique security vulnerabilities. Complex queries can consume disproportionate server resources, enabling Denial-of-Service (DoS) attacks with minimal traffic volume. Unlike traditional DoS attacks that require high-volume traffic floods, GraphQL-based attacks can achieve similar effects through carefully crafted queries that exploit features such as circular relationships, field duplication, and nested structures.

This section explores how existing fuzzing frameworks perform when applied to GraphQL security testing, with particular focus on their limitations for discovering DoS vulnerabilities. The analysis revealed fundamental mismatches between traditional fuzzing approaches and the unique characteristics of GraphQL attack surfaces, motivating the development of domain-specific testing techniques.

A.3.1 Limitations of Existing Fuzzing Frameworks for GraphQL

Traditional fuzzing approaches, designed primarily for binary executables and file-based inputs, face significant challenges when applied to GraphQL APIs. These

limitations parallel the challenges observed in multiprocess fuzzing, where standard tools failed to address domain-specific complexities.

EvoMaster's Approach and Limitations. EvoMaster represents one of the most sophisticated automated testing tools available for API testing, employing evolutionary algorithms to generate testcases for both REST and GraphQL APIs. The tool implements both white-box and black-box testing strategies, using genetic algorithms to evolve populations of test inputs that maximise functional coverage of the target application.

For GraphQL testing, EvoMaster generates syntactically valid queries by parsing the target schema and creating random combinations of fields, arguments, and nesting structures. The tool employs several evolutionary techniques:

1) Schema-Based Generation. EvoMaster parses the GraphQL schema to understand available types, fields, and relationships, then generates queries that conform to the schema's structural constraints. This ensures that generated queries are syntactically valid and can be processed by the target server.

2) Random Search Exploration. The tool implements random search strategies to explore the query space, generating diverse combinations of fields and arguments to exercise different code paths within the resolver implementations.

3) Evolutionary Optimisation. Query populations evolve over time using genetic operators such as mutation and crossover, with fitness functions designed to reward queries that trigger novel server responses or error conditions.

However, EvoMaster's design reveals fundamental limitations when applied to DoS vulnerability discovery:

Functional Focus vs. Performance Testing. EvoMaster was primarily designed for functional testing—discovering logical errors, authentication bypasses, and data validation failures. Its fitness functions optimise for response diversity and error generation rather than resource consumption, making it poorly suited for identifying performance-related vulnerabilities.

Random Search Inefficiency. The tool's random search approach generates queries uniformly across the schema space, without understanding which GraphQL features

are most likely to cause resource exhaustion. Features such as circular references, deep nesting, and field duplication require specific patterns that random generation rarely produces.

Lack of Resource Feedback. EvoMaster evaluates queries based on response codes, response content, and error messages, but does not incorporate performance metrics such as response time or resource utilisation. This blind spot prevents the tool from recognising when queries cause server stress without triggering functional failures.

GraphQL-Specific Attack Vector Blindness. The tool treats GraphQL queries as generic tree structures without understanding the semantic implications of specific GraphQL features. Attack vectors such as alias overloading, query batching, and argument manipulation require domain-specific knowledge that EvoMaster lacks.

A.3.2 Empirical Evaluation of EvoMaster's DoS Discovery Capabilities

To quantify these limitations, we conducted systematic experiments comparing EvoMaster's performance against dedicated DoS discovery techniques. The evaluation employed multiple GraphQL applications from the BenGQL benchmark, focusing on the tool's ability to generate resource-intensive queries.

The experimental results confirmed the theoretical limitations:

Query Complexity. EvoMaster generated predominantly simple queries with shallow nesting and minimal field duplication. Complex attack patterns that combine multiple resource-intensive features were rarely produced by the random search process.

Resource Impact. Queries generated by EvoMaster showed minimal impact on server response times, typically executing within normal performance ranges. The tool failed to discover queries capable of causing significant resource exhaustion or server delays.

Attack Vector Coverage. Analysis of generated queries revealed poor coverage of GraphQL-specific attack vectors. Circular object traversal, alias overloading, and query batching patterns appeared infrequently and were not systematically explored.

These limitations highlight a broader challenge in applying general-purpose testing tools to domain-specific security problems. Just as coverage-guided fuzzers required fork-awareness to effectively test multiprocess software, GraphQL security testing requires understanding of GraphQL-specific attack semantics.

A.3.3 Wendigo's Deep Reinforcement Learning Solution

Wendigo addresses these limitations through a domain-specific approach that leverages Deep Reinforcement Learning (DRL) to discover DoS-capable queries. The tool was developed collaboratively, with particular focus on understanding how existing fuzzing frameworks perform in the GraphQL domain and identifying the key insights necessary for effective automation. Figure A.4 illustrates the Wendigo query discovery process.

The Wendigo approach introduces several innovations that directly address EvoMaster's limitations:

Performance-Oriented Feedback. Instead of optimising for functional diversity, Wendigo uses server response time as its primary reward signal. This shift in feedback mechanism enables the DRL agent to learn which query features contribute to resource exhaustion, directly addressing the performance testing gap in existing tools.

GraphQL-Aware State Space. The tool models the GraphQL query space using domain-specific abstractions that capture attack-relevant features. The state space explicitly represents circular relationships, field duplication counts, alias usage, and argument limits, enabling the agent to systematically explore resource-intensive query patterns.

Structured Action Space. Rather than generating queries through random mutations, Wendigo employs structured actions that correspond to specific GraphQL

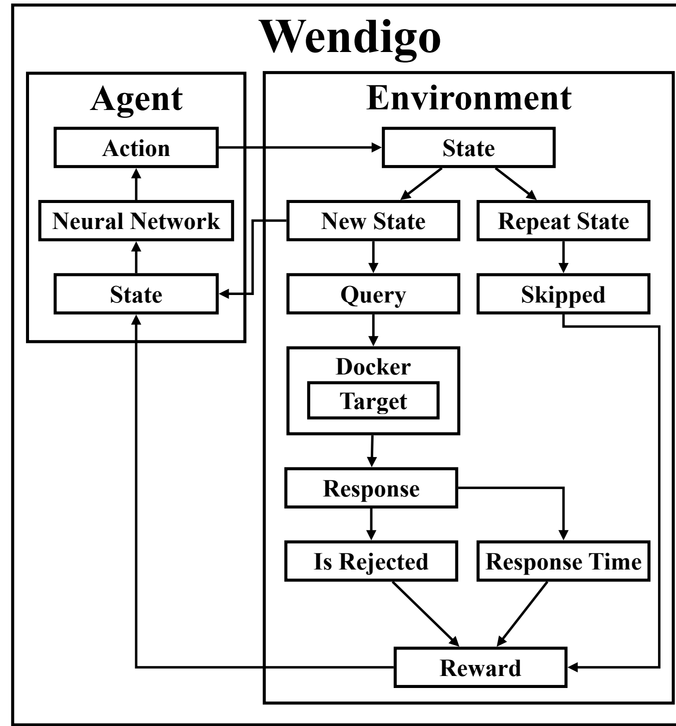

Figure A.4: WENDIGO query discovery workflow showing the DRL agent's interaction with the GraphQL environment.

attack vectors. Actions include adding circular references, duplicating fields, creating aliases, and manipulating argument values, ensuring that the exploration process focuses on security-relevant modifications.

Incremental Query Building. The DRL agent builds queries incrementally, starting from simple structures and progressively adding complexity based on performance feedback. This approach enables discovery of sophisticated attack patterns that would be unlikely to emerge from random generation.

A.3.4 Experimental Results and Comparative Analysis

The evaluation of WENDIGO demonstrated substantial improvements over EVOMASTER and other baseline approaches. Figure A.5 shows the response time comparison between WENDIGO and baseline approaches on the DVGA (Damn Vulnerable GraphQL Application) test target.

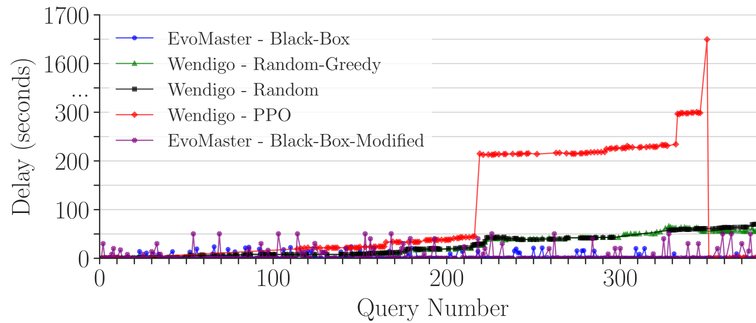

Figure A.5: Wendigo response time evaluation showing superior DoS query discovery compared to EvoMaster and random baselines on DVGA target. Results demonstrate Wendigo's ability to discover resource-intensive queries that cause significant server delays.

DoS Query Discovery. Wendigo successfully discovered queries capable of causing severe server delays with minimal request volumes. In some cases, queries were found that could achieve DoS effects using only 2 requests per hour, demonstrating the power of GraphQL-specific attack patterns. The results show Wendigo achieving response times over 10× higher than baseline approaches.

Attack Vector Utilisation. The DRL approach systematically explored all major GraphQL attack vectors, generating queries that combined multiple resource-intensive features to maximise server impact. This comprehensive coverage contrasted sharply with EvoMaster's sparse exploration of attack-relevant patterns.

Learning Efficiency. The performance-oriented feedback mechanism enabled rapid convergence towards effective attack patterns, with the DRL agent quickly identifying which query modifications led to increased response times and focusing exploration accordingly.

The comparative analysis revealed key insights about the requirements for effective GraphQL security testing:

Domain-Specific Understanding. Success in GraphQL security testing requires deep understanding of GraphQL semantics and attack vectors. General-purpose testing tools that treat GraphQL as a generic input format miss crucial attack opportunities.

Performance Feedback. Security-focused testing must incorporate performance

metrics alongside functional correctness measures. Tools that optimise purely for functional diversity fail to identify resource exhaustion vulnerabilities.

Structured Exploration. Random exploration strategies are insufficient for discovering complex attack patterns that require specific combinations of features. Structured search approaches that understand attack semantics achieve significantly better results.

These insights parallel the lessons learned from multiprocess fuzzing, where domain-specific understanding and appropriate feedback mechanisms proved essential for effective automation. The success of WENDIGO demonstrates how these principles can be applied beyond traditional coverage-guided fuzzing to address security testing challenges in complex, distributed systems.

A.4 Concluding Remarks

The GraphQL testing work presented in this appendix illustrates how the fundamental automation challenges addressed in the main chapters of this dissertation extend across different domains of software testing. Whilst the specific technical details differ between multiprocess fuzzing and GraphQL testing, the underlying patterns remain consistent: complex, distributed systems require domain-specific understanding, appropriate feedback mechanisms, and systematic evaluation methodologies to achieve effective automation.

The three contributions presented – BENGQL, KRAKQL, and WENDIGO – collectively demonstrate the breadth of techniques required to address automation challenges in distributed systems:

Infrastructure and Methodology. BENGQL demonstrates the critical importance of systematic evaluation infrastructure for advancing automated testing research. The framework's design reflects the same principles employed in the fork-awareness evaluation: controlled experimental conditions, standardised metrics, and reproducible execution environments. This parallel reinforces the broader theme that progress in automated testing requires not only novel techniques but also rigorous evaluation methodologies.

Search-Based Approaches. KrakQL's application of evolutionary algorithms to schema discovery illustrates how search-based techniques can address exploration challenges when traditional coverage metrics are unavailable. The success of novelty search in navigating the GraphQL schema space parallels the evolutionary principles underlying coverage-guided fuzzing, demonstrating the generalisability of these approaches across different problem domains.

Domain-Specific Adaptation. Wendigo's superior performance compared to general-purpose tools highlights the necessity of domain-specific understanding in automated testing. Just as fork-awareness required deep understanding of process lifecycle semantics, effective GraphQL security testing requires understanding of GraphQL attack vectors and performance characteristics. This pattern suggests that the most significant advances in automated testing come from combining general algorithmic principles with domain-specific insights.

The shift from grey-box coverage-guided fuzzing to black-box alternative feedback mechanisms represents a natural evolution of automated testing techniques. As software systems become increasingly complex and distributed, traditional visibility mechanisms such as code coverage become insufficient or unavailable. The success of response time feedback in Wendigo and field suggestion feedback in KrakQL demonstrates that creative application of available signals can achieve effective automation even under severe observability constraints.

Looking forward, the convergence of techniques observed throughout this dissertation suggests several promising directions for future research:

Hybrid Feedback Mechanisms. The combination of coverage-guided feedback with alternative signals such as performance metrics, error patterns, and system responses could enhance the effectiveness of automated testing across multiple domains. Techniques that dynamically adapt their feedback strategies based on available visibility mechanisms could provide robust automation for diverse system architectures.

Cross-Domain Algorithm Transfer. The successful application of evolutionary algorithms, search-based techniques, and reinforcement learning across both multi-process fuzzing and GraphQL testing suggests that algorithmic innovations in one

domain can often be adapted to others. Systematic exploration of these cross-domain applications could accelerate progress in automated testing research.

Unified Evaluation Frameworks. The development of benchmarking frameworks such as BenGQL for GraphQL testing and the fork-awareness evaluation methodology for multiprocess fuzzing demonstrates the value of systematic evaluation infrastructure. Future work could explore unified frameworks that enable comparison of automated testing techniques across different application domains.

The work presented in this appendix reinforces the central thesis of this dissertation: effective automation of software testing for complex, distributed systems requires the integration of domain-specific understanding with principled algorithmic approaches and systematic evaluation methodologies. The techniques developed for GraphQL testing demonstrate that these principles extend well beyond traditional coverage-guided fuzzing, providing a foundation for addressing automation challenges across the broader landscape of modern software systems.

Broader Implications. The evolution from coverage-guided fuzzing of single-process applications to comprehensive testing of distributed systems – whether multiprocess software or GraphQL APIs – reflects the broader transformation of software architecture towards increasingly complex, interconnected systems. As software continues to evolve towards microservices, serverless architectures, and distributed computing paradigms, the automation techniques developed in this dissertation provide essential foundations for maintaining software quality and security in these complex environments.

The success of domain-specific approaches in both multiprocess and GraphQL testing suggests that future advances in automated testing will increasingly require deep understanding of specific system architectures, communication patterns, and operational semantics. However, the underlying algorithmic principles – evolutionary search, feedback-driven exploration, and systematic evaluation – remain constant across domains, providing a stable foundation for continued innovation in automated testing research.

Appendix B

Stateful Fuzzing

This appendix presents contributions to advancing stateful fuzzing through two research projects: LibAFLstar and the OPC UA fuzzing benchmark. These works address instantiations of the four core challenges (Chapter 2) in the context of stateful protocol testing, where the
emphExecution Synchronisation involves state transitions, the
emphOracle Problem requires understanding protocol semantics,
emphObservability encompasses state coverage tracking, and
emphContext-Aware Input Generation demands protocol-aware message construction.

B.1 LibAFLstar: Fast and State-Aware Protocol Fuzzing

LibAFLstar [45] represents a breakthrough in stateful protocol fuzzing, achieving over 30× performance improvements compared to existing approaches whilst maintaining comprehensive state exploration. The fuzzer addresses instantiations of the core challenges in stateful contexts: state model exploration (
emphExecution Synchronisation across protocol states), semantic correctness detection (
emphOracle Problem for protocol violations), comprehensive state coverage (
emphObservability of protocol paths), and intelligent message generation (
emphContext-Aware Input Generation for protocol-specific inputs).

B.1.1 Infrastructure Contributions

The primary contribution to LibAFLstar involved establishing the complete evaluation infrastructure, ensuring reproducible and fair comparisons with competing approaches. This infrastructure work proved critical for demonstrating the practical advantages of the new fuzzing techniques.

Docker Environment Setup

A comprehensive Docker-based evaluation environment was designed and implemented that standardises the execution of three different stateful fuzzers: AFLNet, ChatAFL, and LibAFLstar. This containerised approach addresses several critical challenges:

- **Dependency isolation**: Each fuzzer requires specific library versions and compiler configurations. The Docker environment encapsulates these dependencies, preventing conflicts and ensuring consistent builds across different host systems.
- **Reproducible builds**: The automated build scripts compile each target with identical optimisation flags and instrumentation settings, eliminating performance variations due to build configuration differences.
- **Fair resource allocation**: Container resource limits ensure each fuzzer receives identical CPU and memory allocations, preventing one tool from monopolising system resources during comparative evaluations.
- **Automated data collection**: The infrastructure automatically collects coverage metrics, crash reports, and performance statistics at regular intervals, storing results in a structured format for subsequent analysis.

The Docker environment supports parallel execution of multiple fuzzing campaigns, enabling large-scale experiments across different configurations and random seeds. This capability proved essential for the statistical significance of our evaluation results.

Target Preparation and Patching

A substantial portion of the work involved manually patching protocol implementations to enable persistent-mode fuzzing—a critical optimisation for achieving high throughput. Unlike traditional fork-based fuzzing, persistent mode requires targets to process multiple inputs within a single execution, necessitating careful modifications to server initialisation and cleanup routines.

Manual patches were created for the following targets:

- **ProFTPD**: Modified the main server loop to reset connection state between inputs whilst preserving global configuration. This required careful tracking of file descriptors and memory allocations to prevent resource leaks during extended fuzzing campaigns.
- **Pure-FTPd**: Implemented state cleanup routines that reset authentication status and working directory whilst maintaining the process address space. The patches ensure deterministic behaviour across thousands of iterations.
- **Bftpd**: Restructured the connection handling logic to support in-memory input injection, bypassing network socket operations entirely. This transformation improved fuzzing throughput by over 100×.
- **Live555**: Adapted the RTSP server's event loop to process fuzzer-provided inputs synchronously, eliminating timing dependencies that cause non-determinism in coverage feedback.
- **Dcmtk**: Modified the DICOM protocol implementation to reset session state between inputs whilst preserving loaded image data, reducing initialisation overhead.
- **Dnsmasq**: Patched the DNS server to process queries from memory buffers rather than network sockets, enabling deterministic replay of packet sequences.

The LightFTP setup was adapted from prior work [64], whilst the Lighttpd configuration originated from another thesis [65]. These existing configurations required only minor adjustments for integration with our evaluation framework.

Each patch required deep understanding of the target's architecture, memory management patterns, and protocol state machine. The modifications preserve semantic correctness whilst enabling efficient fuzzing—a delicate balance that took considerable engineering effort to achieve.

Evaluation Harness Development

Beyond individual target patches, a unified evaluation harness was developed that coordinates fuzzing campaigns across different tools and configurations. The harness implements:

- **Automated corpus minimisation**: Reduces seed inputs to minimal sets that maintain coverage, accelerating fuzzer startup.
- **State model extraction**: Interfaces with LearnLib to automatically infer protocol state machines for targets lacking specifications.
- **Performance monitoring**: Tracks execution speed, memory usage, and coverage evolution at fine granularity.
- **Crash triage**: Automatically deduplicates and categorises discovered vulnerabilities based on stack traces and program state.

B.1.2 Experimental Results

The developed infrastructure enabled comprehensive evaluation across six protocol implementations over 24-hour fuzzing campaigns. LibAFLstar achieved 1.4× higher coverage than AFLNet and ChatAFL whilst processing inputs 30× faster. These improvements stem from the combination of persistent-mode execution (enabled by my patches) and the novel state-aware scheduling algorithms.

The evaluation revealed that different targets benefit from different optimisations: some gain most from persistent mode, others from state-aware scheduling. This insight, made possible by the standardised evaluation infrastructure, guides practitioners in selecting appropriate techniques for their specific testing scenarios.

B.2 Fuzzing OPC UA: A Research Experience

The OPC Unified Architecture (OPC UA) represents one of the most complex industrial communication protocols, supporting everything from simple sensor readings to sophisticated publish-subscribe patterns with encrypted channels. Testing OPC UA implementations poses unique challenges due to the protocol's rich semantics and extensive state space.

B.2.1 Project Overview

The development of comprehensive OPC UA fuzzing support was led across three major stateful fuzzers, making this critical industrial protocol accessible to automated testing for the first time. This work involved not just technical implementation but also careful design of abstraction functions that capture protocol semantics whilst enabling efficient fuzzing.

The project emerged from the recognition that existing protocol fuzzing benchmarks focus primarily on simple text-based protocols (FTP, HTTP, RTSP) that fail to exercise the complexity found in industrial systems. OPC UA, with its binary encoding, session management, secure channels, and subscription mechanisms, provides a more realistic test of stateful fuzzing capabilities.

B.2.2 Technical Contributions

Protocol Integration

OPC UA protocol support was implemented for AFLNet, ChatAFL, and LibAFLstar, requiring different integration strategies for each tool:

AFLNet Integration: Developed abstraction functions that parse OPC UA's binary message format, extracting message types and status codes for state inference. The implementation handles the protocol's variable-length encoding and nested structures, correctly identifying message boundaries even in malformed inputs.

ChatAFL Enhancement: Extended the LLM-based grammar inference with OPC UA-specific prompts that guide the model toward generating semantically valid message sequences. This required encoding domain knowledge about session establishment, secure channel negotiation, and subscription management into the prompt engineering.

LibAFLstar Adaptation: Designed prefixes that navigate the OPC UA state machine efficiently, reaching deep protocol states that handle complex operations like monitored item creation and historical data access. The prefixes bypass expensive cryptographic handshakes when security testing is not the focus.

Benchmark Development

The FreeOpcUa server was integrated into the ProFuzzBench framework, automating the entire fuzzing pipeline:

- **Build automation**: Created scripts that compile FreeOpcUa with appropriate instrumentation for each fuzzer, handling the project's complex dependency tree including the UA SDK and cryptographic libraries.
- **Seed corpus generation**: Developed a comprehensive seed collection methodology, capturing diverse protocol interactions including session management, browsing, reading, writing, and subscription operations.
- **Harness implementation**: Built specialised harnesses that bypass network layers and inject inputs directly into the protocol stack, achieving 50× throughput improvement over network-based fuzzing.
- **State model inference**: Applied active learning techniques to extract the implemented state machine from FreeOpcUa, revealing deviations from the specification that impact fuzzing effectiveness.

Comparative Analysis

The developed infrastructure enabled the first systematic comparison of stateful fuzzing approaches on industrial protocols. Key findings include:

- AFLNet provides the most straightforward setup but achieves limited coverage due to inefficient state exploration
- ChatAFL discovers interesting message sequences through LLM guidance but suffers from high computational overhead
- LibAFLstar achieves superior throughput and coverage but requires substantial engineering effort for integration

These insights, documented in our research experience paper, guide practitioners in selecting appropriate tools based on their testing objectives and available resources.

B.2.3 Impact and Future Directions

The OPC UA fuzzing infrastructure has already discovered multiple vulnerabilities in production implementations, demonstrating the practical value of stateful fuzzing for industrial protocols. The work establishes a foundation for testing other complex protocols in critical infrastructure, including IEC 61850, Modbus/TCP, and MQTT.

Future extensions could incorporate semantic knowledge about industrial control logic, enabling detection of logical vulnerabilities beyond memory corruption. Integration with formal verification tools could provide stronger guarantees about protocol compliance and security properties.

B.3 Lessons Learned

Through these projects, several key insights emerged about practical stateful fuzzing:

Infrastructure is crucial: The effort invested in building robust evaluation infrastructure pays dividends through reproducible results and fair comparisons. Without standardised environments, performance claims remain suspect.

Manual optimisation matters: Whilst automated techniques continue advancing, manual patches and target-specific optimisations still provide order-of-magnitude improvements. Deep understanding of target applications remains invaluable.

Protocol complexity varies widely: Simple text protocols like FTP provide useful baselines but fail to exercise the full capabilities of modern fuzzers. Real-world protocols like OPC UA reveal limitations and opportunities invisible in toy examples.

Tooling gaps persist: Despite progress, significant engineering effort is still required to apply stateful fuzzing to new protocols. Better abstractions and automation could democratise access to these powerful techniques.

These contributions advance the state of the art in stateful fuzzing whilst providing practical tools and insights for the security community. The combination of infrastructure development, manual optimisation, and systematic evaluation establishes a foundation for continued progress in automated protocol testing.

Bibliography

[1] I. Bojanova and C. E. C. Galhardo. “ Bug, Fault, Error, or Weakness: Demystifying Software Security Vulnerabilities ”. In: *IT Professional* (Jan. 2023), pp. 7–12. ISSN: 1941-045X. DOI: 10.1109/MITP.2023.3238631. URL: https://doi.ieeecomputersociety.org/10.1109/MITP.2023.3238631.

[2] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang. “FUDGE: fuzz driver generation at scale”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia, 2019, pp. 975–985. ISBN: 9781450355728. DOI: 10.1145/3338906.3340456. URL: https://doi.org/10.1145/3338906.3340456.

[3] C. Zhang, Y. Zheng, M. Bai, Y. Li, W. Ma, X. Xie, Y. Li, L. Sun, and Y. Liu. “How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation”. In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2024. Vienna, Austria, 2024, pp. 1223–1235. ISBN: 9798400706127. DOI: 10.1145/3650212.3680355. URL: https://doi.org/10.1145/3650212.3680355.

[4] D. Liu, O. Chang, J. metzman, M. Sablotny, and M. Maruseac. *OSS-Fuzz-Gen: Automated Fuzz Target Generation*. Version https://github.com/google/oss-fuzz-gen/tree/v1.0. May 2024. URL: https://github.com/google/oss-fuzz-gen.

[5] G. Castiglione, M. Maugeri, and G. Bella. “Poster: Machine Learning for Vulnerability Detection as Target Oracle in Automated Fuzz Driver Generation”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by M. Egele, V. Moonsamy, D. Gruss, and M. Carminati. 2025, pp. 140–145. ISBN: 978-3-031-97620-9.

[6] ISO Central Secretary. *Software and systems engineering – Software testing (Part 1: General Concepts)*. en. Standard. International Organization for Standardization, 2022. URL: https://www.iso.org/standard/81291.html.

[7] N. I. of Standards and Technology. *Security and Privacy Controls for Information Systems and Organizations*. Tech. rep. U.S. Department of Commerce, 2020. DOI: 10.6028/NIST.SP.800-53r5.

[8] X. Zhu and M. Böhme. "Regression Greybox Fuzzing". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. Virtual Event, Republic of Korea, 2021, pp. 2169–2182. ISBN: 9781450384544. DOI: 10.1145/3460120.3484596. URL: https://doi.org/10.1145/3460120.3484596.

[9] J. Kim and S. Hong. "BugOss: A benchmark of real-world regression bugs for empirical investigation of regression fuzzing techniques". In: *Journal of Systems and Software* (2024), p. 112119. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2024.112119. URL: https://www.sciencedirect.com/science/article/pii/S016412122400164X.

[10] 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". In: *IEEE Transactions on Software Engineering* (2021), pp. 2312–2331. DOI: 10.1109/TSE.2019.2946563.

[11] J. Li, B. Zhao, and C. Zhang. "Fuzzing: a survey". In: *Cybersecurity* (June 2018), p. 13. ISSN: 2523-3246. DOI: 10.1186/s42400-018-0002-y. URL: https://doi.org/10.1186/s42400-018-0002-y.

[12] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels". In: *26th USENIX Security Symposium (USENIX Security 17)*. Aug. 2017, pp. 167–182. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo.

[13] A. Arcuri. "EvoMaster: Evolutionary Multi-context Automated System Test Generation". In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 2018, pp. 394–397. DOI: 10.1109/ICST.2018.00046.

[14] A. Arcuri, M. Zhang, S. Seran, J. P. Galeotti, A. Golmohammadi, O. Duman, A. Aldasoro, and H. Ghianni. “Tool report: EvoMaster—black and white box search-based fuzzing for REST, GraphQL and RPC APIs”. In: *Automated Software Engg.* (Nov. 2024). ISSN: 0928-8910. DOI: 10.1007/s10515-024-00478-1. URL: https://doi.org/10.1007/s10515-024-00478-1.

[15] V. Atlidakis, P. Godefroid, and M. Polishchuk. “RESTler: Stateful REST API Fuzzing”. In: *Proceedings of the 41st International Conference on Software Engineering*. ICSE’19. Montreal, Quebec, Canada, 2019, pp. 748–758. DOI: 10.1109/ICSE.2019.00083. URL: https://doi.org/10.1109/ICSE.2019.00083.

[16] W. Yang, M. R. Prasad, and T. Xie. “A grey-box approach for automated GUI-model generation of mobile applications”. In: FASE’13. Rome, Italy, 2013, pp. 250–265. ISBN: 9783642370564. DOI: 10.1007/978-3-642-37057-1_19. URL: https://doi.org/10.1007/978-3-642-37057-1_19.

[17] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce. “Echidna: effective, usable, and fast fuzzing for smart contracts”. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2020. Virtual Event, USA, 2020, 557–560. ISBN: 9781450380089. DOI: 10.1145/3395363.3404366. URL: https://doi.org/10.1145/3395363.3404366.

[18] J. Yang, X. Zhao, H. Zhang, L. He, S. Wang, and N. Gou. “CSAFuzzer: Fuzzing smart contracts combining with static analysis”. In: *Empirical Softw. Engg.* (Feb. 2025). ISSN: 1382-3256. DOI: 10.1007/s10664-025-10623-3. URL: https://doi.org/10.1007/s10664-025-10623-3.

[19] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. “Automated whitebox fuzz testing.” In: *Ndss*. 2008, pp. 151–166.

[20] C. Cadar, D. Dunbar, and D. Engler. “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs”. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI’08. San Diego, California, 2008, pp. 209–224.

[21] P. Godefroid, M. Y. Levin, and D. Molnar. “SAGE: whitebox fuzzing for security testing”. In: *Communications of the ACM* (2012), pp. 40–44.

[22] C. Cadar and K. Sen. “Symbolic execution for software testing: three decades later”. In: *Commun. ACM* (Feb. 2013), pp. 82–90. ISSN: 0001-0782. DOI:

10.1145/2408776.2408795. URL: https://doi.org/10.1145/2408776.2408795.

[23] V.-T. Pham, M. Böhme, and A. Roychoudhury. "Model-based whitebox fuzzing for program binaries". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE '16. Singapore, Singapore, 2016, pp. 543–553. ISBN: 9781450338455. DOI: 10.1145/2970276.2970316. URL: https://doi.org/10.1145/2970276.2970316.

[24] M. Zalewski. *American Fuzzy Lop (AFL): security-oriented fuzzer*. https://github.com/google/AFL. Accessed: 2025-09-06.

[25] LLVM Project. *libFuzzer: in-process, coverage-guided fuzzing engine*. https://llvm.org/docs/LibFuzzer.html. Accessed: 2025-09-06.

[26] D. Steinhöfel and A. Zeller. "Input invariants". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singapore, 2022, pp. 583–594. ISBN: 9781450394130. DOI: 10.1145/3540250.3549139. URL: https://doi.org/10.1145/3540250.3549139.

[27] J. A. Zamudio Amaya, M. Smytzek, and A. Zeller. "FANDANGO: Evolving Language-Based Testing". In: *Proc. ACM Softw. Eng.* (June 2025). DOI: 10.1145/3728915. URL: https://doi.org/10.1145/3728915.

[28] C. Daniele, S. B. Andarzian, and E. Poll. "Fuzzers for Stateful Systems: Survey and Research Directions". In: *ACM Comput. Surv.* (Apr. 2024). ISSN: 0360-0300. DOI: 10.1145/3648468. URL: https://doi.org/10.1145/3648468.

[29] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti. "LibAFL: A Framework to Build Modular and Reusable Fuzzers". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. Nov. 2022, pp. 1051–1065. ISBN: 978-1-4503-9450-5. DOI: 10.1145/3548606.3560602. URL: https://dl.acm.org/doi/10.1145/3548606.3560602 (visited on 06/26/2025).

[30] R. Malmain, A. Fioraldi, and A. Francillon. "LibAFL QEMU: A Library for Fuzzing-oriented Emulation". en. In: *Proceedings 2024 Workshop on Binary Analysis Research*. 2024. ISBN: 979-8-9894372-0-7. DOI: 10.14722/bar.2024.23007. URL: https://www.ndss-symposium.org/wp-content/uploads/bar2024-7-paper.pdf (visited on 06/26/2025).

[31] P. D., M. Clauß, and K. Patterson. *libdesock: A de-socketing library for fuzzing.* https://github.com/fkie-cad/libdesock. GitHub repository, latest release Sept. 12, 2024. Sept. 2024.

[32] S. B. Andarzian, C. Daniele, and E. Poll. "Green-Fuzz: Efficient Fuzzing for Network Protocol Implementations". en. In: *Foundations and Practice of Security.* Ed. by M. Mosbah, F. Sèdes, N. Tawbi, T. Ahmed, N. Boulahia-Cuppens, and J. Garcia-Alfaro. 2024, pp. 253–268. ISBN: 978-3-031-57537-2. DOI: 10.1007/978-3-031-57537-2_16.

[33] J. Robben and M. Vanhoef. "Netfuzzlib: Adding First-Class Fuzzing Support to Network Protocol Implementations". en. In: *Computer Security - ESORICS 2024.* Ed. by J. Garcia-Alfaro, R. Kozik, M. Choraś, and S. Katsikas. 2024, pp. 65–84. ISBN: 978-3-031-70890-9. DOI: 10.1007/978-3-031-70890-9_4.

[34] Van-Thuan Pham. *aflnwe: Network-Enabled Version of AFL Fuzzer.* https://github.com/thuanpv/aflnwe. Accessed: 2025-09-06.

[35] X. Zhang, C. Zhang, X. Li, Z. Du, B. Mao, Y. Li, Y. Zheng, Y. Li, L. Pan, Y. Liu, and R. Deng. "A Survey of Protocol Fuzzing". In: *ACM Comput. Surv.* (Oct. 2024). ISSN: 0360-0300. DOI: 10.1145/3696788. URL: https://doi.org/10.1145/3696788.

[36] V.-T. Pham, M. Böhme, and A. Roychoudhury. "AFLNet: a greybox fuzzer for network protocols". In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST).* IEEE. 2020, pp. 460–465.

[37] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury. "Large Language Model guided Protocol Fuzzing". en. In: *Proceedings 2024 Network and Distributed System Security Symposium.* 2024. ISBN: 978-1-891562-93-8. DOI: 10.14722/ndss.2024.24556. URL: https://www.ndss-symposium.org/wp-content/uploads/2024-556-paper.pdf (visited on 06/26/2025).

[38] R. Natella. "Stateafl: Greybox fuzzing for stateful network servers". In: *Empirical Software Engineering* (2022), p. 191.

[39] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz. "Ijon: Exploring Deep State Spaces via Fuzzing". In: *2020 IEEE Symposium on Security and Privacy (SP).* 2020, pp. 1597–1612. DOI: 10.1109/SP40000.2020.00117.

[40] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury. "Stateful Greybox Fuzzing". In: *31st USENIX Security Symposium (USENIX Security 22).* Aug.

2022, pp. 3255–3272. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/ba.

[41] A. Andronidis and C. Cadar. “SnapFuzz: high-throughput fuzzing of network applications”. In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. 2022, pp. 340–351. ISBN: 978-1-4503-9379-9. DOI: 10.1145/3533767.3534376. URL: https://dl.acm.org/doi/10.1145/3533767.3534376 (visited on 06/26/2025).

[42] P.-A. Arras, A. Andronidis, L. Pina, K. Mituzas, Q. Shu, D. Grumberg, and C. Cadar. “SaBRe: load-time selective binary rewriting”. en. In: *International Journal on Software Tools for Technology Transfer* (Apr. 2022), pp. 205–223. ISSN: 1433-2787. DOI: 10.1007/s10009-021-00644-w. URL: https://doi.org/10.1007/s10009-021-00644-w (visited on 06/26/2025).

[43] Z. Luo, J. Yu, F. Zuo, J. Liu, Y. Jiang, T. Chen, A. Roychoudhury, and J. Sun. “Bleem: Packet Sequence Oriented Fuzzing for Protocol Implementations”. In: *32nd USENIX Security Symposium (USENIX Security 23)*. Aug. 2023, pp. 4481–4498. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/luo-zhengxiong.

[44] N. Bars, M. Schloegel, N. Schiller, L. Bernhard, and T. Holz. “No Peer, no Cry: Network Application Fuzzing via Fault Injection”. In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. CCS ’24. 2024, pp. 750–764. ISBN: 979-8-4007-0636-3. DOI: 10.1145/3658644.3690274. URL: https://dl.acm.org/doi/10.1145/3658644.3690274 (visited on 06/26/2025).

[45] C. Daniele, T. Bethe, M. Maugeri, A. Continella, and E. Poll. “LibAFLstar: Fast and State-Aware Protocol Fuzzing”. en. In: *Computer Security - ESORICS 2025*. 2025.

[46] R. Natella and V. T. Pham. “ProFuzzBench: A benchmark for stateful protocol fuzzing”. In: *ISSTA 2021 - Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. July 2021, pp. 662–665. ISBN: 9781450384599. DOI: 10.1145/3460319.3469077.

[47] F. Bellard. “QEMU, a fast and portable dynamic translator”. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC ’05. Anaheim, CA, 2005, p. 41.

[48] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz. "Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types". en. In: 2021, pp. 2597–2614. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo (visited on 06/26/2025).

[49] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz. "Nyx-net: network fuzzing with incremental snapshots". en. In: *Proceedings of the Seventeenth European Conference on Computer Systems*. Mar. 2022, pp. 166–180. ISBN: 978-1-4503-9162-7. DOI: 10.1145/3492321.3519591. URL: https://dl.acm.org/doi/10.1145/3492321.3519591 (visited on 06/26/2025).

[50] D. Maier, O. Bittner, M. Munier, and J. Beier. "FitM: binary-only coverage-guided fuzzing for stateful network protocols". In: *Workshop on Binary Analysis Research (BAR)*. 2022.

[51] Google. *syzkaller: Unsupervised Coverage-Guided Kernel Fuzzer*. https://github.com/google/syzkaller. Accessed: 2025-09-06.

[52] S. Pailoor, A. Aday, and S. Jana. "MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation." In: *USENIX Security Symposium*. 2018, pp. 729–743.

[53] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu. "MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs". In: *29th USENIX Security Symposium (USENIX Security 20)*. Aug. 2020, pp. 2325–2342. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu.

[54] S. Padhiyar and K. Sivaramakrishnan. "ConFuzz: Coverage-guided property fuzzing for event-driven programs". In: *Practical Aspects of Declarative Languages: 23rd International Symposium, PADL 2021, Copenhagen, Denmark, January 18-19, 2021, Proceedings 23*. Springer. 2021, pp. 127–144.

[55] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu. "Context-sensitive and directional concurrency fuzzing for data-race detection". In: *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS)*. 2022.

[56] D. Wolff, Z. Shi, G. J. Duck, U. Mathur, and A. Roychoudhury. “Greybox Fuzzing for Concurrency Testing”. In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ASPLOS ’24. La Jolla, CA, USA, 2024, pp. 482–498. ISBN: 9798400703850. DOI: 10.1145/3620665.3640389. URL: https://doi.org/10.1145/3620665.3640389.

[57] M. Eisele, M. Maugeri, R. Shriwas, C. Huth, and G. Bella. “Embedded fuzzing: a review of challenges, tools, and solutions”. en. In: *Cybersecurity* (Sept. 2022), p. 18. ISSN: 2523-3246. DOI: 10.1186/s42400-022-00123-y. URL: https://doi.org/10.1186/s42400-022-00123-y (visited on 06/26/2025).

[58] Y.-T. Cheng and S.-M. Cheng. “Firmulti Fuzzer: Discovering Multi-process Vulnerabilities in IoT Devices with Full System Emulation and VMI”. In: *Proceedings of the 5th Workshop on CPS&IoT Security and Privacy*. CPSIoT-Sec ’23. Nov. 2023, pp. 1–9. ISBN: 979-8-4007-0254-9. DOI: 10.1145/3605758.3623493. URL: https://dl.acm.org/doi/10.1145/3605758.3623493 (visited on 06/26/2025).

[59] H. Xiao, Z. Wei, J. Dai, B. Li, Y. Zhang, and M. Yang. “HouseFuzz: Service-Aware Grey-Box Fuzzing for Vulnerability Detection in Linux-Based Firmware”. In: *2025 IEEE Symposium on Security and Privacy (SP)*. ISSN: 2375-1207. May 2025, pp. 3801–3819. DOI: 10.1109/SP61157.2025.00213. URL: https://ieeexplore.ieee.org/abstract/document/11023421 (visited on 06/26/2025).

[60] M. Maugeri, C. Daniele, G. Bella, and E. Poll. “Evaluating the Fork-Awareness of Coverage-Guided Fuzzers”. en. In: *Proceedings of the 9th International Conference on Information Systems Security and Privacy*. 2023, pp. 424–429. ISBN: 978-989-758-624-8. DOI: 10.5220/0011648600003405. URL: https://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0011648600003405 (visited on 06/26/2025).

[61] J. Metzman, L. Szekeres, L. Maurice Romain Simon, R. Trevelin Sprabery, and A. Arya. “FuzzBench: An Open Fuzzer Benchmarking Platform and Service”. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. 2021, pp. 1393–1403. ISBN: 9781450385626. DOI:

10.1145/3468264.3473932. URL: https://doi.org/10.1145/3468264.3473932.

[62] M. Maugeri, C. Daniele, and G. Bella. “Forkfuzz: Leveraging the Fork-Awareness in Coverage-Guided Fuzzing”. en. In: *Computer Security. ESORICS 2023 International Workshops*. Ed. by S. Katsikas, H. Abie, S. Ranise, L. Verderame, E. Cambiaso, R. Ugarelli, I. Praça, W. Li, W. Meng, S. Furnell, B. Katt, S. Pirbhulal, A. Shukla, M. Ianni, M. Dalla Preda, K.-K. R. Choo, M. Pupo Correia, A. Abhishta, G. Sileno, M. Alishahi, H. Kalutarage, and N. Yanai. 2024, pp. 291–308. ISBN: 978-3-031-54129-2. DOI: 10.1007/978-3-031-54129-2_17.

[63] GraphQL Foundation. *GraphQL Specification*. Accessed: 2025-08-30. 2015. URL: https://spec.graphql.org/.

[64] T. Bethe. “Fallaway: High Throughput Stateful Fuzzing by making AFL* State-Aware”. MA thesis. University of Twente, 2024.

[65] L. Cantarella. “Benchmarking Stateful Fuzzers over Lighttpd”. MA thesis. University of Catania, 2024.